Introduction to Object-Oriented Programming

Lesson 1: Introduction: Learn Object-Oriented Programming Using Java
Let's Get Going
What is Java?
What is a Java Applet?
Your Very First Applet
What Does It Mean To Compile?
Brackets and Blocks of Code

Lesson 2: Java Objects
Objects: The Source of Java's Power
Let's Break It Down
Java Classes and Objects
The Structure of Applets
Our Original Example
Experiment and Learn!

Lesson 3: Going Further with Java Objects
More on Objects
Extending a Class
Instantiation - Creating an object to use
Instantiating Multiple Objects from a Single Class
Using Multiple Constructors

Lesson 4: Variables
Variables: Containers for Values
Variable Scope
Accessing Variables from Another Class

Lesson 5: More on Variables
Setting Access Levels and Static Methods
Setting Access Levels on Member Variables
private
protected
package
public
Static Variables (also called Class Variables)
Final Variables

Lesson 6: Methods
Methods
Methods Make Things Happen
Returning Values
Passing Values as Parameters
Setting Access Levels on Methods
Static Methods (Class Methods)

Lesson 7: The Java API
Packages and APIs
Organizing Functionality
Packages: Pre-Made Tools
Using the Java API
Where does it all come from?
Where in our filesystem are these Java API directories?
What classes are available in the Java API?

Lesson 8: **Control Structures: Loops**
Loops
Your First Loop
For Loops

Lesson 9: **Control Structures: If and Switch Conditionals**
Using Conditional Statements for Decision Making
The If Statement
If/Else Statements
Complex Test Clauses

Lesson 10: **Operators**
Operators
Binary Operators
Assignment (equals)
Addition
Subtraction
Multiplication
Division
Modulus
Short-hand Addition
Short-hand Subtraction
Short-hand Multiplication
Short-hand Division
Short-hand Modulus

Binary Conditional Operators
Equality
Less Than
Greater Than
Less Than or Equal To
Greater Than or Equal To

Unary Operators
Negation
Increment
Decrement

Lesson 11: **Data-Typing and Conversion**
Converting Data Types
Converting Integers to Strings
Using Data Types
More Static Methods--'Sticky Code'
So, when to convert a static reference to a method?

Lesson 12: **Object Arrays**
*Converting Between Data Types*
  *The Array*
  *Creating Arrays in Java*
  *Something Different*

Lesson 13: **Advanced Object Orientation Part I - OO Design**
*Advanced Object Orientation Part I - OO Design*
  *Getting 'Object Oriented'*
  *What now?*

Lesson 14: **Advanced Object Orientation Part II - Abstracts**
*Advanced Object Orientation - Abstracts*
  *Thinking in the Abstract*
  *Not Entirely Abstract*
  *Variables - Abstract or Not*

Lesson 15: **Advanced Object Orientation - Interfaces**
*Advanced Object Orientation - Interfaces*
  *Multiple Inheritance*
  *Using Interfaces for Multiple-Inheritance Style Data-Typing*
  *Using Interfaces to Help our Waitress*
  *Inheritance with Interfaces*

*Copyright © 1998-2014 O'Reilly Media, Inc.*

*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See [http://creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode) for more information.*
Introduction: Learn Object-Oriented Programming Using Java

Welcome to the O'Reilly School of Technology's Object-Oriented Programming using Java course.

Course Objectives

When you complete this course, you will be able to:

- demonstrate understanding of classes, constructors, objects, and instantiation.
- access variables and modifier keywords.
- develop methods using parameters and return values.
- build control structures in an object-oriented environment.
- convert data types using API methods and objects.
- design object-oriented programs using scope, inheritance, and other design techniques.
- create an object-oriented application using Java packages, APIs, and interfaces, in conjunction with classes and objects.

In this course, you’ll be introduced to the concepts, fundamental syntax, and the thought processes behind true object-oriented programming. Completion of this course will give you the tools and basic knowledge you need to learn more specific object-oriented programming techniques in languages such as Java, C++, C#, and VB.NET.

From beginning to end, you will learn by doing your own Java applet-based projects. These projects, as well as the final project, will add to your portfolio and provide needed experience. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a user-active approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you learn to learn a new skill.

We’ll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to learn to learn. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, and you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you do break something, that's an indication to us that we need to improve our system!

- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.

- **Experiment.** Wander from the path often and explore the possibilities. We can’t anticipate all of your questions and ideas, so it’s up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.

- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you’re learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
Use all available resources! In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.

Have fun! Relax, keep practicing, and don’t be afraid to make mistakes! Your instructor will keep you at it until you’ve mastered the skill. We want you to get that satisfied, “I’m so cool! I did it!” feeling. And you’ll have some projects to show off when you’re done.

Lesson Format

We’ll try out lots of examples in each lesson. We’ll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you’ll type the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

<table>
<thead>
<tr>
<th>CODE TO TYPE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>White boxes like this contain code for you to try out (type into a file to run).</td>
</tr>
<tr>
<td>If you have already written some of the code, new code for you to add looks like this.</td>
</tr>
<tr>
<td>If we want you to remove existing code, the code to remove will look like this.</td>
</tr>
<tr>
<td>We may also include instructive comments that you don’t need to type.</td>
</tr>
</tbody>
</table>

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

<table>
<thead>
<tr>
<th>INTERACTIVE SESSION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type look like this.</td>
</tr>
</tbody>
</table>

Code and information presented in a gray OBSERVE box is for you to inspect and absorb. This information is often color-coded, and followed by text explaining the code in detail:

<table>
<thead>
<tr>
<th>OBSERVE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gray &quot;Observe&quot; boxes like this contain information (usually code specifics) for you to observe.</td>
</tr>
</tbody>
</table>

The paragraph(s) that follow may provide addition details on information that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tips provide information that might help make the tools easier for you to use, such as shortcut keys.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WARNING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warnings provide information that can help prevent program crashes and data loss.</td>
</tr>
</tbody>
</table>

Although this is your first Java course, you'll learn Java using Java Applets (we'll explain what Java Applets are in a minute). Using applets from the start is a unique feature of the OST Java course. Most other courses begin using text-based programs which generally return some pretty boring output. We use Java Applets right away because this allows you to create visually rewarding programs early on. Also, since Java Applets run in a web browser, we'll be able to use the web to communicate ideas on Java programming. In later courses you'll learn how to make servlets, do Java programming for Windows, and other cool stuff.
Let’s Get Going

What is Java?

Java is a programming language originally developed by Sun Microsystems as an alternative to other languages like C++. Before Java was created, it was likely that a piece of software written for one type of computer would require many changes in order to run on another type of computer. The intention with Java was to create a language that would allow a piece of software to be written once and then be able to run on many different types of machines (like those running Microsoft Windows, Mac OS, or UNIX to name a few).

What is a Java Applet?

An applet is a Java program that can be sent along with a web page when a user requests the page. A Java Applet runs within a web browser. The word "applet" was coined to mean "small application." To create an applet, you must create source files that extend the java.applet.Applet class. If you don’t know what this means, don’t worry, we’re just getting started. For now, just be aware that applets are written in a special way and that you need a web browser in order to run them.

Your Very First Applet

At OST we believe in learning by doing, so let's do something! Let's create our first Java Applet!

You will use CodeRunner to experiment with the lesson examples. The large white textarea you see below is the Editor. This is where you'll type your code. Make sure that you are using the proper syntax. Locate the Syntax: select box on the CodeRunner toolbar. Then click on the select box to see a list of the different modes that can be used:

In this course we will be using only the HTML and Java syntax modes of CodeRunner. Go ahead and select the Java syntax mode so we can work on our first example. In Java mode, type the code below as shown:
// These are Java comments. This syntax works for single lines.

/* These are also Java comments. Notice the syntax used to implement them. They work over multiple lines. */

import java.applet.*;
import java.awt.*;

public class CakeApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("I am a Cake.", 25, 25);
    }
}

Save this file as CakeApplet.java by clicking the button; this is the Action Button. The Action Button can do different things depending on what syntax CodeRunner is set to. In Java syntax mode, this button will save and compile your Java file. You must save this file as CakeApplet.java with all capitalization and punctuation exactly as you see here. Later, you’ll see why we’re so strict about the names you give your files.

You may not realize it, but you’ve just created your first Java source file. A source file is a set of programming statements that defines the behavior of your program. Just as a recipe is a set of instructions for baking a blueberry pie, the statements in your source file are instructions (in a human-readable format) given to the computer to execute your program. As you’ll see, this particular program doesn’t do anything.

To run the applet that we’ve just created, and so we can see it on the web, the applet needs to be embedded within an HTML document. We embed applets in much the same way as we embed images in an HTML file.

Let’s make the HTML file! Click the New File button, be sure to change the syntax mode of CodeRunner to HTML, and type the following code into the Editor below:

CODE TO TYPE:

```
<HTML>
<BODY>
<APPLET CODE="CakeApplet.class" HEIGHT="200" WIDTH="200"></APPLET>
</BODY>
</HTML>
```

Go ahead and save this file as CakeApplet.html, by clicking and typing "CakeApplet.html" in the Save As field.

Now, let’s see what this applet can do. Since you’ve already saved and compiled CakeApplet.java, you should be able to run it by Previewing the html file. You can do this by clicking the Action Button while CodeRunner is in HTML syntax mode. A new window opens that says I am a Cake.

Seems like a lot of work just to get a message that says "I am a Cake," huh? Thankfully, Java can do much more than...
just display short messages like this on a web page.

**Note**  You can also view your applet in a regular browser window by going to the URL: [http://yourlogin.oreillystudent.com/CakeApplet.html](http://yourlogin.oreillystudent.com/CakeApplet.html)

Be sure to replace *yourlogin* with the name you use to log in to these courses.

Let's take a closer look at the HTML file you just created.

```
<APPLET CODE="CakeApplet.class" HEIGHT="200" WIDTH="200"></APPLET>
```

The height and width of the applet is controlled by the `HEIGHT` and `WIDTH` attributes of the `APPLET` tag. In this case, the applet will be 200 pixels wide and 200 pixels high.

You may have noticed that you referenced a file named "CakeApplet.class" using the `CODE` attribute of the `APPLET` tag.

Wait a second—didn't we name our Java file "CakeApplet.java"? Well, we did. But remember that CakeApplet.java is a source file which defines our program in *human-readable format*. We need to convert our source file into a *machine-readable format* in order for the computer to make sense of this code. To accomplish this, we *compile* our source file. It just so happens that compiled Java files are given the `.class` file extension. This file is automatically saved to your account. In the list of files in your account, you see two files: the *source file* named CakeApplet.java and the *class file* named CakeApplet.class. The class file is the compiled version of the source file.

### What Does It Mean To Compile?

You're probably wondering what it means to compile a source file. Let me give you an analogy that might help you understand. Remember the blueberry pie that we were going to bake earlier? We compared the *recipe* for the pie to the *source file* used to create our program. In the same way that we would then use an oven to transform the raw ingredients into a delicious blueberry pie, the Java compiler transforms our source file into *"bytecode"* (machine-readable code.)

The bytecode is then *interpreted* by the Java Virtual Machine (JVM for short). The Java Virtual Machine is either software or hardware that allows your computer to run Java programs. Once the JVM has interpreted the bytecode, the program is sent to the computer in a form that can then be executed. Java software can run on many different types of computers because the JVM will interpret the program into a form that a particular computer can understand.

Your Java source files will always be compiled into files with a `.class` extension. These `.class` files contain the bytecode to be interpreted by a Java Virtual Machine. Since we are using applets to learn Java, the JVM resides in your web browser.

### Brackets and Blocks of Code

Let's take a closer look at the Java file we created.
In our example, the blue and red code make up our comments. Comments don’t really do anything—they’re simply notes to anyone who happens to look at the source code.

As our sample comments explain, comments are lines that the Java compiler ignores. The lines preceded by pairs of // (forward slashes) and contained within /* and */ (forward slash and an asterisk) are ignored by the compiler. Usually you’ll see comments used at the beginning of programs to give information about the author, copyright information, and some notes about what the program does.

Next, we see packages that need to be imported into our program. Packages are imported so that we can include code in our program. Although we cannot see the packages within the code in the box above, they are being used. This concept may be a bit confusing right now, but don’t worry if you don’t quite understand it yet. We will discuss it in depth later.

Note There is a semicolon (;) at the end of each of these lines. Semicolons are used to indicate the end of a command. You must use a semicolon at the end of each command.

Blocks of code are defined using curly brackets, or braces ({}). The outside block of code is a Class. Again, don’t panic if this isn’t perfectly clear to you just yet. We’ll learn a lot more about this later. Notice that the inside block of code is indented. We purposely indent this code (including the closing brace) so that we can tell where the blocks of code begin and end. Throughout the course, practice using the same indentations you see in our examples so that your code will be more readable. After you write a few programs it will come to you naturally!

Congratulations on completing the first steps toward learning Java! We’ve got a long way to go, but you’ve already learned the building blocks you’ll need to become a skilled Java Applet programmer. See you at the next lesson.

Copyright © 1998-2014 O’Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/ for more information.
Objects: The Source of Java's Power

In the last lesson, we compiled our first Java Applet. We also learned what it means to compile a Java source file. But we still don’t know what the code in the source file is actually doing. Over the next few lessons, we’ll break down our applet source file to get a better understanding of that.

We'll look at the big picture first, then examine big chunks of code to understand what each one does, and eventually go over each individual line of code. By approaching our task this way, we'll understand Java's structure first, and then have a foundation from which to explore the details.

Let's Break It Down

You created two files in the previous lesson—an applet named CakeApplet.java and an HTML page named CakeApplet.html. We used the HTML page to call the applet. Open these files in the CodeRunner Editor by finding them in the File Browser, to the left of this frame:

![File Browser](image)

Double-click each file to load it into the CodeRunner Editor. CodeRunner automatically opens the files in the correct Syntax Mode.

<table>
<thead>
<tr>
<th>OBSERVE: your HTML file</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;HTML&gt;</code></td>
</tr>
<tr>
<td><code>&lt;BODY&gt;</code></td>
</tr>
<tr>
<td><code>&lt;APPLET CODE=&quot;CakeApplet.class&quot; HEIGHT=&quot;200&quot; WIDTH=&quot;200&quot;&gt;&lt;/APPLET&gt;</code></td>
</tr>
<tr>
<td><code>&lt;/BODY&gt;</code></td>
</tr>
<tr>
<td><code>&lt;/HTML&gt;</code></td>
</tr>
</tbody>
</table>

We called the applet using CakeApplet.class, even though we named the file CakeApplet.java.
This applet printed the words "I am a Cake" to the screen. We'll discuss the details later, but before we do, let's talk a little about classes and objects—we'll encounter these words often throughout the course.

**Java Classes and Objects**

You may not have realized it, but creating that applet was your first exposure to a Java class definition. The word "class" has a unique definition in Java. In Java, class is defined as the blueprint that describes the properties and methods of a specific object. So what are objects in Java? For that matter, what are properties and methods in Java?

We used a cake in our latest example, so let's use a cake to understand these concepts too. Think of a Java object as a cake (we could use any real-world object, but I happen to like cake). A cake has various properties, such as flavor and color. Properties of Java objects are similar. Think of Java properties as adjectives that describe Java objects.

Staying with our analogy, a method in Java works like a verb. Methods describe actions that can typically be applied to the object. A method of a cake object might be to eat, decorate, or taste it. In Java, we use parentheses to indicate a method, so you would use `eat()`, `decorate()`, or `taste()`. So far then, we know that:

- A property is an attribute that describes an object. It is like an adjective.
- A method is something we do to or on an object. It is like a verb.

In our example, the class definition creates a CakeApplet class. Think of the class definition as a cake pan that helps create cakes. Different cake pans can make different shapes or types of cakes. A specific cake pan is a template for each cake that will be made using that particular pan. That is, every cake made with that pan will have the same shape and will be decorated the same way. In this case, CakeApplet creates an applet that can be run in a web browser.

We still haven't created any objects using the code in the example in the box above. But soon enough, we'll learn how to use our class to make objects that we can use and reuse. In the same way that we can make as many cakes as we want to using the same pan, we can make as many objects as we want to using the same class.
One **Cake class** can be used to create many Cake objects.

A class is a template or blueprint for an object, just like a cake pan is a template or blueprint to make cakes.

**OBSERVE:** Your Java Applet

```java
public class CakeApplet extends Applet {

// A Cake object called myCake

// A Cake object called hisCake

// A Cake object called herCake

We won't worry about the word `public` right now, but you can probably guess that it has something to do with classes being made available for everyone to use. We have to declare classes as one of three types—`public`, `private`, or `protected`. Private and protected class declarations are used to guard against the confusion that could occur if two different programmers use the same class name. Mostly, the classes we create will be declared as public, although we will discuss private and protected classes in more detail later.

In this line of code, we declare a new **class** named **CakeApplet**. The CakeApplet class definition lays out the various properties and methods that each CakeApplet object will have.

The second part of our class definition says **extends Applet**. Just as our CakeApplet is a Java class, so is **Applet**. But we didn't create the Applet class—Oracle did it for us. There are many other classes like this that have been created by Oracle and other people and made available for us to use. One of the great advantages of object-oriented programming languages is that we can use and reuse our classes, as well as classes other programmers have already created. Programmers make classes and objects available to each other all over the world. This makes creating new programs and objects much easier. To become a Java programmer, you need to be able to create your own Java classes, and also know how to use classes that other people have already written.

The **Applet** class was designed to give new programs (such as our **CakeApplet**) the **default** information required to function within a web browser environment. The **Applet class is a mold for making applets.** We can use it without knowing specifically how it was made or seeing any of the code used to create it, just like we can use a cake pan without knowing how it was made or from what material.

Also, the **Applet** class defines security limitations and other features that all Java applets must follow. So, how does our CakeApplet access these properties and features? Our CakeApplet **inherits** them by defining the **Applet** class as a **super-class**. By **extending** the **Applet** class, our CakeApplet (the **sub-class**) is given all of the properties of the super-class. Then our CakeApplet has the ability to change those properties and/or add new properties without requiring us to create them from scratch. It's as if we are given a mold, and then given the ability to make a more complicated mold by adding pieces to it.

In the next part of the code, we use two methods that are available through the Applet class.

**OBSERVE:**

```java
public void paint(Graphics g) {
    g.drawString("I am a Cake.", 25, 25);
}
```
This is the part of the code that does the work. For this program, it prints the words **I am a Cake** to the screen. We use the **method** named **paint()**. Again, we use the word **public**, a sure sign that we are declaring something. Next, we see the word **void**. This indicates the type of data the method will return. Think of a method as a machine that does something. Our **paint()** method does something too. However, it doesn’t return anything that can be used elsewhere, so we use the word **void**.

Next we see **paint(Graphics g)**. This time we don’t have a choice about the name. We have to call this method **paint()**. This method requires a **Graphics** object as a parameter, so we pass it a **Graphics** object that we name **g**. That’s a lot to digest all at once! Don’t worry, we’ll discuss parameters in greater detail later. For now, be aware that we could have named the Graphics object anything we wanted; we just chose **g** for Graphics. And, logically, the Graphics object is used to create graphics.

Because we have used the Graphics object, we can now use any of the methods and properties associated with it. **drawstring** is one of these methods. **drawstring** allows us to write a string of characters to the screen. **drawstring** takes three parameters—the string, the horizontal position of the string, and the vertical position of the string. In this case, the words **I am a Cake** make up the string. It will be located 25 pixels to the right and 25 pixels down from the upper left corner of the applet area.

### The Structure of Applets

Applets have a special structure. Because applets live on a web page displayed by your browser, applets must be secure and execute upon loading.

To start and execute the applet, the JVM automatically calls certain methods that are present within the applet. One of the methods that the JVM calls automatically is **paint()**. Others include **init()**, **start()**, **stop()**, and **destroy()**. These methods are called regardless of whether they are listed in our program. We put them in our applet in order to add functionality to them as they are called. For instance, we added code to the **paint()** method to print **I am a Cake** to the screen.

When the page is loaded, the methods are called in the following order:

1. **init()**: called as the applet gets to the browser
2. **start()**: called when the HTML page is completely loaded
3. **paint()**: called after start(); supplies graphics and text to the applet
4. **stop()**: called after paint(); stops the applet
5. **destroy()**: called after stop(); clears memory after the applet is finished executing

If we want something to happen, we have to put the code that makes it happen within one or more of those methods, just like we did in the **paint()** method above. Later we’ll try using other methods from within the paint method.

To help illustrate these concepts more clearly, let’s create an applet that shows the life cycle of an applet. Just like before, start by clicking the New File button, and set the Syntax Mode to Java. Now, type the following code into the Editor:
import java.applet.*;
import java.awt.*;

public class AppletLifeCycle extends Applet {
    StringBuffer buffer;

    public void init() {
        buffer = new StringBuffer();
        addItem("initializing the Applet ");
    }

    public void start() {
        addItem("starting the Applet ");
    }

    public void stop() {
        addItem("stopping the Applet ");
    }

    public void destroy() {
        addItem("preparing for unloading");
    }

    void addItem(String newWord) {
        System.out.println(newWord);
        buffer.append(newWord);
        repaint();
    }

    public void paint(Graphics g) {
        //Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);

        //Draw the current string inside the rectangle.
        g.drawString(buffer.toString(), 5, 15);
    }
}

Save the file as AppletLifeCycle.java and click the Compile button. A dialogue box appears saying "No Syntax Errors."

Remember, we'll need an HTML file to run the Applet. Click the New File button again, switch to HTML Syntax Mode, and enter the following HTML file:

```html
<HTML>
<BODY>

<APPLET CODE="AppletLifeCycle.class" HEIGHT="200" WIDTH="600"></APPLET>

</BODY>
</HTML>
```

Save the HTML code as AppletLifeCycle.html and click.
The Action Button behaves differently depending on what Syntax Mode CodeRunner is currently set to. In HTML Syntax Mode, the Action Button previews the current document. In Java Syntax Mode, the Action Button saves and compiles our Java files. The Save Button merely saves a Java file, but does not compile it.

The words "Initializing the Applet" and "Starting the Applet" are printed by the init() and start() methods. But why didn't our stop() and destroy() methods display their messages?

Our stop() and destroy() methods were indeed called, but by the time they were called, the browser no longer allowed our Applet to modify the screen. So, the words "stopping the Applet" and "preparing for unloading" were never displayed.

**Our Original Example**

Let's go back to our CakeApplet example. Open both CakeApplet.java and CakeApplet.html in the CodeRunner editor.

We'll create a method to be used with our CakeApplet class. Because we are creating the method ourselves, we can name it whatever we want. Let's call it frostCake(). frostCake() will accept two parameters: the color of the frosting and a graphics object. We must pass a graphics object in order to actually display the color. Edit your CakeApplet.java code to look like the following:

```java
import java.applet.*;
import java.awt.*;
public class CakeApplet extends Applet {
    public void paint(Graphics g) {
        frostCake(Color.blue, g);
        g.setColor(Color.black);
        g.drawString("I am a BLUE cake.", 25, 25);
    }

    public void frostCake(Color myColor, Graphics myGraphics) {
        myGraphics.setColor(myColor);
        myGraphics.fillRect(0, 0, 50, 50);
    }
}
```

Save and compile this file and preview it when prompted. It should be a little more interesting now.

The browser executes the paint() method automatically, but it won't execute frostCake() unless we specifically tell it to do so. Therefore, we call frostCake() from inside the paint() method. This is the reason we added the first line to the paint() method.

```java
frostCake(Color.blue, g);
```

When calling frostCake(), we send it two parameters. The first parameter is Color.blue, which indicates that we want to pass the blue property of the Color object to the method. The second parameter is g. Recall that we have already defined g as a graphics object in the paint() method.

Let's take a closer look at the frostCake() method:
```java
public void frostCake(Color myColor, Graphics myGraphics) {
    myGraphics.setColor(myColor);
    myGraphics.fillRect(0, 0, 50, 50);
}
```

Again, we use `public void` to indicate that we are declaring a new method. Inside the parentheses are `Color` `myColor` and `Graphics` `myGraphics`. These define the types of data that the method expects. In other words, the `frostCake()` method expects to be passed a `color` and a `graphics` object each time it is called. The values of these parameters can change each time the method is called, but it will not execute unless it is passed the same types of values as those given in the definition.

For the first parameter we use a `variable` we named `myColor`. `myColor` is a variable of type `Color`. We will discuss variables more in the next lesson. For now, think of variables as containers to hold values. This value can change (which is why it is called a variable), but it is always held in the same location. In Java, we must describe the kind of container it will be once the variable is defined. There are many pre-existing types of containers in Java, and `Color` is one of them.

For the second parameter we use a `Graphics` object named `myGraphics`. This is the same `type` of object as the graphics object `g` we used earlier.

Next, we see `myGraphics.setColor(myColor)`. `setColor()` is a pre-existing method that is accessible through the Graphics object `myGraphics`. `setColor()` requires a parameter—the color we want `myGraphics` to be. We use `setColor()` in the `paint()` method as well.

In the `paint()` method, we use `g.setColor(Color.black)`. This means that when the next statement `g.drawString("I am a blue cake.", 25, 25)` is executed, a black color will be used. That's why the words `I am a blue cake.` are written in black when you preview the applet. When we use the `setColor()` method within the `frostCake()` method, we use `myGraphics.setColor(myColor)`. Because we have passed `Color.blue` to the `frostCake()` method, when we call it, the statement `myGraphics.setColor(Color.blue)` is actually being executed. So when the next statement, `myGraphics.fillRect(0, 0, 50, 50)` is executed, a blue color is used, and the rectangle is blue when you preview the applet.

We’ve covered a lot of complicated ideas here. The most important part to understand is the structure of applets. As we move through the course, we’ll learn more about the specifics. By recognizing the correct structure, you’re ready to play around with code and see results.

**Experiment and Learn!**

Try changing the color of the rectangle to green in two different ways. First, change the color within the `paint()` method. Save/Compile and preview your code and see if it works. Then, change the color within the `frostCake()` method.

Try to create your own applet based on what we've done so far. You can add some other methods to our CakeApplet.java file. We’ve only used `setColor`, `drawString`, and `fillRect`.

Go to Oracle's website to check out a list of the available methods along with short descriptions of each one. Try to get at least one other method working in your CakeApplet.java program. Good luck!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
More on Objects

Welcome back. In this lesson we’ll go over objects in greater detail and look at two concepts related to using classes and objects. The first concept is extending classes. In the last lesson, we created the CakeApplet class by extending the applet class. The applet class was created by someone else. Next, we’ll create a new class by extending a class that we create ourselves. Finally, we’ll compare extending classes to instantiating objects.

Extending a Class

```java
import java.awt.*;
import java.applet.*;

public class CakeApplet extends Applet {
    public void paint(Graphics g) {
        frostCake(Color.blue, g);
        g.setColor(Color.black);
        g.drawString("I am a BLUE cake.", 25, 25);
    }

    public void frostCake(Color myColor, Graphics myGraphics) {
        myGraphics.setColor(myColor);
        myGraphics.fillRect(0, 0, 50, 50);
    }
}
```

Now, let's create an applet that's just like this one, but make the cake red. Instead of rewriting all of the code, we can extend our CakeApplet and make another class that uses some or all of the functionality of the CakeApplet class. Make a new file, set the Syntax Mod to Java and type the code below into the Editor:

```java
import java.awt.*;
import java.applet.*;

public class RedCakeApplet extends CakeApplet {
    public void paint(Graphics g) {
        frostCake(Color.red, g);
        g.setColor(Color.black);
        g.drawString("I am a RED cake.", 25, 25);
    }
}
```

Now make a new file in HTML Syntax Mode and type this code into the Editor below:

```html
<APPLET CODE="RedCakeApplet.class" HEIGHT="200" WIDTH="200"></APPLET>
</HTML>
```
Save the HTML file as RedCakeApplet.html. Switch back to Java and Save/Compile the applet as RedCakeApplet.java. When prompted, Preview the applet. You get the same result as before, except this time the cake is red.

In our new Java file we **extended** the CakeApplet class that we created earlier. That is, we took all of the abilities and properties of the CakeApplet class and modified them slightly. Notice that we are also using the `paint()` method in our new class. The new `paint()` method **overrides** the `paint()` method previously defined in *CakeApplet*. That is, the new one replaces the old one.

But we didn’t **override** the `frostCake()` method. We used it as it was already written. We were able to do that because the `frostCake()` method was defined in the CakeApplet class we extended. In fact, we were also able to use the `paint()` method (which was defined in the Applet class) because CakeApplet extended Applet. When we call the `frostCake` method, we are using a method that we **inherited** from CakeApplet. When we call the `paint` method, we are using a method that we inherited from CakeApplet, which inherited it from Applet. And we are passing a different color (**Color.red**) to the `frostCake` method, which makes the rectangle red.

Our RedCakeApplet is, in every way, a CakeApplet. RedCakeApplet is also an Applet, because CakeApplet extends Applet. Our RedCakeApplet **extends** Applet. RedCakeApplet has all of the functionality of our CakeApplet class and the Applet class. We can make RedCakeApplet more complex than CakeApplet by adding functionality to RedCakeApplet.

This relationship between classes is called **inheritance**. In Java, a class can only have one **super-class** (a class from which it extends and inherits). But a class can have any number of **sub-classes**. These **sub-classes** are also known as **children**. For example, if we made a YellowCakeApplet and GreenCakeApplet to go along with our RedCakeApplet, all three classes would be **children** of CakeApplet. If we changed the CakeApplet **super-class**, all three **sub-classes** would be effected. However, if we changed one of the **sub-classes** there would be no effect on CakeApplet or the other sub-classes.

Class inheritance is similar to genetic inheritance in nature. The only major difference is that **classes can inherit from only one super-class**. Here’s a graphical representation of this concept:

![Diagram of class inheritance](image)

Classes are like the strange fish above that have only one parent.

**Instantiation - Creating an object to use**

Okay, now suppose we don’t want our Java program to **inherit** all of the properties of another class, but we still want to use the functionality available through that class. Or, maybe we want to extend one class and use functionality from another class. In Java we can only **extend** one class when creating a new class. This can be limiting, especially when we want to use the functionality of the hundreds of classes available in Java. Fortunately, we can handle such situations by creating a new applet. Create a new Java file and then type the code below into the Editor:
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;

    public Cake(int top, int left) {
        topLocation = top;
        leftLocation = left;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a Cake.", leftLocation + 50, topLocation + 50);
    }
}

Save/Compile this file using the name Cake.java. We are unable to preview this file because it's not an applet, it's just a class file called Cake.class. If you check the list of files in your account, you'll see this file included now.

Now that you've successfully saved and compiled this file, let's create a new applet that will use the Cake class we just created. We'll call our new applet NewCakeApplet.java. In Java Syntax Mode, create a new file and type the code below into the Editor:

import java.applet.*;
import java.awt.*;

public class NewCakeApplet extends Applet {
    Cake myCake;

    public void init() {
        myCake = new Cake(0,0);
    }

    public void paint(Graphics g) {
        myCake.frostCake(Color.red, g);
    }
}

Next, we need to call the constructor. After all, the cake has to be made, right? The constructor is like the chef who bakes the cake. We call the constructor in the init() method because making the cake is an initialization. Alternatively, we could have put it in the start() method--either way is fine, so long as we call it using one of the methods that is automatically called before the paint() method. The syntax used to call the constructor looks like this:

    myCake = new Cake(0,0);

With this code, you create a new Cake using the Cake class which was created in Cake.java. This particular class takes (0,0) as its parameters. (You'll see why when we discuss the Cake class later.) Because myCake is a type of Cake, we can use all of the properties and methods associated with Cake, including the frostCake() method. Here's how we call the frostCake() method:
public void paint(Graphics g) {
    myCake.frostCake(Color.red, g);
}

We use the `paint()` method to call `frostCake()`. Because the `frostCake()` method can now be accessed through the `myCake` object, we must use dot notation, like this:

```
    myCake.frostCake(Color.red, g);
```

This process is called instantiation. We created an instance of an object that was created in our old class, within our new class.

Let's take a closer look at how we created the Cake class:

```
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;

    public Cake(int top, int left) {
        topLocation = top;
        leftLocation = left;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a Cake.", leftLocation + 50, topLocation + 50);
    }
}
```

Our Cake class definition is different from the class definitions we've used before in that it doesn't extend `applet`. That's because Cake isn't an applet, it's just a class, so it doesn't have the `paint()` method or any of the other self-calling methods available for use.

Let's check out the first few lines of our Cake class definition:

```
int topLocation;
int leftLocation;

public Cake(int top, int left) {
    topLocation = top;
    leftLocation = left;
}
```

First, we declare two variables. We'll cover variables in greater detail later in the course. For now, be aware that you must always declare a variable before you use it, by specifying the type of variable it is. In this case, both of our variables are of type `int`, which means that the value stored in these variables will always be an integer. The names of our new variables are `topLocation` and `leftLocation`.

Next, we declare a special kind of method called a `constructor`. In order to **instantiate** an object, you must
Next, we declare a special kind of method called a constructor. In order to instantiate an object, you must have a constructor. You can always recognize constructors because they always have the same name as the class from which you are instantiating the object. Because we are instantiating the Cake object, the constructor is named Cake.

Constructors are used to initialize the object's member variables. This particular constructor takes two parameters: int top and int left. These will be used to determine where the cake will be colored and where the words I am a Cake will be written. A class can have as many constructors as you like, and each constructor can have as many parameters as you like. Because an object needs a constructor in order to be instantiated, a constructor will automatically be generated if you do not declare one. This default constructor will have no parameters.

Once the constructor is defined, the frostCake() method is declared. The frostCake() method we're using here is essentially the same as the frostCake() method used in CakeApplet.java. That is, we are using the method to color the cake and to write I am a Cake. The main difference between the two methods is that we are using the variable's leftLocation and topLocation in the new frostCake() method:

```java
public void frostCake(Color myColor, Graphics g) {
    g.setColor(myColor);
    g.fillRect(leftLocation, topLocation, 50, 50);
    g.setColor(Color.black);
    g.drawString("I am a Cake.", leftLocation + 50, topLocation + 50);
}
```

So where do we get the value of these variables? Remember, we called the constructor in NewCakeApplet:

```java
myCake = new Cake(0,0);
```

This creates a new Cake. The value of the top and left parameters in the Cake constructor will be set equal to 0 and 0. In other words, top=0 and left=0. This also means that the value of topLocation is equal to 0 and the value of leftLocation is equal to 0.

**Instantiating Multiple Objects from a Single Class**

While you're still in Java Syntax Mode, modify NewCakeApplet.java to look like this:

```java
import java.applet.*;
import java.awt.*;

public class NewCakeApplet extends Applet {

    Cake myCake,yourCake;

    public void init() {
        myCake = new Cake(0,0);
        yourCake = new Cake(100,0);
    }

    public void paint(Graphics g) {
        myCake.frostCake(Color.red, g);
        yourCake.frostCake(Color.blue, g);
    }
}
```

Save/Compile this file, naming it NewCakeApplet.java, then preview it with HTML. So, what have we done? There are two colored rectangles and two lines of text that read I am a Cake. We instantiated two new Cake objects. We used myCake, just like we did the first time. We also declared yourCake as a new Cake.
We passed different values for top and left when we called the constructor. We set the position properties of yourCake so that they were a little different from myCake.

When we called the frostCake() method through yourCake, we passed a different color than the one we used for myCake. That's why yourCake is blue and myCake is still red. By instantiating two Cake objects, we have created two objects from the Cake class, each with different properties.

**Using Multiple Constructors**

Each class can have as many constructors as you want them to have. You might need more than one constructor in order to set different properties when you instantiate an object from that class. Switch back to the Cake.java file in the CodeRunner Editor so we can add another constructor to the Cake class. In Java Syntax Mode, type the code below into the Editor:

```
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;
    String cakeFlavor;

    public Cake(int top, int left) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = "";
    }

    public Cake(int top, int left, String flavor) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a" + cakeFlavor + "Cake.", leftLocation + 50, topLocation + 50);
    }
}
```

Save/Compile this file using the name Cake.java.

Before we test this out, let's look at what we did. We created another constructor for Cake. It looks almost exactly like the first constructor, but with one important difference—this constructor is defined using three parameters.

**CODE TO TYPE:**

```
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;
    String cakeFlavor;

    public Cake(int top, int left) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = "";
    }

    public Cake(int top, int left, String flavor) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a" + cakeFlavor + "Cake.", leftLocation + 50, topLocation + 50);
    }
}
```
The second constructor uses `topLocation` and `leftLocation` as parameters, just like the first constructor does. But it also uses a third parameter called `cakeFlavor`. We use this variable in the `frostCake()` method to indicate the flavor of the Cake when we write text to the screen. Now we have two constructors—one that sets the `cakeFlavor` and one that does not.

We declared the `cakeFlavor` variable just like we declared `topLocation` and `leftLocation`. `cakeFlavor` is a `String` variable. We've also modified our original constructor by adding the `cakeFlavor`. This property is initialized to an empty string, which ensures that the `cakeFlavor` property is always defined. In this case, `cakeFlavor` is a string variable:

```java
String cakeFlavor;
```

Now let's use these two different constructors. In the CodeRunner Editor, switch back into the `NewCakeApplet.java` file and make the necessary changes. Edit `NewCakeApplet` so it looks like this:

```java
import java.applet.*;
import java.awt.*;

public class NewCakeApplet extends Applet {
    Cake myCake, yourCake, anotherCake;

    public void init() {
        myCake = new Cake(0,0);
        yourCake = new Cake(100,0);
        anotherCake = new Cake(0,100,"lemon");
    }

    public void paint(Graphics g) {
        myCake.frostCake(Color.red,g);
        yourCake.frostCake(Color.blue,g);
        anotherCake.frostCake(Color.yellow,g);
    }
}
```

Save/Compile this file using the name `NewCakeApplet.java`. Make sure you have the appropriate HTML file so you can preview the applet with HTML.

This time, we instantiated three objects from the Cake class. The first two objects (`myCake` and `yourCake`) use only the first constructor, so they look the same as they did before. The third object (`anotherCake`) uses the second constructor. The value of the `cakeFlavor` variable is set equal to `lemon`.

Nice work. You're doing great so far, already instatiating three objects from a class--good job! We'll create and use lots more classes and constructors as we continue through the course.

*Copyright © 1998-2014 O'Reilly Media, Inc.*
Variables: Containers for Values

So far, we’ve looked at several examples of Java Applets and gotten a better understanding of the structure of Java programs. We also learned about variables. A variable is a container that allows us to store information temporarily by giving that information a unique name. In this lesson we’ll delve deeper still into variables and ways to use them.

Let’s get started! Open the latest version of Cake.java into the CodeRunner Editor:

```java
import java.awt.*;
public class Cake {
    int topLocation;
    int leftLocation;
    String cakeFlavor;

    public Cake(int top, int left) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = "";
    }

    public Cake(int top, int left, String flavor) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a" + cakeFlavor + "Cake.", leftLocation + 50, topLocation + 50);
    }
}
```

The Cake class we created in the last lesson made extensive use of variables. All of that **bold** code in the box above are variables. Each **light blue** variable (such as **top**) is a type of variable called a **parameter**. The value of each variable is set with the assignment operator = (an equals sign). Alternatively, the value can also be set when the method is called. The values of **top** and **left** were set when the constructor **Cake(0,0)** was called in NewCakeApplet.java:

```java
    myCake = new Cake(0,0);
```

This code set the value of **top** equal to 0 and the value of **left** equal to 0.

Variables must be **declared**. That is, the name of the variable and its **data type** must be set before the variable can be used. For example, the following variables are declared as data types **int** and **String** in Cake.java:

```java
    int topLocation;
    int leftLocation;
    String cakeFlavor;
```
**int** is an abbreviation for **integer**. When we declare a variable of type **int** we are specifying that the variable will only hold values that are integers. **int** is known as a **primitive** data type, which means that it is built into Java.

This table details all of the primitive data types available in Java:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Allowed Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>8 bits</td>
<td>true or false</td>
</tr>
<tr>
<td>char</td>
<td>16 bits</td>
<td>single lower/upper case letter, single digit, special characters ($, *, etc.), and escape sequences</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>long</td>
<td>64 bits</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>-3.4029234E38 to 3.4029234E38</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>-1.7976931348623157E+308 to 1.7976931348623157E+308</td>
</tr>
</tbody>
</table>

We'll use all of these data types during this course.

Here are some examples of how different values might appear and their corresponding data types:

<table>
<thead>
<tr>
<th>Literal</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>int</td>
</tr>
<tr>
<td>8864</td>
<td>long</td>
</tr>
<tr>
<td>37.266</td>
<td>double</td>
</tr>
<tr>
<td>87.363</td>
<td>float</td>
</tr>
<tr>
<td>26.77e3</td>
<td>double</td>
</tr>
<tr>
<td>'c'</td>
<td>char</td>
</tr>
<tr>
<td>true</td>
<td>boolean</td>
</tr>
<tr>
<td>false</td>
<td>boolean</td>
</tr>
</tbody>
</table>

The other data type used in our sample code above is **String**. **String** is not a primitive data type. It's a data type known in Java as a **reference**. A reference is any data type that is not primitive, usually a **class**, an **array**, or an **interface**. We'll discuss arrays and interfaces later. For now, let's focus on classes.

Recall that we can declare a class to be of the Cake data type, as we did in **NewCakeApplet.java**:

```
Cake myCake;
```

And in the way we created the Cake class, someone else created the **String** class, so we can use **String** as a data type as well.

A **String** is a sequence of characters. The String class uses the primitive data type **char** to create a string object. Every time you want to print text to the screen, you use a string. In fact, we did that when we used the **g.drawString()** method:

```
g.drawString("I am a " + cakeFlavor + " Cake.", leftLocation + 50,topLocation + 50);
```

See the quotation marks around **I am a** and **Cake**? They indicate that we want to print these exact words to the screen. If we did not put quotation marks around these words, they would be interpreted as variables instead. The program would look for the values given to these variables and those values would be printed to the screen. Because **cakeFlavor** is a variable, there are no quotation marks around it.

We use plus signs (+) between the strings of words with quotation marks and the **cakeFlavor** variable without quotation marks to indicate that the two will be **concatenated**, and to make sure that only one string of words is printed to the screen.

Every time we set the value of a String, we put that value within quotation marks (unless we are setting the value equal to a variable). The general syntax used is:

```
String theVariableName = "the text you want to print to the screen";
```
Variable Scope

The scope of a variable is essentially the boundary for which the variable is valid. Because variable names often get used more than once, we need boundaries to make it clear when the variable is applied. Variables are confined to the block of code in which they are declared. The variable's scope is limited to that block of code.

One way to visualize variable scope is to think of class and method definitions as fences that contain variables. One end of the fence is indicated by { and the other end of the fence is indicated by }. Now think of the variables as farm animals that live inside the fence (just play along). There are three variables in the picture below—a Chicken, a Sheep, and a Fox. The Chicken is confined to the method 1 fence and can't go anywhere else. The Sheep is also confined. But the Fox is able to go anywhere within the FarmAnimals class fence. This includes the method 1 fence and the method 2 fence. Therefore, our Chicken and Sheep are not safe. Variables that behave like the Fox are known as member variables.

Now take a look at Cake.java. topLocation, leftLocation, and cakeFlavor are used throughout the Cake class. They are member variables. They can be used within smaller blocks of code (such as constructors).

The variables top, left, and flavor, are declared within constructors or methods. The scope of these variables is limited to these blocks of code. They are local variables. They can only be used within their local block of code.

Let's try some of this stuff out. Break the Cake class by using a variable outside of the variable's scope. Create a new Java file and type the following into the Editor:
import java.awt.*;

public class BrokenCake {
    int topLocation;
    int leftLocation;
    String cakeFlavor;
    
    public BrokenCake(int top, int left) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = "";
    }
    
    public BrokenCake(int top, int left, String flavor) {
        String lostVariable;
        lostVariable = "WONDERFUL";
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }
    
    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a " + lostVariable + " Cake.", leftLocation + 50, topLocation + 50);
    }
}

When we Save/Compile this file (using the name BrokenCake.java), we get an error!

OBserve:

users/username/BrokenCake.java:29: Undefined variable: lostVariable
  g.drawString("I am a " + lostVariable + " Cake.", leftLocation + 50, topLocation + 50);
^ 1 error

LostVariable was declared in the BrokenCake constructor, and it can only be used in the code between { and } that define this constructor. To make the value of lostVariable available to other methods, it must be declared as a member variable. Modify the BrokenCake source file to look like this:
import java.awt.*;

public class BrokenCake {
    int topLocation;
    int leftLocation;
    String cakeFlavor;
    String lostVariable;

    public BrokenCake(int top, int left) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = "";
    }

    public BrokenCake(int top, int left, String flavor) {
        lostVariable = "WONDERFUL";
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a " + lostVariable + " Cake.", leftLocation + 50, top Location + 50);
    }
}

Save/Compile. Success! This time BrokenCake compiled without errors. Now that we’ve given our lostVariable the proper scope by declaring it outside of a method, we can access its value from our frostCake() method.

**Accessing Variables from Another Class**

We can also access variables in one class from another class. Let’s try it, by creating another applet that uses the Cake class. Create a new Java file and type the following code into the Editor:
```java
import java.applet.*;
import java.awt.*;

public class AnotherCakeApplet extends Applet {

    Cake myCake;

    public void init() {
        myCake = new Cake(0,0,"chocolate");
    }

    public void paint(Graphics g) {
        g.setColor(Color.black);
        g.drawString(myCake.cakeFlavor, 10, 10);
    }
}
```

Save/Compile this file as AnotherCakeApplet.java. To preview the new applet, type the following into the Editor below (make sure you create a New File in HTML Syntax Mode):

```html
<HTML>
<BODY>
<APPLET CODE="AnotherCakeApplet.class" HEIGHT="200" WIDTH="200"></APPLET>
</BODY>
</HTML>
```

Save this HTML file as AnotherCakeApplet.html. Then preview the file by clicking The Action Button Compile in HTML Syntax Mode.

Here we accessed the cakeFlavor member variable from the Cake object myCake. We used the dot notation (myCake.cakeFlavor) to access it. Dot notation is used whenever we want to access an object or property that exists within another object. For example, to access my weight, I would use the syntax scott.weight. To access my personality I would use the syntax scott.personality. You have seen dot notation used before with a method:

```
myCake.frostCake(Color.red,g);
```

So if I wanted to run 5 miles (and that's a big if!), I would use the syntax scott.run(5).

You can't always access all of the member variables in one class from another class, though. We'll learn more about that in the next lesson. See you there!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
More on Variables

Setting Access Levels and Static Methods

Setting Access Levels on Member Variables

We build Java classes not only for our own purposes, but for other programmers to use as well. Keep that in mind. Java’s great popularity and success stem from the fact that it lends itself to the creation of reusable code. In fact, you are going to contribute to its ongoing success as you write your own programs.

Potential problems exist when making classes available to other programmers, though. Different variables or methods may have been given the same name, or a programmer might change the value of a variable in a class and in doing so, prevent the code from working correctly.

Fortunately, we can protect our variables from other programmers’ applications by controlling access to them. We control access to variables by specifying that they are either private, package, protected, or public (these are listed in order, from the most restrictive to the least restrictive variable). This table shows the access level permitted by each:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>class</th>
<th>subclass</th>
<th>package</th>
<th>world</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

private

This is the most restrictive level of access you can place on a member variable (or method as we’ll see in the next lesson.) It restricts access to the class that contains the member variable. Let’s see how it works using an example. Open our now famous Cake.java file in the Editor. Add the red code to your existing code by typing it into the Editor:
import java.awt.*;
import java.applet.*;

public class CakeExample {
    int topLocation;
    int leftLocation;
    private String cakeFlavor;

    public CakeExample(int top, int left) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = "chocolate";
    }

    public CakeExample(int top, int left, String flavor) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation,topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a" + cakeFlavor + "Cake.", leftLocation + 50, topLocation + 50);
    }
}

Save/Compile this file using the name CakeExample.java

Of course, in order to test this code we must use an applet that instantiates the CakeExample.java class. Open up the file AnotherCakeApplet.java that we saved in the last lesson. Add the red code to that file as shown:

import java.awt.*;
import java.applet.*;

public class AnotherCakeApplet extends Applet {
    CakeExample myCake;

    public void init() {
        myCake = new CakeExample(0,0);
    }

    public void paint(Graphics g) {
        g.setColor(Color.black);
        g.drawString("I am a" + myCake.cakeFlavor + "Cake.", 0, 0);
        myCake.frostCake(Color.black, g);
    }
}

Save/Compile this file as AnotherCakeApplet.java. As you may have guessed, we'll need an HTML file to run our Applet. Create a new HTML file and type the code below into the Editor:
Be sure to save the HTML file using the name `AnotherCakeApplet.html`. This might be a good time to double-check and make sure that you have both `AnotherCakeApplet.java` and `CakeExample.java` saved in your account; we’ll be using them in the upcoming examples.

When you Preview `AnotherCakeApplet.java` with HTML you’ll see that it didn't work. What happened?

Well, the first step in figuring that out is to look at the compile errors that were generated when you compiled this applet:

```
/users/username/AnotherCakeApplet.java:18: Variable cakeFlavor in class CakeExample
    g.drawString(myCake.cakeFlavor, 0, 0);
    ^
1 error
```

The error states that the variable `cakeFlavor` in class `CakeExample` is not accessible from class `AnotherCakeApplet`. This is because we set the accessibility of `cakeFlavor` (in the `CakeExample` class) to `private`.

A `private` variable is only accessible from the class in which it is declared. An error was generated because we tried to access this variable from `AnotherCakeApplet`. Private variables are used to help set values and are usually returned through a method. (We will discuss this more in the next lesson.)

A `protected` variable is available to the class in which it is declared AND any subclass that extends that class. It is also available to any class that is in its package (we’ll talk more about packages in a bit).

Take another look at the file `CakeExample.java`. Try changing `private String cakeFlavor` to `protected String cakeFlavor`. You don’t get an error message! That’s because the `AnotherCakeApplet` and `CakeExample` classes are in the same package.

We’ll see examples of this later, when we have learned more about packages. Be sure to keep an eye out for those protected variables when you learn more about packages!

A `package` variable is declared by leaving off all other declarations.

Think of a package as, well, a package. It’s a box that keeps related objects together. The items in a package are all related. They all have access to each other’s member variables, while keeping them secret from classes that lie outside of the package.

We’ll see some examples of this later in the lesson devoted to packages.

This access level makes all of the members available to any class.
To make a variable public, we put the keyword `public` in front of it. Take another look at the file `CakeExample.java`. Change `private String cakeFlavor` to `public String cakeFlavor`. Then Preview AnotherCakeApplet.java with HTML. This time it should work!

**Static Variables (also called Class Variables)**

So, let's say we want to instantiate three objects from the same class, for example, three different objects from the Cake class: `myCake`, `hisCake`, and `herCake`. Each object will have separate instances of the member variables. For example, one of the member variables is `cakeColor`, which stores the color of the cake. We can set the value of `cakeColor` to three different colors, then each of the three cakes will be a different color. If we change one of the member variables (like `myCake.cakeColor`), the other two member variables (`hisCake.cakeColor` and `herCake.cakeColor`) will not change.

If we make `cakeColor` a static variable, and we change one of the member variables in one of the objects, then all of the variables will change.

Let's try an example. First, we'll make a class that doesn't use static and an applet that instantiates three ColorCake objects. We'll see that changing a member variable in one object doesn't affect the member variables in the other objects. Create a new Java file and type the following code into the Editor:

```
import java.awt.*;
public class ColorCake {
    int topLocation;
    int leftLocation;
    Color cakeColor;

    public ColorCake(int top, int left, Color fillcolor) {
        topLocation = top;
        leftLocation = left;
        cakeColor = fillcolor;
    }

    public void frostCake(Graphics g) {
        g.setColor(cakeColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
    }
}
```

Save/Compile this file using the name `ColorCake.java`.

We've declared a variable called `cakeColor` that is of reference type `Color`. Just like `String`, `Color` is an object someone else made for us. The variable `cakeColor` will be set when the constructor `ColorCake` is called in another class and the Color parameter `fillColor` is set.

Now we need an applet that instantiates `colorCake`. Let's call this applet `ColorCakeApplet.java`. Create another new Java file and type the following code into the Editor:

```
```
import java.applet.*;
import java.awt.*;

public class ColorCakeApplet extends Applet {

    ColorCake myCake, hisCake, herCake;

    public void init() {
        myCake = new ColorCake(0,0,Color.orange);
        hisCake = new ColorCake(0,75,Color.red);
        herCake = new ColorCake(0,150,Color.blue);
    }

    public void paint(Graphics g) {
        myCake.frostCake(g);
        hisCake.frostCake(g);
        herCake.frostCake(g);
    }
}

Before compiling this file, let's make an HTML page so we can see the applet in action. Create a new HTML file and type the following code into the Editor:

<HTML>
< BODY >
< APPLET CODE="ColorCakeApplet.class" HEIGHT="200" WIDTH="200"></APPLET>
</ BODY >
</ HTML >

Now go back to ColorCakeApplet.java. Save/Compile the Java file using the name ColorCakeApplet.java, then Preview the applet when prompted.

Your applet displays an orange square, a red square, and a blue square on the screen. Now let's change the value of the cakeColor variable for myCake. We'll see that hisCake and herCake are not effected by this change. Add the green line of code below to ColorCakeApplet.java:

CODE TO TYPE:

import java.applet.*;
import java.awt.*;

public class ColorCakeApplet extends Applet {

    ColorCake myCake, hisCake, herCake;

    public void init() {
        myCake = new ColorCake(0,0,Color.orange);
        hisCake = new ColorCake(0,75,Color.red);
        herCake = new ColorCake(0,150,Color.blue);
    }

    public void paint(Graphics g) {
        myCake.frostCake(g);
        hisCake.frostCake(g);
        herCake.frostCake(g);
    }
}

Before compiling this file, let's make an HTML page so we can see the applet in action. Create a new HTML file and type the following code into the Editor:

<HTML>
< BODY >
< APPLET CODE="ColorCakeApplet.class" HEIGHT="200" WIDTH="200"></APPLET>
</ BODY >
</ HTML >

Now go back to ColorCakeApplet.java. Save/Compile the Java file using the name ColorCakeApplet.java, then Preview the applet when prompted.

Your applet displays an orange square, a red square, and a blue square on the screen. Now let's change the value of the cakeColor variable for myCake. We'll see that hisCake and herCake are not effected by this change. Add the green line of code below to ColorCakeApplet.java:

CODE TO TYPE:

import java.applet.*;
import java.awt.*;

public class ColorCakeApplet extends Applet {

    ColorCake myCake, hisCake, herCake;

    public void init() {
        myCake = new ColorCake(0,0,Color.orange);
        hisCake = new ColorCake(0,75,Color.red);
        herCake = new ColorCake(0,150,Color.blue);
    }

    public void paint(Graphics g) {
        myCake.frostCake(g);
        hisCake.frostCake(g);
        herCake.frostCake(g);
    }
}
import java.applet.*;
import java.awt.*;

public class ColorCakeApplet extends Applet {

    ColorCake myCake, hisCake, herCake;

    public void init() {
        myCake = new ColorCake(0,0,Color.orange);
        hisCake = new ColorCake(0,75,Color.red);
        herCake = new ColorCake(0,150,Color.blue);
    }

    public void paint(Graphics g) {
        myCake.cakeColor = Color.green;
        myCake.frostCake(g);
        hisCake.frostCake(g);
        herCake.frostCake(g);
    }

}

Save/Compile the file again, then preview it using HTML. Your applet displays a green square, a red square, and a blue square on the screen. We changed the myCake object so that it would be green. All of the other objects remained the same.

Now let's make cakeColor a static member variable. Open up the original ColorCake.java file. Add the word static to the cakeColor declaration:

import java.awt.*;

public class ColorCake {
    int topLocation;
    int leftLocation;
    static Color cakeColor;

    public ColorCake(int top, int left, Color fillcolor) {
        topLocation = top;
        leftLocation = left;
        cakeColor = fillcolor;
    }

    public void frostCake(Graphics g) {
        g.setColor(cakeColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
    }

}

Save/Compile this file. Now open the ColorCakeApplet.java file again. Save/Compile ColorCakeApplet.java and preview the applet when prompted.

What happened? All three squares are green now! This is the result of making the cakeFlavor variable
**static.** When we changed the value of cakeColor for the myCake object, the color of the hisCake object and herCake object changed too. Think of "static" as something that ties instances of the class together. Changing a static value in one object changes the value in all objects.

A class can access the value of a **static** variable without **instantiating** the object. In the above example, we can use the statement `ColorCake.cakeColor`, access the value held in the cakeColor variable, and change its value without instantiating myCake, or hisCake, or herCake. Static variables are also called **class variables** because the value can be accessed through a class instead of an object.

### Final Variables

We can also declare a variable to be **final**. Declaring a variable as **final** makes it constant. Once it's set, it cannot be changed by anybody. Let's build on our last example to see this tool in action. Once again, open up the file `ColorCake.java`. Add the word **final** to the cakeColor declaration:

```
import java.awt.*;
public class ColorCake {
    int topLocation;
    int leftLocation;
    static final Color cakeColor = Color.black;

    public ColorCake(int top, int left, Color fillcolor) {
        topLocation = top;
        leftLocation = left;
        cakeColor = fillcolor;
    }

    public void frostCake(Graphics g) {
        g.setColor(cakeColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
    }
}
```

Save/Compile this file using the name `ColorCake.java`. We get an error:

```
/Users/username/ColorCake.java:12: Can't assign a value to a final variable: cakeColor
cakeColor = fillcolor;
^ 1 error
```

What happened? We defined the cakeColor variable as **final**, meaning its value cannot change. Once this value is set, it is permanent. So, when we tried to modify the value of cakeColor, the compiler gave us an error. Remember that for future reference.

Phew! We are done with variables for now, but we'll revisit these concepts throughout the course.
Methods

Methods Make Things Happen

We've used methods throughout the course. In this lesson, we'll go over methods in greater detail, and we'll also investigate parameters, returning values, using static methods, and protecting methods.

Let's take a look at some of the ways we've used methods so far. Open up the latest version of Cake.java.

```
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;
    String cakeFlavor;

    public Cake(int top, int left) {
        topLocation = top;
        leftLocation = left;
    }

    public Cake(int top, int left, String flavor) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a" + cakeFlavor + "Cake.", leftLocation + 50, topLocation + 50);
    }
}
```

All of the code in green defines a method. We've discussed these methods in previous lessons, so they're probably familiar to you. The Cake method is a special method called a constructor. A constructor can be located easily because it has the same name as the class. Constructors are used to "construct" an object from a class. They allow us to set different properties when we instantiate an object from a class.

We have also used the frostCake() method extensively. We used it to create a colored square and print words to the screen.

Whenever a method is defined, parentheses are used. The parentheses contain the parameters of the method. A parameter is a container that will hold the values passed to the method when it is called. The frostCake() method requires two parameters. Each parameter is separated by a comma. The two parameters of the frostCake() method are myColor and g:
You can define a method using as many parameters as you like, or you don’t have to use parameters at all if you don’t need them. But you must always use parentheses, even if there are no parameters within them. If `frostCake()` didn’t have any parameters, it would be defined like this:

```
OBSERVE:

    public void frostCake() {
```

The statements that make up the method are placed in between open and closed curly brackets — `{ and }`.

### Returning Values

In the last lesson we saw that we could use variables to get values from the class. We can also get values from methods. Have you noticed that when we define methods (with the exception of constructors), we put the keyword `void` in front of the method name? We used `void` in the `frostCake()` definition:

```
OBSERVE:

    public void frostCake(Color myColor, Graphics g) {
```

`Void` indicates that the method does not return any value to us; it only executes the statements within the curly brackets. But what if we want to retrieve a value from a method? Suppose we want to give a method two words, use the method to combine the two words, then retrieve this new word? Let's try it! Add the following method to the `Cake.java` file:
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;
    String cakeFlavor;

    public Cake(int top, int left) {
        topLocation = top;
        leftLocation = left;
    }

    public Cake(int top, int left, String flavor) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a" + cakeFlavor + "Cake.", leftLocation + 50, topLocation + 50);
    }

    public String combine(String firstString, String secondString) {
        String combinedString = firstString + secondString;
        return combinedString;
    }
}

Before you Save/Compile this, notice we used String when we defined the combine() method where we would have normally used void. When we call the combine() method, it will return a string! We can access this string with the variable combinedString.

Save/Compile the file Cake.java.

Now let's make an applet that uses this class and our new method. We'll need an HTML page to preview our applet, so let's go ahead and make that now. Create a new file in HTML Syntax Mode, type the code below into the Editor:

```html
<HTML>
<BODY>

<APPLET CODE="AnotherCakeApplet.class" HEIGHT="200" WIDTH="200"></APPLET>

</BODY>
</HTML>
```

Save this as AnotherCakeApplet.html! Open up the file AnotherCakeApplet.java into the Editor. In CodeRunner, modify the source code so that it looks exactly like this:
Save/Compile this file using the name AnotherCakeApplet.java, then Preview the applet when prompted. You’ll see the words AngelFood printed to the screen. The combine() method was called in AnotherCakeApplet.java:

The combine() method returns the value stored in combinedString. The returned value will be stored in cakeType when the method is called. In this example, the combine() method returns the value "AngelFood", so that's the value stored in the cakeType variable. This is why the word AngelFood is printed to the screen.

### Passing Values as Parameters

Parameters are really just variables. They are used to store values which are passed to the method when it is called. In the combine() method, the parameter firstString is used to store “Angel” and the parameter secondString is used to store ”Food”.

Notice that we must declare what type of data the parameter will store. In this case, each parameter will store a String.

The scope of a parameter is limited to its method. So in our example, firstString and secondString have no meaning outside of the statements within the combine() method.

### Setting Access Levels on Methods

In the last lesson you learned how to set access levels for member variables. We can also set access levels for methods. They can be public, private, protected, or package: -No access modifier (package access): the default access level, visible to the package only. -Public modifier: visible to any other Java class in the world. -Private modifier: visible to the current class only. -Protected modifier: visible to sub-class of the current class, and any class in the same package as the current class. These access levels work in exactly the same way as they do for variables. So, if we set the access level for the combine() method to private in the Cake class, we could not have called it in our AnotherCakeApplet class. Instead, we would be limited to using the combine() method locally, within the Cake class.

Try changing the access level on the combine() method to determine if any level other than public works.

### Static Methods (Class Methods)

CODE TO TYPE: AnotherCakeApplet.java

```java
import java.applet.*;
import java.awt.*;

public class AnotherCakeApplet extends Applet {
    Cake myCake;

    public void init() {
        myCake = new Cake(0,0);
    }

    public void paint(Graphics g) {
        String cakeType = myCake.combine("Angel","Food");
        g.setColor(Color.black);
        g.drawString(cakeType, 10, 10);
    }
}
```
Static methods are extremely useful in Java. The static keyword has roughly the same effect on methods as it did on variables. If a method is declared as static, then you can call this method without instantiating an object from the class in which it resides.

Let's change our last example so that the `combine()` method is static. Open up the file `Cake.java` into the CodeRunner Editor, and change the code as shown:

```java
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;
    String cakeFlavor;

    public Cake(int top, int left) {
        topLocation = top;
        leftLocation = left;
    }

    public Cake(int top, int left, String flavor) {
        topLocation = top;
        leftLocation = left;
        cakeFlavor = flavor;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
        g.drawString("I am a" + cakeFlavor + "Cake.", leftLocation + 50, topLocation + 50);
    }

    public static String combine(String firstString, String secondString){
        String combinedString = firstString + secondString;
        return combinedString;
    }
}
```

Save/Compile this file using the name `Cake.java`.

Now switch the file so that `AnotherCakeApplet.java` is displayed in the Editor. Change `AnotherCakeApplet.java` so that it looks like this:
import java.applet.*;
import java.awt.*;

public class AnotherCakeApplet extends Applet {

    // Cake myCake; notice this is commented out.

    public void init() {
        //
        myCake = new Cake(0,0);
    }

    public void paint(Graphics g) {
        String cakeType = Cake.combine("Angel","Food");
        g.setColor(Color.black);
        g.drawString(cakeType, 10, 10);
    }
}

You can see that we've commented out the lines that instantiate the myCake object from the Cake class, and that we reference the Cake class directly when we call the combine method.

**Note**

Commenting out a line of code means that line won’t be recognized by the program. We’ve done it here in Java using two back-slashes (//). We’ll go over this in detail later, so don’t worry about it for now.

Save/Compile this file using the name AnotherCakeApplet.java, and preview the file when prompted. It should work the same as before, except this time we did not instantiate an object.

So why would we want to use static methods anyway? It doesn’t always make sense to instantiate an object when you want to use a method. For instance, Java supplies us with a Math object that has methods to perform mathematical operations, like Math.sqrt() (the square root function). We don’t have to instantiate any Math objects to use these methods since they are static. We simply call Math.sqrt() just like we called Cake.combine() above.

In our example we just changed a local variable and returned it. We should keep in mind that static methods can only work with static variables.

Before you leave this lesson, try experimenting on your own with this:

- Change the Cake class above so that combinedString is a member variable (not just defined for the method). Make sure that combine() is still a Static Method. The idea is to see that the class won’t compile unless the combined String variable is defined as static.
- Try using the Math object. Make a call to Math.sqrt() in AnotherCakeApplet.java and print out the result. Try taking the square root of 9 like this Math.sqrt(9); (set it to a variable then print that variable out).

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Packages and APIs

Organizing Functionality

Have you ever been to an open street market? You know, the type that has lots of little stores that each specializes in selling a certain type of product?

The butcher shop sells meats and poultry, the market sells fruits and vegetables, and the flower shop sells flowers. Well, in the same way that these markets have organized into specialty stores, we can organize our Java classes into packages—groups of related classes with a specialized functionality.

Packages: Pre-Made Tools

Let's jump right in and create our own package—later we'll see how this applies to the Java API.

First, we need to create a new folder in our account. Right-click anywhere in the File Browser and choose New Folder. Name your new folder fruit (all lowercase).

Next, we want to create a couple of classes for our new package. Create the two new Java source files shown in the examples below. Add the following to what will be named AnOrange.java:

```java
package fruit;

public class AnOrange {
    public AnOrange (java.awt.Graphics g) {
        g.setColor(java.awt.Color.orange);
        g.fillOval(5, 5, 50, 50);
    }
}
```

The other Java file will be named Pear.java. Add the following code to this file:

```java
package fruit;

public class Pear {
    public Pear (java.awt.Graphics g) {
        g.setColor(java.awt.Color.green);
        g.fillOval(110, 5, 40, 75);
        g.fillOval(105, 30, 50, 50);
    }
}
```

Save and compile these files. Make sure you save them to the new fruit directory.

We've created a couple of new Classes that represent fruits. But we can't preview them because they're not applets. So, before we worry about how these classes work, let's go ahead and make a Java Applet that instantiates our new "fruit" objects. Be sure to save this Java Applet in your home directory (not in the fruit
directory, but in the directory where you created the fruit directory). Add `FruityApplet.java` to your Editor as shown below:

```java
import java.applet.*;
import java.awt.*;

public class FruityApplet extends Applet {
    public void paint(Graphics g) {
        fruit.AnOrange anOrangeObject = new fruit.AnOrange(g);
        fruit.Pear aPearObject = new fruit.Pear(g);
    }
}
```

In our fruit classes, we did something new in our "fruit" source files, we defined a package statement. It should look something like this:

```java
package fruit;
```

Later, when our FruityApplet instantiated some Orange and Pear objects, we told the JVM to look for those classes in the `fruit` package by typing this code. It looked like this:

```java
fruit.AnOrange anOrangeObject = new fruit.AnOrange(g);
```

The package name we defined has exactly the same name as the directory where we stored these fruit class definitions. We can use these classes through a package reference because they were each defined as part of the `fruit` package (`package fruit;`).

We know that if we put a bunch of classes into a directory, we can define all of those classes as parts of a package. But when we use those classes, why do we always have to type `fruit` before the class name? Well, let's see what happens when we don't. Change `FruityApplet.java` to look like this:

```java
import java.applet.*;
import java.awt.*;

public class FruityApplet extends Applet {
    public void paint(Graphics g) {
        fruit.AnOrange anOrangeObject = new fruit.AnOrange(g);
        fruit.Pear aPearObject = new fruit.Pear(g);
    }
}
```

When we Save/Compile, we get an error:
By now you may have noticed that the Java Compiler sometimes gives cryptic messages. This particular error is telling us that the compiler didn't know where to find the "AnOrange" class that we created, so it couldn't possibly compile our FruityApplet.

It might seem like the compiler has given us the same error twice--it didn't. The compiler first complained about our initial reference to the AnOrange class ("AnOrange anOrangeObject"), and then it complained about our reference to the AnOrange constructor ("new AnOrange(g)") in the second error.

You also might have noticed that the compiler did not complain about the Pear class and object constructor. Well, this may seem kind of lazy, but the compiler will often stop compiling once it has found 2 errors. Hey, don't ask me...I wouldn't have made it that way! :)

We can tell the compiler where to find classes in the fruit package by adding "fruit." before any reference to our fruit package's classes, but there should be an easier way, don't you think? Fortunately, there is. Change FruityApplet.java to look like this:

```java
import java.applet.*;
import java.awt.*;
import fruit.*;
public class FruityApplet extends Applet {
    public void paint(Graphics g) {
        AnOrange anOrangeObject = new AnOrange(g);
        Pear aPearObject = new Pear(g);
    }
}
```

Save/Compile. By adding that import statement, we no longer have to type "fruit." before referencing classes in our fruit package. The import statement allows us to tell the JVM to search through that package automatically, whenever a class definition is referenced--basically we are giving the compiler a hint.

**Using the Java API**

Earlier in the course, I mentioned that there are already a number of existing Java Classes that we can use to develop our own programs. The Java API (short for "Application Programming Interface") is a large collection of these pre-made classes organized into packages.

If we extend our earlier analogy comparing an open street market to Java Packages, we might think of an API as being the city block where those stores are located. Generally, an API is a collection of packages, each with some specialized purpose. The Java API is such a collection. The Java API is almost always the basis for Java software and other Java APIs, such as Oracle's JavaMail API, which helps in developing E-Mail related software in Java. That is, the JavaMail API uses (and often extends) many of the classes contained in the Java API.

To view the Java API, visit [http://docs.oracle.com/javase/7/docs/api/](http://docs.oracle.com/javase/7/docs/api/)

When we want to borrow a class from the Java API, we start by telling the JVM where, that is, in which package to look for the class. Edit the code in FruityApplet.java to look like this:
// we're taking out the import java.applet.*; statement
import java.awt.*;
import fruit.*;

public class FruityApplet extends Applet {
    public void paint(Graphics g) {
        AnOrange anOrangeObject = new AnOrange(g);
        Pear aPearObject = new Pear(g);
    }
}

Save/Compile. Whoops...we get an error! Check out the error from above source file:

OBSERVE:
/users/username/FruityApplet.java:5: Superclass Applet of class FruityApplet not found.
public class FruityApplet extends Applet {
^ 1 error

Yup--our class requires the java.applet.Applet class from the java.applet package. When the compiler doesn't know where to find java.applet.Applet, it gives us errors. Edit the code in FruityApplet.java to look like this:

CODE TO TYPE:
import java.awt.*;
import fruit.*;

public class FruityApplet extends java.applet.Applet {
    public void paint(Graphics g) {
        AnOrange anOrangeObject = new AnOrange(g);
        Pear aPearObject = new Pear(g);
    }
}

No big surprise here. We gave the compiler an explicit reference to the location of the java.applet.Applet class, so it knew where to look. Now let's undo what we just did and see what happens. Edit the code in FruityApplet.java so it looks like this:

CODE TO TYPE:
import java.awt.*;
import fruit.*;

public class FruityApplet extends java.applet.Applet {
    public void paint(Graphics g) {
        AnOrange anOrangeObject = new AnOrange(g);
        Pear aPearObject = new Pear(g);
    }
}

Save/Compile. That's interesting. We can import only one class from a package. We're able to do it in our example because we know that we only need the java.applet.Applet class from the java.applet package. Normally we'd just say import java.applet.* So why would we do it any other way? Well, we've done some
pretty in-depth research and discussion at OST on this subject. Basically, neither syntax is significantly better than the other. Just be aware that you can import classes using either syntax.

Where does it all come from?

You may have noticed that the names of the packages we’ve discussed here are defined using dot notation (i.e. something.somethingelse.whatever). In the case of packages, the dots represent directories. For instance, "import fruit.*;" points to classes in the "fruit" directory. (Those classes MUST define themselves as being part of the fruit package.)

In the same way, "import java.applet.*;" points to those classes stored in a directory named applet, which in turn is stored in a directory named java. For those of us familiar with Microsoft Windows filesystems (and don’t panic if you aren’t), it would look like this: "C:\java\applet\*" (in UNIX, it would look like /java/applet/*.)

OBSERVE:

```java
import java.applet.*;
```

This snippet of code tells our program to look in the directory named java with the sub-directory applet for any class.

The compiler also always imports a couple of extra things. The compiler always automatically looks for classes in the same directory as our class (as long as they don't belong to another package!) Check it out:

OBSERVE:

```java
import *;
```

Additionally, the compiler always automatically imports one of the Java API packages: java.lang. This package has classes that are at the core of the Java language:

OBSERVE:

```java
import java.lang.*;
```

Where in our filesystem are these Java API directories?

We can see that the Java API directories aren’t in our OST directory. We know that our fruit directory and package are in the same directory as our FruityApplet, so the JVM will start looking in the same directory as the class we’re running (in this case our FruityApplet.) But, there’s definitely no "java/lang" directory in our account, so where’s the compiler finding that?

The Compiler and JVM know by default where to find the directory structure containing the Java API (you tell the JVM when you install it, but that’s a conversation for another course). In any case, a functioning JVM (such as the one in our browser) has to be able to locate the packages contained in the Java API.

What classes are available in the Java API?

Javasoft, Oracle's Java Organization provides the list of all default (available to everybody) packages in several formats via the web. To see the complete documentation for these packages, click here.

So far, so good. Now you can create your own Java Packages/API's for use in your programs, and import packages from the Java API (or for that matter any package) into your programs. You’ve come a long way! I’ll see you in the next lesson where we’ll start exploring the Java API!!
Control Structures: Loops

Loops

In previous lessons, we coded things "one at a time". Java also allows us to do things in loops. Loops help you to build more efficient code, and will allow you to execute blocks of code repeatedly. In this lesson, we'll learn how to use loops to automate code.

Your First Loop

Let's build our first loop! We will do this by first creating code that does not use a loop so that we can understand how loops can be used to simplify code. Create a new source file named Loops.java.

CODE TO TYPE:

```java
import java.applet.*;
import java.awt.*;

public class Loops extends Applet {
    public void paint(Graphics g) {
        int leftLocation=10;
        int topLocation=10;

        g.drawString("the value is 1", leftLocation, topLocation+10);
        g.drawString("the value is 2", leftLocation, topLocation+20);
        g.drawString("the value is 3", leftLocation, topLocation+30);
        g.drawString("the value is 4", leftLocation, topLocation+40);
        g.drawString("the value is 5", leftLocation, topLocation+50);
    }
}
```

Save/Compile. Here we have created some simple code to display five strings. The strings are very similar. The only difference is the value that is displayed, and the location at which they are displayed. Each time the string is printed, a new value is added to the topLocation variable so that the values will be printed 10 pixels below the previous string.

Run this code to see for yourself what happens. Of course, first we'll need to create an HTML file to display our applet.

CODE TO TYPE:

```html
<HTML>
<BODY>

<APPLET CODE="Loops.class" HEIGHT="200" WIDTH="200"></APPLET>

</BODY>
</HTML>
```

Save this as Loops.html. Preview your applet. You should see five strings, each string displayed directly below the previous one.

As simple as the above code is, we can use loops to reduce the amount of typing. Modify Loops.java so that it looks like the file below.
import java.applet.*;
import java.awt.*;

public class Loops extends Applet {
    public void paint(Graphics g) {
        int leftLocation=10;
        int topLocation=10;
        int count=1;

        while (count <= 5) {
            g.drawString("the value is " + count, leftLocation, topLocation+(count*10));
            count = count+1;
        }
    }
}

A while loop is executed as long as the while loop's condition evaluates to true. In the case of this while loop, the condition will evaluate to true as long as the value of the count variable is less than or equal to 5. Each time this is the case, a single string is displayed, and the count variable is incremented by 1.

The first time the loop is executed, the value is 1 is printed to the screen since the initial value of count is set equal to 1. It is located 20 pixels down from the top of the applet. The second time the loop is executed, the value is 2 is printed to the screen since the value of count is now 2. It is located 30 pixels down from the top of the applet. The last time the loop is executed, the value is 5 is printed to the screen and count is set equal to 6. At this point the value of count is no longer less than or equal to 5 so the while loop condition is no longer true. The loop no longer executes.

If you are not familiar with loops this code may appear to be more complicated than our initial code. However, as you continue to create more complex programs you will find that loops are very useful, especially if you have multiple statements to execute repeatedly.

For Loops

For loops are similar to while loops in that they are used to execute a block of code repeatedly. Because you can specify the exact number of times the loop iterates, for loops are typically used when the number of iterations is known before entering the loop.

Let's take a closer look! Edit Loops.java so that it uses a for loop instead:

import java.applet.*;
import java.awt.*;

public class Loops extends Applet {
    public void paint(Graphics g) {
        int leftLocation=10;
        int topLocation=10;

        for (int count=1; count<=5; count=count+1) {
            g.drawString("the value is " + count, leftLocation, topLocation+(count*10));
        }
    }
}

Save/Compile this file, then Preview as usual. You should see the exact same result as you did before.
Just like a while loop, a for loop is executed as long as the loop's condition evaluates to true. The loop used above is known as a three-parameter loop control expression because it has three parts — an initializer, a condition, and a counting expression.

Initially, the count variable is set equal to 1. Since count variable is less than or equal to 5, this will be the value assigned to count the first time the loop is executed. Once all statements within the loop are complete, the value of count is increased by 1, and the condition of the loop is checked again. The loop will continue to iterate as long as the condition evaluates to true. In the case of this for loop, this means that the loop will be executed five times.

It is common to see syntax such as:

```
for (int count=1; count<=5; count++) {
    // statements here
}
```

Note: count++ is an equivalent to executing the statement count = count + 1. You will learn more about operators later in the course.

Loops are an integral part of programming. In the next lesson you will learn about if statements, another type of conditional statement. See you there!
Using Conditional Statements for Decision Making

In the last lesson, we saw how loops could be used to execute blocks of code repeatedly, using both while loops and for loops. However, we haven't gotten our code to make any decisions. Conditional statements can be used to perform different actions depending on whether a specified condition evaluates to true or false.

The If Statement

Recall our Cake example from previous lessons. Suppose we want to use the Cake constructor and the frostCake method to create three different types of cakes. The color of the cake and the message written on top of the cake will be determined by what occasion the cake is celebrating.

Let's start by modifying the Cake.java file:

```java
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;
    String theMessage;

    public Cake(int top, int left, String msg) {
        topLocation = top;
        leftLocation = left;
        theMessage = msg;
    }

    public void frostCake(Color myColor, Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 100, 50);
        g.setColor(Color.black);
        g.drawString(theMessage, leftLocation, topLocation + 25);
    }
}
```

Save/Compile this file, keeping the name Cake.java. Note the modifications to the file. They were made so that a message could be written on top of the cake.

Now let's work on a new file called CakeIf.java. This file will be used to create the three different types of cake.
Save/Compile this file using the name CakeIf.java.

Here we have assigned the value anniversary to the string variable cakeType. There is then a series of three if statements to check the value of the variable. The equality operator (==) is used to determine whether two values are the same.

The first condition is checked, but since cakeType is not the same as birthday, the code within the if statement is skipped, and the second if statement is checked. This time the condition evaluates to true! A new Cake is instantiated and the frostCake method is called. Finally the third condition is checked. Like the first condition, it evaluates to false and the code is not executed.

Note You will learn more about operators later in the course.

Run this code to see for yourself what happens.

Preview your applet. You should see a pink rectangular cake with the message Happy Anniversary!

Try changing the value of cakeType to "birthday", compile your code and view the applet again. What do you see now?
**If/Else Statements**

In reality, you would probably want to make more than just three types of cakes. Does this mean that you have to have an if statement for each and every occasion you wish to celebrate? What if you want to have a default cake that is made when you don’t have a statement that covers a certain occasion? This is where else statements can come in handy!

Modify CakeIf.java so that it looks like the file below.

```java
import java.applet.*;
import java.awt.*;

public class CakeIf extends Applet{
    String cakeType;
    Cake birthdayCake, anniversaryCake, graduationCake, anyCake;

    public void paint(Graphics g) {
        cakeType = "wedding";
        if (cakeType == "birthday") {
            birthdayCake = new Cake(20, 20, "Happy Birthday!");
            birthdayCake.frostCake(Color.red, g);
        } else if (cakeType == "anniversary") {
            anniversaryCake = new Cake (20, 20, "Happy Anniversary!");
            anniversaryCake.frostCake(Color.pink, g);
        } else if (cakeType == "graduation") {
            graduationCake = new Cake(20, 20, "Con-GRAD-ulations!");
            graduationCake.frostCake(Color.orange, g);
        } else {
            anyCake = new Cake(20, 20, "Yay!");
            anyCake.frostCake(Color.yellow, g);
        }
    }
}
```

Save/Compile this file, keeping the name CakeIf.java.

Note that we added else before the last two if statements from the previous example, and a final else clause at the end. Doing this makes it so that the if statements are grouped, and every time that the code is executed one of the statements will evaluate to true. Once a statement evaluates to true, none of the other cases are checked. In other words, if cakeType were set equal to birthday the first statement would evaluate to true, and its value would not be checked against either anniversary or graduation.

In the case of the above code, cakeType is set equal to wedding. Since all three if statements evaluate to false, the final else statement evaluates to true, and a yellow cake it created. You can think of it as a sort of catch-all for other occasions.

Preview your applet. You should see a yellow rectangular cake with the message Yay!

**Complex Test Clauses**

Sometimes, there are several possible true answers to the test that we want to run. For example, what if you were making a birthday cake for someone that hates the color red?

Modify CakeIf.java so that it looks like the file below:
import java.applet.*;
import java.awt.*;

public class CakeIf extends Applet{

    String cakeType, recipient;
    Cake birthdayCake, anniversaryCake, graduationCake, anyCake;

    public void paint(Graphics g) {
        cakeType = "birthday";
        recipient = "hatesRed";

        if (cakeType == "birthday" && recipient == "hatesRed") {
            birthdayCake = new Cake(20, 20, "Happy Birthday!");
            birthdayCake.frostCake(Color.blue, g);
        }
        else if (cakeType == "birthday") {
            birthdayCake = new Cake(20, 20, "Happy Birthday!");
            birthdayCake.frostCake(Color.red, g);
        }
        else if (cakeType == "anniversary") {
            anniversaryCake = new Cake(20, 20, "Happy Anniversary!");
            anniversaryCake.frostCake(Color.pink, g);
        }
        else if (cakeType == "graduation") {
            graduationCake = new Cake(20, 20, "Con-GRAD-ulations!");
            graduationCake.frostCake(Color.orange, g);
        }
        else {
            anyCake = new Cake(20, 20, "Yay!");
            anyCake.frostCake(Color.yellow, g);
        }
    }
}

Save/Compile this file using the name CakeIf.java.

In this case, we used the AND operator (&&) to check the value of both the cakeType and the recipient variables. Both conditions must be true in order to execute the code within that block.

Preview your applet. You should see a blue rectangular cake with the message Happy Birthday!

There is another operator known as the OR operator (||). It is used so that if either condition evaluates to true, code will be executed. Give it a try!
import java.applet.*;
import java.awt.*;

public class CakeIf extends Applet{

    String cakeType, recipient;
    Cake birthdayCake, anniversaryCake, graduationCake, anyCake;

    public void paint(Graphics g) {

        cakeType = "new baby";

        if (cakeType == "birthday" || cakeType == "new baby") {
            birthdayCake = new Cake(20, 20, "Happy Birthday!");
            birthdayCake.frostCake(Color.red, g);
        } else if (cakeType == "birthday" && recipient == "hatesRed") {
            birthdayCake = new Cake(20, 20, "Happy Birthday!");
            birthdayCake.frostCake(Color.blue, g);
        } else if (cakeType == "anniversary") {
            anniversaryCake = new Cake (20, 20, "Happy Anniversary!");
            anniversaryCake.frostCake(Color.pink, g);
        } else if (cakeType == "graduation") {
            graduationCake = new Cake(20, 20, "Con-GRAD-ulations!");
            graduationCake.frostCake(Color.orange, g);
        } else {
            anyCake = new Cake(20, 20, "Yay!");
            anyCake.frostCake(Color.yellow, g);
        }
    }
}

Save/Compile this file using the name CakeIf.java. Then Preview your applet. What do you see?

Now that you know how to use both loops and if statements, you can really start creating more complex code. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Operators

Operators are useful tools in Java, just as they are in mathematics. **Operators** allow you to add, subtract, divide, multiply, check inequalities, and such. In Java, operators work with numbers and strings.

Let's get a feel for operators in Java. There isn't much for you to do in this lesson, so you may want to minimize CodeRunner for now.

Click on any operator for a description and an example of it:

=  +  -  *  /  %=  %=  <=  >=  -=  +=

The **variables** we use in these examples will be named `x`, `y`, and `answer`. Every time you see one of these names, be aware that it could hold any value that the respective operator allows.

Operators in Java are divided into **three types** -- **binary**, **unary conditional**, and **unary**. Let's begin with **binary**.

**Binary Operators**

**Binary operators** require two values. In the first few examples below, there are actually 2 operators: an = (equals sign) and something else. The equals sign (=) is always evaluated last (more on operator precedence later), and sets the value of one thing to the value of something else. When you see = by itself and then some other operator later, we are actually setting the value of what comes before = to the value of the operator that comes afterward. (By the way, did you realize that you just learned the = operator? Nice.) Here we go!

**Assignment (equals)**

```
=  
```

This operator assigns values to variables. It works like this:

```
answer = x
```

```
answer: = x:  
```

[Do It]

**Addition**

```
+  
```

Addition works like this:

```
answer = x + y
```

**String Add** (concatenation) (Try typing words into last two boxes):

```
answer: = x:  + y:  
```

[Do It]

**Numerical Add** (Try typing numbers in the last two boxes):

```
answer: = x:  + y:  
```

[Do It]

**Subtraction**

```
-  
```

This operator can only be used with numbers or variables of numerical type. It works like this:

```
answer = x - y
```
Multiplication

* 

(Yes, that is an asterisk.) This operator can only be used with numbers or variables of numerical type. It works like this:

\[ \text{answer} = x \times y \]

Division

/ 

This operator can only be used with numbers or variables of numerical type. It works like this:

\[ \text{answer} = x / y \]

Modulus

% 

When you do long division, you often end up with a remainder. This operator calculates the remainder, and can only be used with numbers or variables of numerical type. It works like this:

\[ \text{answer} = x \% y \]

So, if you did "answer = 13 \% 5", answer would equal '3'.

Short-hand Addition

+= 

It works like this:

\[ x += y \]

This operator does the same thing as "x = x + y". It reads like this in English: "x becomes x plus y".

Short-hand Subtraction

-=
It works like this:

\[ x - = y \]

This operator does the same thing as "x = x - y". It reads like this in English: "x becomes x minus y".

**Short-hand Multiplication**

\[ * = \]

It works like this:

\[ x * = y \]

This operator does the same thing as "x = x * y". It reads like this in English: "x becomes x times y".

**Short-hand Division**

\[ /= \]

It works like this:

\[ x /= y \]

This operator does the same thing as "x = x / y". It reads like this in English: "x becomes x divided by y".

**Short-hand Modulus**

\[ %= \]

It works like this:

\[ x %= y \]

This operator does the same thing as "x = x % y". It reads like this in English: "x becomes x modulus y".

**Binary Conditional Operators**

Binary Conditional Operators also operate on two values, however the result is always **TRUE** or **FALSE**.

**Equality**

\[ == \]

This operator is used to determine whether two things are equal. If they are, it returns **TRUE**, if not it returns **FALSE**. This operator is used mainly in if and loop statements. It works like this:

\[ x == y \]
Try either numbers or strings below:

$x \quad == \quad y \quad$ returns \quad Do It

**Less Than**

$<$

This operator is used to determine whether one number is less than another. If it is, it returns TRUE. If not, it returns FALSE. It works like this:

$x < y$

Try some numbers below:

$x \quad < \quad y \quad$ returns \quad Do It

**Greater Than**

$>$

This operator is used to determine whether one number is greater than another. If it is, it returns TRUE. If not, it returns FALSE. It works like this:

$x > y$

Try some numbers below:

$x \quad > \quad y \quad$ returns \quad Do It

**Less Than or Equal To**

$\leq$

This operator is used to determine whether one number is less than or equal to another. If it is, it returns TRUE. If not, it returns FALSE. It works like this:

$x \leq y$

Try some numbers below:

$x \quad \leq \quad y \quad$ returns \quad Do It

**Greater Than or Equal To**

$\geq$

This operator is used to determine whether one number is greater than or equal to another. If it is, it returns TRUE. If not, it returns FALSE. It works like this:

$x \geq y$

Try some numbers below:
Unary Operators

Unary Operators operate on only one value.

Negation

- 
This operator can only be used with numbers or variables of numerical type. It makes a number negative. It works like this:

- \( x \)

Increment

++

This operator can only be used with numbers or variables of numerical type. It takes a value and adds one (1) to it. It works like this:

\( x++ \)

This operator does the same thing as \( x = x + 1; \) In English: \"x becomes x + 1\".

Decrement

--

This operator can only be used with numbers or variables of numerical type. It takes a value and subtracts one (1) from it. It works like this:

\( x-- \)

All things considered, operators aren't so tough. But it's important for you to know to the specific implementation of each of them in Java before you go on. Reread the operator explanations until you're familiar with them. Before long, they'll come to you naturally.

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Converting Data Types

So far the only thing we've printed to the screen are String values. In this lesson, we'll learn why that's the case. We'll also learn how to convert variable data types into other variable types.

Converting Integers to Strings

Let's create a new applet class to experiment with and learn about data conversion. Create a new Java source file just like the following:

```java
import java.applet.*;
import java.awt.*;

public class DataTypeConverter extends Applet {
    private int xPosition = 125;

    public void paint(Graphics g) {
        g.drawString(xPosition, 10, 10);
    }
}
```

Save/Compile this as DataTypeConverter.java. When you compile, you get an error. Take a look:

```
/Users/kendellw/DataTypeConverter.java:9: Incompatible type for method.
Can't convert int to java.text.AttributedCharacterIterator.
g.drawString(xPosition, 10, 10);
^ 1 error
```

`AttributedCharacterIterator`. What is that? Well, imagine you have several containers that hold boxes: a square container that holds a square box, a triangular container that holds a triangular box, and a circular container that holds a circular box. Then imagine that we have a new box that we want to put into each of these containers. To fit our new box into the square box, we need to fold the new box into a square. Before we can put that same new box into the triangular container, we need to unfold the box and proceed to fold the box into a triangular shape, right? (Just go with it.)

Java data types work in much the same way. By looking at the API (Application Programming Interface—a list of all of the functions and objects available in Java), you'll see that the `java.awt.Graphics.drawString()` method requires that we supply the value we want printed to the screen in the form of a String data type. However, the value we want printed, `xPosition`, is of type `int`. Therefore, before we can print out the value of `xPosition` with the `drawString()` method, we must convert the `int` data type to a `String` data type. In the previous example, when we didn't convert `xPosition`, the `AttributedCharacterIterator` class realized that the value passed was not of data type `String`, so it gave us an error.

Update your `DataTypeConverter.java` source file as shown:
import java.applet.*;
import java.awt.*;

public class DataTypeConverter extends Applet {
    private int xPosition = 125;
    private String xPositionString;

    public void paint(Graphics g) {
        xPositionString = Integer.toString(xPosition);
        g.drawString(xPositionString, 10, 10);
    }
}

Save/Compile. Create an html file to view your applet. Preview your applet. Now the applet works. To make the example applet work, we needed to convert the int value of xPosition to a String data type before we passed it as an argument to the drawString() method. So now we know that if a method receives a parameter that is not of an acceptable type, the method won’t work. This is really inconvenient if the variables we have are not compatible with the methods we need to use. To solve this problem, we convert the data type into something the method can use. To convert integer data types, we must use a method that has been provided to us within the java.lang.Integer class. As we move on, you’ll see that the java.lang.Integer class provide methods for converting integers into other data types, and the java.lang package provides classes for converting other primitive data types as well.

Using Data Types

Let’s try using various operators on a few String data types. Modify your DataTypeConverter.java source file as shown:

import java.applet.*;
import java.awt.*;

public class DataTypeConverter extends Applet {
    private int xPosition = 125;
    private String somethingToAdd = "5.12345";

    public void paint(Graphics g) {
        xPositionString = Integer.toString(xPosition);
        g.drawString(somethingToAdd + xPosition, 10, 10);
    }
}

Save/Compile. Preview the Applet. When you preview the code, you’ll see that rather than arithmetically adding our two "numbers" together (which would have resulted in 130.12345, right?), it combined our two values as one big String (5.12345125). Adding these two String values together is called concatenation. This is unique to the String class. If you "add" two String data types together, the concatenation of the two Strings being added together is returned in the form of a new String. This type of concatenation is unique String class.

Note You could also use g.drawString(somethingToAdd + xPosition, 10, 10) with the same result, because using the plus sign automatically converts primitives to Strings if they follow a String value.

Suppose we actually want to add our two String data types together numerically. That is, we actually want to figure out the sum of the two Strings. What would we need to do? The Strings would need to be converted to a primitive data type that could be added together. We’d need to convert the String variable somethingToAdd into a numeric value. However, if we check the Java API documentation, you’ll notice that there is no java.lang.String.toFloat() method. So how would we do this? Modify your DataTypeConverter.java source file as shown:
import java.applet.*;
import java.awt.*;
public class DataTypeConverter extends Applet {
    private int xPosition = 125;
    private String somethingToAdd = "5.12345";
    public void paint(Graphics g) {
        double xPositionDoubleValue = new Integer(xPosition).doubleValue();
        double somethingToAddDoubleValue = new Double(somethingToAdd).doubleValue();
        double answerDoubleValue = somethingToAddDoubleValue + xPositionDoubleValue;
        g.drawString("Before Conversion: " + Integer.toString(xPosition) + " + " + somethingToAdd + " = " + Integer.toString(xPosition) + somethingToAdd, 10, 10);
        g.drawString("After Data Conversion: " + new Double(xPositionDoubleValue).toString() + " + " + new Double(somethingToAddDoubleValue).toString() + " = " + new Double(answerDoubleValue).toString(), 10, 50);
    }
}

Once again, Save/Compile your applet and Preview it. If you take a closer look at the code we've added, you'll see that we used double data types to store our data. This makes sense because we'll use decimal places when we finish adding our values together. But the data types we started with are of type int and String. So, what do we need to do in order to work with these two different data types? Right! We need to convert our data from int and String types into double values. Good call! To accomplish this task, we first instantiate a new Double object from the java.lang.Double class. If you take a look at the Java API, you'll see that the java.lang.Double class provides a constructor that takes in a String object, and creates a Double object that stores the String data as double data. Once the new Double object is created, we're able to use the doubleValue() method to return our String data as a double value.

Once we've arithmetically added our new double values, we can then print them to our applet by converting them to a String value. Be aware, java.lang.Double objects are not the same as a double primitive! Rather, the java.lang.Double class was written to work with double primitives. All these classes (java.lang.Double, java.lang.Integer, java.lang.Float, and so on) were created to provide a wrapper class for primitive data types. These wrapper classes provide data type conversion and other methods for primitive data types. So, in our example, when we instantiated our java.lang.Double objects, we used the methods provided in the wrapper class to output the primitive data types these classes "wrap."

In the last example, we chose to use double values because we wanted to preserve the decimal places in our solution. We could have chosen float value, which also supports decimal places just as easily, but without as much precision. Let's try it and see what happens! Modify your DataTypeConverter.java source below file by adding the blue code as shown:
CODE TO TYPE: Modify DataTypeConverter.java

```java
import java.applet.*;
import java.awt.*;

public class DataTypeConverter extends Applet {
    private int xPosition = 125;
    private String somethingToAdd = "5.12345";

    public void paint(Graphics g) {
        float xPosValue = new Integer(xPosition).floatValue();
        float addValue = new Float(somethingToAdd).floatValue();
        float answerValue = addValue + xPosValue;

        g.drawString("Before Conversion: " + Integer.toString(xPosition) + " + " + somethingToAdd + " = " + Integer.toString(xPosition) + somethingToAdd, 10, 10);
        g.drawString("After Data Conversion: " + new Float(xPosValue).toString() + " + " + new Float(addValue).toString() + " = " + new Float(answerValue).toString(), 10, 50);
    }
}
```

Save/Compile and Preview. It looks identical to what we did before right? Well, pretty much, but if you look closely at the results, you'll notice that the float values didn't add up correctly. In fact, the result is .00001 off of the mathematically correct result. We know that the float data type doesn't store numbers with as much accuracy as the double data type does. When the JVM performs a mathematical operation, the answer is returned according to the number of significant digits, as defined by that data type--it rounds the answer. Sometimes, we don't get a precise answer when using inappropriate data types, so it's extremely important to choose the correct data type for the information we need to store. However, since a double data type is more accurate, it takes up more space in computer memory than a float. We have to take into consideration not only the precision of the data we need, but also how much of the system's resources we're using.

If we had coded this example properly, instead of creating an example full of incorrect data types, `xPosition` and `somethingToAdd` would have been defined from the beginning as double data types because both values would eventually be converted to double values before we could work with them.

More Static Methods--'Sticky Code'

If you've been paying close attention (and I know you have), you know that when we were converting Integer data types to other data types, we didn't have to `instantiate` a new Integer object. Why is this you ask? Well, instead of instantiating a new Integer object, we were able to call `java.lang.Integer.toString()` directly, and pass our Integer value to it as a parameter.

Sometimes classes define **static methods** that don't require any information about specific instantiated objects of that class type. That is, static methods are available globally to any class that knows where to find them, regardless of whether the class containing the method has been instantiated as an object. In our example, we didn't have to instantiate a new Integer object to use the static `toString()` method that the Integer class defines. Let's try an example to help you understand the concept of static methods a bit better. Modify your `DataTypeConverter.java` source file as shown:
import java.applet.*;
import java.awt.*;

public class DataTypeConverter extends Applet {
    private int xPosition = 125;
    private String somethingToAdd = "5.12345";

    public void paint(Graphics g) {
        float xPositionFloatValue = new Integer(xPosition).floatValue();
        float somethingToAddFloatValue = new Float(somethingToAdd).floatValue();
        float answerFloatValue = somethingToAddFloatValue + xPositionFloatValue;

        g.drawString("Before Conversion: " + Integer.toString(xPosition) + " + " + somethingToAdd + " = " + Integer.toString(xPosition) + somethingToAdd, 10, 10);

        g.drawString("After Data Conversion: " + Float.toString(xPositionFloatValue) + " + " + Float.toString(somethingToAddFloatValue) + " = " + Float.toString(answerFloatValue), 10, 50);
    }
}

Save/Compile and Preview. Take a look at the code. Pretty cool, huh? Now, let’s try to convert between a String and float data types. Modify your DataTypeConverter.java source file again:

import java.applet.*;
import java.awt.*;

public class DataTypeConverter extends Applet {
    private int xPosition = 125;
    private String somethingToAdd = "5.12345";

    public void paint(Graphics g) {
        float xPositionFloatValue = new Integer(xPosition).floatValue();
        float somethingToAddFloatValue = Float.parseFloat(somethingToAdd);
        float answerFloatValue = somethingToAddFloatValue + xPositionFloatValue;

        g.drawString("Before Conversion: " + Integer.toString(xPosition) + " + " + somethingToAdd + " = " + Integer.toString(xPosition) + somethingToAdd, 10, 10);

        g.drawString("After Data Conversion: " + Float.toString(xPositionFloatValue) + " + " + Float.toString(somethingToAddFloatValue) + " = " + Float.toString(answerFloatValue), 10, 50);
    }
}

Save/Compile and preview this applet. In this example, we’re trying to convert somethingToAdd and xPosition into float values, then arithmetically add them together, and finally print out the results. Take a closer look at the source code. If our earlier examples are an indication, this change seems like it should work. But, when you compile and preview this time around, you get an error! Check it out:
There are two reasons for this error. If you check the Java API documentation, you'll see that the java.lang.Float class doesn't define a method named `floatValue()` that supports String data type parameters. In fact, the `floatValue()` method doesn't take any parameters at all. Let's fix that and try something a little different. Modify your `DataTypeConverter.java` source file as shown:

```java
import java.applet.*;
import java.awt.*;

public class DataTypeConverter extends Applet {
    private int xPosition = 125;
    private String somethingToAdd = "5.12345";

    public void paint(Graphics g) {
        float xPositionFloatValue = new Integer(xPosition).floatValue();
        float somethingToAddFloatValue = Float.floatValue();
        float answerFloatValue = somethingToAddFloatValue + xPositionFloatValue;

        g.drawString("Before Conversion: " + Integer.toString(xPosition) + " + " + somethingToAdd + " = " + Integer.toString(xPosition) + somethingToAdd, 10, 10);
        g.drawString("After Data Conversion: " + Float.toString(xPositionFloatValue) + " + " + Float.toString(somethingToAddFloatValue) + " = " + Float.toString(answerFloatValue), 10, 50);
    }
}
```

Save/Compile again. When we try to compile, we still get an error! Take a look:

```
OBSERVE:
/users/kendellw/DataTypeConverter.java:10: Can't make static reference to method float floatValue() in class java.lang.Float.
float somethingToAddFloatValue = Float.floatValue();
1 error
```

There's that word "static" again, which brings us to the second reason that the Java compiler returns an error: when the Java compiler tells us that we "can't make reference" to something, it means that the thing we tried to reference didn't have the keyword "static" applied to it in the source code. We can verify this in our example by checking the Java API documentation. We see that `Float.floatValue()` method is not a static method. While we're looking through the API documentation, we can also see that the `Float.toString()` method is a static method.

### So, when to convert a static reference to a method?

By getting friendly and familiar with the Java API! The more you program in Java, the more you'll access the Java API documentation for answers to your burning questions. Eventually, you'll remember the API methods that you use most often. Familiarity with the API makes a Java programmer much more efficient.

Let's get back to our example and change the modified `DataTypeConverter.java` to fix the errors that were returned. We want our code to turn the String variable `somethingToAdd` into a float value. Instead of trying to
use a static method to accomplish this task, let’s create a new Float object. Take a look at the Java API documentation. The java.lang.Float class has a constructor that takes in a String object and converts the String object into a float value. After the Float object is created, we can use the floatValue() method to get the float value out of the Float object. Take a few minutes to figure out what you need to do to fix the errors. Now edit the DataTypeConverter.java source file as shown below:

```java
import java.applet.*;
import java.awt.*;

public class DataTypeConverter extends Applet {
    private int xPosition = 125;
    private String somethingToAdd = "5.12345";

    public void paint(Graphics g) {
        float xPositionFloatValue = new Integer(xPosition).floatValue();
        float somethingToAddFloatValue = new Float(somethingToAdd).floatValue();
        float answerFloatValue = somethingToAddFloatValue + xPositionFloatValue;

        g.drawString("Before Conversion: " + Integer.toString(xPosition) + " + " + somethingToAdd + " = " + Integer.toString(xPosition) + somethingToAdd, 10, 10);

        g.drawString("After Data Conversion: " + Float.toString(xPositionFloatValue) + " + " + Float.toString(somethingToAddFloatValue) + " = " + Float.toString(answerFloatValue), 10, 50);
    }
}
```

Did your solution match up with mine?

In the next lesson, we’ll go over Arrays; a class that allows us to store more than one piece of data. See you there.

Copyright © 1998-2014 O’Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Converting Between Data Types

In this lesson we'll learn how to use arrays; variables that can hold multiple values or Objects.

**The Array**

An array is like a filing cabinet or a rolodex; a place where you organize and store data. For example, here's a sample array called names (it already contains a few names):

An array called names:

The beauty of an array is in the way we access the slots within it. For instance, if we wanted to know what name was in the third slot of this array, we would ask for:

<table>
<thead>
<tr>
<th>OBSERVE:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>names[3]</code></td>
</tr>
</tbody>
</table>

The value returned would be "Tony."

If we wanted to change the name in the fourth slot to Josh, we would do this:

<table>
<thead>
<tr>
<th>OBSERVE:</th>
</tr>
</thead>
</table>

The array then becomes:

| 0 scott | 1 kendell | 2 Trish | 3 Tony | 4 Josh | 5 Debra | 6 Curt |

Of course, the array above is only a graphical representation of an array, but it's the picture I want you to have in your mind when we discuss arrays.

**Creating Arrays in Java**

Let's create a new applet we can use to play with some Arrays, and at the same time, we'll go ahead and instantiate a new Array. Create a new Java source file. Feel free to substitute the names of your friends:
We initialized a new variable: names. We did not instantiate any new objects. By applying those [] (brackets) to our data type declaration, we told the JVM that we're creating a new array to hold String values. In the case of this array, we're giving it seven placeholders, otherwise known as indices.

Next, in the paint() method, we filled our array with some String values. In each case, we provided a different index for each value. Our indices never got larger than 6. But we started our indices with 0 (zero), and we used all of our placeholders. In Java, all array indices start with 0, so the last index that we can use would be:

\[
\text{number of placeholders} - 1
\]

Get it? Cool.

Okay, now let's do something with our new array. Think about which type of control structure do you think would take the best advantage of a sequential set of numbers. Add the for loop (in bold) to UsingArrays:
import java.applet.*;
import java.awt.*;

public class UsingArrays extends Applet {
    public String[] names = new String[7];

    public void init() {
    }

    public void paint(Graphics g) {
        names[0] = "scott";
        names[1] = "kendell";
        names[2] = "Trish";
        names[3] = "Tony";
        names[4] = "Mike";
        names[5] = "Debra";
        names[6] = "Curt";

        for (int x = 0; x < 7; x++) {
            g.drawString(names[x], 10, 20 * x + 10);
        }
    }
}

Something Different

Let's refine what we've already done. Continue to feel free to substitute the names of your friends:

import java.applet.*;
import java.awt.*;

public class UsingArrays extends Applet {

    public String[] names = {"scott", "kendell", "Trish", "Tony", "Mike", "Debra ", "Curt"};

    public void init() {
    }

    public void paint(Graphics g) {
        for (int x = 0; x < 7; x++) {
            g.drawString(names[x], 10, 20 * x + 10);
        }
    }
}

In this example we instantiated our String values in a different way: using a list. A Java list is a set of data, separated within {} curly brackets. Our array takes each element on this list, and places it in one of our array's placeholders. The list syntax is sometimes more convenient than assigning values in the various positions of our array individually—but not always. You'll know which to use when you need it. Use the force. But one day, when you're writing your own programs in Java, you'll probably have a situation where you don't know exactly how many indices your array will have. When that day comes, you will have Java arrays at your
 disposal, which allow you to *query* their size. Make the following edit to *UsingArrays*:

```java
import java.applet.*;
import java.awt.*;

public class UsingArrays extends Applet {
    public String[] names = {"scott", "kendell", "Trish", "Tony", "Mike", "Debra ", "Curt"};

    public void init() {
    }

    public void paint(Graphics g) {
        for (int x = 0; x < names.length; x++) {
            g.drawString(names[x],10,20 * x + 10);
        }
    }
}
```

Save/Compile and Preview. When Java creates an array, it actually *instantiates an Array Object*. One of the variable values in this Array Object is the *length*: a value representing the *number of indices* in the array. So, if we don’t know the length of an array, we can always query that array to find out. Alright then, onward!

*Copyright © 1998-2014 O'Reilly Media, Inc.*

*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See [http://creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode) for more information.*
Getting 'Object Oriented'

Suppose we were making a Java Applet that represented a Waitress. Of course, our Waitress is going to serve food to our customers. In our case, our Waitress will be serving Cake and Pie.

I think it would make the most sense for us to create Java Objects for the Cakes and Pies that we serve to our customers. Let's start by modifying good old Cake.java:

```java
import java.awt.*;

public class Cake {
    int topLocation;
    int leftLocation;
    Color myColor;

    public Cake(int top, int left, Color thisColor) {
        topLocation = top;
        leftLocation = left;
        myColor = thisColor;
    }

    public void bake(Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
    }
}
```

Save/Compile this, keeping the name Cake.java

Create a new Java file for the Pie Class as shown:

```java
import java.awt.*;

public class Pie {
    int topLocation;
    int leftLocation;
    Color myColor;

    public Pie(int top, int left, Color thisColor) {
        topLocation = top;
        leftLocation = left;
        myColor = thisColor;
    }

    public void bake(Graphics g) {
        g.setColor(myColor);
        g.fillOval(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
    }
}
```
Save/Compile this as, you guessed it, Pie.java.

I don't think there's anything about these new objects that will catch you by surprise, but did you notice that both our Cake and Pie objects have a bake method? Let's create a Waitress Applet to bake our Cakes and Pies. Create a new Java file and add the following code:

```
import java.applet.*;
import java.awt./*;

public class Waitress extends Applet {

    Cake myCake;
    Pie myPie;

    public void paint(Graphics g) {
        // instantiate the objects, giving a position and Color
        myCake = new Cake(0, 0, Color.yellow);
        myPie = new Pie(0, 100, Color.blue);

        // call the bake() in the Cake and Pie objects
        myCake.bake(g);
        myPie.bake(g);
    }
}
```

Save/Compile Waitress.java and create an HTML page for it. Try it out--you'll see that our "cake" and "pie" were painted to the applet.

Let's think about our objects and our Waitress Applet for a moment. Our objects (Cake and Pie) each have a method named bake() that takes one parameter--a java.awt.Graphics Object. And, of course, real Cake and Pie are both foods.

You know, if we apply Java's Object Orientation capabilities to our applet, it would make sense for both of our "Food Objects" to extend a Food class. So let's create a Food class, and change our Cake and Pie objects to make them extend Food. They'll both be Food objects, right? Go ahead and create the Food Class:

```
import java.awt.*;

public class Food {

    public Food() {
    }

    public void bake(Graphics g) {
    }
}
```

Modify the Cake class as shown:
Modify the Pie class as shown:

```java
import java.awt.*;

public class Pie extends Food {
    int topLocation;
    int leftLocation;
    Color myColor;

    public Pie(int top, int left, Color thisColor) {
        topLocation = top;
        leftLocation = left;
        myColor = thisColor;
    }

    public void bake(Graphics g) {
        g.setColor(myColor);
        g.fillOval(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
    }
}
```

Make sure that you've Save/Compiled all three files.

By extending Food we make sure that even if our Cake and Pie objects don't define a bake() method, they will definitely inherit the bake() method from the Food class.

To get a better understanding of how that's useful, let's call on a couple of old friends, the loop and the array, and see how they might help our Waitress. Edit the Waitress Applet, like this:
When we Save/Compile this, we see that our Waitress Applet does exactly the same thing as it did before. Let's take a look:

- Instead of defining variables for our Cake and Pie objects, the first thing we did was create an **Array of Food objects**.
- Our Cake and Pie object **extend** Food. Although they add some of their own features specific to being a Cake or a Pie, they are indeed still Food objects. Since they are Food objects, when we instantiate our Cake and Pie objects, we can store them in an **Array of Food objects**.
- Since we put our Cake and Pie objects into an **Array of Food objects**, we were able to use a **For Loop** to iterate through each item and call their **bake()** methods.
- Both our Cake and Pie classes defined the **bake()** method (over-riding the bake() in Food.) We put all kinds of Food objects into our Array. But because our Food class defined the **bake()** method, when we actually called the **bake()** method, we didn't have to tell the JVM what kind of Food we were serving, just that it was a Food object (in a Food Array). The JVM called the **bake()** method in our Cake and Pie object, without considering the types of Food they were.

**What now?**

Good question. So far we've learned some neat tricks to use with **Object-Oriented Design (OO)**. Next, we'll work with a fairly complicated example to learn about **Abstract Classes**. Make sure you save all of the files from this lesson for use in the next lesson! See you there...

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See [http://creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode) for more information.
Advanced Object Orientation - Abstracts

Thinking in the Abstract

Let's start this lesson by testing the limits of Object-Oriented Design. First we'll create a new type of Food. Create the Donut class as shown:

```java
import java.awt.*;
public class Donut extends Food {
    public Donut() {
    }
}
```

Save/Compile it. Okay, now let's modify our Waitress to bake() a Donut. Edit the Waitress Applet so it looks like this:

```java
import java.applet.*;
import java.awt.*;
public class Waitress extends Applet {
    public Food[] myMeal = new Food[3];
    public void paint(Graphics g) {
        // instantiate the objects, giving a position and Color,
        // store them in the myMeal array
        myMeal[0] = new Cake(0, 0, Color.yellow);
        myMeal[1] = new Pie(0, 100, Color.blue);
        myMeal[2] = new Donut();
        // call the bake() in ALL of the Cake and Pie objects
        for (int x = 0; x < myMeal.length; x++) {
            myMeal[x].bake(g);
        }
    }
}
```

With that code, we increased the size of our Array, and then instantiated a Donut object and stored it. Adding more Food objects isn't so tough, huh? But, hold on--there is one problem. Our Donut class did not over-ride (define) the Food class’s bake() method.

When our Waitress called our Donut's bake() method, it actually called the Food class's bake() method that Donut inherited. Food doesn't do anything with the bake() method.

Our Object-Oriented design ensures that when we call the bake() method of a Food object, we don’t run into problems. That's good, but we also want to require Food objects to implement certain methods. Let's modify the Food class:
import java.awt.*;

public abstract class Food {
    public Food() {
    }
    public abstract void bake(Graphics g);
}

Save/Compile this new abstract class, then let's experiment with our Donut. Open and Compile the Donut class:

import java.awt.*;
public class Donut extends Food {
    public Donut() {
    }
}

You might have noticed that our Food.bake() method no longer defines a body (it no longer has {} curly brackets.) Because of the changes we made to Food, we get an error when we compile the Donut class now. Here's the error we get from our example:

Observe:

/users/username/Donut.java:1: class Donut must be declared abstract.
It does not define void bake(java.awt.Graphics) from class Food.

public class Donut extends Food {
  ^
1 error

If we think back to our Food class, we placed the keyword abstract in our class definition. Take a look:

Observe:

import java.awt.*;

public abstract class Food {
    public Food() {
    }
    public abstract void bake(Graphics g);
}

Here's a quick example of the way this concept of abstract may work in real life. Imagine someone asked us to describe a candle. We might describe it as some wax with a wick sticking out of it that you can ignite so that it gives off light.

That pretty much describes any candle, but we know there are many different kinds of candles, with various types of wax, wicks, scents, colors, and other qualities. When we described the candle, we described it in the abstract sense; it was simply a generalization of the features of any candle.
The Java keyword `abstract` works in essentially the same way. When we use the `abstract` keyword before a method, we're telling the compiler that a class that extends this class MUST define this method.

Let's fix our Donut class. Modify the Donut class code so it looks like this:

```java
import java.awt.*;

public class Donut extends Food {
    public Donut() {
    }

    public void bake(Graphics g) {
        g.setColor(Color.pink);
        g.fillOval(0, 100, 50, 50);
        g.setColor(Color.gray);
        g.fillOval(15, 115, 20, 20);
    }
}
```

Declaring our `Food` class abstract forces our `Donut` class to define the `bake()` method with its own instructions. As a result, our Waitress Applet can have some assurance that this class is going to behave like a `Food`.

**Not Entirely Abstract**

Still, not all methods in an `abstract` class have to be `abstract`. Let's test this claim. Modify the Food class as shown:

```java
import java.awt.*;

public abstract class Food {
    public Food() {
    }

    public abstract void bake(Graphics g);

    public String whatAmI() {
        return "I am a Food";
    }
}
```

We just added a method to our `abstract` Food class. But we did not make this new method `abstract`. So it stands to reason that this method would be inherited and usable by all sub-classes, right? Let's find out using our Waitress Applet. Modify the Waitress Applet as shown:
import java.applet.*;
import java.awt.*;

public class Waitress extends Applet {

    public Food[] myMeal = new Food[3];

    public void paint(Graphics g) {
        // instantiate the objects, giving a position and Color,
        // store them in the myMeal array
        myMeal[0] = new Cake(0, 0, Color.yellow);
        myMeal[1] = new Pie(0, 100, Color.blue);
        myMeal[2] = new Donut();

        // call the bake() in ALL of the Cake and Pie objects
        for (int x = 0; x < myMeal.length; x++) {
            myMeal[x].bake(g);
        }

        g.drawString(myMeal[2].whatAmI(), 125, 125);
    }
}

Sure enough, our Donut class inherits the whatAmI() method from Food. When we paint the results to the applet, “I am a Food” is painted to the screen.

Not all of the methods in our abstract classes have to be abstract! If we don’t declare a method abstract, the sub-class will inherit the functionality of that method.

Variables - Abstract or Not

Not. When you think about it, there’s no reason to declare abstract variables. If you define them normally in the abstract class, then they will be defined in the sub-classes automatically and won’t need additional initialization (not to mention, you’ll write less code.)

If all of this abstract business isn’t perfectly clear to you just yet, don’t worry. You’ll be exposed to this concept again later and become more comfortable with it. In the next lesson we’re going to explore Interfaces. Interfaces work a lot like abstract classes, but have their own capabilities. See you there!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Advanced Object Orientation - Interfaces

Multiple Inheritance

So, within the Waitress Applet and Food objects we created in the last lesson, we created an *Abstract Food class* to require its *sub-classes* (Cake, Pie, and Donut) to define the *bake()* method. That way, our Waitress can bake all kinds of *Food* objects using simplified code. But wait! There's a problem with our model of a Waitress. After our customers are finished eating their Cake, Pie, and Donuts, they want our Waitress to bring them a their *Bill* (invoice). Right now our Waitress only knows how to deal with *Food* objects. But a *Bill* is not a *Food*. It's a *Payment* object.

Our *Bill* is a *Payment* object. So naturally, our *Bill* object should extend a *Payment* class. Let's create an *Abstract Payment class* that describes what our *Payment* objects should be like. Create the *Payment* class:

```
import java.awt.*;

public abstract class Payment {
    public abstract void printItem(Graphics g);
}
```

Pretty straight forward, right? We define an *Abstract* class that requires its *sub-classes* to *implement* the *printItem()* method. Let's go ahead and create our *Bill* class:

```
import java.awt.*;

public class Bill extends Payment {
    public Bill () {
        //a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
    }
}
```

When we Save/Compile this, we don't get any errors, but we still have a problem. We can't simply store our *Bill* object in our Waitress's *Food Array* because it's not a sub-class of *Food*.

Wouldn't it be nice if our *Bill* class could extend both *Payment AND Food*? Let's try it. Modify the Bill class so it looks like this:

```
import java.awt.*;

public class Bill extends Payment {
    public Bill () {
        //a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
    }
}
```
import java.awt.*;

public class Bill extends Payment, Food {

    public Bill () {
        // a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
        g.drawLine(110, 140, 140, 130);
    }
}

We learned that a class can only extend one class. Not surprisingly, we got an error when we Save/Compiled. Check out the error from above example:

```
/users/username/Bill.java:3: Multiple inheritance is not supported.
    public class Bill extends Payment, Food {
       ^
/users/username/Bill.java:3: '{' expected.
    public class Bill extends Payment, Food {
       ^
2 errors
```

Well, we've demonstrated that we can't extend more than one class at a time. So, how do we handle this situation? It would be nice to store our Food and Payment objects in one big array. That way, our Waitress could deliver all of our object types at once.

Let's consider the reason we can't extend multiple data types. Suppose we added a non-abstract method named deliver() to both our Food and Payment classes. Don't actually modify your Food and Payment classes (it won't work anyway). Just check out the updated Food class:

```
import java.awt.*;

public abstract class Food {

    public Food() {
    }

    public abstract void bake(Graphics g);

    public void deliver(Graphics g) {
        g.fillRect(100, 100, 50, 50);
    }
}
```

Take a look at the updated Payment class:
In these imaginary classes, we created a new method: `deliver()`. Now, let's imagine what would happen if we were allowed to extend both of these classes. Look over the `Bill` class:

```java
import java.awt.*;

public class Bill extends Payment, Food {
    public Bill () {
        //a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        deliver(g);
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
    }
}
```

Think about this situation. Suppose our `Bill` class could extend both `Food` and `Payment`. What would happen if `Bill` called the `deliver()` method?

By extending both `Food` and `Payment`, our `Bill` class inherits `Food.deliver()` (paints a rectangle) and `Payment.deliver()` (paints a Circle). So, when our `Bill` class calls this inherited method, does it paint a Rectangle, a Circle, both, or neither?

In C++, programmers have to consider this potential problem because C++ classes can have multiple inheritance. In Java, that problem can't exist because a class can not extend more than one class.

### Using Interfaces for Multiple-Inheritance Style Data-Typing

It's convenient that we don't have to worry about multiple-inheritance issues in Java. But multiple-inheritance offers some advantages that we might like to use. It would be great if our `Food` and `Payment` classes were completely Abstract, that is, if they defined only Abstract methods and no methods were actually inherited by sub-classes.

We need Java Interfaces. Interfaces are essentially pure abstract classes. In fact, we won't even have to use the Abstract keyword, because the compiler already knows that they're Abstract. Let's experiment with a Java interface. Create the `DeliverableItem` Interface as shown:

```java
import java.awt.*;

public abstract class Payment {
    public abstract void printItem(Graphics g);

    public void deliver(Graphics g) {
        g.fillOval(100, 100, 50, 50);
    }
}
```
Save/Compile that as DeliverableItem.java (use the same file-naming scheme that we used for all other types of Java classes.) Now, let's modify our Bill class to take advantage of this new interface. Revise your code for the Bill class so it looks like this:

CODE TO TYPE:

import java.awt.*;

public interface DeliverableItem {
    public void deliver(Graphics g);
}

public class Bill extends Payment implements DeliverableItem {
    public Bill () {
        // a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
    }
}

When we Save/Compile, we get an error:

OBSERVE:

/users/username/Bill.java:4: class Bill must be declared abstract. It does not define void deliver(java.awt.Graphics) from interface DeliverableItem.
  public class Bill extends Payment implements DeliverableItem {
^ 1 error

This error resembles an error we got in the last lesson. DeliverableItem is really no different from an Abstract Class, except that an Interface can only define abstract methods (there's no danger of inherited methods interfering with one another.)

By implementing one or more Interface(s), we can create objects that can function as multiple data types. In our case, we didn't define the deliver() method that our Interface required, so we got an error. Let's fix our Bill class:

CODE TO TYPE:

import java.awt.*;

public interface DeliverableItem {
    public void deliver(Graphics g);
}

public class Bill extends Payment implements DeliverableItem {
    public Bill () {
        // a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
    }
}

public class Bill extends Payment implements DeliverableItem {
    public Bill () {
        // a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
    }
}

When we Save/Compile, we get an error:

OBSERVE:

/users/username/Bill.java:4: class Bill must be declared abstract. It does not define void deliver(java.awt.Graphics) from interface DeliverableItem.
  public class Bill extends Payment implements DeliverableItem {
^ 1 error

This error resembles an error we got in the last lesson. DeliverableItem is really no different from an Abstract Class, except that an Interface can only define abstract methods (there's no danger of inherited methods interfering with one another.)

By implementing one or more Interface(s), we can create objects that can function as multiple data types. In our case, we didn't define the deliver() method that our Interface required, so we got an error. Let's fix our Bill class:

CODE TO TYPE:

import java.awt.*;

public interface DeliverableItem {
    public void deliver(Graphics g);
}

public class Bill extends Payment implements DeliverableItem {
    public Bill () {
        // a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
    }
}

When we Save/Compile, we get an error:

OBSERVE:

/users/username/Bill.java:4: class Bill must be declared abstract. It does not define void deliver(java.awt.Graphics) from interface DeliverableItem.
  public class Bill extends Payment implements DeliverableItem {
^ 1 error

This error resembles an error we got in the last lesson. DeliverableItem is really no different from an Abstract Class, except that an Interface can only define abstract methods (there's no danger of inherited methods interfering with one another.)

By implementing one or more Interface(s), we can create objects that can function as multiple data types. In our case, we didn't define the deliver() method that our Interface required, so we got an error. Let's fix our Bill class:
import java.awt.*;

public class Bill extends Payment implements DeliverableItem {
    public Bill () {
        //a constructor that doesn't do anything
    }

    public void printItem(Graphics g) {
        g.setColor(Color.white);
        g.fillRect(100, 100, 50, 50);
        g.setColor(Color.black);
        g.drawLine(110, 110, 140, 110);
        g.drawLine(110, 120, 140, 120);
        g.drawLine(110, 130, 140, 130);
    }

    public void deliver(Graphics g) {
        printItem(g);
    }
}

Using Interfaces to Help our Waitress

Let's see how using Interfaces might help our Waitress Applet. Modify the Cake, Pie, and Donut classes as shown, so that they all implement DeliverableItem:

import java.awt.*;

public class Cake extends Food implements DeliverableItem {
    int topLocation;
    int leftLocation;
    Color myColor;

    public Cake(int top, int left, Color thisColor) {
        topLocation = top;
        leftLocation = left;
        myColor = thisColor;
    }

    public void bake(Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
    }

    public void deliver(Graphics g) {
        bake(g);
    }
}

Now modify the Pie class:
import java.awt.*;

public class Pie extends Food implements DeliverableItem {

    int topLocation;
    int leftLocation;
    Color myColor;

    public Pie(int top, int left, Color thisColor) {
        topLocation = top;
        leftLocation = left;
        myColor = thisColor;
    }

    public void bake(Graphics g) {
        g.setColor(myColor);
        g.fillOval(leftLocation, topLocation, 50, 50);
    }

    public void deliver(Graphics g) {
    }
}

And modify the Donut class:

import java.awt.*;

public class Donut extends Food implements DeliverableItem {

    public Donut() {
    }

    public void bake(Graphics g) {
        g.setColor(Color.pink);
        g.fillOval(0, 100, 50, 50);
        g.setColor(Color.gray);
        g.fillOval(15, 115, 20, 20);
    }

    public void deliver(Graphics g) {
    }
}

It goes without saying that you should Save/Compile all of these files.

Okay, now our Cake, Pie, Donut, and Bill classes all implement DeliverableItem. Let's see how our Waitress can make use of this new Object-Oriented Design! Modify the Waitress Applet as shown:
import java.applet.*;
import java.awt.*;

public class Waitress extends Applet {
    
    public DeliverableItem[] myMeal = new DeliverableItem[4];

    public void paint(Graphics g) {
        // instantiate the objects, giving a position and Color,
        // store them in the myMeal array
        myMeal[0] = new Cake(0, 0, Color.yellow);
        myMeal[1] = new Pie(0, 100, Color.blue);
        myMeal[2] = new Donut();
        myMeal[3] = new Bill();
        
        // call the bake() in ALL of the Cake and Pie objects
        for (int x = 0; x < myMeal.length; x++) {
            myMeal[x].deliver(g);
        }
    }
}

When we Save/Compile, we see that our Bill has been delivered along with our Cake, Pie, and Donut Food objects. Because all of the items implement the DeliverableItem interface, we can store them all in a DeliverableItem Array.

Inheritance with Interfaces

In our previous example, we took each of our individual objects and, by implementing DeliverableItem, we enabled them all to be used as DeliverableItem objects. But we did not necessarily guarantee that all Foods were DeliverableItems. Suppose that we decided we wanted all Food objects to be DeliverableItems. Here's how we might do that. Modify the Food Class as shown:

import java.awt.*;
public abstract class Food implements DeliverableItem {
    public Food() { }
    
    public abstract void bake(Graphics g);
    
    public void deliver(Graphics g) {
        bake(g);
    }
}

We changed our Abstract Food class so that it implements DeliverableItem. We also implemented the non-Abstract deliver() method. Because the deliver() method is not Abstract, we know it will be inherited by all sub-classes of Food. Modify your code as shown below to see why this is useful:
public class Cake extends Food {
    int topLocation;
    int leftLocation;
    Color myColor;

    public Cake(int top, int left, Color thisColor) {
        topLocation = top;
        leftLocation = left;
        myColor = thisColor;
    }

    public void bake(Graphics g) {
        g.setColor(myColor);
        g.fillRect(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
    }
}

Modify the Pie Class:

import java.awt.*;

public class Pie extends Food {
    int topLocation;
    int leftLocation;
    Color myColor;

    public Pie(int top, int left, Color thisColor) {
        topLocation = top;
        leftLocation = left;
        myColor = thisColor;
    }

    public void bake(Graphics g) {
        g.setColor(myColor);
        g.fillOval(leftLocation, topLocation, 50, 50);
        g.setColor(Color.black);
    }
}

Modify the Donut class:
You may have noticed that our Cake, Pie, and Donut classes are in the original state they were in before we ever started working with Interfaces. Do you suppose this means that they no longer implement DeliverableItem?

When we think about how inheritance works, we notice that our Cake, Pie, and Donut classes inherit ALL properties of Food. Because we inherited the Food.deliver() method, we don't need to explicitly implement a deliver() method in each of our Cake, Pie, and Donut classes.

There is a potential down-side to using inheritance in this way. Now, no matter what we do, all Food objects are also DeliverableItems. In our example, this is just fine. But it's possible that we might not always want multiple data-typing. Fortunately, we always have the option of implementing in individual classes available, just like we did in the previous example.

So after your final project, that's a wrap for this lesson and the course! In later courses, we're going to do even more with Interfaces. Interfaces and Abstract classes become extremely useful as Java programs become more and more complicated. Good job! It's been great having you here.

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.