

Java Programming 1: Introduction to Java and the Eclipse Development Environment

Lesson 1: **Introduction to Java**

[Windows Settings](#)

[An Applet](#)

[Using Eclipse](#)

[Our Second Applet](#)

[Java is an Object-Oriented Language](#)

[Structured Programming](#)

Lesson 2: **Object-Oriented Programming**

[Introduction](#)

[What is Object-Oriented Programming?](#)

[Demystifying the Program](#)

[Reading Code](#)

[What About the Other One?](#)

[Hierarchy Structure in Eclipse](#)

Lesson 3: **Applets**

[Applets](#)

[What Can Applets Do?](#)

[Getting Images](#)

[Applet Uses Other Classes](#)

Lesson 4: **An Applet's Life Cycle**

[Applets Continued](#)

[Applet Life Cycle](#)

[Adding Methods](#)

[Control](#)

[Watching a Life](#)

Lesson 5: **Decisions, Decisions, Decisions**

[Program Control Using If Statements](#)

[If Statements](#)

[Placement of Block Braces](#)

[Comparison Operators and Logic](#)

[Comparison Operators](#)

Lesson 6: **Objects and Classes**

[Objects](#)

[What is an Object?](#)

[Classes](#)

[Java Data Types](#)

Lesson 7: **Classes and Instances**

[Object Design](#)

[Who gets what?](#)

[Initialization and Constructors](#)

[Making an Applet for Dukes](#)

[Another Applet for Dukes](#)

Lesson 8: **Using the API: Introductory Graphics**

[Using Java Provided Classes](#)

[java.awt.Graphics Class](#)

[Using the API](#)

[Methods, Parameters \(or Arguments\), and the Dot Operator](#)

[Sequencing](#)

[The java.awt.Color Class](#)

Lesson 9: **Drawing with Graphics**

[Making Pictures](#)

[Back to Graphics](#)

Lesson 10: **Methods and Method Invocation**

[Methods](#)

[Creating and Using Methods](#)

[Tracing method calls](#)

Lesson 11: **Writing Classes - Building With Methods**

[More on Methods](#)

[Local Variables](#)

[Results and Return](#)

[Building on methods](#)

[Overloading](#)

[How Does Java Find the Right Method?](#)

[Summary](#)

[Method Declarations](#)

[main: an important method](#)

Lesson 12: **Adding Interaction using Components and Listeners**

[Revisiting the Dukes Class and Applet](#)

[A User Modification Example](#)

[Introduction to Interfaces](#)

[An Analogy: Antenna as an Interface](#)

[The Listener Interfaces](#)

Lesson 13: **Modularity: Modifiers, Permissions, and Scope**

[Class Specifications](#)

[Modularity](#)

[Modifiers](#)

[Access Modifiers--Permissions](#)

[What Permissions Allow](#)

Lesson 14: **Class Members, Constants and main**

[Static Members](#)

[Static: Making Your Own](#)

[static and main](#)

[Constructors](#)

[Instantiation](#)

[Constants use the final modifier](#)

[Template and Summary](#)

Lesson 15: **[All Together Now](#)**

[Interaction and Playing Around](#)

[Putting It All Together](#)

[Using Inheritance on our Own](#)

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.*

Introduction to Java

Welcome to the O'Reilly School of Technology's Java Programming 1 course, Introduction to Java and the Eclipse Integrated Development Environment (IDE).

Course Objectives

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

- build Java applications and applets in the Eclipse IDE.
- create control structures, classes, objects, and methods.
- add interaction to programs using components and listeners.
- apply the Java API to draw graphics.
- demonstrate understanding of modularity, modifiers, permissions, scope, and inheritance.

In this course, you'll learn the fundamental concepts and syntax of the Java programming language. Throughout this course, you will build examples using the Eclipse Java IDE, which is supplied as a Learning Sandbox. Completion of this course will give you a basic understanding of Object-Oriented techniques in Java, as well as using the Eclipse IDE.

From beginning to end, you'll learn by doing your own Java projects, within the Eclipse Learning Sandbox we affectionately call "Ellipse." These projects will add to your portfolio and provide needed experience. All you need is a browser and internet connection we provide all the software you need online.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take the *useractive* 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.

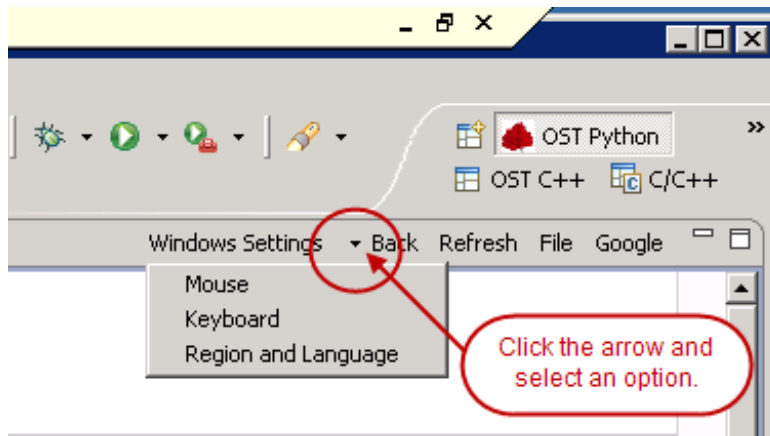
Here are some tips for using O'Reilly School of Technology courses effectively:

- **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

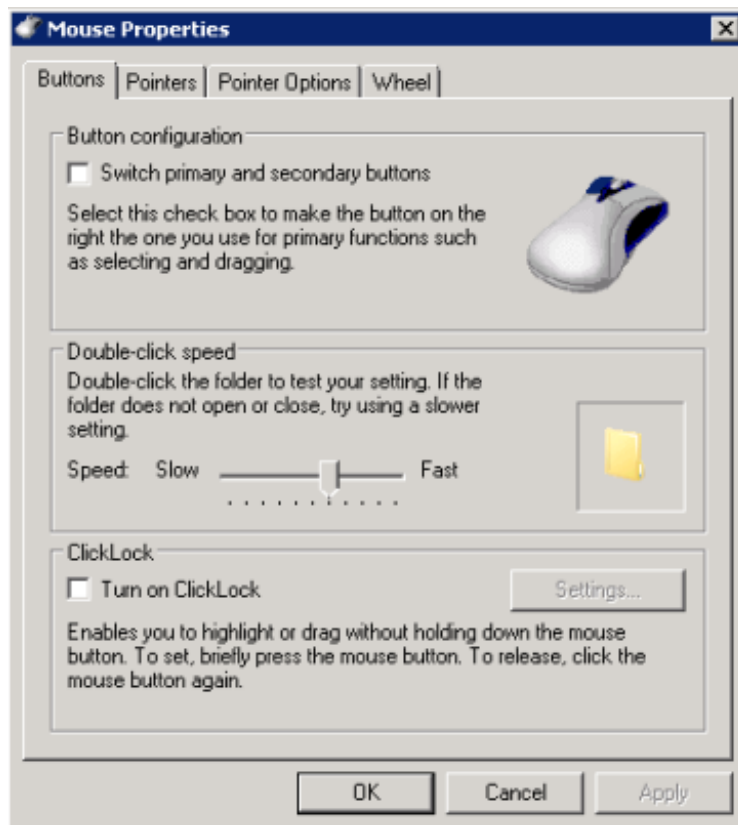
some projects to show off when you're done.

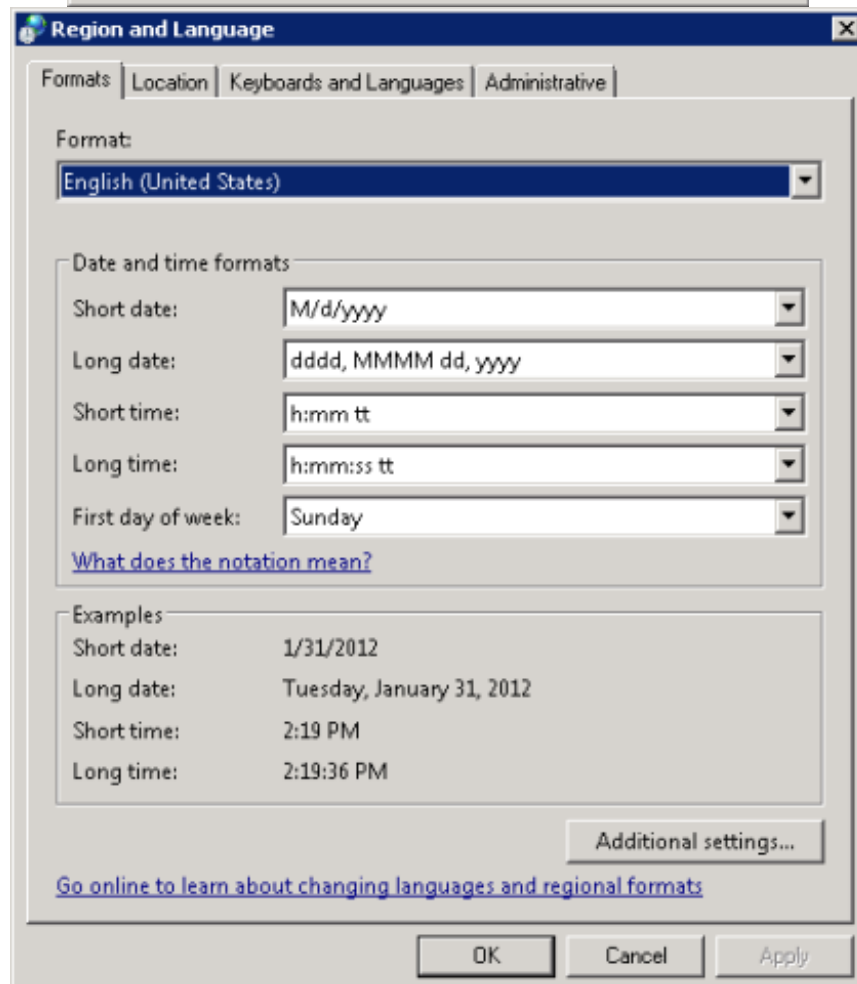
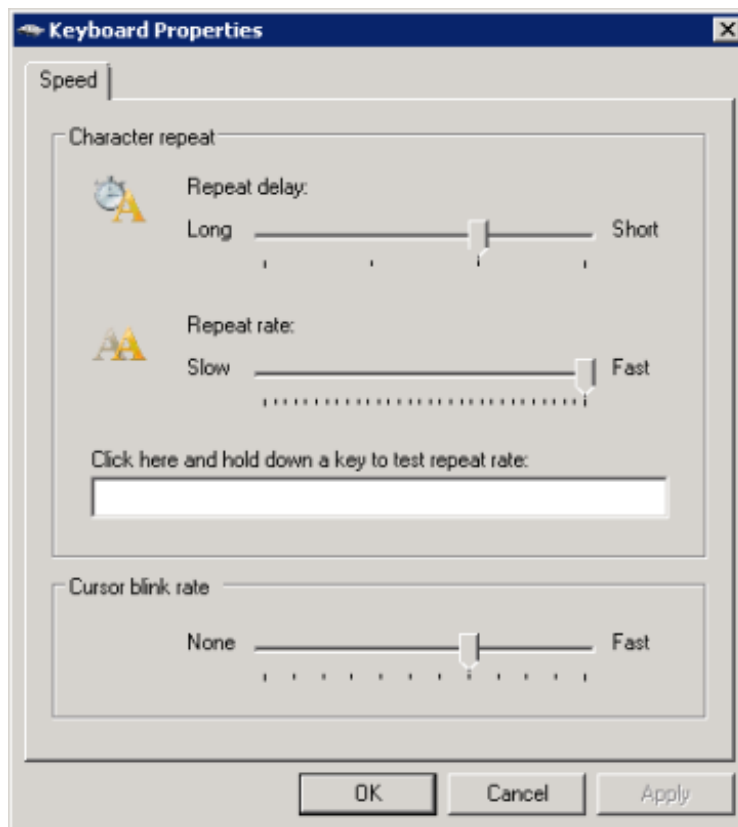
Windows Settings

If you like, you can set your own Windows mouse, keyboard, and region; for example, if you are left-handed, you can switch the left and right button functionality on the mouse, or you can change date fields to use date formats for your local region. Click the down arrow on the Windows Settings button at the top right of the screen:



We won't discuss the details of these dialog boxes, but feel free to ask your instructor if you have questions.





Now, before we get started programming in Java, let me show you how the material will be presented.

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:

CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).
If you have already written some of the code, new code for you to add looks like this.
If we want you to remove existing code, the code to remove ~~will look like this~~.

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:

INTERACTIVE SESSION:

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.

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:

OBSERVE:

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to observe.

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:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

An Applet

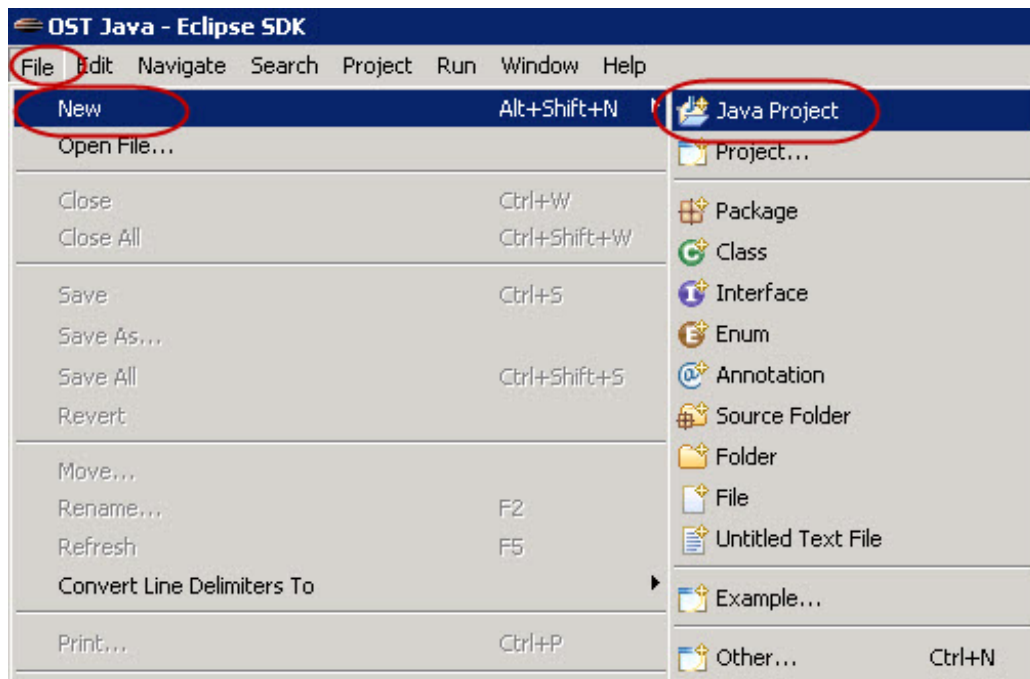
If you're new to programming, allow me to introduce you to the "Hello World" example. Traditionally it's the first program you write in any language. I would bet that a "Hello World" program has been written in every computer language.

Using Eclipse

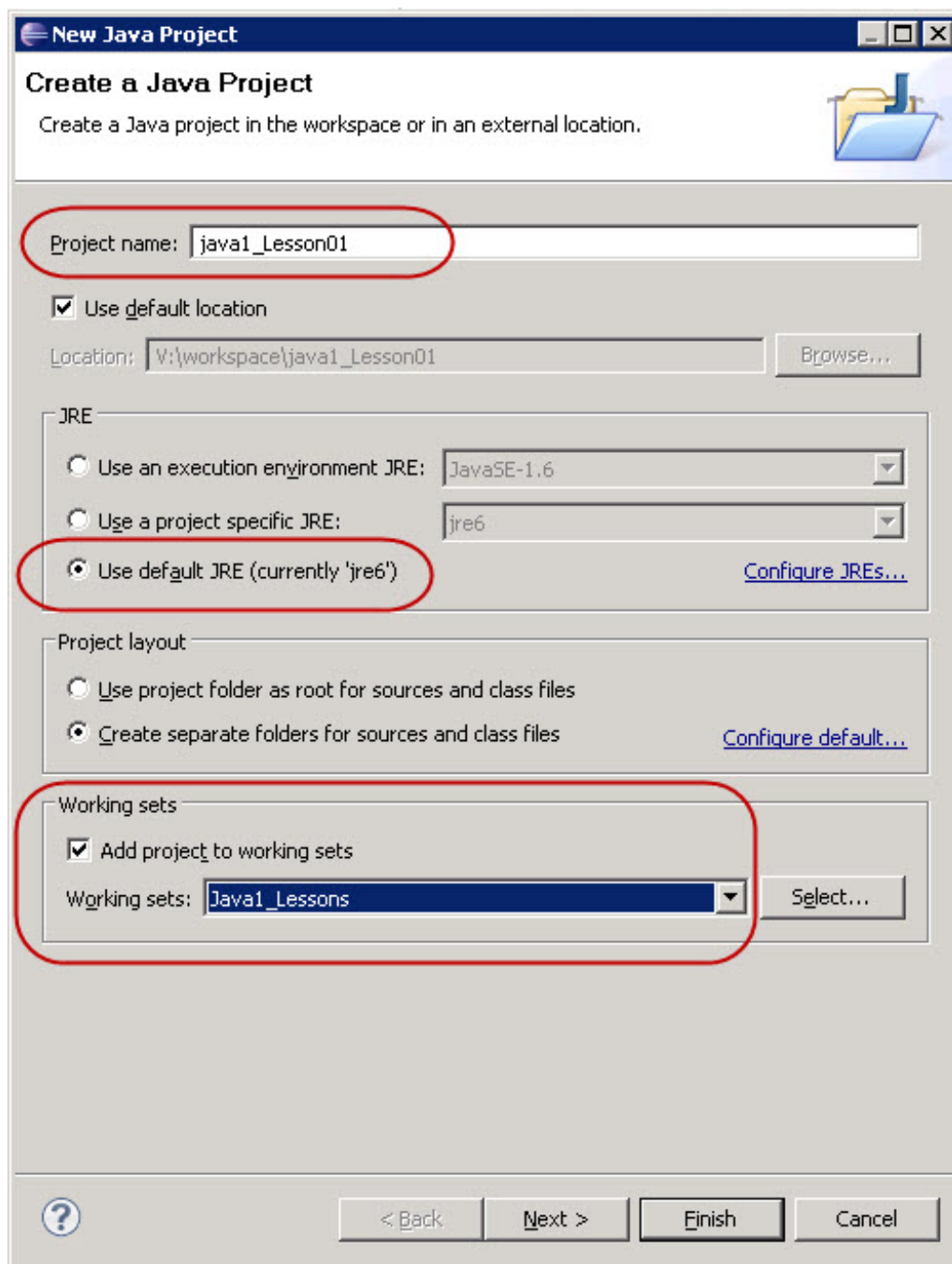
The program filling up your screen right now is an Integrated Development Environment (IDE) called Eclipse. An IDE is essentially an editor that is customized to help you to program. Our IDE, Eclipse, is customized to help you with Java.

We're inching ever closer to making something, but first we need to set up the environment for our first file. In Eclipse, all files must be within *projects*. In this course, a *project* is the same thing as a *folder*.

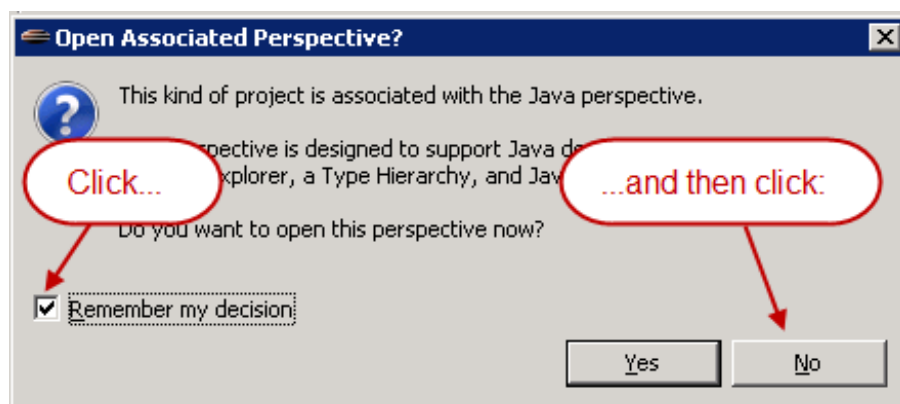
To start our first project, select **File | New | Java Project**:



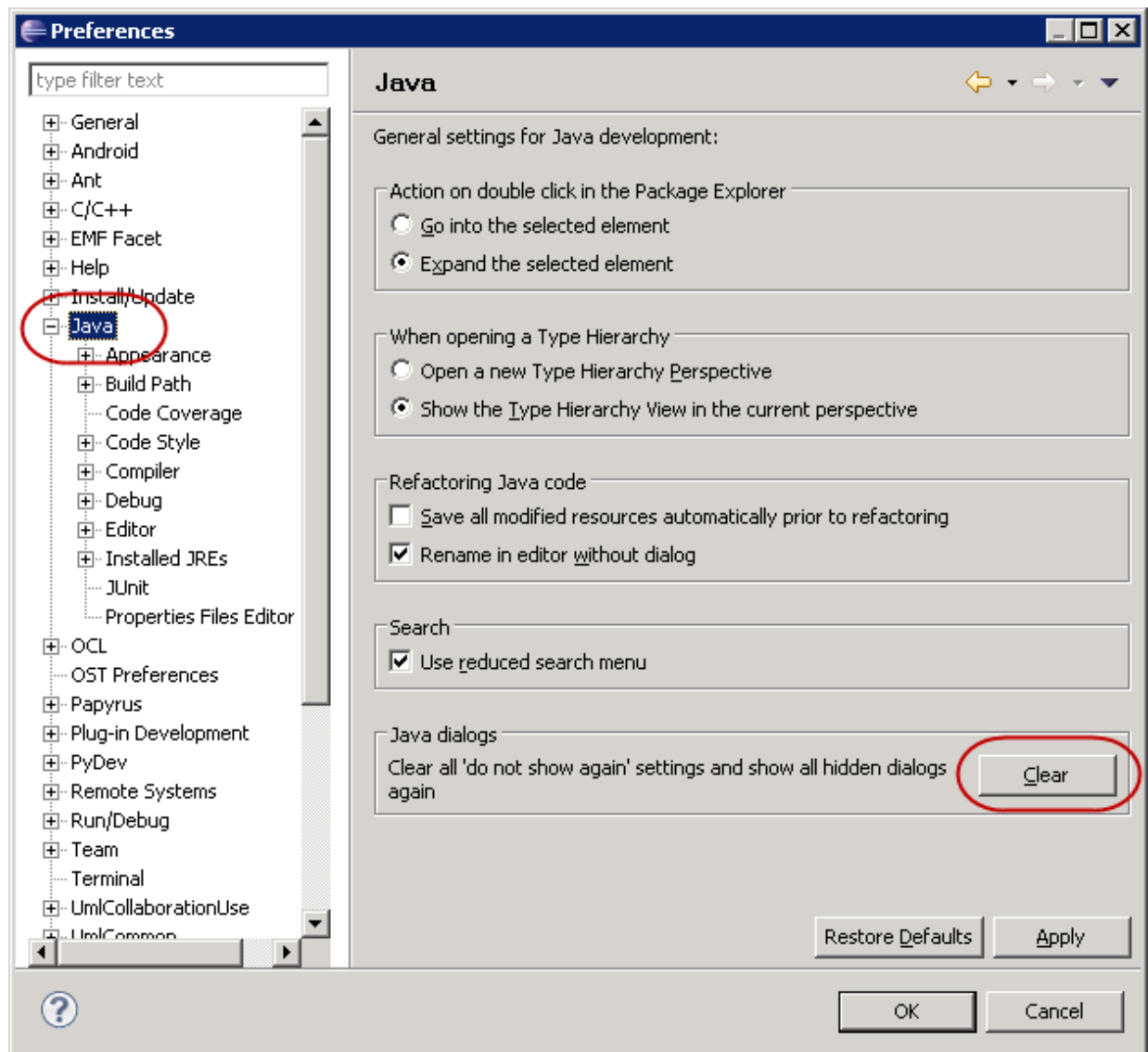
Name your project. (We'll be making a whole lot of projects, so we'll choose names that will help us keep them organized.) Call this one **java1_Lesson01**, choose to **Use default jre** (Java Runtime Environment), and be sure and put it in the **Java1_Lessons Working Set**.



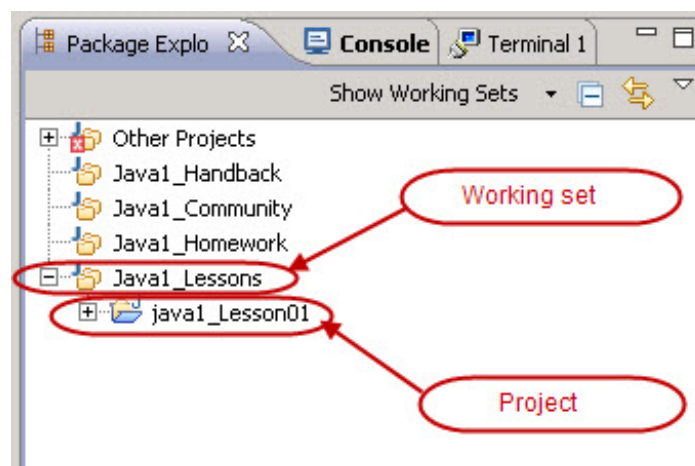
If you see the dialog below, go ahead and check the **Remember my decision** box and then click **No**.



If you clicked **Yes** on the above dialog by mistake, select the **Windows** menu and click **Preferences**. When the dialog appears, click **Java** on the left, and then click **Clear** as shown:



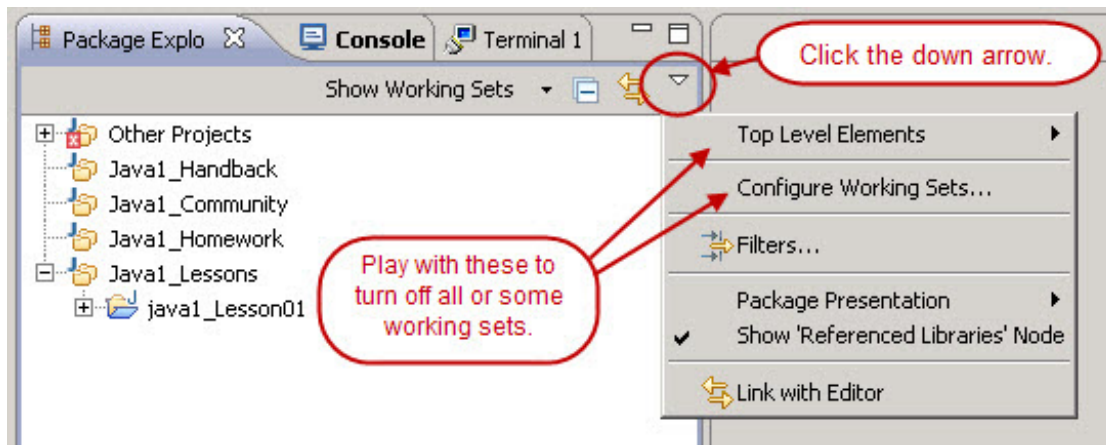
Now you can see the **java1_Lesson01** project listed in the **Package Explorer** panel on the lower-left corner of your screen:



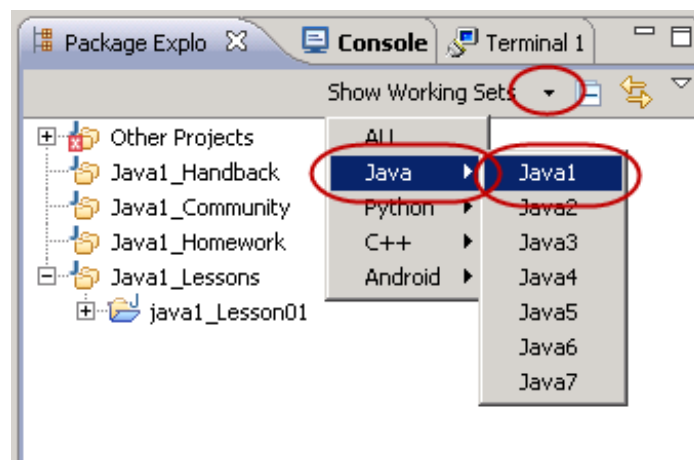
This hierarchical view of the resources (directories and files) in Eclipse is commonly called the *workspace*. You now have a project named **java1_Lesson01** in your workspace.

You probably noticed items in the workspace like **Java1_Lessons**, **Java1_Homework**, and so on. We put them there to help you stay organized. These are called *working sets*. A working set is like a folder, but is actually just an association of files. The difference between a working set and a folder is that a working set doesn't have any depth in the file system, so file and folder references don't even see them. You can turn working sets on and off in Eclipse. You can either turn all working sets off or turn only some of them off if

things get too cluttered. Try it! Click the white down arrow on the Package Explorer tab:

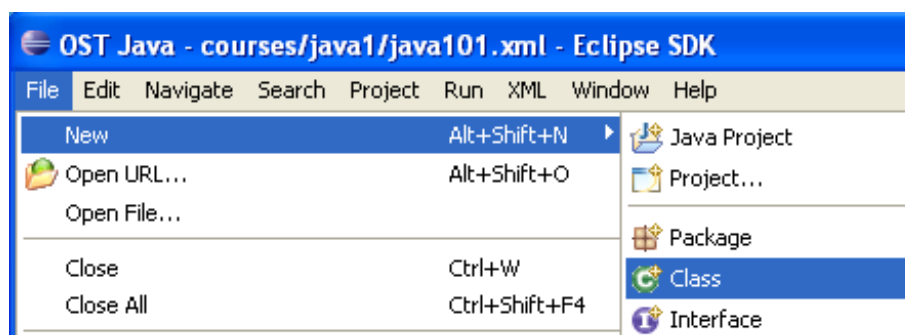


Now that you've played around with working sets, make sure they're all back in place. To easily show just the working sets for this course, click the small black down arrow on the **Show Working Sets** button in the Package Explorer, then select **Java**, and then **Java1**:

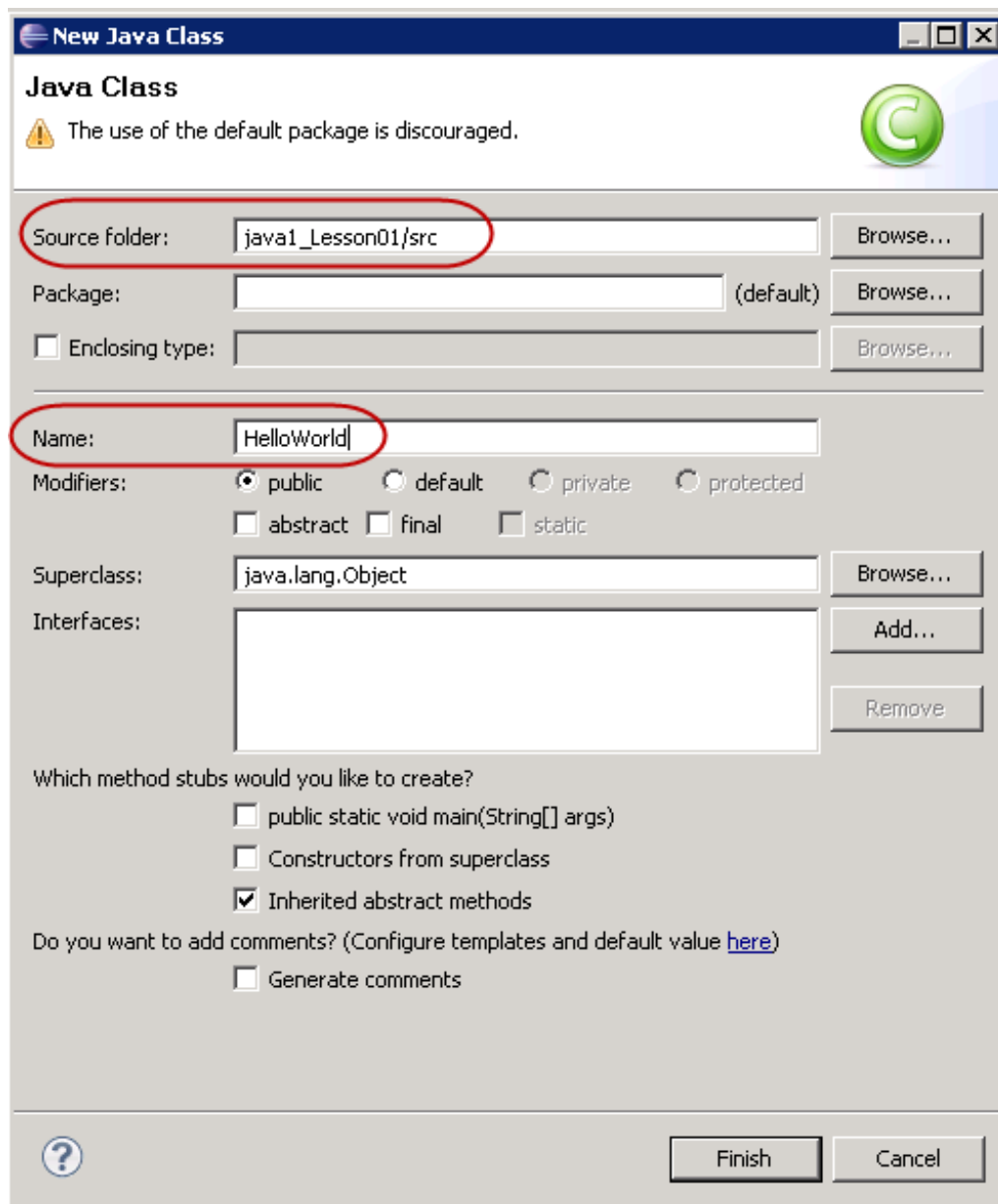


We're about to create a new *class* file to put into our project. We'll talk about classes in detail later, but right now, let's get going and make one!

Select **java1_Lesson01** in the Package Explorer so it's highlighted. In the top **OST Java - Eclipse SDK** menu bar, choose **File | New | Class** as shown (if the **New** submenu doesn't include **Class**, choose **Other...**, then double-click on **Class**):



In your New Java Class window, if Source folder: does not already contain **java1_Lesson01/src**, click **Browse...** to look for and select the **java1_Lesson01/src** folder (or type **java1_Lesson01/src**). Give your class the name **HelloWorld** as shown:



The image shows the 'New Java Class' dialog box in an IDE. It has a title bar with a blue icon and the text 'New Java Class'. Below the title bar is a section titled 'Java Class' with a warning icon and the text 'The use of the default package is discouraged.' and a green 'C' icon. The dialog contains several input fields and buttons. The 'Source folder:' field is set to 'java1_Lesson01/src' and is circled in red. The 'Package:' field is empty and set to '(default)'. The 'Enclosing type:' checkbox is unchecked. The 'Name:' field is set to 'HelloWorld' and is circled in red. The 'Modifiers:' section has radio buttons for 'public' (selected), 'default', 'private', and 'protected', and checkboxes for 'abstract', 'final', and 'static'. The 'Superclass:' field is set to 'java.lang.Object'. The 'Interfaces:' field is empty. There are buttons for 'Browse...', 'Add...', and 'Remove'. Below these fields is a section titled 'Which method stubs would you like to create?' with checkboxes for 'public static void main(String[] args)', 'Constructors from superclass', and 'Inherited abstract methods' (checked). Below this is a section titled 'Do you want to add comments? (Configure templates and default value [here](#))' with a checkbox for 'Generate comments'. At the bottom are buttons for '?', 'Finish', and 'Cancel'.

New Java Class

Java Class

The use of the default package is discouraged.

Source folder: Browse...

Package: Browse...

☐ Enclosing type: Browse...

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

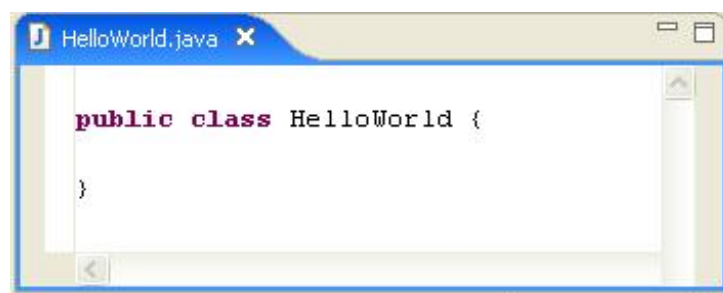
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

? Finish Cancel

Click **Finish**. You see this content in the lower-right *Editor Window*:



The image shows a code editor window titled 'HelloWorld.java'. The code inside is:

```
public class HelloWorld {  
  
}
```


This Editor Window allows you to create, write, and of course, edit code.

Notice that the name of the *file* for the code is `HelloWorld.java`. Java *source code* always has the same name as the *class*, followed by the **.java** extension.

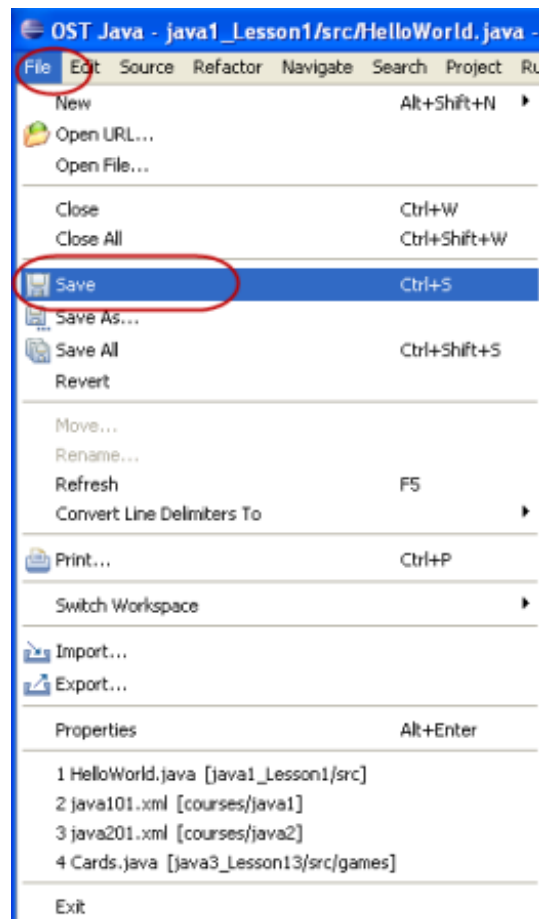
Okay, now let's add the code for our first Applet! Type the code in the editor below as shown:

CODE TO TYPE:


```
import java.applet.Applet;  
import java.awt.Graphics;  
  
public class HelloWorld extends Applet {  
    public void paint(Graphics g) {  
        g.drawRect(0, 0, 100, 100);  
        g.drawString("Hello World!", 5, 15);  
    }  
}
```

If you see one of these  beside the **public class HelloWorld extends Applet {** line, just ignore it. This symbol is a *warning* that can be ignored for now; it won't affect the running of your code.

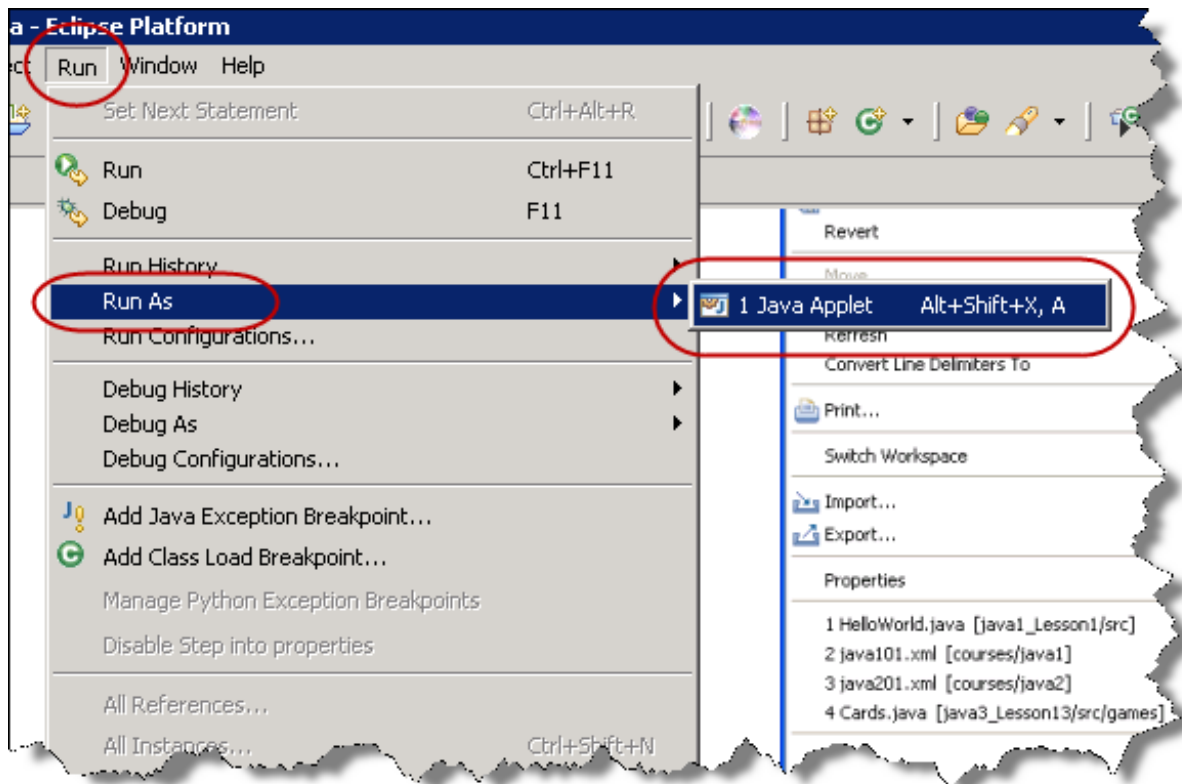
Eclipse has a special way of running Applets, but they usually need to be run on a web page. Now that we've got that code typed, let's save and run it! Click in the Editor Window where your code is written (the HelloWorld.java file). In the very top Eclipse menu bar (not the O'Reilly tab bar), select **File | Save**:



Note

There are a few different ways to save a file. You can select **File | Save** as above, or click the  in the Eclipse toolbar, or you can press **Ctrl+S**. In the future, when we want you to save a file, we'll show the Save icon.

Now, from the top menu bar, select **Run | Run As | Java Applet**. (If **Java Applet** isn't there, go back to the Editor Window and click in the HelloWorld.java editor and try again.)



Tip

As with saving, there's more than one way to run your Java code; just use the one you like best. When we want you to run a program, we'll show the **Run** icon (🟢).

Now you have a small browser running in the upper-left corner of your computer screen. The browser is labeled Appletviewer and displays the output of your Applet. It prints out **HelloWorld!** inside a box:

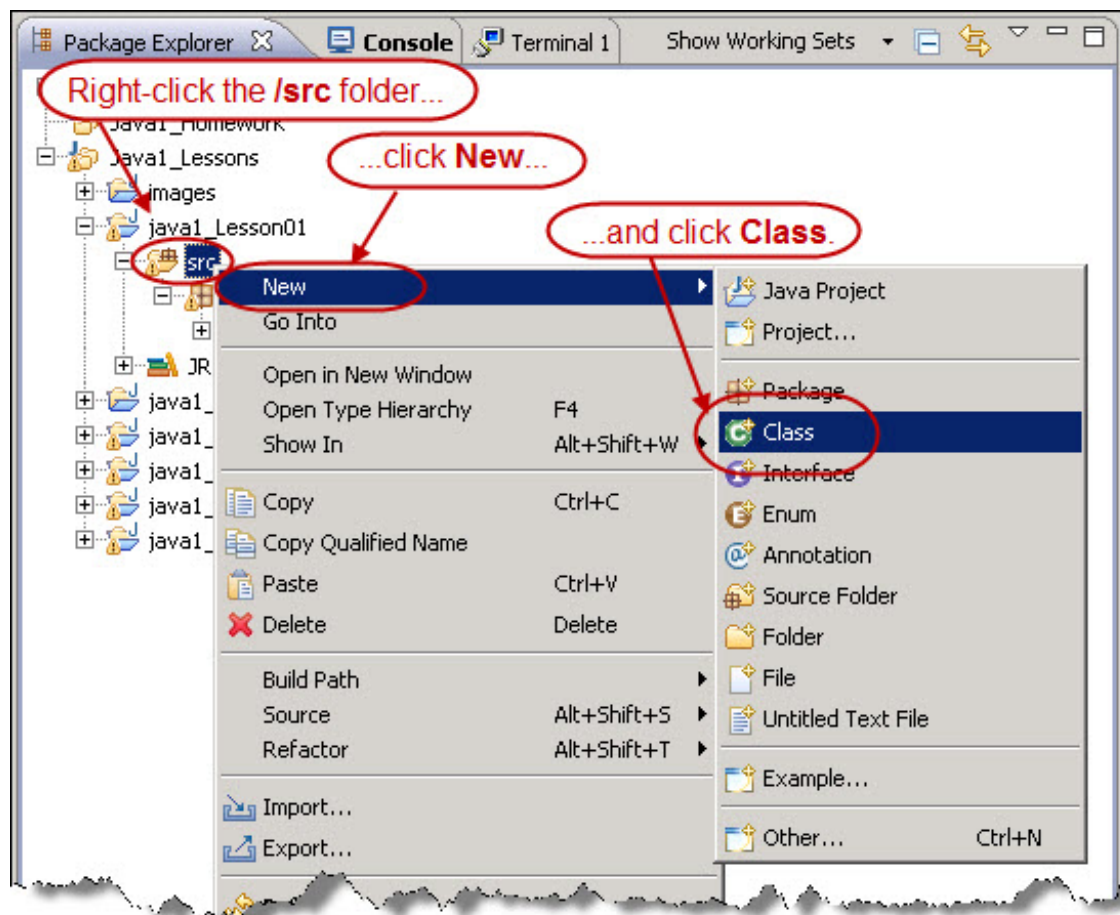


Sweet! You've officially created and run your first Java Applet! To close the Appletviewer, click the **x** in the upper-right corner.

Our Second Applet

The next Applet we'll create is similar to the first one, but it uses different code. To begin, we'll create a new Class file in `java1_Lesson01`, and name it **HelloWorld2**.

Here's another way to create a class. Right-click the `java1_Lesson01/src` folder in the Package Explorer, and select **New | Class**:



Name it **HelloWorld2** in the dialog box:

New Java Class

Java Class

⚠ The use of the default package is discouraged.

Source folder: Browse...

Package: Browse...

☐ Enclosing type: Browse...

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

Finish Cancel


Now let's give **HelloWorld2** some code.

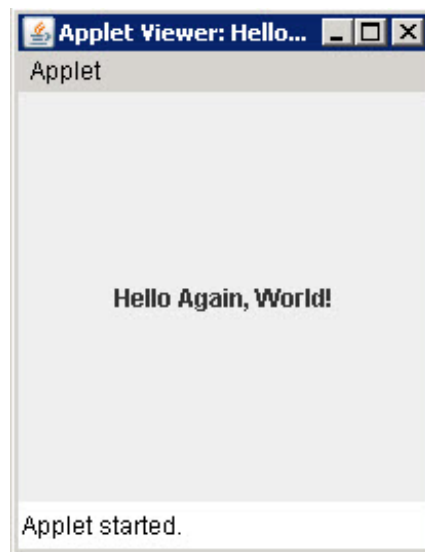
Type the code below (seriously, type the code, don't just cut and paste!) into the **HelloWorld2** file as shown:

CODE TO TYPE:

```
import java.awt.*;
import javax.swing.*;

public class HelloWorld2 extends JApplet {
    public void init() {
        Container contentPane = getContentPane();
        JLabel label = new JLabel("Hello Again, World!", SwingConstants.CENTER);
        contentPane.add(label);
    }
}
```

 Now save and run it, using the Eclipse **File** and **Run** menus like we did earlier (or you can use the  shortcut).



Yes, you've done it again! This Applet looks a little different from your first one, but it does roughly the same thing. Good job! Again, click the upper-right **x** to close the Appletviewer.

Java is an Object-Oriented Language

Structured Programming

In order to appreciate object-oriented programming, let's take a look at its predecessors: *procedural languages*, such as C and Fortran. They consist of *procedures* or *routines* which simply contain a series of computational steps to be followed.

The very first *construct* of computing is **sequencing**, which means the code follows the lines in sequence, one after the other. If a programmer makes mistakes and a language isn't meticulously written, the steps could be hard to figure out. Here's an example of such a program in a procedural language:

OBSERVE:

```
10 i = 0
20 i = i + 1
30 if i <= 10 then goto 80
40 if i > 10 then goto 60
50 goto 20
60 print "Program Completed."
70 end
80 print i; " squared = "; i * i
90 goto 20
```

Let's *trace* (follow the execution of) this program. We'll go through each of the steps and see what they do:

OBSERVE:

```
10: Set i to 0
20: Set i to i(0) + 1, or 1
30: Since i is now 1 and 1 is less than < 10, go to 80
80: Print the value of i (which here is 1), and the text " squared = ", and then
    the product of i * i (here, it is 1)
90: Go to line 20
20: Set i to i(now 1) + 1, or 2
30: Since i is now 2 and 2 is still less than < 10, go to 80 again
80: Print the value of i (which here is 2), and the text " squared = ", and then
    the product of i * i (now, it is 4)
90: Go to line 20
20: Set i to i(now 2) + 1, or 3
... It continues looping until i is 10 or more at line 30, at which point it pas
ses to...
40: Go to line 60
60: Print "Program Completed."
70: End the program
```

The **same** program in a **structured programming** language would look something like this:

OBSERVE:

```
for i = 1 to 10
    print i; " squared = "; i * i
next i
print "Program Completed."
```

Notice that you don't need to include the line numbers in your code. The code is *structured* in such a way that it doesn't send you all over the place! Structured code is preferred because:

- It's easier to follow.
- It's easier to prove that it is correct.
- It's easier to *debug*.
- It's more concise.

Object-oriented programming was developed to avoid the common pitfalls of *procedural programming*.

Let's create and run that program *in* Java to see how well Duke handles it!

Create a new class in the **Java1_Lesson01** project named **StructuredDemo**.

Go ahead, we'll wait.

Now, type the code into **StructuredDemo** as shown:

CODE TO TYPE:

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

public class StructuredDemo extends Applet {

    public void paint(Graphics g){
        for (int i=1; i <10; i++)
            g.drawString(i + " squared = " + i*i, 10, 15*i);
        g.drawString("Program Completed", 10, 180);
    }
}
```



Run it. It works great! When you finish, close the Appletviewer as before.

Nice! We have written not *one*, not *two*, but *three* Java Applets, and we've run them in the Eclipse IDE!



If you haven't closed them already, you can still see a tab for each program at the bottom of your window. Click on the **x** in each one's tab to close them, or Eclipse will close them for you when you Exit.

Good job so far, but we're just getting started!

Now, go back to the syllabus page by clicking the **Back** button and complete any Quizzes and Projects for this lesson.

See you in the next lesson!

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



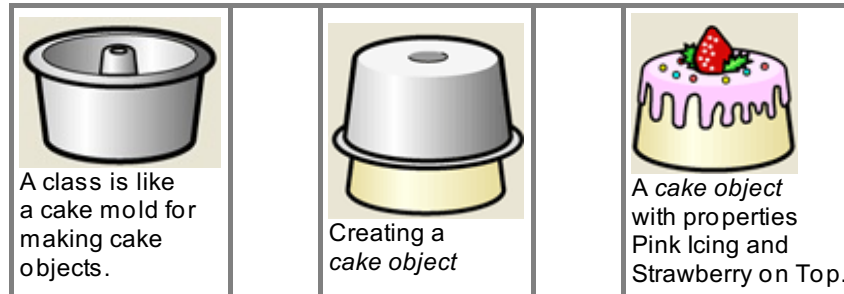
*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Object-Oriented Programming

Introduction

What is Object-Oriented Programming?

Every piece of code that makes up an object-oriented program is known as an *object*. The code representation of an object is a *class*, and classes produce *objects*. Think of an object like a cake that you bake. A cake can have all kinds of properties: shape, flavor, icing, and so on. To continue with this analogy, a class is like a particular cake mold. Each *instance* of cake made using that mold may be slightly different because it has different properties. Other people can use your mold to make cakes that accomplish their own tasks. Keep this cake metaphor in mind as we continue to discuss classes, objects, instances, and properties.



In Java, every object has *properties* and *methods*. The properties determine an object's current *state*, while methods are actions that can alter that state. Classes are nouns, and methods are verbs! For instance, we might EAT a cake. So EAT could be a cake's method in Java.

To write an object-oriented program:

1. Determine which objects we already have at our disposal.
2. Use these objects to make more objects to perform additional tasks.

When we program in Java, we're telling objects what to do to each other. After a Java program is written, other programmers can create new classes around the objects in the program, and then we can use those new objects too. Objects in Java allow programmers to cooperate and share.

Along with Java, there is a *library* of objects for everyone to use as a base. Over time, the library has grown. Each new version of Java not only fixes problems (bugs) found in previous versions, but provides us with even more objects (classes) to use. In order to do object-oriented programming, you'll want to become familiar with the library of available classes. The library is provided online by Oracle, and is called the Application Programming Interface (API).

Whenever we're looking at a class in the API, we'll give you a heads-up with one of these API icons:



To see the API page, click the link. Closing the window or clicking on the **Lesson View** tab will bring you back here.

The API is also called the *Class Library*. Notice that the Oracle page has API documents for numerous versions of Java. Usually you'll want to run the newest version of Java because the newer versions have fixed more bugs and have made more classes available.

Look on your menu for the **API** button in the row of icons under the Eclipse menu bar. Click this to see the API for the most current Java version. The more you program with Java, the more you'll rely on the API.

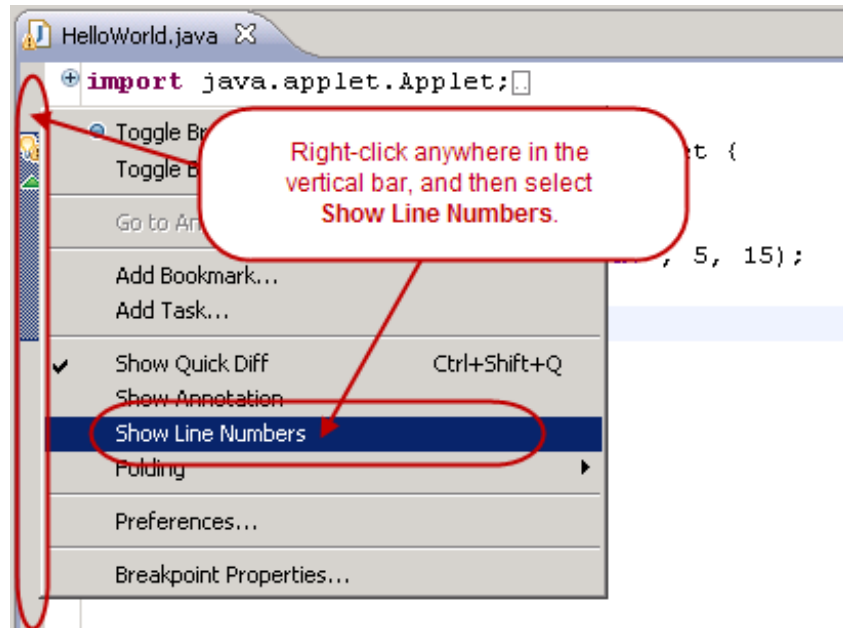
Now that we know some OOP terminology, let's look at the objects in your first program.

Demystifying the Program

Reading Code

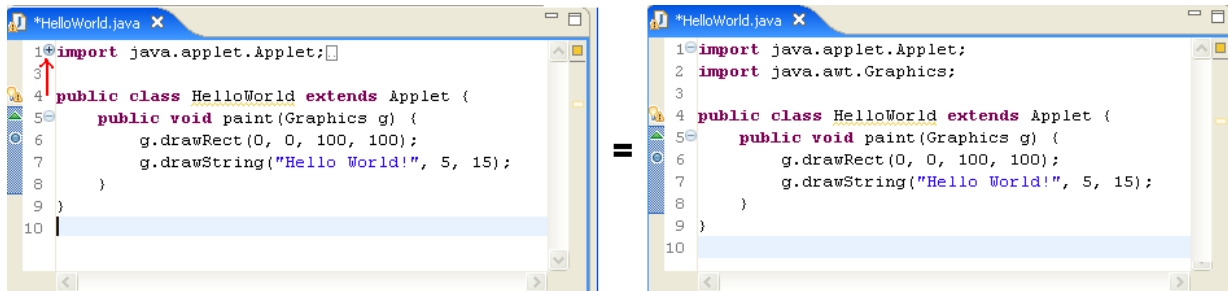
Once again, go to the Package Explorer and open your **java1_Lesson01** project. In the **src** folder, find **HelloWorld.java** and open it in the editor.

For now, so we can work through the code, let's use line numbers next to the code. To make line numbers visible, right-click the vertical frame bar at the left of the editor window, and select **Show Line Numbers**:



It's possible that you have "collapsed" blocks of code in your program. If you see a **+** sign next to the number 1 line of your code, click on it. If your **+** sign is on line 2, delete the empty line 1 by going to the beginning of line 2 in front of **import** and pressing the **Backspace** key. Now your line numbers should match up.

Eclipse was simply "collapsing" a block of code lines to focus on the other code. The lines are all still there, though. The plus sign (**+**) means "expand" (indicating collapsed code) and minus (**-**) means "collapse" (indicating there is a block of code you *can* collapse).



Okay, now let's break the code down one line at a time:

OBSERVE:

```
1. import java.applet.Applet;
```

This line tells Java that something in your code is going to use one of those classes from the Java API; Java will *import* all the stuff for you so that you can use it. Here the object type (class) we want to use is called **Applet**. Java can find this class in the *package* named **java.applet**.

Packages are directories that hold a collection of related objects (classes). The **java.applet** package contains the **Applet** class (a really useful class).

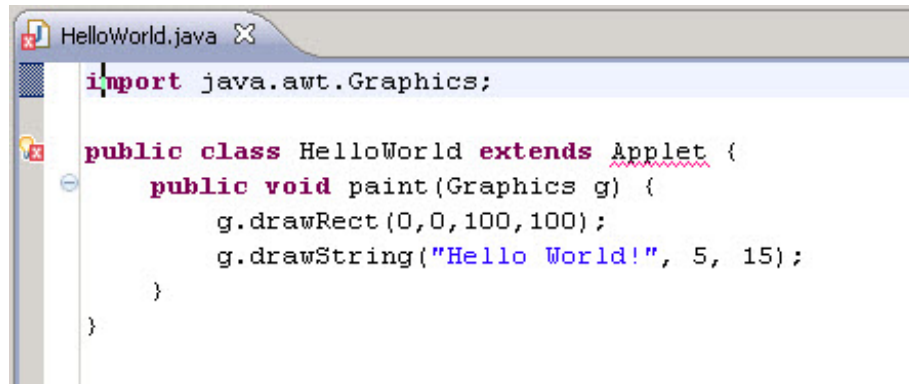
API Go to the API page for the [java.applet package](#). Click on the link. Scroll down to the **Class Summary** and click on the **Applet** class (we'll discuss the information that the API gives us later).

Let's see what would happen if we didn't import the Applet package. Try removing the import statement as shown:

CODE TO TYPE:

```
1 import java.applet.Applet;
2 import java.awt.Graphics;
3
4 public class HelloWorld extends Applet {
5     public void paint(Graphics g) {
6         g.drawRect(0, 0, 100, 100);
7         g.drawString("Hello World!", 5, 15);
8     }
9 }
```

Now you'll see this:




See the light bulb and X in the red box on the left panel? They're telling you that something's missing.

This means there's an error in our program. Eclipse will not run with errors, but fortunately, it suggests a remedy. See how the word *Applet* is underlined with a red zig-zag? Try moving your mouse over it. It says **Applet cannot be resolved to a type.**


That means that Eclipse can't identify **Applet** because the package that contains the class Applet is not known ("cannot be resolved"), so the class information is not available.

If you click on the light bulb, it will suggest ways to fix the problem. When you click on one of the suggested remedies, Eclipse will implement it. For instance, click on the light bulb and then double-click on the choice **import 'Applet' (java.applet)** to put it back in the code.

If the import statement for Applet is on line 2, delete the empty line 1, so the numbers line up again.

The  icon signifies errors. You must fix these in order to run your code.

Note

The  icon is only a warning—Java is uncomfortable with some part of your code, but will still run.

Now, back to our HelloWorld code:

OBSERVE:

```
2 import java.awt.Graphics;
```

In line 2, we're importing another class. Can you tell from the syntax of the line which part is the package and which part identifies the class? By convention, package names begin with lower-case characters and classes begin with upper-case letters. Our line of code ends with a ; (semicolon). The semicolon in Java syntax tells the compiler that it has arrived at the end of a line of code.

So in this case, **java.awt** is the package and **Graphics** is the class.

You can learn more about the **Graphics** class at:

1. [API](#)
2. [java.awt](#)
3. [Graphics](#)

We'll take a closer look at the **Graphics** class in future lessons as well.

OBSERVE:

```
3
```

Line 3 is a blank line that helps make the code easier to read. Here, it separates the block of import statements from the class definition.

OBSERVE:

```
4 public class HelloWorld extends Applet {
```

Line 4 signifies the beginning of the definition of our first **class**.

The first word, **public**, is a *modifier*. The modifier tells Java something about *access*—who can use the class or method or variable. In this case, it's **public**. For this particular course, all of our classes will be public, but many of our methods and variables will be **private** or **protected**. We'll explore access more later, but if you're curious and want to know more right now, go ahead and Google "Java modifiers."

The next keyword in our code is **class**, which tells Java we're creating a class.

This particular class is called **HelloWorld**.

The keyword **extends** tells Java that we want HelloWorld to inherit all the **Applet** code—we are defining HelloWorld to be an Applet. Another option is **implements**, which we'll get into later too.

At the beginning of this lesson, you ran an Applet. It opened a window on your computer with a nice frame around it and typed in words for you. You didn't have to type them in yourself because Java already had a class (**Applet**) written for you to implement. So, when you *inherit* a class that Java has already written (and so is already present in the API), you get access to all of its (the *parent* class's) properties.

The curly bracket, or *brace* **{** at the end of line 4 tells Java that instructions for the HelloWorld applet's task begin there. We'll tell Java when our **HelloWorld** class definition is done by including a closing brace **}** on line 9.

You can see which pairs of braces match up by clicking the mouse directly after an opening or closing brace—the matching brace will be highlighted on the screen. In our example, if you try clicking right after the closing curly bracket on line 9, the editor makes a little box around the **{** on line 4, so we know that those two brackets match. This can be really handy in large programs!

Okay, so what's in the class? Take a look:

OBSERVE:

```
5 public void paint(Graphics g) {
6     g.drawRect(0, 0, 100, 100);
7     g.drawString("Hello World!", 5, 15);
8 }
```

In lines 5-8, we define a component of the **HelloWorld** class. We already know that classes may have properties and actions. This particular class doesn't have properties yet, but it does have an action available (starting at line 5) called **paint**. In OOP, the actions that a class can take are called **methods**. In this code you've defined the method **paint()**. Take another look at that block of code:

OBSERVE:

```
public void paint(Graphics g) {
    g.drawRect(0, 0, 100, 100);
    g.drawString("Hello World!", 5, 15);
}
```

Similar to class definitions, all method definitions begin with a *modifier*. Here again we use the modifier **public**. The next keyword in the method definition (**void**) tells us the type of object the method will **return**. This particular object doesn't return anything, so we use the keyword **void**. We use this keyword on all methods that don't return objects. Don't worry, we'll see some methods that do return objects in our upcoming lessons.

This method is **passed** another object called a **Graphics** area. The code states that the **Graphics** area is named **g**. We use methods from the Graphics object (conveniently, someone at Oracle has already created that object). When we write **someNoun.someVerb()**, we are telling Java to access the method called **someVerb()** from the object **someNoun**. In this case, we wrote **g.drawRect(0, 0, 100, 100);** and **g.drawString("Hello World!", 5, 15);** because we knew the Graphics object has **drawRect()** and **drawString()** methods we can use. How did we know all of that? We looked it up in the [Java API!](#)

This definition of the **paint()** method creates a Graphics area for the Applet so you can "paint" or "draw" on it. In this code, you are drawing two things:

g.drawRect(0, 0, 100, 100) tells the Applet to draw a rectangle on the **Graphics** area **g**. The numbers in our code indicate the four corners of our rectangle in pixels. I'll leave it to you to experiment to figure out which number corresponds to which corner! You can do that by changing the numbers and rerunning the Applet.

g.drawString("Hello World!", 5, 15) is telling the Applet to draw the words "Hello World" on the **Graphics** area **g**. The numbers indicate where you want to place the words (string) on the Graphics area. Again, I'll leave it to you to experiment and play with the numbers.

Braces **{}** in Java are always matched with their nearest partner. The **{** at the end of line 5 is closed with the **}** at line 8. Use the mouse click trick we learned earlier and see for yourself. The **{** at the end of line 4 is closed with the **}** at line 9.

Now let's change the size of the rectangle and write something different on the Applet.

If you don't already have **HelloWorld.java** in the Editor window, look for a **HelloWorld.java** tab in the Editor window and click it. If there is no **HelloWorld.java** tab, open the **java1_Lesson01** folder and its default package in the Package Explorer area, then double-click the **HelloWorld.java** file. Edit the code as shown below:

CODE TO TYPE:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
        g.drawString("On to new things!", 5, 15);
    }
}
```



Save and run it. Use the mouse to resize the Appletviewer window and note how the box changes! Are you impressed? (Remember to close the applet after running it.)

Let's change the line again to see what it does:

CODE TO TYPE:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawRect(0, 0, getSize().width - 50, getSize().height - 100);
        g.drawString("I'm not really impressed yet", 5, 15);
    }
}
```



Save and run it again.

Changing the two lines of code altered the size of the surrounding rectangle and wrote different statements.

Close the Appletviewer, and try experimenting more with the rectangle size. Make sure to save and then run it. You can always refer to the original code above, in case you *really* mess up!

Now let's go over class names. The class that we defined above with the name **HelloWorld** must have the file name **HelloWorld.java**.

WARNING

Java is **case-sensitive** and every character matters. The file must have the same name as the class you define, plus the **.java** extension. So, if you change the class name, you must change the file name as well.

In your HelloWorld class, try to change your name of **HelloWorld** to **HelloWorld6**. You can view suggestions by clicking once on the light bulb warning. This trick is really handy for correcting errors.

In our example, you'll be instructed to rename the type (class) HelloWorld, or rename the compilation unit (file) to HelloWorld6.java.

The .java file is called the *source code* for the class. Eclipse calls it the "compilation unit" because Java compiles source files to convert programs into something the machine understands. We'll look into the compilation of Java in greater detail in a future lesson.

For now, double-click on **Rename type to 'HelloWorld'** to fix the name.

What About the Other One?

So what about the other Applet we made? How does it work? I'm happy you asked. We'll go over it now, but don't panic if you don't quite understand the explanation in this section. By the end of this course you will! Open **HelloWorld2** from your **java1_Lesson01/src** folder and turn on line numbers as before:

OBSERVE:

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class HelloWorld2 extends JApplet
5 {
6     public void init()
7     {
8         Container contentPane = getContentPane();
9         JLabel label = new JLabel("Hello Again, World!", SwingConstants.CENTE
10         R);
11         contentPane.add(label);
12     }
13 }
```

When we look at the Applets resulting from these two classes, they look similar, but the code is very different. As programmers, we choose from lots of options that determine how our code will execute tasks. That's why some code might look absolutely beautiful, but other code might be nearly impossible to follow. While your first priority when writing any code is to make sure it works properly, you also want to keep in mind that other programmers will use and edit the code you write. If your original code isn't well-written, that can become a real burden to others in the community. Write good code in the first place, and make sure subsequent programmers don't curse your name!

In this course, we want to instill principles of good software engineering, so that your code will run correctly, look beautiful, and be useful and clean for programmers who will use it in the future.

Now, as it turns out, the two Applets we wrote are both composed of good, solid code. But they're just two of the many ways we could have written them. I'll show you another one. In the **HelloWorld2.java** file, turn on the line numbers again (right-click the vertical bar left of the Editor window and select **Show Line Numbers**). Expand any compressed lines. If you have a blank line at the top, remove that blank line. Here's how it works:

OBSERVE:

```
1 import java.awt.*;
2 import javax.swing.*;
3
```

In the code above, the imports have an * (asterisk) at the end of the package names. This tells Java that you might want to use *any* of the classes in the **java.awt** or **javax.swing** packages. With the *, your code can then use any class in the package. Java will retrieve the class at the same time, whether it used the import with the * or the specific class name.

OBSERVE:

```
4 public class HelloWorld2 extends JApplet
5 {
```

The code above is a **JApplet**; our first was an **Applet**. The **JApplet** class is in the **javax.swing** package, so we imported that package instead of **java.applet**.

OBSERVE:

```
6     public void init()
7     {
```

The code above is the initial definition of the **init()** method.

OBSERVE:

```
8         Container contentPane = getContentPane();
```

The code above gets the **Container** from the **JApplet** instead of from the **Graphics** area used in **Applets**.

OBSERVE:

```
9             JLabel label = new JLabel("Hello Again, World!", SwingConstants.CENTE
R);
```

The code above creates a **JLabel** and then writes "Hello Again, World!" on the label, rather than into the **Graphics** area.

OBSERVE:

```
10            contentPane.add(label);
```

The line above adds the label to the Container from the **JApplet**.

OBSERVE:

```
11        }
12    }
```

The lines of code above end the definition of the **init()** method, then end the definition of the **HelloWorld2** class. Notice the indentation that enables you to see the matching braces more easily:

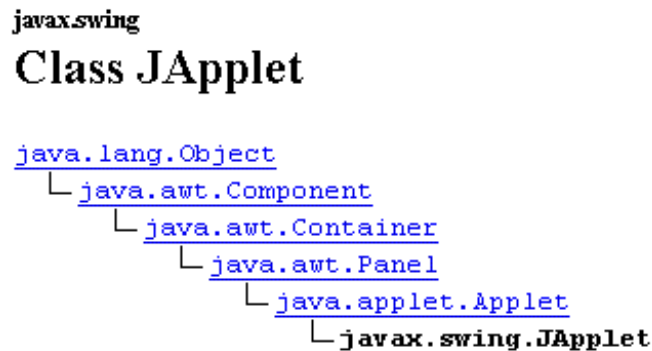
The line 7 { matches up with line 11's } and
the line 5 { matches up with line 12's }.

This indentation is not required by Java, but it's good form for programming and allows better readability.

Though it may not seem like it, the biggest difference between these two Applets is that one is an **Applet** and the other is a **JApplet**. Both classes are Applets because **JApplet** inherits from **Applet**. We know this is the case by checking in the API. Look at the **import** statements. They tell Java where to look for the classes that you didn't write, and it's also where *you* can find the same information.

Clicking on the top left corner of any class's API page will show you the class's *inheritance tree*.

API Go to the [JApplet](#) class in the API and check it out! The JApplet inheritance tree looks like this:



This tells you that the class **JApplet** inherits from the **Applet** class, which inherits from **Panel**, which inherits from **Container**, which inherits from **Component**, which inherits from **Object**.

Everything located *above* a specific class in an inheritance tree is referred to as the class's *super* or *ancestor* (the one directly above it is referred to as its *parent*). Everything located *below* a specific class in an inheritance tree is referred to as the class's *sub* or *subclass* (also referred to as the class's *child*).

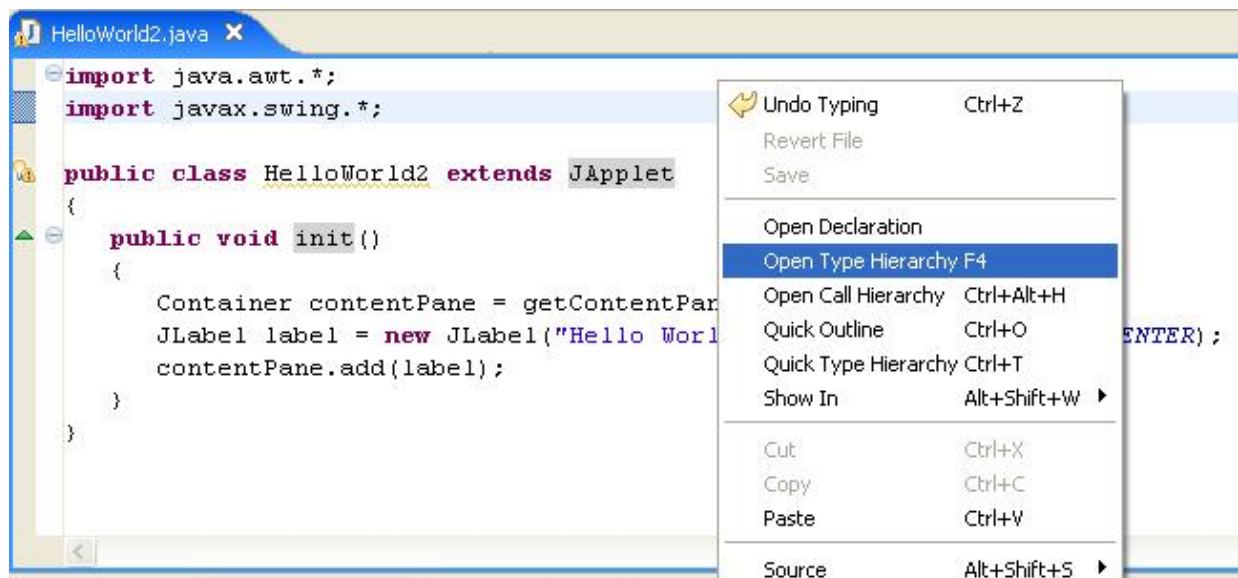
So, **JApplet** has a *parent* (or *super*) of **Applet**, and **Applet** has a *subclass* (or *child*) of **JApplet**.

In future lessons, we'll explore the information in the API in more detail.

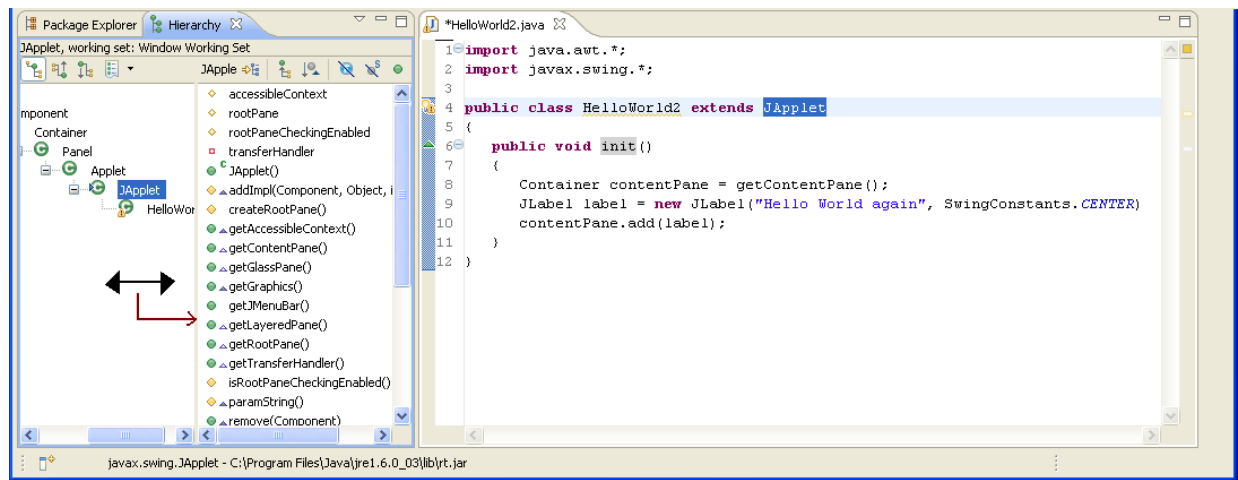
Hierarchy Structure in Eclipse

Another way to view the hierarchy of classes is through the Eclipse hierarchy window.

Make sure you have **HelloWorld2.java** open in Eclipse. Highlight the word **JApplet** in line 4. Right-click it and select **Open Type Hierarchy** from the pop-up menu:



A **Hierarchy** tab appears where the Package Explorer window used to be. It's the same inheritance hierarchy tree that you saw in the API.



In the API you can see that every class inherits from the **Object** class. This is the essence of object-oriented programming.

Again, note that the imports have ***** at the ends of the package names. This tells Java that you might want to use any of the classes in the **java.awt** or **javax.swing** packages. The ***** is used commonly in Computer Science as a *wildcard*. That means anything (that fits the circumstances) can replace it. In this case, the "circumstances" are that the package must be **java.awt** or **javax.swing**—for example, importing **java.awt.*** means you can use any object from the **java.awt** package.

But note this exception: package wildcards only work for classes. Specifically, if you have two packages, say **java.awt** and **java.awt.event**, **import java.awt.*** does *not* get all of the classes in the package **java.awt.event**. You would need to import both **java.awt.*** and **java.awt.event.*** to get the classes from both packages.

Phew! That's it for this lesson 2! Wait a sec, let Duke take a picture of you to capture this moment of accomplishment!



You're looking great—I can't wait to see you in the next lesson!

Remember: once you finish the lesson, go back to the syllabus page by clicking the **Syllabus** tab above and complete the Quiz(zes) and Project(s).

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.

Applets

Applets

We have already created and run a basic Applet. An Applet is a class that's provided for us by Java. All object-oriented programming depends on classes. Specifically, we **define Classes of Objects** so we can use them in our programs. Programmers also create and define new classes for their own specific purposes. Still other classes are **imported** from Java. You can check out the various types of class functions in the API. In this lesson, we'll focus primarily on the most commonly used Classes in Java—**Applets**.

What Can Applets Do?

Our first Applet printed "Hello World." In Java, a *string* of characters (In English, we would identify those characters as words, phrases, or sentences) is defined within a Java Class of its own called the **String** Class. We create **Strings** in Java using double quotation marks: **"This is a string"**. Whatever you put inside the quotation marks is quoted exactly, becoming the String. (This is the only place in Java programming where we can get away with a typo!)

So, in the **paint()** method of our HelloWorld Applet, we told the **Graphics** Class (using g) to use methods **drawString()** and **drawRect()** to print a **String** ("Hello World") and draw a rectangle. Take a look at HelloWorld's **paint()** method:

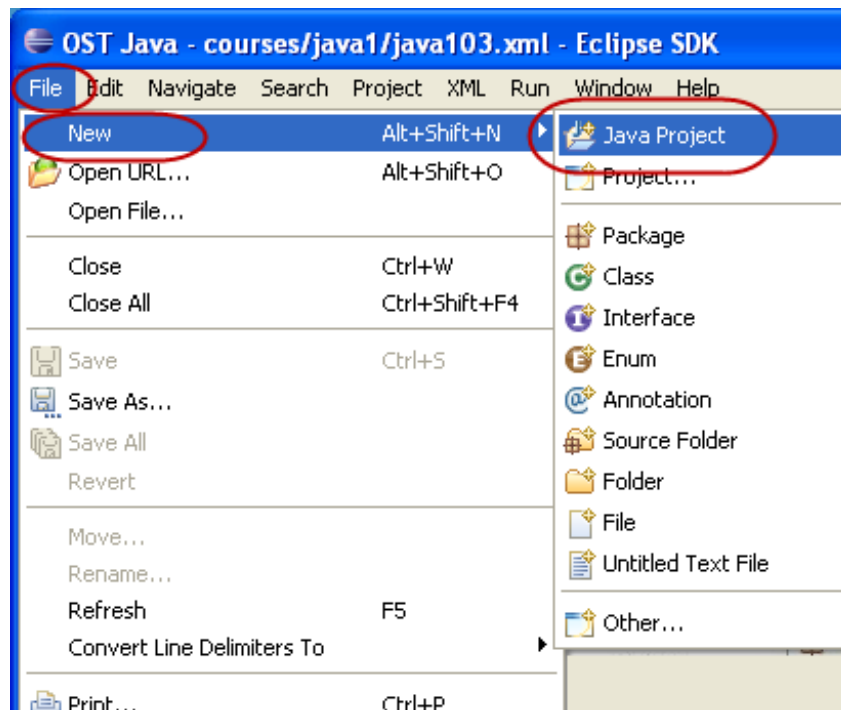
OBSERVE:

```
g.drawString("Hello World!", 5, 15); // put String "Hello World" at (x,y) loc
ation of (5,15)
and
g.drawRect(0, 0, 100, 100); // drew a Rectangle with top-left corner at (0,0)
with width and height of 100
```

So, what else can Applets do? Let's create a new Eclipse project and find out.

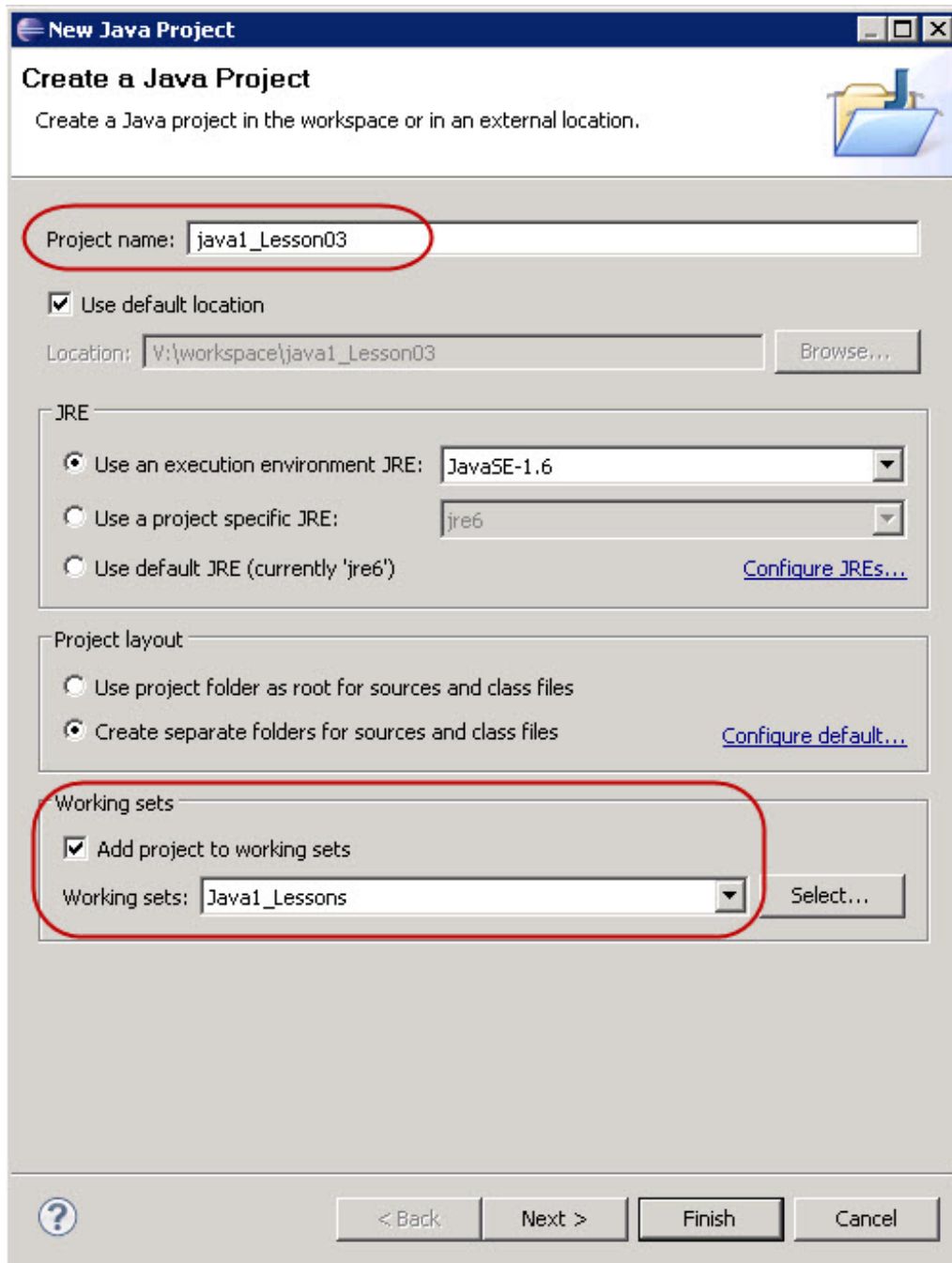
Generally, programmers put their code into source (src) folders, so Eclipse makes **src** folders for each Java project too. Go ahead and create an Eclipse project to hold your classes for **java1_Lesson03**. We did this before in Lesson 1, but in case you need a little help remembering how, follow these steps:

Select **File | New | Java Project**:



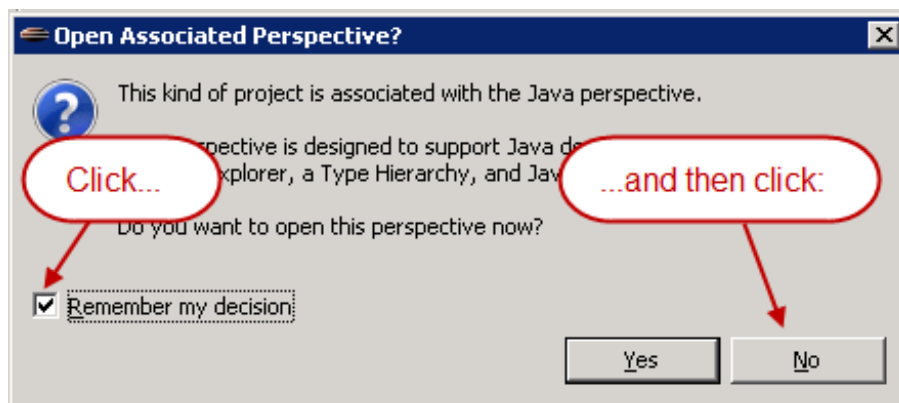
In the New Java Project window, name your project **java1_Lesson03**, add it to the **Java1_Lessons** working

set, and click **Finish**:




The "New Java Project" dialog box is shown. It has a title bar "New Java Project" and a subtitle "Create a Java Project". Below the subtitle is the text "Create a Java project in the workspace or in an external location." and a folder icon. The dialog is divided into several sections. The "Project name:" field contains "java1_Lesson03" and is circled in red. Below it is a checked checkbox "Use default location" and a "Location:" field containing "V:\workspace\java1_Lesson03" with a "Browse..." button. The "JRE" section has three radio buttons: "Use an execution environment JRE:" (selected), "Use a project specific JRE:", and "Use default JRE (currently 'jre6')". The first radio button has a dropdown menu showing "JavaSE-1.6". The second radio button has a dropdown menu showing "jre6". There is a "Configure JREs..." link. The "Project layout" section has two radio buttons: "Use project folder as root for sources and class files" and "Create separate folders for sources and class files" (selected). There is a "Configure default..." link. The "Working sets" section has a checked checkbox "Add project to working sets" and a "Working sets:" dropdown menu showing "Java1_Lessons". There is a "Select..." button. At the bottom are buttons for "< Back", "Next >", "Finish", and "Cancel".

If you see the option to "Open Associated Perspective," check the **Remember my decision** box and then click **No**. We want to keep our own perspective environment.



The "Open Associated Perspective?" dialog box is shown. It has a title bar "Open Associated Perspective?" and a subtitle "This kind of project is associated with the Java perspective." Below the subtitle is a question mark icon and the text "This kind of project is associated with the Java perspective." and "Do you want to open this perspective now?". There is a checked checkbox "Remember my decision:". There are "Yes" and "No" buttons. Red arrows point from the "Remember my decision:" checkbox to the "Yes" button and from the "No" button to the "Remember my decision:" checkbox. Red text "Click..." is in a red oval pointing to the "Remember my decision:" checkbox. Red text "...and then click:" is in a red oval pointing to the "No" button.

If you forget to click **No** and your window goes crazy, remember you always have your "panic button": 

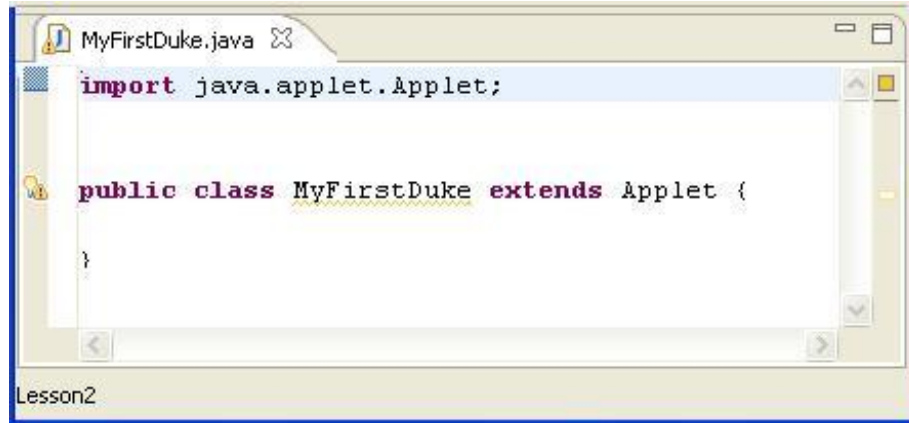
This button will always bring you back to the proper perspective.

You should see **java1_Lesson03** in your Package Explorer Window.

Now, make a new Java Class in the java1_Lesson03 project file.

1. Select **java1_Lesson03** in the Package Explorer so it's highlighted.
2. Right-click **java1_Lesson03** in the Package Explorer, or select **File** from the Eclipse Menu, and then choose **New | Class**.
3. Name it **MyFirstDuke**, make the Superclass **java.applet.Applet**, and click **Finish**.

You'll see this code in your Editor workspace window:





Because you specified that the Superclass was **java.applet.Applet**, the Eclipse *code generator* performed these helpful tasks for you:

- imported java.applet.Applet
- extended Applet

Now, type the **MyFirstDuke** java Class as shown:

CODE TO TYPE:
<pre>import java.applet.Applet; import java.awt.*; public class MyFirstDuke extends Applet { public void paint(Graphics graph) { Image action = getImage(getDocumentBase(), "../images/duke/dukeWave.gif"); graph.drawImage(action, 10, 10, Color.white, this); graph.drawString("I am a waving duke", 10, 130); } }</pre>

 Save and run it.

Oops! 

Hey, wait a minute! We told Java to get an image and draw it—we see "I am a waving Duke," but no Duke! Why can't we see the image?



Sometimes Java will tell us about errors, and sometimes we have to catch them ourselves. Usually if Java *can* run the code, it will. (For example, Java *cannot* run specific code if it's unable to find a Java class that you told it to use.) In this situation, Java did what it could, but it didn't draw the image we told it to, because it was unable to find it. Java probably couldn't find it because I hadn't given it to you yet.

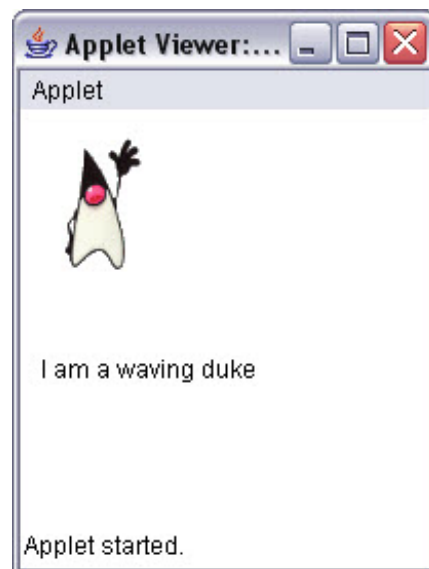
Getting Images

Let me give that image to you now. [Click here](#) to get the image files you'll need for this lesson. When prompted for a working set, select **Java1_Lessons**.

Now you have an **/images** folder that you can use for the entire course. It should be listed under Java1_Lessons with your java1_Lesson01 and java1_Lesson03 projects in the Package Explorer view.

- Open the **/images** folder to see the **/duke** subfolder containing the images.
- Go back to your java1_Lesson03 Project and click on **MyFirstDuke.java** (in the default package).
- Run the **MyFirstDuke** Applet again—right-click **MyFirstDuke.java** and select **Run As | Java Applet**.

Now you'll see this:



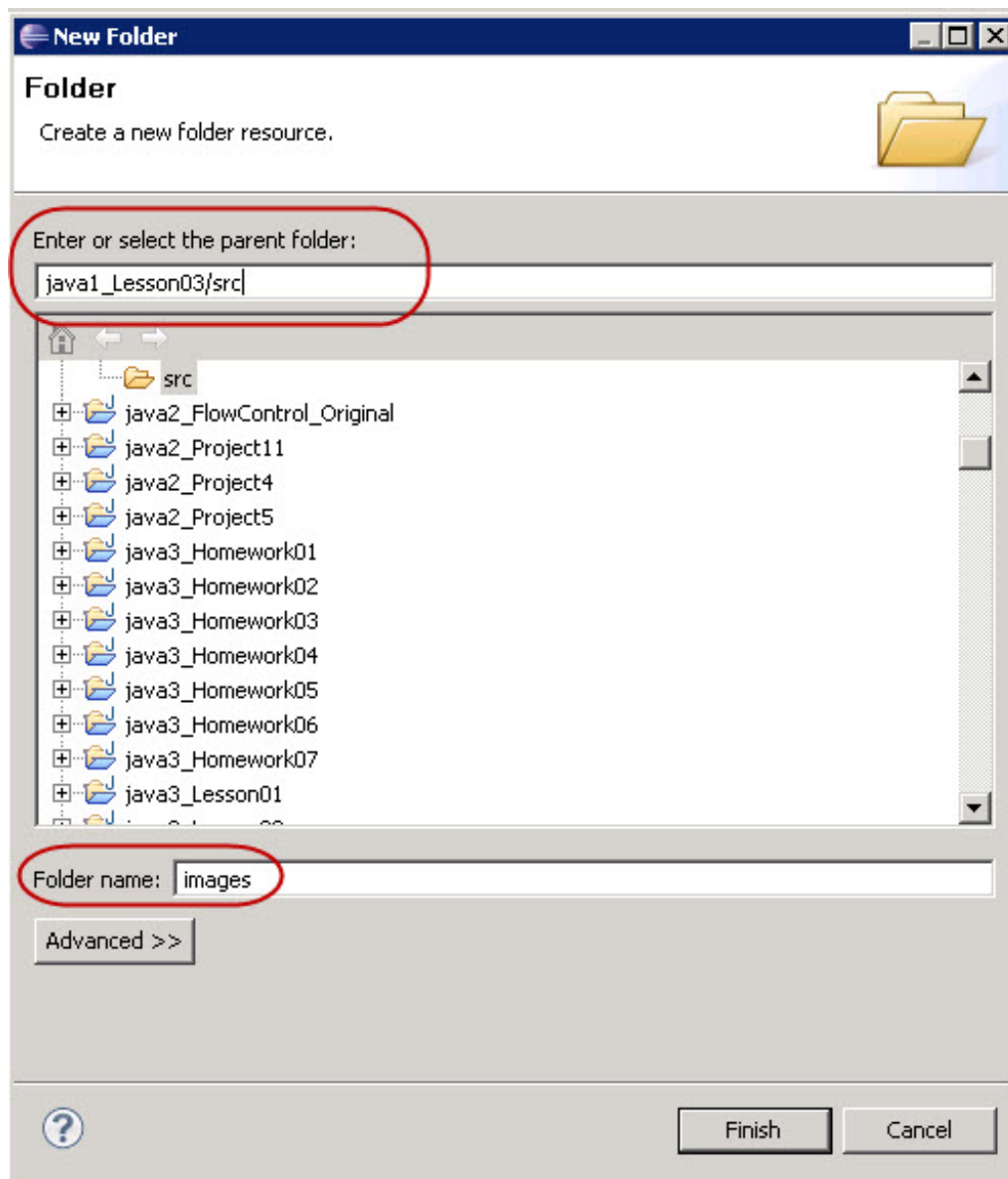
Ahh, much better! Now, look at the code. Notice that the compiler finds the image using the following line:

OBSERVE:

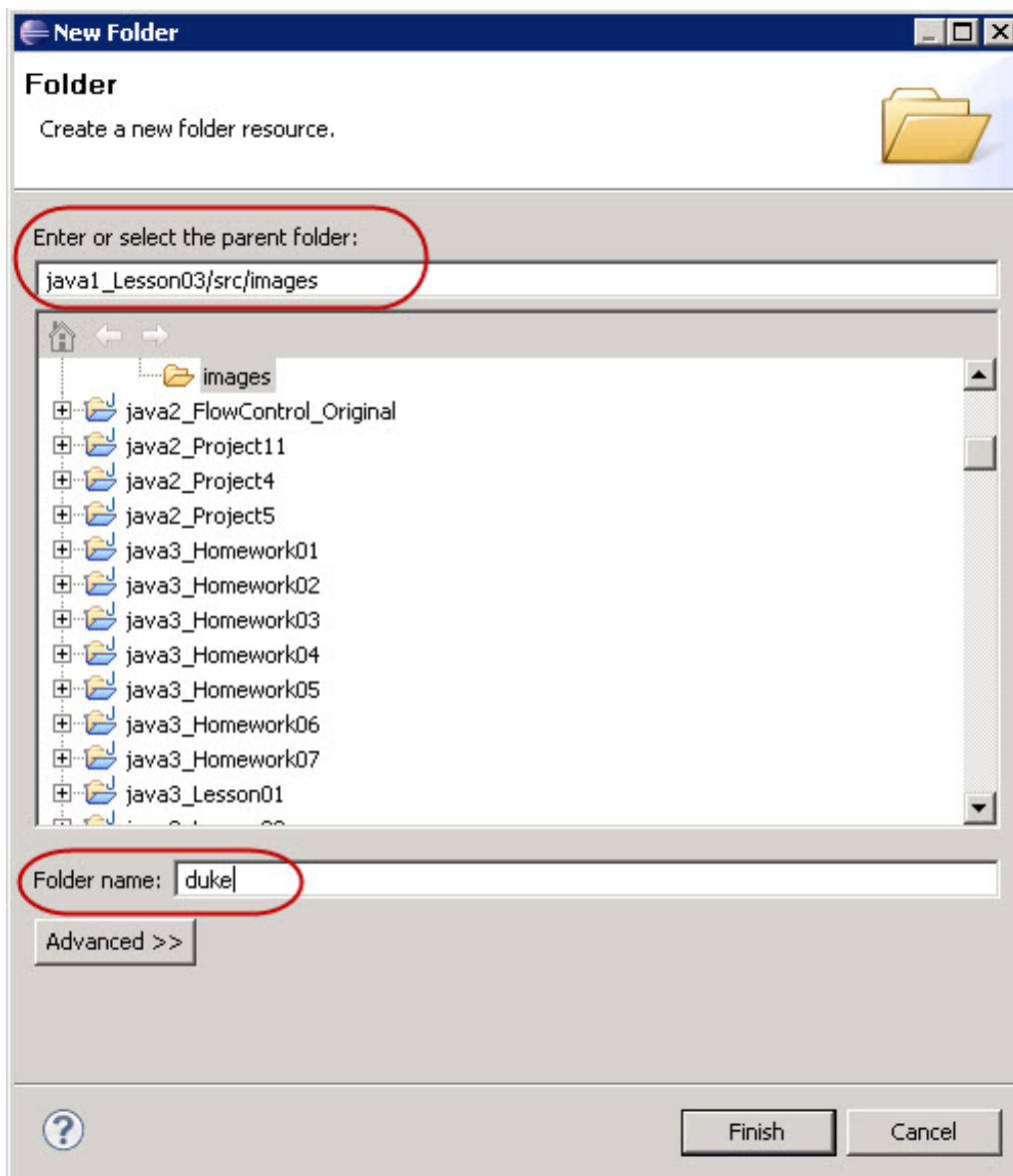
```
Image action = getImage(getDocumentBase(), "../..//images/duke/dukeWave.gif");
```

The **two dots** are important. We need to direct Java to the right place. Each pair of dots tells Java to go up one directory. So Java goes **up two directories** to **java1_Lesson03** and the main **Java1_Lessons**, then down through the **/images** and **/duke** directories, to the **dukeWave.gif** file.

Follow the numbers 1 through 6 on this chart to see how Java finds the image from the MyFirstDuke.java class:



1. Right-click the new **/images** folder to see the popup menu.
2. Choose **New | Folder** (notice again, I said **Folder**).
3. Name it **duke**.
4. Click **Finish**.



You'll see **images.duke**. And now to make the copy:

1. Go to the **images** folder that we originally downloaded, then go into the **duke** subfolder.
2. Right-click on **dukeWave.gif** and choose **Copy**.
3. Right-click your new **images.duke** folder and choose **Paste**.

You'll see the copy now. Finally, get rid of the dots by giving the new path to the file you just placed:

CODE TO TYPE:

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

public class MyFirstDuke extends Applet {

    public void paint(Graphics graph) {

        Image action = getImage(getDocumentBase(),"images/duke/dukeWave.gif");
        graph.drawImage(action, 10, 10, Color.white, this);

        graph.drawString("I am a waving duke", 10,130);

    }
}
```



Save and run it.

Applet Uses Other Classes

There are a few more interesting characteristics for us to explore in the Class. Open your **MyFirstDuke.java** file in the Package Explorer, if it's not already open.

In an earlier lesson, we learned that every class inherits from the Class **Object**. We also learned that we had to import packages so Java could find classes. Another important accepted practice or *convention* when using Classes is that they always begin with capital letters.

Look through the **MyFirstDuke** code in the Editor Workspace for words beginning with capital letters:

OBSERVE:

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

public class MyFirstDuke extends Applet {

    public void paint(Graphics graph) {

        Image action = getImage(getDocumentBase(),"../images/duke/dukeWave.gif");
        graph.drawImage(action, 10, 10, Color.white, this);

        graph.drawString("I am a waving duke", 10,130);
    }
}
```

Okay, we cheated. **drawString** does not begin with a capital letter, but it *is* using the **String** class.

We learned earlier that Java will not run if it cannot find the Classes it needs. We told the **MyFirstDuke** class to **import** the **Applet** class and we are writing the **MyFirstDuke** class in the code, so Java can see them. We saw that the **Graphics** class was imported if we used the *wildcard* import of **java.awt.***.

So, what about **Image** and **Color** and **String**? Where are we importing them?

What if we specifically say **import java.awt.Graphics**?

CODE TO TYPE:

```
import java.applet.Applet;
import java.awt.Graphics;


public class MyFirstDuke extends Applet {

    public void paint(Graphics graph) {

        Image action = getImage(getDocumentBase(),"../images/duke/dukeWave.gif");
        graph.drawImage(action, 10, 10, Color.white, this);

        graph.drawString("I am a waving duke", 10,130);
    }
}
```

Remember, if you see **+import java.applet.Applet;** or **+import java.awt.*;** in the imports, click on the encircled + sign.

Do you see the  and the wavy lines (///) under **Image** and **Color**?

Put your mouse over the . It says that **Image cannot be resolved to a type** and **Color cannot be resolved**.

Modify the import section as shown:

CODE TO TYPE:

```
import java.applet.Applet;
import java.awt.Image;

public class MyFirstDuke extends Applet {

    public void paint(Graphics graph) {

        Image action = getImage(getDocumentBase(),"../images/duke/dukeWave.gif");
        graph.drawImage(action, 10, 10, Color.white, this);

        graph.drawString("I am a waving duke", 10,130);
    }
}
```

Now the errors are on **Graphics** and **Color**. Do you know what to do and the reason behind that solution?

Modify the import section as shown:

CODE TO TYPE:

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;

public class MyFirstDuke extends Applet {

    public void paint(Graphics graph) {

        Image action = getImage(getDocumentBase(),"../images/duke/dukeWave.gif");
        graph.drawImage(action, 10, 10, Color.white, this);

        graph.drawString("I am a waving duke", 10,130);
    }
}
```

Ahh. All is well again. But it's easier to use the wildcard and get the entire java.awt package!

Let's give it a try just to make sure:

API See the API for the [java.awt package](#).

Scroll down to the **Class Summary**. (There are a lot of classes there.) Then, scroll down to see that the package specifically contains the **Color**, **Graphics**, and **Image** classes.

The wildcard (*) can be very handy.

Finally, let's go over the **String**. There's one special package that you never need to import. Java imports it by default. That package is **java.lang**. The class **String** is in the API in **java.lang**. Let's take a look at it.

API Go to the [top-level API page](#). Notice this is the top-level API page to *all* of the packages (Header of **Packages**). Scroll down to the **java.lang** package and click it. Then, scroll down to the **Class Summary** and find the **String** class.

Tip Classes in the **java.lang** package do not need to be imported.

As you scrolled down, you may have noticed the **System** class as well (it's just below the **String** class). This may help you to see why the **java.lang** package doesn't need to be imported. The **System** class will always be needed behind the scenes.

When programming with Java, the API will be your most useful tool, so definitely get friendly with it!

In the next lesson, we'll experiment with our **MyFirstDuke** class to see how Applets work. See you there!



*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.*

An Applet's Life Cycle

Applets Continued

In this lesson, we're going to examine the life cycle of an Applet. We're also going to start encapsulating tasks into methods we create ourselves, instead of putting all of them into the `paint()` method.

Applet Life Cycle

Let's get familiar with the Applet's life cycle. Applets are born, they live fulfilling lives, and then they die. In this lesson, we'll delve into this cyber-miracle more deeply.

Every time you open an Applet, you're opening an *instance*, or *object*, from the **Applet** class. I know, I keep repeating this, but I will spare no redundancy if it helps you get this stuff down! Hence we'll address objects in still greater detail in future lessons. But in this lesson, we want to look at instances by observing an Applet's life cycle.

In order to create a new Applet, we write a class that *extends* the pre-existing **Applet** class. Your newly defined Java class will be a subclass of **Applet** and thus will inherit all of its capabilities. When your Applet loads (because you tell it to **run**), here's what happens:

- An instance of your Applet's class is created.
- The Applet initializes itself.
- The Applet starts running.

To get a visual representation of this process, we'll incorporate more images. Since we already have an **/images** folder, let's use the images in it for this lesson as well.

We'll start out by making an applet that did the same thing as the applet in the last lesson, but this time we'll move some of the tasks that we put inside the `paint()` method into another method. This will be our first attempt at modularity, which is a fancy word for making things more useful to other programs.

Make a new project for Lesson 4, and name it **java1_Lesson04**. Make a new Class in this project named **MySecondDuke** (remember to name the Superclass **java.applet.Applet**). Now let's program!

CODE TO TYPE:

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

public class MySecondDuke extends Applet{

    public void init(){
        setBackground(Color.pink);
    }

    public void paint(Graphics g) {
        Image action = think(g);
        // get the action image for Duke
        g.drawImage(action, 10, 10, Color.white, this);
    }

    public Image think(Graphics graph){
        graph.drawString("I am a thinking Duke", 10,130);
        Image myAction = getImage(getDocumentBase(),"../..//images/duke/thinking.
gif");
        return myAction;
    }
}
```



Save and run it.

Let's break down this code to see what's going on here:

OBSERVE:

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

public class MySecondDuke extends Applet{

    public void init(){
        setBackground(Color.pink);
    }

    public void paint(Graphics g) {
        Image action = think(g);
        // get the action image for Duke
        g.drawImage(action, 10, 10, Color.white, this);
    }

    public Image think(Graphics graph){
        graph.drawString("I am a thinking Duke", 10,130);
        Image myAction = getImage(getDocumentBase(),"../images/duke/thinking.
gif");
        return myAction;
    }
}
```

We're *overriding* the `init()` method from the `Applet` class and the `paint()` method from `java.awt.container.paint`. We've also made a new method called `think()`, which isn't overriding any methods from a superclass. It's called inside of the `paint()` method, it prints "I am a thinking Duke" to the screen and uses the `getImage()` method to grab the image `"../images/duke/thinking.gif"`. Then the graphics object draws that graphic to the screen via the `g.drawImage()` method. Notice we stored the image in a variable named `action`, which is of variable type `Image`. Then the `think()` method returns the action using `return myAction`. Follow the arrows below to see a visual representation:

Breaking down the Applet

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

public class MySecondDuke extends Applet{

    public void init(){
        setBackground(Color.pink);
    }

    public void paint(Graphics g) {
        Image action = think(g);
        // get the action image for Duke
        g.drawImage(action, 10, 10, Color.white, this);
    }

    public Image think(Graphics graph){
        graph.drawString("I am a thinking Duke", 10,130);
        Image myAction = getImage(getDocumentBase(),"../images/duke/thinking.gif");
        return myAction;
    }
}
```

The Graphics Object we named `g` gets picked up by the `think(g)` method, which renames the object `graph` in the method definition. The Graphics object `graph` then uses its `drawString()` method to write "I am a thinking duke." Inside the definition of the `think()` method, we store the variable called `myAction`. And inside of the variable `myAction` is an image called `thinking.gif`. The `think()` method *returns* the image to `paint()`, which stores it in `action`. The Graphics object `g` inside the `paint()` method has a method named `drawImage()`. `drawImage()` draws the contents of the variable `action` which is the `dukewave.gif` image. Got it? Good.

API See the API for the methods available to the [Graphics](#) object.

Now let's get back to the Applet's life cycle. When you **exit** the Applet, it stops and if necessary, Java will do a "cleanup." Cleanup includes things like closing files and removing unnecessary memory access. Before Java, programmers had to do this tedious cleanup work every time they wrote a program. The main steps in an Applet life cycle are:

- *Initialize* itself.
- *Start* running.
- *Stop* running.
- Perform a *final cleanup*, in preparation for being unloaded (*destroyed*).

API Go to the [Applet](#) page, scroll down to the **Method Summary**, and look for methods (the second column has method names):

```
destroy()
init()
start()
stop()
```

These methods are a built-in part of the Applet class and you inherit them when you **extend Applet**.

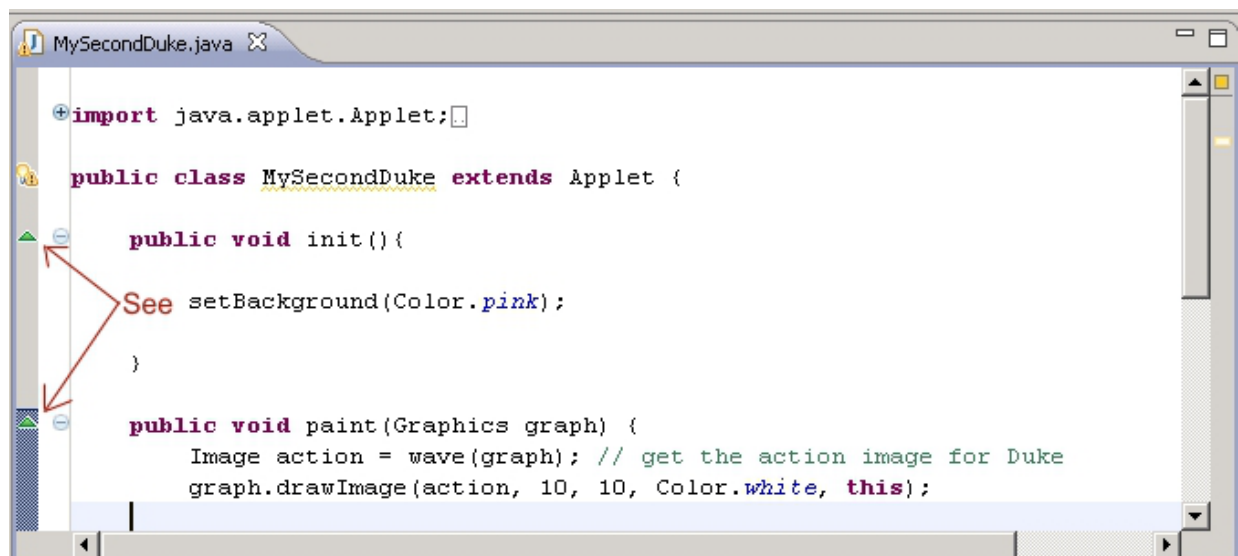
As creator or implementer of a new Class, you can *override* the parents just as we did with **MySecondDuke.java** and all of the classes we've written so far. This means that when you write a method in *your* code with the *same name* as a method of its parent (or any other superclass), the one that *you* wrote is the one that Java will use instead of the one in the parent. Java will use the *most specific* method—that is, the one farthest down in the Applet's hierarchical family tree. In your program, you've actually overridden the **init()** method by making your own method. Try removing the init method and see what happens. Your program should still work, but you won't see the nice shocking pink background anymore.

Note

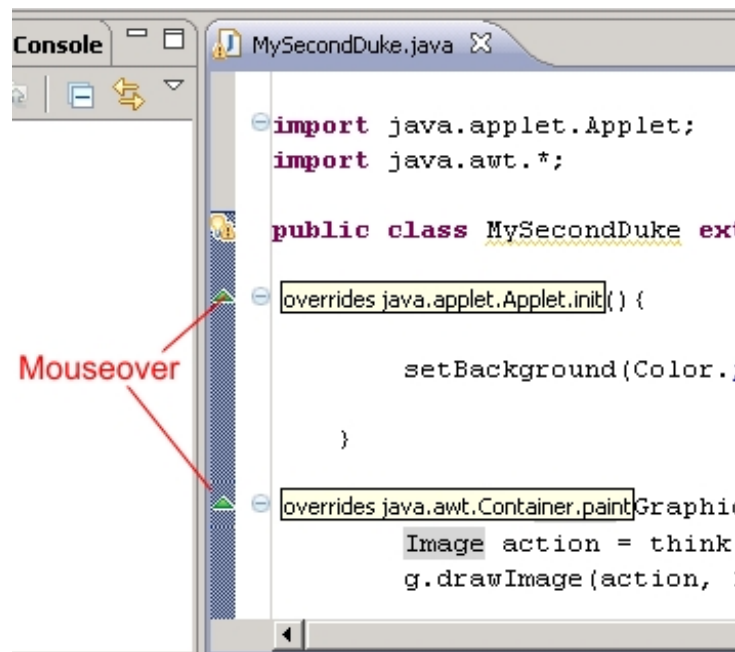
When you override a method, you're using one of the major capabilities of object-oriented programming, *polymorphism*. *Poly* (more than one; many) and *morphism* (the condition of having a specified form)—in computer languages, this means that the same name can be used in different places with different meanings. Although initially this can be confusing, ultimately it's a very useful trait. Trust me.

Note that in our earlier Applets, we did not have an **init()** method, but in this one we do.

Look at your MySecondDuke class in the Workspace. See the  icons in the left column?

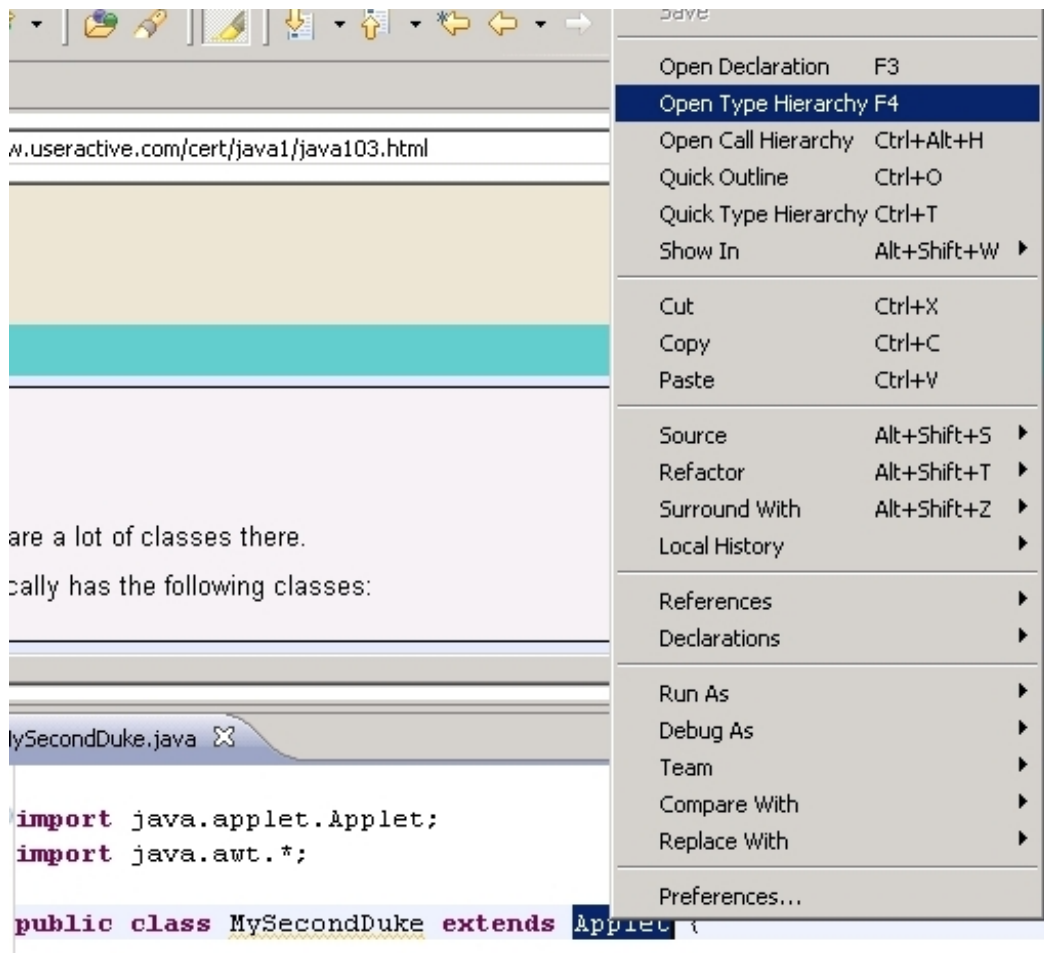


Move your mouse over those icons you see at the `init()` and the `paint()` methods. They tell you exactly *which* of the superclasses has *implemented* the method before. So *whose* code are you overriding?

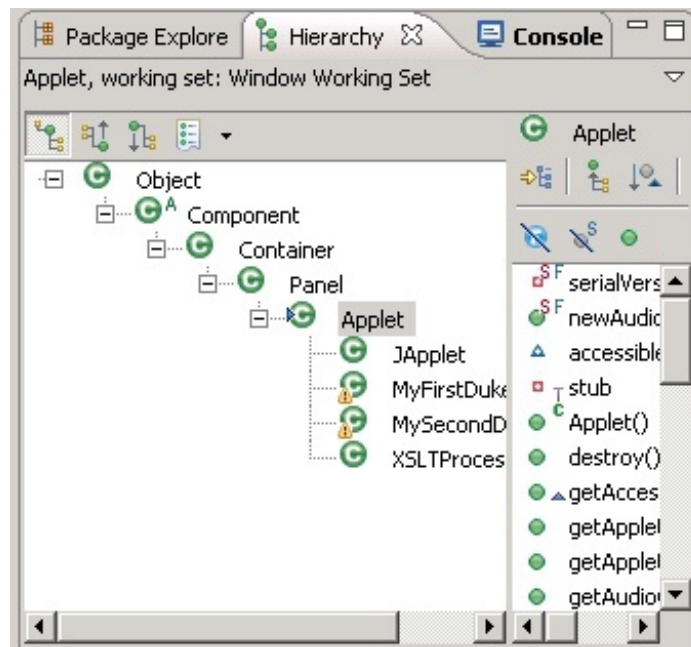


Hmm, `init()` says we are overriding `java.applet.Applet.init`, as we would expect, since we inherit from **Applet**—but `paint()` says we are overriding `java.awt.Container.paint`.

Let's look at our Applet's ancestral trail to learn more. Do an **open type hierarchy** on the Applet. Highlight the word **Applet**, then right-click it and select **Open type hierarchy**.



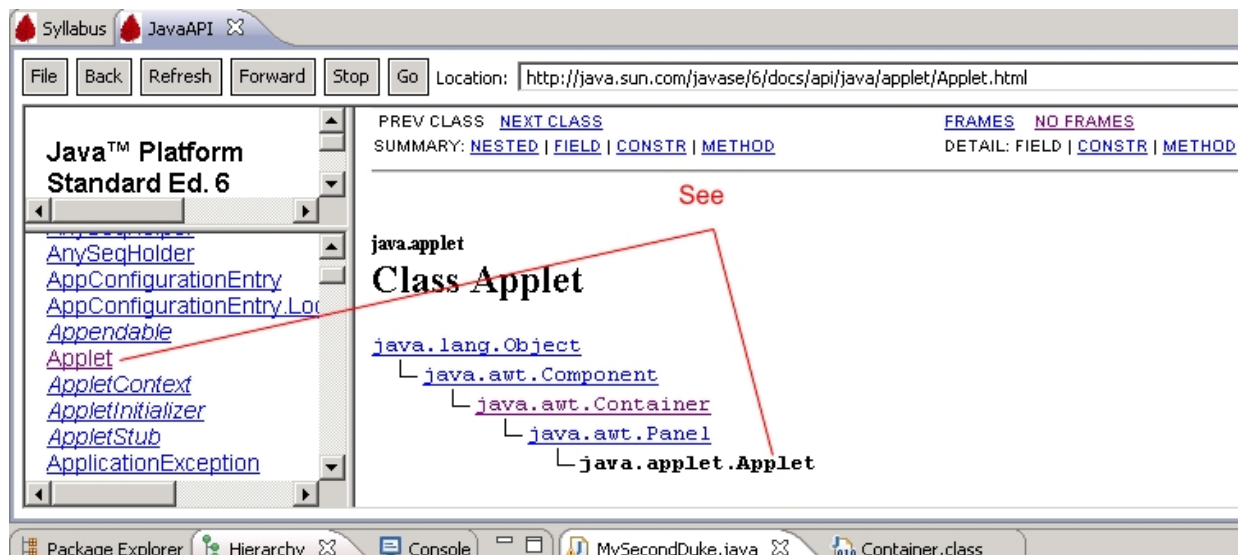
You'll see the following on the left:



Notice that in the hierarchy tree (you might have to scroll up a bit), **Container** is a **superclass** of **Applet** and hence is a **superclass** of any class that **extends Applet**.

So, even though we had not written an **init()** method earlier, *we had one* by default because our Applet inherits the default **init()** from the parent Applet class.

API Find and click the Applet class (look for the word Applet in the left column). You should see something like this:



In the Applet class, scroll down to the **init()** method and click its link.

You'll see (near the bottom of the description) a more detailed description of the method:

"The implementation of this method provided by the Applet class does nothing." Huh.

This must be the reason we didn't see anything special in our previous Applets. Not every Applet has to override every method. In fact, some very simple Applets (like our **HelloWorld**) override no methods at all. Later we'll add more graphical user interface components to our Applets in the **init()** method.

So what about that **paint()** method? How and when does *it* get started?

Well, after the Applet has completed initialization, it displays itself onscreen in the Graphics area of the Applet through the **paint()** method.

The **paint()** method is in the **Container** class, so look in that class for information about it in the API. Since **Container** is a **superclass** of **Applet**, we have a link to it through the Applet class's hierarchy at the top of

its API page.

API At the top of the Applet API page, click on the link to the **Container** class in the Applet's hierarchy. Scroll down under the **Method Summary** provided in the Applet class. Notice the additional frames showing *all* of the methods that the Applet class inherits and from whom they inherit.

Methods inherited from class java.awt. Panel
addNotify

Methods inherited from class java.awt. Container
add , add , add , add , addContainerListener , addImpl , addPropertyChangeListener , addPropertyChangeListener , applyComponentOrientation , areFocusTraversalKeysSet , countComponents , deliverEvent , doLayout , findComponentAt , findComponentAt , getAlignmentX , getAlignmentY , getComponent , getComponentAt , getComponentAt , getComponentCount , getComponents , getComponentZOrder , getContainerListeners , getFocusTraversalKeys , getFocusTraversalPolicy , getInsets , getLayout , getListeners , getMaximumSize , getMinimumSize , getMousePosition , getPreferredSize , insets , invalidate , isAncestorOf , isFocusCycleRoot , isFocusCycleRoot , isFocusTraversalPolicyProvider , isFocusTraversalPolicySet , layout , list , list , locate , minimumSize , paint , paintComponents , paramString , preferredSize , print , printComponents , processContainerEvent , processEvent , remove , remove , removeAll , removeContainerListener , removeNotify , setComponentZOrder , setFocusCycleRoot , setFocusTraversalKeys , setFocusTraversalPolicy , setFocusTraversalPolicyProvider , setFont , setLayout , transferFocusBackward , transferFocusDownCycle , update , validate , validateTree

Any class that extends Applet gets the methods defined in the Applet class, *and all* of those methods defined in Applet's superclasses. This trait of object-oriented languages is called *modularization*.

This page probably has more information than we need now, but it is good to know how to follow related links.

Go back to the Applet class page, scroll down to the Methods inherited from class java.awt.Container frame, and click on the **paint()** method link there. .

Again, this is more information than we need at the moment, but notice that the method is *passed* a **Graphics** object on which to **paint**.

We don't paint on the Applet, we paint on this Graphics area. One of the things that the Applet does upon initialization is create this Graphics area to give to us in the paint() method. Once the method is used, we can access this Graphics object via the instance of it that is passed to us.

Look at the code for **MySecondDuke.java**. Notice the first line of the **paint(Graphics g)** method. Now look at **MyFirstDuke.java** (in java1_Lesson03). Notice the first line of the **paint(Graphics graph)** method there. Now, look at the **paint** method specification in the API for **Container**: public void paint(Graphics g).

Note

In all three methods above, **Graphics** appears first because the method must pass a **Graphics** object. This specification tells us the **type** of Object needed.

You can give whatever name you like to the *instance* of the Graphics object that is passed, as long as the same name is used throughout the block of code within braces.

Hence one class may call the Graphics object **g**, while another might call it **graph**. This is another nice thing about *modularity*. You don't have to worry about the name some other programmer gives to an object. Within your own code, variable names are *local*, meaning your names will only be seen by the code that contains them.

Open your **MySecondDuke.java** file in the Editor, if it's not still open. Edit it as shown:

CODE TO TYPE:

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

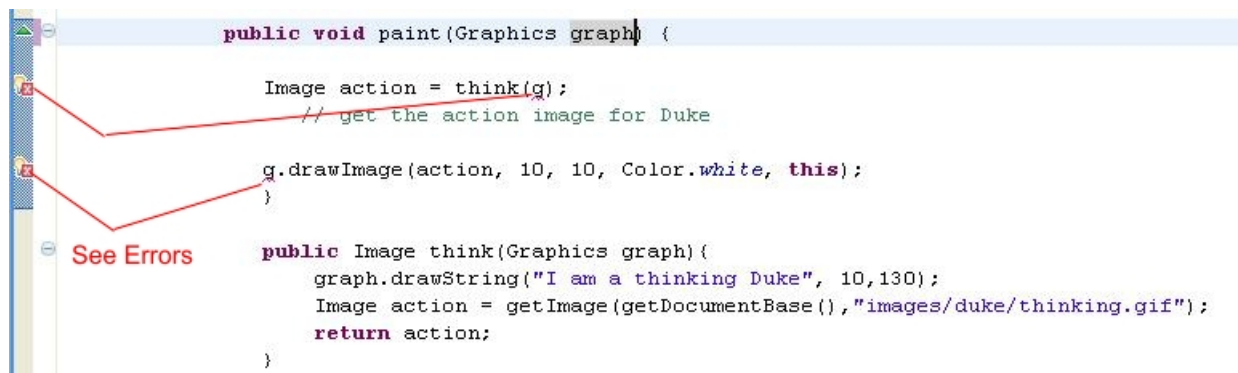
public class MySecondDuke extends Applet{

    public void init(){
        setBackground(Color.pink);
    }

    public void paint(Graphics graph) {
        Image action = think(g);
        // get the action image for Duke
        g.drawImage(action, 10, 10, Color.white, this);
    }

    public Image think(Graphics graph){
        graph.drawString("I am a thinking Duke", 10,130);
        Image action = getImage(getDocumentBase(),"../images/duke/thinking.gif");
        return action;
    }
}
```

It looks like we have some errors to consider.



If you move your mouse over the error symbols, they both say "g cannot be resolved."

To fix it, change the underlined **gs** to **graph** in the **paint()** method.

(You could also change the code back to the way it was initially—Java will accept either fix.)



Save and run it, just to make sure everything still works.

Phew! I'm glad we fixed that. The issue there was one of *consistency*. When we changed the parameter to "graph," Java didn't know what "g" represented anymore. Notice, however, that the block of code defined for **public Image think(Graphics graph){** can specify whatever variable name we want. The {} (braces) designate the *scope* within which a variable name is known. In other words, other parts of the program know nothing about **g** because **g** is inside the paint() method's scope {}, which is inside of the rest of the program.

And of course, the **paint** method does what it says: it paints.

Once the Applet is initialized, the Applet executes the **start()** method and any other methods you've written.

Note

When you draw something *after* the start of the Applet, use the method **repaint()** rather than **paint**. **repaint()** will clear the screen and then call the paint method so that it doesn't paint over previous material.

Adding Methods

Let's work on an Applet that has more capabilities and learn more about the Applet life cycle. There were three actions

for Duke in **MySecondDuke.java**. Two of them, **init()** and **paint(Graphics g)**, were inherited **Applet** methods. We made the third method, **think(Graphics graph)**, from scratch, specifically for our Duke Applet.

This is another example of modularity and good programming practice. We changed **MyFirstDuke** (which stored all of its tasks in the paint method) into a program with separate methods for separate actions. By incorporating this modularity to methods, we increase our program's flexibility.

Control

While some things (methods, objects, attributes) are inherited from the Applet, *you* have the power to control the action of the Applet as well.

In **MySecondDuke.java**--> Now, we'll make a new class that defines more actions and hence allow Duke to do more stuff. Don't worry if you don't understand all of the code being used in this Applet. We'll explain it all later.

In your **java1_Lesson04/src** folder, create a new class named **Duke**, with the Superclass **java.applet.Applet**.

Type the following into **Duke.java**:

CODE TO TYPE:

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

public class Duke extends Applet {
    int test = 0;
    Image action;

    public void paint(Graphics g) {

        switch (test%3) {
            case 0: action= this.write(g); break;
            case 1: action= this.think(g); break;
            case 2: action= this.wave(g); break;
        }

        g.drawImage(action, 10, 10, Color.white, this);

        test = test + 1;
        // Show that Restart repaints to make a new action
    }

    public Image write(Graphics graph) {
        graph.drawString("I am a writing Duke", 10, 130);
        Image myAction = getImage(getDocumentBase(), "../images/duke/penduke.gif");
        return myAction;
    }

    public Image think(Graphics graph) {
        graph.drawString("I am a thinking Duke", 10, 130);
        Image myAction = getImage(getDocumentBase(), "../images/duke/thinking.gif");
        return myAction;
    }

    public Image wave(Graphics graph) {
        graph.drawString("I am a waving Duke", 10, 130);
        Image myAction = getImage(getDocumentBase(), "../images/duke/dukeWave.gif");
        return myAction;
    }
}
```

Before we run it, let's take a closer look at the code:

OBSERVE:

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

public class Duke extends Applet {
    int test = 0;
    Image action;

    public void paint(Graphics g) {

        switch (test%3) {
            case 0: action= this.write(g); break;
            case 1: action= this.think(g); break;
            case 2: action= this.wave(g); break;
        }

        g.drawImage(action, 10, 10, Color.white, this);

        test = test + 1;
        // Show that Restart repaints to make a new action
    }

    public Image write(Graphics graph){
        graph.drawString("I am a writing Duke", 10,130);
        Image myAction = getImage(getDocumentBase(),"../images/duke/penduke.g
if");
        return myAction;
    }

    public Image think(Graphics graph){
        graph.drawString("I am a thinking Duke", 10,130);
        Image myAction = getImage(getDocumentBase(),"../images/duke/thinking.
gif");
        return myAction;
    }

    public Image wave(Graphics graph){
        graph.drawString("I am a waving Duke", 10,130);
        Image myAction = getImage(getDocumentBase(),"../images/duke/dukeWave.
gif");
        return myAction;
    }
}
```

Essentially, this Applet works like the one we did before, except now we have three methods instead of one. We also added a **switch** statement that takes an integer named **test** (we know it's an integer because we declared it using **int test = 0**;). We mod out by 3 using **test%3** (reads "test mod 3"). "Mod out" means to take the remainder. In this case, it means to divide by 3 and take the remainder, so $5\%3 = 5/3 = (1 \text{ with remainder } 2) = 2$. Then the switch statement gets 0, 1, or 2 successively—if it's 0, it calls the **write()** method, and so on.



Save and run it.



```
case 0: action = this.write(graph); break ;
case 1: action = this.think(graph); break ;
case 2: action = this.wave(graph); break ;
```

Watching a Life

In the life-cycle of an Applet, we see that it is *initialized* only once. So logically, the **init()** method would be run only one time as well. However, Applets are usually seen in browsers. When the user leaves the page—for example, to go to another page—the browser stops and destroys the applet. The state of the applet is not preserved. When the user returns to the page, the browser initializes and starts a new instance of the applet. Similarly, when another window on the computer covers the Applet, the Applet must be repainted when it's fully uncovered again. When the Applet is started again, the **start()** and **paint(Graphics g)** methods (via **repaint()**) are called.

Look for the first action (the one you see now) on the list that we have in our paint method. Notice what's next. In our code, we made it so that each time the Applet is Restarted (and hence repainted), Duke's activity will change in a specific order.

Click **Applet** (in the upper left corner) and select **Restart**. Notice Duke's action and its location on the list.

Click **Applet | Restart** again and observe the change.

Do it again.

The actions go in the sequence listed.

Now, note some other aspects of the applet's life-cycle:

Select **Applet | Stop** (note activity—should be empty).

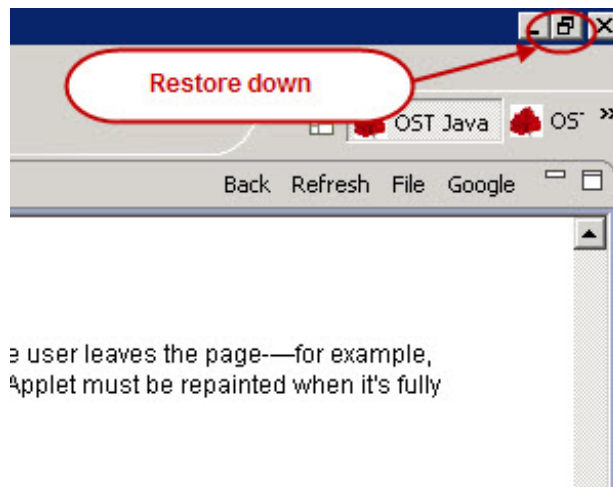
Select **Applet | Start** (note activity—should be next in sequence).

Select **Applet | Stop** (note activity—should be empty).

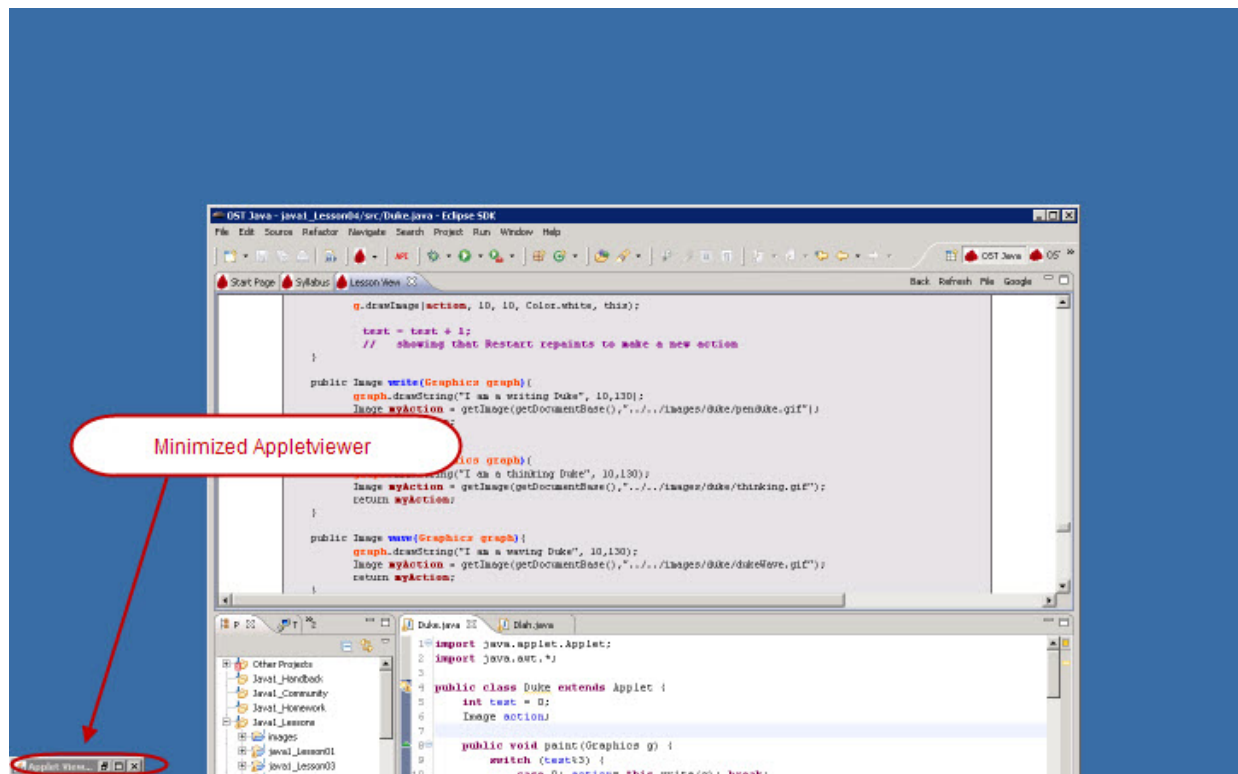
Select **Applet | Restart** (note activity—should be next in sequence).

Keep the applet open for now.

Now, we'll **Restore Down** the lesson window so we can see the Appletview "behind" it. Click the **Restore Down** icon at the upper right of the lesson window:




Minimize the Appletviewer window:



Now get it back by clicking its **Restore Up** icon (note the activity—it should be updated—showing next in sequence).

Now let's open another instance! Notice Duke's action and its location on the list. Move the Appletviewer to the right so we can launch a new instance of the Applet and you can still see it and this lesson.

 Click in the editor window and run the applet again. You now have two Appletviewers running. You can run as many applets as you need in this fashion.

Restart the first one (note activity—should be next in sequence). Second applet does not change.

Restart on the second one (note activity—should be next in sequence). First applet does not change.

Stop on the second one (note activity—should be empty). First applet does not change.

Each time you run a new Applet, you get exactly that—a *new instance* of an Applet. Later lessons will illustrate this further and provide more information about writing **Classes**.

For more on the Applet life cycle, see the [Oracle Tutorial](#).

Congratulations! Your Java skills are getting stronger and stronger.



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.

Decisions, Decisions, Decisions

Program Control Using If Statements

All algorithms are made up of the following *control* constructs, which direct the flow of the program:

- sequences (assignment statements, IO calls)
- repetitions/loops (while, for, do)
- decisions/selections (if/then, switch)
- method invocation

This lesson focuses specifically on control given *decision statements* (sometimes called *conditionals*). There are two types of decision statements:

- if statements
- switch statements

We actually used a switch statement in the last lesson, and we'll cover them again here. But first, let's discuss if statements:

If Statements

An **if** statement consists of three major parts:

OBSERVE:

```
if (boolean)
{
    statements
}
else
{
    statements
}
```

- the **if**: the condition being tested (inside parentheses) *must* result in a *boolean*—a fancy word for "true or false."
- the **statement(s)** to be executed: **if** the condition is **true**—notice Java does *not* use the keyword **then** (it is implied).
- the **else**: (optional) if present, the **statement(s)** that follow it are executed if the condition is **false**.

Note

Java uses parentheses () for:

- conditions in decision statements (**if** and **switch**).
- formal and actual method parameters.
- precedence in math and logic expressions.

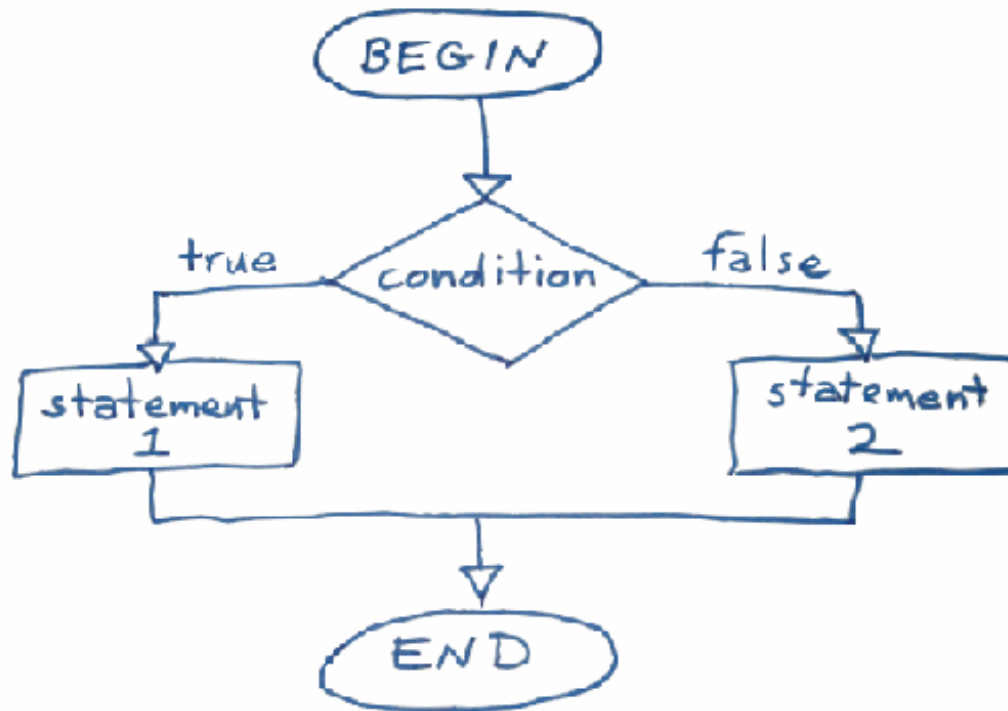
Here is an example of the syntax used when only one statement is to be performed. In this example, **statement1** is executed if the **condition** is true, OR **statement2** is executed if the **condition** is false. If, after the **if** or **else**, there is only *one* statement to execute, no braces {} are needed around the statement.

OBSERVE:

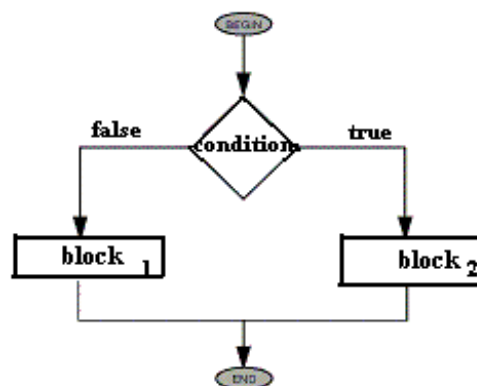
```
if (condition)
    statement1;
else
    statement2;
```

Below is a *flow chart* illustrating the flow of control after an **if** statement. Inside the diamond-shaped figure, a

question is asked. Depending on whether the answer to the question is **false** or **true**, control will go to the left or right respectively. (The rectangles indicate execution statements in the code.) When either one of these is completed, control will then continue sequentially down the remaining code statements.



The "Statements" in the flow chart above could also be *blocks* of statements:



For the remainder of this lesson, after **if** statements, we'll assume they are single statements or blocks of statements (in braces). The braces indicate a *block*, that is, that more than one statement needs to be executed.

Note Java **statements** end in semi-colons (;). There is no ; after a block. The block simply groups the statements within it.

An **else** statement may not even be needed, so a decision might look like this:

OBSERVE:

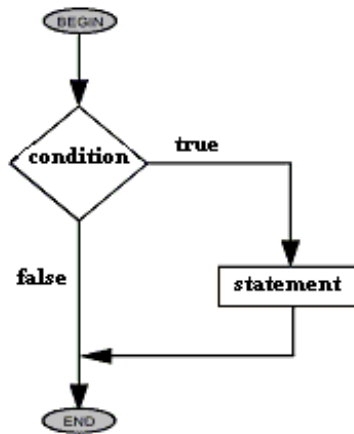
```
if (condition)
    statement1;
```

Remember that spaces and blank lines mean nothing to Java, so the previous code could also be written as a one-liner:

OBSERVE:

```
if (condition) statement1;
```

Those two statements are equivalent and hence both have the same flow chart:



Okay, now let's try an example.

Make a new project in the **Java1_Lessons** working set named **java1_Lesson05**. Make a new class for this project named **TestApplet** that extends `java.applet.Applet` (i.e. the SuperClass is `java.applet.Applet`). Type in the code below:

CODE TO TYPE:

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

public class TestApplet extends Applet {

    public void paint (Graphics g) {

        int myNumber = 1;

        if (myNumber == 1)
        {
            g.drawString("My number matches",10,20);
        }

    }

}
```



Save and run it.

In the **paint()** method, we set the variable **myNumber** to 1. Then we asked if **myNumber** was equal to 1 in the condition of the **if** statement (`myNumber == 1`). In java, like in other languages, checking "is equal to" is done using two equals signs `==`. A single equals sign `=` is used to assign values to variables. In this case, since `"1 == 1"` is **true**, the *statement* `g.drawString("My number matches", 10,10);` was executed and we saw "My number matches" printed to the screen.

Try changing **myNumber = 0** and then run it again. This time it shouldn't print anything.

Now let's try this:

CODE TO TYPE:

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

public class TestApplet extends Applet {

    public void paint (Graphics g) {

        int myNumber = 0;

        if (myNumber == 1)
        {
            g.drawString("My number matches",10,20);
        }
        else {
            g.drawString("My number doesn't match",10,20);
        }
    }
}
```



Save and run it. This time, "My number doesn't match" prints.

Now, try this:

CODE TO TYPE:

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

public class TestApplet extends Applet {

    public void paint (Graphics g) {

        int myNumber = 0;

        if (myNumber == 1);
        {
            g.drawString("My number matches",10,20);
        }
        else {
            g.drawString("My number doesn't match",10,20);
        }
    }
}
```

Save it. Note the word **else** has a wavy underline and a red X in the left column.

When you move the mouse over the **else** error, you see the description **Syntax error on token "else", delete this token**. Since we added a ; at the end of the **if** line, it thinks the **else** is there without an **if**.

And actually, if you remove the **else**, the error will go away. Try it. You'll get a "logical error" which means it won't work the way you intended, but Java won't "yell" at you because, syntactically, it's okay to end the **if** statement without doing anything. A semi-colon ; simply ends a statement. It seems a bit odd because it won't actually DO anything, but it's okay according to Java.

Tip

This may seem kind of obvious, but when you get errors, only follow the suggestion if it makes sense. Otherwise, look at your code until it *does* make sense before changing things!

Remove the ; so the line reads `if(myNumber == 1)` and save the program.

Placement of Block Braces

Indentations and new lines mean nothing to Java. Programmers use them to help make their code easier to read. Semicolons, parentheses, and brackets make it possible for Java to read our code.

Let's look at some more examples.

Make a new Class in your **java1_Lesson05** project named **Driving** that extends `java.applet.Applet`.

CODE TO TYPE:

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

public class Driving extends Applet {
    int age = 15;

    public void paint(Graphics g) {
        if (age > 15)
            g.drawString("Age is " + age, 50, 50);
            g.drawString ("You may drive", 50, 70);
    }
}
```



Save and run it. Now try some different ages to see what happens.

WARNING Before driving a motor vehicle, check the laws in your state!

Because the `g.drawString("You may drive", 50, 70);` line in the `paint()` method is not within an **else** or within a block defined by `{` and `}` for the **if** statement, it always prints. But is that what we meant to happen? We want the **if** statement to determine whether we are old enough to drive or not and that should depend on the value stored in **age**.

Notice again that indentation does not matter to Java. Java is interested only in the placement of the braces. Add braces as shown below:

CODE TO TYPE:

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

public class Driving extends Applet {
    int age = 15;

    public void paint(Graphics g) {
        if (age >15)
        {
            g.drawString("Age is " + age, 50, 50);
            g.drawString ("You may drive", 50, 70);
        }
    }
}
```



Save and run it.

Try some other numbers by changing the value of the **age** variable. (Later in this course we'll take input from a user and then change the value of variables like **age** accordingly)

Let's make it so no matter what age we put into the age variable, we'll get the right output.

CODE TO TYPE:

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

public class Driving extends Applet {
    int age = 15;

    public void paint(Graphics g) {
        if (age > 15)
        {
            g.drawString ("Congratulations!", 50, 50);
            g.drawString ("You may drive", 50, 70);
        }
        else
        {
            g.drawString ("Wait!", 50, 50);
            g.drawString ("You may not drive yet", 50, 70);
        }
        g.drawString("Age is " + age, 50, 90);
    }
}
```



Run it.

Now, you might be wondering if we can simply get rid of **else** and instead just use a sequence of **if** statements. Let's give it a try.

CODE TO TYPE:

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

public class Driving extends Applet {
    int age = 15;

    public void paint(Graphics g) {
        if (age > 15)
        {
            g.drawString ("Congratulations", 50, 50);
            g.drawString ("You may drive", 50, 70);
        }
        if (age < 15)
        {
            g.drawString ("Wait a few years", 50, 50);
            g.drawString ("You may not drive yet", 50, 70);
        }
        g.drawString("Age is " + age, 50, 90);
    }
}
```



Save and run it. What happened? It's an easy mistake to make.

If you have two **ifs**, and you're using less than and greater than signs, make sure that one actually has the value for **equals** as well, otherwise you'll miss the "edge case"—people who are exactly 15! Let's go ahead and fix that:

CODE TO TYPE:

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

public class Driving extends Applet {
    int age = 15;

    public void paint(Graphics g) {
        if (age > 15)
        {
            g.drawString ("Congratulations", 50, 50);
            g.drawString ("You may drive", 50, 70);
        }
        if (age <= 15)
        {
            g.drawString ("Wait a few years", 50, 50);
            g.drawString ("You may not drive yet", 50, 70);
        }
        g.drawString("Age is " + age, 50, 90);
    }
}
```

NOW the two examples are equivalent. Why would we use **elses** rather than two **ifs**? One reason is speed. Java does not have to go back to the age variable in memory to see what its value is a second time if there is an **else**. Or perhaps you, as a programmer, think that two **ifs** will be more efficient or more clear than one. This could be true, but you still need to be careful.

So now that we've talked about the last two parts of the **if** statement, maybe we should discuss the first part a little more!

Comparison Operators and Logic

The first component of **if** statements is the conditional. Is something **true** or **false**? Just like we have the arithmetic operators (-, *, /, and %) to manipulate numbers and change variable values, we also have operators to determine how variable values are related and to **compare** them.

Comparison Operators

We've already used some comparison operators in the examples above, such as <, >, and ==. Here's a more complete list:

>	greater than
<	less than
==	equal to (true when two <i>objects</i> are <i>the same</i> , or two primitive data types have the same value)
!=	not equal to
<=	less than or equal to
>=	greater than or equal to

(More operators exist that deal with comparisons of single *bits* at a time, but that's beyond the scope of this class.)

Let's look more closely at the **==**. Remember that in computer languages, one equals sign tells the compiler to put the value on the right into the address on the left, as we did with **int age = 15**. For *two* equals signs, when the variables on both sides are **instances** of some **Class**, you're asking if the variable on the left side *points to the same object* as the variable on the right side. If the variable has a value that is a primitive data type, then the **==** is comparing to see if the value on the left is the same as the value on the right.

We often have more than one thing to compare when we make decisions. Given this, we need to use additional *logic*. We'll cover computer logic in depth in the second course of this series.

See you in the next lesson...



*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.*

Objects and Classes

Objects

In the last lesson, we created and ran Applets. To reiterate, a running Applet is an example of an **Object**. In object-oriented programming, everything is done through Objects. This lesson will focus on those Objects that people like us in the programming have defined and created them for specific purposes.

What is an Object?

An Object is a combination of **data (properties, variables, fields)** and **actions (functions, methods)**.

An Object has a state of being or **attribute** such as **colorOfEyes**, **profession**, **length**, or **location**, and behaviors or **methods** it can perform to change its state of being, like **getNewJob**, **payAttention**, or **walk**.

You may have noticed that we use some funky words. Well, they're created that way because computer compilers/interpreters do not like spaces in variable and method names. When creating names for their objects, programmers cram the words together without spaces, begin each name with a lower case letter, and capitalize the first letter of each subsequent word in the name, all according to convention.

Sometimes programmers will use more technical terms to describe state and behavior and refer to an Object as a combination of data (properties, variables, fields) and actions (functions, methods).

But it's useful to think of Objects as *real Objects* when designing a program. That the characteristics and capabilities of Objects belong to the very objects that should possess them. In programming terms, the methods and attributes that the Object is capable of are contained within the Object definition.

Inexperienced programmers tend to put information in all kinds of weird places. You want your Objects to be organized clearly and contain the right information, since other programmers might use your Objects too. Like we said, sharing is the greatest benefit of Object Oriented programming!

For example, if I were to ask you what color my cat Missy's eyes were, in what Object might you look to find the answer? (Pretend that we've created all of these as Java Objects):

- Me (since Missy is *my* cat)?
- Humans (since she is mine and I am a human)?
- Eyes?
- Cats?
- Missy?



All of these things are Objects, but which one should have a value for the color of her eyes? Hopefully you recognized that information about the color of the cat Missy's eyes belongs in the Object Missy. The Object Missy should have an **attribute** of **eyeColor** and the value of that attribute would be **green**.

In object-oriented programming, the first thing we do is define Classes of Objects. We define an instance of an Object by letting it have the characteristics and behaviors that belong to that Object.

Objects can also inherit from other Objects. For instance, Missy is a cat. She shares attributes in common with other cats like fur and claws, and actions like purring and hacking up furballs. What does Missy inherit from Cats? Certainly she inherits legs, a tail, whiskers, and such. But what makes Missy uniquely Missy? That would be her specific fur color, eye color, some behaviors, and the like.

If we were to program Missy in Java we'd make a Missy class that creates instances of Missy.

OBSERVE:

```
import java.cats.*

public class Missy extends cats {

    Color colorOfEyes = color.green;
    Color colorOfFur = color.gray;
    Boolean hungry = True;

    public Sound meow() {

        if (hungry) {

            Sound talk = "cry";
        }
        else
        {
            Sound talk = "purrurr";
        }

        return talk;
    }

}
```

What Java syntax did we use to specify inheritance? In fact, you may remember doing something similar before when you extended Applet. In our Duke Applet from the previous lesson:

OBSERVE:

```
public class Duke extends Applet
```

Since our **Duke** inherited from **Applet**, any Objects made from these class templates inherited all of the attributes and actions of the **Applet** class too. Knowing the similarities and differences among Classes will help you decide the best classes for your Objects to inherit from, and also help you define Classes.

Consider our friend Duke again. In the next section, we will enhance our earlier code and make a template for Duke objects. (He is simply *never* going away!)



Classes

The declaration (or definition) of the structure of objects of a certain kind is called a *Class*. A class is a template or a blueprint for creating instances of objects.

The Duke Class we made earlier is a template that creates instances which are specifically Duke Objects. Think of Class as a machine that creates Objects. You don't actually get an Object from a Class until the Java code is executed, but as you'll see in the upcoming example, you can get different instances of Objects from a single Class. We can do that by setting attributes and calling methods that are made available to us by the Class.

To define a Class, we give it *attributes* that define that particular type of Object and thus provide its state (the *variables*) and also the behaviors that the Object is capable of performing (the *methods*).

Note The methods and fields of a Class are sometimes called the Class **Members**

Let's take yet another look at Duke. So far in our program we haven't given Duke any attributes. We've simply printed pictures of Duke doing different things. Let's change that. Let's start giving Duke some attributes. What attributes should we give Duke?

What makes Duke a Duke? Well, Duke is an Applet because he extends Applet. But Duke isn't just an Applet—he has some of his own attributes as well. You can see he always has a pointy head, two arms, a round nose, and a tooth-shaped body. Hmm, does a Duke have to be a boy? Could there be a Duke with a blue nose? It's really all up to you.

Our next task in Java will be to add a new attribute to our Duke. Every Duke has a nose. Let's suppose some Dukes have different color noses. Let's make a Duke class that has an attribute of **noseColor** with two possible nose colors: red and blue. Before we pick Duke's nose (as it were), let's go over some effective techniques to use when designing Objects:

- When choosing **objects/classes**, look for **nouns** in the program. Our class is called **Duke**.
- When choosing **methods**, look for **verbs** in the program. Ours are **wave()**, **write()**, and **think()**.
- When choosing a program's **fields** and **attribute values** look for persistent **characteristics** of the objects. Now we're about to add an attribute called **noseColor**.

Let's make a new project for Lesson 6, named **java1_Lesson06**.

Let's build on the Duke class we created already. Copy the Duke.java file from java1_Lesson04 to java1_Lesson06:

1. Select **java1_Lesson04** in the Package Explorer window.
2. Click the **+** sign to open the Project; if the **default package** is closed, click on its **+** as well.
3. Right-click **Duke.java** and select **Copy**.
4. Right-click the **java1_Lesson06/src** folder and choose **Paste**. You should see a **default package** with **Duke.java** in it.

Now, let's edit our Duke.java and check out how Objects work:

Open the **java1_Lesson06/src** subfolder, and its **default package**. Then, double-click the **Duke.java** program to open it in your Editor Window.

Remember, click the encircled **+** sign if you see this in the imports:

```
+import java.awt.*;
```

You should see:

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

Note

The forward slashes **//** in Java indicate *comments*. You can use them to provide helpful information within your code. When Java runs the code, it ignores everything from the **//** to the end of the line.

This next example will be a little different from stuff we've done before. We won't look at the actions of Duke one at a time, but instead we'll observe them *randomly*.

Edit Duke.java as follows (adding code that looks like **this** and removing code like **this**):

CODE TO TYPE:

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

public class Duke extends Applet {

    int test = 0;
    Image action;
    Color noseColor = Color.red;

    public void paint(Graphics g) {
        // Next line randomly picks just to show that different noses are possible.
        int rint = (int) (Math.random() * 2); // Gives either a 0 or a 1.
        if (rint == 0) {
            noseColor = Color.red;
        } else {
            noseColor = Color.blue;
        }
        // Randomly let this duke do something - one of 3 choices.
        switch ((int) (Math.random() * 3)) // Gives a number between 0 and 2 inclusive.
        {
            case 0: action= this.write(g); break;
            case 1: action= this.think(g); break;
            case 2: action= this.wave(g); break;
        }

        resize(300,300); // Resize the applet window.
        g.drawImage(action, 10, 10, Color.white, this);

        test = test + 1;
        // Show that Restart repaints to make a new action.

        if (noseColor != Color.red) {
            g.drawString("My nose feels funny", 10, 145);
        }
    }

    public Image write(Graphics graph){
        graph.drawString("I am a writing Duke", 10,130);
        if (noseColor == Color.red){
            action = getImage(getDocumentBase(),"../images/duke/penduke.gif");
        } else {
            action = getImage(getDocumentBase(),"../images/duke/penduke2.gif");
        }
        return action;
    }

    public Image think(Graphics graph){
        graph.drawString("I am a thinking Duke", 10,130);
        if (noseColor == Color.red){
            action = getImage(getDocumentBase(),"../images/duke/thinking.gif");
        } else {
            action = getImage(getDocumentBase(),"../images/duke/thinking2.gif");
        }
        return action;
    }


    public Image wave(Graphics graph){
        graph.drawString("I am a waving Duke", 10,130);
        if (noseColor == Color.red){
            action = getImage(getDocumentBase(),"../images/duke/dukeWave.gif");
        }
    }
}
```

```

);
    } else {
        action = getImage(getDocumentBase(), "../../images/duke/dukeWave2.gif");
    }
    return action;
}
}

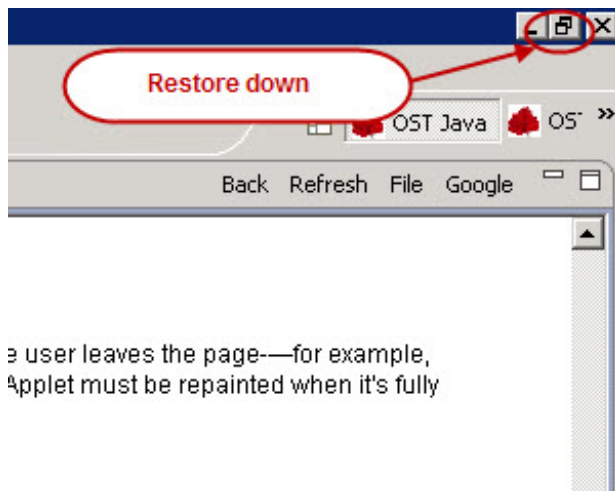
```

Make sure you've typed it all correctly. You shouldn't see any errors; just the one warning about the serializable class at the beginning of the class definition.

 Save and run it.

Note


Notice the applet window is larger. We set that with the statement **resize(300,300)**. The first argument sets the window's width, the second sets the height.





When a user leaves the page—for example, the applet must be repainted when it's fully

Restore down the lesson window. Keep the first Duke

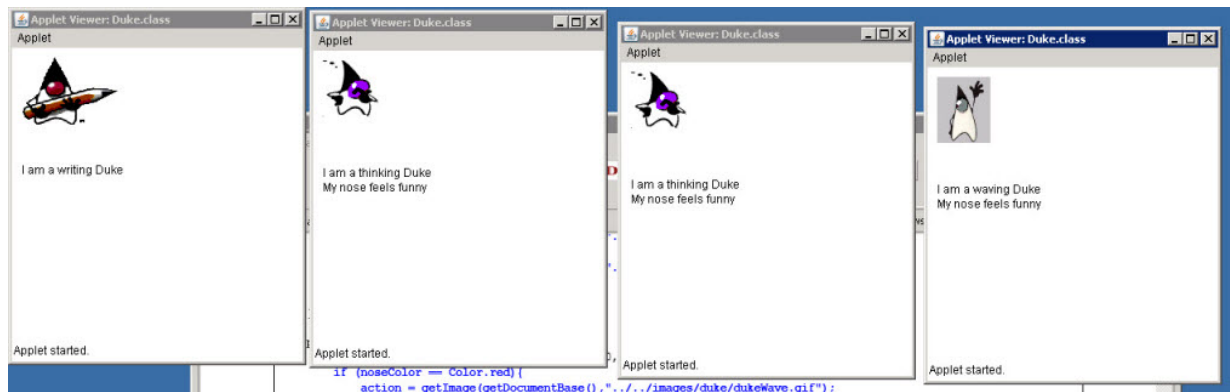
Applet running and move it to the right.

 Run it again to make another one and move it to the right.

 Run it again and move it to the right.

And  Run it yet again.

Now the Applets are all onscreen:



Notice that sometimes we get a Duke with a **red nose** and sometimes we get a Duke with a **blue nose**.

What have we done? This time we didn't *Restart* an existing Applet, but *Ran* a new one. Each time you run an Applet, you're creating a new Applet and hence a new Duke *Object*. Notice in the code that Duke **extends** the Class Applet, so Duke *inherits* from Applet and thus we say Duke is an Applet.

So, every time we run the Applet, we get a *new* Applet Object *and* we are getting a new and different Duke (Note that because our algorithm is random, we might also get the same Duke twice). Each of these windows shows a new *instance* of the Applet and hence a new *instance* of Duke—with all of the inherited characteristics from Applet *and* all of the characteristics specified in the Duke template.

Note

Each time we use a Class to make one of the things that are described by our Class template, we are creating an *instance* of the class (called an *Object*).

We call this *instantiation*.

Specifically, we have a class template named **Duke** which **extends Applet**. The code written describes the class's attributes and methods. Each time we run the code, we generate a brand new Duke possessing all of those attributes and methods.

Since everything in Java is not an Applet, we don't always have Applets to make different instances for us. In order to make instances of Classes, Classes usually provide a special method (a **Constructor**) that is *called or invoked* when someone wants to "construct" an instance of that class. (More on this topic later.)

Let's look at the code in a bit more detail to get a thorough understanding of each part:

OBSERVE:

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

public class Duke extends Applet {

    Image action;
    Color noseColor = Color.red;

    public void paint(Graphics graph) {

        // Next line randomly picks just to show that different noses are possible.
        int rint = (int)(Math.random() * 2); // Gives either a 0 or a 1.
        if (rint == 0) {
            noseColor = Color.red;
        } else {
            noseColor = Color.blue;
        }
        // Randomly let this duke do something - one of 3 choices.
        switch ((int)(Math.random() * 3)) // Gives a number between 0 and 2 in
clusive.
        {
            case 0: action= this.write(graph); break;
            case 1: action= this.think(graph); break;
            case 2: action= this.wave(graph); break;
        }
        graph.drawImage(action, 10, 10, Color.white, this);
        if (noseColor != Color.red){
            graph.drawString("My nose feels funny", 10,145);
        }
    }

    public Image wave(Graphics g) {
        g.drawString("I am a waving Duke", 10,130);
        if (noseColor == Color.red){
            action = getImage(getDocumentBase(),"../images/duke/dukeWave.gif"
);
        } else {
            action = getImage(getDocumentBase(),"../images/duke/dukeWave2.gif
");
        }
        return action;
    }

    public Image write(Graphics g){
        g.drawString("I am a writing Duke", 10,130);>
        if (noseColor == Color.red){
            action = getImage(getDocumentBase(),"../images/duke/penduke.gif"
);
        } else {
            action = getImage(getDocumentBase(),"../images/duke/penduke2.gif"
);
        }
        return action;
    }

    public Image think(Graphics g){
        g.drawString("I am a thinking Duke", 10,130);
        if (noseColor == Color.red){
            action = getImage(getDocumentBase(),"../images/duke/thinking.gif"
);
        } else {
            action = getImage(getDocumentBase(),"../images/duke/thinking2.gif
");
        }
        return action;
    }
}
```

```
}
```

Color noseColor = Color.red; initially sets the noseColor attribute to red. The next couple of references to noseColor are within an *if statement* that checks a random integer we call **rint**, which can be 0 or 1. If it's zero, the noseColor stays red; if it's 1 (that is, "else not 0"), we set the noseColor attribute to blue. The next reference to noseColor appears where we check to see **if the noseColor isn't red**; if it isn't, we print "My nose feels funny". The remaining references to noseColor all check to see if the attribute is set to red or not. If it is, then we put up the picture of Duke with a red nose, otherwise we put up the alternate picture of Duke.

We also changed our switch statement to **work randomly**, rather than in sequence.

Notice in the example above, the variable **rint** is declared inside of the **paint(Graphics graph)** method. This is called a **local variable**. Its scope is solely within the block of code (that is to say, within the **curly braces { }**) in which it was declared. This means that if we were to try to access this variable from anywhere outside of the **paint()** method, we would get a compile error, because the **rint** variable only exists within the **paint()** method.

Local variables can be defined within any block of code defined by curly braces or any conditional or loop that does not need curly braces. However, defining them in a conditional or loop that does not need curly braces would be useless, since they could only be accessed in that single line of code.

Example

```
if(x > 10){  
    int y = 30;  
}  
System.out.println(y);
```

In this short example, the variable **y** only exists within the **if** statement, so the **System.out.println(y)** statement would cause a compile error.

Java Data Types

The Java programming language is strongly typed, which means that all variables must first be declared before they can be used. This involves stating the variable's **type** and name, as you've already seen in these lessons. Like:

Image Action;

and:

Color noseColor = Color.red;

and:

int rint = (int)(Math.random() * 2);

Some of the types are *primitive* data types and some of them are *Object* Data Types. You can tell the difference because Object data types are Capitalized and primitive data types start with a lowercase letter (like int).

Declaring a variable tells your program that a field exists (with whatever name you choose, like **noseColor** or **Action**), it holds data, and has some initial value. If you don't give an initial value, most types have a default initial value. A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other primitive data types. A primitive type is predefined and is named by a reserved keyword. The eight primitive data types supported by the Java programming language are (you don't need to understand all of this but read over it. We'll be covering these in more detail in the second course of this series. Pay particular attention to int, double, and boolean):

byte	An 8-bit signed two's-complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
short	A 16-bit signed two's-complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save

	memory in large arrays, in situations where the memory savings actually matters.
int	A 32-bit signed two's-complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.
long	A 64-bit signed two's-complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you need a range of values wider than those provided by int.
float	A single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section 4.2.3 of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency—for that, use the java.math.BigDecimal class instead. Numbers and Strings cover BigDecimal and other useful classes provided by the Java platform.
double	A double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section 4.2.3 of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
boolean	Has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information.
char	A single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

In this course we've been using int and boolean a lot.

The other kinds of types we're using are Object types like Color and Image, when you declare a variable to be of those types it really means that you're either going to create an instance of that type with the variable name, or you're going to set the variable to an object of that type.

When we say **Color noseColor = Color.RED**, we are setting noseColor to be a Color object. We could also say **Color noseColor = new Color(255,0,0)**; ((255,0,0) is the RGB value of red). In fact Color.RED is defined in the color class exactly the same way using *new Color(255,0,0)*. We'll cover this more later. For now, I just want you to know what you are typing when you type in the variable declarations.

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.

Classes and Instances

Object Design

Okay kids, it's time to get serious about making things more Object Oriented. So far we've played around with an Applet that executes actions and has characteristics, but everything we've done has been within a single Class. In this lesson we'll put Duke into a class of his own and let the Applet do what it's good at--grabbing stuff and putting it on the screen. And we'll let the Duke Class do what it should be good at, which is creating instances of Dukes that execute different actions and characteristics.

Who gets what?

Since you know how to start a new project now, let's create one. Call it **java1_Lesson07** and then give Duke a nice clean Class of his own that we'll call **Dukes** (plural). This time though, the class won't be extending Applet. Instead we'll create a different Class for that. It will use the Class we're making now. You'll see how this all works in a minute.

To do

1. Start a new project called **java1_Lesson07**.
2. Make a new class named **Dukes** (plural!).
3. The dialog should have **Source folder:** java1_Lesson07 (or java1_Lesson07/src).
4. Give it **Name:** **Dukes**.
5. This time **DON'T change the superclass: java.lang.Object**.
6. Click **Finish**.

Dukes will **not** be an Applet, which is why its Superclass should remain **java.lang.Object**.

Type in the blue code below:

CODE TO TYPE

```

public class Dukes {

    private Color noseColor = Color.red; // default Dukes have red noses
    private String action = "../images/duke/dukeWave.gif"; //default Duke
s are friendly
    private String whatDoing = "Give me something to do";
    private String message= "";

    public Dukes()
    {
        int rint = (int)(Math.random() * 3); // randomly generates a 0, 1,
or 2
        if (rint == 0)
        {
            noseColor = Color.blue; // more often red by default
            action = "../images/duke/dukeWave2.gif";
            message = "What's up with the blue nose!";
        }
    }

    public String getAction()
    {
        return whatDoing;
    }

    public String getActionImage()
    {
        return action;
    }

    public Color getNoseColor()
    {
        return noseColor;
    }

    public String getMessage()
    {
        return message;
    }

    public String write()
    {
        whatDoing = "I am a writing Duke";
        if (noseColor == Color.red)
        {
            action = "../images/duke/penduke.gif";
            message = "";
        }
        else
        {
            action = "../images/duke/penduke2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String think()
    {
        whatDoing = "I am a thinking Duke";
        if (noseColor == Color.red)
        {
            action = "../images/duke/thinking.gif";
            message = "";
        }
        else

```

```
        {
            action = "../../images/duke/thinking2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String wave()
    {
        whatDoing = "I am a waving Duke";
        if (noseColor == Color.red)
        {
            action = "../../images/duke/dukeWave.gif";
            message = "";
        }
        else
        {
            action = "../../images/duke/dukeWave2.gif";
            message = "My nose feels funny";
        }
        return action;
    }
}
```

This is just a Java class and won't actually run, but you should **SAVE** it anyway. Only applications and Applets will run. Go ahead and try to run it, just so you can observe the error.

Now let's break the code down bit by bit to see what this class is all about.

The first part of the definition of the Dukes Class is located on the first line. Immediately after that we have some **Attributes** (variables) defined. Ignore the *italicized text* for now; we'll discuss that shortly. For now, pay attention to the **colored text**:

OBSERVE: The colored text below

```
public class Dukes {  
  
    private Color noseColor = Color.red; // default Duke's have red nose  
s  
    private String action = "../..images/duke/dukeWave2.gif"; //default  
dukes are friendly  
    private String whatDoing = "Give me something to do";  
    private String message = "";  
  
    public Dukes()  
    {  
        int rint = (int)(Math.random() * 3); // randomly generates a 0,  
1, or 2  
        if (rint == 0)  
        {  
            noseColor = Color.blue; // more often red by default  
            action = "../..images/duke/dukeWave2.gif";  
            message = "What's up with the blue nose!";  
        }  
    }  
  
    public String getAction()  
    {  
        return whatDoing;  
    }  
  
    public String getActionImage()  
    {  
        return action;  
    }  
  
    public Color getNoseColor()  
    {  
        return noseColor;  
    }  
  
    public String getMessage()  
    {  
        return message;  
    }  
  
    public String write()  
    {  
        whatDoing = "I am a writing Duke";  
        if (noseColor == Color.red)  
        {  
            action = "../..images/duke/penduke.gif";  
            message = "";  
        }  
        else  
        {  
            action = "../..images/duke/penduke2.gif";  
            message = "My nose feels funny";  
        }  
        return action;  
    }  
  
    public String think()  
    {  
        whatDoing = "I am a thinking Duke";  
        if (noseColor == Color.red)  
        {  
            action = "../..images/duke/thinking.gif";  
            message = "";  
        }  
        else
```

```

        {
            action = "../images/duke/thinking2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String wave()
    {
        whatDoing = "I am a waving Duke";
        if (noseColor == Color.red)
        {
            action = "../images/duke/dukeWave.gif";
            message = "";
        }
        else
        {
            action = "../images/duke/dukeWave2.gif";
            message = "My nose feels funny";
        }
        return action;
    }
}

```

If we had inherited from something other than the default of `java.lang.Object`, we'd see the **extends** keyword identifying the **superclass** on the first line, just like before when we extended the `Applet` class. You don't need to extend `Object` because Java assumes every `Class` extends it. As soon as the `Class` is named in the definition, we begin defining its attributes (**noseColor**, **whatDoing**, **action**, and **message**).

Notice that not only did we define the **type** these attributes take, but we made them **private**. So what's this **public** and **private** stuff all about, you ask? Well, when we make `Classes`, we usually make them for other `Classes` to access and use. That's why we set permissions on the *members* of a `Class` that other `Classes` can access. **Permissions** can be **public**, **private**, or **protected**. (We'll discuss other important aspects of these modifiers in a later lesson.) For now we'll just study two permissions: **public** and **private**.

The Instance Variables (attributes) listed above are all **private** and the class definition and **access methods** are all **public**. The **reserved words** of **public** and **private** are modifiers present in order to indicate access capabilities for Fields and Methods. Since `Classes` are made for other `Classes` to use, we use **public** and **private** to determine the capabilities they can access. **Public** Methods usually only serve to get or change the values of **private** members, and are referred to as **getters** (gets) and **setters** (puts/changes). They are sometimes referred to as **accessors** and **mutators** as well. The idea behind all of this is for our code to be **encapsulated** and our **data/information to be hidden**. That way we only allow the values of the variables to be changed or accessed **through these accessor and mutator methods**. Thus the variables themselves are private, but access to them *may* be public (of course, these members are accessible inside of their own classes). Limiting access this way helps to prevent data corruption.

For example, in this `Class` we've got some **accessors** called **getAction()**, **getActionImage()**, **getNoseColor()**, and **getMessage()**. These methods were made specifically to allow programmers to **get** the values of the variables **whatDoing**, **action**, **noseColor**, and **message**, respectively, without accessing them directly. So now we have two sets of methods, one made up of accessors { **getAction()**, **getActionImage()**, **getNoseColor()**, and **getMessage()** } and the other made up of actions that `Dukes` can *perform* { **write()**, **think()**, and **wave()** }.

In the `Dukes` class, we allow each of the `Dukes` to have characteristics. The `Class` defines attributes of **noseColor** (of type **Color**), an **action** this particular `Duke` can take (of type **String**), and the **message** this particular `Duke` can give us (of type **String**).

Accessor Methods in Dukes

```
public String getAction()
{
    return whatDoing;
}

public String getActionImage()
{
    return action;
}

public Color getNoseColor()
{
    return noseColor;
}

public String getMessage()
{
    return message;
}
```

The remaining methods are the now familiar **actions** of a Dukes object, that is, they are the things that Dukes can do: `write()`, `think()`, `wave()`.

Action Methods in Dukes

```
public String write()
{
    whatDoing = "I am a writing Duke";
    if (noseColor == Color.red)
    {
        action = "../images/duke/penduke.gif";
        message = "";
    }
    else
    {
        action = "../images/duke/penduke2.gif";
        message = "My nose feels funny";
    }
    return action;
}

public String think()
{
    whatDoing = "I am a thinking Duke";
    if (noseColor == Color.red)
    {
        action = "../images/duke/thinking.gif";
        message = "";
    }
    else
    {
        action = "../images/duke/thinking2.gif";
        message = "My nose feels funny";
    }
    return action;
}

public String wave()
{
    whatDoing = "I am a waving Duke";
    if (noseColor == Color.red)
    {
        action = "../images/duke/dukeWave.gif";
        message = "";
    }
    else
    {
        action = "../images/duke/dukeWave2.gif";
        message = "My nose feels funny";
    }
    return action;
}

}
```

Finally, there's one special method type called the **Constructor**:

Dukes Constructor Dukes()

```
public Dukes()
{
    int rint = (int)(Math.random() * 3); // randomly generates a 0,
    1, or 2
    if (rint == 0)
    {
        noseColor = Color.blue; // more often red by default
        action = "../images/duke/dukeWave2.gif";
        message = "What's up with the blue nose!";
    }
}
```

The *italicized code* from above defines this Class's **Constructor**. A constructor is the method that's called to create an instance of a Class. If you don't provide a constructor definition, a default constructor without parameters is created automatically (we'll get to that later). **Constructors always have the same name as that of the Class**. That's why there is a method called **Dukes()** in this code. When we write another Class that instantiates this Dukes Class, we'll have to call this method which then *constructs* Dukes instances for us (we'll give this a try later in the lesson).

In this Dukes Class, the Constructor determines the **noseColor** of this Duke's nose, the **action** this Duke takes, and the initial **message** to send from this Duke. The code in this particular constructor picks a random number from among 0, 1, and 2. Only if it selects 0 does it give Duke a blue nose, otherwise his nose stays red.

It's important to be able to discern the various parts of the definitions of classes, although in essence, a class only has **methods** and **variables**. In the API they use the term **Fields** instead of "Variables", so you might see a **Field Summary**. Don't worry, it's just a variable.

In addition, a lot of code organization is done using **blocks**. Java uses **blocks** of code to specify definitions for classes, methods, and other groupings of code. Brackets **{ }** are used to begin and end blocks. Blocks are also called **compound statements** because they can be used to define or group together more than one statement.

To do

1. Let's match some blocks of code.

Eclipse can help us identify blocks of code. Click directly **after** an opening bracket **{** anywhere in the Dukes.java class. Notice that there's a little rectangle around its closing bracket **}**. Eclipse will identify the matching opening bracket when you click after the closing bracket.

Remember, the **{ }** blocks help us identify the methods of a class. The fields/variables will **not** have blocks of their own and should always be either at the very beginning of the class definition block, after the opening **{**, or at the very end of the class definition block, before the closing **}**. This will make your program more readable.

Initialization and Constructors

Defaults

Classes are the templates for Objects. The Class definition sets up the way Objects will look when they're instantiated. Defaults can be used to give instances initial standard values. In our definition of Dukes, Dukes will "normally" have red noses, and then those Dukes will have default actions.

Default values

```
private Color noseColor = Color.red; // default Duke's have red noses
private String action = "../images/duke/dukeWave2.gif"; //default dukes a
re friendly
private String whatDoing = "Give me something to do";
private String message = ""; //initial message is blank
```

In addition to the above default values for the attributes, a Class's constructor is used to make a specific instance of the Class and is usually used to reset some of the Attributes for that instance. In our case, the Dukes constructor is used to decide the color of this particular Duke's nose. Normally Dukes will have a **noseColor** by default, but our Constructor has code in it that can change that upon instantiation depending on a random variable (**rint**). Similarly, we set the Dukes default **action** to wave. However, if you look in the Dukes constructor, we are generating random numbers of 0, 1, and 2 that will ultimately determine which characteristics an instantiated Duke will have. If the random number happens to be 0, we are giving that instance of Dukes a blue nose. Since the blue-nosed, waving Duke must have a blue nose, we are also changing the default waving red-nosed Duke (dukeWave.gif) to a waving blue-nosed Duke (image of dukeWave.gif).

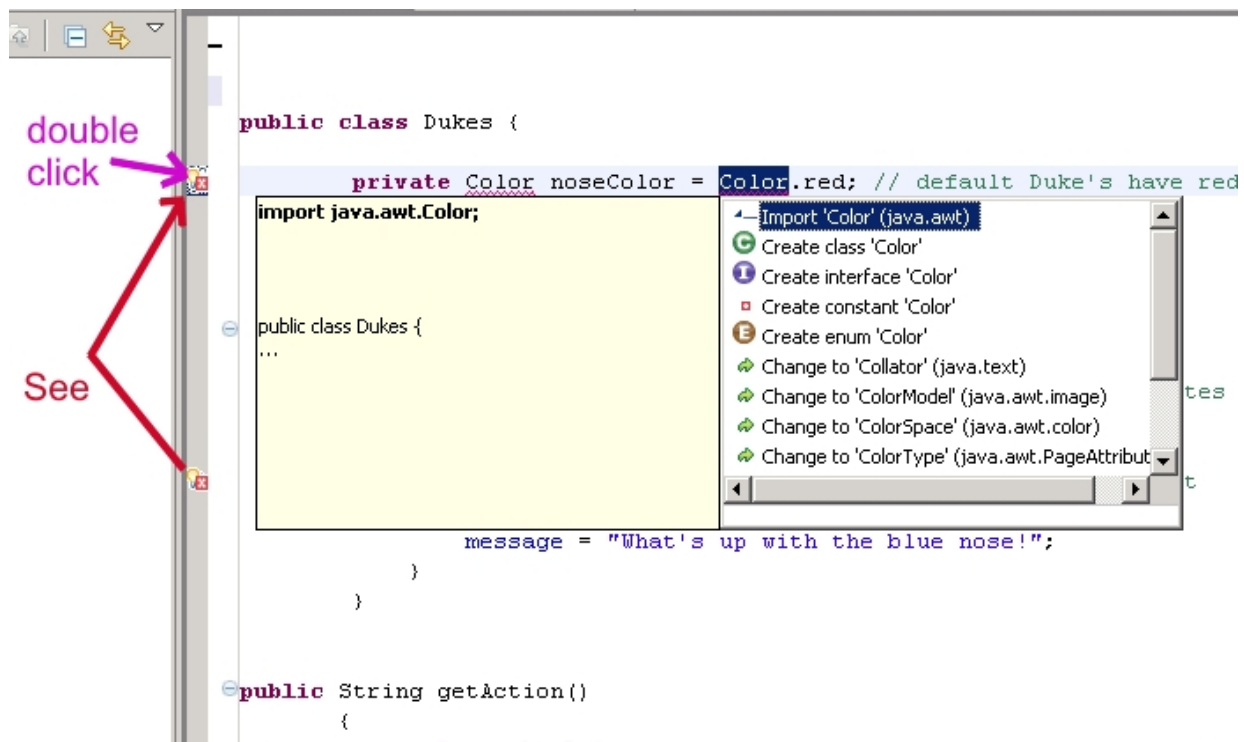
Constructors have specific characteristics that differ from other methods. Remember, they always have the same name (case-sensitive) as the class, and the programmer never specifies that they "return" anything, since the constructor always returns exactly one thing--an instance of this type of class.

We're going to get Duke's fingers moving in the next sections by building some Applets that use the Dukes class we created in this section. This might be a good time to take a break or maybe even take a nap.



Making an Applet for Dukes

If you haven't got it opened already, open up the **Dukes.java** file we worked on in the previous section. You may notice that Eclipse is giving us some errors:



Double-click on those errors for suggestions for fixing them. Always try the first suggested fix first because it might also fix some of the errors farther on down the chain. Go ahead and try the first suggestion. It worked, didn't it? Did all of your errors go away? Cool, huh? At the top of your Dukes.java file you should see this:

At the top of your Dukes.java file:

```
import java.awt.Color;
```

That's because we're using the colors from the Color object.

Since the class `Dukes` is not an Applet (remember--he is a `java.lang.Object`), we need to make an Applet that will use instances of `Dukes` and display them.

Start a **new class** called **DukesApplet** and **make sure** it extends Applet this time. Its superclass should be **java.applet.Applet**.

CODE TO TYPE: DukesApplet

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

public class DukesApplet extends Applet{

    Dukes myDuke;

    public void init()
    {
        myDuke = new Dukes();
    }

    public void paint(Graphics g)
    {
        String action="";
        switch ((int)(Math.random() * 3))
        {
            case 0: action= myDuke.write(); break;
            case 1: action= myDuke.think(); break;
            case 2: action= myDuke.wave(); break;

        }

        Image myAction = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10,130);
        g.drawString(myDuke.getMessage(), 10,145);
        g.drawImage(myAction, 10, 10, Color.white, this);
    }
}
```



Run it.

The output is the same as before, but the code is much different this time. Let's look at what this code is doing differently.

Notice that this Class **extends Applet**. As we mentioned, the Java people already wrote the Class **Applet** and the Class **Graphics** used by Applets. Now let's take a look at this Applet and see how we've incorporated the Dukes class we made earlier.

DukesApplet

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

public class DukesApplet extends Applet{

    Dukes myDuke;

    public void init()
    {
        myDuke = new Dukes();
    }

    public void paint(Graphics g)
    {
        String action="";
        switch ((int) (Math.random() * 3))
        {
            case 0: action= myDuke.write(); break;
            case 1: action= myDuke.think(); break;
            case 2: action= myDuke.wave(); break;
        }

        Image myAction = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10,130);
        g.drawString(myDuke.getMessage(), 10,145);
        g.drawImage(myAction, 10, 10, Color.white, this);
    }
}
```

Here we instantiated and used an object from a Class we created. In doing so, we created an **Instance Variable** that has the type of the object name. In this case we wrote: **Dukes myDuke**, thus the instance variable myDuke is of type Dukes. Now we need to give **myDuke** a value. That's where the **Constructor Dukes()** from the Dukes class comes in. When we write **myDuke = new Dukes()**, Java uses the constructor to create an instance of Dukes, which in turn sets any attributes we put into the Dukes() constructor. Now that we have an instance of Dukes on our hands (myDukes), we can use the methods of Dukes however we wish. To access methods of Dukes, we simply use the syntax **myDukes.methodName()**.

Now instead of having to write the think(), wave(), and write() methods, we can simply reuse the methods that are in the Dukes() Class. We also used the getAction() and getMessage() methods to access the message and action that were set in the Dukes class. Also, notice that the **action** variable in this Class is not the same as the action variable in the Dukes Class.

This code is more modular and encapsulated than our first Applet. This new Applet specifies things to perform using only Applet's native methods, such as **init()** and **paint(Graphics g)**. It's a better design.

Notice the **init()** method. It's in this method that we **create** or **instantiate** our **instance** of the Dukes class. Classes that are not Applets are instantiated with the **new** command.

Like all variables, you can declare and instantiate (give value) in one line:

```
Dukes myDuke = new Dukes();
```

In general, when you create a new object by invoking a constructor, you can do it in one or two lines.

OBSERVE: Declare and Instantiate

```
Dukes myDuke;
myDuke = new Dukes();

or

Dukes myDuke = new Dukes();
```

In the first case, **Dukes myDuke** is only **declaring** that myDuke is going to be of **type** Dukes. Remember when Eclipse said, "cannot resolve the type" of something? Everything in Java must be **declared** as a **type**

so Java knows where to look for it. The `myDuke = new Dukes()` creates an instance with the `new` command and the Classes constructor `Dukes()`.

In the second case above, `Dukes myDuke = new Dukes()` both *declares* and *creates* the instance in one line.

In the next section we'll see the difference between the two cases.

Another Applet for Dukes

We've been touting the power of object-oriented design for a while now, so let's see how to take advantage of its capabilities by reusing the Dukes class in another Applet. In this Applet we'll create **two** instances of Dukes. Make a new Class called `TwoDukesApplet`, and **remember to make the superclass `java.applet.Applet`**. *Be sure to create this file in the Lesson07/src folder as well.*

CODE TO TYPE: TwoDukesApplet

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

public class TwoDukesApplet extends Applet {

    Dukes myDuke, yourDuke;    // two declarations for Duke instances
    String myAction, yourAction;    // each will have their own action

    public void init() {

        myDuke =new Dukes();           // instantiate first Duke
        myAction = myDuke.getActionImage(); // his first action

        yourDuke =new Dukes();         // instantiate second Duke
        yourAction = yourDuke.think();  // his first action is to think

        resize(400,200); //resize the applet window so that we can see both duke
s
    }

    public void paint(Graphics g) {

        Image myChoice = getImage(getDocumentBase(), myAction); // get and show image
for first Duke
        g.drawString(myDuke.getAction(), 10,165);
        g.drawString(myDuke.getMessage(), 10,180);
        g.drawImage(myChoice, 20, 50, Color.white, this);

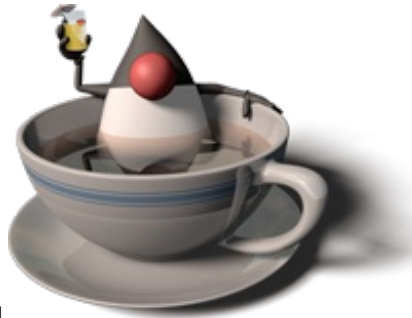
        Image yourChoice = getImage(getDocumentBase(), yourAction); // get and show
image for second Duke
        g.drawString(yourDuke.getAction(), 200,165);
        g.drawString(yourDuke.getMessage(), 200,180);
        g.drawImage(yourChoice, 200, 50, Color.white, this);
    }
}
```



Save and Run it.

Now you should see two instances of Duke on this Applet. Notice that we used `resize(400,200)` to make the Applet big enough so we could see both Dukes.

As the course progresses, **you** will implement and change Classes in many different ways. For now, the goal is to understand the basics: Classes define variables and methods which help to define the Class itself and specify its capabilities. In the next lesson we'll write our own Classes again, and we'll also **use** some of the classes that Java has written for us to make coding easier.



But for right now, we're exactly where we want to be!

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.*

Using the API: Introductory Graphics

Using Java Provided Classes

In this lesson we'll focus on using the **Graphics** class and its capabilities. This will serve two purposes: first we'll learn how to do some drawing, and second we'll learn more about using pre-defined classes from the Java API.

java.awt.Graphics Class

In most of the Classes we've created so far, we inherited from **Applet**. To do this, we imported using **import java.applet.Applet**; so the Java compiler could find the class and retrieve its **inherited** properties and methods. Most object-oriented programming is not about writing code from scratch, but using code that others have already written. Java provides a large library of code, as well as documentation to help us figure out what's available in that library and how to use it. The documentation is located in the [API](#). (And there's a handy **API** link on the menu too.)

You might recall that we used the **Graphics** Class in many of the Classes we created in earlier lessons. In an Applet, the Graphics area is **passed** to the **paint(Graphics g)** method when it gets instantiated. The **Graphics** area is where you can "draw" or "print" things. For instance, in our HelloWorld Applet, we printed some text. The **Graphics** class allows us to do stuff like that. In this lesson, we'll work with the **Graphics** and **Applet** Classes to see some of the capabilities they offer.

Alright, you know the drill. **Start a new project called java1_Lesson08**. Just in case, I'll remind you how to do it one more time:

To do

1. Go to the File menu: **File | New | Java Project**.
2. Give it the name **java1_Lesson08**.
3. Click **Finish**.
(If it asks whether you'd like to "Open Associated Perspective", say "No"--we want to keep our own perspective environment.)

Now start a **new Class called FirstLine** and make the superclass **java.applet.Applet**:

CODE TO TYPE: Simple Applet Using Graphics

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

public class FirstLine extends Applet {

    public void paint(Graphics g)
    {
        g.drawLine(20,10,40,40);
    }
}
```



Run this Applet. You should see a line. I guess that's kind of cool, huh? Now play around and change the numbers in the drawline() method to see if you can get the line to move up from left to right.

If there were instance variables or class variables present, we'd put them either at the beginning of the class, **before** any method definitions, or at the end of the class **after** all of the method definitions.

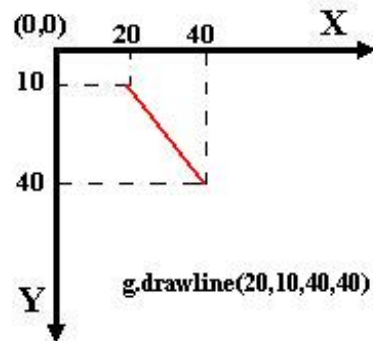
In this class there is only one method defined. Method definitions are easy to find because they will have parentheses for their parameters and then curly brackets to indicate their block of code.

One of the methods defined in the FirstLine class is the **paint(Graphics g)** method.

The **Graphics g** in the argument of the paint method is called the method's **formal parameter**. It indicates that this method must be given or **passed** an instance of a **Graphics** object. The **Graphics** instance that is

passed to the method is called the **actual parameter** and has been given the variable name of **g**. Since **g** now represents the **Graphics object**, we can execute "Graphics" actions on it.

As you can see from this little Applet we made, **Graphics** objects can draw lines. Manipulating pixels enables us to determine the location and color of our graphics. To control location, we consider horizontal and vertical or (x,y) coordinates, with (0,0) being defined as the top left corner.



In this example, we drew a line from pixel location (20,10) to pixel location (40,40). So in the `g.drawLine(20,10,40,40)` method, the first two numbers are the coordinates of the start of the line and the second two numbers are the end of the line.

We see **two** methods in the `FirstLine` class:

1. `paint(Graphics g)` is a **method we are defining** for our new **subclass** `FirstLine` of the **class** `Applet`. The method `paint()` has one formal parameter of **type** `Graphics`.

2. `drawLine(20,10,40,40)` is a **pre-defined method** (although we can enter any parameters we wish) of the **class** `Graphics`, of which `g` is an **instance**. We are **invoking (using or calling)** this method. To do that, we need to look at the **API** in the `Graphics` class to see how it was defined so that we know its formal parameters and then when we use it, we can pass it the right things.

Using the API

As we've mentioned, the API contains the existing classes Java provides for programmers.

To do

1. Go to the API opening page [here](#) and locate the **Packages** header.

Note

Even though we often provide these links for you during lessons, try to get used to going to the **API** button on the menu bar instead.

Packages are groupings of Classes that are related in some fashion. These packages are put into different **directories** or **folders** for us to access, but in order for Java to find them when we're running our programs, they have grouped them into what they call **packages**.

The packages that we'll find particularly useful now are:

- `java.applet`
- `java.awt`
- `java.awt.event`
- `java.lang`

But there are many more. In this lesson, we'll mainly focus on one class that lives in the `java.awt` package. (By the way, **awt** stands for **abstract window toolkit**.)

In the API page that opened:

To do

1. Click on the `java.awt` link.
2. Scroll down until you see the header **Class Summary**.
3. Scroll down to locate the **Graphics** Class and then click on it.

You should get a page that looks like [this](#), although you may see more frames for other packages and classes on the left. Let's take a closer look at some important aspects shown in the API. Keep one of the browsers with the Graphics class open so we can compare.

```
java.awt
Class Graphics

java.lang.Object
└─ java.awt.Graphics

Direct Known Subclasses:
    DebugGraphics, Graphics2D
```

```
public abstract class Graphics
extends Object
```

In the top left corner, the API tells us that the Graphics class is in the `java.awt` package. Notice Classes are named with the package name first, and that package names are separated with periods and start with lower case letters. Also note that Classes always start with capital letters. That makes it easy to differentiate between the package and the class.

The API also displays an **inheritance tree**:

```
java.lang.Object
└─ java.awt.Graphics
```

This indicates that Graphics **is an** Object.

It also indicates which **package** each of the classes is in (Object is in `java.lang` and Graphics is in `java.awt`). Remember from Lesson 2 that **every object** in Java inherits from the class `java.lang.Object`. That's one reason you never need to import the package `java.lang`. Because it's **always** needed, Java imports it by default.

Next the API tells us all of the Classes that are provided in Java that have the Graphics class as a **parent** (a superclass).

```
Direct Known Subclasses:
    DebugGraphics, Graphics2D
```

In the API page for the Graphics class:

To do

1. Click on the **DebugGraphics** Subclass link.
2. Note its inheritance chain. Its Superclass is java.awt.Graphics.
3. Go back to the Graphics page.
4. Click on the **Graphics2D** Subclass link.
5. Note its inheritance chain.

To reiterate and make absolutely certain this will be emblazoned into your memory banks for all eternity, inheritance is an important part of object-oriented programming because it allows us to use Classes that have already been written.

Finally, let's look at the `public abstract class Graphics extends Object`. This is the first line of code that defines the class Graphics. It starts the definition of the Class and tells us where this class belongs in an inheritance tree. It tells us the most **specific** class from which it inherits. You can see all of the other Classes that it inherits from the description of this Class in the API.

Note

If you inherit from a specific Class, you inherit all of the **ancestors** (the entire inheritance chain) as well.

Let's take a look at other stuff the API offers us for the Graphics class.

To do

1. Scroll down the API page until you see a Header that says **Constructor Summary**.

It should look like this:

Constructor Summary	
protected	Graphics() Constructs a new Graphics object.
Method Summary	
abstract void	clearRect (int x, int y, int width, int height) Clears the specified rectangle by filling it with the background color of the current drawing surface.
abstract void	clipRect (int x, int y, int width, int height) Intersects the current clip with the specified rectangle.

Constructors are methods as well, but they are special because we use them to create an instance of the Class (instantiation). And of course, the Constructor has the same name as the Class.

In the **Method Summary** you see **many** methods that are defined in the Graphics class. Except for the Constructor method, all methods begin with lower-case letters.

To do

1. Scroll down through the methods of the Graphics class to get a feel for what's available.

The most commonly used Graphics methods are **draw** and **fill**. You can find out more about each method and its usage by clicking on them.

To do

1. Go to the drawLine method--drawLine(int x1, int y1, int x2, int y2)--in the API Graphics listing and click on it.

abstract void	<u>drawLine</u> (int x1, int y1, int x2, int y2) Draws a line, using the current color, between the points (x1, y1) and (x2, y2) in this graphics context's coordinate system.
abstract void	<u>drawOval</u> (int x, int y, int width, int height) Draws the outline of an oval.
abstract void	<u>drawPolygon</u> (int[] xPoints, int[] yPoints, int nPoints) Draws a closed polygon defined by arrays of x and y coordinates.
void	<u>drawPolygon</u> (Polygon p)

Methods, Parameters (or Arguments), and the Dot Operator

Now let's look at the specifications of the drawLine method in our class FirstLine. In defining the method **paint()** for Applets, Java specifies that its method has a *parameter* that is **passed**. This parameter is a **Graphics** object (you can see for yourself in the API). In our FirstLine class code for the paint method, we name this Graphics object **g**. Since we are *defining* the method, we could name the variable that holds the object whatever we like. But once you name it, you'll have to use the same name throughout your method definition. This isn't hard to do here though, because we only have one line so far.

Because the **paint()** method is present for all **Applet**, the **Graphics** area is always created and given to us through the Applet. This is another example of inheritance. We have inherited all of the traits of Applets and the Applet has provided this Graphics area by passing it to the paint method every time our Applet is displayed and repainted.

We **invoked** or **called** the Graphics drawLine() method by typing **g.drawLine(20,10,40,40);**. This line uses the **dot operator** of Java. In front of a dot is the **object** and after the dot is either one of that object's **variables/fields** or a call to one of that object's **methods**. If you see parentheses, it is calling a method. If you do not see parentheses, it is calling a variable. In this example, we are telling Java to go to the **object** we named **g** and to use one of its methods called **drawLine()**. Inside the parentheses are the "actual parameters" we want Java to use.

To make sure we used this method correctly, we compare our method call and its **actual parameters** to the **drawLine()** method specification and the **formal parameters** in the API.

On the API page, for the method drawLine() we see:

The screenshot shows the Java API documentation for the **drawLine** method. The browser address bar shows the URL: [http://java.sun.com/javase/6/docs/api/java/awt/Graphics.html#drawLine\(int,%20int,%20int,%20int\)](http://java.sun.com/javase/6/docs/api/java/awt/Graphics.html#drawLine(int,%20int,%20int,%20int)). The left sidebar shows the "Platform" section with "Ed. 6" selected. The main content area displays the method signature: **public abstract void drawLine**(int x1, int y1, int x2, int y2). Below the signature is the description: "Draws a line, using the current color, between the points (x1, y1) and (x2, y2) in this graphics context's coordinate system." Under the "Parameters:" section, the following details are listed: x1 - the first point's x coordinate, y1 - the first point's y coordinate, x2 - the second point's x coordinate, and y2 - the second point's y coordinate.

This definition lets us know that if we have a Graphics object, we can use the method drawLine() if we pass it four integers (int). The method drawLine() will then assign the four integers as stated. The first two values will be (x,y) for the first point in the line, and the second two values will be the (x,y) for the second point. Here are a couple of alternate illustrations of parameters:

- **public abstract void drawLine(int x1,int y1,int x2,int y2)** The method definition line (sometimes called the method's **signature**) provides **formal parameters**.
- **g.drawLine(20,10,40,40);** Here we see the use of specific numbers to send to the method for it to use, hence it provides what are called **actual parameters**.

We use the API to find method definitions and make sure we send the proper type and number of parameters.

Sequencing

Let's use some of the methods in the Graphics class to draw something in an Applet.

We'll be using a programming *construct* called **sequencing** as well. Sequencing means that if you give Java a list of things to do, it will do them in order, one after another. After we get Java to draw our first line, we'll have it do even more work for us.

Start another class called MyPicture that extends Applet:

CODE TO TYPE:

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

public class MyPicture extends Applet {

    public void init()
    {
        this.setBackground(Color.lightGray);
    }

    public void paint(Graphics g)
    {
        g.drawLine(0,0,100,100);
        g.setColor(Color.RED); // make a red ball
        g.fillOval(45, 15, 40, 40);

        g.setColor(Color.GREEN); // a couple of support beams
        g.fillRect(5, 5, 4, 95);
        g.fillRect(65, 65, 4, 35);

        g.setColor(Color.BLACK); // a landing strip
        g.drawLine(100,100,200,100);
    }
}
```



Run it.

There's lots for us to check out, even in this simple program. First, an Applet always starts by having its **init** method called and then its **paint** method. Once inside the paint method, we see the use of **line sequencing**, particularly with use of the Graphics object **g**. Notice the change of color in the code and Applet, and also that the colors are specified by typing Color.lightGray, Color.Red, and so on.

To do

1. Go to the Graphics API and read the specifications of the methods that start with:
 - draw (e.g. **drawOval**, **drawPolygon**, **drawString**)
 - fill (e.g. **fillRect**, **fillOval**, **fillArc**)
 - get (e.g. **getColor**)

The java.awt.Color Class

Shapes in a Graphics object can be filled in once a color is specified. By default the color of the graphics "pen" is black. The class [java.awt.Color](#) provides a list of color possibilities.

Here are some of the Color classes used in the MyPicture.java class:

- this.setBackground(Color.lightGray);
- g.setColor(Color.red);

Look at the MyPicture.java class to check out these "method calls" in detail:

- **this.setBackground(Color.lightGray);** Notice the word **this** within the code. Because that line is within the `init()` method of a class that is an Applet, the **this** means that you are telling **this Applet** to set its background. Thus we would look for the method in the Applet class.
- **g.setColor(Color.red);** These method calls are all in the paint method of the Applet. Note that *before* the dot operator there is a **g**. **g** is a Graphics object/instance, so we are telling Java that we want to use the **Graphics** method of **setColor()** on our **g** object.

To find more about **setBackground(Color.lightGray)**, go to the [Applet API](#) and look for the method **setBackground(Color c)**. Applet inherits background color from **Component**. Look in the **method inheritance section** -- about a third of the way down the page you'll see:

Methods inherited from class java.awt.[Component](#)

To find out more about **setColor(Color.red)**, go to the [Graphics API](#) and look at its methods. Click on **setColor(Color c)** for more detail.

Let's discuss the keyword **this** a bit more. When we write the definition of a Class, we don't know the name the user will give to the instance when they use our class. Thousands of programmers might end up using the class, each one giving a different name for each instance. That's why we use the **reserved word** **this** within the code of a class definition: to indicate that we are telling *this object* (the object that we are currently using) to invoke the method given. In the above example, we are telling the instance of the MyPicture Applet to have a background color of lightGray with the line: `this.setBackground(Color.lightGray);`.

Similarly, when we have an instance of an object, sometimes we'll want to tell its **parent** to do something. Since you won't always know the parent of every instance inside the definition of the class code, Java uses the **reserved word** **super**. So, **this** means "this one," "me," while **super** means "this instance class's parent."

Okay, back to our discussion of Color. Click on the [Color API](#). Scroll down the Color API page to see the **Field Summary**. Now we'll get to see some **Class Variables**.

Note In the API, Java uses the term **Fields** for the Instance and Class **Variables**.

To do

1. Scroll down to the **Field Summary** for **Color**.
2. Look at the left column.

This is a list of the Variables that the class **Color** provides for us. You can see the two words **static** **Color** throughout the list.

Note

The **reserved word** of **static** is a **modifier** for the Field and indicates that the Variable (or Method) is a **Class Variable** (or a **Class Method** when applied to methods). Static variables will be covered in a later lesson.

Recall that Classes have two components: **Variables** and **Methods**. However, they each come in two forms: **Instance** and **Class**. So these are the four combinations available:

- Instance Variables
- Class Variables
- Instance Methods
- Class Methods

That's it! Once you've got this concept down, it's a whole lot easier to read code.

So what are the differences between Instance Variables and Class Variables? Here are some important ones:

- A **Class Variable** is shared by **all** of the instances in a Class. For example, all members of the

Class "Human" have **two** eyes. But the **color** of eyes changes from one human to the next. So numberOfEyes would be a Class Variable set to 2 in the Human class, but colorOfEyes would be an **Instance Variable** that changes with each instance.

- Because the Class Variables are shared by all in the Class, their values are not stored in separate places in the computer's memory. Values for Class Variables are in a single location in memory. This means that if the value changes for one instance, it changes for **all** instances of the Class.
- A Class Variable (or Method) can be **called** from **either** the Class, or an instance of the Class. For example, the value of the variable **BLACK** in the Color class is expected to be the same whenever it is called, thus it does not need to be "created" each time it is used. This makes us happy. The Color class will be used quite often, and it would be a pain, for example, to have to make a new instance of Color every time we wanted to get the color black.

Let's look at the way we specified Colors in our code: **Color.lightGray** or **Color.RED**.

Note that we have a **dot operator** here, and that after the dot there can be either a method or a variable. And we know it's not a method because methods **always** have parentheses.

Recall that convention dictates that Classes begin with capital letters. Look at how we accessed the colors: Color.RED

Here we're accessing Color's variable **RED** through its **class** name **Color**. In other words we didn't create an instance like this: Color mycolor = new Color(255,0,0); and call myColor.RED, instead we simply called Color.RED.

We are able to do this because the variable **RED** is a Class Variable.

We know that we can do this here because the **keyword static** is in the Class's specification of the variable (look it up in the API!). We'll cover static variables more in Lesson 14, but basically static variables are declared static because they aren't expected to change and we can access them through their Class name instead of their instance name (they aren't instance dependent *instance variables*.)

So what else was in the **Field Summary** for the class **Color**?

1. The variables tell us that they are **static** (which always means **Class** variables), and they also contain the word "Color" and a link to the Color class. This is because each of the Class Variables in the class **Color** is Colors:

- black is a Color
- BLACK is a Color
- blue is a Color
- BLUE is a Color

The description of the Fields/Variables is telling you the **type** of Object the Variable is.

2. Each of the Class variables in the class **Color** is either in all CAPITAL or all lower-case letters.

Look at some of the methods of the Class **Color**. You can make a color lighter or darker using these methods. For example:

```
Color.red.darker();
```

Here Java goes to the **Color** class, gets the Class Variable **red**, then invokes the method **darker()** on that **Color**.

But there is something tricky going on here. It turns out that Color.RED, is a Class variable AND an Object. That's because the definition of RED in the Class Color is **Color RED = new Color(255,0,0);**, so RED is an instance of the Color class. And so, we can call things like **Color.red.darker()**; since Color.red is a Color object itself. We could also simply call **Color noseColor = new Color(255,0,0);** and then call **noseColor.darker()**; to make it darker.

To do

1. **Open** the **MyPicture** class and **Run** it (don't close it).
2. **Edit** the **MyPicture** class's **paint** method as shown below in **blue**.

CODE TO EDIT

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

public class MyPicture extends Applet {

    public void init()
    {
        this.setBackground(Color.lightGray.darker());
    }

    public void paint(Graphics g)
    {
        g.drawLine(0,0,100,100);
        g.setColor(Color.RED.darker()); // make a red ball
        g.fillOval(45, 15, 40, 40);

        g.setColor(Color.GREEN); // a couple of support beams
        g.fillRect(5, 5, 4, 95);
        g.fillRect(65, 65, 4, 35);

        g.setColor(Color.BLACK); // a landing strip
        g.drawLine(100,100,200,100);
    }
}
```



Save and Run it. Compare the two Applets' colors.

Great work so far. We're really making progress. In the next lesson, we'll expand our artistic palette to include even more Java 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.

Drawing with Graphics

Making Pictures

Back to Graphics

In this section we'll use the **Graphics** Class to make drawings. First, let's make an Applet we can use to test our drawings.

1. Make a new project for Lesson 9: **File | New | Java Project**.
2. Name the Project **java1_Lesson09**.
3. Click **Finish**. (If "Open Associated Perspective" appears, click "No." We want to keep our own perspective environment.)
4. Make an Applet in this project called **DrawTest**.

The code for the Applet is fairly simple:

CODE TO TYPE: DrawTest

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

public class DrawTest extends Applet
{
    public void init()
    {
        setBackground(Color.cyan);
    }

    public void paint(Graphics g)
    {
        // empty for now until we have made code of images to draw
    }
}
```

Okay, now we need to add a Class that draws something. This first drawing will use only ovals and lines. Let's try to draw **Cartman** from the TV show South Park, using only the primitive shapes available in the Java API.



1. Look in the [Graphics](#) API for some methods that draw.
2. Go to the Graphics API listing of methods for the **fillOval()** method.
3. Click on the **fillOval()** link there.

You should see this:

fillOval

```
public abstract void fillOval(int x,  
                             int y,  
                             int width,  
                             int height)
```

Fills an oval bounded by the specified rectangle with the current color.

Parameters:

x - the *x* coordinate of the upper left corner of the oval to be filled.

y - the *y* coordinate of the upper left corner of the oval to be filled.

width - the width of the oval to be filled.

height - the height of the oval to be filled.

See Also:

[drawOval\(int, int, int, int\)](#)

Sometimes the API doesn't provide an *exact* method to help us execute a task, so we have to work around it, but in this case it does. for our purpose here, a circle **IS** considered a type of oval. A circle is in essence a symmetrical oval, an oval that has the same height and width. So in order to draw circles, we use the same methods that make ovals.

Alright, let's get to work! In the java1_Lesson09 project, **create a new Class called Cartman**.

CODE TO TYPE: Cartman

```
import java.awt.*;

public class Cartman {

    Graphics g;    // make the Graphics area an instance variable so the methods
can use it

    public Cartman(Graphics graph)    // the class Constructor
    {
        this.g = graph;    // give the graph instance to the Instance Variable we na
med g
    }

    public void drawMe()
    {

        g.setColor(Color.PINK);
        g.fillOval(10,30,180,150); //Cartman's face

        g.setColor(Color.white);
        g.fillOval(50,66,35,53); //Cartman's eyes
        g.fillOval(78,66,35,53);

        g.setColor(Color.black); //Cartman's eyeballs
        g.fillOval(63,86,10,10);
        g.fillOval(90,86,10,10);

        g.setColor(Color.black); //Cartman's mouth
        int [] xValues = {56,89,109};
        int [] yValues = {140,150,140};
        g.fillPolygon(xValues, yValues, 3);

    }    // end drawMe method

}    // end Cartman class
```

Save it.

Okay, now we need the Applet to see this class. Go back to your DrawTest.java file. Edit the paint method you find there by adding the code below:

CODE TO TYPE

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

public class DrawTest extends Applet
{
    public void init()
    {

        setBackground(Color.cyan);

    }

    public void paint(Graphics g)
    {
        Cartman myCartman = new Cartman(g);
        myCartman.drawMe();
    }
}
```



Run it. Not a bad likeness if you ask me. In the projects for this lesson, you'll make him look even more like Cartman.

Now let's take a look at the **fillPolygon()** method in **Graphics** in the API. Notice that this method is **overloaded** - look at its two signatures:

abstract void	fillPolygon (int[] xPoints, int[] yPoints, int nPoints) Fills a closed polygon defined by arrays of x and y coordinates.
void	fillPolygon (Polygon p) Fills the polygon defined by the specified Polygon object with the graphics context's current color.

Hold on a minute. There are **two** fillPolygon() methods? In the previous lesson we mentioned the concept of **polymorphism**--that sometimes the same name is used in different places. On occasion, a method might **override** its parent. Now in this Class we see an example of another important form of polymorphism called **overload**.

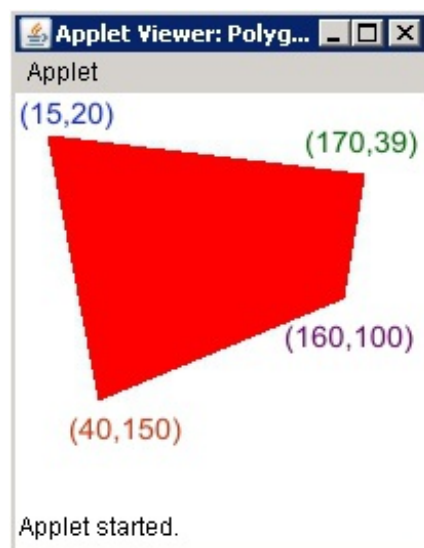
Overloading is when a specific class has more than one method with the same name. Hmm, this sounds like it could get tricky. But Java figures out which one to use by reading the method's **full signature**. The signature is a method's definition and is determined by its name and the number and types of parameters it takes. You can have methods with the same name, and even the same parameters, but they'll all have different signatures.

A **signature** of a method is much like the signature of a human; each signature is unique. As such, in Java, a signature can be used to identify a method, then decide which one to use. To accomplish this task, Java:

- Identifies the type of class calling the method (i.e., the object in front of the dot operator -- like g in g.drawLine).
- Observes the name of the method.
- Observes the **number** and the **type** of parameters (if there is more than one method with the same name).
- Ensures that the proper values for Java have been passed. In other words, the values passed (actual parameters) are consistent with those defined in the class (formal parameters).

So looking at the API again, the **fillPolygon()** method takes 3 parameters (**int[] xpoints, int[] ypoints, int npoints**) or 1 parameter (**Polygon p**). In the Cartman example, we're using the method that takes 3 parameters. If we had a Polygon object to pass to it, we could use the method with 1 parameter--in fact we'll try that a bit later, but for now let's go over what we've done here using the method that takes 3 parameters. You can tell that the first 2 parameters are sets of integers because they take an integer array int[] (Arrays are sets and are indicated by square brackets. We'll study arrays in great detail in the next Java course).

Let's consider an example. Suppose you wish to fill these 4 points in the polygon: **(15,20)**, **(170,39)**, **(160,100)** and **(40,150)**. The shape would be:



Go to the Graphics class and look at the spec for "drawPolygon(int[] xPoints, int[] yPoints, int nPoints)." See

how it indicates that the first parameter is an array of **x** coordinates?

The array of **x** coordinates would look like this: **[15,170,160,40]**

The array of **y** coordinates would look like this: **[20,39,100,150]**

The method call to a graphics object **g** would be:

OBSERVE:

```
int [] xValues = {15,170,160,40}; // declare the arrays
int [] yValues = {20,39,100,150};
g.fillPolygon(xValues, yValues, 4);
```

The number **4** in the argument of the **fillPolygon()** method is the number of points in the polygon we're drawing.

Now let's get back to our example. In our example we drew a triangle for Cartman's mouth. We passed the **fillPolygon()** method two integer arrays (3 xValues and 3 yValues) and passed the number of points (3). To show the polymorphism of the **fillPolygon()** method, let's create a Polygon object and pass that to **fillPolygon()**. Change the Cartman class as follows:

Change the code in blue in the Cartman Class

```
import java.awt.*;

public class Cartman {

    Graphics g; // make the Graphics area an instance variable so the methods can use it

    public Cartman(Graphics graph) // the class Constructor
    {
        this.g = graph; // give the graph instance to the Instance Variable we named g
    }

    public void drawMe()
    {
        g.setColor(Color.PINK);
        g.fillOval(10,30,180,150); //Cartman's face

        g.setColor(Color.white);
        g.fillOval(50,66,35,53); //Cartman's eyes
        g.fillOval(78,66,35,53);

        g.setColor(Color.black); //Cartman's eyeballs
        g.fillOval(63,86,10,10);
        g.fillOval(90,86,10,10);

        g.setColor(Color.black); //Cartman's mouth
        int [] xValues = {56,89,109};
        int [] yValues = {140,150,140};
        Polygon shapeThing = new Polygon(xValues, yValues, 3);
        g.fillPolygon(shapeThing);

    } // end drawMe method

} // end Cartman class
```



Save and Run the Applet again.

You should see exactly the same thing. The only difference in the code here is that we passed **fillPolygon()** a Polygon object. Not much of a leap, but nevertheless an example of polymorphism. We'll see many more examples of this in the near future.

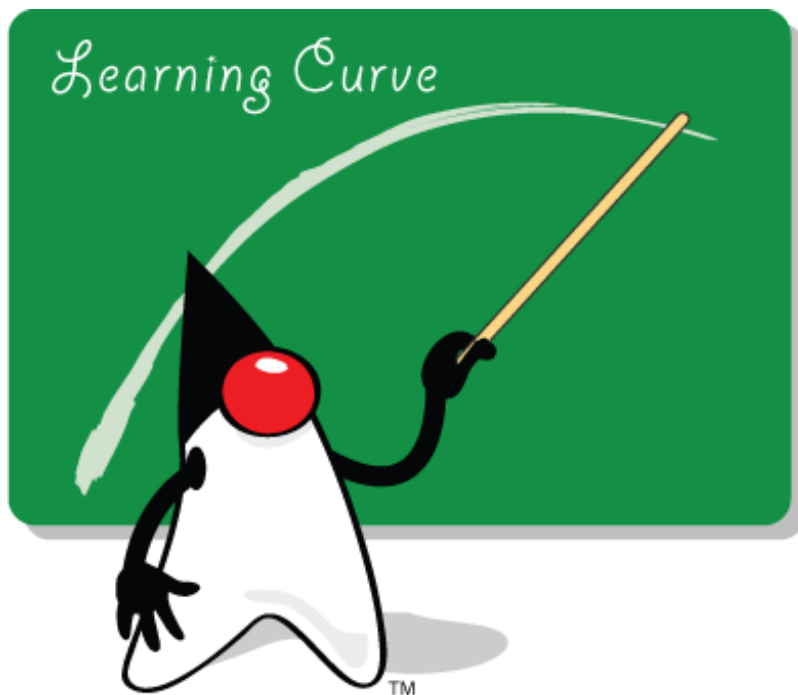
So to reiterate, as we are prone to do, the central ideas of object-oriented programming are:

- Inheritance
- Polymorphism
- Encapsulation

These characteristics allow us to use the API for large portions of our programming. As the course continues, we'll see more of these principles of object-oriented design in action.

Hopefully the API is fast becoming your new best friend. Officially **API** is an acronym for **Application Programming Interface**, but we can think of some better words! Let me try...

- **A**bsolute **P**roductivity **I**ncrease
- **A**rchived **P**rogramming **I**ntelligence
- **A**vailable **P**otential **I**deas
- **A**void **P**rogram **I**lliteracy
- **A**rouse **P**rogrammer **I**nsight
- **A**ll **P**acked **I**n



You're doing great!

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.

Methods and Method Invocation

In this lesson we'll get a chance to trace the steps of an object-oriented program as it runs.

Methods

There are only two things in a Class:

- **Fields** (data types)
- **Methods**

They are often called the **Class Members**.

The two kinds of **Fields** (data types and variables to access these data types) that Java allows are:

- **Objects** (instances of Classes) and
- **Primitive Data Types** (numbers, characters, boolean)

But right now, we're going to focus on **Methods**. Specifically we want to learn how they're written, and how and when they're invoked.

Creating and Using Methods

There are two ways to obtain methods to use when you're creating object-oriented programs. You can:

- Write your own.
- Import them from existing Classes.

We've been *invoking* or *calling* methods from Classes we found in the API since Lesson 1. For example, we used `g.drawString("Hello World", 50, 50);` by invoking the method `drawString()` on the *instance* `g` of the class **Graphics**.

Usually, a Java program consists of both calls to methods from the API, and calls to methods that a programmer has written specifically for her project.

Below is code for a Class named **TriangleClassDemo**. The **Graphics** Class has a lot of methods available to draw various geometric objects, but none for triangles. We know that drawing a triangle just means to draw three lines, but if we wanted to draw 6 triangles, it would be easier to make 6 calls to a `drawTriangle` method than 18 calls to a `drawLine` method where we have to think about where the lines meet each time!

If the API isn't going to hand us an easy method, that's fine. We can just make one ourselves. The API can't do it all, but we can.

Make a new project for Lesson 10 called **java1_Lesson10**. Also make a new class called **TriangleClassDemo** that has `java.applet.Applet` as its superclass.

CODE TO TYPE: Triangles

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

public class TriangleClassDemo extends Applet {

    public void paint(Graphics g) {
        this.drawTriangle(g, 80, 120, 100, 110);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int
height){
        g.drawLine(bottomX, bottomY, bottomX+base, bottomY);
        g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);
        g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);
    }
}
```



Run it.

Isn't that a nice triangle?

Let's look at the two methods that we see defined in the **TriangleClassDemo** class: **paint()** and **drawTriangle()**

The first method we see in the class body is **paint(Graphics g)**. By now we're pretty familiar with **paint()**, so the only concept we'll review now is the use of **this** in the call. Take a look at this code:

```
this.drawTriangle(g, 80, 120, 100, 110);
```

We've seen the use of **this** with **Instance Variables** before. In Lesson 4 we took the **graph** object that was passed to us in the **paint()** method and gave it to the Instance Variable **g** **this.g = graph;**

The **this** in front of the variable indicates that it's an IV (Instance Variable) or a CV (Class Variable) of the instance of **this** particular Class.

We also saw its use in **Methods** when we set the background color of our Applet in Lesson 4:

```
this.setBackground(Color.lightGray);
```

Its use here is similar. Using **this**, we are saying that there is a method **drawTriangle()** in **this** class (or in a class from which it inherits) and we want to invoke it. Sure enough, if you look at the next defined method in our class, there it is:

```
private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height)
```

Note

The use of the keyword **this** in front of the dot operator for method calls within a given class is optional.

Edit the **TriangleClassDemo** class's **paint()** method as shown:

CODE TO TYPE

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

public class TriangleClassDemo extends Applet {

    public void paint(Graphics g) {
        this.drawTriangle(g, 80, 120, 100, 110);
        drawTriangle(g, 125, 140, 60, 70);
        // demonstrating we don't really NEED "this"
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height){
        g.drawLine(bottomX, bottomY, bottomX+base, bottomY);
        g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);
        g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);
    }
}
```



Run it.

Tip

If a **Field** or **Method** is accessed *without* a dot operator (i.e., with no instance name preceding it), then the **Class Member** being accessed is always one of the **current** class or one of its ancestors.

The only other method defined in this class is one we wrote ourselves using method calls to drawLine in Graphics. It has this declaration:

OBSERVE:

```
private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height){
    g.drawLine(bottomX, bottomY, bottomX+base, bottomY);
    g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);
    g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);
}
```

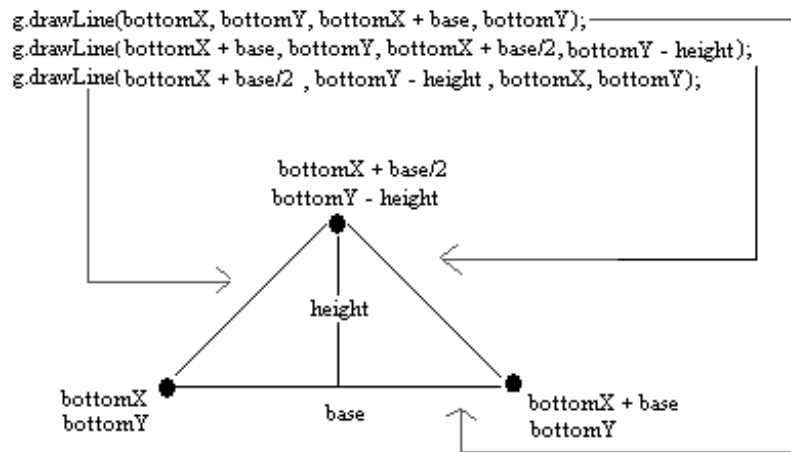
The *required* elements of a method declaration (most of which are on the first line) are:

1. The method's **return type**--in this example, the return type is **void** which means nothing is returned. (We'll discuss return type more later.)
2. The method's name--in this example, it's **drawTriangle**.
3. A pair of **parentheses ()**--in this example, (Graphics g, int bottomX, int bottomY, int base, int height).
4. A **body between braces { }**--in this example, the body includes the three **drawLine()** calls.

In the body of the drawTriangle method, the three lines for our triangle are being drawn. Our method draws an **isosceles triangle** so it will have a base, a height, and equal sides. The formal parameters requested (**Graphics g, int bottomX, int bottomY, int base, int height**) provide:

- The Graphics area on which to draw the lines--**Graphics g**
- The (x,y) location for the bottom left corner of the triangle--**int bottomX, int bottomY**
- The length of the base (bottom) and the length of the height--**int base, int height**

This diagram shows how the parameters passed will be used to draw the lines that make the triangle:



In the next section we'll begin our trace.

Tracing method calls

Recall that **Applets** are started by the browser in which they're embedded. The browser gets the **Applet** code, calls its **init()** and **start()** methods (if they've been defined; otherwise it inherits these methods from its **superclasses**). Then the Applet's **paint(Graphics g)** method is called.

Since this particular Applet does not have specified init and start methods, it inherits them from its **superclass Applet** and then calls our **paint(Graphics g)** method.

Note that the **paint(Graphics g)** method is specified as **public**:

```
public void paint(Graphics g) {
```

Note also that the **drawTriangle()** method is specified as **private**:

```
private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height){
```

So why does **paint()** have to be **public** you ask?

When a method is going to be called from *outside* of an instance of the Class itself, it must be made *accessible* to others. Its **permission** must be made **public**. This Applet's paint method is called from the browser so it needs to be accessible to the public.

When you want a method to be accessible **only from within an instance of the Class** (like when you use *this*), you make the permission modifier of the method **private**. Currently, our own method of paint is calling this method and we do not expect any other Class instances to use our **drawTriangle** method, so we made it **private**.

In an earlier lesson, we saw the use of **public** and **private** when we discussed accessors and mutators. Permissions are an important tool for maintaining the integrity, usefulness, and encapsulation of Classes, Fields, and Methods. But let's not get distracted now, just when we're about to feel the POWER of method writing!

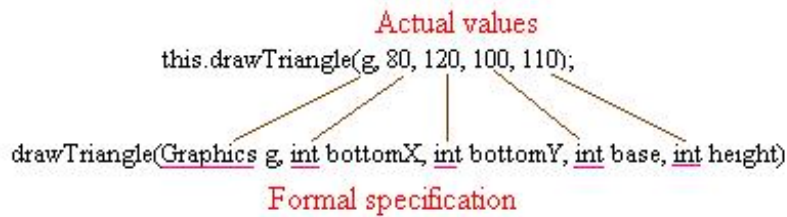


So far, the **TriangleClassDemo** Applet has been started and the paint method has been called. Inside the paint method body (between the **{}**) there was one line, then we added another to demonstrate that we can use *this* or not.

When Java runs into this statement:

```
this.drawTriangle(g, 80, 120, 100, 110);
```

...it sees *this* in front of the dot operator, so it knows to look inside the class to find the method that you want it to run. Java finds the declaration of the method and matches the actual parameters in the method call to the formal parameters of the method declaration. Here's a visual representation of what's happening:



The *actual parameters'* types match all of the formal parameters' types, so Java passes the values (a reference to the **g** object):

- **g** to **g**
- **80** to **bottomX**
- **120** to **bottomY**
- **100** to **base**
- **110** to **height**

Programming languages **pass** method parameters in two ways:

- **By value:** Primitive data types are passed giving the **value** of the variable, not the address of the variable.
Changing the value inside the method will not change anything outside the method.
- **By reference:** Object variables are passed by giving the **address** of the instance pointed to by the variable.
So if you change variables of the object within the method, it changes them outside the scope of the method too, because you gave the method the actual location of the object's information.

WARNING

When objects are passed to a method and the method returns, the passed-in reference still references the same object as before. However, the values of the object's fields may be changed in the method.

So, Java started the Applet, got into the paint method, then the paint method immediately sent us to the drawTriangle method.

Hmm. Would it make a difference if **in** the code for our class, we switched the two method definitions? Does paint have to be defined first since it is used first? Let's find out.

Move the block of code that defines the **paint()** method so that it follows the drawTriangle method's body, but is **before** the closing **}** of the entire class as shown below:

CODE TO EDIT

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

public class TriangleClassDemo extends Applet {

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int
height) {
        g.drawLine(bottomX, bottomY, bottomX+base, bottomY);
        g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);
        g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);
    }

    public void paint(Graphics g) {
        this.drawTriangle(g, 80, 120, 100, 110);
        drawTriangle(g, 125, 140, 60, 70);
    }
}
```



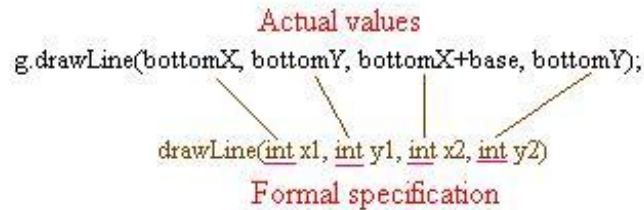
Run it. If you placed everything according to the instructions, it shouldn't have made any difference at all to

the result.

Note

The order in which the methods are defined within a class is irrelevant. It's the order in which they're **invoked** that counts.

Now that we are within the **drawTriangle()** method, the first thing Java sees is another method call: **g.drawLine(bottomX, bottomY, bottomX+base, bottomY);** This time though, the instance variable in front of the method call is not *this*, but **g**. Since **g** is of type **Graphics**, Java goes to the **Graphics** class next to see if it has a method defined that matches the actual parameters in the call. Success.



Sweet. Because these are all **int** (integers), Java will give the value of the actual parameter variables to the method parameters:

- `x1 = bottomX`
- `y1 = bottomY`
- `x2 = bottomX + base`
- `y2 = bottomY`

Notice that the variable **names** you use in the actual call don't have to match the formal specifications' names. The names don't matter because (for primitive data types) it's the *value* of the variable being passed. Also, when you pass Objects, you are passing the *address* of the variable, but an address can have more than one name. Sometimes this can cause undesired side-effects when more than one variable points to the same object.

When Java sees the method call **g.drawLine(bottomX, bottomY, bottomX+base, bottomY);** it goes to the **Graphics** class and its implementation of **drawLine()**, and then executes that method.

When that line of code is finished, Java goes to the next `g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);` and matches up its parameters, then goes to the **Graphics** class and its implementation of **drawLine()** and executes *that* method.

Then Java executes that entire process again for the method call `g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);`

When Java sees it's finished with the **drawTriangle()** method, it goes back to the original method that called it, the **paint()** method. Since **paint()** has nothing else in it, Java is done. If you didn't remove the second call to **drawTriangle()** though, Java would start the whole process again.

So far so good. Keep going, you're doing a great job!

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.

Writing Classes - Building With Methods

More on Methods

Local Variables

In the last lesson, the parameters inside the `drawTriangle` method may have been a little hard to read, especially in the `drawLine()` methods:

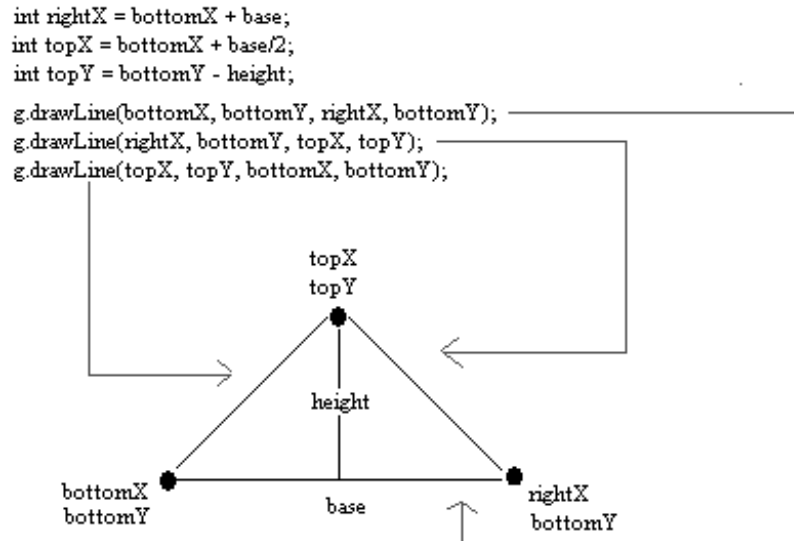
```
g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);
```

We generally think of triangles as 3 lines connecting 3 points, not so much a set of parameters like we've been given here.

So let's use the parameters passed to us (`g`, `bottomX`, `bottomY`, `base`, and `height`) to create the 3 points of the triangle *before* using the `drawLine()` method.

We'll edit our code to use variables that are **not** Instance Variables, Class Variables, or Method Parameters. Instead we'll use **Local Variables**. Local variables are variables that are only known within the **scope** of their **block of code**. In this case, within the body of a method.

Here's an illustration of what we're doing:



Because the bottom of the triangle is horizontal, the `bottomY` coordinate is used twice.

Once we have the formal parameters passed to the method `drawTriangle()`, we can use them to make three new local variables:

```
int rightX = bottomX + base;
int topX = bottomX + base/2;
int topY = bottomY - height;
```

Then we can use these as coordinates for three points:

<`topX`, `topY`>, <`bottomX`, `bottomY`>, and <`rightX`, `bottomY`>

Now it'll be easier to decipher the parameters we send in the three `drawLine()` calls (as shown in the diagram above).

Make a new project for Lesson 11 called `java1_Lesson11`, and make a new class called `MethodDemo` that uses the super `java.applet.Applet`. Then, edit `MethodDemo` as follows:

CODE TO TYPE: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void paint(Graphics g) {
        drawTriangle(g, 80, 120, 100, 110);
        drawTriangle(g, 125, 140, 60, 70);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height) {
        int rightX = bottomX + base;
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        //easier to read drawLine calls
        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }
}
```



Run it. It should look familiar.

Two important things you should know about local variables:

- Local variables *do not* get *default values*. You must give them initial values.
- Local variables only exist within their block of code. The same is true of method parameters. That is, the scope of local variables and method parameters is solely within the body of the method in which they are defined.

In your new **MethodDemo** Class, edit the Local Variables as follows:

CODE TO EDIT: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void paint(Graphics g) {
        drawTriangle(g, 80, 120, 100, 110);
        drawTriangle(g, 125, 140, 60, 70);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height) {
        rightX = bottomX + base;
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        //easier to read drawLine calls
        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }
}
```

You should see this error: **rightX cannot be resolved**. The error occurs because we haven't declared the variable **type**.

Now edit the code as follows and see what happens:

CODE TO EDIT: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void paint(Graphics g) {
        drawTriangle(g, 80, 120, 100, 110);
        drawTriangle(g, 125, 140, 60, 70);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height) {
        int rightX;
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        //easier to read drawLine calls
        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }
}
```

Not surprisingly, because we just said that local variables *do not get default values*, we get the error: **The local variable rightX may not have been initialized**

Edit the code again so it looks like this:

CODE TO EDIT: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void paint(Graphics g) {
        rightX = 42;
        drawTriangle(g, 80, 120, 100, 110);
        drawTriangle(g, 125, 140, 60, 70);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height) {
        int rightX = bottomX + base;
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        //easier to read drawLine calls
        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }
}
```

This time you get the message: **rightX cannot be resolved**, because it's outside of the scope of the method in which it was declared. Okay, but it's not initialized there. Go ahead and initialize it:

CODE TO EDIT: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void paint(Graphics g) {
        int rightX = 42;
        drawTriangle(g, 80, 120, 100, 110);
        drawTriangle(g, 125, 140, 60, 70);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height) {
        int rightX = bottomX + base;
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        //easier to read drawLine calls
        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }
}
```

Cool, all the errors go away. But, since both **rightX** occurrences are individually declared in each method, they have **different values!**

To prove this, edit it as follows:

CODE TO EDIT: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void paint(Graphics g) {
        int rightX = 42;
        g.drawString("rightX before the method call is " + rightX, 5,170);
        drawTriangle(g, 80, 120, 100, 110);
        drawTriangle(g, 125, 140, 60, 70);
        g.drawString("rightX after the method call is " + rightX, 5,200);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height) {
        int rightX = bottomX + base;
        g.drawString("rightX in the method is " + rightX, 5,185);
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        //easier to read drawLine calls
        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }
}
```



Run it.

And that's why they're called **local**! Here's a diagram that shows how scope works:

Class

```
String Cvariable = "I'm a class var";
```

Method A

```
String myVar1 = "inside A";
```

Method B

```
String myVar2 = "inside B";
```

Methods **can** have their own local variables. In the diagram above, the Class variable **Cvariable** can be accessed by **Method A** and **Method B**. However, the method variable **myVar1** is only accessible in **Method A**, while **myVar2** is only accessible in **Method B**.

Note

For better readability and class design, limit the scope of your variables, and keep them as small as possible.

Of course local variables are not only for enhanced readability; sometimes they're used for computations of methods. If local variables are only needed for a given method, then they should only be present while that method is being called.

WARNING

Local variables are **re-initialized** each time the method is called. Their previous value will not be present when the method is called again.

Results and Return

In lesson 10, we mentioned that the only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}. So far we have covered everything except the **return** type.

To get results from a method (some call these *queries* as opposed to *commands*), we need to use the reserved word **return**.

Methods that return information have two important traits:

- The return **type** must be specified in the method declaration.
- The method must use the reserved word **return** followed by an expression that matches the specified return type.

Let's start a new class called **ReturnDemo** that extends Applet. **Type** into ReturnDemo as shown by the code in **blue** below:

CODE TO TYPE: ReturnDemo.java

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

public class ReturnDemo extends Applet {

    public void paint(Graphics g) {
        int answer = areaRectangle(30,40);
        g.drawString("area of rectangle is " + answer, 20, 20);
    }

    private int areaRectangle(int sidel, int side2) {
        int area = sidel * side2;
        return area;
    }
}
```



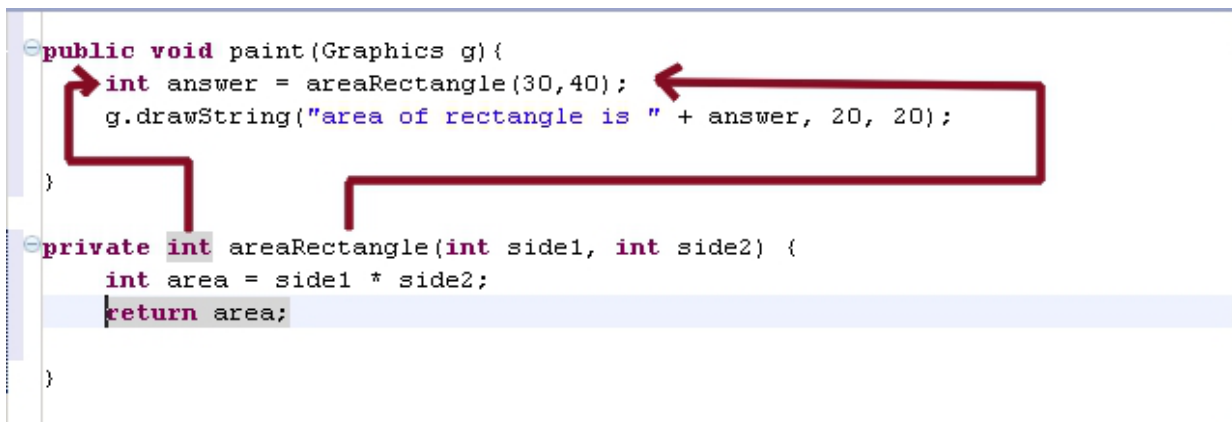
Run it.

Notice that when a call to a method **returns** something, you can't just call the method with

areaRectangle(30,40);

Because we asked it to *return* something, we need a place in memory to **put** the answer that it returns.

Actually, we *could* make that call without choosing a destination for our result, and we wouldn't get any errors, but it would pretty much be like doing nothing at all. Java will execute the method, but then since it has no place to **put** the returned result, Java will go on its merry way as if the method never existed.



Let's trace the code.

In the **paint()**, **areaRectangle(30,40);** is a call to a method which has a return type of **int**. So we must declare a variable to put the returned value into and declare it to be of the same type that the method returns (in this case **int**). We did that with **int answer = areaRectangle(30,40);**

Likewise, since **areaRectangle()** has an **int** return type, the variable **area** in the line **return area;** must be of type **int**. (That's why we have **int area = sidel * side2;**).

Edit the **ReturnDemo.java** class as shown:

CODE TO EDIT: ReturnDemo.java

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

public class ReturnDemo extends Applet {

    public void paint(Graphics g) {
        int answer = 0;
        areaRectangle(30,40);
        g.drawString("area of rectangle is " +answer, 20, 20);
    }

    private int areaRectangle(int sidel, int side2) {
        int area = sidel * side2;
        return area;
    }
}
```



Run it. Of course! We told the computer that the answer was 0 and it stayed that way. This seems pretty simple, but it's pretty important to remember: **Computers will do exactly what you tell them to do!**

Okay, let's go over one more groovy aspect of computer languages. Edit the **paint()** method as shown:

CODE TO EDIT: ReturnDemo.java

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

public class ReturnDemo extends Applet {

    public void paint(Graphics g) {
        g.drawString("area of rectangle is " + areaRectangle(30,40), 20, 20);
    }

    private int areaRectangle(int sidel, int side2) {
        int area = sidel * side2;
        return area;
    }
}
```

Remember earlier when we said that anywhere you can put a value, you can put an expression? Well, we just demonstrated that here. Java will go and do the method, and since it's in the **String** parameter for the **drawString()** method, it will automatically cast the returned **int** value to a **String**, concatenate it to the previous **String**, "area of rectangle is," and print the whole **String** with the concatenated result.

Go ahead and **Run** it.

Building on methods

Now that we have a handy **drawTriangle()** method, let's use it to build a house. Open the **MethodDemo** class you worked on earlier in this lesson, and change the **paint()** method and add a **drawHouse()** method as shown in **blue** (remove the code shown in **red**):

CODE TO EDIT: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void paint(Graphics g) {
        drawHouse(g, 10, 100, 70, 30);
        drawHouse(g, 100, 50, 60, 20);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height) {
        int rightX = bottomX + base;
        g.drawString("rightX in the method is " + rightX, 5, 185);
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }

    private void drawHouse(Graphics g, int bottomX, int bottomY, int width, int height) {
        int rightX = bottomX + width;
        int topX = bottomX + width/2;
        int topY = bottomY - height;
        int halfHeight = height/2;

        g.drawRect(bottomX, topY, width, height);
        this.drawTriangle(g, bottomX, topY, width, halfHeight);
    }
}
```



Run it. Hey, they kind of look like envelopes, too. So go ahead and think of them as houses or envelopes.

After a method is finished, control returns to the method from which it was initially invoked. Let's **trace** this program so we can see that more clearly.

OBSERVE: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void 1paint(Graphics g) {
        2drawHouse(g, 10, 100, 70, 30);

        9drawHouse(g, 100, 50, 60, 20);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height) {
        7int rightX = bottomX + base;
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        8g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }

    private void 3drawHouse(Graphics g, int bottomX, int bottomY, int width, int height) {
        4int rightX = bottomX + width;
        int topX = bottomX + width/2;
        int topY = bottomY - height;
        int halfHeight = height/2;

        5g.drawRect(bottomX, topY, width, height);
        6this.drawTriangle(g, bottomX, topY, width, halfHeight);
    }
}
```

Let's check out the steps Java takes to execute this program in order:

1. The Applet calls its inherited **init()** and **start()**, and then it calls **paint()**.

2. In the **paint()** method, **drawHouse(g, 10, 100, 70, 30);** is called.

3. Java sets the formal parameters to:

1. g = g
2. bottomX = 10
3. bottomY = 100
4. width = 70
5. height = 30

4. Inside of the **drawHouse()** method, some local variables are set:

1. int rightX = bottomX + width;
2. int topX = bottomX + width/2;
3. int topY = bottomY - height;
4. int halfHeight = height/2;

5. Java sees **g.drawRect(bottomX, topY, width, height)**, so it goes to the package **java.awt**, to the class **Graphics**, finds the method **drawRect()**, and uses the actual parameters passed to run it and draw the rectangle.

6. Java sees a call to *this* class's method named **drawTriangle()**, so Java goes to its definition and sets its formal parameters.

7. Some local variables, based on the actual parameters, are defined inside of **drawTriangle()**:

1. `int rightX = bottomX + base;`
2. `int topX = bottomX + base/2;`
3. `int topY = bottomY - height;`

8. Java encounters each of the **g.drawLine()** methods one at a time, each time it goes to **java.awt** and finds the **drawLine()** method in the **Graphics** class.

9. Once Java is done with the **drawTriangle()** method, it's done with the first **drawHouse()** method, but Java is not done with the **paint()** method. Because Java sees *another* call to **drawHouse()**, it performs steps 2 through 8 above all over again, but this time with different parameters.

We can see now that the *order* in which we **define** our methods makes no difference. (But the order in which we **call** them **can** matter). Java will go where you tell it to go within the methods.

We've come a really long way, but let's not stop here. Let's add a little more and paint the houses.

Overloading

How Does Java Find the Right Method?

Edit **MethodDemo** to include the **blue** code below:

CODE TO EDIT: MethodDemo.java

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

public class MethodDemo extends Applet {
    public void start(){
        resize(400,200);           // make it bigger so we do not have to expand
    }

    public void paint(Graphics g) {
        drawHouse(g, 50, 50, 70, 30);
        // for these, added another parameter for house color
        drawHouse(g, Color.red, 100, 50, 60, 20);
        drawHouse(g, Color.cyan, 150, 100, 160, 50);
    }

    private void drawTriangle(Graphics g, int bottomX, int bottomY, int base, int height){
        int rightX = bottomX + base;
        int topX = bottomX + base/2;
        int topY = bottomY - height;

        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }

    private void drawHouse(Graphics g, int bottomX, int bottomY, int width, int height){

        int rightX = bottomX + width;
        int topX = bottomX + width/2;
        int topY = bottomY - height;
        int halfHeight = height/2;

        g.drawRect(bottomX, topY, width, height);
        this.drawTriangle(g, bottomX, topY, width, halfHeight);
    }

    // provide another drawHouse method that paints if passed a color
    private void drawHouse(Graphics g, Color paintMe, int bottomX, int bottomY, int width, int height){
        int topY = bottomY - height;
        drawHouse(g, bottomX, bottomY, width, height); //draw the house using the original signature of drawHouse.
        g.setColor(paintMe);           // set color to that passed
        g.fillRect(bottomX, topY, width, height);
    }
}
```



Run it. How about that? We painted the side of the houses. Good job! (We'll leave it to you to fill in the roof! Maybe you'll want to overload drawTriangle?)

What? Two method definitions with the same name? Yep, but they have different parameters. In fact the second definition has 6 parameters. The second parameter in the second definition is a **Color**. How does Java know which one to use?

Java takes these steps to find a method that has been invoked:

1. Methods are invoked with the dot operator, so Java always knows what *kind* of Object is being used to execute the method.
2. Once Java knows the object, it knows the *Class* the object is an instance of because every variable must be declared as a *type*.
3. Once Java knows the Class, it looks through that Class's methods for the proper method name.

4. If more than one method has the same name, we have *overloading*, so Java will check parameters (number and type) to find the match. The number and type of parameters determine the method's **signature**.

Overloaded methods are differentiated by the number and type of the arguments passed to them (their *signature*). In a given Class, you can't declare more than one method with the same name *and* the same number and type of arguments, because the compiler can't tell them apart.

WARNING

The compiler doesn't consider return type when differentiating methods, so you can't declare two methods with the same signature, even if they have different return types.

Because the compiler does not consider return type, different sources have different definitions for the *signature* of a method.

For a language to distinguish between overloaded method calls (that is, when the class type and method have the same name), it uses the method's name and the parameter number and types to prevent ambiguity.

Summary

Method Declarations

One more time: The only *required* elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

Method declarations have six components, in order:

1. Modifiers--such as **public** or **private** (permissions), and others we'll see in the next lesson.
 2. Return type--the data type of the value returned by the method, or **void** if the method does not return a value.
 3. Method name--similar to variable names, with the added recommendation that it begin with a *verb* since methods are *actions*.
 4. The parameter list in parentheses--a comma-delimited list of input parameters, preceded by their data types, and enclosed by parentheses ().
- If there are no parameters, you still must use empty parentheses.
5. An exception list--we'll cover this later.
 6. The method body, enclosed between braces--the method's code, including the declaration of local variables, goes here.

main: an important method

We mentioned earlier that Java programs that do not run on a browser **require** a *main* method to get them started. The main method is the top-level method that initiates execution of a program that is not running on a web browser. It looks like this:

```
public static void main (String[] args) { }
```

When we want to start an application, we need to find the Class that has the main method in it (call it *ClassWithMain*), and use this command:

```
java ClassWithMain
```

Good design practice dictates that the main method should do nothing but instantiate and start the Classes of the application. Given this, some programmers define a class named **Main.java** and the only Member of that class is the *main* method. This makes it easy to know how to start Classes, because instead of looking at all of the Classes to find the *main*, you can simply call `java Main` every time.

Finally, note that if you do **not** do this, and you have your *main* within a Class called *ExampleClass*, then the main code needs to **explicitly** instantiate the Class and call a method to get it started.

Start a new Class called **ExampleClass**. This time it doesn't extend *Applet* so leave the superclass as `java.lang.Object`.

CODE TO TYPE: ExampleClass

```
public class ExampleClass
{
    int testInstanceVariable = 42;

    public static void main(String[] args)
    {
        System.out.println("The value of the instance variable is " + testInstanceV
        ariable);
    }
}
```

Looks simple, huh? Although there are some errors generated in this Class that mention *static*, and indicate **Class** Variables or Methods, we'll hold off on discussing them in depth until 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.

Adding Interaction using Components and Listeners

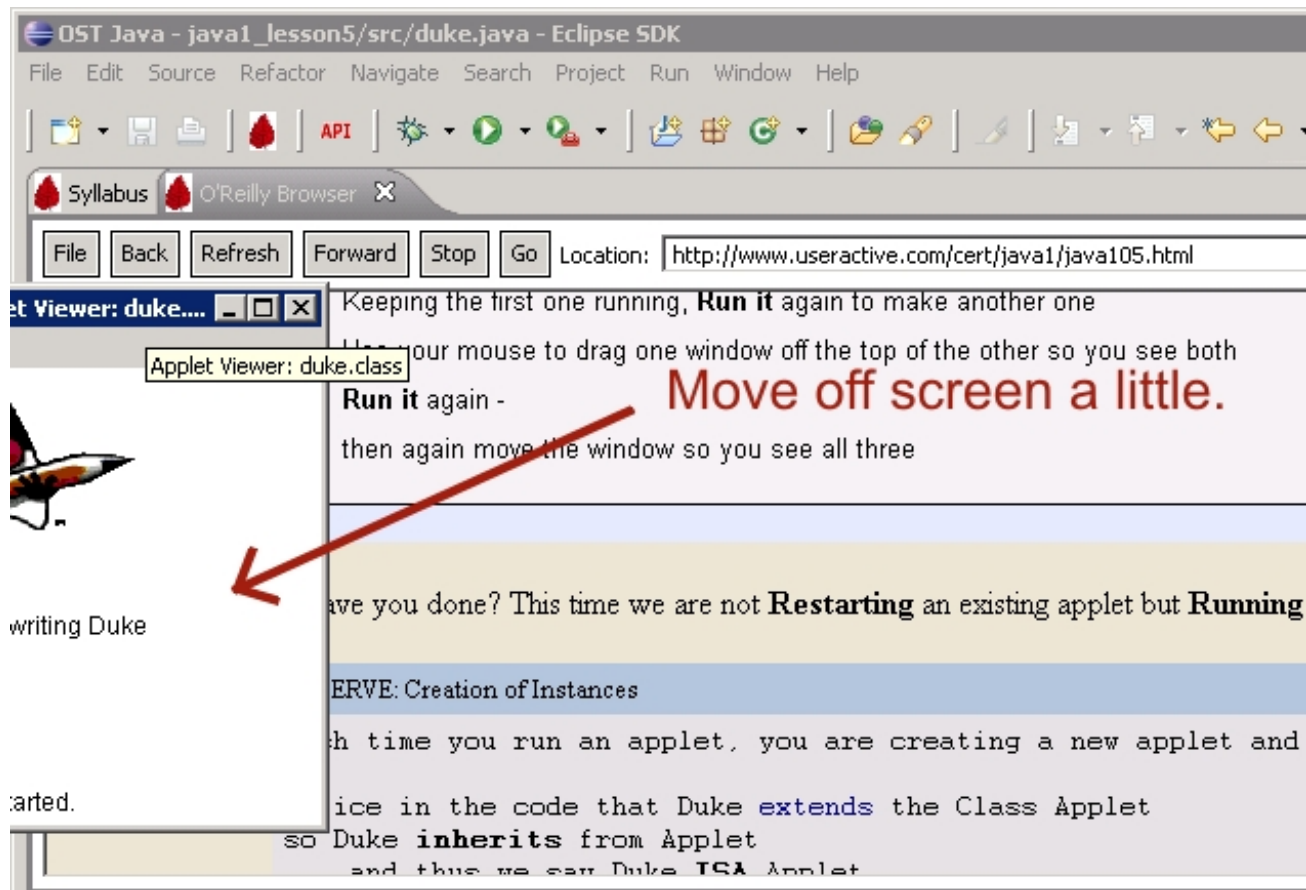
Revisiting the Dukes Class and Applet

Let's revisit our old pal Duke (deep down you know you love him). Grab the Dukes class and the Dukes Applet we made in lesson 7.

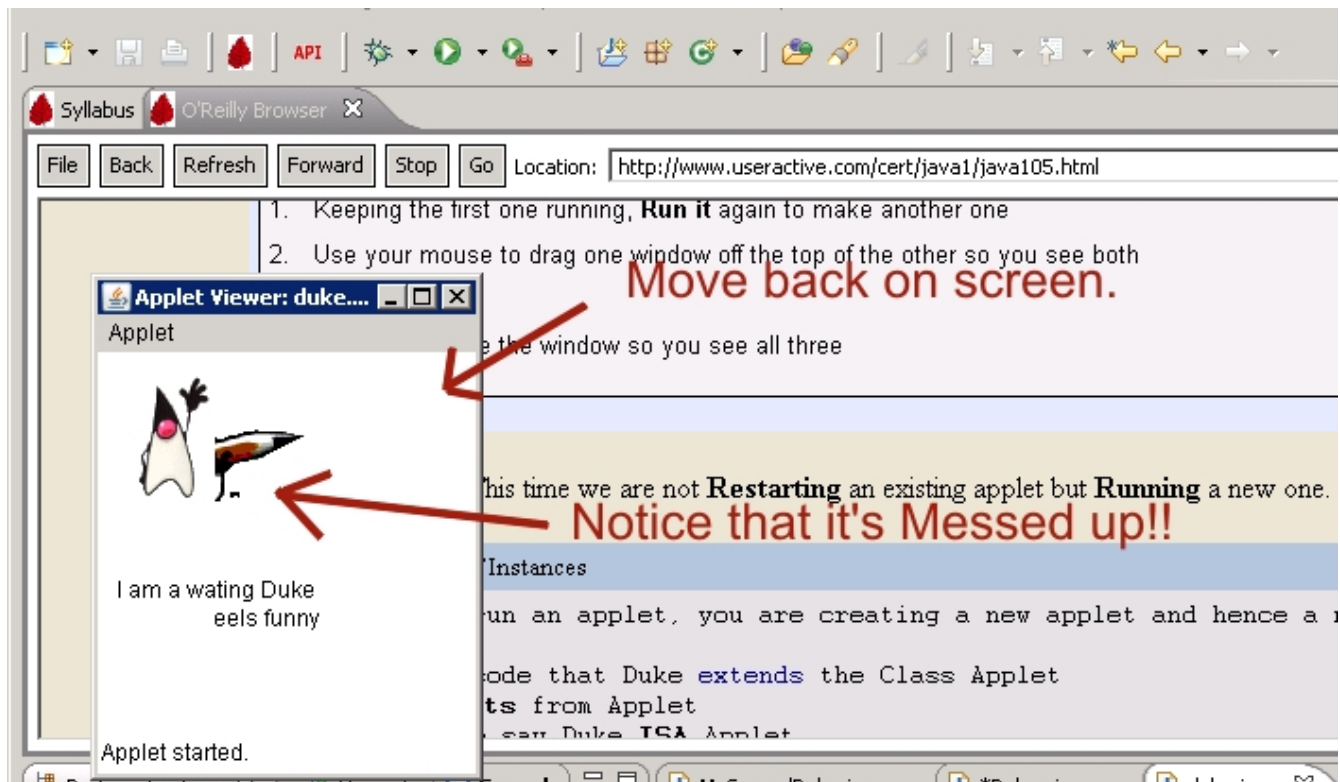
To do

1. Start a new project called java1_Lesson12.
2. Go to your **java1_Lesson7** project and **copy Dukes.java** (right-click and Copy).
3. Go to your **java1_Lesson12** project and **paste Dukes.java** into the **src folder**.
4. Now do the same thing with **DukesApplet.java** (copy and paste from java1_Lesson7/src).
5. Open up **DukesApplet.java** and **Run** it.

If you move the window around, Duke can get a little messed up. Try moving the Applet off of the screen like this and see:



Now, move it back on screen and notice any **changes**:



Do it a few more times.

The Applet gets messed up when we refresh the screen like that.

The problem is that Dukes only changes the part of itself that needs to be repainted (in this case, the part that was offscreen). The call to paint the Applet comes from the Applet itself, not from the user. But Java is a language for building software, and software is for humans to use. So let's give the users something to click on that makes Dukes change upon user direction instead.

A User Modification Example

Now let's get the user to interact with our Applet. Again, using the power of **modularity**, we'll leave the Dukes class as it is, and just present it differently, using a different Applet. The **DukesApplet** class will change because that's where the "presentation" or Graphical User Interface (GUI) is located in Applets. In this example we're going to add a GUI component called a "List" and use a "Listener" which is a type of Interface. We'll refer to the example to explain these new concepts in detail later in the lesson:

Start a new class called **DukesAppletGUI**. Make sure it has the `java.applet.Applet` Superclass.

CODE TO TYPE IN BLUE: DukesAppletGUI.java

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

public class DukesAppletGUI extends Applet implements ActionListener{

    Dukes myDuke;        // Instance Variable giving the instance name "myDuke"
    String action;        // Instance Variable telling what action is being done

    public void init()      // init method
    {
        List actionList = new List(3); // makes a list to choose from
        actionList.add("wave");        // give the list 3 choices
        actionList.add("think");
        actionList.add("write");

        actionList.addActionListener(this); // tell Java to listen for user input
        add(actionList);                    // add the list to the Applet

        myDuke =new Dukes();                // make an instance of Duke
        action = myDuke.getActionImage();    // see what Duke's current action is
    }

    public void paint(Graphics g)    // paint method
    {
        Image myAction = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10,165);
        g.drawString(myDuke.getMessage(), 10,180);
        g.drawImage(myAction, 20, 50, Color.white, this);
    }

    public void actionPerformed(ActionEvent evt)
    {
        String userChoice = evt.getActionCommand();
        if (userChoice == "write") action = myDuke.write();
        else if (userChoice == "think") action = myDuke.think();
        else if (userChoice == "wave") action = myDuke.wave();

        repaint(); // if a different choice has been made, call our paint through repai
    }
}
```



Run it. To pick a choice in the box, double-click on it. Scroll for more choices.

Before we examine this code in detail, let's discuss **interfaces**.

Introduction to Interfaces

Interfaces are quite simple to create and use, but they can still be tricky to understand. In this course we're just going to learn the basics, but there's more to come on interfaces later in the Java course series. For Java programmers, the word *interface* can have several meanings. For instance, there is a Graphical User *Interface* (GUI), which is an interface between users and a piece of software. But that's not the definition we're using in this lesson, even though we're using a Listener *Interface* to make a GUI for our program. We're using the term **Interface** here to mean a **type**, sort of like a Class that defines methods, but doesn't *implement* them. The term **interface** is used because this type is *analogous* to an actual interface you're used to using, but it's more like an interface between Java objects.

So Java has a **type** called an **Interface** that defines methods, but doesn't implement them. And when we say *doesn't implement the methods*, we mean there are no brackets {}, and so no code between them to be implemented. So the methods in an interface **don't do anything--yet**. Let's take a look. The ActionListener interface we're using in this example looks like this in the API:

Definition of the ActionListener Interface from the API

```
public interface ActionListener extends java.util.EventListener{

    public void actionPerformed(ActionEvent e);

}
```

We found the definition in the [API entry on the interface ActionListener](#). What else do we know about this interface? In addition to the definition, we know that methods aren't implemented in an interface.

This particular ActionListener interface only has *one* method, but an interface can have any number of methods. Notice that the **actionPerformed()** method doesn't have a body, so it doesn't do anything. It's set up to receive an object of type **ActionEvent**, but because its return type is **void**, it isn't supposed to return anything. An interface's methods are implemented when we **implement** them, when we define its methods. Once we do that, then our object can claim that it's the same type of object as the interface.

Why would anyone want to define an interface? Why would anyone want to define methods that aren't implemented? Well, there are several reasons. It simulates multiple inheritance, and aids in polymorphism. And once we implement an interface, other objects know **for sure** that we are implementing a set of methods, and we know that if we implement these objects, that we can receive all that those objects have to offer us **for sure**. In fact, **by rule of implementation** we agree as programmers to implement **all** of the methods defined in the interface we are implementing. This concept is best illustrated in the **Listeners** interface.

You can see that in our **DukesAppletGUI** we've obeyed the rules of implementation by implementing all of the methods of ActionListener:

DukesAppletGUI implements ActionListener Methods

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

public class DukesAppletGUI extends Applet implements ActionListener{

    Dukes myDuke;        // Instance Variable giving the instance name "myDuke"
    String action;        // Instance Variable telling what action is being done

    public void init()      // init method
    {
        List actionList = new List(3); // makes a list to choose from
        actionList.add("wave");         // give the list 3 choices
        actionList.add("think");
        actionList.add("write");

        actionList.addActionListener(this); // tell Java to listen for user input
        add(actionList);                     // add the list to the Applet

        myDuke =new Dukes();                 // make an instance of Duke
        action = myDuke.getActionImage();     // see what Duke's current action is
    }

    public void paint(Graphics g)    // paint method
    {
        Image myAction = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10,165);
        g.drawString(myDuke.getMessage(), 10,180);
        g.drawImage(myAction, 20, 50, Color.white, this);
    }

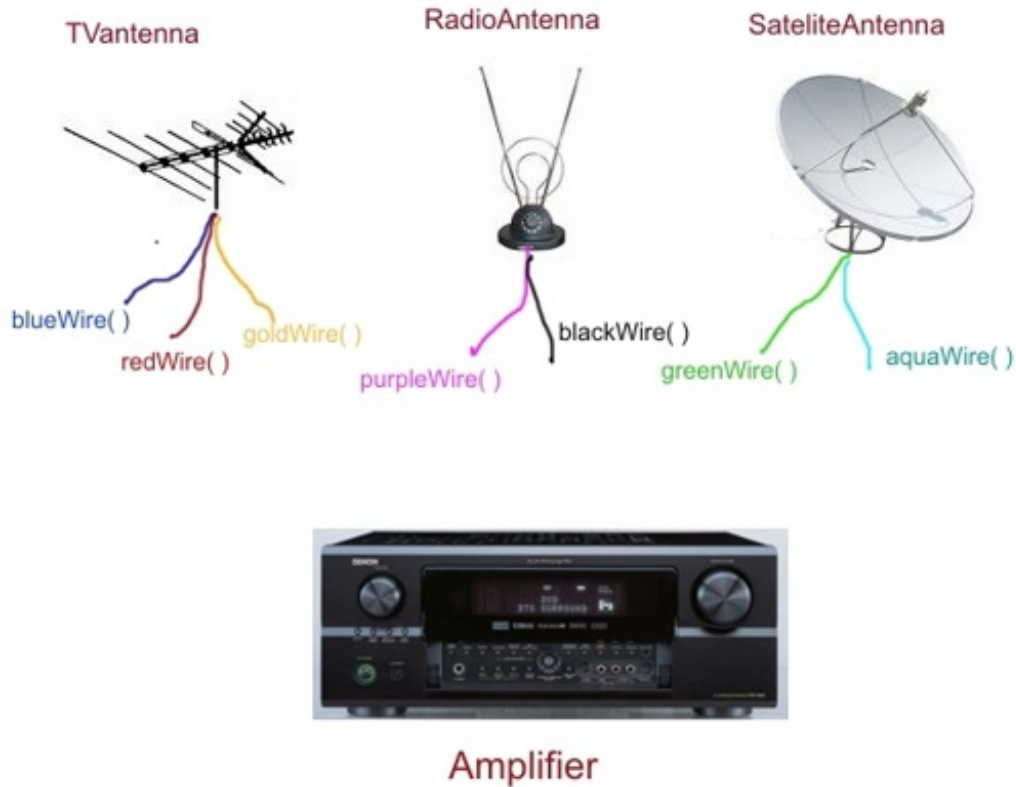
    public void actionPerformed(ActionEvent evt)
    {
        String userChoice = evt.getActionCommand();
        if (userChoice == "write") action = myDuke.write();
        else if (userChoice == "think") action = myDuke.think();
        else if (userChoice == "wave") action = myDuke.wave();

        repaint(); // if a different choice has been made, call our paint through repai
    }
}
```

By implementing **ActionListener**'s methods, we've actually given them something to do! Our DukesAppletGUI has followed the rules of implementation, so now the ActionListener interface is not only an Applet, but an ActionListener as well. And now our Dukes Applet can receive messages from objects that send them to ActionListener, and even our Dukes Applet does something with those things.

An Analogy: Antenna as an Interface

Listeners are great examples of interfaces and also lend themselves to a nice analogy. I like to think of an interface as a type of antenna. An antenna is really an interface between something sending a signal and your radio, which receives the signal and then does something with it. Taking our analogy a little further, your radio's amplifier will implement an antenna in order to receive signals. Of course, this amplifier will take signals and process them. Well, suppose each type of antenna has different kinds and numbers of wires, which play the part of *methods* in our example. So in the analogy, the **Amplifier** is a **Class**, the **Antennas** are **interfaces**, and the **wires** are **methods** defined in each of the interfaces. Oh, and the **RadioStation** is an **object** that sends out different signals, or parameters. Check out this illustration:



In our analogy, in order to implement one of the Antenna, our **amplifier** *must* connect or implement all of the wires of that Antenna.

To take our analogy further still, if we want to play a radio station (a **RadioStation** object, if you will) on our Amplifier, then we need to **implement** the **RadioAntenna** interface by implementing all of its methods (wires): **purpleWire()** and **blackWire()**. The RadioAntenna interface has a **blackWire()** and a **purpleWire()**, but they don't actually do anything until we implement them with our Amplifier. Once we do that, we can receive signals from the RadioStation object.

Let's really get carried away with our Antenna analogy now and write down Java code that represents this situation. First, let's define the RadioAntenna interface:

The RadioAntenna Interface

```
public interface RadioAntenna {

    public void purpleWire(Signal S);

    public void blackWire(Groud G);

}
```

Now suppose we have an Amplifier Class that **implements** this interface and instantiates the RadioStation object (which sends stuff to the interface).

Amplifier class instantiating the RadioAntenna

```
public Class Amplifier extends ElectronicDevice implements RadioAntenna{

    RadioStation Jazz = new RadioStation(103.4);
    Jazz.addRadioAntenna(this);

    public void purpleWire(LeftSignal l){

        Code that does something with l

    }

    public void blackWire(RightSignal r){

        Code that does something with r

    }

}
```

Now, the Amplifier Class has properly implemented the RadioAntenna interface, and the **RadioStation** is sending signals. Since we have an Antenna interface, the RadioStation object has an **addRadioAntenna()** method. So when the RadioStation sends a signal, Java automatically calls the *registered* interface methods **purpleWire()** and **blackWire()** as needed. (All of that happens behind the scenes and we don't need to worry about it.) Those methods would be defined in the RadioStation Class (remember this is an analogy).

The Listener Interfaces

Now let's examine this code and learn what's happening here. Even though it looks like we've done a lot, in reality we've only introduced two new things to this Applet. We've added a **List** Object which, when clicked, will send messages to our interface. We've also *implemented* a type of **interface** called a **Listener**. Our particular "Listener" is **ActionListener** (there are others). A listener's job is to "listen" for events that users can perform on a computer like click, double-click, move the mouse, and so on. By implementing the ActionListener *Interface*, we've made our DukesAppletGUI a Listener, and it's listening for particular events. Look at the color coding of our code below so we can discuss this further:

DukesAppletGUI.java

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

public class DukesAppletGUI extends Applet implements ActionListener{

    Dukes myDuke;        // Instance Variable giving the instance name "myDuke"
    String action;        // Instance Variable telling what action is being done

    public void init()      // init method
    {
        List actionList = new List(3); // makes a list to choose from
        actionList.add("wave");         // give the list 3 choices
        actionList.add("think");
        actionList.add("write");

        actionList.addActionListener(this); // tell Java to listen for user input
        add(actionList);                    // add the list to the Applet

        myDuke = new Dukes();               // make an instance of Duke
        action = myDuke.getActionImage();    // see what Duke's current action is
    }

    public void paint(Graphics g) // paint method
    {
        Image myAction = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10, 165);
        g.drawString(myDuke.getMessage(), 10, 180);
        g.drawImage(myAction, 20, 50, Color.white, this);
    }

    public void actionPerformed(ActionEvent evt)
    {
        String userChoice = evt.getActionCommand();
        if (userChoice == "write") action = myDuke.write();
        else if (userChoice == "think") action = myDuke.think();
        else if (userChoice == "wave") action = myDuke.wave();

        repaint(); // if a different choice has been made, call our paint through repaint
    }
}
```

This example works just like our analogy! By **implementing** the **ActionListener** interface, we've made it so that objects like buttons and lists can call our **actionPerformed(ActionEvent evt)** and pass it a parameter of type **ActionEvent**. Then our implemented **actionPerformed()** method takes that event and processes it with the code we added to our Applet. In this case, the **ActionEvent** object that was passed to us has its own method called **getActionCommand()**, which grabs the item in the **List** that was clicked. Our code checks each possible *userChoice* and calls our Dukes object with the corresponding method [write(), think(), or wave ()].

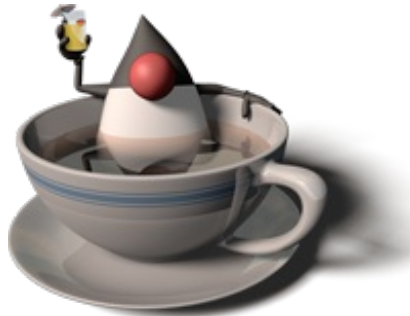
Finally, let's talk about the **List** object. Like the other Components in AWT, such as buttons and check boxes, this **List** is an Object that we create an instance of using the **new** call. The **List** object takes a number as a parameter so that it knows how many items are in the selection list (in this case, 3). We named this instance of **List**, **actionList**. The **List** object has its own methods, one of which is **add()**. We call **add()** from our instance as **actionList.add("think")**. That way, we add all of the choices ("think", "wave", and "write") to the **List**.

You might have wondered how the **List** object knows to call the **actionPerformed()** method at all. Well, if you look in the API, the **List** component has a method called **addActionListener(ActionListener S)** which takes an **ActionListener** as its parameter. In this case, **this** is **DukesAppletGUI**. We tell our **List** object to add **this** by calling **actionList.addActionListener(this)**. And of course **DukesAppletGUI** is an **ActionListener** because it implemented the **ActionListener** interface. The call to **addActionListener()** tells Java to call all of the appropriate methods listed in the **ActionListener** interface which Java knows. As you'll see in a later lesson, these components can add other listeners too, and we can implement other interfaces to capture their events.

Finally, we *add* the instantiated **List actionList** to the Applet using the Applet's inherited **add()** method, a method

specifically for adding components to the Applet.

As the course progresses, **you** will implement and change many of these Classes in all kinds of ways. For now, the goal is to understand the basics of Classes--that they define variables and methods, which help to define the Class itself and its capabilities. In the next lesson we'll write our own classes again, but we'll **use** some of the Classes that Java has written for us to make coding easier. See you there! Cheers!



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.*

Modularity: Modifiers, Permissions, and Scope

Class Specifications

Programming styles and languages may differ, but the basic ideas behind object-oriented design remain the same:

- **modularity**
- **encapsulation** (includes data-abstraction and information hiding)
- **inheritance**
- **polymorphism** (overloading, overriding)

We'll return to our beloved **Dukes** class to illustrate how some aspects of object-oriented design are used to implement the Java language. And we'll get to play around and learn some more tricks. After all, programming really is all about the tricks!

By the end of this lesson, you'll have a more complete picture of the structure of **Classes** which will include:

- **Modifiers**
 - Access Modifiers: Permissions
 - Class Methods and Variables
- **Scope**

Modularity

Alright, let's get started with the Dukes class from Lesson 7. You remember Dukes, right? How could you forget?

So far we've re-designed our Dukes code a couple of times to demonstrate modularity:

- Our first Duke example (java1_Lesson3) was all in one Applet file. Here's the running Applet.
- Later we edited Dukes to separate the Dukes information from the Applet information. The running Applet didn't look any different, but the code was cleaner.
- Then we edited our project again to allow the user to choose what action for Duke to take. We kept the same Dukes.java class, but only edited the GUI component choice in the Applet. Here is the running Applet.

One of the great characteristics of modular code (with unique Classes) is that it's much easier to edit, modify, and reuse. In this lesson, we'll change our Applet's appearance by editing the Applet Class. Then we'll add aspects to our Dukes by editing the Dukes Class.

First, let's add to our bucket of tricks. We'll edit the Applet to make a **drop-down menu** list to use instead of the scrollable list we did earlier. The easiest way to make these changes will be to edit stuff we already have. Let's grab a copy of the **DukesAppletGUI.java** file we made in the last lesson, and also reuse the **Dukes.java** we've been using all along.

Once again, make a **new project for Lesson 13** and call it **java1_Lesson13**. Once you've done that, let's copy some files we've already made.

**To
do**

1. *Copy and Paste* **Dukes.java** from the java1_Lesson7 project to the java1_Lesson13 project.
2. *Copy and Paste* **DukesAppletGUI.java** from the java1_Lesson12 project to the java1_Lesson13 project.

Now let's edit DukesAppletGUI.java:

CODE TO EDIT IN BLUE: DukesAppletGUI.java

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

public class DukesAppletGUI extends Applet implements ItemListener{

    Dukes myDuke;
    String action;

    public void init() {

Choice actionList = new Choice();
        actionList.add("wave");
        actionList.add("think");
        actionList.add("write");

actionList.addItemListener(this);
        add(actionList);

        myDuke =new Dukes();
        action = myDuke.getActionImage();
    }

    public void paint(Graphics g)    // paint method
    {
        Image myAction = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10,165);
        g.drawString(myDuke.getMessage(), 10,180);
        g.drawImage(myAction, 20, 50, Color.white, this);
    }

    public void itemStateChanged(ItemEvent evt){
        int whichOne = ((Choice)evt.getItemSelectable()).getSelectedIndex();
        switch (whichOne)
        {
            case 0: action= myDuke.wave(); break;
            case 1: action= myDuke.think(); break;
            case 2: action= myDuke.write(); break;
        }
        repaint();
    }
}
```

Now **Run** it. Notice that now we have a drop-down menu option rather than the scroll choice box we had before.

The code in the **itemSelectable()** method implemented the **ItemListener** interface and made the code a little cleaner. But really, there aren't a whole lot of changes happening here. Only about 4 lines needed to be edited to change the GUI component into a Choice list. You can really see it here---modularity and reusable code mean less work for us!

Modifiers

Access Modifiers--Permissions

Let's add some capabilities to our Duke and give him some angst. Like the rest of us, Dukes can get quite angry having to do all that thinking, writing, and waving constantly. Since we are changing a characteristic of Dukes, the Class that is changed is, of course, Dukes.

We're going to have our Duke get angry in three ways: **randomly**, via the **Applets Constructor**, or by **user interaction**. Lots of things can make Duke angry, it seems!

If you want to allow characteristics of an instance to be specified at the time of instantiation, you should provide code in that class for a **Constructor** of the class, with proper parameters that specify desired values for the variables.

Edit the Dukes.java file as we have below:

CODE TO EDIT: Dukes.java

```
import java.awt.Color;

public class Dukes {

    private Color noseColor = Color.red; // default Dukes have red noses
    private boolean angry = false; // default Dukes aren't disgruntled
    private String action = "../..images/duke/dukeWave.gif";
    private String whatDoing = "Give me something to do";
    private String message= "";
    private String angryMessage= "";

    public Dukes() {
        // give Duke instance random values for traits

        int rint = (int)(Math.random() * 3); // randomly generates a 0, 1, or 2
        if (rint == 0)
        {
            noseColor = Color.blue; // more often red by default
            action = "../..images/duke/dukeWave2.gif";
            message = "What's up with the blue nose!";
        }

        // randomly decide if Duke is angry
        rint = (int)(Math.random() * 3);
        if (rint == 1)
        {
            angry = true;
            angryMessage = "I QUIT!!";
            Dukes myDuke = new Dukes(noseColor, true);
        }
    }

    // Or, when the applet instantiates the Duke, let it say if he is
    angry--a new Constructor.

    public Dukes(Color nose, boolean isMad) {
        // give Duke instance specified values for traits that are passed from the class that instantiated

        noseColor = nose;
        angry = isMad;
    }

    // Add methods to access new variables

    public String getAngryMessage()
    {
        return angryMessage;
    }

    public void setAngryMessage(String newMessage)
    {
        angryMessage = newMessage;
    }

    public boolean isAngry()
    {
        return angry;
    }

    public void setMood()
    {
        // toggle the boolean value. If it was true it becomes false; if false
        it becomes true
        angry = !angry;
        if (angry == true)
            angryMessage= "I QUIT!!";
    }
}
```

```

        else
            angryMessage= "";
    }

    public String getAction()
    {
        return whatDoing;
    }

    public String getActionImage()
    {
        return action;
    }

    public Color getNoseColor()
    {
        return noseColor;
    }

    public String getMessage()
    {
        return message;
    }

    public String write(){
        whatDoing = "I am a writing Duke";
        if (noseColor == Color.red)
        {
            action = "../..//images/duke/penduke.gif";
            message = "";
        }

        else {
            action = "../..//images/duke/penduke2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String think(){
        whatDoing = "I am a thinking Duke";
        if (noseColor == Color.red)
        {
            action = "../..//images/duke/thinking.gif";
            message = "";
        }

        else
        {
            action = "../..//images/duke/thinking2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String wave(){
        whatDoing = "I am a waving Duke";
        if (noseColor == Color.red)
        {
            action = "../..//images/duke/dukeWave.gif";
            message = "";
        }

        else
        {
            action = "../..//images/duke/dukeWave2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

```



```
}  
}
```

Go ahead and **save** this.

Again, notice that thanks to modularity, we only worry about the specific changes we want to make, and not the other methods.

And notice we now have two Constructors, one with **no parameters**:

```
public Dukes()
```

and one with **two parameters**:

```
public Dukes(Color nose, boolean isMad)
```

We use multiple constructors because sometimes, when you instantiate a class, you know the variables that you want to have set in that instance and so pass them as parameters, and other times you simply want to use the defaults, so no parameters need to be passed. In our Constructor with no parameters, the Dukes decides randomly whether Duke is angry. If he is angry, then it calls the other constructor and sets the parameters for us (the second one being **true**). If WE want to DECIDE whether Duke is angry, we instead call the Constructor *with* parameters to specify as such (second parameter is **true** or **false**).

```
Dukes myDuke = new Dukes(noseColor, false);
```

We now are allowing someone else's applet to determine Dukes action and not necessarily deciding randomly. That's pretty cool.

WARNING

Do not use return types for Constructors--not even void!
Constructors always return an instance of the type of Object they are constructing, so they do not need a return type.

Since we are allowing Duke to have a new characteristic (angry), and he's expressing this characteristic, we added a variable so we can see what he says:

```
private String angryMessage= "";
```

Notice in the code that the **instance variables** are all **private** because we don't want them changed directly. This is an example of **encapsulation** (information hiding). (You'll see later why we want to hide variables from others and make them accessible through methods only.)

We've also added methods the user can use to change the characteristics and/or see what they are (instead of accessing them directly).

```
public String getAngryMessage()  
public void setAngryMessage(String newMessage)  
public boolean isAngry()  
public void setMood()
```

Notice that our access and change methods are **public** to allow others to use them to find and set attributes. Users of the code (access from other classes) cannot see the information unless we allow them to with public methods, and they can only change variables by using the methods we provide.

Let's use this class, and then discuss access permissions in depth.

Now let's change the user interface in DukesAppletGUI.java to take advantage of these new features in Dukes.java.

CODE TO EDIT: DukesAppletGUI for new Variable access and Constructor

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

public class DukesAppletGUI extends Applet implements ItemListener{

    Dukes myDuke;
    String action;

    public void init() {

        Choice actionList = new Choice();
        actionList.add("wave");
        actionList.add("think");
        actionList.add("write");

        actionList.addItemListener(this);
        add(actionList);

        myDuke =new Dukes();
        action = myDuke.getActionImage();

        Checkbox isAngry = new Checkbox("angry", myDuke.isAngry());
add(isAngry);
        isAngry.addItemListener(this);
    }

    public void paint(Graphics g) {

        Image actionChoice = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10,165);
        g.drawString(myDuke.getMessage(), 10,180);
        g.drawImage(actionChoice, 20, 50, Color.white, this);
g.drawString(myDuke.getAngryMessage(), 110,110);
    }

    public void itemStateChanged(ItemEvent evt){
if (evt.getItem().toString() == "angry")
myDuke.setMood();
        else
        {
            int which = ((Choice)evt.getItemSelectable()).getSelectedIndex();
            switch (which)
            {
                case 0: action= myDuke.wave(); break;
                case 1: action= myDuke.think(); break;
                case 2: action= myDuke.write(); break;
            }
    } // make sure you see the curly bracket here too!
            repaint();
        }
    }
}
```

And **Run** it.

We've added a **Checkbox** to allow us (the user) to decide whether the Duke is angry, and if he is, we let him send a clear message.

Modularity is used in this class (DukeAppletGUI) and the Dukes class, and both contain the variable name "isAngry". For the Applet, it's the name of a Checkbox instance (in the init() method); for the Dukes it's a method to see if Duke IS angry. As long as there is an instance name in front of the dot (e.g., myDuke.isAngry()), Java knows exactly where to get the right use of "isAngry". If there is not a dot with an instance name in front of it, Java knows to look in the class itself (i.e., the code that uses it).

This is a nice feature of object-oriented programming languages--you never have to worry about variables that someone else used, because the compiler will know which variable should be used by its presence in a certain class.

What Permissions Allow

Classes, instance and class variables, and instance and class methods can all have **access modifiers**. Two commonly used permissions that promote encapsulation of class information are:

- **public**
- **private**

We have already seen that variables defined in a method are *local* to the method and are not known outside of the method (by variable **scope**). Method variables are implicitly private to the method in which they are defined. In this section, we are not looking at the method variables, but access to the method itself.

First, within a Class, any instance or class variables of a class are accessible by any method of that class since they are defined at the same level or "above" the methods themselves.

Modifiers for permissions indicate who **outside the class** may access things. For example, the private variables (and methods) can only be seen (or accessed) by other methods of that same class; that's why we call them *private*. **Private** variables require public "get" and "put" methods for "outside" access and changes.

Public means accessible by any class.

The Ps for Permissions: public, private, protected, package

- **public**--any and all classes can access (as long as its package is visible (imported)).

```
public void AnyOneCanAccess() {}
```

- **private**--accessible only to those within the class they are defined. They are not available to subclasses.

```
private String CreditCardNumber;
```

- **protected**--all classes in package and subclasses of the class inside and outside package.

```
protected String FamilySecrets;
```

- "friendly"--no specific declaration, the default, also known as "package" because it allows access to any objects inside the same package. ("Package" will be covered in detail in a later course in this series.)

```
void MyPackageMethod() {}
```

Modifier	Visibility
public	All classes where package is visible
private	None (only within own class)
protected	Classes in package and subclasses inside or outside package
none (default)	Classes in same package

To do

1. Open the DukesAppletGUI.java file in the Editor.
2. Look in the `init()` method, at the line:
`Checkbox isAngry = new Checkbox("angry", myDuke.isAngry());`

Within that line is a call to the "get" method of the `Dukes` Class's `isAngry()` method through `myDuke.isAngry()`.

Tip

As a naming convention, names of methods that `return` booleans (`true` or `false`) start with `is` in front of the variable name. Accessor methods use `get` as seen in `isAngry()`. Getter methods `get` values and start with `get`, setter methods `set` values and start with `set`.

Try to `get` the value of the instance variable `angry` **without** going through the accessor method: i.e., using: `myDuke.angry`:

CODE TO EDIT IN BLUE:DukesAppletGUI.java

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

public class DukesAppletGUI extends Applet implements ItemListener{

    Dukes myDuke;
    String action;

    public void init() {

        Choice actionList = new Choice();
        actionList.add("wave");
        actionList.add("think");
        actionList.add("write");

        actionList.addItemListener(this);
        add(actionList);

        myDuke =new Dukes();
        action = myDuke.getActionImage();

        Checkbox isAngry = new Checkbox("angry", myDuke.angry);
        add(isAngry);
        isAngry.addItemListener(this);
    }

    public void paint(Graphics g)    // paint method
    {
        Image myAction = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10,165);
        g.drawString(myDuke.getMessage(), 10,180);
        g.drawImage(myAction, 20, 50, Color.white, this);

    }

    public void itemStateChanged(ItemEvent evt){
        int whichOne = ((Choice)evt.getItemSelectable()).getSelectedIndex();
        switch (whichOne)
        {
            case 0: action= myDuke.wave(); break;
            case 1: action= myDuke.think(); break;
            case 2: action= myDuke.write(); break;

        }
        repaint();
    }
}
```

Now **try** to **run** this.

You should see this error:

The field Dukes.angry is not visible

Since you are "in" the DukesAppletGUI class and not the Dukes class, you can't see Dukes' private variables or methods.

We *could* make the angry variable in the Dukes class public to fix the error:

To do

1. Open the Dukes.java class.
2. Change
`private boolean angry = false;`
to
`public boolean angry = false;`
3. Save it.
4. Go back to the DukeAppletGUI class that had the errors.

All the errors are gone now because the variable is public (it's not *illegal* to expose our private parts in Java, but it *is* rude--and it violates our data-hiding convention, too. Let's see if it mattered here (it can sometimes, as we'll see later in this lesson):

To do

1. Save both classes and Run the applet.
2. Click the angry Checkbox over and over.

Whew, everything still works fine in this example, but there are cases where it could result in **data corruption**. (We'll cover that in a later course in this series as well.)

Encapsulation and data-hiding prevent data corruption because they prevent users from changing one aspect of code without considering the consequences that change may have on other aspects.

Some of the great advantages of modularity and encapsulation are:

- Your code is safer from user corruption, because even when your classes allow changes, you get to determine what users can and cannot access.
- If your code has an error in it, you can easily trace it back to your encapsulated chunk of code. This way you only have to change it once--not throughout your program.

Be sure and put the Dukes.java and DukeAppletGUI.java back into their original form by making the **angry** variable **private** in Dukes.java, and fixing the line **Checkbox isAngry = new Checkbox("angry", myDuke.isAngry());** in DukeAppletGUI.java.

Keep going, you're doing great!

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.

Class Members, Constants and main

Static Members

In previous lessons we used methods that use *static* variables such as `Color.red`. Also called **Class Variables** (as opposed to *Instance Variables*), **static** variables don't change. We also used *static Class Methods* when we used methods like `Math.random()`.

Let's take a look at these in the API [API](#). Click on the API link provided in Eclipse, find the `java.lang` package, and scroll down to the `Math` class. Notice that the modifiers in the left column all say **static**.

Now go back to the `java.lang` package. Get the `Integer` class.

Notice that most of the Fields and Methods in that Class also have the modifier **static**.

Recall that a Class can contain only:

- **Fields**
- **Methods**

But Fields and Methods will be either:

- **Instance** members **or**
- **Class** (keyword **static**) members

Members of a class (Fields and Methods) are generally only accessible once you have instantiated the class to make an **instance**. Once you have the instance, you can use the dot operator to get the values of the fields, or to invoke the methods.

Members that have the **static** modifier are **Class variables or methods**, so they're accessible **either** through the dot operator **or** through their **Class name**. As such, it isn't necessary to create an instance to use a Class Variable or Class Method. **Class variables and methods** often belong to Classes that are commonly used for auxiliary purposes in your Classes (for example, `java.lang.Math`).

Okay, let's get crackin' and put our new knowledge to work. Create a new project for this lesson called `java1_Lesson14`, and copy and paste `Dukes.java` and `DukesAppletGUI.java` from the `java1_Lesson13` project to `java1_Lesson14` project.

We're going to change an *instance* reference to a *class* reference and notice that Eclipse will yell at us, and make some suggestions for fixing it.

Open `Dukes.java` and look for Class references:

OBSERVE: In Dukes.java, notice the use of Class variables and methods.

```
import java.awt.Color;

public class Dukes {

    private Color noseColor = Color.red; // default Duke's have red noses
    private boolean angry = false; // default Duke's aren't usually disgruntled
    private String action = "../..images/duke/dukeWave.gif";
    private String whatDoing = "Give me something to do";
    private String message= "";
    private String angryMessage= "";

    public Dukes() {
        // give Duke instance random values for traits

        int rint = (int) (Math.random() * 3); // randomly generates a 0, 1, or 2
        if (rint == 0)
        {
            noseColor = Color.blue; // more often red by default
            action = "../..images/duke/dukeWave2.gif";
            message = "What's up with the blue nose!";
        }

        // randomly decide if Duke is angry
        rint = (int) (Math.random() * 3);
        if (rint == 1)
        {
            angry = true;
            angryMessage = "I QUIT!!";
            Dukes myDuke = new Dukes(noseColor, true);
        }
    }

    // Or, when the applet instantiates the Duke, let it say if he is angry-
    // a new Constructor.

    public Dukes(Color nose, boolean isMad) {
        // give Duke instance specified values for traits which are passed from the class th
        // at instantiated

        noseColor = nose;
        angry = isMad;
    }

    // Add methods to access new variables

    public String getAngryMessage()
    {
        return angryMessage;
    }

    public void setAngryMessage(String newMessage)
    {
        angryMessage = newMessage;
    }

    public boolean isAngry()
    {
        return angry;
    }

    public void setMood()
    {
        // toggle the boolean value. If it was true it becomes false; if false it beco
        // mes true
        angry = !angry;
        if (angry == true)
            angryMessage= "I QUIT!!";
    }
}
```



```
        else
            angryMessage= "";
    }

    public String getAction()
    {
        return whatDoing;
    }

    public String getActionImage()
    {
        return action;
    }

    public Color getNoseColor()
    {
        return noseColor;
    }

    public String getMessage()
    {
        return message;
    }

    public String write(){
        whatDoing = "I am a writing Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/penduke.gif";
            message = "";
        }

        else {
            action = "../..images/duke/penduke2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String think(){
        whatDoing = "I am a thinking Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/thinking.gif";
            message = "";
        }

        else
        {
            action = "../..images/duke/thinking2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String wave(){
        whatDoing = "I am a waving Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/dukeWave.gif";
            message = "";
        }

        else
        {
            action = "../..images/duke/dukeWave2.gif";
            message = "My nose feels funny";
        }
        return action;
    }
}
```

```
}  
}
```

Tip

According to convention, Classes start with capital letters and Instances start with lower-case letters. This allows us to identify whether methods and variables are Class members (static).

Now, let's experiment on `DukesAppletGUI.java`. Try using the Class name instead of the instance name (`Dukes` instead of `myDuke`) in a method call:

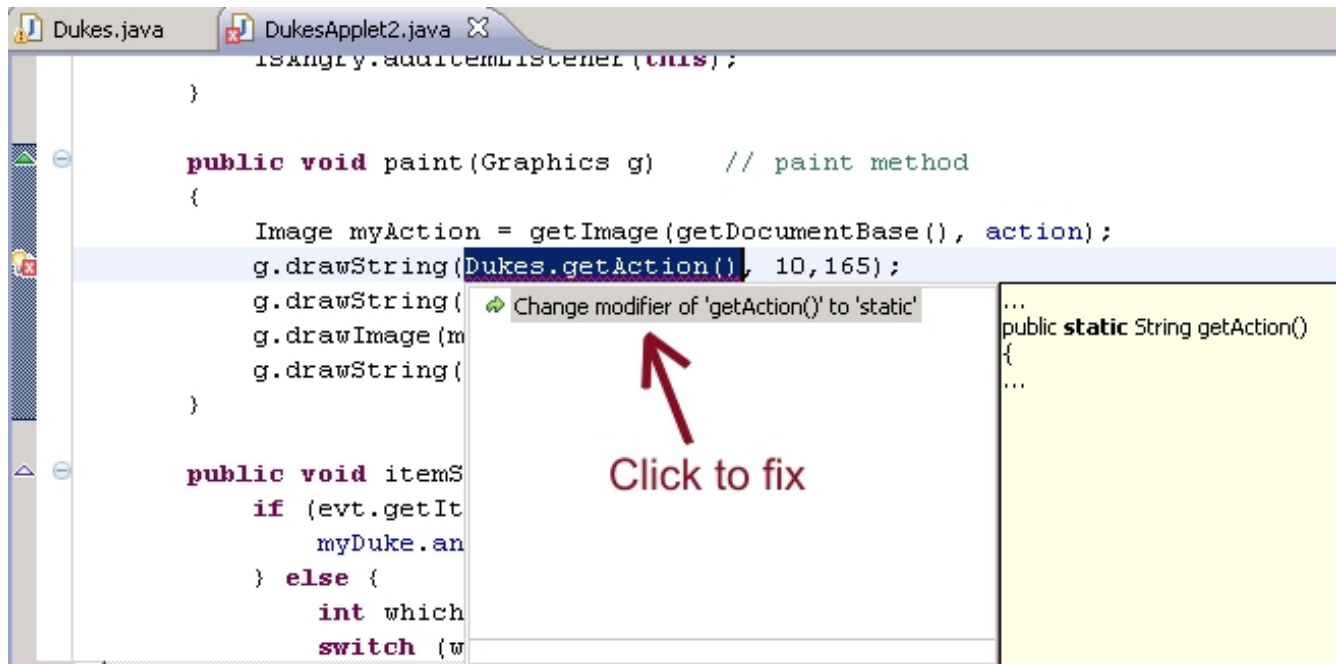
CODE TO TYPE: In the code below, change `myDuke` to `Dukes`.

```
import java.applet.Applet;  
import java.awt.*;  
import java.awt.event.*;  
  
public class DukesAppletGUI extends Applet implements ItemListener{  
  
    Dukes myDuke;  
    String action;  
  
    public void init() {  
  
        Choice actionList = new Choice();  
        actionList.add("wave");  
        actionList.add("think");  
        actionList.add("write");  
  
        actionList.addItemListener(this);  
        add(actionList);  
  
        myDuke = new Dukes();  
        action = myDuke.getActionImage();  
  
        Checkbox isAngry = new Checkbox("angry", myDuke.isAngry());  
        add(isAngry);  
        isAngry.addItemListener(this);  
    }  
  
    public void paint(Graphics g) {  
  
        Image actionChoice = getImage(getDocumentBase(), action);  
        g.drawString(Dukes.getAction(), 10, 165);  
        g.drawString(myDuke.getMessage(), 10, 180);  
        g.drawImage(actionChoice, 20, 50, Color.white, this);  
        g.drawString(myDuke.getAngryMessage(), 110, 110);  
    }  
  
    public void itemStateChanged(ItemEvent evt){  
        if (evt.getItem().toString() == "angry")  
            myDuke.setMood();  
        else  
        {  
            int which = ((Choice)evt.getItemSelectable()).getSelectedIndex();  
            switch (which)  
            {  
                case 0: action= myDuke.wave(); break;  
                case 1: action= myDuke.think(); break;  
                case 2: action= myDuke.write(); break;  
            }  
        } // make sure you see the curly bracket here too!  
        repaint();  
    }  
}
```

You'll get this error:

Cannot make static reference to non-static method `getAction()` from type `Dukes`

The problem here is that we've made a reference to a Class member that hasn't been declared *static*. Alright then, let's follow Eclipse's advice and make the `getAction()` method static just to see what happens. In fact, let Eclipse do it for you:



Or you can go to `Dukes.java` and add the **static** modifier to the `getAction()` method yourself:

CODE TO TYPE: Make getAction() static in Dukes.java.

```
import java.awt.Color;

public class Dukes {

    private Color noseColor = Color.red; // default Duke's have red noses
    private boolean angry = false; // default Duke's aren't usually disgruntled
    private String action = "../..images/duke/dukeWave.gif";
    private String whatDoing = "Give me something to do";
    private String message= "";
    private String angryMessage= "";

    public Dukes() {
        // give Duke instance random values for traits

        int rint = (int)(Math.random() * 3); // randomly generates a 0, 1, or 2
        if (rint == 0)
        {
            noseColor = Color.blue; // more often red by default
            action = "../..images/duke/dukeWave2.gif";
            message = "What's up with the blue nose!";
        }

        // randomly decide if Duke is angry
        rint = (int)(Math.random() * 3);
        if (rint == 1)
        {
            angry = true;
            angryMessage = "I QUIT!!";
            Dukes myDuke = new Dukes(noseColor, true);
        }
    }

    // Or, when the applet instantiates the Duke, let it say if he is angry-
    // a new Constructor.

    public Dukes(Color nose, boolean isMad) {
        // give Duke instance specified values for traits which are passed from the class th
        // at instantiated

        noseColor = nose;
        angry = isMad;
    }

    // Add methods to access new variables

    public String getAngryMessage()
    {
        return angryMessage;
    }

    public void setAngryMessage(String newMessage)
    {
        angryMessage = newMessage;
    }

    public boolean isAngry()
    {
        return angry;
    }

    public void setMood()
    {
        // toggle the boolean value. If it was true it becomes false; if false it beco
        // mes true
        angry = !angry;
        if (angry == true)
            angryMessage= "I QUIT!!";
    }
}
```

```
        else
            angryMessage= "";
    }

    public static String getAction()
    {
        return whatDoing;
    }

    public String getActionImage()
    {
        return action;
    }

    public Color getNoseColor()
    {
        return noseColor;
    }

    public String getMessage()
    {
        return message;
    }

    public String write(){
        whatDoing = "I am a writing Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/penduke.gif";
            message = "";
        }

        else {
            action = "../..images/duke/penduke2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String think(){
        whatDoing = "I am a thinking Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/thinking.gif";
            message = "";
        }

        else
        {
            action = "../..images/duke/thinking2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String wave(){
        whatDoing = "I am a waving Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/dukeWave.gif";
            message = "";
        }

        else
        {
            action = "../..images/duke/dukeWave2.gif";
            message = "My nose feels funny";
        }
        return action;
    }
}
```

```
}  
}
```

Now we get a different error:

Cannot make static reference to non-static field whatDoing

We are going to make **whatDoing** static to see if the reference to Dukes can then be made. We can do that either by using the Eclipse trick or by typing it for ourselves:

CODE TO TYPE: In Dukes.java, make whatDoing static.

```
import java.awt.Color;

public class Dukes {

    private Color noseColor = Color.red; // default Duke's have red noses
    private boolean angry = false; // default Duke's aren't usually disgruntled
    private String action = "../..images/duke/dukeWave.gif";
    private static String whatDoing = "Give me something to do";
    private String message= "";
    private String angryMessage= "";

    public Dukes() {
        // give Duke instance random values for traits

        int rint = (int)(Math.random() * 3); // randomly generates a 0, 1, or 2
        if (rint == 0)
        {
            noseColor = Color.blue; // more often red by default
            action = "../..images/duke/dukeWave2.gif";
            message = "What's up with the blue nose!";
        }

        // randomly decide if Duke is angry
        rint = (int)(Math.random() * 3);
        if (rint == 1)
        {
            angry = true;
            angryMessage = "I QUIT!!";
            Dukes myDuke = new Dukes(noseColor, true);
        }
    }

    // Or, when the applet instantiates the Duke, let it say if he is angry-
    // a new Constructor.

    public Dukes(Color nose, boolean isMad) {
        // give Duke instance specified values for traits which are passed from the class th
        // at instantiated

        noseColor = nose;
        angry = isMad;
    }

    // Add methods to access new variables

    public String getAngryMessage()
    {
        return angryMessage;
    }

    public void setAngryMessage(String newMessage)
    {
        angryMessage = newMessage;
    }

    public boolean isAngry()
    {
        return angry;
    }

    public void setMood()
    {
        // toggle the boolean value. If it was true it becomes false; if false it beco
        // mes true
        angry = !angry;
        if (angry == true)
            angryMessage= "I QUIT!!";
    }
}
```

```
        else
            angryMessage= "";
    }

    public static String getAction()
    {
        return whatDoing;
    }

    public String getActionImage()
    {
        return action;
    }

    public Color getNoseColor()
    {
        return noseColor;
    }

    public String getMessage()
    {
        return message;
    }

    public String write(){
        whatDoing = "I am a writing Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/penduke.gif";
            message = "";
        }

        else {
            action = "../..images/duke/penduke2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String think(){
        whatDoing = "I am a thinking Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/thinking.gif";
            message = "";
        }

        else
        {
            action = "../..images/duke/thinking2.gif";
            message = "My nose feels funny";
        }
        return action;
    }

    public String wave(){
        whatDoing = "I am a waving Duke";
        if (noseColor == Color.red)
        {
            action = "../..images/duke/dukeWave.gif";
            message = "";
        }

        else
        {
            action = "../..images/duke/dukeWave2.gif";
            message = "My nose feels funny";
        }
        return action;
    }
}
```



```
}  
}
```

Now go back to **DukesAppletGUI.java** and **run** it.

Tip Eclipse uses italics to let you know when a method or field is static, like in "Color.red".

You shouldn't have any errors. **However, we have completely changed our Class!** Eclipse's suggestions have led us astray from the original intent of the Dukes Class. So we can't always take Eclipse's advice. There is a reason that that **whatDoing** WASN'T *static* in the first place -- because we wanted **whatDoing** to be an instance variable i.e. we wanted it to be different for each instance. Now, when we change **whatDoing**, the change is applied to **all instances** of that Class. So now all of instances of Dukes will be doing the SAME thing. That's not what we want. Let's go ahead and change it back.

To do

1. Change the whatDoing back to an Instance Variable by removing the static keyword.
2. Remove the static keyword from getAction()
3. Change the Dukes.getAction() reference in the applet back to myDukes.getAction()

There, all is well again! The purpose of this exercise was to give you a feel for static variables and references. In the next section we'll make some class variables that should be class variables.

Static: Making Your Own

For normal, everyday Classes, Java has both Class Variables and Instance Variables.

Defining variables within a Class:

- **If** a value changes for each instance of a Class, **then** it should be an instance variable.
- **If** a value remains the same for every instance of a Class, **then** it should be a class variable.

Here are a couple of examples to help illustrate this concept:

An employee's **salary** (instance variable), compared to the **topSalary** of all employees (class variable).

A human's **colorOfEyes** (instance variable), compared to the **numberOfEyes** of all humans (class variable).

You'll understand the logic behind both kinds of variables even better once you see how they're implemented in Java:

- Instance variables each occupy their own space in memory. Changes to a value for one instance will have no effect on other instances.
- Class variables all occupy the same space in memory. If the value ever changes, it changes for **every** instance in the Class, because they share the same memory address location.

With this in mind, let's think about access. Instances have information about the class, but classes do not have individual instance access. Instances should be able to access Class information, but unless you have the reference or handle to the instance, you cannot access instance information.

Let's look at an example of access to class and instance variables and methods:

Make a new Class called **Employee** (that doesn't extend Applet).

CODE TO TYPE: Employee.java

```
class Employee {  
  
    private static int topSalary = 195000;  
  
    public static void setTopSalary (int s) {  
        if (s > topSalary)  
            topSalary = s;  
    }  
}
```

Notice that we have a **class variable** (denoted by *static*) that is changed through a **class method** (also denoted by *static*).

Now suppose another Class wants to create **Employees**. To make testing easier, we'll add a **main()** method to our Employee class. The **main()** method is used to create an application rather than an Applet. In it, we will invoke the method from the Class (**Employee**) as well as invoke it from an instance (**e2**). We'll explain more about this later in the lesson.

Edit the **Employee** class as shown:

CODE TO TYPE: Add a main method to the Employee Class to test static members.

```
class Employee {  
  
    private static int topSalary = 195000;  
  
    public static void setTopSalary (int s) {  
        if (s > topSalary)  
            topSalary = s;  
    }  
  
    public static void main(String[] args){  
        Employee e1, e2;  
        e1 = new Employee();  
        e2 = new Employee();  
  
        Employee.setTopSalary(199000);  
        // calling by class; can we do this?  
        e2.setTopSalary(199001);  
        // calling by instance; can we do this?  
    }  
}
```

Notice that we get warnings, but no errors. Class variables and methods can be **accessed** from either individual instances or the Class. This makes sense; if a member of the Employee class gets a salary increase, that Employee's new salary may change the existing top salary. Or, the top salary can be changed by the "boss", the Class as a whole.

However, remember that if the **Class variable** is changed for (or by) one instance, it changes for all. Why?

The **class variable** is located at the same place in the computer memory for everyone. Changes made by one affect all.

If a variable is an instance variable, there are copies in each object; if a variable is a class variable, there is only one copy, which is shared.

Note

If you declare a method static, that method can only access other variables of the Class that are declared static as well. Class methods cannot change instance variables.

Let's add an instance variable and a class method to our Employee class.

CODE TO TYPE:

```
class Employee {  
  
    private static int topSalary = 195000;  
    int hoursPerWeek;  
  
    public static void setTopSalary (int s) {  
        if (s > topSalary)  
            topSalary = s;    // will work    (CM, CV)  
    }  
  
    public static void addMoreHours () {  
        hoursPerWeek++; // won't work (CM, IV)  
    }  
  
    public static void main(String[] args){  
        Employee e1, e2;  
        e1 = new Employee();  
        e2 = new Employee();  
  
        Employee.setTopSalary(199000);  
        // calling by class; can we do this?  
        e2.setTopSalary(199001);  
        // calling by instance; can we do this?  
    }  
}
```

There's that error again.

Cannot make a static reference to the non-static field hoursPerWeek

Okay, now remove the **static** keyword from the **addMoreHours()** method, so it looks like this:

```
public void addMoreHours () {
```

Java is happy now.



Now, add two lines of code to the main method as shown below:

CODE TO TYPE:

```
class Employee {

    private static int topSalary = 195000;
    int hoursPerWeek;

    public static void setTopSalary (int s) {
        if (s > topSalary)
            topSalary = s;    // will work    (CM, CV)
    }

    public static void addMoreHours () {
        hoursPerWeek++; // won't work (CM, IV)
    }

    public static void main(String[] args){
        Employee e1, e2;
        e1 = new Employee();
        e2 = new Employee();

        Employee.setTopSalary(199000);
        // calling by class; can we do this?
        e2.setTopSalary(199001);
        // calling by instance; can we do this?
        e1.hoursPerWeek = 40;
        Employee.hoursPerWeek = 45;
    }
}
```

Error again! **Cannot make a static reference to the non-static field Employee.hoursPerWeek**

Why? Consider what the expression **Employee.hoursPerWeek** is doing: it's trying to access the non-static field of **hoursPerWeek** through the **Employee** Class name, a **static** reference. Instance variables belong to individual instances. Classes cannot access them--it does not make sense. For instance, the **Class** of Humans cannot access the instance variable **colorOfEyes** of a specific Human, because the class name "**Human**" does not specify **which** one of its instances it's trying to access.

Consider the expression **e1.hoursPerWeek**: It's trying to access the non-static (and therefore instance) field of **hoursPerWeek**, through the **instance e1**. There's no problem there. A specific employee **would** work a specific number of hours in a week.

Now, change the line **Employee.hoursPerWeek = 45;** to **e2.hoursPerWeek = 45;** That should work.

static and main

The example above used something new, the **main()** method.

The **main()** method is used for all Java programs that are applications and **not** Applets. Specifically, if you want a **program** to **Run**, then you need either an applet or a Class that has a **main()** method to get the application started.

The **main()** method is *always* **static**. That's why Java can get into the Class where the method is located to start the code before you ever have an instance of the class.

Start a new Class called **ExampleWithMain** (not an Applet).

CODE TO TYPE: ExampleWithMain

```
public class ExampleWithMain
{
    private int testInstanceVariable = 42;

    public static void main(String[] args)
    {
        System.out.println("The value of the instance variable is " + this.testInstanceVariable);
    }
}
```

Darn, another error! **Cannot use this in a static context**

Let's remove **this** from the expression **this.testInstanceVariable** and make it:

```
System.out.println("The value of the instance variable is " + testInstanceVariable);
```

Grrrr. Another error. **Cannot make a static reference to a non-static field testInstanceVariable**

This error occurs because **main()** is a static method, and **testInstanceVariable** is an instance variable.

These specific problems illustrate some important stuff about the **main()** method. Including **main()** **static**, allows Java to get started, but we have not instantiated the class yet, so we do not have an instance.

Edit your code as follows:

CODE TO TYPE: ExampleWithMain

```
public class ExampleWithMain
{
    private int testInstanceVariable = 42;

    public static void main(String[] args)
    {
        ExampleWithMain myExample = new ExampleWithMain();
        System.out.println("The value of the instance variable is " + myExample.testInstanceVariable);
    }
}
```

The **main()** method is not really associated with a **field**'s access, and it's not an action for the **class**. A **main** method should only **instantiate** and **start** the top-level Class. **And**, that's good programming practice.

Constructors

Instantiation

We've been using the terms **instantiate** and **initialize** often. So what is the difference?

Let's consider our Dukes example again. Instantiating *makes* an instance of the Class. We *made* **Dukes** instances in the DukeAppletGUI when we wrote **myDuke = new Dukes();**. In the process of being **instantiated**, our **myDuke** was **initialized**. Initializing will set all the variables to the values for this instance. This happens inside the Constructor (or in an Applet, in its **init()** method, and also through defaults. And instances inherit variables from their ancestors.

The sequence to allocate memory and to default initialize is:

1. superclass initialization
2. instance variable initialization
3. constructor initialization

The order makes sense because the specific subclasses lower in a hierarchy **override** their parent and thus

their values (2) will "write over" the earlier default values set in (1). Then, since the constructor may be passed specific parameters, they should "write over" any defaults that the particular Class initially got from (2).

Tip When a Class creates an instance of itself (or instantiates), it initializes its values.

Constants use the final modifier

Constants in a programming language are variables that the programmer wants to be accessible to everyone and that remain the same for all time. The modifier keyword to make a variable remain the same is **final**.

API. Go to the **java.lang** package. Scroll again to the **Math** class. Now scroll down to the **Field Summary**. Finally, click on either of the **Fields (E or PI)**.

You see these modifiers:

- **public static final double E**
- **public static final double PI**

Good. We know what all of these modifiers mean now, but the combination of these **public static final** makes PI a **double** variable that is a **constant**. Notice also, under the description in the API, it says **Constant Field Values**. Click on it. It takes you to all of the Field constants that Java has defined. Because constants always have the static modifier, they can be accessed through the Class name `java.lang.Math.PI` or for `java.lang` classes, simply through the class and the variable **Math.PI**.

Note The **final** modifier indicates that once a field receives a value, it can never be changed. Using commonly accepted naming conventions, fields marked as **final** should be **UPPER_CASE** with words separated by the **underscore** character.

Classes, methods, fields, or variables may each have this modifier and it means essentially the same thing--that it cannot change--though it might be said a bit differently:

Modifier	Used on	Meaning
final	class method field variable	Cannot be subclassed Cannot be overridden and dynamically looked up Cannot change its value. static final fields are compile-time constants. Cannot change its value.

Template and Summary

There are additional modifiers for Classes, Methods, and Fields, but they are beyond the scope of this course.

For more information on modifiers, check out: [Java Modifiers](#)

The typical order for modifiers is (<access> choices specified above and the [] indicating 0 or 1 time)
<access> [static] [final] ...

P>

Below is an example template for Class Structure (modifiers may vary for your use):

OBSERVE:

```
import packageName.ClassName.java

import packageName.*

public class ClassName extends super {

    // Instance and Class Variable declarations: Declare and possibly set modifiers and values for variables

    private int instanceVariable;
    private static int classVariable;

    // Define Constructors (if none - looks at supers. No return type)

    public ClassName (formal parameters) {

        code

    }

    // Define methods with no returned values

    public void methodName (formal parameters) {

        code

    }

    // Define methods with return values

    public returnType methodName (formal parameters) {

        code
        return ... // returnType of value returned must match method signature

    }

}
```

Here's one more link to Java Modifiers from the online text of [tutorialspoint](#). Most of the less commonly used Java modifiers will be explored in future courses.

See you in the next and final 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.

All Together Now

Interaction and Playing Around

You've done a great job building some useful tools during the course so far; now let's put them all together. Let's give our Duke some love. It'll make him feel better and it'll give us a chance to practice using classes, objects, and instantiation. Later, we'll make real applications that do more useful things, but for now, let's play!

In this section, we'll:

- Make another Class and practice making graphical images like circles, hearts, up arrows, and other cool stuff.
- Give the **Dukes** class the ability to show these graphics **If** the specific instance of a Duke is "angry".
- Add a GUI component to our **Applet** Class so the user can specify whether Duke is in a good mood or if we should show him some love by giving him a Pin.

Putting It All Together

The **Graphics** Class allows us to draw basic shapes. Let's use these capabilities and make a separate Class that can draw figures.

Below is a class named `MoreImages` that adds some shapes to Java's **Graphics** Class. We'll use this Class to allow our Dukes to make various graphics.

Let's make a new project for lesson 15 called `java1_Lesson15`, and a new Class called **PinImages**.

After we have **PinImages** bug free, we'll make a simple Applet and demo some of the figures in the `paint` method of an Applet to see how they look. We'll also get to see the potential of our `ButtonImages`. Ultimately, you'll choose a method to make a Button, so pay attention as you type this in and notice the kinds of shapes it allows.

CODE TO TYPE: PinImages

```
// This class will be used by DukesMood which will inherit from Dukes
// It will give a Duke who can make Pins using hearts, crowns, stars, triangles,
// etc.

import java.awt.*;

public class PinImages
{
    public void drawHeart(Graphics g, Color col, int x, int y, int width){
        // I "heart" Duke
        g.setColor(col);
        g.fillArc(x,y,width/2,width/2,0,180);
        g.fillArc(x+ width/2,y,width/2,width/2,0,180);
        int [] xTriangle = {x, x+width, x+width/2};
        int [] yTriangle = {y+width/4, y+width/4, y+width};
        g.fillPolygon(xTriangle, yTriangle, 3);
    }

    public void fillTriangle(Graphics g, int bottomX, int bottomY, int base, int
height){
        // isosceles base at bottom
        g.drawLine(bottomX, bottomY, bottomX+base, bottomY);
        g.drawLine(bottomX+base, bottomY, bottomX+base/2, bottomY-height);
        g.drawLine(bottomX+base/2, bottomY-height, bottomX, bottomY);
        int [] xTriangle = {bottomX, bottomX+base, bottomX+base/2};
        int [] yTriangle = {bottomY, bottomY, bottomY-height};
        g.fillPolygon(xTriangle, yTriangle, 3);
    }

    public void upArrow(Graphics g, Color col, int x, int y, int arrowbase){
        // Duke is movin' on UP
        g.setColor(col);
        fillTriangle(g, x, y, arrowbase, arrowbase/2);
        g.fillRect(x+ 3*arrowbase/8, y, arrowbase/4, arrowbase);
    }
}
```

Save your code for the PinImage Class.

Now we need an Applet so we can look at our artwork and demonstrate some of the things that the PinImage class can draw. Make a new class called **ImageMaker**, so we can see these PinImages displayed.

CODE TO TYPE: Demo

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

public class ImageMaker extends Applet
{
    PinImages demo;

    public void init() {
        setBackground(Color.black);
        demo = new PinImages();
    }

    public void paint(Graphics g)
    {
        demo.drawHeart(g, Color.pink, 10,10, 25);
        demo.upArrow(g, Color.orange, 10, 70, 30);

        // an example Pin using a PinImage shape
        g.setColor(Color.red);
        g.fillOval(100,100, 80,80);
        g.setColor(Color.white);
        g.drawString("I",135,120);
        demo.drawHeart(g, Color.pink, 125,125, 25);
        g.setColor(Color.white);
        g.drawString("Duke!",125,170);
    }
}
```

Now **Run** it.

The first couple of expressions in the **paint()** method use the instance **demo**, but the later ones use **g**.

That's because the **drawHeart()**, and **upArrow()** methods are defined in the class **PinImages** which has an instance of **demo**.

The **setColor()**, **fillOval()**, and other methods are defined in the Class **Graphics** which has an instance of **g**.

So what else is interesting here?

We imported `java.awt.*` so that we could see the `Graphics` class. We don't need to import anything for the `PinImages` class because Classes located in the same directory are in the same "default" package.

Using Inheritance on our Own

Because we are going to add more capabilities to our Dukes rather than editing to add them into the old Dukes class, let's take advantage of the concept of inheritance. Dukes that can make Pins will be called **DukesPins** and will be a **subclass** of **Dukes**.

Create another new class in `java1_Lesson15` called **DukesPin**.

It will **inherit** from **Dukes**, so let's just copy our Dukes class into `java1_Lesson15`:

Copy **Dukes.java** from `java1_Lesson14` and paste it into `java1_Lesson15/src`.

We'll be adding Dukes, as well as things that relate to showing Pins, to our new Applet `DukesPin`. But we're only giving Pins to angry Dukes; after all, they're the ones who really need love. So, if someone changes whether Duke is angry, we have to change his ability to get Pins. We'll **override** **Duke's** **setMood()** method.

Also notice how our Constructor calls **super()**. **super()** is a call to the parent's constructor.

We actually don't need to do this, because Java calls a parent's constructor by default, but it's here as an example, in case you wanted to use the default constructor **and** add more to it. Anyway, Java will use **DukesPin's** methods first, and then will inherit all of the other methods and variables from the **Dukes** class.

CODE TO TYPE: Type this into the DukesPin class

```
import java.awt.Color;

public class DukesPin extends Dukes {

    private boolean showingPin;

    public DukesPin() {
        super();
        // you could add more here and Java will do the parent's first and the
n come back for more
    }

    public DukesPin(Color nose, boolean love) {
        super(nose, love);
    }

    public boolean isShowingPin() {
        return showingPin;
    }

    public void switchShowingPin() {
        showingPin = !showingPin;
        if (showingPin && !angry)
        {
            angryMessage= "I don't get a Pin, I'm not angry";
            showingPin = !showingPin; // don't let them show Pins since not angry
        }
    }

    public void setMood() {
        super.setMood(); // let the parent do the work first, then do what we
need in addition
        if (angry == false) showingPin = false;
    }
}
```

Hey, what is going on here? Why all the **red** in our new class?

All the errors are a result of the access to the variables **angry** and **angryMessage**. We inherited these variables from **Dukes.java**. Recall that we gave these variables the **private** permissions. And who can see variables if the access is **private**? Only the instances of the Class **itself**. Our new Class *inherited* from **Dukes**, so **Dukes** is its parent/super. Our new Class is not the same Class as **Dukes**.

Can you fix this?

Since you are the author of the code for the Class **Dukes**, you can change the permission to **protected**. Remember that **protected** gives access to Classes in the package **and subclasses** inside or outside the package (for now, just think of a package as all the Classes you're currently using).

Note If you click once on the red "x", Eclipse will show the error and suggest ways to fix it.

In the **Dukes** Class, change the access permissions of these two variables to:

```
protected boolean angry = false;
```

and

```
protected String angryMessage= "";
```

Now go back to your DukesPin class and all should be well.

That's nice, but if the Class you're using is from Java's API, you can't just go in and change that code.

But *methods* are often present within Classes in order to allow access to the variables. Does the **Dukes**

Class have such methods? Why yes, it does. Let's see if we can fix our new Class by using accessor methods:

Change the permissions in the **Dukes** class back so they are all **private** again. Save it so your errors come back in **DukesPin**.

Look in the **Dukes** Class and find the accessors and mutators for the needed variables.

Here are the accessor "get" methods (notice that the mutator "sets" values):

OBSERVE:

```
public String getAngryMessage()
{
    return angryMessage;
}

public boolean isAngry()
{
    return angry;
}
```

Sweet. Let's use them:

CODE TO TYPE: Type this into the DukesPin class

```
import java.awt.Color;

public class DukesPin extends Dukes {

    private boolean showingPin;

    public DukesPin() {
        super();
        // you could add more here and Java will do the parent's first and the
n come back for more
    }

    public DukesPin(Color nose, boolean love) {
        super(nose, love);
    }

    public boolean isShowingPin() {
        return showingPin;
    }

    public void switchShowingPin() {
        showingPin = !showingPin;
        if (showingPin && !isAngry())
        {
            setAngryMessage("I don't get a Pin, I'm not angry");
            showingPin = !showingPin; // don't let them show Pins when not angry
        }
    }

    public void setMood() {
        super.setMood(); // let the parent do the work first, then do what we
need in addition
        if (isAngry() == false) showingPin = false;
    }
}
```

Excellent! All is well. Let's see what this guy looks like now.

Make a new Class in the java1_Lesson15 Project called **DukesPinApplet**.

```

import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class DukesPinApplet extends Applet implements ItemListener{

    DukesPin myDuke;
    String action;
    Checkbox showPin;
    Graphics g;

    public void init() {

        Choice actionList = new Choice();
        actionList.add("wave");
        actionList.add("think");
        actionList.add("write");

        actionList.addItemListener(this);
        add(actionList);

        myDuke =new DukesPin();
        action = myDuke.getActionImage();

        Checkbox changeMood = new Checkbox("Angry", myDuke.isAngry());
        add(changeMood);
        changeMood.addItemListener(this);

        showPin = new Checkbox("ShowPin");
        add(showPin);
        showPin.addItemListener(this);
    }

    public void paint(Graphics g) {
        this.g = g;
        Image actionChoice = getImage(getDocumentBase(), action);
        g.drawString(myDuke.getAction(), 10,165);
        g.drawString(myDuke.getMessage(), 10,180);
        g.drawImage(actionChoice, 20, 50, Color.white, this);

        g.drawString(myDuke.getAngryMessage(), 110,140);
        if (myDuke.isShowingPin())
            makePin();
        else clearPin();
    }

    public void itemStateChanged(ItemEvent evt){

        if (evt.getItem().toString() == "Angry")
        {
            myDuke.setMood();
            if (!myDuke.isAngry())
                showPin.setState(false);
        }
        else if (evt.getItem().toString() == "ShowPin")
        {
            myDuke.switchShowingPin();
            if (showPin.getState() && !myDuke.isAngry())
                showPin.setState(false);
        }
        else
        {
            int which = ((Choice)evt.getItemSelectable()).getSelectedIndex();
            switch (which)
            {
                case 0: action= myDuke.wave(); break;
                case 1: action= myDuke.think(); break;
            }
        }
    }
}

```

```

        case 2: action= myDuke.write(); break;
    }
}
repaint();
}

public void makePin()
{
    PinImages images =new PinImages();
    // make Pin
    g.setColor(Color.red);
    g.fillOval(120,50, 80,80);
    // put something in Pin
    g.setColor(Color.white);
    g.drawString("I",155,70);
    images.drawHeart(g, Color.pink, 145,75, 25);
    g.setColor(Color.white);
    g.drawString("Duke!",145,120);
}

public void clearPin()
{
    g.setColor(Color.white);
    g.fillOval(120,50, 80,80);
}
}

```



Save and Run it.

Click the different checkbox choices to see how the GUI reacts.

API Go to the API **java.awt** package. Notice that the Classes **Choice** and **Checkbox** both **implement ItemSelectable**. That's why you can then implement the **ItemListener**. However, since all three of the components the user can choose from cause **ItemEvents**, you need to tell Java **which** action to take depending on which of the **Checkboxes** or **Choice** the user clicked. You do that in the **ItemListener's itemStateChanged** method.

Great job! Now that you understand the concepts behind objects, you'll really be able to take off!

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.