



Apex Workbook

Apex Workbook, Summer '16



CONTENTS

| | |
|--|----|
| Apex Workbook | 1 |
| Part 1: Orientation | 3 |
| Creating Warehouse Custom Objects | 3 |
| Using the Developer Console | 3 |
| Activating the Developer Console | 4 |
| Using the Developer Console to Execute Apex Code | 4 |
| Summary | 6 |
| Creating Sample Data | 6 |
| Creating and Instantiating Classes | 6 |
| Creating an Apex Class Using the Developer Console | 6 |
| Calling a Class Method | 8 |
| Creating an Apex Class Using the Salesforce User Interface | 9 |
| Summary | 9 |
| Part 2: Apex Language Fundamentals | 11 |
| Primitive Data Types and Variables | 11 |
| String | 12 |
| Boolean and Conditional Statements | 13 |
| Time, Date, and Datetime | 14 |
| Integer, Long, Double and Decimal | 15 |
| Null Variables | 16 |
| Enums | 16 |
| Summary | 17 |
| Comments, Case Sensitivity, Collections and Loops | 17 |
| Comments | 17 |
| Case Sensitivity | 18 |
| Arrays and Lists | 18 |
| Loops | 19 |
| Sets and Maps | 21 |
| Summary | 22 |
| Classes, Interfaces and Properties | 22 |
| Defining Classes | 22 |
| Private Modifiers | 23 |
| Constructors | 24 |
| Static Variables, Constants, and Methods | 25 |
| Interfaces | 26 |
| Property Syntax | 27 |
| Summary | 27 |

Contents

| | |
|--|-----------|
| sObjects and the Database | 28 |
| What is an sObject? | 28 |
| SOQL and SOSL Queries | 29 |
| Traversing and Querying sObject Relationships | 30 |
| SOQL For Loops | 31 |
| Apex Data Manipulation Language | 32 |
| Summary | 34 |
| Exception Handling | 34 |
| What Is an Exception? | 35 |
| Try, Catch, and Finally Statements | 35 |
| Built-In Exceptions and Common Methods | 38 |
| Catching Different Exception Types | 42 |
| Creating Custom Exceptions | 43 |
| Summary | 45 |
| Part 3: Apex in Context | 46 |
| Executing Data Operations as a Single Transaction | 46 |
| Adding Custom Business Logic Using Triggers | 48 |
| Creating a Trigger | 48 |
| Invoking the Trigger | 49 |
| Summary | 50 |
| Apex Unit Tests | 50 |
| Adding a Test Utility Class | 51 |
| Add Test Methods | 52 |
| Run Tests and Code Coverage | 54 |
| Summary | 55 |
| Running Apex Within Governor Execution Limits | 55 |
| Scheduled Execution of Apex | 58 |
| Adding a Class that Implements the Schedulable Interface | 58 |
| Adding a Test for the Schedulable Class | 59 |
| Scheduling and Monitoring Scheduled Jobs | 60 |
| Summary | 61 |
| Apex Batch Processing | 61 |
| Adding a Batch Apex Class | 62 |
| Adding a Test for the Batch Apex Class | 64 |
| Running a Batch Job | 65 |
| Summary | 66 |
| Apex REST | 67 |
| Add a Class as a REST Resource | 67 |
| Creating a Record Using the Apex REST POST Method | 68 |
| Retrieving a Record Using the Apex REST GET Method | 69 |
| Summary | 70 |
| Visualforce Pages with Apex Controllers | 70 |
| Enabling Visualforce Development Mode | 70 |

Contents

| | |
|--|----|
| Creating a Simple Visualforce Page | 71 |
| Displaying Product Data in a Visualforce Page | 72 |
| Using a Custom Apex Controller with a Visualforce Page | 75 |
| Using Inner Classes in an Apex Controller | 76 |
| Adding Action Methods to an Apex Controller | 78 |
| Summary | 81 |

APEX WORKBOOK

Force.com Apex is a strongly-typed, object-oriented programming language that allows you to write code that executes on the Force.com platform. Out of the box, Force.com provides a lot of high-level services, such as Web services, scheduling of code execution, batch processing, and triggers—all of which require you to write Apex.

About the Apex Workbook

This workbook provides an introduction to both the Apex programming language, as well as the contexts in which you can use Apex—such as triggers.

This workbook does assume you know a little about programming. If you don't, you'll still manage to follow along, but it will be a little more difficult. We recommend [Head First Java](#) to start learning about programming. Although the book is about Java, Java is quite similar to Apex in many ways, and it will provide the foundation you need.

The workbook is organized into three chapters:

- **Part 1: Orientation** shows you the basics: how to create a simple Apex class, and how to use the Developer Console to execute Apex snippets.
- **Part 2: Apex Language Fundamentals** looks at the syntax, type system, and database integration found in the Apex language.
- **Part 3: Apex in Context** looks at how to use Apex to write triggers, unit tests, scheduled Apex, batch Apex, REST Web services, and Visualforce controllers.

The goal is to give you a tour of Apex, not build a working application. While touring along, feel free to experiment. Change the code a little, substitute a different component—have fun!

Intended Audience

This workbook is intended for developers new to the Force.com platform who want an introduction to Apex development on the platform, and for Salesforce admins who want to delve more deeply into app development using coding. If you're an admin just getting started with Force.com, see the [Force.com Platform Fundamentals](#) for an introduction to point-and-click app development.

Supported Browsers

Microsoft Edge

Salesforce supports Microsoft Edge on Windows 10 for Salesforce Classic. Note these restrictions.

- The HTML solution editor in Microsoft Edge isn't supported in Salesforce Knowledge.
- Microsoft Edge isn't supported for the Developer Console.
- Microsoft Edge isn't supported for Salesforce CRM Call Center built with CTI Toolkit version 4.0 or higher.

Before You Begin

You'll need a Force.com environment that supports Force.com development. These tutorials are designed to work with a Force.com Developer Edition environment, which you can get for free at <https://developer.salesforce.com/signup>.

1. In your browser go to <https://developer.salesforce.com/signup>.
2. Fill in the fields about you and your company.
3. In the `Email Address` field, make sure to use a public address you can easily check from a Web browser.
4. Enter a unique `Username`. Note that this field is also in the *form* of an email address, but it does not have to be the same as your email address, and in fact, it's usually better if they aren't the same. Your username is your login and your identity on `developer.salesforce.com`, and so you're often better served by choosing a username that describes the work you're doing, such as `develop@workbook.org`, or that describes you, such as `firstname@lastname.com`.
5. Read and then select the checkbox for the `Master Subscription Agreement`.
6. Enter the Captcha words shown and click **Submit Registration**.
7. In a moment you'll receive an email with a login link. Click the link and change your password.

It would also help to have some context by learning a little about Force.com itself, which you can find in the first few tutorials of the [Force.com Workbook](#).

For your convenience, we created a repository of the large code samples contained in this workbook. You can download them from http://bit.ly/ApexWorkbookCode_Spring12.

After You Finish

After you've finished the workbook, you'll be ready to explore a lot more Apex and Force.com development:

- Download the Apex Cheat Sheet at https://developer.salesforce.com/page/Cheat_Sheets.
- Learn more about Force.com and Visualforce from the companion Force.com Workbook and Visualforce Workbook at https://developer.salesforce.com/page/Force.com_workbook.
- Discover more about Force.com, and access articles and documentation, by visiting Salesforce Developers at <https://developer.salesforce.com>. In particular, be sure to check out the [Apex Developer Guide](#).

PART 1: ORIENTATION

In this set of tutorials you set up custom objects and sample data. Also, you learn a few essential skills that you will need before diving into the Apex language.



Tip: You must complete tutorial 1 and 3 in Chapter 1 to support tutorials in chapters 2 and 3. Tutorials 2 and 4 are optional if you're comfortable using development tools—tutorial 2 shows you how to use the Developer Console that you'll use to run all the samples in this workbook, and tutorial 4 shows you how to create a class and call its methods.

- [Creating Warehouse Custom Objects](#) contains the steps for creating the custom objects that are used in the tutorials.
- [Using the Developer Console](#) shows how to use the Developer Console, an essential debugging tool, to execute snippets of Apex and work with the execution log output. You'll be using the Developer Console in this workbook as you learn and debug the language.
- [Creating Sample Data](#) contains sample code that you can use to programmatically create the sample data referenced in the tutorials.
- [Creating and Instantiating Classes](#) introduces Apex classes, which are fundamental to Apex code development.

Creating Warehouse Custom Objects

This workbook has examples that use custom objects. These custom objects are also common with other workbooks, for example, the Force.com workbook, and represent objects used to manage a warehouse application. The objects are:

- Merchandise
- Invoice Statement
- Line Item

You can create these objects using one of two different methods.

- Create the objects manually by completing Tutorials 1, 2, and 3 in the Force.com Workbook (60 minutes).
- Install a package into your org that creates the objects for you (5 minutes).

The remainder of this tutorial explains the second option, how to install a package into your fresh DE org that deploys the custom objects.

While you are logged into your DE org:

1. Using the same browser window that is logged in, open a new browser tab and use it to load http://bit.ly/ApexWorkbookPackage1_4.
2. Click **Continue** > **Next** > **Next** > **Install**.
3. Click **View Components**, then take a quick look at the components you just deployed into your org, including three custom objects (Merchandise, Invoice Statement, and Line Item).

After you're done, you can close this second browser tab and return to the original tab.

Using the Developer Console

The Developer Console lets you execute Apex code statements. It also lets you execute Apex methods within an Apex class or object. In this tutorial you open the Developer Console, execute some basic Apex statements, and toggle a few log settings.

Activating the Developer Console

After logging into your Salesforce environment, the screen displays the current application you're using (in the diagram below, it's **Warehouse**), as well as your name.


Open the Developer Console under **Your Name** or the quick access menu (⚙️).

You can open the Developer Console at any time.

Using the Developer Console to Execute Apex Code

The Developer Console can look overwhelming, but it's just a collection of tools that help you work with code. In this lesson, you'll execute Apex code and view the results in the Log Inspector. The Log Inspector is a useful tool you'll use often.

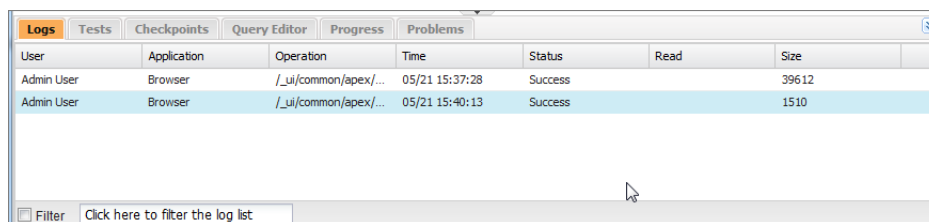
1. Click **Debug > Open Execute Anonymous Window** or CTRL+E.
2. In the Enter Apex Code window, enter the following text: `System.debug('Hello World');`

 **Note:** `System.debug()` is like using `System.out.println()` in Java (or `printf()` if you've been around a while ;-). But, when you're coding in the cloud, where does the output go? Read on!

3. Deselect **Open Log** and then click **Execute**.



Every time you execute code, a log is created and listed in the *Logs* panel.

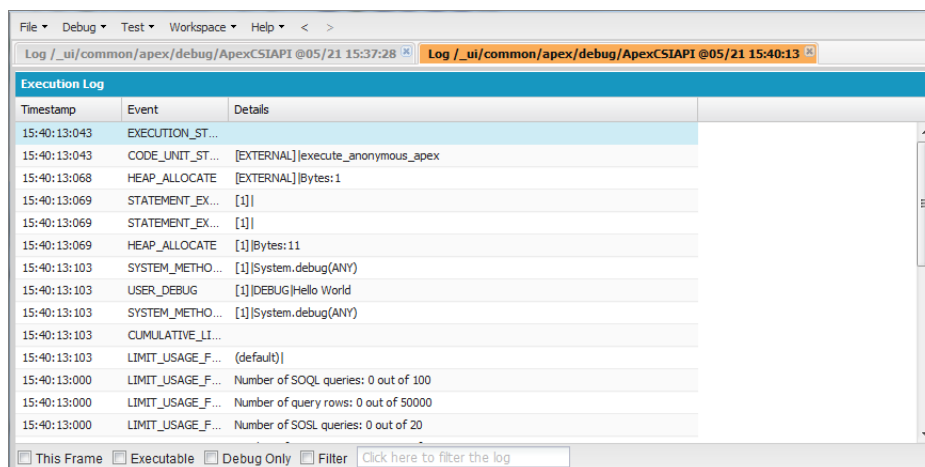


| User | Application | Operation | Time | Status | Read | Size |
|------------|-------------|----------------------|----------------|---------|------|-------|
| Admin User | Browser | /_ui/common/apex/... | 05/21 15:37:28 | Success | | 39612 |
| Admin User | Browser | /_ui/common/apex/... | 05/21 15:40:13 | Success | | 1510 |

Double-click a log to open it in the Log Inspector. You can open multiple logs at a time to compare results.

Log Inspector is a context-sensitive execution viewer that shows the source of an operation, what triggered the operation, and what occurred afterward. Use this tool to inspect debug logs that include database events, Apex processing, workflow, and validation logic.

The Log Inspector includes predefined perspectives for specific uses. Click **Debug > Switch Perspective** to select a different view, or click CTRL+P to select individual panels. You'll probably use the Execution Log panel the most. It displays the stream of events that occur when code executes. Even a single statement generates a lot of events. The Log Inspector captures many event types: method entry and exit, database and web service interactions, and resource limits. The event type `USER_DEBUG` indicates the execution of a `System.debug()` statement.



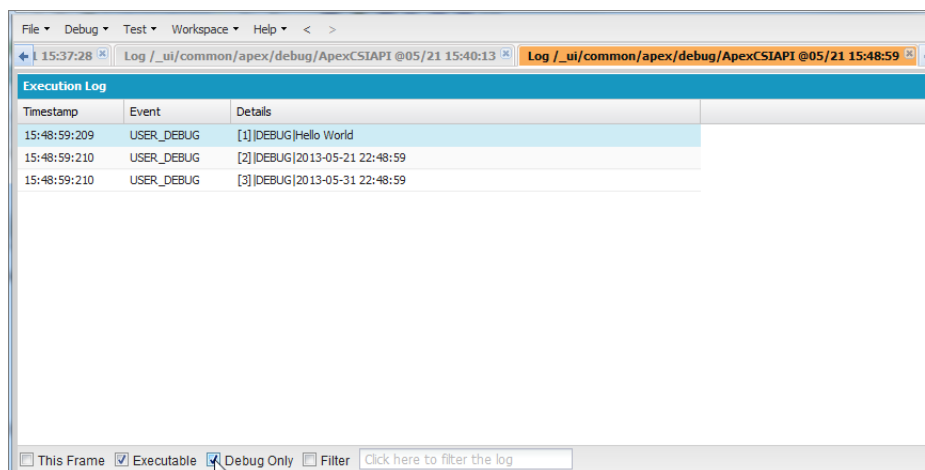
1. Click **Debug > Open Execute Anonymous Window** or CTRL+E and enter the following code:

```
System.debug( 'Hello World' );
System.debug( System.now() );
System.debug( System.now() + 10 );
```

2. Select **Open Log** and click **Execute**.
3. In the Execution Log panel, select **Executable**. This limits the display to only those items that represent executed statements. For example, it filters out the cumulative limits.
4. To filter the list to show only **USER_DEBUG** events, select **Debug Only** or enter *USER* in the **Filter** field.



Note: The filter text is case sensitive.



Congratulations—you have successfully executed code on the Force.com platform and viewed the results!

Tell Me More...

Help in the Developer Console

To learn more about the Developer Console, click **Help > Help Docs...** in the Developer Console.

Anonymous Blocks

The Developer Console allows you to execute code statements on the fly. You can quickly evaluate the results in the **Logs** panel.

The code that you execute in the Developer Console is referred to as an *anonymous block*. Anonymous blocks run as the current user and can fail to compile if the code violates the user's object- and field-level permissions. Note that this isn't the case for Apex classes and triggers.

Summary

To execute Apex code and view the results of the execution, use the Developer Console. The detailed execution results include not only the output generated by the code, but also events that occur along the execution path. Such events include the results of calling another piece of code and interactions with the database.

Creating Sample Data

Prerequisites:

- [Creating Warehouse Custom Objects](#)

Some tutorials in this workbook assume you already have sample data in your database. To be able to execute the examples, you first need to create some sample records.

Use the Developer Console to populate the sample objects you created in Tutorial 1.

1. Open the Developer Console, then click **Debug > Open Execute Anonymous Window** to display the **Enter Apex Code** window.
2. If you installed the package in Tutorial 1, execute the following code:

```
ApexWorkbook.loadData();
```

If you manually created your schema, copy, paste, and execute the code from the following gist URL: <https://gist.github.com/1886593>

3. Once the code executes, close the console.

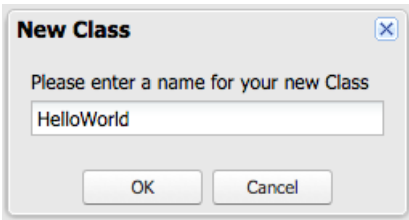
Creating and Instantiating Classes

Apex is an object-oriented programming language, and much of the Apex you write will be contained in classes, sometimes referred to as blueprints or templates for objects. In this tutorial you'll create a simple class with two methods, and then execute them from the Developer Console.

Creating an Apex Class Using the Developer Console

To create an Apex class in the Developer Console:

1. Open the Developer Console.
2. Click **File > New > Apex Class**.
3. Enter `HelloWorld` for the name of the new class and click **OK**.



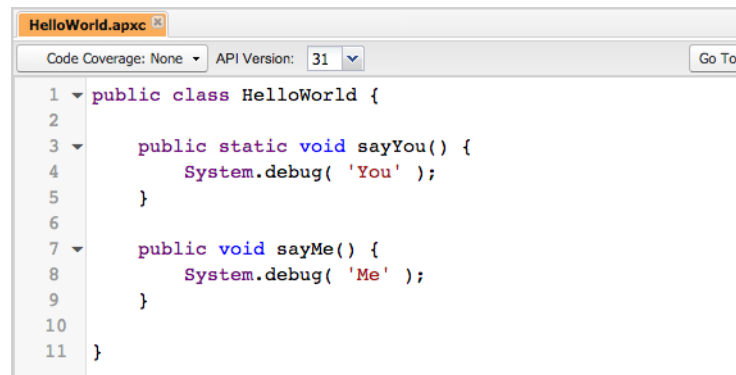
4. A new empty `HelloWorld` class is created. Add a static method to the class by adding the following text between the braces:

```
public static void sayYou() {  
    System.debug( 'You' );  
}
```

5. Add an instance method by adding the following text just before the final closing brace:

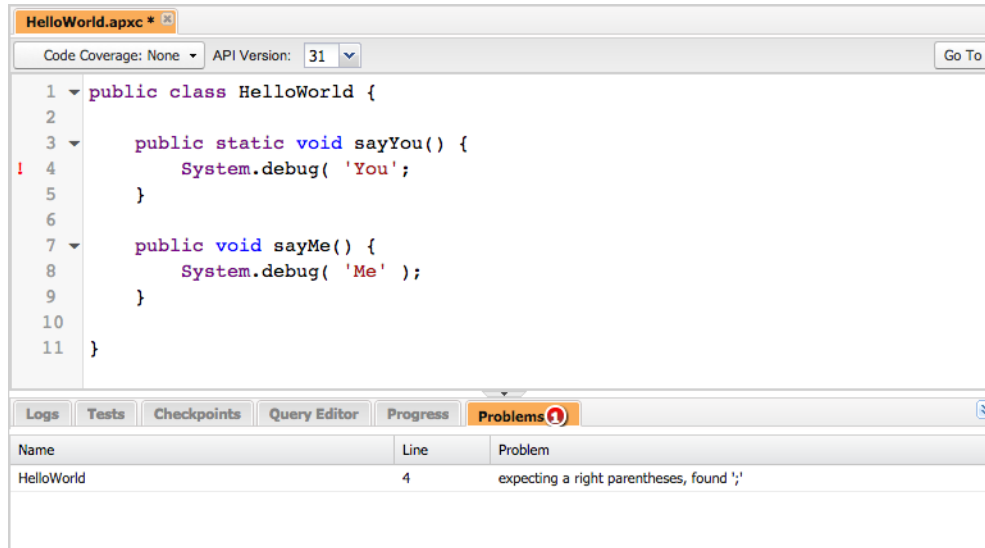
```
public void sayMe() {  
    System.debug( 'Me' );  
}
```

6. Click **File** > **Save**.



Tell Me More...

- You've created a class called `HelloWorld` with a static method `sayYou()` and an instance method `sayMe()`. Looking at the definition of the methods, you'll see that they call another class, `System`, invoking the method `debug()` on that class, which will output strings.
- If you invoke the `sayYou()` method of *your* class, it invokes the `debug()` method of the `System` class, and you see the output.
- The Developer Console validates your code in the background to ensure that the code is syntactically correct and compiles successfully. Making mistakes, such as typos in your code, is inevitable. If you make a mistake in your code, errors appear in the Problems pane and an exclamation mark is added next to the pane heading: **Problems!**
- Expand the Problems panel to see a list of errors. Clicking on an error takes you to the line of code where this error is found. For example, the following shows the error that appears after you omit the closing parenthesis at the end of the `System.debug` statement.



Re-add the closing parenthesis and notice that the error goes away.

Calling a Class Method

Now that you've created the `HelloWorld` class, follow these steps to call its methods.

1. Execute the following code in the Developer Console Execute Anonymous Window to call the `HelloWorld` class's static method. (See [Activating the Developer Console](#) if you've forgotten how to do this.) If there is any existing code in the entry panel, delete it first. Notice that to call a static method, you don't have to create an instance of the class.

```
HelloWorld.sayYou();
```

2. Open the resulting log.
3. Set the filters to show `USER_DEBUG` events. (Also covered in [Activating the Developer Console](#)). "You" appears in the log:

| Execution Log | | |
|---------------|------------|--------------|
| Timestamp | Event | Details |
| 23:03:48:156 | USER_DEBUG | [4]DEBUG:You |

4. Now execute the following code to call the `HelloWorld` class's instance method. Notice that to call an instance method, you first have to create an instance of the `HelloWorld` class.

```
HelloWorld hw = new HelloWorld();
hw.sayMe();
```

5. Open the resulting log and set the filters. "Me" appears in the Details column. This code creates an instance of the `HelloWorld` class, and assigns it to a variable called `hw`. It then calls the `sayMe()` method on that instance.
6. Clear the filters on both logs, and compare the two execution logs. The most obvious differences are related to creating the `HelloWorld` instance and assigning it to the variable `hw`. Do you see any other differences?

Congratulations—you have now successfully created and executed new code on the Force.com platform!

Creating an Apex Class Using the Salesforce User Interface

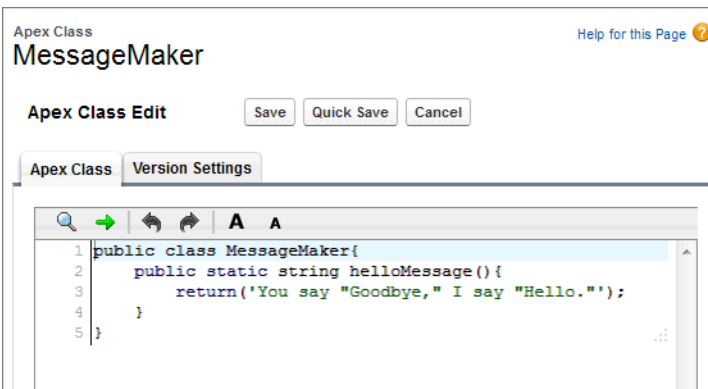
You can also create an Apex class in the Salesforce user interface.

1. From Setup, enter *Apex Classes* in the *Quick Find* box, then select **Apex Classes**.
2. Click **New**.
3. In the editor pane, enter the following code:

```
public class MessageMaker {  
}
```

4. Click **Quick Save**. You could have clicked **Save** instead, but that closes the class editor and returns you to the Apex Classes list. **Quick Save** saves the Apex code, making it available to be executed, but lets you continue editing—making it easier to add to and modify the code.
5. Add the following code to the class:

```
public static string helloMessage() {  
    return('You say "Goodbye," I say "Hello."');  
}
```



6. Click **Save**.

You can also view the class you've just created in the Developer Console and edit it.

1. In the Developer Console, click **File > Open**.
2. In the Entity Type panel, click **Classes**, and then double-click **MessageMaker** from the **Entities** panel.

The `MessageMaker` class displays in the source code editor. You can edit the code there by typing directly in the editor and saving the class.

Summary

In this tutorial you learned how to create and list Apex classes. The classes and methods you create can be called from the Developer Console, as well as from other classes and code that you write.

Tell Me More...

- Alternatively, you can use the [Force.com IDE](#) to create and execute Apex code. For more information, search for “Force.com IDE” on the Developer Force site: <https://developer.salesforce.com/>.

PART 2: APEX LANGUAGE FUNDAMENTALS

Prerequisites:

- The tutorials in this chapter use the Developer Console for executing code snippets. To learn how to use the Developer Console, complete [Using the Developer Console](#).
- The samples in [sObjects and the Database](#) are based on the warehouse custom objects and sample data. To create these, complete [Creating Warehouse Custom Objects](#) and [Creating Sample Data](#).

In [Chapter 1](#) you learned how to create and execute Apex. In this chapter you learn much of the fundamental Apex syntax, data types, database integration and other features that let you create Apex-based application logic.



Tip: If you're familiar with Java, you can glance through or even skip Chapter 2 since Apex has many similarities with Java. You might still want to check out [sObjects and the Database](#) though, which is more specific to Apex, before proceeding to [Part 3: Apex in Context](#).

Here are the tutorials that this chapter contains and a brief description of each.

- [Primitive Data Types and Variables](#) covers the primitive data types and shows how to create and use variables of these types.
- [Comments, Case Sensitivity, Collections and Loops](#) looks at some of the fundamental constructs for creating collections and loops, and adding comments within a class. This tutorial also discusses case sensitivity.
- [Classes, Interfaces and Properties](#) covers some of the basic class and interface constructions.
- [sObjects and the Database](#) introduces a new type that represents objects in the database, and describes how to manipulate these objects.
- [Exception Handling](#) shows how to code for when things go wrong.

In short, this chapter looks at the fundamental Apex constructs that you will need to construct Apex-based logic. Chapter 3 shows you how to call into this logic from database triggers, unit tests, the scheduler, batch Apex, REST Web services, and Visualforce.

Primitive Data Types and Variables

Apex has a number of primitive data types. Your data is stored in a variable matching one of these types, so in this tutorial you will learn a little about most of the available types and how to manipulate their values. Use the Developer Console to execute all of the examples in this tutorial.

These are the data types and variables that this tutorial covers.

- **String:** Strings are set of characters and are enclosed in single quotes. They store text values such as a name or an address.
- **Boolean:** Boolean values hold true or false values and you can use them to test whether a certain condition is true or false.
- **Time, Date and Datetime:** Variables declared with any of these data types hold time, date, or time and date values combined.
- **Integer, Long, Double and Decimal:** Variables declared with any of these data types hold numeric values.
- **Null variables:** Variables that you don't assign values to.
- **Enum:** An enumeration of constant values.

String

Use the `String` data type when you need to store text values, such as a name or an address. Strings are sets of characters enclosed in single quotes. For example, 'I am a string'. You can create a string and assign it to a variable simply by executing the following:

```
String myVariable = 'I am a string.';
```

The previous example creates an instance of the `String` class, represented by the variable `myVariable`, and assigns it a string value between single quotes.

You can also create strings from the values of other types, such as dates, by using the `String` static method `valueOf()`. Execute the following:

```
Date myDate = Date.today();
String myString = String.valueOf(myDate);
System.debug(myString);
```

The output of this example should be today's date. For example, 2012-03-15. You'll likely see a different date.

The `+` operator acts as a concatenation operator when applied to strings. The following results in a single string being created:

```
System.debug('I am a string' + ' cheese');
```

The `==` and `!=` operators act as a case insensitive comparisons. Execute the following to confirm that both the comparisons below return `true`:

```
String x = 'I am a string';
String y = 'I AM A STRING';
String z = 'Hello!';
System.debug (x == y);
System.debug (x != z);
```

The `String` class has many instance methods that you can use to manipulate or interrogate a string. Execute the following:


```
String x = 'The !shorn! sheep !sprang!.';
System.debug (x.endsWith('.'));
System.debug (x.length());
System.debug (x.substring(5,10));
System.debug (x.replaceAll ('!(.*?)!', '$1'));
```

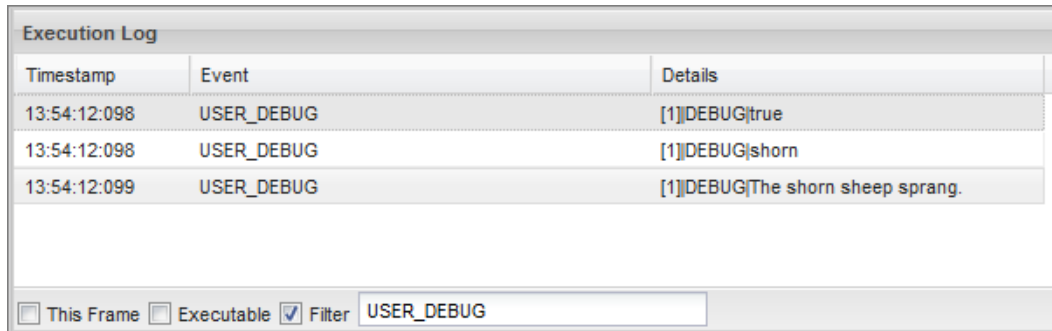
This is the output.

| | |
|----------|-------------------------|
| [2]DEBUG | true |
| [3]DEBUG | 27 |
| [4]DEBUG | shorn |
| [5]DEBUG | The shorn sheep sprang. |

Let's take a look at what each method does.

- The `endsWith` method returns `true` because the string ends with the same string as that in the argument.
- The `length` method returns the length of the string.
- The `substring` method produces a new string starting from the character specified in the first index argument, counting from zero, through the second argument.
- The `replaceAll` method replaces each substring that matches a regular expression with the specified replacement. In this case, we match for text within exclamation points, and replace that text with what was matched (the `$1`).

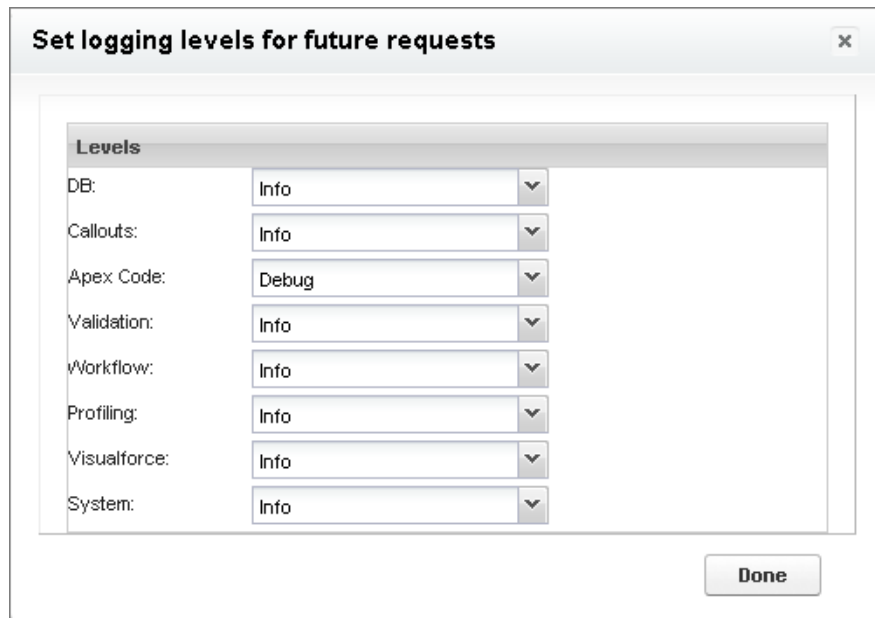
 **Tip:** You can filter the log output in the Developer Console to view only lines that contain “USER_DEBUG”. See [Tutorial 2: Lesson 2](#) for steps of how to do this. That way, you can view only the debug statements of the previous example without having to read the whole log output.



| Timestamp | Event | Details |
|--------------|------------|----------------------------------|
| 13:54:12:098 | USER_DEBUG | [1]DEBUG true |
| 13:54:12:098 | USER_DEBUG | [1]DEBUG shorn |
| 13:54:12:099 | USER_DEBUG | [1]DEBUG The shorn sheep sprang. |

☐ This Frame
 ☐ Executable
 ☒ Filter

In addition, you can set the log level of System to `Info` in the Developer Console to exclude logging of system methods. To access the log levels, click **Log Levels** and then set **System** to `Info`.



Set logging levels for future requests ✕

| Levels | |
|--------------|---------|
| DB: | Info ▼ |
| Callouts: | Info ▼ |
| Apex Code: | Debug ▼ |
| Validation: | Info ▼ |
| Workflow: | Info ▼ |
| Profiling: | Info ▼ |
| Visualforce: | Info ▼ |
| System: | Info ▼ |

Done

In future lessons, you won't be asked to use the filter or even use `System.debug` to show the values. We'll just assume you're doing it!

Boolean and Conditional Statements

Use the Boolean data type when a variable has a `true` or `false` value. You've already encountered Boolean values in the previous lesson as return values: the `endsWith` method returns a Boolean value and the `==` and `!=` String operators return a Boolean value based on the result of the string comparison. You can also create a variable and assign it a value:

```
Boolean isLeapYear = true;
```

There are several standard operators on Booleans. The negation operator `!` returns `true` if its argument is `false`, and conversely. The `&&` operator returns a logical AND, and the `||` operator a logical OR. For example, all these statements evaluate to `false`:

```
Boolean iAmFalse = !true;
Boolean iAmFalse2 = iAmFalse && true;
Boolean iAmFalse3 = iAmFalse || false;
```

Use the `if` statement to execute logic conditionally, depending on the value of a Boolean:

```
Boolean isLeapYear = true;
if (isLeapYear) {
    System.debug ('It\'s a leap year!');
} else {
    System.debug ('Not a leap year.');
```

Escape sequences: In the previous example, notice that there is a backslash (`\`) character inside the argument of the first `System.debug` statement: `'It\'s a leap year!'`. Single quotes have a special meaning in Apex—they enclose String values. To use them inside a String value, escape them by prepending a backslash (`\`) character for each single quote. This way, Apex knows not to treat the single quote character as the end marker of a String but as a character value within the String. Like the single quote escape sequence, Apex provides more escape sequences that represent special characters inside a String and they are: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

In the previous example, the `else` part is optional. The blocks, the statements within the curly braces, can contain any number of statements that are executed when the condition is met. For example, this code generates the output of the two debug statements:

```
if ('Hello'.endsWith('o')) {
    System.debug('me');
    System.debug('me too!');
}
```

If a block only contains a single statement, the curly braces can be optionally omitted. For example:

```
if (4 > 2) System.debug ('Yep, 4 is greater than 2');
```

There is also a ternary conditional operation, which acts as short hand for an if-then-else statement. The syntax is as follows:

```
x ? y : z
```

and can be read as: if `x`, a Boolean, is `true`, then the result is `y`; otherwise it is `z`. Execute the following:

```
Boolean isIt = true;
String x = 'You are ' + (isIt ? 'great' : 'small');
System.debug(x);
```

The resulting string has the value `'You are great'`.

Time, Date, and Datetime

There are three data types associated with dates and times. The `Time` data type stores times (hours, minutes, second and milliseconds). The `Date` data type stores dates (year, month and day). The `Datetime` data type stores both dates and times.

Each of these classes has a `newInstance` method with which you can construct particular date and time values. For example, execute the following:

```
Date myDate = Date.newInstance(1960, 2, 17);
Time myTime = Time.newInstance(18, 30, 2, 20);
```

```
System.debug(myDate);
System.debug(myTime);
```

This outputs:

```
1960-02-17 00:00:00
```

```
18:30:02.020Z
```

The Date data type does hold a time, even though it's set to 0 by default.

You can also create dates and times from the current clock:

```
Datetime myDateTime = DateTime.now();
Date today = Date.today();
```

The date and time classes also have instance methods for converting from one format to another. For example:

```
Time t = DateTime.now().time();
```

Finally, you can also manipulate and interrogate the values by using a range of instance methods. For example, `Datetime` has the `addHours`, `addMinutes`, `dayOfYear`, `timeGMT` methods and many others. Execute the following:

```
Date myToday = Date.today();
Date myNext30 = myToday.addDays(30);
System.debug('myToday = ' + myToday);
System.debug('myNext30= ' + myNext30);
```

You'll get something like this as the output.

```
2012-02-09 00:00:00
```

```
2011-03-10 00:00:00
```

Integer, Long, Double and Decimal

To store numeric values in variables, declare your variables with one of the numeric data types: Integer, Long, Double and Decimal.

Integer

A 32-bit number that doesn't include a decimal point. Integers have a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647.

Long

A 64-bit number that doesn't include a decimal point. Longs have a minimum value of -2^{63} and a maximum value of $2^{63}-1$.

Double

A 64-bit number that includes a decimal point. Doubles have a minimum value of -2^{63} and a maximum value of $2^{63}-1$.

Decimal

A number that includes a decimal point. Decimal is an arbitrary precision number. Currency fields are automatically assigned the type Decimal.

Execute the following to create variables of each numeric type.

```
Integer i = 1;
Long l = 2147483648L;
Double d = 3.14159;
Decimal dec = 19.23;
```

You can use the `valueOf` static method to cast a string to a numeric type. For example, the following creates an `Integer` from string `'10'`, and then adds 20 to it.

```
Integer countMe = Integer.valueOf('10') + 20;
```

The `Decimal` class has a large number of instance methods for interrogating and manipulating the values, including a suite of methods that work with a specified rounding behavior to ensure an appropriate precision is maintained. The `scale` method returns the number of decimal places, while methods like `divide` perform a division as well as specify a final scale. Execute the following, noting that the first argument to `divide` is the number to divide by, and the second is the scale:

```
Decimal decBefore = 19.23;
Decimal decAfter = decBefore.Divide(100, 3);
System.debug(decAfter);
```

The value of `decAfter` will be set to 0.192.

Null Variables

If you declare a variable and don't initialize it with a value, it will be `null`. In essence, `null` means the absence of a value. You can also assign `null` to any variable declared with a primitive type. For example, both of these statements result in a variable set to `null`:

```
Boolean x = null;
Decimal d;
```

Many instance methods on the data type will fail if the variable is `null`. In this example, the second statement generates a compilation error.

```
Decimal d;
d.addDays(2);
```

This results in the following error: line 2, column 1: Method does not exist or incorrect signature: `[Decimal].addDays(Integer)`.

See [Exception Handling](#) to learn more about exceptions and exception handling.

Enums

Use enumerations (enums) to specify a set of constants. Define a new enumeration by using the `enum` keyword followed by the list of identifiers between curly braces. Each value in the enumeration corresponds to an `Integer` value, starting from zero and incrementing by one from left to right. Because each value corresponds to a constant, the identifiers are in upper case. For example, this example defines an enumeration called `Season` that contains the four seasons:

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

In the previous example, the `Integer` value of `WINTER` is 0, `SPRING` 1, `SUMMER` 2, `FALL` 3. Once you define your enumeration, you can use the new enum type as a data type for declaring variables. The following example uses the `Season` enum type that is defined first and creates a variable `s` of type `Season`. It then checks the value of the `s` variable and writes a different debug output based on its value. Execute the following:

```
public enum Season {WINTER, SPRING, SUMMER, FALL}
Season s = Season.SUMMER;
if (s == Season.SUMMER) {
    // Will write the string value SUMMER
    System.debug(s);
} else {
```

```
System.debug('Not summer.');
```

This is what you'll see in the debug output: SUMMER.

In addition to enumerations that you can create for your own use, Apex provides built-in enumerations. One example is `System.LoggingLevel` which is used to specify the logging level of the debug output of the `System.debug` method.

Unlike Java, the enum type has no constructor syntax.

Summary

In this tutorial, you learned about the various primitive data types (String, Boolean, and Date types) and learned how to write conditional statements. You also learned about null variables.

Tell Me More...

Here are some additional data types that Apex provides to hold specific types of data.

ID

The ID data type represents an 18-character object identifier. Force.com sets an ID to an object once it is inserted into the database. For example, an ID value can be 'a02D00000006YLCyIAO'.

Blob

The Blob data type represents binary data stored as a single object. Examples of Blob data are attachments to email messages or the body of a document. Blobs can be accepted as Web service arguments. You can convert a Blob data type to String or from String using the `toString` and `valueOf` methods, respectively. The Blob data type is used as the argument type of the `Crypto` class methods.

Comments, Case Sensitivity, Collections and Loops

The previous tutorials showed variable declarations, conditional, and assignment statements. In this tutorial, you receive a tour de force through much of the Apex syntax. Use the Developer Console to execute all of the examples in this tutorial.

Here is an overview of what this tutorial covers.

- Comments: Comments are lines of text that you add to your code to describe what it does.
- Case sensitivity: Apex is case insensitive.
- Collections: Apex supports various types of collections—arrays, lists, sets, and maps.
- Loops: Apex supports do-while, while, and for loops for executing code repeatedly.

Comments

Comments are lines of text that you add to your code to describe what it does. Comments aren't executable code. It's good practice to annotate your code with comments as necessary. This makes the code easier to understand and more maintainable. Apex has two forms of comments. The first uses the `//` token to mark everything on the same line to the right of the token as a comment. The second encloses a block of text, possibly across multiple lines, between the `/*` and `*/` tokens.

Execute the following. Only the debug statement runs:

```
System.debug ('I will execute');    // This comment is ignored.  
/*
```

```
I am a large comment, completely ignored as well.  
*/
```

Case Sensitivity

Unlike Java, Apex is case insensitive. This means that all Apex code, including method names, class names, variable names and keywords, can be written without regard to case. For example, `Integer myVar;` and `integer MYVAR;` are equivalent statements. All of the following statements print out today's date using the `System.today` method when you execute them in the Developer Console:

```
System.debug ( System.today() );  
System.debug ( System.Today() );  
System.debug ( SyStEm.Today() );
```

A good practice is for class names to start with an uppercase letter and method names to start with a lowercase letter.

Arrays and Lists

Apex has a list collection type that holds an ordered collection of objects. List elements can be accessed with an index or can be sorted if they're primitive types, such as Integers or Strings. You'll typically use a list whenever you want to store a set of values that can be accessed with an index. As you'll see in later tutorials, lists are also used to hold the results of queries.

You can access an element of a list using a zero-based index, or by iterating over the list. Here's how to create a new list, and display its size:

```
List<Integer> myList = new List<Integer>();  
System.debug(myList.size());
```

Arrays in Apex are synonymous with lists—Apex provides an array-like syntax for accessing lists. Here is an alternative way to create exactly the same list:

```
Integer[] myList = new List<Integer>();
```

You can also define a list variable and initialize it at the same time as shown in the following example, which displays the string 'two':

```
List<String> myStrings = new List<String> { 'one', 'two' };
```

To add a new element to a list, use the `add` method.

```
myList.add(101);
```

You can use the array notation to get or modify an existing value.

```
// Get the first element  
Integer i = myList[0];  
// Modify the value of the first element  
myList[0] = 100;
```


Try It Out

This snippet creates a list and adds an integer value to it. It retrieves the first element, which is at index 0, and writes it to the debug output. This example uses both the array notation, by specifying the index between brackets, and the `get` method to retrieve the first element in the list.

```
Integer[] myList = new List<Integer>();
//Adds a new element with value 10 to the end of the list
myList.add(10);

// Retrieve the first element of the list
// using array notation
Integer i = myList[0];
// or using the get method
Integer j = myList.get(0);
System.debug('First element in the array using myList[0] is ' + i);
System.debug('First element in the array using myList.get(0) is ' + j);
```

Here is a portion of the output when you run this snippet in the Developer Console:

| | | |
|--------------|------------|--|
| 17:07:19:034 | USER_DEBUG | [10]DEBUG First element in the array using myList[0] is 10 |
| 17:07:19:034 | USER_DEBUG | [11]DEBUG First element in the array using myList.get(0) is 10 |

This next snippet creates a list and adds an integer value to it. It modifies the value of the first element and writes it to the debug output. Finally, it writes the size of the list to the debug output. This example uses both the array notation, by specifying the index between brackets, and the `set` method to modify the first element in the list.

```
Integer[] myList = new List<Integer>{10, 20};

// Modify the value of the first element
// using the array notation
myList[0] = 15;
// or using the set method
myList.set(0,15);
System.debug ('Updated value:' + myList[0]);

// Return the size of the list
System.debug ('List size: ' + myList.size());
```

Here is a portion of the output when you run this snippet in the Developer Console:

| | | |
|--------------|------------|---------------------------|
| 17:09:08:033 | USER_DEBUG | [8]DEBUG Updated value:15 |
| 17:09:08:033 | USER_DEBUG | [11]DEBUG List size: 2 |

Loops

To repeatedly execute a block of code while a given condition holds true, use a loop. Apex supports do-while, while, and for loops.

While Loops

A do-while loop repeatedly executes a block of code as long as a Boolean condition specified in the `while` statement remains true. Execute the following code:

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

The previous example executes the statements included within the do-while block 10 times and writes the numbers 1 through 10 to the debug output.

| |
|-------------|
| [3]DEBUG 1 |
| [3]DEBUG 2 |
| [3]DEBUG 3 |
| [3]DEBUG 4 |
| [3]DEBUG 5 |
| [3]DEBUG 6 |
| [3]DEBUG 7 |
| [3]DEBUG 8 |
| [3]DEBUG 9 |
| [3]DEBUG 10 |

The while loop repeatedly executes a block of code as long as a Boolean condition specified at the beginning remains true. Execute the following code, which also outputs the numbers 1 - 10.

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

For Loops

There are three types of for loops. The first type of for loop is a traditional loop that iterates by setting a variable to a value, checking a condition, and performing some action on the variable. Execute the following code to write the numbers 1 through 10 to the output:

```
for (Integer i = 1; i <= 10; i++){
    System.debug(i);
}
```

A second type of for loop is available for iterating over a list or a set. Execute the following code:

```
Integer[] myInts = new Integer[] {10,20,30,40,50,60,70,80,90,100};
for (Integer i: myInts) {
    System.debug(i);
}
```

The previous example iterates over every integer in the list and writes it to the output.

| |
|--------------|
| [3]DEBUG:10 |
| [3]DEBUG:20 |
| [3]DEBUG:30 |
| [3]DEBUG:40 |
| [3]DEBUG:50 |
| [3]DEBUG:60 |
| [3]DEBUG:70 |
| [3]DEBUG:80 |
| [3]DEBUG:90 |
| [3]DEBUG:100 |

The third type of for loop is discussed in [Tutorial 8: Lesson 4](#).

Sets and Maps

Besides Lists, Apex supports two other collection types: *Sets* and *Maps*.

Sets

A set is an unordered collection of objects that doesn't contain any duplicate values. Use a set when you don't need to keep track of the order of the elements in the collection, and when the elements are unique and don't have to be sorted.

The following example creates and initializes a new set, adds an element, and checks if the set contains the string 'b': You can run this example in the Developer Console.

```
Set<String> s = new Set<String>{'a','b','c'};
// Because c is already a member, nothing will happen.
s.add('c');
s.add('d');
if (s.contains('b')) {
    System.debug ('I contain b and have size ' + s.size());
}
```

After running the example, you will see this line in the output:

| | | |
|--------------|------------|--------------------------------------|
| 12:39:20:050 | USER_DEBUG | [5]DEBUG:I contain b and have size 4 |
|--------------|------------|--------------------------------------|

Maps

Maps are collections of key-value pairs, where the keys are of primitive data types. Use a map when you want to store values that are to be referenced through a key. For example, using a map you can store a list of addresses that correspond to employee IDs. This example shows how to create a map, add items to it, and then retrieve one item based on an employee ID, which is the key. The retrieved address is written to the debug output.

```
Map<Integer,String> employeeAddresses = new Map<Integer,String>();
employeeAddresses.put (1, '123 Sunny Drive, San Francisco, CA');
employeeAddresses.put (2, '456 Dark Drive, San Francisco, CA');
System.debug('Address for employeeID 2: ' + employeeAddresses.get(2));
```

After running the example, you will see this line in the output:

| | | |
|--------------|------------|--|
| 12:46:50:035 | USER_DEBUG | [4]DEBUG Address for employeeID 2: 456 Dark Drive, San Francisco, CA |
|--------------|------------|--|

Maps also support a shortcut syntax for populating the collection when creating it. The following example creates a map with two key-value pairs. If you execute it, the string 'apple' will be displayed in the debug output.

```
Map<String,String> myStrings =  
new Map<String,String>{ 'a'=>'apple', 'b'=>'bee' };  
System.debug(myStrings.get('a'));
```

Sets and maps contain many useful methods. For example, you can add all elements of one set to another using the `addAll` method on a set. Also, you can return the list of values in a map by calling `values`.

Summary

In this tutorial, you learned how to add comments to your code. In addition, you learned that Apex is a case-insensitive language. Finally, you were introduced to collections (lists, maps, and sets) and loops.

Classes, Interfaces and Properties

Apex is an object-oriented programming language and this tutorial examines its support for these all important objects or class instances as they're sometimes called. Objects are created from classes—data structures that contains class methods, instance methods, and data variables. Classes, in turn, can implement an interface, which is simply a set of methods. Use the Developer Console to execute all of the examples in this tutorial.

Here is an overview of what this tutorial covers.

- **Classes and Objects:** Classes are templates from which you can create objects.
- **Private Modifiers:** The private modifier restricts access to a class, or a class method or member variable contained in a class, so that they aren't available to other classes.
- **Static Variables, Constants and Methods:** Static variables, constants, and methods don't depend on an instance of a class and can be accessed without creating an object from of a class.
- **Interfaces:** Interfaces are named sets of method signatures that don't contain any implementation.
- **Properties:** Properties allow controlled read and write access to class member variables.

Defining Classes

Apex classes are similar to Java classes. A *class* is a template or blueprint from which objects are created. An object is an instance of a class. For example, a *Fridge* class describes the state of a fridge and everything you can do with it. An instance of the *Fridge* class is a specific refrigerator that can be purchased or sold.

An Apex 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 referred to as member variables. Methods are used to control behavior, such as purchasing or selling an item.

Methods can also contain local variables that are declared inside the method and used only by the method. Whereas class member variables define the attributes of an object, such as name or height, local variables in methods are used only by the method and don't describe the class.

[Creating and Instantiating Classes](#) in Chapter 1 of this workbook shows how to create a new class. Follow the same procedure, and create the following class:

```
public class Fridge {
    public String modelNumber;
    public Integer numberInStock;

    public void updateStock(Integer justSold) {
        numberInStock = numberInStock - justSold;
    }
}
```

You've just defined a new class called `Fridge`. The class has two member variables, `modelNumber` and `numberInStock`, and one method, `updateStock`. The `void` type indicates that the `updateStock` method doesn't return a value.

You can now declare variables of this new class type `Fridge`, and manipulate them. Execute the following in the Developer Console:

```
Fridge myFridge = new Fridge();
myFridge.modelNumber = 'MX-O';
myFridge.numberInStock = 100;
myFridge.updateStock(20);
Fridge myOtherFridge = new Fridge();
myOtherFridge.modelNumber = 'MX-Y';
myOtherFridge.numberInStock = 50;
System.debug('myFridge.numberInStock=' + myFridge.numberInStock);
System.debug('myOtherFridge.numberInStock=' + myOtherFridge.numberInStock);
```

This creates a new instance of the `Fridge` class, called `myFridge`, which is an object. It sets the values of the variables in the object, and then calls the `updateStock` method, passing in an argument of value 20. When this executes, the `updateStock` instance method will subtract the argument from the `numberInStock` value. Next, it creates another instance of the `Fridge` class and sets its stock number to 50. When it finally outputs the values, it displays 80 and 50.

| | | |
|--------------|------------|---|
| 23:39:28:045 | USER_DEBUG | [8]DEBUG myFridge.numberInStock=80 |
| 23:39:28:045 | USER_DEBUG | [9]DEBUG myOtherFridge.numberInStock=50 |

Private Modifiers

The class, class methods, and member variables were all declared using the `public` keyword until now. This is an access modifier that ensures other Apex classes also have access to the class, methods, and variables. Sometimes, you might want to hide access for other Apex classes. This is when you declare the class, method, or member variable with the `private` access modifier.

By declaring the member variables as `private`, you have control over which member variables can be read or written, and how they're manipulated by other classes. You can provide public methods to get and set the values of these private variables. These getter and setter methods are called properties and are covered in more detail in [Property Syntax](#). Declare methods as `private` when these methods are only to be called within the defining class and are helper methods. Helper methods don't represent the behavior of the class but are there to serve some utility purposes.



Note: By default, a method or variable is `private` and is visible only to the Apex code within the defining class. You must explicitly specify a method or variable as `public` in order for it to be available to other classes.

Let's modify our `Fridge` class to use `private` modifiers for the member variables.

1. Modify the `Fridge` class and change the modifier of both variables to `private`:

```
private String modelNumber;
private Integer numberInStock;
```

2. Click **Quick Save**.
3. Execute the following in the Developer Console:

```
Fridge myFridge = new Fridge();
myFridge.modelNumber = 'MX-EO';
```

You'll receive an error warning: `Variable is not visible: modelNumber`. The variable `modelNumber` is now only accessible from within the class—a good practice.

4. To provide access to it, define a new public method that can be called to set its value and another to get its value. Add the following inside the class body of `Fridge`.

```
public void setModelNumber(String theModelNumber) {
    modelNumber = theModelNumber;
}

public String getModelNumber() {
    return modelNumber;
}
```

5. Click **Quick Save**.
6. Execute the following:

```
Fridge myFridge = new Fridge();
myFridge.setModelNumber('MX-EO');
System.debug(myFridge.getModelNumber());
```

This will execute properly. The call to the `setModelNumber` method passes in a string which sets the `modelNumber` value of the `myFridge` instance variable. The call to the `getModelNumber` method retrieves the model number, which is passed to the `System.debug` system method for writing it to the debug output.

Constructors

Apex provides a default constructor for each class you create. For example, you were able to create an instance of the `Fridge` class by calling `new Fridge()`, even though you didn't define the `Fridge` constructor yourself. However, the `Fridge` instance in this case has all its member variables set to `null` because all uninitialized variables in Apex are `null`. Sometimes you might want to provide specific initial values, for example, number in stock should be 0 and the model number should be a generic number. This is when you'll want to write your own constructor. Also, it's often useful to have a constructor that takes parameters so you can initialize the member variables from the passed in argument values.

Try adding two constructors, one without parameters and one with parameters.

1. Add the following to your `Fridge` class:

```
public Fridge() {
    modelNumber = 'XX-XX';
    numberInStock = 0;
}

public Fridge(String theModelNumber, Integer theNumberInStock) {
```

```

    modelNumber = theModelNumber;
    numberInStock = theNumberInStock;
}

```

The constructor looks like a method, except it has the same name as the class itself, and no return value.

2. You can now create an instance and set the default values all at once using the second constructor you've added. Execute the following:

```

Fridge myFridge = new Fridge('MX-EO', 100);
System.debug (myFridge.getModelNumber());

```

This will output 'MX-EO'. You'll often see classes with a variety of constructors that aid object creation.

Static Variables, Constants, and Methods

The variables and methods you've created so far are instance methods and variables, which means you have to first create an instance of the class to use the `modelNumber` and `numberInStock` variables. Each individual instance has its own copy of instance variables, and the instance methods can access these variables. There are times when you need to have a member variable whose value is available to all instances, for example, a stock threshold variable whose value is shared with all instances of the `Fridge` class, and any update made by one instance will be visible to all other instances. This is when you need to create a static variable. Static variables are associated with the class and not the instance and you can access them without instantiating the class.

You can also define static methods which are associated with the class, not the instance. Typically, utility methods that don't depend on the state of an instance are good candidates for being static.

1. Modify the `Fridge` class by adding the following static variable:

```

public static Integer stockThreshold = 5;

```

2. Execute the following in the Developer Console:

```

System.debug ( Fridge.stockThreshold );

```

This will output 5. Note how you didn't have to create an instance of the `Fridge` class using the `new` operator. You just accessed the variable on the class.

3. You can also change the value of this static variable by accessing it through the class name.

```

// Modify the static stock threshold value
Fridge.stockThreshold = 4;
System.debug ( Fridge.stockThreshold );

```

This will write 4 to the output.

4. Sometimes you want to declare a variable as being a constant—something that won't change. You can use the `final` keyword to do this in Apex; it indicates that the variable might be assigned to a value no more than once. Modify the static variable you just declared to as follows:

```

public static final Integer STOCK_THRESHOLD = 5;

```

You can still output the value of the field, for example, `Fridge.STOCK_THRESHOLD`; will work, but you can now not assign any other value to the field, for example, `Fridge.STOCK_THRESHOLD = 3`; won't work.

- Let's define a static class method that prints out the values of a given object that gets passed in as an argument. This will be a great help for debugging. Add a new method to the `Fridge` class:

```
public static void toDebug(Fridge aFridge) {
    System.debug ('ModelNumber = ' + aFridge.modelNumber);
    System.debug ('Number in Stock = ' + aFridge.numberInStock);
}
```

- Test out this new method by calling it in the Developer Console and passing in a `Fridge` instance:

```
Fridge myFridge = new Fridge('MX-Y', 200);
Fridge.toDebug(myFridge);
```

This is the output you'll get in the Developer Console.

| | | |
|--------------|------------|---------------------------------|
| 23:36:29:039 | USER_DEBUG | [23]DEBUG ModelNumber = MX-Y |
| 23:36:29:039 | USER_DEBUG | [24]DEBUG Number in Stock = 200 |

You now have an easy way to dump any object you create to the Developer Console!

Interfaces

An interface is a named set of method signatures (the return and parameter definitions), but without any implementation. Interfaces 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. For example, a fridge is a type of kitchen appliance, and so is a toaster. Since every kitchen appliance has a model number, the corresponding interface can have a `getModelNumber` method. However, the format of the model number is different for different appliances. The `Fridge` class and the `Toaster` class can implement this method such that they return different formats for the model number.

Interfaces can be handy—they specify a sort of contract. If any class implements an interface, you can be guaranteed that the methods in the interface will appear in the class. Many different classes can implement the same interface.

Try it out by creating an interface that is implemented by the `Fridge` and `Toaster` classes.

- Create an interface in the same way that you create a class:

```
public interface KitchenUtility {

    String getModelNumber();

}
```

- Modify your `Fridge` class to implement this interface. Simply add the words in bold to the definition of the class on the first line.

```
public class Fridge implements KitchenUtility {
```

- Now define a new class called `Toaster` that also implements the `KitchenUtility` interface.

```
public class Toaster implements KitchenUtility {

    private String modelNumber;

    public String getModelNumber() {
        return 'T' + modelNumber;
    }
}
```



```

    }

}

```

Because both the `Toaster` and `Fridge` classes implement the same interface, they will both have a `getModelNumber` method. You can now treat any instance of `Toaster` or `Fridge` as a `KitchenUtility`.

4. The following example creates an instance of a `Fridge` and `Toaster`. It then creates an array of `KitchenUtility` objects using these two objects and treating them as `KitchenUtility` instances.

```

Fridge f = new Fridge('MX', 200);
Toaster t = new Toaster();
KitchenUtility [] utilities = new KitchenUtility[] { f, t };
String model = utilities[0].getModelNumber();
System.debug(model);

```

Property Syntax

In [Private Modifiers](#), you modified the variables to be private, ensuring that they can only be accessed through a method. That's a common pattern when developing Apex classes, and there is a shorthand syntax that lets you define a variable and code that should run when the variable is accessed or retrieved.

1. Add a new property, `ecoRating`, to the `Fridge` class by adding the following:

```

public Integer ecoRating {

    get { return ecoRating; }

    set { ecoRating = value; if (ecoRating < 0) ecoRating =0; }

}

```

Think of this as creating a variable `ecoRating`, as well as code that should run when the value is retrieved (the code in the `get` block) and code that should run when the value is set (the code in the `set` block). An automatic variable named `value` is made available to you, so that you know what value is being set. In this case, the properties setter checks for negative `ecoRatings`, and adjusts them to 0.

2. Execute the following code to see a negative rating is converted to 0.

```

Fridge myFridge = new Fridge('E', 10);
myFridge.ecoRating = -5; // calls the setter
System.debug (myFridge.ecoRating); // calls the getter

```

This will output 0.

Summary

In this tutorial, you learned how to define and instantiate a class, and how to add public and private member variables, constants, constructors and methods to your class. You also learned about interfaces and properties.

Tell Me More...

Here are some additional resources to explore.

Subclasses

Apex supports subclasses, allowing you to create a class that extends another class. The subclass inherits all the functionality of that parent class. It can also have additional methods and member variables, and can override the behavior of existing parent class methods.

Static Methods and Instance Methods

Static methods are methods declared with the `static` keyword. They're generally useful as utility methods and never depend on a particular instance member variable value. Because you can only associate a static method with a class, the static method cannot access any instance member variable values of the class. Static variables are only static within the scope of the request. They are not static across the server, or across the entire organization.

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.

Security of Executing Code

Unlike code snippets run in the execute anonymous window in the Developer Console, Apex code in classes (and triggers) runs in system context. Object and field level security settings are not enforced. This means that an Apex class has access to all data in your organization. Make sure you don't inadvertently delete data or expose sensitive data. With great power, comes great responsibility! Note that you can enforce sharing permissions of the currently logged-in user by declaring a class with the `with sharing` keyword. To learn more about triggers, see [Adding Custom Business Logic Using Triggers](#).

For more details, see the [Apex Developer Guide](https://developer.salesforce.com/docs) on the Salesforce Developers documentation site (<https://developer.salesforce.com/docs>).

sObjects and the Database

Apex is tightly integrated with the database, the Force.com persistence layer. In this tutorial, you'll learn how the language can be used to create, persist, and update database objects called sObjects, as well as query the database and iterate over the results. Use the Developer Console to execute all of the examples in this tutorial.

What is an sObject?

An sObject is any object that can be stored in the Force.com platform database. These are not objects in the sense of instances of Apex classes; rather, they are representations of data that has or will be persisted.

These persisted objects can be treated as first class citizens within the Apex language, which makes the database integration particularly intuitive and easy to use.

sObject is a generic abstract type that corresponds to any persisted object type. The generic sObject can be cast into a specific sObject type, such as an account or the `Invoice_Statement__c` custom object.

This creates an invoice statement, which corresponds to the `Invoice_Statement__c` custom object, without setting any fields and assigns the new invoice statement to an sObject.

```
sObject s = new Invoice_Statement__c();
```

The second example creates an invoice statement with some initial values for the `Description__c` and `Status__c` fields and assigns it to a variable of type `Invoice_Statement__c`, which is an sObject type also.

```
Invoice_Statement__c inv = new Invoice_Statement__c(Description__c='Test Invoice',  
Status__c='Pending');
```

This example shows how to cast an sObject variable to another sObject type. It casts the `mySObjectVar` variable to the `Invoice_Statement__c` sObject type.

```
Invoice_Statement__c inv = (Invoice_Statement__c)mySObjectVar;
```

Before inserting a new sObject record, you must set all required fields for a successful insertion. You'll learn in [Apex Data Manipulation Language](#) how to insert new records, among other things, using the Data Manipulation Language (DML).

The fields of an sObject can be set either by passing them as arguments in the constructor or after creating the sObject type by using the dot notation. This example shows how to use the dot notation to set the invoice statement's `Description__c` field to a string value.

```
inv.Description__c = 'Test Invoice';
```

You can also use the dot notation to read field values.

```
ID id = inv.Id;
String x = inv.Name;
```

Now try creating an sObject, and setting and reading its fields. Execute the following:

```
Invoice_Statement__c inv = new Invoice_Statement__c();
inv.Description__c = 'Large invoice';
System.debug('Invoice Description: ' + inv.Description__c);
```

The output of the previous snippet is `Invoice Description: Large invoice`.

SOQL and SOSL Queries

The same way database systems support a query language for data retrieval, the Force.com persistence layer also provides two query languages.

- Salesforce Object Query Language (SOQL) is a query-only language. While similar to SQL in some ways, it's an object query language that uses relationships, not joins, for a more intuitive navigation of data. This is the main query language that is used for data retrieval of a single sObject and its related sObjects. You'll see an example in a minute.
- Salesforce Object Search Language (SOSL) is a simple language for searching across all multiple persisted objects simultaneously. SOSL is similar to Apache Lucene.

You can write queries directly in Apex without much additional code since Apex is tightly integrated with the database.

SOQL Query Examples

A SOQL query is enclosed between square brackets. This example retrieves an sObject (a record from the database) that has the name field value equal to 'Pencils':

```
sObject s = [SELECT Id, Name FROM Merchandise__c WHERE Name='Pencils'];
```

This next example retrieves all matching merchandise items, assuming that there are zero or more merchandise items, and assigns them to a list. It shows how you can include a variable in a SOQL query by preceding it with a colon (:).

```
String myName = 'Pencils';
Merchandise__c[] ms = [SELECT Id FROM Merchandise__c WHERE Name=:myName];
```

Execute the following code to retrieve the first matching merchandise item and assign its `Total_Inventory__c` field to a variable:

```
Double totalInventory = [SELECT Total_Inventory__c
                        FROM Merchandise__c
```

```
WHERE Name = 'Pencils'])[0].Total_Inventory__c;
System.debug('Total inventory: ' + totalInventory);
```

This is what you'll get in the output.

Total inventory: 1000.0

SOSL Query Example

SOSL statements evaluate to a list of lists of sObjects, where each list contains the search results for a particular sObject type. Here's an example that searches all field across all Merchandise__c and Invoice_Statement__c sObjects. Execute the following:

```
List<List<SObject>> searchList = [FIND 'Pencil*' IN ALL FIELDS RETURNING
                                Merchandise__c (Id, Name), Invoice_Statement__c];
Merchandise__c[] merList = ((List<Merchandise__c>)searchList[0]);
Invoice_Statement__c[] invList = ((List<Invoice_Statement__c>)searchList[1]);
System.debug('Found ' + merList.size() + ' merchandise items.');
```

You'll get something similar to this in the output.

Found 1 merchandise items.

Found 0 invoice statements.

Traversing and Querying sObject Relationships

sObject Relationships and Dot Notation

If two sObjects are related to each other via a relationship, you can get a parent sObject of an sObject using the dot notation syntax:

```
sObjectTypeName parentObject = objectA.RelationshipName;
```

You can also access the fields of the parent sObject by appending it to the relationship name:

```
DataType s = objectA.RelationshipName.FieldName;
```

Similarly, you can get the child sObjects of an sObject using the same syntax. The only difference is that you now have a collection of one or more sObject child records, while in the previous case there is only one parent record. The syntax is the following:

```
List<sObjectTypeName> children = objectA.ChildRelationshipName;
```

Querying sObject Relationships

If an sObject is related to another by a master-detail or lookup relationship, you can query the parent sObject field by specifying the relationship name and field name in your SELECT statement as follows:

```
SELECT RelationshipName.Field FROM sObjectName WHERE Where_Condition [...]
```

To fetch child sObjects, specify a nested query that retrieves all request child sObjects and their fields as follows:

```
SELECT field1, field1, ..., (Nested query for child sObjects)
FROM sObjectName WHERE Where_Condition [...]
```

Try It Out

This example shows how to traverse the master-detail relationship that exists between an invoice statement and a line item. It first queries the name of the parent invoice statement for a specific line item by specifying `Invoice_Statement__r.Name` in the query. Next, it retrieves the invoice statement sObject and its name from the returned line item sObject through this statement: `li.Invoice_Statement__r.Name`. Execute the following:

```
Line_Item__c li = [SELECT Invoice_Statement__r.Name FROM Line_Item__c LIMIT 1];
// Traverses a relationship using the dot notation.
System.debug('Invoice statement name: ' + li.Invoice_Statement__r.Name);
```

The `Invoice_Statement__r` field in the SELECT statement ends with `__r`. This suffix indicates that this field is a relationship field. It acts like a foreign key and references the parent invoice statement of the line item queried.

The output returned looks something like:

```
Invoice statement name: INV-0000.
```

This second example demonstrates the retrieval of child sObjects. It retrieves child line items of an invoice statement using the nested query (`SELECT Value__c FROM Line_Items__r`). It then obtains the child line items of the invoice statement through the returned invoice statement sObject.

```
Invoice_Statement__c inv = [SELECT Id, Name, (SELECT Units_Sold__c FROM Line_Items__r)
                           FROM Invoice_Statement__c
                           WHERE Name='INV-0000'];
// Access child records.
List<Line_Item__c> lis = inv.Line_Items__r;
System.debug('Number of child line items: ' + lis.size());
```

The nested query retrieves child records from `Line_Items__r`. The `__r` suffix in `Line_Items__r` indicates that this is the name of relationship. This nested query gets the child line items of the invoice statements using the master-detail relationship represented by `Line_Items__r`.

The sample invoice statement has one line item, so the output of this example is:

```
Number of child line items: 1.
```

SOQL For Loops

Queries can be embedded in the special *for* syntax. This syntax can be used to loop through the sObjects returned by the query, one at a time, or in batches of 200 sObjects when using a list variable to hold the query results. Using a list variable to hold the query results in the SOQL for loop is a good way to query a large number of records since this helps avoid the heap limit, which is one of the governor execution limits. You'll learn more about governor limits in [Running Apex Within Governor Execution Limits](#) in Chapter 3.

Here is a SOQL for loop example. In this example, each iteration of the for loop operates on a single sObject returned by the query. This is inefficient if you perform database operations inside the for loop because they execute once for each sObject and you're more likely to reach certain governor limits.

```
for (Merchandise__c tmp : [SELECT Id FROM Merchandise__c]) {
    // Perform some actions on the single merchandise record.
}
```

A more efficient way is to use a list variable to hold the batch of records returned by each iteration of the for loop. This allows for bulk processing of database operations. The following example uses a list variable in the for loop.

```
for (Merchandise__c[] tmp : [SELECT Id FROM Merchandise__c]) {
    // Perform some actions on the single merchandise record.
}
```

Apex Data Manipulation Language

In previous lessons in this tutorial, you've seen what an sObject is, how to query sObjects and how to traverse relationships between sObjects. Now, you're going to learn how to manipulate records in the database using the Apex Data Manipulation Language (DML). DML enables you to insert, update, delete or restore data in the database.

Here is an example that inserts a new invoice statement by calling `insert`. Try it out:

```
Invoice_Statement__c inv = new Invoice_Statement__c(Description__c='My new invoice');
// Insert the invoice using DML.
insert inv;
```

After the invoice statement is inserted, the sObject variable `inv` will contain the ID of the new invoice statement.

Now, let's update the invoice statement by changing its status. Execute the following code to modify the just inserted invoice statement's status and update the record in the database.

```
// First get the new invoice statement
Invoice_Statement__c inv = [SELECT Status__c
                             FROM Invoice_Statement__c
                             WHERE Description__c='My new invoice'];

// Update the status field
inv.Status__c = 'Negotiating';
update inv;
```

We're done with this invoice statement, so let's delete it using the `delete` statement. Try this sample.

```
// First get the new invoice statement
Invoice_Statement__c inv = [SELECT Status__c
                             FROM Invoice_Statement__c
                             WHERE Description__c='My new invoice'];

delete inv;
```

Deleting a record places it in the Recycle Bin from where you can restore it. Records in the Recycle Bin are temporarily stored for 15 days before they're permanently deleted. To restore a record, just use the `undelete` DML statement. Notice that we used the `ALL ROWS` keywords in the SOQL query to be able to retrieve the deleted record.

```
Invoice_Statement__c inv = [SELECT Status__c
                             FROM Invoice_Statement__c
                             WHERE Description__c='My new invoice'
                             ALL ROWS];

undelete inv;
```



Note: Apex supports other DML operations such as `merge` and `upsert`. For more information, see the [Apex Developer Guide](#).

Database DML Methods

Alternatively, you can perform DML operations by calling the methods provided by the `Database` class. The DML statements you've just learned also have corresponding Database methods that can be called on the Database class: `Database.DMLOperation`. The Database DML methods take a single `sObject` or a list of `sObjects` as their first argument. They also take a second optional Boolean argument called `opt_allOrNone` that specifies whether the operation allows for partial success. If set to `false` and a record fails, the remainder of the DML operation can still succeed. The Database DML methods return the results of the DML operation performed.

Here is an example that inserts two invoice statements and allows partial success. It then iterates through the DML results and gets the first error for failed records. Try it out:

```
Invoice_Statement__c inv1 = new Invoice_Statement__c(Description__c='My new invoice');
Invoice_Statement__c inv2 = new Invoice_Statement__c(Description__c='Another invoice');
// Insert the invoice using DML.
Database.SaveResult[] lsr = Database.insert(
    new Invoice_Statement__c[]{inv1, inv2}, false);

// Iterate through the results and
// get the first error for each failed record.
for (Database.SaveResult sr:lsr){
    if(!sr.isSuccess())
        Database.Error err = sr.getErrors()[0];
}
```



Note: Setting the `opt_allOrNone` argument to `false` is a way to avoid getting an exception when a DML operation fails. You'll learn more about exceptions in [Exception Handling](#).

After the invoice statements have been inserted, let's delete them. This next example performs a query first to get the invoices created in the previous example and deletes them. It then iterates through the results of the delete operation and fetches the first error for failed records. Execute the following:

```
Invoice_Statement__c[] invs = [SELECT Id
                                FROM Invoice_Statement__c
                                WHERE Description__c='My new invoice'
                                OR Description__c='Another invoice'];
// Delete the invoices returned by the query.
Database.DeleteResult[] drl = Database.delete(invs, false);

// Iterate through the results and
// get the first error for each failed record.
for (Database.DeleteResult dr:drl){
    if(!dr.isSuccess())
        Database.Error err = dr.getErrors()[0];
}
```

As you've seen in the previous section, deleted records are placed in the Recycle Bin for 15 days. In this example, we'll restore the records we just deleted by calling `Database.undelete`. Notice that we used the `ALL ROWS` keywords in the SOQL query to be able to retrieve the deleted records.

```
Invoice_Statement__c[] invs = [SELECT Status__c
                                FROM Invoice_Statement__c
                                WHERE Description__c='My new invoice'
                                OR Description__c='Another invoice'
                                ALL ROWS];
// Restore the deleted invoices.
```

```
Database.UndeleteResult[] undelRes = Database.undelete(invs, false);

// Iterate through the results and
// get the first error for each failed record.
for (Database.UndeleteResult dr:undelRes){
    if (!dr.isSuccess())
        Database.Error err = dr.getErrors()[0];
}
```

When to Use DML Statements and Database DML Statements

Typically, you will want to use Database methods instead of DML statements if you want to allow partial success of a bulk DML operation by setting the `opt_allOrNone` argument to `false`. In this way, you avoid exceptions being thrown in your code and you can inspect the rejected records in the returned results to possibly retry the operation. (You'll learn about exceptions in the next tutorial: [Exception Handling](#).) Database methods also support exceptions if not setting the `opt_allOrNone` argument to `false`.

Use the DML statements if you want any error during bulk DML processing to be thrown as an Apex exception that immediately interrupts control flow and can be handled using try/catch blocks. This behavior is similar to the way exceptions are handled in most database procedure languages.

Summary

In this tutorial, you learned about sObjects and how to write queries to extract information from the database. You also learned how to use Apex DML to perform insert, update, delete and restore operations.

Tell Me More...

Here are some additional resources to explore.

Rolling Back Transactions and Savepoints

Apex supports rolling back transactions. You can generate a savepoint which sets a point in the request that corresponds to a state in the database. Any DML statement that occurs after the savepoint can be discarded and the database can be restored to the same initial condition. See [Executing Data Operations as a Single Transaction](#) in Chapter 3 of this workbook to learn more about Apex transactions.

Locking Statements

Apex allows you to lock an sObject record to prevent other code from making changes to it. Use the `FOR UPDATE` SOQL statement to lock a record.

sObject Describes

Apex provides methods to perform describes of sObjects. You can obtain a list of all sObjects, as well as a list of fields for an sObject and field attributes. For more information, see the [Apex Developer Guide](#).

Exception Handling

In this tutorial, you'll learn about exceptions in Apex and how to handle exceptions in your code. Also, you'll get an overview of built-in exceptions, and you'll create and throw your own exceptions.

Use the Developer Console to execute all of the examples in this tutorial.

What Is an Exception?

Exceptions note errors and other events that disrupt the normal flow of code execution. `throw` statements are used to generate exceptions, while `try`, `catch`, and `finally` statements are used to gracefully recover from exceptions.

There are many ways to handle errors in your code, including using assertions like `System.assert` calls, or returning error codes or Boolean values, so why use exceptions? The advantage of using exceptions is that they simplify error handling. Exceptions bubble up from the called method to the caller, as many levels as necessary, until a `catch` statement is found that will handle the error. This relieves you from writing error handling code in each of your methods. Also, by using `finally` statements, you have one place to recover from exceptions, like resetting variables and deleting data.

What Happens When an Exception Occurs?

When an exception occurs, code execution halts and any DML operations that were processed prior to the exception are rolled back and aren't committed to the database. Exceptions get logged in debug logs. For unhandled exceptions, that is, exceptions that the code doesn't catch, Salesforce sends an email to the developer with the organization ID and user ID of the running user, as well as the exception message.

If you run into an exception that occurred in Apex code while using the standard user interface, an error message appears on the page showing you the text of the unhandled exception as shown below:

The screenshot shows a Salesforce 'New Merchandise' form. At the top, there's a header 'Merchandise Edit' with buttons for 'Save', 'Save & New', and 'Cancel'. Below this, a red error message is displayed: 'Error: Invalid Data. Review all error messages below to correct your data. Apex trigger myMerchandiseTrigger caused an unexpected exception, contact your administrator: myMerchandiseTrigger: execution of BeforeInsert caused by: System.NullPointerException: Attempt to de-reference a null object: Trigger.myMerchandiseTrigger: line 3, column 1'. Below the error, there's an 'Information' section with a legend indicating that a red bar next to a field name means it is 'Required Information'. The form fields are: 'Merchandise Name' (Erasers), 'Description' (White erasers), 'Price' (1.50), and 'Total Inventory' (120). The 'Owner' is listed as 'Test User'.

Try, Catch, and Finally Statements

Apex uses `try`, `catch` and `finally` statements to handle exceptions. Here is an example of what these statements look like and the order in which they should be written.

```
try {
    // Perform some database operations that
    // might cause an exception.
} catch(DmlException e) {
    // DmlException handling code here.
} catch(Exception e) {
    // Generic exception handling code here.
} finally {
    // Perform some clean up.
}
```

At least a `catch` block or a `finally` block must be present with a `try` block.

The `try` statement identifies a block of code in which an exception can occur. If you have code that you think could generate an exception, wrap this section of your code in a `try` block, and add a `catch` block after it. Only exceptions thrown from the code wrapped in the `try` block are handled by the `catch` and `finally` blocks.

The `catch` statement identifies a block of code that handles a particular type of exception. In the previous example, notice that there are two `catch` statements. You can have as many `catch` statements as you like, one for each exception type you want to catch, or no `catch` statement if you add a `finally` statement.

Order `catch` statements from specific to generic. All exceptions are considered to be of type `Exception`, so if you catch the generic `Exception` type first, the other `catch` statements won't execute—only one `catch` block executes.

In the `catch` statement, handle the exception received. For example, you can perform some logging, send an email, or do some other processing.

The `finally` statement always executes regardless of whether an exception was thrown or the type of exception that was thrown. A single `try` statement can have up to one associated `finally` statement. You can add any final cleanup code here, such as freeing up resources.

Try It Out

To see an exception in action, execute some code that causes a DML exception to be thrown. Execute the following in the Developer Console:

```
Merchandise__c m = new Merchandise__c();
insert m;
```

The `insert` DML statement in the example causes a `DmlException` because we're inserting a merchandise item without setting any of its required fields. This is the exception error that you see in the debug log.

```
System.DmlException: Insert failed. First exception on row 0; first error:
REQUIRED_FIELD_MISSING, Required fields are missing: [Description, Price, Total
Inventory]: [Description, Price, Total Inventory]
```

Next, execute this snippet in the Developer Console. It's based on the previous example but includes a try-catch block.

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
} catch(DmlException e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}
```

Notice that the request status in the Developer Console now reports success. This is because the code handles the exception.

Any statements in the `try` block occurring after the exception are skipped and aren't executed. For example, if you add a statement after `insert m;`, this statement won't be executed. Execute the following:

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
    // This doesn't execute since insert causes an exception
    System.debug('Statement after insert.');
```

```
} catch(DmlException e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}
```

In the new debug log entry, notice that you don't see a debug message of `Statement after insert`. This is because this debug statement occurs after the exception caused by the insertion and never gets executed. To continue the execution of code statements after an exception happens, place the statement after the try-catch block. Execute this modified code snippet and notice that the debug log now has a debug message of `Statement after insert`.

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
} catch(DmlException e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}
// This will get executed
System.debug('Statement after insert.');
```

Alternatively, you can include additional try-catch blocks. This code snippet has the `System.debug` statement inside a second try-catch block. Execute it to see that you get the same result as before.

```
try {
    Merchandise__c m = new Merchandise__c();
    insert m;
} catch(DmlException e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}

try {
    System.debug('Statement after insert.');
```

// Insert other records

```
}
catch (Exception e) {
    // Handle this exception here
}
```

The finally block always executes regardless of what exception is thrown, and even if no exception is thrown. Let's see it used in action. Execute the following:

```
// Declare the variable outside the try-catch block
// so that it will be in scope for all blocks.
XmlStreamWriter w = null;
try {
    w = new XmlStreamWriter();
    w.writeStartDocument(null, '1.0');
    w.writeStartElement(null, 'book', null);
    w.writeCharacters('This is my book');
    w.writeEndElement();
    w.writeEndDocument();

    // Perform some other operations
    String s;
    // This causes an exception because
    // the string hasn't been assigned a value.
    Integer i = s.length();
} catch(Exception e) {
    System.debug('An exception occurred: ' + e.getMessage());
} finally {
    // This gets executed after the exception is handled
```

```

    System.debug('Closing the stream writer in the finally block.');
```

```
// Close the stream writer
w.close();
}
```

The previous code snippet creates an XML stream writer and adds some XML elements. Next, an exception occurs due to accessing the null String variable `s`. The catch block handles this exception. Then the finally block executes. It writes a debug message and closes the stream writer, which frees any associated resources. Check the debug output in the debug log. You'll see the debug message `Closing the stream writer in the finally block.` after the exception error. This tells you that the finally block executed after the exception was caught.



Note: Some exceptions can't be handled, such as exceptions that the runtime throws as a result of reaching a governor limit. You'll learn more about governor limits in [Running Apex Within Governor Execution Limits](#) in Chapter 3.

Built-In Exceptions and Common Methods

Apex provides a number of exception types that the runtime engine throws if errors are encountered during execution. You've seen the `DmlException` in the previous example. Here is a sample of some of the built-in exceptions:

DmlException

Any problem with a DML statement, such as an `insert` statement missing a required field on a record.

For an example that makes use of `DmlException`, see [Try, Catch, and Finally Statements](#).

ListException

Any problem with a list, such as attempting to access an index that is out of bounds.

Try out some code that does some things on purpose to cause this exception to be thrown. Execute the following:

```

try {
    List<Integer> li = new List<Integer>();
    li.add(15);
    // This list contains only one element,
    // but we're attempting to access the second element
    // from this zero-based list.
    Integer i1 = li[0];
    Integer i2 = li[1]; // Causes a ListException
} catch(ListException le) {
    System.debug('The following exception has occurred: ' + le.getMessage());
}
```

In the previous code snippet, we create a list and add one element to it. Then, we attempt to access two elements, one at index 0, which exists, and one at index 1, which causes a `ListException` because no element exists at this index. This exception is caught in the catch block. The `System.debug` statement in the catch block writes the following to the debug log: `The following exception has occurred: List index out of bounds: 1.`

NullPointerException

Any problem with dereferencing a null variable.

Try out some code that does some things on purpose to cause this exception to be thrown. Execute the following:

```

try {
    String s;
    Boolean b = s.contains('abc'); // Causes a NullPointerException
} catch(NullPointerException npe) {
```

```

        System.debug('The following exception has occurred: ' + npe.getMessage());
    }

```

In the previous example, we create a String variable named `s` but we don't initialize it to a value, hence, it is null. Calling the `contains` method on our null variable causes a `NullPointerException`. The exception is caught in our catch block and this is what is written to the debug log: `The following exception has occurred: Attempt to de-reference a null object`.

QueryException

Any problem with SOQL queries, such as assigning a query that returns no records or more than one record to a singleton `sObject` variable.

Try out some code that does some things on purpose to cause this exception to be thrown. Execute the following:

```

try {
    // This statement doesn't cause an exception, even though
    // we don't have a merchandise with name='XYZ'.
    // The list will just be empty.
    List<Merchandise__c> lm = [SELECT Name FROM Merchandise__c WHERE Name='XYZ'];
    // lm.size() is 0
    System.debug(lm.size());

    // However, this statement causes a QueryException because
    // we're assigning the return value to a Merchandise__c object
    // but no Merchandise is returned.
    Merchandise__c m = [SELECT Name FROM Merchandise__c WHERE Name='XYZ' LIMIT 1];
} catch (QueryException qe) {
    System.debug('The following exception has occurred: ' + qe.getMessage());
}

```

The second query in the above code snippet causes a `QueryException`. We're attempting to assign a `Merchandise` object to what is returned from the query. Note the use of `LIMIT 1` in the query. This ensures that at most one object is returned from the database so we can assign it to a single object and not a list. However, in this case, we don't have a `Merchandise` named `XYZ`, so nothing is returned, and the attempt to assign the return value to a single object results in a `QueryException`. The exception is caught in our catch block and this is what you'll see in the debug log: `The following exception has occurred: List has no rows for assignment to SObject`.

SObjectException

Any problem with `sObject` records, such as attempting to change a field in an `update` statement that can only be changed during `insert`.

Try out some code that does some things on purpose to cause this exception to be thrown. Execute the following:

```

try {
    Merchandise__c m = [SELECT Name FROM Merchandise__c LIMIT 1];
    // Causes an SObjectException because we didn't retrieve
    // the Total_Inventory__c field.
    Double inventory = m.Total_Inventory__c;
} catch (SObjectException se) {
    System.debug('The following exception has occurred: ' + se.getMessage());
}

```

Our code snippet queries any `Merchandise` object that is in the database. Note the use of `LIMIT 1` in the query. Since we have sample merchandise items, the first object in the query will be returned and assigned to the `Merchandise` variable `m`. However, we retrieved only the `Name` field in the query and not `Total_Inventory`, so when we attempt to get the `Total_Inventory` value from the merchandise object, we get an `SObjectException`. This exception is caught in our catch block and this is what you'll see in the debug

log: The following exception has occurred: SObject row was retrieved via SOQL without querying the requested field: Merchandise__c.Total_Inventory__c.

Common Exception Methods

You can use common exception methods to get more information about an exception, such as the exception error message or the stack trace. The previous example calls the `getMessage` method, which returns the error message associated with the exception. There are other exception methods that are also available. Here are descriptions of some useful methods:

- `getCause`: Returns the cause of the exception as an exception object.
- `getLineNumber`: Returns the line number from where the exception was thrown.
- `getMessage`: Returns the error message that displays for the user.
- `getStackTraceString`: Returns the stack trace as a string.
- `getTypeName`: Returns the type of exception, such as `DmlException`, `ListException`, `MathException`, and so on.

Try It Out

Let's see what these methods return by running this simple example.

```
try {
    Merchandise__c m = [SELECT Name FROM Merchandise__c LIMIT 1];
    // Causes an SObjectException because we didn't retrieve
    // the Total_Inventory__c field.
    Double inventory = m.Total_Inventory__c;
} catch(Exception e) {
    System.debug('Exception type caught: ' + e.getTypeName());
    System.debug('Message: ' + e.getMessage());
    System.debug('Cause: ' + e.getCause());    // returns null
    System.debug('Line number: ' + e.getLineNumber());
    System.debug('Stack trace: ' + e.getStackTraceString());
}
```

The output of all `System.debug` statements looks like the following:

```
17:38:04:149 USER_DEBUG [7]|DEBUG|Exception type caught: System.SObjectException
17:38:04:149 USER_DEBUG [8]|DEBUG|Message: SObject row was retrieved via SOQL without
querying the requested field: Merchandise__c.Total_Inventory__c
17:38:04:150 USER_DEBUG [9]|DEBUG|Cause: null
17:38:04:150 USER_DEBUG [10]|DEBUG|Line number: 5
17:38:04:150 USER_DEBUG [11]|DEBUG|Stack trace: AnonymousBlock: line 5, column 1
```

The catch statement argument type is the generic `Exception` type. It caught the more specific `SObjectException`. You can verify that this is so by inspecting the return value of `e.getTypeName()` in the debug output. The output also contains other properties of the `SObjectException`, like the error message, the line number where the exception occurred, and the stack trace. You might be wondering why `getCause` returned null. This is because in our sample there was no previous exception (inner exception) that caused this exception. In [Creating Custom Exceptions](#), you'll get to see an example where the return value of `getCause` is an actual exception.

More Exception Methods

Some exception types, such as `DmlException`, have specific exception methods that apply to only them:

- `getDmlFieldNames(Index of the failed record)`: Returns the names of the fields that caused the error for the specified failed record.
- `getDmlId(Index of the failed record)`: Returns the ID of the failed record that caused the error for the specified failed record.
- `getDmlMessage(Index of the failed record)`: Returns the error message for the specified failed record.
- `getNumDml`: Returns the number of failed records.

Try It Out

This snippet makes use of the `DmlException` methods to get more information about the exceptions returned when inserting a list of `Merchandise` objects. The list of items to insert contains three items, the last two of which don't have required fields and cause exceptions.

```
Merchandise__c m1 = new Merchandise__c(
    Name='Coffeemaker',
    Description__c='Kitchenware',
    Price__c=25,
    Total_Inventory__c=1000);
// Missing the Price and Total_Inventory fields
Merchandise__c m2 = new Merchandise__c(
    Name='Coffeemaker B',
    Description__c='Kitchenware');
// Missing all required fields
Merchandise__c m3 = new Merchandise__c();
Merchandise__c[] mList = new List<Merchandise__c>();
mList.add(m1);
mList.add(m2);
mList.add(m3);

try {
    insert mList;
} catch (DmlException de) {
    Integer numErrors = de.getNumDml();
    System.debug('getNumDml=' + numErrors);
    for(Integer i=0;i<numErrors;i++) {
        System.debug('getDmlFieldNames=' + de.getDmlFieldNames(i));
        System.debug('getDmlMessage=' + de.getDmlMessage(i));
    }
}
```

Note how the sample above didn't include all the initial code in the try block. Only the portion of the code that could generate an exception is wrapped inside a `try` block, in this case the `insert` statement could return a DML exception in case the input data is not valid. The exception resulting from the `insert` operation is caught by the `catch` block that follows it. After executing this sample, you'll see an output of `System.debug` statements similar to the following:

```
14:01:24:939 USER_DEBUG [20]|DEBUG|getNumDml=2
14:01:24:941 USER_DEBUG [23]|DEBUG|getDmlFieldNames=(Price, Total Inventory)
14:01:24:941 USER_DEBUG [24]|DEBUG|getDmlMessage=Required fields are missing: [Price,
Total Inventory]
14:01:24:942 USER_DEBUG [23]|DEBUG|getDmlFieldNames=(Description, Price, Total Inventory)
14:01:24:942 USER_DEBUG [24]|DEBUG|getDmlMessage=Required fields are missing:
[Description, Price, Total Inventory]
```

The number of DML failures is correctly reported as two since two items in our list fail insertion. Also, the field names that caused the failure, and the error message for each failed record is written to the output.

Catching Different Exception Types

In the examples of the previous lesson, we used the specific exception type in the catch block. We could have also just caught the generic Exception type in all examples, which catches all exception types. For example, try running this example that throws an SObjectException and has a catch statement with an argument type of Exception. The SObjectException gets caught in the catch block.

```
try {
    Merchandise__c m = [SELECT Name FROM Merchandise__c LIMIT 1];
    // Causes an SObjectException because we didn't retrieve
    // the Total_Inventory__c field.
    Double inventory = m.Total_Inventory__c;
} catch(Exception e) {
    System.debug('The following exception has occurred: ' + e.getMessage());
}
```

Alternatively, you can have several catch blocks—a catch block for each exception type, and a final catch block that catches the generic Exception type. Look at this example. Notice that it has three catch blocks.

```
try {
    Merchandise__c m = [SELECT Name FROM Merchandise__c LIMIT 1];
    // Causes an SObjectException because we didn't retrieve
    // the Total_Inventory__c field.
    Double inventory = m.Total_Inventory__c;
} catch(DmlException e) {
    System.debug('DmlException caught: ' + e.getMessage());
} catch(SObjectException e) {
    System.debug('SObjectException caught: ' + e.getMessage());
} catch(Exception e) {
    System.debug('Exception caught: ' + e.getMessage());
}
```

Remember that only one catch block gets executed and the remaining ones are bypassed. This example is similar to the previous one, except that it has a few more catch blocks. When you run this snippet, an SObjectException is thrown on this line: `Double inventory = m.Total_Inventory__c;`. Every catch block is examined in the order specified to find a match between the thrown exception and the exception type specified in the catch block argument:

1. The first catch block argument is of type `DmlException`, which doesn't match the thrown exception (`SObjectException`).
2. The second catch block argument is of type `SObjectException`, which matches our exception, so this block gets executed and the following message is written to the debug log: `SObjectException caught: SObject row was retrieved via SOQL without querying the requested field: Merchandise__c.Total_Inventory__c`.
3. The last catch block is ignored since one catch block has already executed.

The last catch block is handy because it catches any exception type, and so catches any exception that was not caught in the previous catch blocks. Suppose we modified the code above to cause a `NullPointerException` to be thrown, this exception gets caught in the last catch block. Execute this modified example. You'll see the following debug message: `Exception caught: Attempt to de-reference a null object`.

```
try {
    String s;
    Boolean b = s.contains('abc'); // Causes a NullPointerException
} catch(DmlException e) {
```



```

        System.debug('DmlException caught: ' + e.getMessage());
    } catch(SObjectException e) {
        System.debug('SObjectException caught: ' + e.getMessage());
    } catch(Exception e) {
        System.debug('Exception caught: ' + e.getMessage());
    }

```

Creating Custom Exceptions

Since you can't throw built-in Apex exceptions but can only catch them, you can create custom exceptions to throw in your methods. That way, you can also specify detailed error messages and have more custom error handling in your catch blocks.

To create your custom exception class, extend the built-in `Exception` class and make sure your class name ends with the word `Exception`. Append `extends Exception` after your class declaration as follows.

```
public class MyException extends Exception {}
```

Here are some ways you can create your exceptions objects, which you can then throw.

You can construct exceptions:

- With no arguments:

```
new MyException();
```

- With a single String argument that specifies the error message:

```
new MyException('This is bad');
```

- With a single Exception argument that specifies the cause and that displays in any stack trace:

```
new MyException(e);
```

- With both a String error message and a chained exception cause that displays in any stack trace:

```
new MyException('This is bad', e);
```

Now that you've seen how to create an exception class and how to construct your exception objects, let's create and run an example that demonstrates the usefulness of custom exceptions.

1. In the Developer Console, create a class named `MerchandiseException` and add the following to it:

```
public class MerchandiseException extends Exception {}
```

You'll use this exception class in the second class that you'll create. Note that the curly braces at the end enclose the body of your exception class, which we left empty because we get some free code—our class inherits all the constructors and common exception methods, such as `getMessage`, from the built-in `Exception` class.

2. Next, create a second class named `MerchandiseUtility`.

```

public class MerchandiseUtility {
    public static void mainProcessing() {
        try {
            insertMerchandise();
        } catch(MerchandiseException me) {
            System.debug('Message: ' + me.getMessage());
            System.debug('Cause: ' + me.getCause());
        }
    }
}

```

```

        System.debug('Line number: ' + me.getLineNumber());
        System.debug('Stack trace: ' + me.getStackTraceString());
    }
}

public static void insertMerchandise() {
    try {
        // Insert merchandise without required fields
        Merchandise__c m = new Merchandise__c();
        insert m;
    } catch(DmlException e) {
        // Something happened that prevents the insertion
        // of Employee custom objects, so throw a more
        // specific exception.
        throw new MerchandiseException(
            'Merchandise item could not be inserted.', e);
    }
}
}

```

This class contains the `mainProcessing` method, which calls `insertMerchandise`. The latter causes an exception by inserting a `Merchandise` without required fields. The catch block catches this exception and throws a new exception, the custom `MerchandiseException` you created earlier. Notice that we called a constructor for the exception that takes two arguments: the error message, and the original exception object. You might wonder why we are passing the original exception? Because it is useful information—when the `MerchandiseException` gets caught in the first method, `mainProcessing`, the original exception (referred to as an inner exception) is really the cause of this exception because it occurred before the `MerchandiseException`.

- Now let's see all this in action to understand better. Execute the following:

```
MerchandiseUtility.mainProcessing();
```

- Check the debug log output. You should see something similar to the following:

```

18:12:34:928 USER_DEBUG [6]|DEBUG|Message: Merchandise item could not be inserted.
18:12:34:929 USER_DEBUG [7]|DEBUG|Cause: System.DmlException: Insert failed. First
exception on row 0; first error: REQUIRED_FIELD_MISSING, Required fields are missing:
[Description, Price, Total Inventory]: [Description, Price, Total Inventory]
18:12:34:929 USER_DEBUG [8]|DEBUG|Line number: 22
18:12:34:930 USER_DEBUG [9]|DEBUG|Stack trace:
Class.EmployeeUtilityClass.insertMerchandise: line 22, column 1

```

A few items of interest:

- The cause of `MerchandiseException` is the `DmlException`. You can see the `DmlException` message also that states that required fields were missing.
- The stack trace is line 22, which is the second time an exception was thrown. It corresponds to the throw statement of `MerchandiseException`.

```
throw new MerchandiseException('Merchandise item could not be inserted.', e);
```

Summary

In this tutorial, you learned about exceptions, how to handle them, Apex built-exceptions and common methods, and how to write and throw your own custom exceptions.

PART 3: APEX IN CONTEXT

In Chapter 2 you learned about the fundamental Apex syntax, data types, database integration, and other features of the language. In this chapter you learn how to call into your Apex classes by using the higher-level Force.com platform features:



Tip: The tutorials in this chapter require that you set up custom objects and sample data in [Creating Warehouse Custom Objects](#) and [Creating Sample Data](#).

Here are the tutorials that this chapter contains and a brief description of each.

- [Executing Data Operations as a Single Transaction](#) goes over transactions and discusses the rollback of transactions.
- [Adding Custom Business Logic Using Triggers](#) contains steps to create a trigger and describes what a trigger is and its syntax.
- [Apex Unit Tests](#) contains steps for creating a test factory class for generating test data, and for creating test methods. This tutorial shows you how to run the tests and verify code coverage.
- [Running Apex Within Governor Execution Limits](#) discusses limits in the Force.com multitenant cloud environment and provides some examples.
- [Scheduled Execution of Apex](#) covers the Apex scheduler that lets you execute Apex and particular times of the day.
- [Apex Batch Processing](#) contains steps to create and execute a batch Apex job.
- [Apex REST](#) contains steps for creating an Apex class and exposing it as a REST resource, and calling the REST methods from Workbench.
- [Visualforce Pages with Apex Controllers](#) contains steps for creating a Visualforce page and controller to view and order inventory items.

Executing Data Operations as a Single Transaction

Prerequisites:

- [Creating and Instantiating Classes](#)
- [sObjects and the Database](#)

What Is an Apex Transaction?

An *Apex transaction* represents a set of operations that are executed as a single unit. All DML operations in a transaction either complete successfully, or if an error occurs in one operation, the entire transaction is rolled back and no data is committed to the database. The boundary of a transaction can be a trigger, a class method, an anonymous block of code, a Visualforce page, or a custom Web service method.

All operations that occur inside the transaction boundary represent a single unit of operations. This also applies for calls that are made from the transaction boundary to external code, such as classes or triggers that get fired as a result of the code running in the transaction boundary. For example, consider the following chain of operations: a custom Apex Web service method causes a trigger to fire, which in turn calls a method in a class. In this case, all changes are committed to the database only after all operations in the transaction finish executing and don't cause any errors. If an error occurs in any of the intermediate steps, all database changes are rolled back and the transaction isn't committed.

How Are Transactions Useful?

Transactions are useful when several operations are related, and either all or none of the operations should be committed. This keeps the database in a consistent state. There are many business scenarios that benefit from transaction processing. For example, transferring funds from one bank account to another is a common scenario. It involves debiting the first account and crediting the second account with the amount to transfer. These two operations need to be committed together to the database. But if the debit operation succeeds and the credit operation fails, the account balances will be inconsistent.

Try It Out

This example shows how all database `insert` operations are rolled back when the last operation causes a validation rule failure. In this example, the `invoice` method is the transaction boundary. All code that runs within this method either commits all changes to the database or rolls back all changes. In this case, we add a new invoice statement with a line item for the pencils merchandise. The Line Item is for a purchase of 5,000 pencils specified in the `Units_Sold__c` field, which is more than the entire pencils inventory of 1,000.

The sample Line Item object you created in Chapter 1 includes a validation rule. This validation rule checks that the total inventory of the merchandise item is enough to cover new purchases. Since this example attempts to purchase more pencils (5,000) than items in stock (1,000), the validation rule fails and throws a run-time exception. Code execution halts at this point and all DML operations processed before this exception are rolled back. In this case, the invoice statement and line item won't be added to the database, and their `insert` DML operations are rolled back.

1. Add the following class through the Developer Console.
2. For the class name, type *MerchandiseOperations* and replace the auto-generated code with this example.

```
public class MerchandiseOperations {
    public static Id invoice( String pName, Integer pSold, String pDesc) {
        // Retrieve the pencils sample merchandise
        Merchandise__c m = [SELECT Price__c, Total_Inventory__c
            FROM Merchandise__c WHERE Name = :pName LIMIT 1];
        // break if no merchandise is found
        System.assertNotEquals(null, m);
        // Add a new invoice
        Invoice_Statement__c i = new Invoice_Statement__c(
            Description__c = pDesc);
        insert i;

        // Add a new line item to the invoice
        Line_Item__c li = new Line_Item__c(
            Name = '1',
            Invoice_Statement__c = i.Id,
            Merchandise__c = m.Id,
            Unit_Price__c = m.Price__c,
            Units_Sold__c = pSold);
        insert li;

        // Update the inventory of the merchandise item
        m.Total_Inventory__c -= pSold;
        update m;
        return i.Id;
    }
}
```

3. In the Developer Console, execute the static `invoice` method.

```
Id invoice = MerchandiseOperations.invoice('Pencils', 5000, 'test 1');
```

This snippet causes the validation rule on the line item to fail on the line item insertion and a `DmlException` is returned. All DML operations are rolled back—the invoice statement and line item aren't committed to the database.

4. Let's find the validation rule error message and the exception in the execution log. Type `VF_PAGE_MESSAGE` next to **Filter**.

The validation rule error message displays in the filtered view (You have ordered more items than we have in stock.)

5. Next, type `exception` in the filter field and inspect the exception.

6. Delete the previous snippet and execute this second chunk of code.

```
Id invoice = MerchandiseOperations.invoice('Pencils', 5, 'test 2');
```

This snippet inserts a new invoice statement with a line item and commits them to the database. The validation rule succeeds because the number of pencils purchased is within the total inventory count.

Adding Custom Business Logic Using Triggers

Triggers are Apex code that execute before or after an insert, update, delete or undelete event occurs on an sObject. Think of them as classes with a particular syntax that lets you specify when they should run, depending on how a database record is modified.

The syntax that introduces a trigger definition is very different to that of a class or interface. A trigger always starts with the trigger keyword, followed by the name of the trigger, the database object to which the trigger should be attached to, and then the conditions under which it should fire, for example, before a new record of that database object is inserted. Triggers have the following syntax:

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

You can specify multiple trigger events in a comma-separated list if you want the trigger to execute before or after insert, update, delete, and undelete operations. The events you can specify are:

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

Creating a Trigger

Prerequisites:

- [Creating Warehouse Custom Objects](#)
- [Using the Developer Console](#)
- [sObjects and the Database](#)

The trigger that you'll create in this lesson fires before the deletion of invoice statements. It prevents the deletion of invoice statements if they contain line items.

1. In the Developer Console, click **File > New > Apex Trigger**.
2. Enter *RestrictInvoiceDeletion* for the name, select *Invoice_Statement__c* from the **sObject** drop-down list, and then click **Submit**.
3. Delete the auto-generated code and add the following.

```
trigger RestrictInvoiceDeletion on Invoice_Statement__c (before delete) {
    // With each of the invoice statements targeted by the trigger
    //   and that have line items, add an error to prevent them
    //   from being deleted.
    for (Invoice_Statement__c invoice :
        [SELECT Id
         FROM Invoice_Statement__c
         WHERE Id IN (SELECT Invoice_Statement__c FROM Line_Item__c) AND
         Id IN :Trigger.old]){
        Trigger.oldMap.get(invoice.Id).addError(
            'Cannot delete invoice statement with line items');
    }
}
```

4. Click **File > Save**.
Once you save the trigger, it is active by default.

Tell Me More...

- The trigger is called *RestrictInvoiceDeletion* and is associated with the *Invoice_Statement__c* sObject.
- The trigger fires before one or more *Invoice_Statement__c* sObjects are deleted. This behavior is specified by the *before delete* parameter.
- The trigger contains a SOQL for loop that iterates over the invoice statements with line items.
- Look at the nested query in the first condition in the *WHERE* clause: *(SELECT Invoice_Statement__c FROM Line_Item__c)*. Each line item has an *Invoice_Statement__c* field that references the parent invoice statement. The nested query retrieves the parent invoice statements of all line items.
- The query checks whether the *Id* values of the invoice statements are part of the set of parent invoice statements that are returned by the nested query: *WHERE Id IN (SELECT Invoice_Statement__c FROM Line_Item__c)*
- The second condition restricts the set of invoice statements queried to the ones that this trigger targets. This is done by checking whether each *Id* value is contained in *Trigger.old*. *Trigger.old* contains the set of old records that are to be deleted but haven't been deleted yet.
- For each invoice statement that meets the conditions, the trigger adds a custom error message by using the *addError* method, which prevents the deletion of the record. This custom error message appears in the user interface when you attempt to delete an invoice statement with line items, as you'll see in the next lesson.

Invoking the Trigger

Prerequisites:

- [Creating Sample Data](#)

In this lesson, you'll invoke the trigger you created in the previous lesson. The trigger fires before the deletion of invoice statements, so you can cause it to fire by deleting an invoice statement either through the user interface or programmatically. In this lesson, you'll perform the deletion through the user interface so you can see the error message returned by the trigger when the invoice statement

has line items. You'll also create an invoice statement with no line items. You'll be able to delete this new invoice statement since the trigger doesn't prevent the deletion of invoice statements without line items.

1. In the Salesforce user interface, click the + tab.
2. Click **Invoice Statements**.
3. With the **View** drop-down list selected to **All**, click **Go!**.
4. Click the sample invoice statement, with a name like **INV-0000**.
5. On the invoice statement's detail page, click **Delete**.
6. Click **OK** when asked for confirmation.

A new page displays with the following error message:

Validation Errors While Saving Record(s)

There were custom validation error(s) encountered while saving the affected record(s). The first validation error encountered was "Cannot delete invoice statement with line items".

Click [here](#) to return to the previous page.

7. Click the link to go back to the invoice statement's page.
8. Click **Back to List: Invoice Statements**.
9. You're now going to create another invoice statement that doesn't contain any line items. Click **New Invoice Statement**.
10. Click **Save**.
11. Let's try to delete this invoice statement. To do so, click **Delete**.
12. Click **OK** when asked for confirmation.

This time the invoice statement gets deleted. When the trigger is invoked, the trigger query only selects the invoice statements that have line items and prevents those records from deletion by marking them with an error. Since this invoice statement doesn't have any line items, it is not part of those records that the trigger marks with an error and the deletion is allowed.

Tell Me More...


- The validation error message that appears when deleting the sample invoice statement is the error message specified in the trigger using the `addError` method on the invoice statement sObject: 'Cannot delete invoice statement with line items.'
- The trigger universally enforces the business rule, no matter where operations come from: a user, another program, or a bulk operation. This lesson performed the deletion manually through the user interface.

Summary

In this tutorial, you exercised the trigger by attempting to delete two invoice statements. You saw how the trigger prevented the deletion of an invoice statement with a line item, and you were able to view the error message in the user interface. In the next tutorial, [Apex Unit Tests](#), you'll cause the trigger to be invoked programmatically. You'll add test methods that attempt to delete invoice statements with and without line items.

Apex Unit Tests

Writing unit tests for your code is fundamental to developing Apex code. You must have 75% test coverage to be able to deploy your Apex code to your production organization. In addition, the tests counted as part of the test coverage must pass. Testing is key to ensuring the quality of your application. Furthermore, having a set of tests that you can rerun in the future if you have to make changes to your code allows you to catch any potential regressions to the existing code.

 **Note:** This test coverage requirement also applies for creating a package of your application and publishing it on the Force.com AppExchange. When performing service upgrades, Salesforce executes Apex unit tests of all organizations to ensure quality and that no existing behavior has been altered for customers.

Test Data Isolation and Transient Nature

By default, Apex test methods don't have access to pre-existing data in the organization. You must create your own test data for each test method. In this way, tests won't depend on organization data and won't fail because of missing data when the data it depends on no longer exists.

Test data isn't committed to the database and is rolled back when the test execution completes. This means that you don't have to delete the data that is created in the tests. When the test finishes execution, the data created during test execution won't be persisted in the organization and won't be available.

You can create test data either in your test method or you can write utility test classes containing methods for test data creation that can be called by other tests.

There are some objects that tests can still access in the organization. They're metadata objects and objects used to manage your organization, such as `User` or `Profile`.

Adding a Test Utility Class

Prerequisites:

- [Adding Custom Business Logic Using Triggers](#)

In this lesson, you'll add tests to test the trigger that you created in the previous tutorial. Because you need to create some test data, you'll add a test utility class that contains methods for test data creation that can be called from any other test class or test method.

1. In the Developer Console, click **File** > **New** > **Apex Class**.
2. For the class name, enter `TestDataFactory` and click **OK**.
3. Delete the auto-generated code and add the following.

```
@isTest
public class TestDataFactory {

    public static Invoice_Statement__c createOneInvoiceStatement(
                                                Boolean withLineItem) {

        // Create one invoice statement
        Invoice_Statement__c testInvoice = createInvoiceStatement();

        if (withLineItem == true) {
            // Create a merchandise item
            Merchandise__c m = createMerchandiseItem('Orange juice');
            // Create one line item and associate it with the invoice statement.

            AddLineItem(testInvoice, m);
        }

        return testInvoice;
    }

    // Helper methods
    //
```

```

private static Merchandise__c createMerchandiseItem(String merchName) {
    Merchandise__c m = new Merchandise__c(
        Name=merchName,
        Description__c='Fresh juice',
        Price__c=2,
        Total_Inventory__c=1000);
    insert m;
    return m;
}

private static Invoice_Statement__c createInvoiceStatement() {
    Invoice_Statement__c inv = new Invoice_Statement__c(
        Description__c='Test Invoice');
    insert inv;

    return inv;
}

private static Line_Item__c AddLineItem(Invoice_Statement__c inv,
                                       Merchandise__c m) {
    Line_Item__c lineItem = new Line_Item__c(
        Invoice_Statement__c = inv.Id,
        Merchandise__c = m.Id,
        Unit_Price__c = m.Price__c,
        Units_Sold__c = (Double)(10*Math.random()+1));

    insert lineItem;

    return lineItem;
}
}

```

4. Click **File > Save**.

Tell Me More...

- This class contains one public method called `createOneInvoiceStatement` that creates an invoice statement and a merchandise item to be used as test data in test methods in the next lesson. It takes a Boolean argument that indicates whether a line item is to be added to the invoice.
- It also contains three helper methods that are used by `createOneInvoiceStatement`. These methods are all private and are used only within this class.
- Even though any Apex class can contain public methods for test data creation, this common utility class is defined with the `@isTest` annotation. The added benefit of using this annotation is that the class won't count against the 3 MB organization code size limit. The public methods in this can only be called by test code.

Add Test Methods

You've added a utility class that's called by the test method to create some data used for testing. Now you're ready to create the class that contains the test methods. Follow this procedure to add a test class and test methods.

1. Click **File > New > Apex Class**.
2. For the class name, enter `TestInvoiceStatementDeletion` and click **OK**.

3. Delete the auto-generated code and add the following.

```

@isTest
private class TestInvoiceStatementDeletion {

    static testmethod void TestDeleteInvoiceWithLineItem() {
        // Create an invoice statement with a line item then try to delete it
        Invoice_Statement__c inv = TestDataFactory.createOneInvoiceStatement(true);
        Test.startTest();
        Database.DeleteResult result = Database.delete(inv, false);
        Test.stopTest();

        // Verify invoice with a line item didn't get deleted.
        System.assert(!result.isSuccess());
    }

    static testmethod void TestDeleteInvoiceWithoutLineItems() {
        // Create an invoice statement without a line item and try to delete it
        Invoice_Statement__c inv = TestDataFactory.createOneInvoiceStatement(false);
        Test.startTest();
        Database.DeleteResult result = Database.delete(inv, false);
        Test.stopTest();

        // Verify invoice without line items got deleted.
        System.assert(result.isSuccess());
    }

    static testmethod void TestBulkDeleteInvoices() {
        // Create two invoice statements, one with and one with out line items
        // Then try to delete them both in a bulk operation, as might happen
        // when a trigger fires.
        List<Invoice_Statement__c> invList = new List<Invoice_Statement__c>();
        invList.add(TestDataFactory.createOneInvoiceStatement(true));
        invList.add(TestDataFactory.createOneInvoiceStatement(false));
        Test.startTest();
        Database.DeleteResult[] results = Database.delete(invList, false);
        Test.stopTest();

        // Verify the invoice with the line item didn't get deleted
        System.assert(!results[0].isSuccess());

        // Verity the invoice without line items did get deleted.
        System.assert(results[1].isSuccess());
    }
}

```

4. Click **File > Save**.**Tell Me More...**

- The class is defined with the `@isTest` annotation. You saw this annotation in the previous lesson to define a common test utility class. In this lesson, this annotation is used to mark the class as a test class to contain test methods that Apex can execute.
- The class contains 3 test methods. Test methods are static, top-level methods that take no arguments. They're defined with the `testmethod` keyword or the `@isTest` annotation. Both of these forms are valid declarations:

Declaration of a test method using the `testmethod` keyword:

```
static testmethod void myTest() {
    // Add test logic
}
```

Declaration of a test method using the `@isTest` annotation:

```
static @isTest void myTest() {
    // Add test logic
}
```

- Here is a description of each test method in this class:
 - `TestDeleteInvoiceWithLineItem`: This test method verifies that the trigger does what it is supposed to do—namely it prevents an invoice statement with line items from being deleted. It creates an invoice statement with a line item using the test factory method and deletes the invoice using the `Database.delete` Apex method. This method returns a `Database.DeleteResult` object that you can use to determine if the operation was successful and get the list of errors. The test calls the `isSuccess` method to verify that it is `false` since the invoice shouldn't have been deleted.
 - `TestDeleteInvoiceWithoutLineItems`: This test method verifies that the trigger doesn't prevent the deletion of invoice statements that don't have line items. It inserts an invoice statement without any line items and then deletes the invoice statement. Like the previous method, it calls the test factory method to create the test data and then calls `Database.delete` for the delete operation. The test verifies that the `isSuccess` method of `Database.DeleteResult` returns `true`.
 - `TestBulkDeleteInvoices`: This third test method performs a bulk delete on a list of invoices. It creates a list with two invoice statements, the first of which has one line item and the second doesn't have any. It calls `Database.delete` by passing a list of invoice statements to delete. Notice that this time we have a second parameter. It is an optional Boolean parameter that indicates whether the delete operation is rolled back on all sObjects if the delete operation fails on some sObjects. We passed the value `false`, which means the delete DML operation isn't rolled back on partial success. Also, the sObjects that don't cause errors are deleted. In this case, the test calls the `isSuccess` method of `Database.DeleteResult` to verify that the first invoice statement isn't deleted but the second one is.
- Each test method executes the delete DML operation within `Test.startTest/Test.stopTest` blocks. Each test method can have only one such block. All code running within this block is assigned a new set of governor limits separate from the other code in the test. This rule ensures that other setup code in your test doesn't share limits and enables you to test the governor limits. Using `Test.startTest` and `Test.stopTest` isn't critical in our case since we're deleting one or two records at a time and won't hit any limits. However, it's a good practice. You can perform prerequisite setup and test data creation before calling `Test.startTest` and include verifications after `Test.stopTest`. The original set of governor limits are reverted to after the call to `Test.stopTest`.

Run Tests and Code Coverage

Now that you've added test methods, the next step is to run them and inspect their results. In addition to ensuring that the tests pass, you can find out how much code was covered by your tests.

Let's run the tests in the `TestInvoiceStatementDeletion` class by using the Developer Console.

1. In the Developer Console, click **Test > Always Run Asynchronously**.

If you don't select Always Run Asynchronously, test runs that include tests from only one class run synchronously. Test runs that include more than one class run asynchronously even if you don't select this option.

2. Click **Test > New Run**.
3. In the Test Classes column, click **TestInvoiceStatementDeletion**.

- To add all test methods in the `TestInvoiceStatementDeletion` class to the test run, click **Add Selected**.

- Click **Run**.

All test methods in the test `TestInvoiceStatementDeletion` class run asynchronously, which means that the user interface in the Developer Console isn't blocked and waiting for the execution of those tests. You can continue using the Developer Console for other tasks and visit the **Tests** tab later to check on the test results. The test framework executes the tests in the background and updates the Developer Console **Tests** tab with the results after the test run finishes.

- After the tests have finished execution, click the **Tests** tab.

The **Tests** tab displays the test run that you've just executed. The test run is represented by an ID, which identifies each submission of tests. A green check mark is displayed next to our test run, which indicates that all tests in our test run succeeded. If you expand the test run node, the next level is the name of the test class. A test run can contain one or more test classes. In this case, our test run contains only one test class. If you expand the test class, all the test methods are listed for that class. The green check marks next to each test method indicate that all test methods succeeded.

| Status | Test Run | Enqueued Time | Duration | Failures | Total |
|--------|-----------------------------------|---------------------------------|----------|----------|-------|
| ✓ | 707c000000ks9O9 | Thu Jan 22 2015 10:23:14 GMT... | | 0 | 3 |
| ✓ | TestInvoiceStatementDeletion | | | 0 | 3 |
| ✓ | TestBulkDeleteInvoices | | 0:01 | | |
| ✓ | TestDeleteInvoiceWithLineItem | | 0:00 | | |
| ✓ | TestDeleteInvoiceWithoutLineItems | | 0:00 | | |

Each time you run tests, the code coverage for the classes and triggers that the tests exercised is updated. The updated code coverage results appear in the Tests tab in the Overall Code Coverage panel. This image shows the code coverage for the `RestrictInvoiceDeletion` trigger.

| Overall Code Coverage | | |
|-------------------------|-------------|-------|
| Class | Percent | Lines |
| Overall | 100% | |
| RestrictInvoiceDeletion | 100% | 2/2 |



Note: If you've created classes from other tutorials, more classes might be listed in the Overall Code Coverage panel.

- In the *Overall Code Coverage* panel, double-click the trigger name, **RestrictInvoiceDeletion**, to view the covered lines. The trigger is opened in the Developer Console code editor. The covered lines are highlighted in blue. Because the trigger has 100% coverage, no uncovered lines appear. Uncovered lines are highlighted in red when present.

Summary

In this tutorial, you learned the syntax of test classes and test methods, and the advantage of using a test class for your test methods annotated with `@isTest`. You created a test data factory class to create test data. You ran all tests and verified test results and code coverage. Last but not least, you learned the importance of having at least 75% test coverage as a requirement for deploying Apex to another organization.

Running Apex Within Governor Execution Limits

Prerequisites:

- [sObjects and the Database](#)

Unlike traditional software development, developing software in a multitenant cloud environment, the Force.com platform, relieves you from having to scale your code because the Force.com platform does it for you. Because resources are shared in a multitenant platform, the Apex runtime engine enforces a set of governor execution limits to ensure that no one transaction monopolizes shared resources. Your Apex code must execute within these predefined execution limits. If a governor limit is exceeded, a run-time exception that can't be handled is thrown. By following best practices in your code, you can avoid hitting these limits. Imagine you had to wash 100 t-shirts. Would you wash them one by one—one per load of laundry, or would you group them in batches for just a few loads? The benefit of coding in the cloud is that you learn how to write more efficient code and waste fewer resources.

The governor execution limits are per transaction. For example, one transaction can issue up to 100 SOQL queries and up to 150 DML statements. There are some other limits that aren't transaction bound, such as the number of batch jobs that can be queued or active at one time.

The following are some best practices for writing code that doesn't exceed certain governor limits.

Bulkifying DML Calls

Making DML calls on lists of sObjects instead of each individual sObject makes it less likely to reach the DML statements limit. The following is an example that doesn't bulkify DML operations, and the next example shows the recommended way of calling DML statements.

Example: DML calls on single sObjects

The for loop iterates over line items contained in the `liList` List variable. For each line item, it sets a new value for the `Description__c` field and then updates the line item. If the list contains more than 150 items, the 151st update call returns a run-time exception for exceeding the DML statement limit of 150. How do we fix this? Check the second example for a simple solution.

```
for(Line_Item__c li : liList) {
    if (li.Units_Sold__c > 10) {
        li.Description__c = 'New description';
    }
    // Not a good practice since governor limits might be hit.
    update li;
}
```

Recommended Alternative: DML calls on sObject lists

This enhanced version of the DML call performs the update on an entire list that contains the updated line items. It starts by creating a new list and then, inside the loop, adds every update line item to the new list. It then performs a bulk update on the new list.

```
List<Line_Item__c> updatedList = new List<Line_Item__c>();

for(Line_Item__c li : liList) {
    if (li.Units_Sold__c > 10) {
        li.Description__c = 'New description';
        updatedList.add(li);
    }
}

// Once DML call for the entire list of line items
update updatedList;
```

More Efficient SOQL Queries

Placing SOQL queries inside `for` loop blocks isn't a good practice because the SOQL query executes once for each iteration and may surpass the 100 SOQL queries limit per transaction. The following is an example that runs a SOQL query for every item in `Trigger.new`, which isn't efficient. An alternative example is given with a modified query that retrieves child items using only one SOQL query.

Example: Inefficient querying of child items

The `for` loop in this example iterates over all invoice statements that are in `Trigger.new`. The SOQL query performed inside the loop retrieves the child line items of each invoice statement. If more than 100 invoice statements were inserted or updated, and thus contained in `Trigger.new`, this results in a run-time exception because of reaching the SOQL limit. The second example solves this problem by creating another SOQL query that can be called only once.

```
trigger LimitExample on Invoice_Statement__c (before insert, before update) {
    for(Invoice_Statement__c inv : Trigger.new) {
        // This SOQL query executes once for each item in Trigger.new.
        // It gets the line items for each invoice statement.
        List<Line_Item__c> liList = [SELECT Id,Units_Sold__c,Merchandise__c
                                   FROM Line_Item__c
                                   WHERE Invoice_Statement__c = :inv.Id];

        for(Line_Item__c li : liList) {
            // Do something
        }
    }
}
```

Recommended Alternative: Querying of child items with one SOQL query

This example bypasses the problem of having the SOQL query called for each item. It has a modified SOQL query that retrieves all invoice statements that are part of `Trigger.new` and also gets their line items through the nested query. In this way, only one SOQL query is performed and we're still within our limits.

```
trigger EnhancedLimitExample on Invoice_Statement__c (before insert, before update) {
    // Perform SOQL query outside of the for loop.
    // This SOQL query runs once for all items in Trigger.new.
    List<Invoice_Statement__c> invoicesWithLineItems =
        [SELECT Id,Description__c,(SELECT Id,Units_Sold__c,Merchandise__c from Line_Items__r)

        FROM Invoice_Statement__c WHERE Id IN :Trigger.newMap.keySet()];

    for(Invoice_Statement__c inv : invoicesWithLineItems) {
        for(Line_Item__c li : inv.Line_Items__r) {
            // Do something
        }
    }
}
```

SOQL For Loops

Use SOQL for loops to operate on records in batches of 200. This helps avoid the heap size limit of 6 MB. Note that this limit is for code running synchronously and it is higher for asynchronous code execution.

Example: Query without a for loop

The following is an example of a SOQL query that retrieves all merchandise items and stores them in a List variable. If the returned merchandise items are large in size and a large number of them was returned, the heap size limit might be hit.

```
List<Merchandise__c> m1 = [SELECT Id,Name FROM Merchandise__c];
```

Recommended Alternative: Query within a for loop

To prevent this from happening, this second version uses a SOQL for loop, which iterates over the returned results in batches of 200 records. This reduces the size of the m1 list variable which now holds 200 items instead of all items in the query results, and gets recreated for every batch.

```
for (List<Merchandise__c> m1 : [SELECT Id,Name FROM Merchandise__c]) {
    // Do something.
}
```

For a complete list of Apex governor execution limits, see the [Force.com Apex Developer's Guide](#).

Scheduled Execution of Apex

The Apex Scheduler lets you have Apex classes run at specified times. This is ideal for daily or weekly maintenance tasks. To take advantage of the scheduler, you need to write an Apex class that implements the `Schedulable` interface, and then schedule it for execution on a specific schedule.

Adding a Class that Implements the Schedulable Interface

Prerequisites:

- [Creating Warehouse Custom Objects](#)
- [Using the Developer Console](#)
- [sObjects and the Database](#)

In this lesson, you'll write a class that implements the `Schedulable` interface, which means it can be scheduled to run at a specified date and time.

1. In the Developer Console, click **File > New > Apex Class**.
2. For the class name, enter *MySchedulableClass* and click **OK**.
3. Delete the auto-generated code and add the following.

```
global class MySchedulableClass implements Schedulable {
    global void execute(SchedulableContext ctx) {
        CronTrigger ct = [SELECT Id, CronExpression, TimesTriggered, NextFireTime
                           FROM CronTrigger WHERE Id = :ctx.getTriggerId()];

        System.debug(ct.CronExpression);
        System.debug(ct.TimesTriggered);

        Merchandise__c m = new Merchandise__c(
            Name='Scheduled Job Item',
            Description__c='Created by the scheduler',
            Price__c=1,
            Total_Inventory__c=1000);

        insert m;
```



```
}
}
```

4. Click **File** > **Save**.

Tell Me More...

- The declaration of the class contains an extra ***implements Schedulable*** at the end. This indicates that the class implements the `Schedulable` interface and must implement the only method that this interface contains, which is this `execute` method:

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

The parameter of this method is a `SchedulableContext` object. It provides the `getTriggerId` method that returns the ID of the `CronTrigger` API object. After a class has been scheduled, a `CronTrigger` object is created that represents the scheduled job.

- The `CronTrigger` object is queried to get additional information about the scheduled job. The Cron expression and the number of times the job has been run already is written to the debug log.
- Finally, the `execute` method creates a merchandise record.

Adding a Test for the Schedulable Class

Prerequisites:

- [Apex Unit Tests](#)

Now that you've added a schedulable class, you're ready to add a test method to ensure that your class has test coverage. In this lesson, you'll add a test class with one test method, which calls `System.Schedule` to schedule the class.

Although you can run tests from the Developer Console, you'll run tests from the class listed in the Apex Classes page. You'll be scheduling the class in the next step from this same page.

- From Setup, enter *Apex* in the *Quick Find* box, select **Apex Classes**, and then click **New**.
- In the code editor box, add the following test class.

```
@isTest
private class TestSchedulableClass {

    // CRON expression: midnight on March 15.
    // Because this is a test, job executes
    // immediately after Test.stopTest().
    public static String CRON_EXP = '0 0 0 15 3 ? 2022';

    static testmethod void test() {
        Test.startTest();

        // Schedule the test job
        String jobId = System.schedule('ScheduleApexClassTest',
                                      CRON_EXP,
                                      new MySchedulableClass());

        // Get the information from the CronTrigger API object
        CronTrigger ct = [SELECT Id, CronExpression, TimesTriggered,
                          NextFireTime
                          FROM CronTrigger WHERE id = :jobId];

        // Verify the expressions are the same
```

```

System.assertEquals(CRON_EXP,
    ct.CronExpression);

// Verify the job has not run
System.assertEquals(0, ct.TimesTriggered);

// Verify the next time the job will run
System.assertEquals('2022-03-15 00:00:00',
    String.valueOf(ct.NextFireTime));
// Verify the scheduled job hasn't run yet.
Merchandise__c[] ml = [SELECT Id FROM Merchandise__c
                        WHERE Name = 'Scheduled Job Item'];
System.assertEquals(ml.size(), 0);
Test.stopTest();

// Now that the scheduled job has executed after Test.stopTest(),
// fetch the new merchandise that got added.
ml = [SELECT Id FROM Merchandise__c
      WHERE Name = 'Scheduled Job Item'];
System.assertEquals(ml.size(), 1);
    }
}

```

3. Click **Save**.
4. Click **Run Test** to execute the test method.

Tell Me More...

- The test method schedules the `MySchedulableClass` class by calling the `System.schedule` method. 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. The `System.schedule` method uses the user's timezone for the basis of all schedules.
- The call to `System.schedule` is included within the `Test.startTest` and `Test.stopTest` block. This ensures that the job gets executed after the `Test.stopTest` call regardless of the schedule specified in the cron expression. Any asynchronous code included within `Test.startTest` and `Test.stopTest` gets executed synchronously after `Test.stopTest`.
- Finally, the test method verifies a new merchandise item got added by the scheduled class.



Tip:

- 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.
- You can only have 100 classes scheduled at one time.

Scheduling and Monitoring Scheduled Jobs

Now that you've seen how to create and test a schedulable class, let's take a look at how to schedule the class using the user interface. You'll also learn how to view the list of scheduled jobs in your organization.

1. Click **Apex Classes** to go back to the Apex Classes page.
2. Click **Schedule Apex**.
3. For the job name, enter `TestSchedulingApexFromTheUI`.

4. Click the lookup button next to Apex class and enter `*` for the search term to get a list of all classes that can be scheduled. In the search results, click `MySchedulableClass`.
5. Select `Weekly` or `Monthly` for the frequency and set the frequency desired.
6. Select the start and end dates, and a preferred start time.
The schedule of a scheduled Apex job is relative to the user's time zone.
7. Click **Save**.
8. To go to the Schedule Jobs page, from Setup, enter `Scheduled Jobs` in the `Quick Find` box, then select **Scheduled Jobs**.
You'll see that your job is now listed in the job queue.
9. Click **Manage** next to the job's name.
The page displays more details about the job, including its execution schedule.

Summary

In this tutorial, you created a class that implements the `Schedulable` interface. You also added a test for it. Finally, you learned how to schedule the class to run at a specified time and how to view the scheduled job in the user interface.

Scheduled jobs can be quite handy if you want to run maintenance tasks on a periodic basis.

Apex Batch Processing

Using batch Apex classes, you can process records in batches asynchronously. If you have a large number of records to process, for example, for data cleansing or archiving, batch Apex is your solution. Each invocation of a batch class results in a job being placed on the Apex job queue for execution.

The execution logic of the batch class is called once for each batch of records. The default batch size is 200 records. You can also specify a custom batch size. Furthermore, each batch execution is considered a discrete transaction. With each new batch of records, a new set of governor limits is in effect. In this way, it's easier to ensure that your code stays within the governor execution limits. Another benefit of discrete batch transactions is to allow for partial processing of a batch of records in case one batch fails to process successfully, all other batch transactions aren't affected and aren't rolled back if they were processed successfully.

Batch Apex Syntax

To write a batch Apex class, your class must implement the `Database.Batchable` interface. Your class declaration must include the `implements` keyword followed by `Database.Batchable<SObject>`. Here is an example:

```
global class CleanUpRecords implements Database.Batchable<SObject> {
```

You must also implement three methods:

- `start` method

```
global (Database.QueryLocator | Iterable<SObject>) start(Database.BatchableContext bc)
{ }
```

The `start` method is called at the beginning of a batch Apex job. It collects the records or objects to be passed to the interface method `execute`.

- 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.

Batches of records are not guaranteed to execute in the order they are received from the `start` method.

- 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.

Invoking a Batch Class

To invoke a batch class, instantiate it first and then call `Database.executeBatch` with the instance of your batch class:

```
BatchClass myBatchObject = new BatchClass();
Database.executeBatch(myBatchObject);
```

In the next steps of this tutorial, you'll learn how to create a batch class, test it, and invoke a batch job.

Adding a Batch Apex Class

Prerequisites:

- [Using the Developer Console](#)

In this lesson, you'll create a batch Apex class that implements the `Database.Batchable` interface. The batch class cleans up the records that are passed in by the `start` method.

1. In the Developer Console, click **File > New > Apex Class**.
2. For the class name, enter `CleanUpRecords` and click **OK**.
3. Delete the auto-generated code and add the following.

```
global class CleanUpRecords implements
    Database.Batchable<sObject> {

    global final String query;

    global CleanUpRecords(String q) {
        query = q;
    }

    global Database.QueryLocator start(Database.BatchableContext BC) {
        return Database.getQueryLocator(query);
    }
}
```

```

global void execute(
    Database.BatchableContext BC,
    List<sObject> scope){
    delete scope;
    Database.emptyRecycleBin(scope);
}

global void finish(Database.BatchableContext BC){
    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('Record Clean Up Status: ' + a.Status);
    mail.setPlainTextBody
        ('The batch Apex job processed ' + a.TotalJobItems +
        ' batches with ' + a.NumberOfErrors + ' failures. ');
    Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
}
}

```

4. Click **File > Save**.

Tell Me More...

- The records provided to this batch class are based on a query that is specified by the `query` variable. This query variable is set in the constructor of this class.
- For each batch of records, the three methods in this class are executed starting with the `start` method, then the `execute` method, then the `finish` method, in this order.
- The `start` provides the batch of records that the `execute` method will process. It returns the list of records to be processed by calling `Database.getQueryLocator(query)`. The maximum number of records that can be returned in the `Database.QueryLocator` object is 50 million.
- The list of batch records to process is passed in the second parameter of the `execute` method. The `execute` method simply deletes the records with the `delete` DML statement. Since deleted records stay in the Recycle Bin for 15 days, the method also empties the Recycle Bin to delete these records permanently.
- When a batch Apex job is invoked, a new record is added in the `AsyncApexJob` table that has information about the batch job, such as its status, the number of batches processed, and the total number of batches to be processed. The `finish` method sends an email to the job's submitter to confirm the job completion. It performs a query on the `AsyncApexJob` object to get the status of the job, the submitter's email address, and other information. It then creates a new email message and sends it using the `Messaging.SingleEmailMessage` methods.

In the next , you'll add a test method that invokes this batch class.

Adding a Test for the Batch Apex Class

Prerequisites:

- [Creating Warehouse Custom Objects](#)
- [sObjects and the Database](#)
- [Apex Unit Tests](#)

In this lesson, you'll add a test class for the `CleanUpRecords` batch class. The test in this class invokes the batch job and verifies that it deletes all merchandise records that haven't been purchased.

1. Click **File > New > Apex Class**.
2. For the class name, enter `TestCleanUpBatchClass` and click **OK**.
3. Delete the auto-generated code and add the following.

```
@isTest
private class TestCleanUpBatchClass {

    static testmethod void test() {
        // The query used by the batch job.
        String query = 'SELECT Id, CreatedDate FROM Merchandise__c ' +
            'WHERE Id NOT IN (SELECT Merchandise__c FROM Line_Item__c)';

        // Create some test merchandise items to be deleted
        // by the batch job.
        Merchandise__c[] ml = new List<Merchandise__c>();
        for (Integer i=0; i<10; i++) {
            Merchandise__c m = new Merchandise__c(
                Name='Merchandise ' + i,
                Description__c='Some description',
                Price__c=2,
                Total_Inventory__c=100);
            ml.add(m);
        }
        insert ml;

        Test.startTest();
        CleanUpRecords c = new CleanUpRecords(query);
        Database.executeBatch(c);
        Test.stopTest();

        // Verify merchandise items got deleted
        Integer i = [SELECT COUNT() FROM Merchandise__c];
        System.assertEquals(i, 0);
    }
}
```

4. Click **File > Save**.

Tell Me More...

- The test class contains one test method called `test`. This test method starts by constructing the query string that is to be passed to the constructor of `CleanUpRecords`. Since a merchandise item that hasn't been purchased is a merchandise item that doesn't have line items associated with it, the SOQL query specifies the following:

```
WHERE Id NOT IN (SELECT Merchandise__c FROM Line_Item__c)
```

The subquery

```
SELECT Merchandise__c FROM Line_Item__c
```

gets the set of all merchandise items that are referenced in line items. Since the query uses the NOT IN operator in the WHERE clause, this means the merchandise items that aren't referenced in line items are returned.

- The test method inserts 10 merchandise items with no associated line items to be cleaned up by the batch class method. Note that the number of records inserted is less than the batch size of 200 because test methods can execute only one batch total.
- Next, the batch class is instantiated with the query with this statement where the `query` variable is passed to the constructor of `CleanUpRecords`:

```
CleanUpRecords c = new CleanUpRecords(query);
```

- The batch class is invoked by calling `Database.executeBatch` and passing it the instance of the batch class:

```
Database.executeBatch(c);
```

- The call to `Database.executeBatch` is included within the `Test.startTest` and `Test.stopTest` block. This is necessary for the batch job to run in a test method. The job executes after the call to `Test.stopTest`. Any asynchronous code included within `Test.startTest` and `Test.stopTest` gets executed synchronously after `Test.stopTest`.
- Finally, the test verifies that all test merchandise items created in this test got deleted by checking that the count of merchandise items is zero.
- Even though the batch class `finish` method sends a status email message, the email isn't sent in this case because email messages don't get sent from test methods.

Running a Batch Job

You can invoke a batch class from a trigger, a class, or the Developer Console. There are times when you want to run the batch job at a specified schedule. This shows you how to submit the batch class through the Developer Console for immediate results. You'll also create a scheduler class that enables you to schedule the batch class.

Begin by setting up some merchandise records in the organization that don't have any associated line items. The records that the test created in the previous don't persist, so you'll create some new records to ensure the batch job has some records to process.

1. Click the *Logs* tab, and then run the following in the Execute window:

```
Merchandise__c[] m1 = new List<Merchandise__c>();
for (Integer i=0;i<250;i++) {
    Merchandise__c m = new Merchandise__c(
        Name='Merchandise ' + i,
        Description__c='Some description',
        Price__c=2,
        Total_Inventory__c=100);
    m1.add(m);
}
insert m1;
```

2. Click **Execute**.

This creates 250 merchandise items, which ensures that our batch class runs twice, once for the first 200 records, and once for the remaining 50 records.

- Let's now submit the batch class by calling `Database.executeBatch` from the Developer Console. Run the following in the Execute window:

```
String query = 'SELECT Id,CreateDate FROM Merchandise__c ' +
    'WHERE Id NOT IN (SELECT Merchandise__c FROM Line_Item__c)';
CleanUpRecords c = new CleanUpRecords(query);
Database.executeBatch(c);
```

You'll receive an email notification for the job's completion. It might take a few minutes for the email to arrive. The email should state that two batches were run.

- To view the status of the batch job execution, from Setup, enter *Apex Jobs* in the *Quick Find* box, then select **Apex Jobs**. Because the job finished, its status shows as completed, and you can see that two batches were processed.

View: All ▾ [Create New View](#)

| Action | Submitted Date ↓ | Job Type | Status | Status Detail | Total Batches | Batches Processed | Failures | Submitted By |
|--------|--------------------|------------|-----------|---------------|---------------|-------------------|----------|---------------------------|
| | 2/10/2012 12:51 PM | Batch Apex | Completed | | 2 | 2 | 0 | User Test |

- To schedule the batch job programmatically, you need to create a class that implements the `Schedulable` interface which invokes the batch class from its `execute` method. First, from Setup, enter *Apex* in the *Quick Find* box, select **Apex Classes**, and then click **New**.
- In the code editor box, add the following class definition.

```
global class MyScheduler implements Schedulable {

    global void execute(SchedulableContext ctx) {
        // The query used by the batch job.
        String query = 'SELECT Id,CreateDate FROM Merchandise__c ' +
            'WHERE Id NOT IN (SELECT Merchandise__c FROM Line_Item__c)';

        CleanUpRecords c = new CleanUpRecords(query);
        Database.executeBatch(c);
    }
}
```

- Follow steps similar to the ones [Scheduling and Monitoring Scheduled Jobs](#) to schedule the `MyScheduler` class.

Summary

In this tutorial, you created a batch Apex class for data cleanup. You then tested the batch class by writing and running a test method. You also learned how to schedule the batch class.

Batch Apex allows to process records in batches and is useful when you have a large number of records to process.

Apex REST

You can create custom REST Web service APIs on top of the Force.com platform or Database.com by exposing your Apex classes as REST resources. Client applications can call the methods of your Apex classes using REST to run Apex code in the platform.

Apex REST supports both XML and JSON for resource formats sent in REST request and responses. By default, Apex REST uses JSON to represent resources.

For authentication, Apex REST supports OAuth 2.0 and the Salesforce session. This tutorial uses Workbench to simulate a REST client. Workbench uses the session of the logged-in user as an authentication mechanism for calling Apex REST methods.



Note: Workbench is a free, open source, community-supported tool (see the Help page in Workbench). Salesforce provides a hosted instance of Workbench for demonstration purposes only—Salesforce recommends that you do not use this hosted instance of Workbench to access data in a production database. If you want to use Workbench for your production database, you can download, host, and configure it using your own resources.

Add a Class as a REST Resource

Prerequisites:

- [Creating Warehouse Custom Objects](#)
- [Using the Developer Console](#)
- [sObjects and the Database](#)

Let's add a class with 2 methods and expose it through Apex REST.

1. In the Developer Console, click **File > New > Apex Class**.
2. For the class name, enter *MerchandiseManager* and click **OK**.
3. Delete the auto-generated code and add the following.

```
@RestResource(urlMapping='/Merchandise/*')
global with sharing class MerchandiseManager {

    @HttpGet
    global static Merchandise__c getMerchandiseById() {
        RestRequest req = RestContext.request;
        String merchId = req.requestURI.substring(
                                req.requestURI.lastIndexOf('/')+1);

        Merchandise__c result =
            [SELECT Name,Description__c,Price__c,Total_Inventory__c
            FROM Merchandise__c
            WHERE Id = :merchId];

        return result;
    }

    @HttpPost
    global static String createMerchandise(String name,
        String description, Decimal price, Double inventory) {
        Merchandise__c m = new Merchandise__c(
            Name=name,
            Description__c=description,
            Price__c=price,
            Total_Inventory__c=inventory);
```

```

        insert m;
        return m.Id;
    }
}

```

4. Click **File** > **Save**.

Tell Me More...

- The class is global and defined with the `@RestResource(urlMapping='/Merchandise/*')` annotation. Any Apex class you want to expose as a REST API must be global and annotated with the `@RestResource` annotation. The parameter of the `@RestResource` annotation, `urlMapping`, is used to uniquely identify your resource and is relative to the base URL `https://instance.salesforce.com/services/apexrest/`. The base URL and the `urlMapping` value form the URI that the client sends in a REST request. In this case, the URL mapping contains the asterisk wildcard character, which means that the resource URI can contain any value after `/Merchandise/`. In Step 3 of this tutorial, we'll be appending an ID value to the URI for the record to retrieve.
- The class contains 2 global static methods defined with Apex REST annotations. All Apex REST methods must be global static.
- The first class method, `getMerchandiseById`, is defined with the `@HttpGet` annotation.
 - The `@HttpGet` annotation exposes the method as a REST API that is called when an HTTP `GET` request is sent from the client.
 - This method returns the merchandise item that corresponds to the ID sent by the client in the request URI.
 - It obtains the request and request URI through the Apex static `RestContext` class.
 - It then parses the URI to find the value passed in after the last `/` character and performs a SOQL query to retrieve the merchandise record with this ID. Finally, it returns this record.
- The second class method, `createMerchandise`, is defined with the `@HttpPost` annotation. This annotation exposes the method as a REST API and is called when an HTTP `POST` request is sent from the client. This method creates a merchandise record using the specified data sent by the client. It calls the `insert` DML operation to insert the new record in the database and returns the ID of the new merchandise record to the client.

Creating a Record Using the Apex REST POST Method

In this lesson, you'll use REST Explorer in Workbench to send a REST client request to create a new merchandise record. This request invokes one of the Apex REST methods you've just implemented.

Workbench's REST Explorer simulates a REST client. It uses the session of the logged-in user as an authentication mechanism for calling Apex REST methods.

You might be able to skip the first few steps in this procedure if you already set up sample data with Workbench in a previous tutorial.

1. Navigate to: <https://developer.salesforce.com/page/Workbench>.
2. If prompted for your credentials, enter your login information and click **Login**.
3. For **Environment**, select **Production**.
4. Accept the terms of service and click **Login with Salesforce**.
5. Click **Allow** to allow Workbench to access your information.
6. After logging in, click **utilities** > **REST Explorer**.
7. Enter your login credentials and click **Log in to Salesforce**.
8. Click **POST**.

9. The URL path that REST explorer accepts is relative to the instance URL of your org, so you only have to provide the path that is appended to the instance URL. In the relative URL box, replace the default URL with `/services/apexrest/Merchandise/`
10. For the request body, insert the following JSON string representation of the object to insert:

```
{
  "name" : "Eraser",
  "description" : "White eraser",
  "price" : 0.75,
  "inventory" : 1000
}
```

Note that the field names for the object to create must match and must have the same case as the names of the parameters of the method that will be called.

11. Click **Execute**.

This causes the `createMerchandise` method to be called. The response contains the ID of the new merchandise record.

12. To obtain the ID value from the response, click **Show Raw Response**, and then copy the ID value, without the quotation marks, that is displayed at the bottom of the response. For example, `"a04R00000007xX1IAI"`, but your value will be different. You'll use this ID in the next lesson to retrieve the record you've just inserted.

Retrieving a Record Using the Apex REST GET Method

In this lesson, you'll use Workbench to send a REST client request to retrieve the new merchandise record you've just created in the previous lesson. This request invokes one of the Apex REST methods you've just implemented.

1. In the REST Explorer, Click **GET**.

2. In the relative URL box, append the ID of the record you copied from Lesson 2 of this tutorial to the end of the URL: `/services/apexrest/Merchandise/`.

3. Click **Execute**.

This causes the `getMerchandiseById` method to be called. The response returned contains the fields of the new merchandise record.

Choose an HTTP method to perform on the REST API service URI below:

☒ GET ☐ POST ☐ PUT ☐ PATCH ☐ DELETE ☐ HEAD

/services/apexrest/Merchandise/a04R00000007xX1IAI

[Expand All](#) | [Collapse All](#) | [Show Raw Response](#)

attributes

- Name: **Eraser**
- Total_Inventory__c: **1000**
- Id: **a04R00000007xX1IAI**
- Price__c: **0.75**
- Description__c: **White eraser**

4. Optionally, click **Show Raw Response** to view the entire response, including the HTTP headers and the response body in JSON format.

Raw Response

```
HTTP/1.1 200 OK
Server:
Content-Encoding: gzip
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 08 Feb 2012 05:13:50 GMT

{
  "attributes" : {
    "type" : "Merchandise__c",
    "url" : "/services/data/v24.0/objects
/Merchandise__c/a04R000000007xX1IAI"
  },
  "Name" : "Eraser",
  "Total_Inventory__c" : 1000,
  "Id" : "a04R000000007xX1IAI",
  "Price__c" : 0.75,
  "Description__c" : "White eraser"
}
```

Summary

In this tutorial, you created a custom REST-based API by writing an Apex class and exposing it as a REST resource. The two methods in the class are called when HTTP `GET` and `POST` requests are received. You also used the methods that you implemented using the REST Explorer in Workbench and saw the raw JSON response.

Visualforce Pages with Apex Controllers

Visualforce is a component-based user interface framework for the Force.com platform. Visualforce allows you to build sophisticated user interfaces by providing a view framework that includes a tag-based markup language similar to HTML, a library of reusable components that can be extended, and an Apex-based controller model. Visualforce supports the Model-View-Controller (MVC) style of user interface design, and is highly flexible.

Visualforce includes *standard controllers* for every sObject available in your organization, which lets you create Visualforce pages that handle common features without writing any code beyond the Visualforce itself. For highly customized applications, Visualforce allows you to extend or replace the standard controller with your own Apex code. You can make Visualforce applications available only within your company, or publish them on the Web.

In this tutorial, you will use Visualforce to create a simple store front page. You'll start with a simple product listing page that does not use Apex as a quick introduction to Visualforce. Then you'll add a few features, like a simple shopping cart, to see how Visualforce connects to a controller written in Apex.

Enabling Visualforce Development Mode

The simplest way to get started with Visualforce is to enable development mode. Development mode embeds a Visualforce page editor in your browser. It allows you to see and edit your code, and preview the page at the same time. Development mode also adds an Apex editor for editing controllers and extensions.

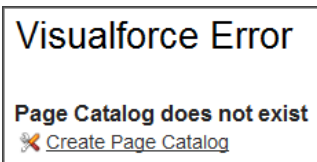
1. From your personal settings, enter *Advanced User Details* in the Quick Find box, then select **Advanced User Details**.
No results? Enter *Personal Information* in the Quick Find box, then select **Personal Information**.
2. Click **Edit**.
3. Select the **Development Mode** checkbox.
4. Click **Save**.

After enabling development mode, all Visualforce pages display with the development mode footer at the bottom of the browser window.

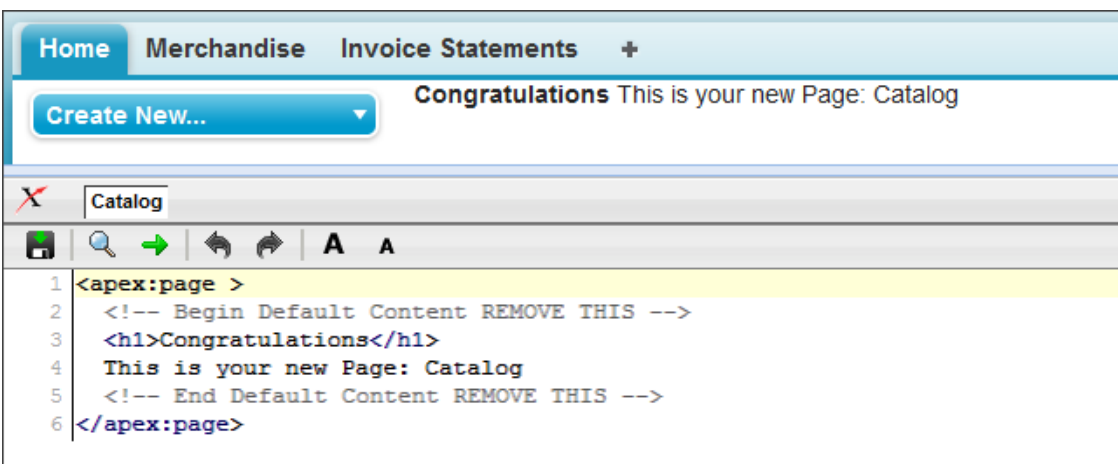
Creating a Simple Visualforce Page

In this lesson you'll create a new, very simple Visualforce page, the equivalent of "Hello World."

1. In your browser, add the text `/apex/Catalog` to the URL for your Salesforce instance. For example, if your Salesforce instance is `https://na1.salesforce.com`, the new URL would be `https://na1.salesforce.com/apex/Catalog`.
You'll get an error message: Page Catalog does not exist.



2. Click the **Create Page Catalog** link to create the new page.
The Catalog page will be created with some default code.
3. The Page Editor displays a preview of the new page above and the code below. It will look like this:



If the Page Editor is collapsed, click the **Expand** (🖥️) button at the bottom right of your browser window.


4. You don't really want the heading of the page to say "Congratulations," so change the contents of the `<h1>` tag to Product Catalog, and remove the comments and other plain text. The code for the page will now look something like this.

```
<apex:page>

    <h1>Product Catalog</h1>

</apex:page>
```

You can add additional text and HTML between the tags, but Visualforce pages must begin with `<apex:page>` and end with `</apex:page>`.

5. Click the **Save** button () at the top of the Page Editor.
The page reloads to reflect your changes.

Notice that the code for the page looks a lot like standard HTML. That's because Visualforce pages combine HTML tags, such as `<h1>`, with Visualforce-specific tags, which start with `<apex:>`.

Displaying Product Data in a Visualforce Page

Prerequisites:

- [Creating Warehouse Custom Objects](#)
- [Creating Sample Data](#)

In this lesson, you'll extend your first Visualforce page to display a list of products for sale. Although this page might seem fairly simple, there's a lot going on, and we're going to move quickly so we can get to the Apex. If you'd like a more complete introduction to Visualforce, see the [Visualforce Workbook](#).

1. In your browser, open your product catalog page at `https://<your-instance>.salesforce.com/apex/Catalog`, and open the Page Editor, if it's not already open.
2. Modify your code to enable the `Merchandise__c` standard controller, by editing the `<apex:page>` tag.

```
<apex:page standardController="Merchandise__c">
```

This connects your page to your `Merchandise__c` custom object on the platform, using a built-in controller that provides a lot of basic functionality, like reading, writing, and creating new `Merchandise__c` objects.

3. Next, add the standard list controller definition.

```
<apex:page standardController="Merchandise__c" recordSetVar="products">
```

This configures your controller to work with lists of `Merchandise__c` records all at once, for example, to display a list of products in your catalog. Exactly what we want to do!

4. Click **Save**. You can also press CTRL+S, if you prefer to use the keyboard.
The page reloads, and if the *Merchandise* tab is visible, it becomes selected. Otherwise you won't notice any change on the page. However, because you've set the page to use a controller, and defined the variable `products`, the variable will be available to you in the body of the page, and it will represent a list of `Merchandise__c` records.
5. Replace any code between the two `<apex:page>` tags with a page block that will soon hold the products list.

```
<apex:pageBlock title="Our Products">

    <apex:pageBlockSection>
```

```

        (Products Go Here)

    </apex:pageBlockSection>

</apex:pageBlock>

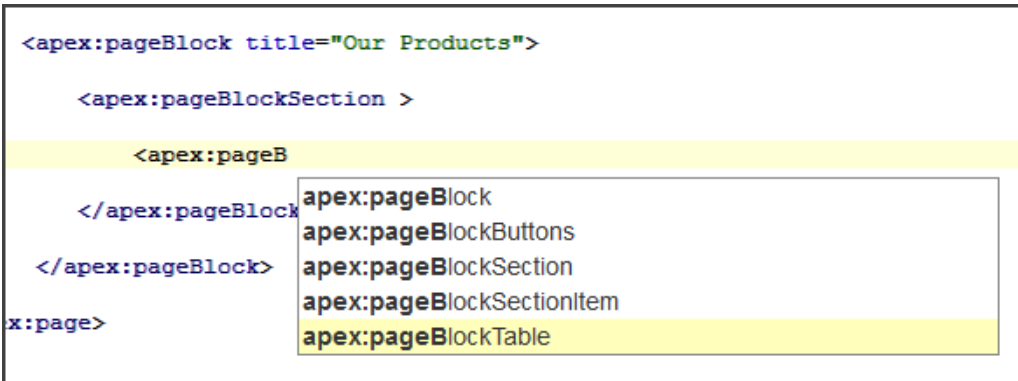
```

The `pageBlock` and `pageBlockSection` tags create some user interface elements on the page, which match the standard visual style of the platform.



Note: From here we'll assume that you'll save your changes whenever you want to see how the latest code looks.

- It's time to add the actual list of products. Select the (Products Go Here) placeholder and delete it. Start typing `<apex:pageB` and use your mouse or arrow keys to select `apex:pageBlockTable` from the drop-down list, and press RETURN.



Notice that the editor inserts both opening and closing tags, leaving your insertion point in the middle.

- Now you need to add some attributes to the `pageBlockTable` tag. The `value` attribute indicates which list of items the `pageBlockTable` component should iterate over. The `var` attribute assigns each item of that list, for one single iteration, to the `pitem` variable. Add these attributes to the tag.

```
<apex:pageBlockTable value="{!products}" var="pitem">
```

- Now you're going to define each column, and determine where it gets its data by looking up the appropriate field in the `pitem` variable. Add the following code between the opening and closing `pageBlockTable` tags.

```

<apex:pageBlockTable value="{!products}" var="pitem">
    <apex:column headerValue="Product">
        <apex:outputText value="{!pitem.Name}"/>
    </apex:column>
</apex:pageBlockTable>

```

- Click **Save** and you'll see your product list appear.

| Our Products | |
|--------------|--|
| Product | |
| Android | |
| Desktop | |
| Laptop | |
| MacBook Air | |
| MacBook Pro | |

The `headerValue` attribute has simply provided a header title for the column, and below it you'll see a list of rows, one for each merchandise record. The expression `{!pitem.Name}` indicates that we want to display the Name field of the current row.

10. Now, after the closing tag for the first column, add two more columns.

```
<apex:column headerValue="Description">
    <apex:outputField value="{!pitem.Description__c}"/>
</apex:column>
<apex:column headerValue="Price">
    <apex:outputField value="{!pitem.Price__c}"/>
</apex:column>
```

11. With three columns, the listing is compressed because the table is narrow. Make it wider by changing the `<apex:pageBlockSection>` tag.

```
<apex:pageBlockSection columns="1">
```

This changes the section from two columns to one, letting the single column be wider.

12. Your code will look something like this.

```
<apex:page standardController="Merchandise__c" recordSetVar="products">

    <apex:pageBlock title="Our Products">

        <apex:pageBlockSection columns="1">

            <apex:pageBlockTable value="{!products}" var="pitem">
                <apex:column headerValue="Product">
                    <apex:outputText value="{!pitem.Name}"/>
                </apex:column>
                <apex:column headerValue="Description">
                    <apex:outputField value="{!pitem.Description__c}"/>
                </apex:column>
                <apex:column headerValue="Price">
                    <apex:outputField value="{!pitem.Price__c}"/>
                </apex:column>
            </apex:pageBlockTable>

        </apex:pageBlockSection>

    </apex:pageBlock>

</apex:page>
```

And there you have your product catalog!

Tell Me More...

- The `pageBlockTable` component produces a table with rows, and each row is found by iterating over a list. The standard controller you used for this page was set to `Merchandise__c`, and the `recordSetVar` to `products`. As a result, the controller automatically populated the `products` list variable with merchandise records retrieved from the database. It's this list that the `pageBlockTable` component uses.
- You need a way to reference the current record as you iterate over the list. The statement `var="pitem"` assigns a variable called `pitem` that holds the record for the current row.

Using a Custom Apex Controller with a Visualforce Page

You now have a Visualforce page that displays all of your merchandise records. Instead of using the default controller, as you did in the previous tutorial, you're going to write the controller code yourself. Controllers typically retrieve the data to be displayed in a Visualforce page, and contain code that will be executed in response to page actions, such as a command button being clicked.

In this lesson, you'll convert the page from using a standard controller to using your own custom Apex controller. Writing a controller using Apex allows you to go beyond the basic behaviors provided by the standard controller. In the next lesson you'll expand this controller and add some e-commerce features to change the listing into an online store.

To create the new controller class:

1. From Setup, enter *Apex Classes* in the *Quick Find* box, then select **Apex Classes**.
2. Click **New**.
3. Add the following code as the definition of the class and then click **Quick Save**.

```
public class StoreFrontController {

    List<Merchandise__c> products;

    public List<Merchandise__c> getProducts() {
        if(products == null) {
            products = [SELECT Id, Name, Description__c, Price__c FROM Merchandise__c];
        }
        return products;
    }
}
```

4. Navigate back to your product catalog page at <https://<your-instance>.salesforce.com/apex/Catalog>, and open the Page Editor, if it's not already open.
5. Change the opening `<apex:page>` tag to link your page to your new controller class.

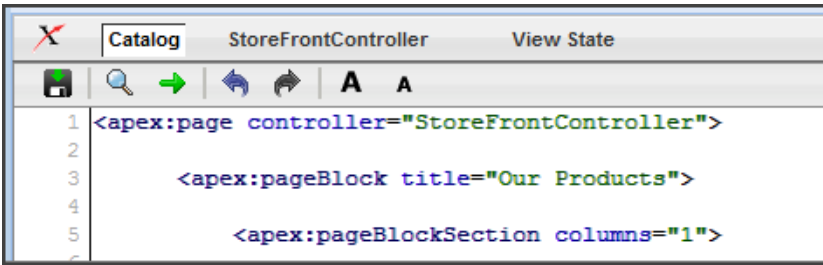
```
<apex:page controller="StoreFrontController">
```

Notice that the attribute name has changed from `standardController` to `controller`. You also remove the `recordSetVar` attribute, because it's only used with standard controllers.

6. Click **Save** to save your changes and reload the page.
The only change you should see is that the *Merchandise* tab is no longer selected.
7. Make the following addition to set the application tab style back to Merchandise.

```
<apex:page controller="StoreFrontController" tabStyle="Merchandise__c">
```

- Notice that above the Page Editor tool bar there is now a **StoreFrontController** button. Click it to view and edit your page's controller code. Click **Catalog** to return to the Visualforce page code.



You'll use this in the next lessons.

Tell Me More...

- As in the previous lesson, the value attribute of the `pageBlockTable` is set to `{!products}`, indicating that the table component should iterate over a list called `products`. Because you are using a custom controller, when Visualforce evaluates the `{!products}` expression, it automatically looks for a method `getProducts()` in your Apex controller.
- The `StoreFrontController` class does the bare minimum to provide the data required by the Visualforce catalog page. It contains that single method, `getProducts()`, which queries the database and returns a list of `Merchandise__c` records.
- The combination of a public instance variable (here, `products`) with a getter method (`getProducts()`) to initialize and provide access to it is a common pattern in Visualforce controllers written in Apex.

Using Inner Classes in an Apex Controller

In the last lesson, you created a custom controller for your Visualforce catalog page. But your controller passes custom objects from the database directly to the view, which isn't ideal. In this lesson, you'll refactor your controller to more correctly use the MVC design pattern, and add some additional features to your page.

- Click **StoreFrontController** to edit your page's controller code.
- Revise the definition of the class as follows and then click **Quick Save**.

```
public class StoreFrontController {

    List<DisplayMerchandise> products;

    public List<DisplayMerchandise> getProducts() {
        if(products == null) {
            products = new List<DisplayMerchandise>();
            for(Merchandise__c item : [
                SELECT Id, Name, Description__c, Price__c, Total_Inventory__c
                FROM Merchandise__c]) {
                products.add(new DisplayMerchandise(item));
            }
        }
        return products;
    }

    // Inner class to hold online store details for item
    public class DisplayMerchandise {
```

```

private Merchandise__c merchandise;
public DisplayMerchandise(Merchandise__c item) {
    this.merchandise = item;
}

// Properties for use in the Visualforce view
public String name {
    get { return merchandise.Name; }
}
public String description {
    get { return merchandise.Description__c; }
}
public Decimal price {
    get { return merchandise.Price__c; }
}
public Boolean inStock {
    get { return (0 < merchandise.Total_Inventory__c); }
}
public Integer qtyToBuy { get; set; }
}
}

```

3. Click **Catalog** to edit your page's Visualforce code.
4. Change the column definitions to work with the property names of the new inner class. Replace the existing column definitions with the following code.

```

<apex:column headerValue="Product">
    <apex:outputText value="{!pitem.Name}"/>
</apex:column>
<apex:column headerValue="Condition">
    <apex:outputText value="{!pitem.Condition}"/>
</apex:column>
<apex:column headerValue="Price">
    <apex:outputText value="{!pitem.Price}"/>
</apex:column>

```

The `outputField` component works automatically with sObject fields, but doesn't work at all with custom classes. `outputText` works with any value.

5. Click **Save** to save your changes and reload the page.
You'll notice that the price column is no longer formatted as currency.
6. Change the price `outputText` tag to the following code.

```

<apex:outputText value="{0,number,currency}">
    <apex:param value="{!pitem.Price}"/>
</apex:outputText>

```

The `outputText` component can be used to automatically format different data types.

7. Verify that your code looks like the following and then click **Save**.

```

<apex:page controller="StoreFrontController" tabStyle="Merchandise__c">

    <apex:pageBlock title="Our Products">

```

```

<apex:pageBlockSection columns="1">

    <apex:pageBlockTable value="{!products}" var="pitem">
        <apex:column headerValue="Product">
            <apex:outputText value="{!pitem.Name}"/>
        </apex:column>
        <apex:column headerValue="Condition">
            <apex:outputText value="{!pitem.Condition}"/>
        </apex:column>
        <apex:column headerValue="Price" style="text-align: right;">
            <apex:outputText value="{0,number,currency}"/>
            <apex:param value="{!pitem.Price}"/>
        </apex:column>
    </apex:pageBlockTable>

</apex:pageBlockSection>

</apex:pageBlock>

</apex:page>

```

Your catalog page will look something like this.

| Our Products | | |
|----------------|-----------|------------|
| Product | Condition | Price |
| Laptop | New | \$500.00 |
| Desktop | New | \$1,000.00 |
| Tablet | New | \$300.00 |
| Rack Server | New | \$3,245.99 |
| Windows Laptop | New | \$445.99 |
| MacBook Air | New | \$1,345.00 |

Tell Me More...

- The `DisplayMerchandise` class “wraps” the `Merchandise__c` type that you already have in the database, and adds new properties and methods. The constructor lets you create a new `DisplayMerchandise` instance by passing in an existing `Merchandise__c` record. The instance variable `products` is now defined as a list of `DisplayMerchandise` instances.
- The `getProducts()` method executes a query (the text within square brackets, also called a SOQL query) returning all `Merchandise__c` records. It then iterates over the records returned by the query, adding them to a list of `DisplayMerchandise` products, which is then returned.

Adding Action Methods to an Apex Controller

In this lesson, you'll add action method to your controller to allow it to handle clicking a new **Add to Cart** button, as well as a new method that outputs the contents of a shopping cart. You'll see how Visualforce transparently passes data back to your controller where it can be processed. On the Visualforce side you'll add that button to the page, as well as form fields for shoppers to fill in.

1. Click **StoreFrontController** to edit your page's controller code.

2. Add the following shopping cart code to the definition of `StoreFrontController`, immediately after the `products` instance variable, and then click **Quick Save**.

```
List<DisplayMerchandise> shoppingCart = new List<DisplayMerchandise>();

// Action method to handle purchasing process
public PageReference addToCart() {
    for(DisplayMerchandise p : products) {
        if(0 < p.qtyToBuy) {
            shoppingCart.add(p);
        }
    }
    return null; // stay on the same page
}

public String getCartContents() {
    if(0 == shoppingCart.size()) {
        return '(empty)';
    }
    String msg = '<ul>\n';
    for(DisplayMerchandise p : shoppingCart) {
        msg += '<li>';
        msg += p.name + ' (' + p.qtyToBuy + ')';
        msg += '</li>\n';
    }
    msg += '</ul>';
    return msg;
}
```

Now you're ready to add a user interface for purchasing to your product catalog.

3. Click **Catalog** to edit your page's Visualforce code.
4. Wrap the product catalog in a form tag, so that the page structure looks like this code.

```
<apex:page controller="StoreFrontController">
    <apex:form>
        <!-- rest of page code -->
    </apex:form>
</apex:page>
```

The `<apex:form>` component enables your page to send user-submitted data back to its controller.

5. Add a fourth column to the products listing table using this code.

```
<apex:column headerValue="Qty to Buy">
    <apex:inputText value="{!pitem.qtyToBuy}" rendered="{! pitem.inStock}"/>
    <apex:outputText value="Out of Stock" rendered="{! NOT(pitem.inStock)}"/>
</apex:column>
```

This column will be a form field for entering a quantity to buy, or an out-of-stock notice, based on the value of the `DisplayMerchandise.inStock()` method for each product.

6. Click **Save** and reload the page.
- There's a new column for customers to enter a number of units to buy for each product.

7. Add a shopping cart button by placing the following code just before the `</apex:pageBlock>` tag.

```
<apex:pageBlockSection>
    <apex:commandButton action="{!addToCart}" value="Add to Cart"/>
</apex:pageBlockSection>
```

If you click **Save** and try the form now, everything works...except you can't see any effect, because the shopping cart isn't visible.

8. Add the following code to your page, right above the terminating `</apex:form>` tag.

```
<apex:pageBlock title="Your Cart" id="shopping_cart">
    <apex:outputText value="{!cartContents}" escape="false"/>
</apex:pageBlock>
```

9. Click **Save**, and give the form a try now. You should be able to add items to your shopping cart! In this case, it's just a simple text display. In a real-world scenario, you can imagine emailing the order, invoking a Web service, updating the database, and so on.
10. For a bonus effect, modify the code on the **Add to Cart** `commandButton`.

```
<apex:commandButton action="{!addToCart}" value="Add to Cart" reRender="shopping_cart"/>
```

If you click **Save** and use the form now, the shopping cart is updated via Ajax, instead of by reloading the page.

Our Products

| Product | Condition | Price | Qty to Buy |
|----------------|-----------|------------|--------------------------------|
| Laptop | New | \$500.00 | <input type="text"/> |
| Desktop | New | \$1,000.00 | <input type="text"/> |
| Tablet | New | \$300.00 | <input type="text"/> |
| Rack Server | New | \$3,245.99 | <input type="text"/> |
| Windows Laptop | New | \$445.99 | <input type="text"/> |
| MacBook Air | New | \$1,345.00 | <input type="text" value="1"/> |

Your Cart

- MacBook Air (1)
- iPhone 5S Gold (2)

Tell Me More...

- As you saw in this lesson, Visualforce automatically mirrored the data changes on the form back to the products variable. This functionality is extremely powerful, and lets you quickly build forms and other complex input pages.
- When you click the **Add to Cart** button, the shopping cart panel updates without updating the entire screen. The Ajax effect that does this, which typically requires complex JavaScript manipulation, was accomplished with a simple `reRender` attribute.
- If you click **Add to Cart** multiple times with different values in the **Qty to Buy** fields, you'll notice a bug, where products are duplicated in the shopping cart. Knowing what you now know about Apex, can you find and fix the bug? One way might be to change a certain List to a Map, so you can record and check for duplicate IDs. Where would you go to learn the necessary Map methods...?

Summary

In this tutorial, you created a custom user interface for your Warehouse application by writing a Visualforce page with an Apex controller class. You saw how Visualforce pages can use the MVC design pattern, and how Apex classes fit into that pattern. And you saw how easy it was to process submitted form data, manage app and session data, and add convenience methods using an inner class.