Java Programming 2: The Java Programming Language

Java Programming Constructs

Welcome to OST's Java 2 course!

Course Objectives

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

- demonstrate knowledge of basic algorithm constructs in Java.
- develop and compile Java applications that utilize primitive data types, statements, expressions, and GUIs.
- output and manipulate strings, fonts, and numbers.
- apply decision logic and operational precedence to Java code.
- implement multidimensional arrays, loops, and branching statements.
- trace code for better software quality.

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

Lesson Objectives

- Identify the control constructs used in structured programming.
- use the main method to start a Java application.
- access Java documentation.
- create a new Java project with a class that prints specific text.

In this course, you'll learn more in-depth concepts and syntax of the Java Programming language. Throughout this course, you'll learn by building examples using the Eclipse Java Development Environment which is supplied as a Learning Sandbox. Completion of this course gives you a basic understanding of Object Oriented techniques in Java as well as using the Eclipse IDE.

From beginning to end, you will learn by doing your own Java projects, within our Eclipse Learning Sandbox we affectionately call "Ellipse". These projects will add to your portfolio and provide needed experience. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

Learning with O'Reilly School of Technology Courses

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

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

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

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

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

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

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

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you

blew through all of the coursework too quickly.

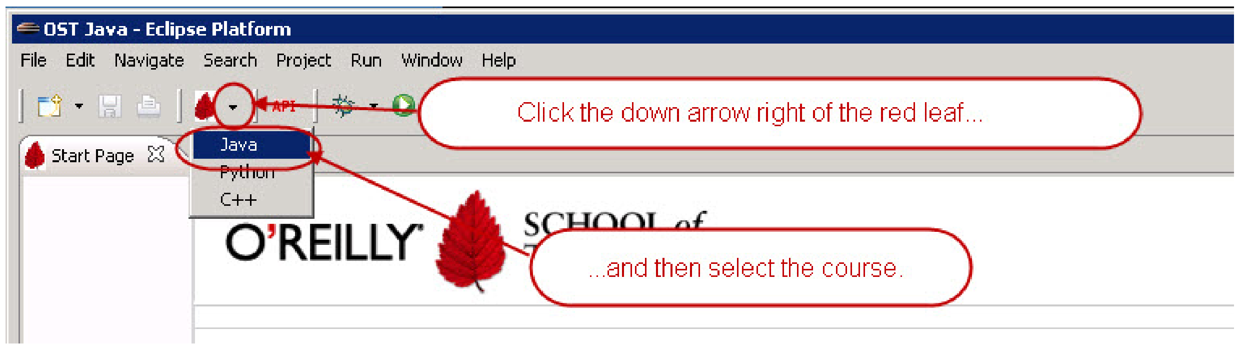
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Before You Start: Working Sets

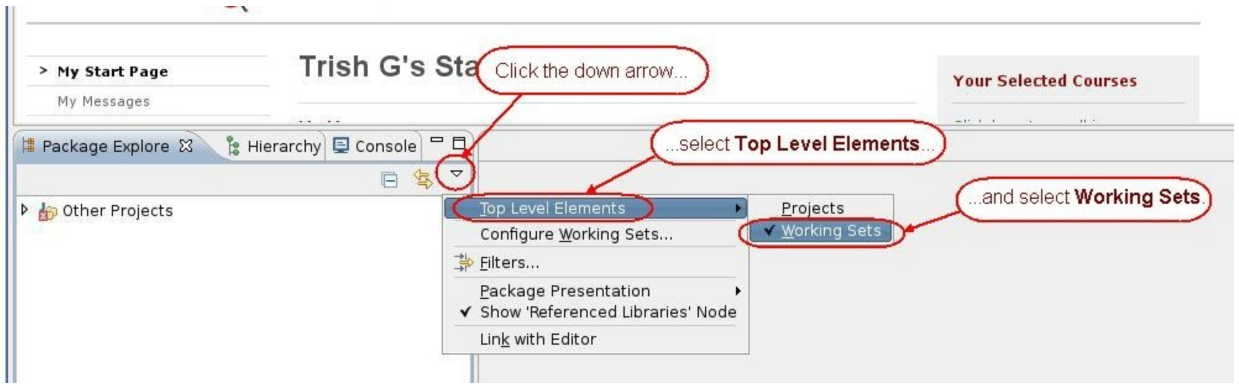
We organize our projects in *Working Sets*. To create the working sets for this course, use the down arrow beside the **Red Leaf** icon on the toolbar. You will use the **Java2_Lessons** working set for the lessons, and the **Java2_Homework** working set for the exercises after each lesson.

Selecting a Working Set

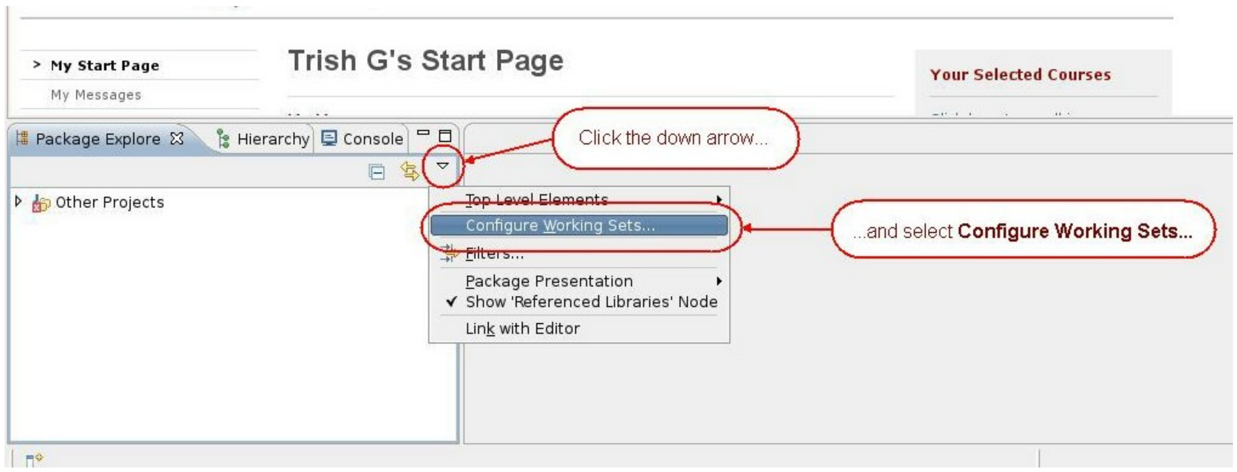
When you start working on a course, or if you just need to reset your perspective, you need to select the appropriate working set. Click the small down arrow beside the Red Leaf icon on the tool bar to see a list of course groups. For this course, select the Jave course group.


OST Java - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Click the down arrow right of the red leaf...
Start Page
Java
Python
C++
O'REILLY
SCHOOL of
...and then select the course.

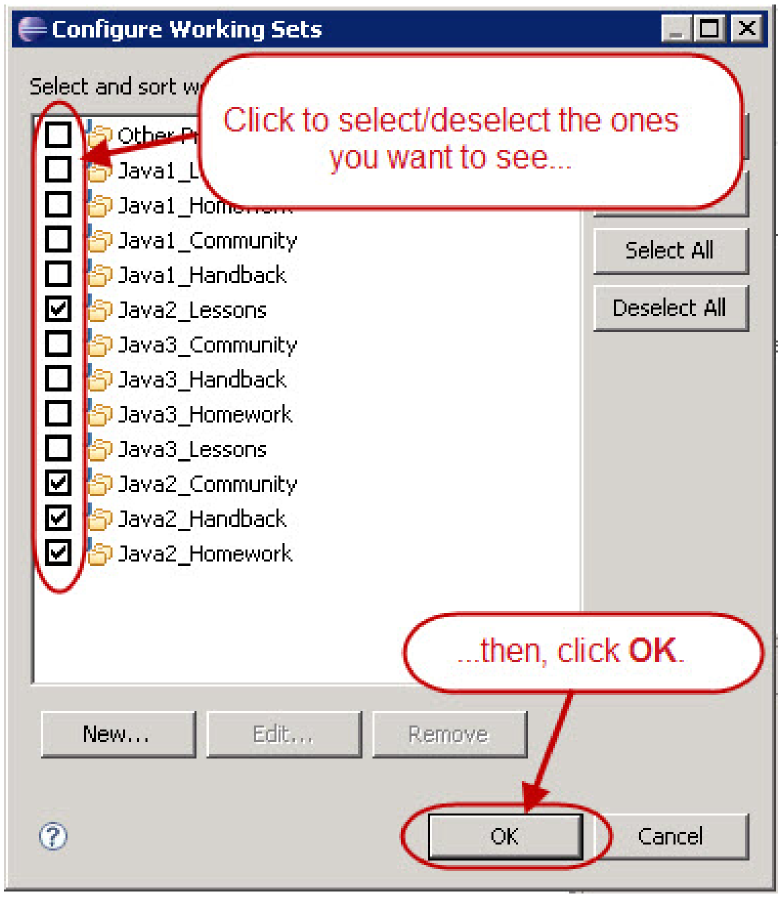
If you do not see working sets in your Package Explorer, set the top-level elements to working sets:



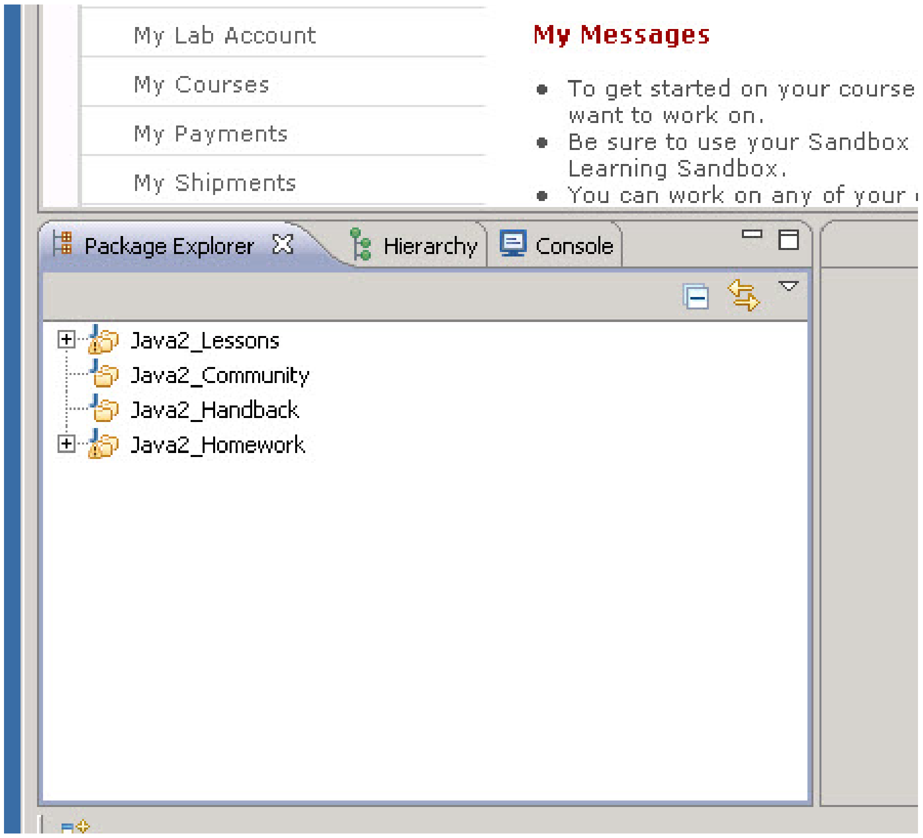
To configure working sets:



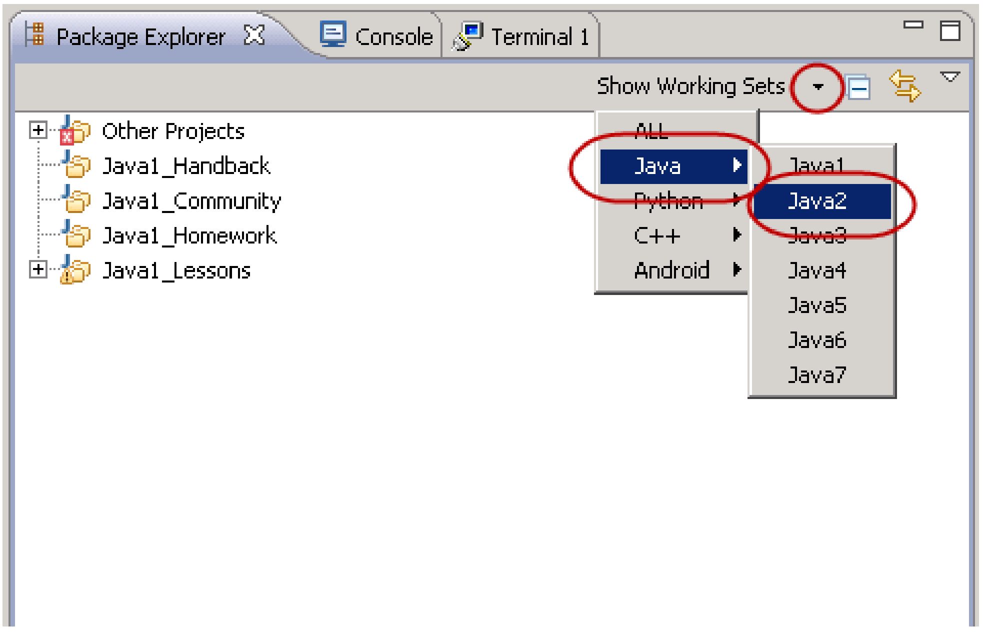
The following dialog appears. Select the working sets you want and deselect the ones you don't want in the Package Explorer view, and click **OK**.



The selected working sets now appear in the Package Explorer:



To easily show just the working sets for this course, click the down arrow on the **Show Working Sets** button in the Package Explorer, then select **Java**, and then **Java2**:

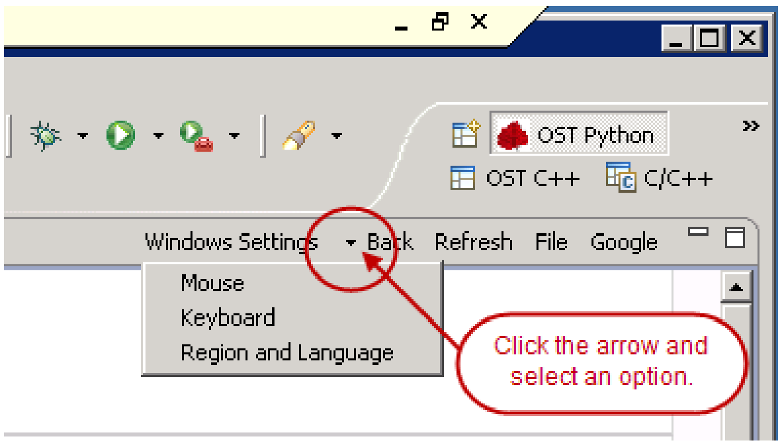


If the working sets already exist in the Package Explorer, they will not be recreated, but the perspective will still change.

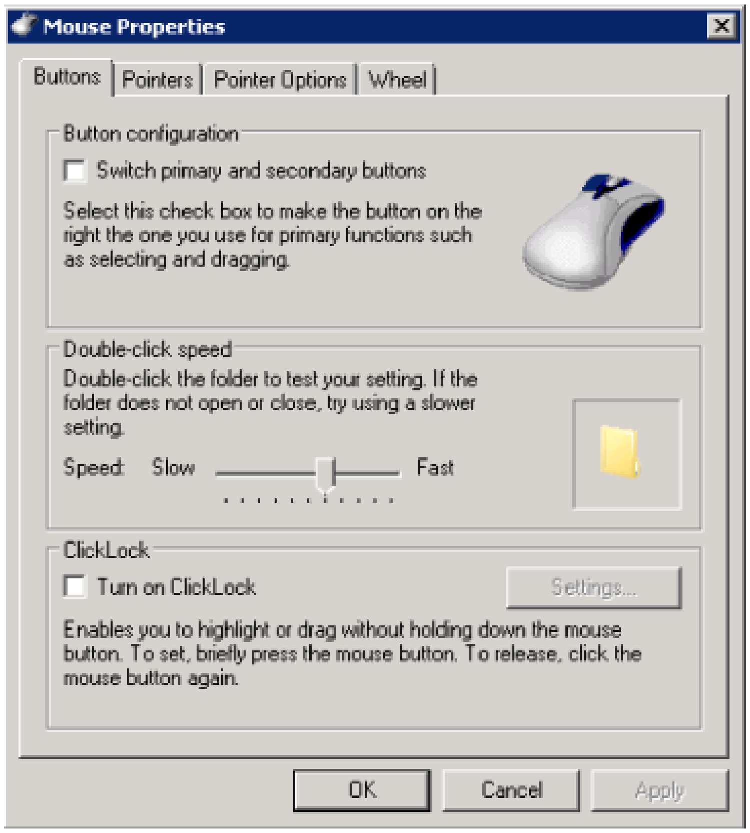
Hooray! Let's get going on some more Java Programming!

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 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 all of the details of these dialog boxes, but feel free to ask your instructor if you have questions.



Button configuration

☐ Switch primary and secondary buttons

Select this check box to make the button on the right the one you use for primary functions such as selecting and dragging.

Double-click speed

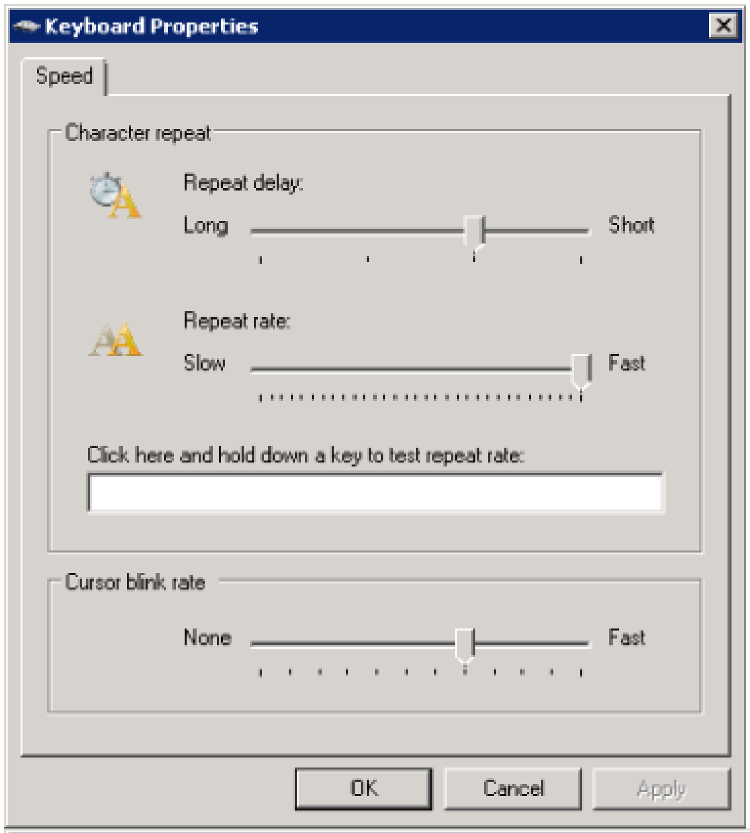
Double-click the folder to test your setting. If the folder does not open or close, try using a slower setting.

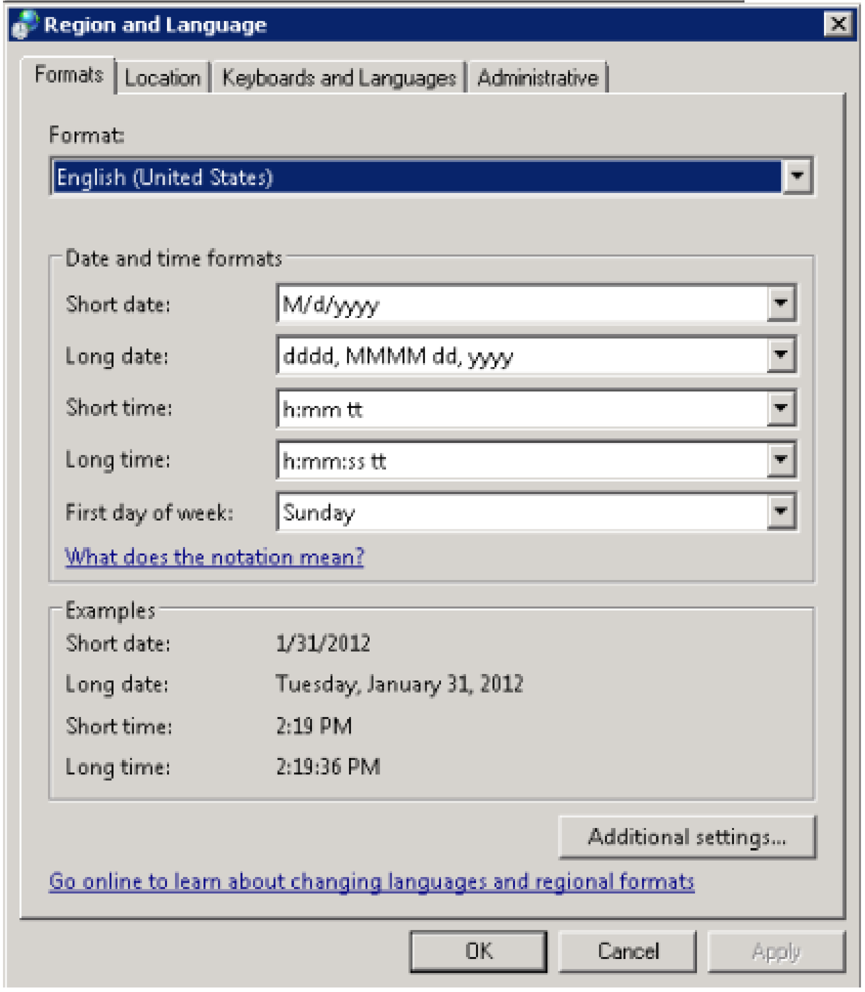
Speed: Slow Fast

ClickLock

☐ Turn on ClickLock

Enables you to highlight or drag without holding down the mouse button. To set, briefly press the mouse button. To release, click the mouse button again.





Format: English (United States)

Date and time formats

Short date:	M/d/yyyy
Long date:	dddd, MMMM dd, yyyy
Short time:	h:mm tt
Long time:	h:mm:ss tt
First day of week:	Sunday

Examples

Short date:	1/31/2012
Long date:	Tuesday, January 31, 2012
Short time:	2:19 PM
Long time:	2:19:36 PM

Fundamental Programming Constructs

The obvious reason that object-oriented programming languages use objects is due to the power that design principles such as inheritance, information hiding and polymorphism provide the programmer. Even though languages may be object-oriented, most also still use the basic constructs of programming and algorithms developed in earlier programming languages. Java is no exception. In this course, we will look into the basic programming constructs used by most computer languages and how Java implements them.

Basic Algorithm Constructs

Programs are computer code that provide sequences of instructions - they can be large or small.

Algorithms are the "recipes" ... the sequence of steps used to achieve the desired goal.

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

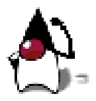
- sequences (assignment statements, IO calls)
- repetitions/loops (while, for, do)
- decisions/selections (if/then, switch)
- method invocation

That's it. When we tell a computer what to do next, we do it through these four **control** mechanisms. So all you need to know to program is to understand the principles and constructs of Objects, and to understand the control constructs above. We have seen sequencing and method invocation in the previous lessons; we will use them even more as the class progresses. However, since what we are "controlling" is what to do with information, and since Objects have information in them, in order to program we also need to know how the computer stores this information so that it can access it.

What and Where?

Consider:

- Algorithms are **what** we **do** to achieve the results we desire.
- In order to **do** something, one must have a **thing** to do some**thing** to!
- In order to have a **thing** we need to **have** it some**where**.
- The **where** in a computer is its **memory**.



In the first few lessons of this course, we will demonstrate the tools and techniques that programs use to determine **what** is **where** to allow the algorithms to do what they are supposed to do. We will also see how to represent the properties of Objects as variables so that they can be accessed and used to make statements, write methods, and hence provide computer legible algorithms.

Yes, yes, I hear you say, aren't these **things** the **object instances** that we get from our **Classes**? Yes, but there is more. In order to illustrate easily in a hands-on fashion, however, we need to introduce you to Java Applications first.

Applications

Applications are computer programs that are built and run in order to execute a specific task on a computer. Applications do not need browsers to run in and are often called **stand-alone programs**, meaning that they do not depend on any other program for their execution. Eclipse is a Java Application or stand-alone program! Unlike when making Applets, when you create applications, you have to make your own "windows" or else view your results from the **command window** or **console**. In this course, we'll run everything through Eclipse and view our results in the console it provides. In the Java 3 we'll make full-fledged applications.

Applications start from computer consoles which, unlike Applets, have no browser to start them. So the programmer of an application has to explicitly start the Classes in the application. In some programming languages (e.g., C, C++, Java), it has been a convention to start applications with a **main** method.

Note Even though the **main method** must be defined within a Class, it doesn't "belong" to any class, it's simply a means by which programmers get their objects started.

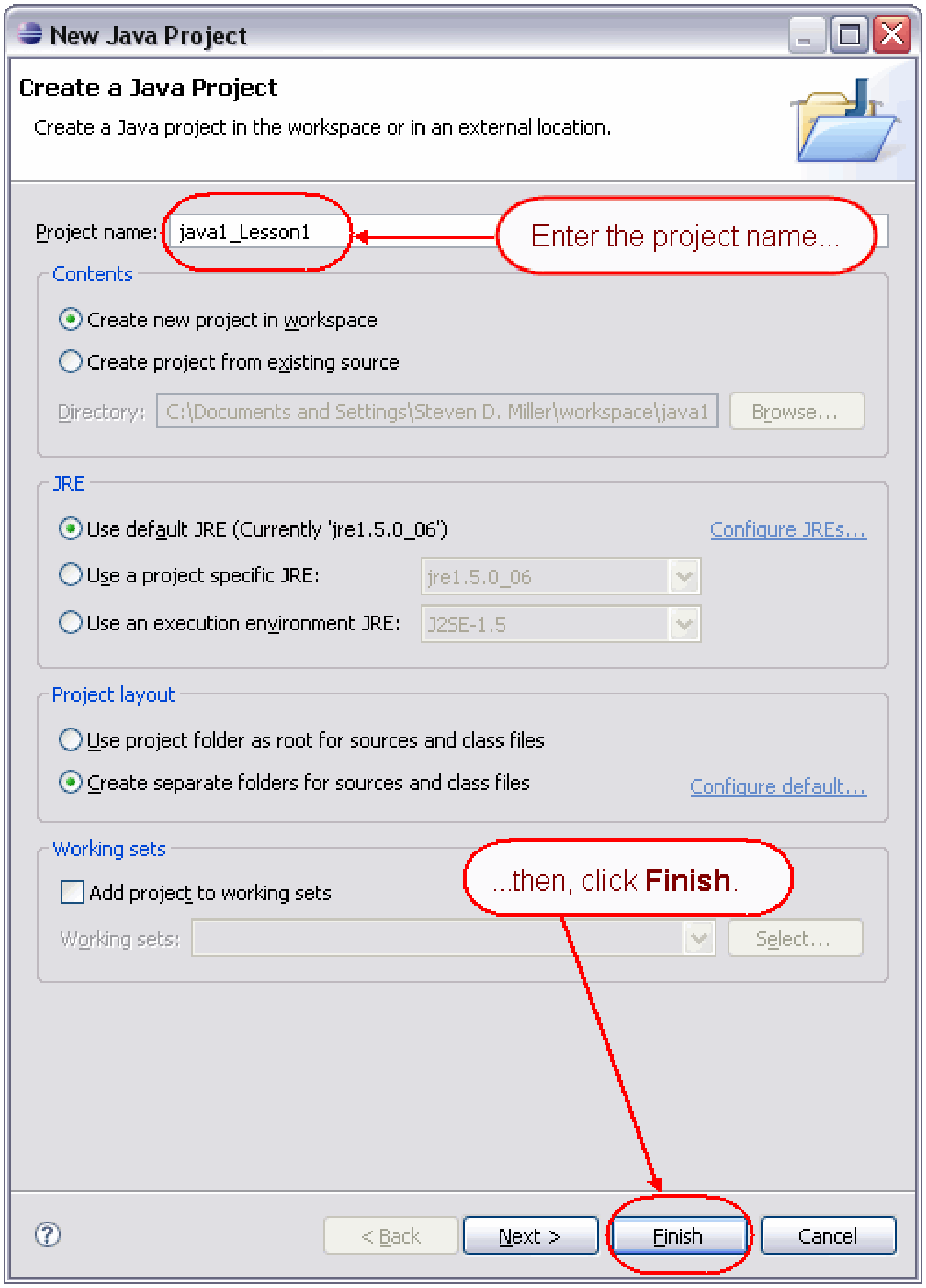
Tip For good programming practice, these are the only two things should be present in the main method:

1. A message to instantiate a class.
2. A message to get that class started.

Our First Application

Let's get started with an example of HelloWorld in an application. Make a **new project** for Lesson 1 for this course, **call it java2_Lesson1**.

Okay now select the **Java2_Lessons** project. Choose **File | New | Java Project**. Enter the name **java2_Lesson1** to differentiate from Course 1 java1_Lesson1. Click **Finish**.



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: java1_Lesson1

Contents

- (●) Create new project in workspace
- ( ) Create project from existing source

Directory: C:\Documents and Settings\Steven D. Miller\workspace\java1

JRE

- (●) Use default JRE (Currently 'jre1.5.0_06')
- ( ) Use a project specific JRE: jre1.5.0_06
- ( ) Use an execution environment JRE: J2SE-1.5

Project layout

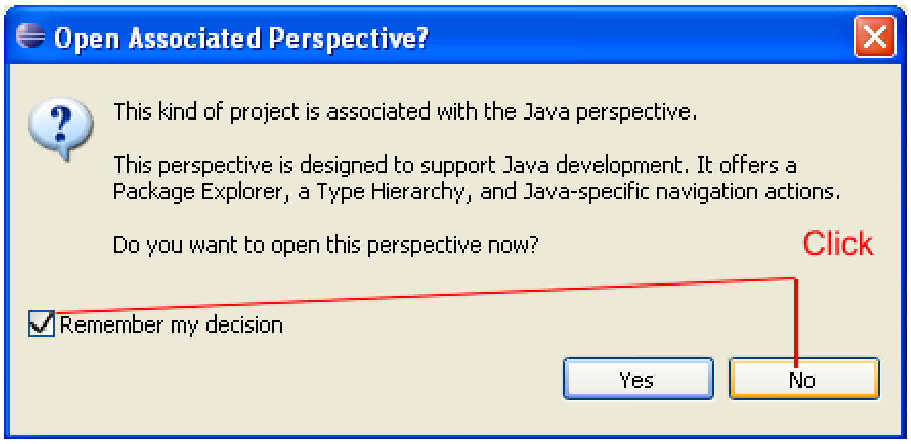
- ( ) Use project folder as root for sources and class files
- (●) Create separate folders for sources and class files

Working sets

- [ ] Add project to working sets

Working sets:

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



Open Associated Perspective?

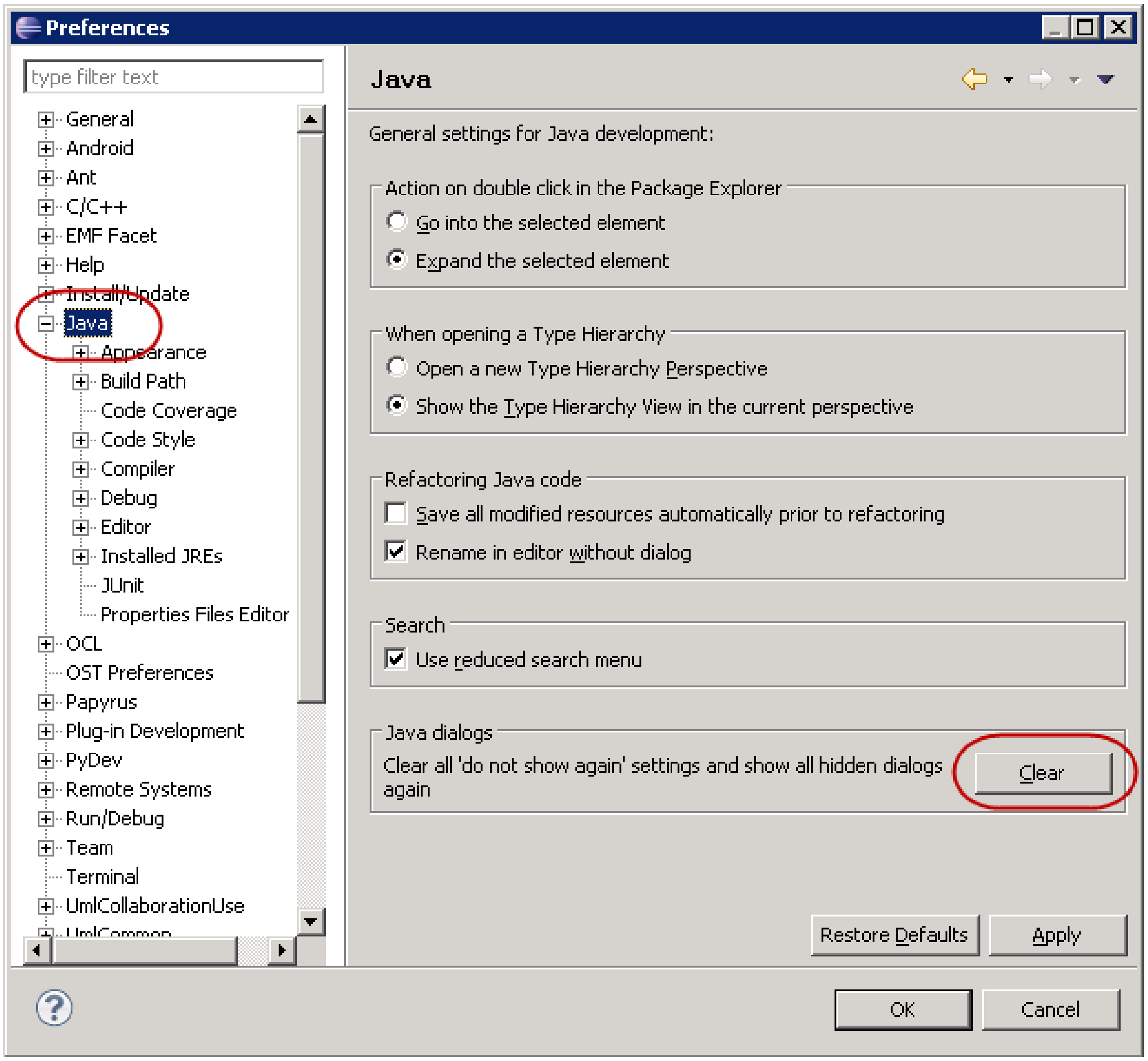
This kind of project is associated with the Java perspective.

This perspective is designed to support Java development. It offers a Package Explorer, a Type Hierarchy, and Java-specific navigation actions.

Do you want to open this perspective now?

- [✓] Remember my decision

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



General settings for Java development:

Action on double click in the Package Explorer
- ○ Go into the selected element
- ● Expand the selected element

When opening a Type Hierarchy
- ○ Open a new Type Hierarchy Perspective
- ● Show the Type Hierarchy View in the current perspective

Refactoring Java code
- ☐ Save all modified resources automatically prior to refactoring
- ☑ Rename in editor without dialog

Search
- ☑ Use reduced search menu

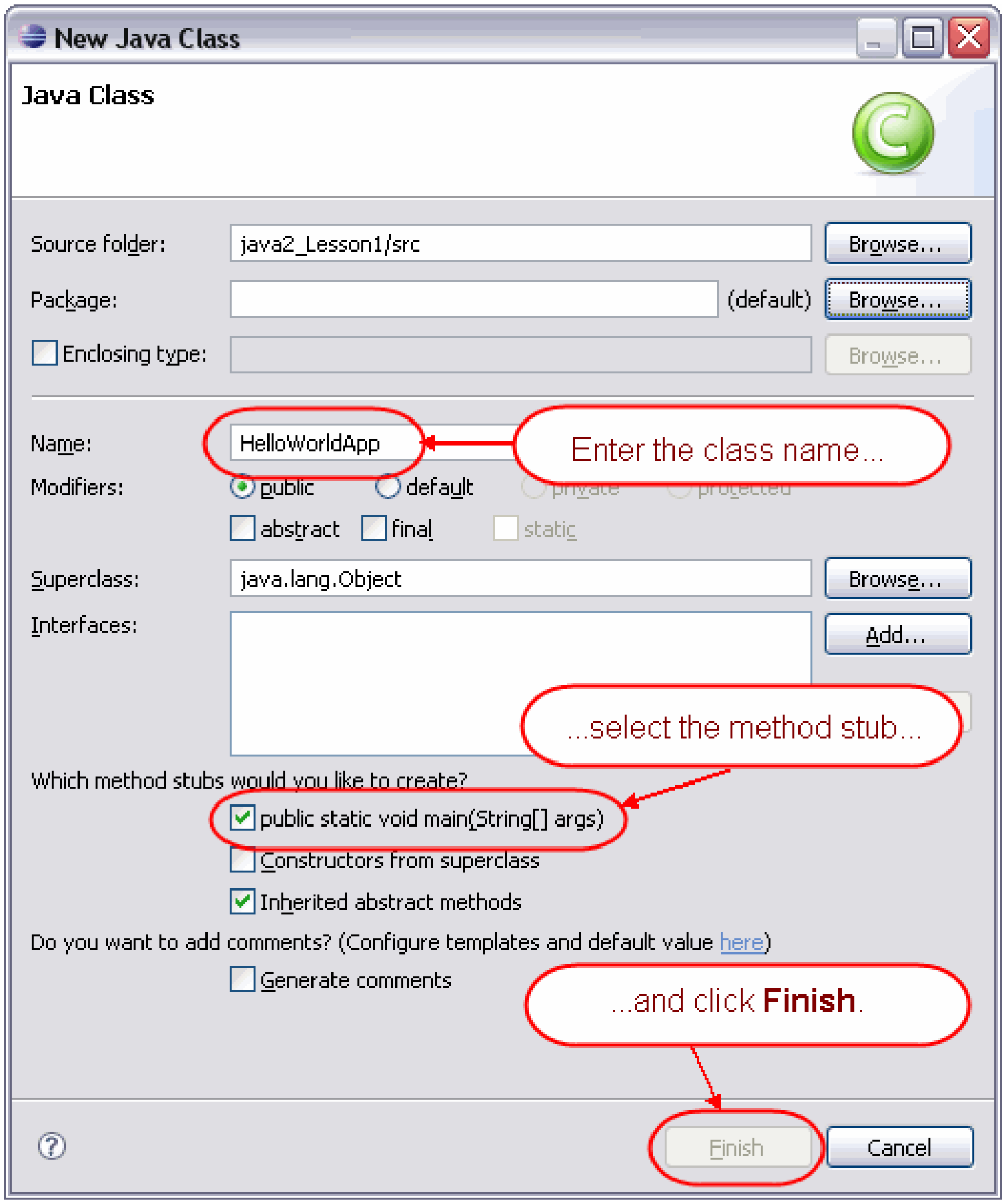
Java dialogs
Clear all 'do not show again' settings and show all hidden dialogs again [Clear]

Remember, if your workspace ever gets messed up, you can always hit the **Reset Button** to make things right again.

Now, let's make the first Class in this project:

Click on java2_Lesson1 and then right-click for the popup menu. Choose **New | Class** as you did in the first course. In the New Java Class window that opens, **Source folder** should be java2_Lesson1. Enter the **Name: HelloWorldApp**. **Since we're creating an application**, under "Which method stubs would you like to create?", choose **public static void main(String [] args)**. You can keep "Inherit abstract methods" checked--you don't need it now, but it doesn't hurt to have it.

Click **Finish**.



New Java Class

Java Class

Source folder: java2_Lesson1/src

Package: (default)

[ ] Enclosing type:

Name: HelloWorldApp

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

[✓] public static void main(String[] args)

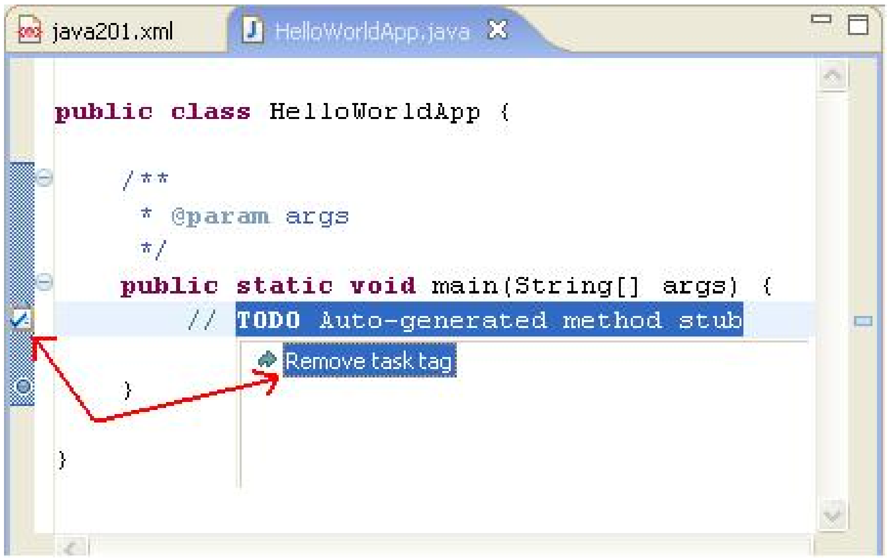
[ ] Constructors from superclass

[✓] Inherited abstract methods

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

[ ] Generate comments

In the resulting code, remove the comment: `// TODO Auto-generated method stub` and the symbol on the left, by clicking on the check box symbol once, then, in the suggestion box that opens, double-click on **Remove task tag**:



```
public class HelloWorldApp {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

Now, type the code shown in blue below.

CODE TO TYPE: HelloWorld Application

```
public class HelloWorldApp
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

Notice that HelloWorldApp does not `extend` anything. This is because an application is **not** an applet and in this case, it is not going to inherit from any special class--**except** the class of **Object**.

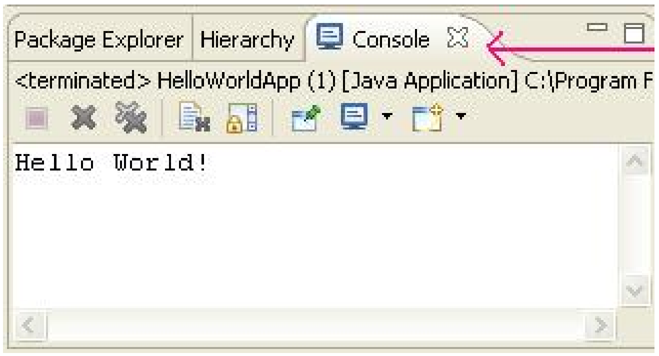
Note By default, all classes in Java inherit from the class **Object**. Nothing ever has to `extend Object`, because *every* object/class inherits from the class `Object`.

Now **save** the new application. To save it, right-click in the Editor Window and choose **Save** OR go to the Eclipse top menu bar and choose **File | Save**.

And **run** it. To do that, you right-click in the Editor Window and choose **Run As | Java Application** OR go to the Eclipse top menu bar and choose **Run | Run As**. That should give you the option **Java Application**.

To save and run an application or Applet quickly, just go to the Eclipse toolbar and click this icon: . Throughout the rest of the course, we'll just show you that icon with instructions to "**Save** and **Run**" your work.

Okay, you should now see the Console tab:



```
<terminated> HelloWorldApp (1) [Java Application] C:\Program F
Hello World!
```

Note The Eclipse tool is really powerful and versatile. If you ever choose **Run As** and neither a *Java Applet* nor a *Java Application* shows up, be sure to click *in the Editor Window* so Eclipse knows you're running the Java.

We'll use the console quite often to try different lines of code and see what happens. In fact, we will use the same HelloWorldApp to test all kinds of stuff.

The System Class

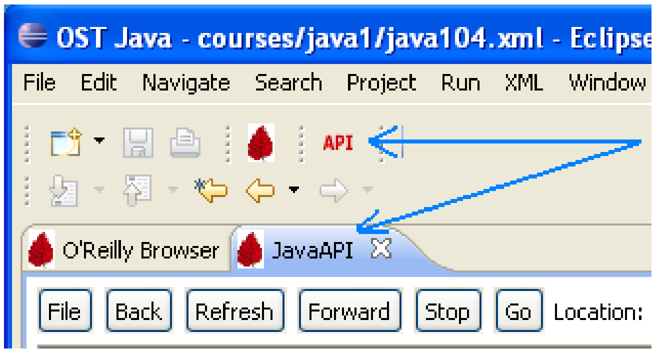
So why did the console open in our HelloWorldApp code above? It opened because we told it to, using the **System** class:

```
System.out.println("Hello World!");
```

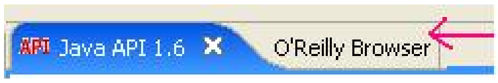
We know that **System** is a class because the word System begins with a capital letter and is not in quotation marks.

We didn't need to import anything into this **HelloWorldApp** class in order to **use** the **System** class, so we know that **System** must be in the package **java.lang**.

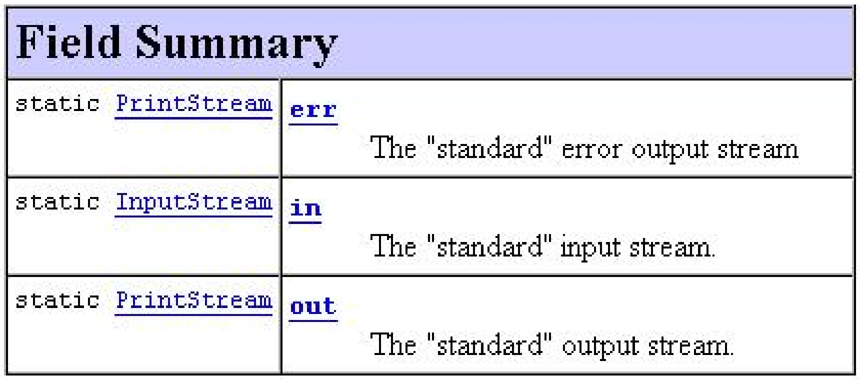
Go to the API using the icon in the Eclipse window:


OST Java - courses/java1/java104.xml - Eclipse
File Edit Navigate Search Project Run XML Window
API
O'Reilly Browser
JavaAPI
File Back Refresh Forward Stop Go Location:

You can always get back to these notes by clicking on the O'Reilly tab--it will keep your place when you toggle back and forth.



In the API, click on the **java.lang** package, scroll down to the **Class Summary**, and click the **System** class. Scroll down to the **Field Summary** of **System**:



Field Summary	
static PrintStream	**err** The "standard" error output stream
static InputStream	**in** The "standard" input stream.
static PrintStream	**out** The "standard" output stream.

In the left **modifier** column, we see that all of the Field variables in **System** are **Class Variables (CVs)**, because they all have the modifier **static**. Because they are CVs, you can access them from the class itself (rather than instantiating with **new**).

So we can say: `System.out`.

Also in the left column is information about variable **types**. We can see that they're **PrintStream** and **InputStream** Objects. Because they are **Objects**, they can use the dot operator as well.

These Class Variables for **System** are the **standard input and output** resources.

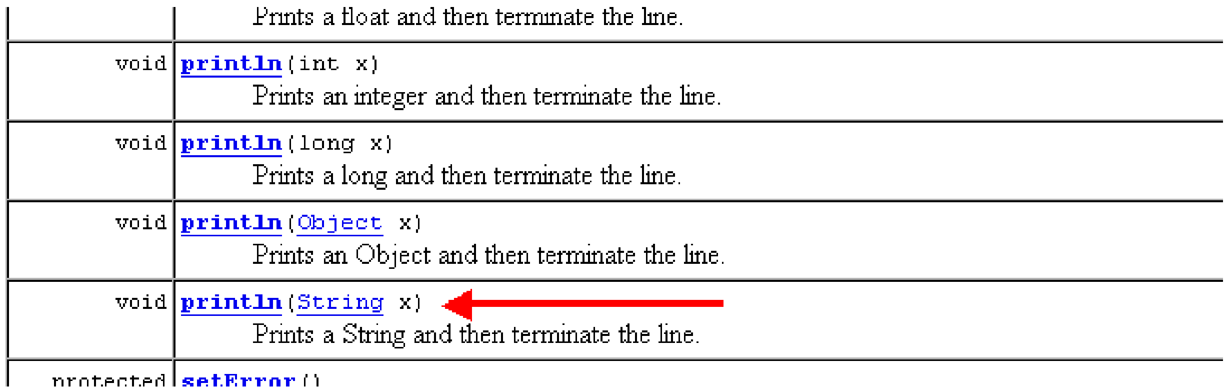
On your API page, click on the Class Variable **in** link.

Programs generally take information in and send information out. The detailed specification will tell us that **standard in** is the keyboard. Standard Input is data that goes into a program, it's usually text. Not all programs need to use standard input to get information, but they can. The **standard output** is the computer screen--particularly, the console with which you are working.

In the Eclipse environment, a window named **Console** opened for you for the **standard out** and **standard error**. In the previous course, we saw an example of **System.err** when we got the **null pointer error**. We'll mess up the main method on purpose in this lesson to see more.

Go back to the API page so you can get to the **out** variable. At the **out** variable, click on the **PrintStream** link. Scroll down the **PrintStream** class to its methods and find **println()**.

There are many **println()** signatures listed. Which one do you think Java is using in our application? Inside the parentheses of **System.out.println("Hello World!");** we see quotation marks, so we know we have a **String**, and so we also know the compiler will use this one:



	Prints a float and then terminate the line.
void	`println(int x)` Prints an integer and then terminate the line.
void	`println(long x)` Prints a long and then terminate the line.
void	`println(Object x)` Prints an Object and then terminate the line.
void	`println(String x)` Prints a String and then terminate the line.
protected	`setError()`

We do not say **where** (x,y) to put it, because it's not in a graphical window like an Applet.

Because **System.out** is one of the **type** of objects of the class **PrintStream**, **System.out** can invoke **PrintStream** methods:

System.out.println("Hello World!");, where **System.out** is the console.

A Closer Look at main

Because `main` is so important to applications, let's take a closer look. The template that the IDE created for you is the standard required template for applications.
Specifically, because all applications (often called **apps**) are started with the **main method**, **all** applications must have a method with this exact syntax and wording:

CODE TO TYPE: main

```
public static void main(String[] args){
    // what goes here is what will vary
}
```

Let's edit it now and see how `System.err` works.

In the HelloWorldApp template that was created for you, remove the code shown in **blue**.

CODE TO REMOVE: HelloWorldApp

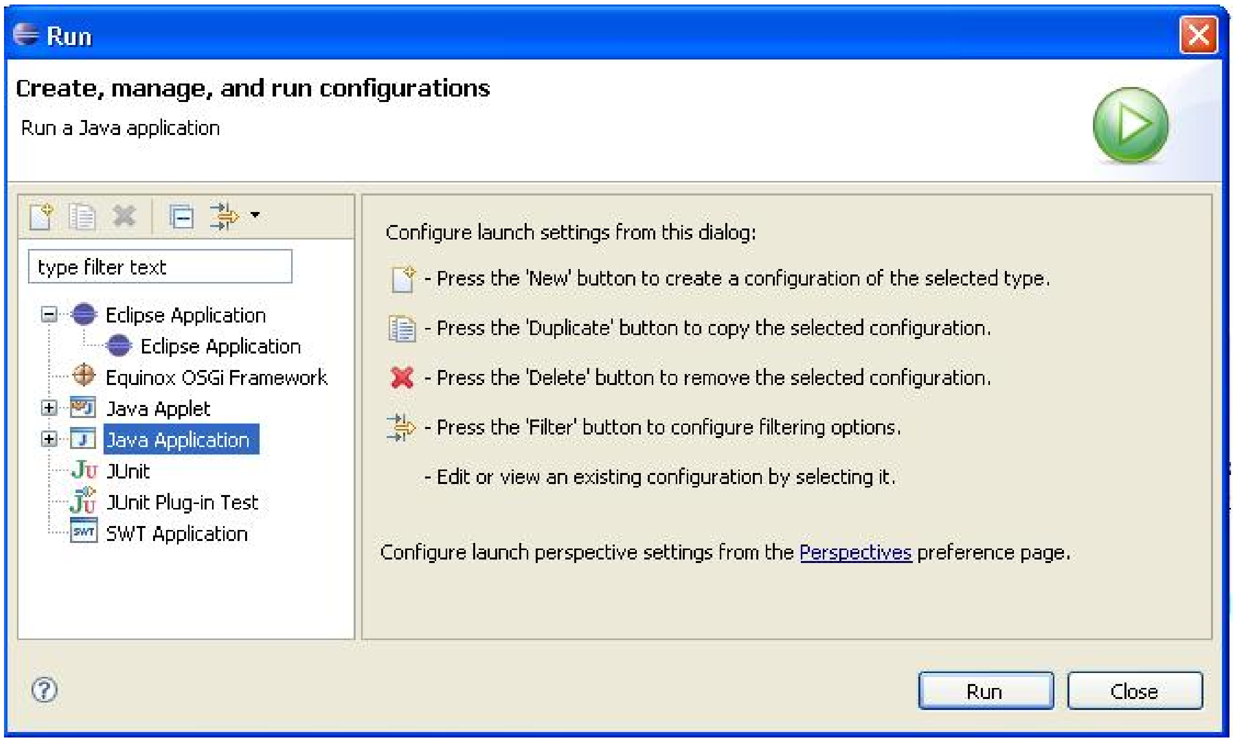
```
public class HelloWorldApp
  {
  public static void main(String[] args)
      {
        System.out.println("Hello World!");
      }
    }
```

Notice that there are no errors showing in the Editor.

Save and **Run** it.

This time something different will happen.

A new window will open:


Run

Create, manage, and run configurations

Run a Java application

type filter text

Eclipse Application
Eclipse Application
Equinox OSGi Framework
Java Applet
Java Application
JUnit
JUnit Plug-in Test
SWT Application

Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

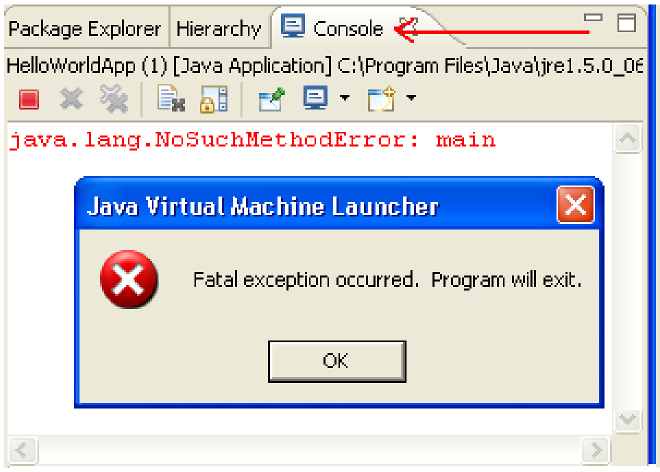
Configure launch perspective settings from the Perspectives preference page.

Run Close

If the **Run** button is gray (so it doesn't respond when you click on it), select **Java Application**, click the **New launch configuration** icon at the top of the dialog, enter the application name (HelloWorldApp) and Project name (java2_Lesson1), and then click **Run**.

Like before, you won't see a browser open, but this time, you'll see the results in the **Console** tab that opened up for you when you ran the application. In this case, we see an **error**.

If your results look like those below on the left, click **OK** in the "Java Virtual Machine Launcher" and you'll get the results you see below on the right:


Package Explorer | Hierarchy | Console

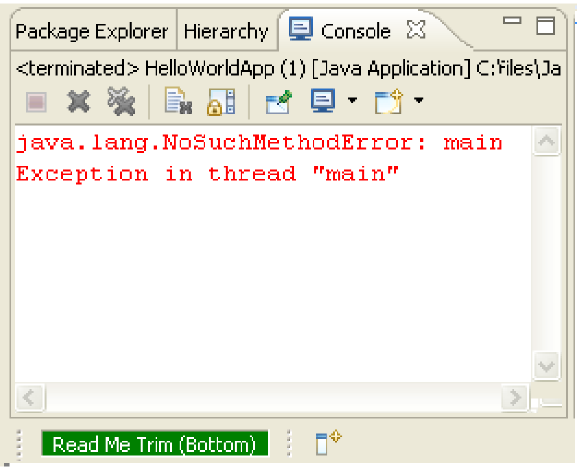
HelloWorldApp (1) [Java Application] C:\Program Files\Java\jre1.5.0_06

```
java.lang.NoSuchMethodError: main
```

Java Virtual Machine Launcher

Fatal exception occurred. Program will exit.

OK


Package Explorer | Hierarchy | Console

<terminated> HelloWorldApp (1) [Java Application] C:\files\Ja

```
java.lang.NoSuchMethodError: main
Exception in thread "main"
```

Read Me Trim (Bottom)

It is not good to see **red** in the console!

java.lang.NoSuchMethodError: main Exception in thread "main"

That red message is telling you that there are problems in your code--it cannot find your **main** method because the syntax is not correct.

This is an example of a **run-time error**, meaning it cannot be seen as an error until you try to run the program.

Eclipse can only catch **compiler errors**. Eclipse looks at the code within the editor and determines if the code there is correct. Specifically, until you run the application, Eclipse does not know that it will **be** an application and so it does not know that you need a **main method**.

Put the **blue** code back where it belongs in HelloWorldApp:

CODE TO ADD BACK: HelloWorldApp

```
public class HelloWorldApp
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

You can **save** and **run** it again to ensure that you replaced the code correctly, but there's nothing in our main method yet, so it isn't very interesting.

Documentation and Comments

In the generated code you see **documentation** in the form of **comments** (marked by / and *).

Java code is inherently hard to read and follow so Documentation is used to help us understand it. Since Java only allows certain syntax in the code, we can't just write whatever we want anywhere. We use special character combinations to include comments that allow us to communicate "normally" between lines of code.

We saw one type of comment when we used **//** and saw that Java would not pay attention to what came after the // on that line. There are three types of comments programmers can use in Java to inform readers of their intentions. Let's try them.

CODE TO TYPE: Comment Types

```

/** Javadoc comment. Used to automatically generate documentation
    that looks like the API pages. The compiler ignores
    everything until it sees the ending characters */

public class HelloWorldApp

/* Multi-line comment is ignored by the compiler until
    the ending characters */
{
    public static void main(String[] args)
    {
    // Single-line comment is ignored until the end of the line.
        System.out.println("Hello World!");
    }
}
```

Save and **Run** the program if you like. It should run the same as before, but the comments will be ignored.

Note The indentation in a program is only used to make it easier for us to read.

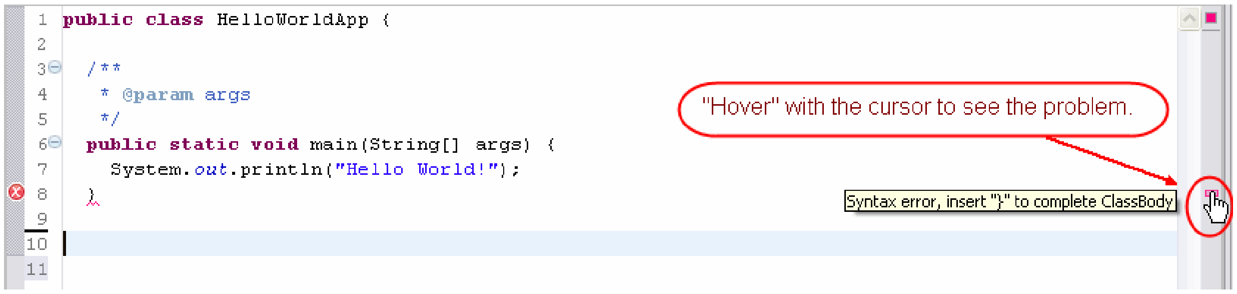
Java does not recognize multiple blank spaces as anything meaningful. It does not need (nor even recognize) this indentation. In fact, Java ignores any extra spaces. Some programmers like to line up the opening left bracket { with the closing right bracket }. Other programmers like to put the opening left bracket at the end of the line that defines what the bracket is opening (the definition of a class or a method).

In the code we, the brackets are lined up.

In the template code, the brackets start directly after the beginning of the class definition line and the method definition line.

Again, to Java this makes no difference **as long as they are there!** If they aren't there, Java will complain.

Try and remove the last "}" from HelloWorldApp. Click in the Editor Window (anywhere). See the little red rectangle in the far right column of the Editor Window? Slowly move the cursor over it so that its finger points right to the rectangle. It tells you that you need to put the **}** bracket back:



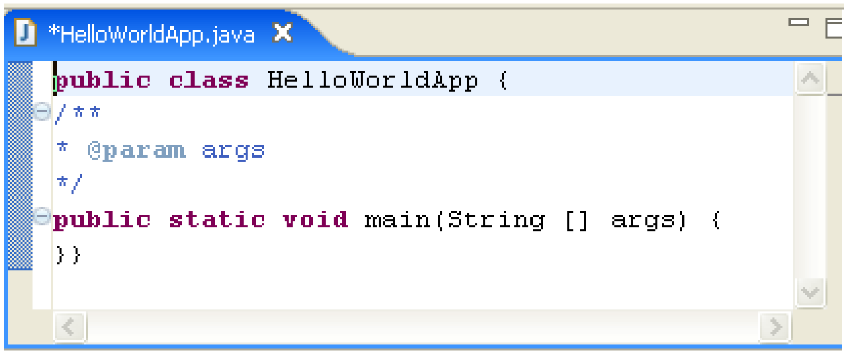
```
public class HelloWorldApp {

  /**
   * @param args
   */
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }


```

Syntax error, insert "}" to complete ClassBody

Now put the bracket back in the code.

Now, move the two ending brackets so that they are on the same line and all the way to the left.

Why not go all out? Remove all empty lines and move all of the code to the left (removing indentations):



```
public class HelloWorldApp {
/**
* @param args
*/
public static void main(String [] args) {
}}
```

There shouldn't be any errors indicated, but the code's kind of ugly.

Because our program is very small now, we don't need to use any comments. The compiler will ignore them anyway, so you can delete them or leave them there; it's up to you.

Let's get going to the next lesson where we can make our application output more interesting and also make some **STATEMENTS!**

Tools of Programming: Applications and Compilers

Lesson Objectives

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

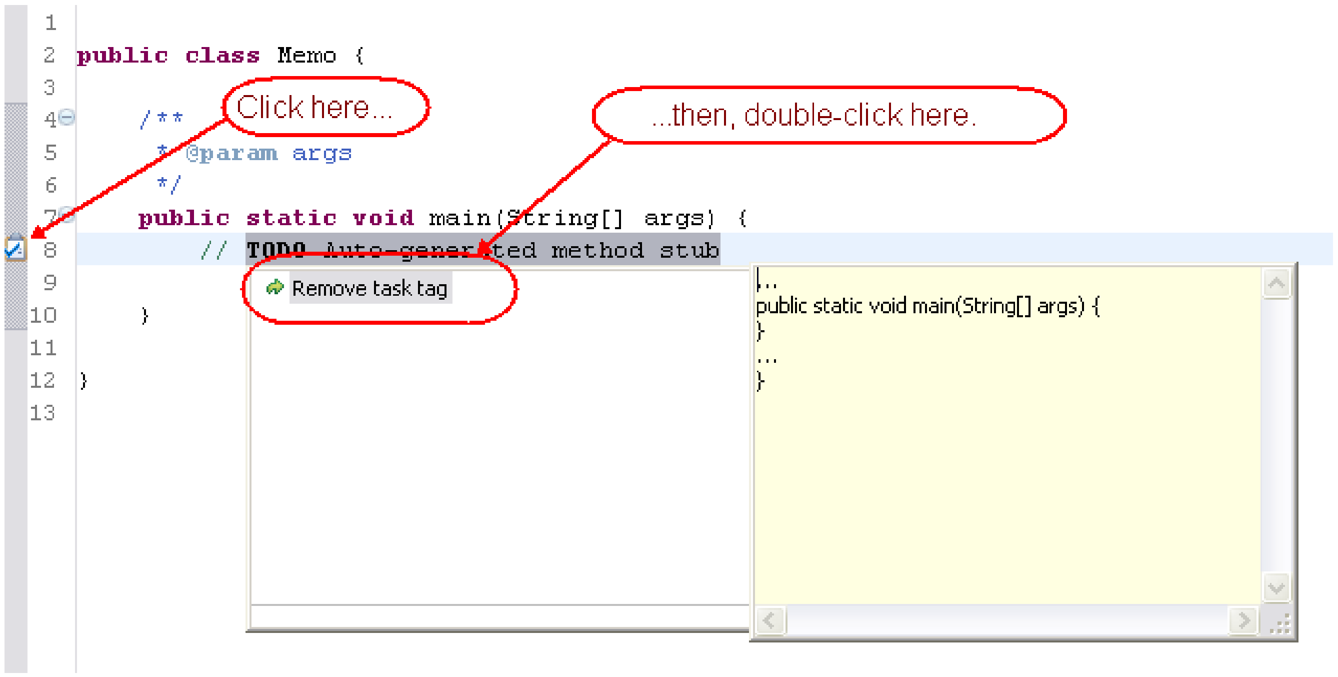
- demonstrate the differences between an application and an applet.
- use anonymous inner class instantiation.
- declare primitive data types.
- choose valid variable names.

Application GUIs

In the Java 1 course we learned that Applets displayed in browsers inherit their display abilities from the **Applet** class. Because of this inheritance, the Applet class instantiates itself automatically when the browser window opens.

We also learned that applications are not shown in a browser or window by default, so if we want a window for output, we'll have to create it ourselves. In order to do that, we can have the application inherit from the Java class **java.awt.Frame**.

Let's make an application with a Window **Frame** to see how they differ from Applets. Create a new Java Project and name it **java2_Lesson2**. Then, add a new Class and name it **Memo** (choose to include the **main method stub** like you did in the last lesson). When the Editor window opens, click on the check box on the left of the "ToDo" stub comment, and then double-click to **Remove task tag**:



```

public class Memo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

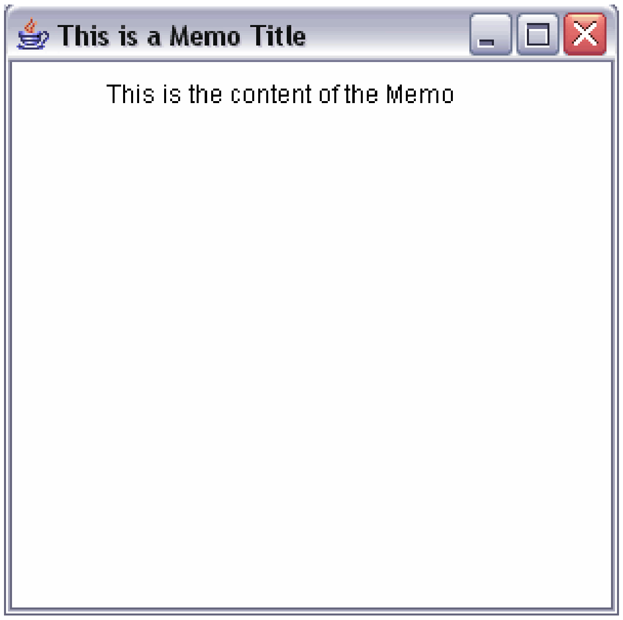
```
...
public static void main(String[] args) {
}
...
}
```

Enter code for **Memo** as shown below in **blue**:

CODE TO TYPE: Memo Class

```
import java.awt.event.*;
import java.awt.*;

public class Memo extends Frame {

    public String message;

    public Memo() {
        super("This is a Memo Title");
        message = "This is the content of the Memo";
    }

    public void paint(Graphics g) {
        g.drawString(message, 50, 50);
    }

    public void start () {
        setSize (300, 300);
        setVisible(true);
    }

    public static void main(String args[]) {
        Memo m;

        m = new Memo();
        m.start();
    }
}
```

Save and **Run** it.



```
This is the content of the Memo
```

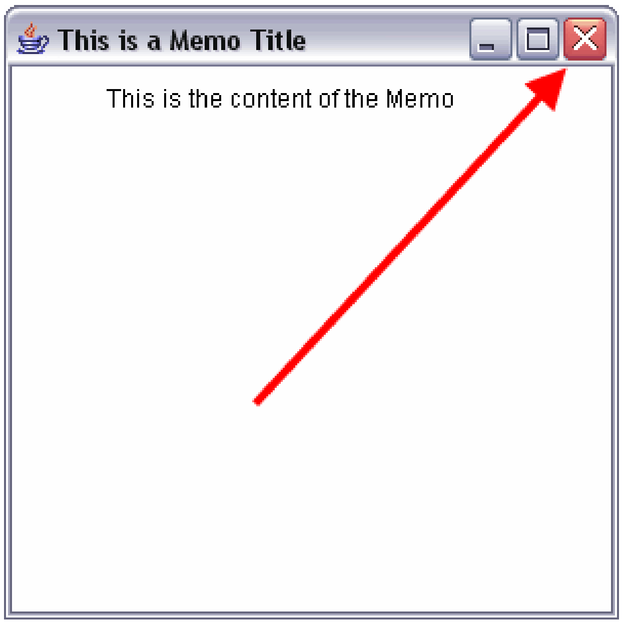
Let's trace this program:

Memo.java Results

```
import java.awt.event.*;
import java.awt.*;

public class Memo extends Frame {

    public String message;

    public Memo() {
        super("This is a Memo Title")[4];
        message = "This is the content of the Memo"[5];
    }

    public void paint(Graphics g) {
        g.drawString(message, 50, 50);
    }

    public void start () {
        setSize (300, 300)[7];
        setVisible(true)[8];
    }

    public static void main(String args[])[1] {
        Memo[2] m;

        m = new Memo()[3];
        m.start()[6];
    }
}
```

1. The program goes to the **main()** method.
2. The **main()** declares the **Memo** class.
3. An instance of the class **Memo** is created. This transfers control to the Constructor method **Memo()**.
4. The Constructor calls the Constructor of its parent **Frame** with the use of **super**.
5. The Constructor sets the Instance Variable of **message**. (If we don't call super, then the java compiler automatically calls super() without parameters. In this case we call it because we want to initialize the super Class with a String.)
6. When the Constructor is finished, the program returns to the **main()** method, which then invokes the instance's **start()** method.
7. The **start()** method sets the size of the Memo **Frame**.
8. Then the **start()** method sets the Memo **Frame** to "visible." In other words, the **start()** makes the Memo **Frame** open a Window to display.

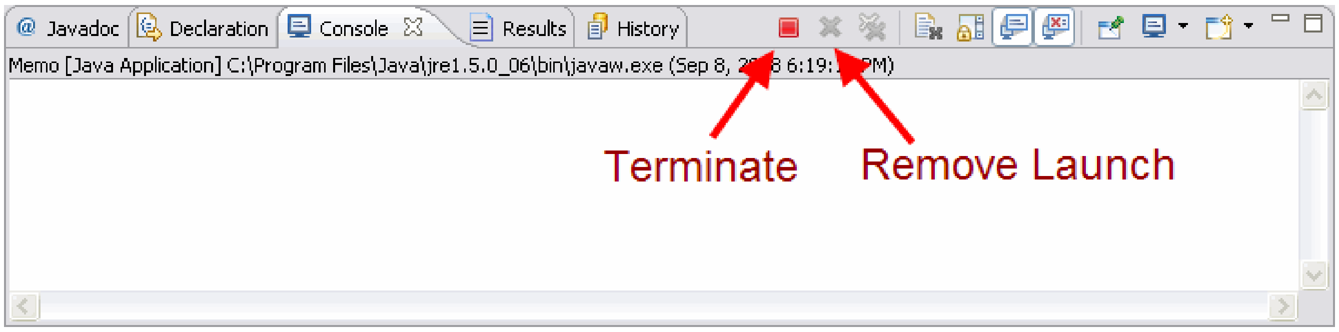
Click on the **X** in the **red** box in the upper right corner of the created window:



That's weird, nothing happened.

That's because the **X** in the **red box** in the upper right corner is not defined in the Java Frame. It's good programming practice to make sure your applications work the way people will expect. You can accomplish this using prompts and menu items.

To close the application (if the programmer leaves you stranded like this), go to the Console and move your mouse over the icons at the top of the Console Window. The first icon is a **red** rectangle that says, "Terminate." The next is an **X** that says, "Remove Launch." Click whichever one is available to you.



```
Memo [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Sep 8, 2008 6:19:1 PM)
```

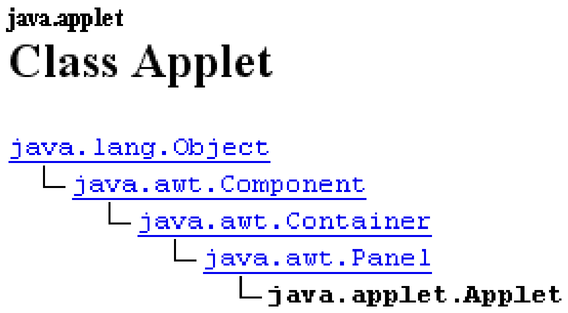
Tip If you aren't running your application in Eclipse, the common key command to stop an application is **[Control+C]**.

To understand the **Memo** code, we'll need the help of the **API**. Imagine that! One aspect of the code that really stands out is that **Frame** does not have a paint method--but then neither did **Applet**; both classes inherit the method from the class **Container**.

Check out both the **Applet** and **Frame** Class hierarchies. They both inherit from **Container** and therefore inherit the method **paint(Graphics g)**.

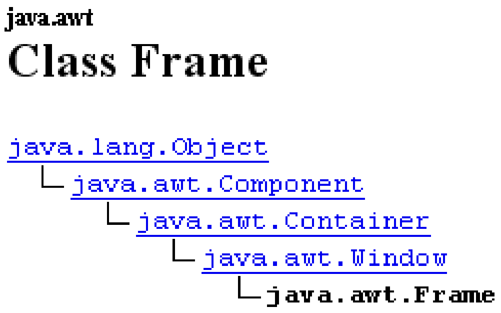
Go to the API using the **API** icon in the Eclipse window. Find the java.applet package, scroll to the **Class Summary**, and then click the **Applet** class.

Take a look at its hierarchical lineage:


java.applet
Class Applet

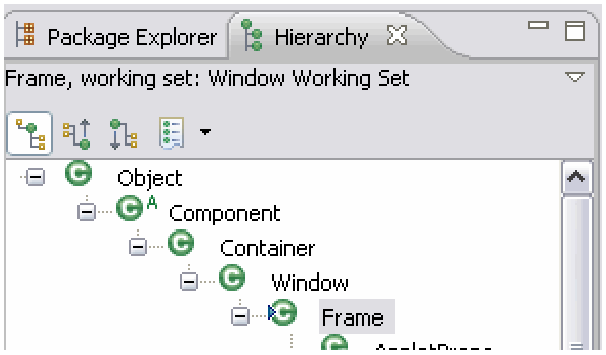
```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Panel
              └java.applet.Applet
```

Here you can see that he class `Applet` inherits from the class `Panel`, which inherits from `Container`, which inherits from `Component`, which inherits from `Object`. Okay, now go back to the **Packages** listing in the API. Click the **java.awt** package and scroll down to the **Class Summary**. Click the **Frame** class and note its hierarchical lineage:


java.awt
Class Frame

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
```

You can also see this using Eclipse. In the **Memo** class in the Editor Window, highlight the word **Frame**, right-click it, and choose "Open Type Hierarchy." The hierarchy displays in the Hierarchy Tab in the left panel:



Frame, working set: Window Working Set

```
Object
  Component
    Container
      Window
        Frame
```

Both the **Applet** and **Frame** classes inherit from **Container**, so they both inherit the method **paint(Graphics g)**. The **paint()** method is used less often in Frames than it is in Applets.

Go back to the Frame Class API page and scan through it and check out its constructors. See if you can identify the one we used by calling **super** in our **Memo** constructor. Find out where the methods that we used (but did not define) **are** defined. Specifically, **setSize** and **setVisible**. (Hint: Look at **Methods inherited from class java.awt.Window**.)

Now, let's experiment. Edit your Memo class by changing the code shown in **blue** below:

CODE TO EDIT: Memo

```
import java.awt.event.*;
import java.awt.*;

public class Memo extends Frame {

  public String message;

  public Memo() {
     super("This is a Memo Title");
     message = "This is the content of the Memo";
  }

  public void paint(Graphics g) {
     g.drawString(message, 50, 50);
  }

  public void start ( ) {
     setSize (300, 300);
     setVisible(false);
  }

  public static void main(String args[]) {
       Memo m;

       m = new Memo();
       m.start( );
  }
}
```

Save and **Run** it.

Wait... wait... wait...and...nothing. Hmm.

Open the Console window and click in it. See the blinking | at the upper left corner? It looks like something happened after all. Since you said not to make Frame visible, it didn't.

Click on the big X to "Remove Launch." Change the Memo code back to **true**.

CODE TO EDIT: Memo

```
import java.awt.event.*;
import java.awt.*;

public class Memo extends Frame {

  public String message;

  public Memo() {
    super("This is a Memo Title");
    message = "This is the content of the Memo";
  }

  public void paint(Graphics g) {
    g.drawString(message, 50, 50);
  }

  public void start ( ) {
    setSize (300, 300);
    setVisible(true);
  }

  public static void main(String args[]) {
    Memo m;

    m = new Memo();
    m.start( );
  }
}
```

Save and **Run** it to make sure it works like before.

For most of our lessons, we'll use **main()** and the console to test Java programming constructs without using a fancy **Frame**. But check out the differences between output on the **Frame** and on the **console** anyway. Generally, users of your application won't see the **console**. Output to the console is used to help programmers debug or trace programs.

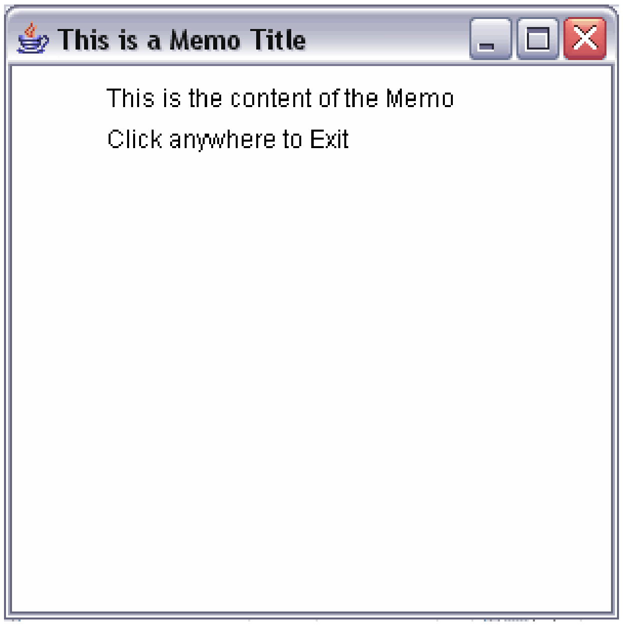
Let's make our program easier for users to exit, by adding some interaction. In the past, we implemented a listener; this time we'll show you a new trick.

Edit the **Memo** class's **paint()** and **start()** methods as shown below in **blue**:

CODE TO EDIT: Memo

```
import java.awt.event.*;
import java.awt.*;

public class Memo extends Frame {

    public String message;

    public Memo() {
        super("This is a Memo Title");
        message = "This is the content of the Memo";
    }

    public void paint(Graphics g) {
        g.drawString(message, 50, 50);

        g.drawString("Click anywhere to Exit", 50, 70);
        System.out.println("What's this then?");

    }

    public void start ( ) {
        setSize (300, 300);
        setVisible(true);
        this.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e){
                setVisible(false);
                dispose(); // free up system resources
                System.exit(0);
            }
        });
        // Quit the app.
    }

    public static void main(String args[]) {
        Memo m;

        m = new Memo();
        m.start( );
    }
}
```

Save and **Run** it. You see the frame defined like this:



```
This is the content of the Memo
Click anywhere to Exit
```

Click anywhere in the frame to close it.

OBSERVE: Look closely at this section of code.

```
public void start ( ) {
    setSize (300, 300);
    setVisible(true);
    this.addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e){
            setVisible(false);
            dispose();
            // free up system resources
            System.exit(0);
        }
    });
    // Quit the app.
}
```

So, how did we do that? Well, we **added a mouse listener using addMouseListener()**, but we did it without implementing the MouseListener Interface. So where does it come from? The **MouseAdapter** class is instantiated inside the **new MouseAdapter()** constructor, all within the method call itself as a parameter of **addMouseListener()**.

This type of instantiation is called an ***anonymous inner class***. It's *anonymous* because we aren't giving the instantiated class a name. The this.**addMouseListener(new MouseAdapter()**{...}**)**; portion of the code means that we are going to create a new MouseAdapter class and add it to the Frame as a MouseListener. We can do this because a MouseAdapter class implements the MouseListener interface (so it's a MouseListener).

Click the API icon in the Eclipse menu. Go to the **java.awt.event** package. Scroll down to the MouseAdapter class. Note that it implements the MouseListener interface.

Note

A **listener interface** with more than one method to be implemented in Java has a **corresponding Adapter class** that implements empty versions of the Listener's methods. This allows us to create an anonymous inner class, while overriding only the methods we wish. Go ahead and look up other interfaces and find their corresponding adapter classes.

After creating the new **MouseAdapter(),** inside the curly braces **{...}** we define the class itself and implement (override) the **mousePressed()** method. Inside the **mousePressed()** method, we set the frame to invisible. We also use the **dispose()** method from the Frame so that native display resources of the operating system are released and their memory is freed up (we do this when things that implemented classes--like Frame--are going away) Then, we call the System class **exit()** method to terminate the program. As a result, when the mouse is pressed in the Frame, the program terminates.

Anonymous inner classes are handy tools for adding listeners that perform a single, well-defined task, associated with a specific component. They are also useful if you want to create a Button to execute a specific task when clicked. Let's try it. Create a new class in your project called **ButtonTest**. Replace the code in the class with the following text:

CODE TO TYPE: ButtonTest

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;


public class ButtonTest extends Applet {
  String msg = "";
  public void init() {
    Button okButton = new Button("Click Here");

    //add an anonymous inner class as an ActionListener.
    okButton.addActionListener(new ActionListener() {
      // Create the actionPerformed method.
      public void actionPerformed(ActionEvent e){
        // Whatever you want to happen when the button is pressed
        // goes in this method.
        msg = "Button Pressed";
        repaint();
      }
    });
    add(okButton);
  }
  public void paint(Graphics g) {
    g.drawString(msg, 20, 60);
  }
}
```

This was just an example to show the use of annonymous inner classes and Java's ability to actually instantiate a listener interface, which is normally not possible. All interfaces are instantiated using the **implements** keyword. But there is **one exception** to this rule. We can **directly** instantiate an interface in Java by using **anonymous inner classes**. Java is smart enough to have the compiler create a new class that implements the listener on the fly.

Anonymous inner classes have access to all of the variables and methods of the class they are created within. In our case, that's all of the private and non-private variables and methods of the Memo class, and all of the non-private variables and methods of the Memo Class's ancestor Classes.

Anonymous inner classes are "fire and forget." Once it is installed as a Listener, we no longer have a handle to the object because they have no reference variable that we can access.

More on Programming Constructs

All languages have syntax and semantics.

- Syntax refers to the grammar or rules of a language.
- Semantics refers to the meaning of the words or phrases.

Computer languages are no different. For more detail, Oracle provides a Learning the Java Language Tutorial . It provides additional information about the "Language Basics" and other topics.

Statements and Expressions

Because Java is an object-oriented language, it includes syntax for writing Classes as well as for writing Methods **within** Classes. Here are a few other things you need to know about Java:

- A Java program is a collection of *statements* (like a story in English is a collection of sentences).
- A *statement* is a segment of code that takes some action in the program.
- When we run a program, we say that the compiler **executes** statements.

- Java statements end in semicolons (;).

Statements form a complete unit of execution. There are various kinds of statements, such as:

- **declaration statements**: For example, int x; (Type VariableName)
- **expression statements** (see next list)
- **flow control statements**

An *expression* is Java code that specifies and evaluates to a single value in the program. Check out a few examples:

- **assignment expressions**: e.g., x = 6; (VariableName = Value)
- **arithmetic expression**: e.g., y = x1 + x2;
- **relation expression**: e.g., (x < 7); (produces true or false)
- **method invocations**: e.g., System.out.println("hello again");

Primitive Data Types

In the examples above we see the use of variables x, y, x1, and x2. In order to have expressions, we need variables. Just like words in languages, **variables in computer languages need to be declared**. We have seen that sometimes Java "cannot resolve type" because an import that Java needed to be able to find the class wasn't there. What Java is saying in that message is, "I cannot understand you because you are using words (variables or Objects) that you have not told me about--so I cannot parse your statements."

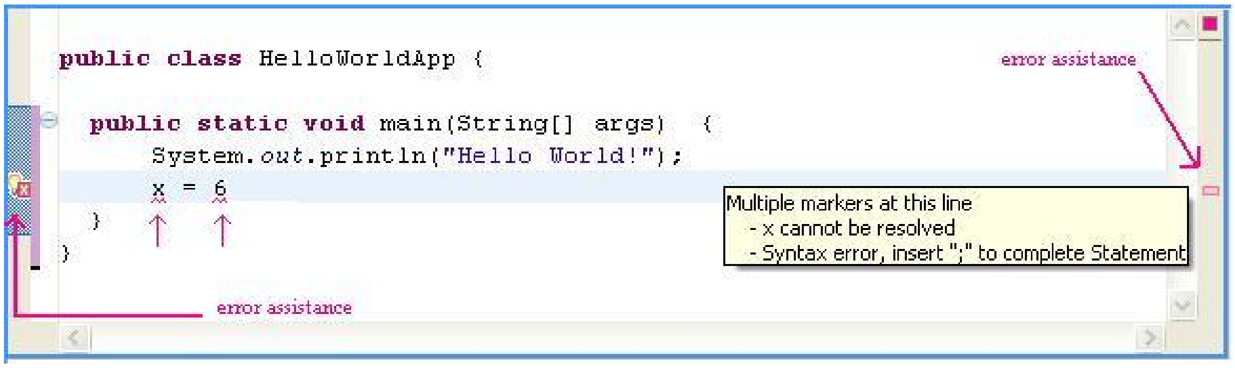
The x's above are not Objects that need to be imported, they are numbers. But they are still things that Java needs to know about. Let's see what Java does if we use them, but do not **declare** them.

Go to your java2_Lesson1 project, open your **HelloWorldApp.java** class, and add the line shown below in **blue**:

CODE TO EDIT: HelloWorldApp

```
public class HelloWorldApp {

   /**
    * @param args
    */
    public static void main(String[] args) {
        System.out.println("Hello World!");
        x = 6
    }
}
```

Remember that both the right column and the left column can indicate syntax errors in your code. If you click either the "X" error indicator on the left or the rectangle on the right, your editor workspace should look like this:



```
public class HelloWorldApp {

  public static void main(String[] args)  {
      System.out.println("Hello World!");
      x = 6
  }
}
```

Multiple markers at this line
- x cannot be resolved
- Syntax error, insert ";" to complete Statement

The arrows point to the errors found by the editor. Let's take a look at the second error. Just like sentences need periods, statements require semicolons in Java. This error message tells you exactly that: **Syntax error, insert ";" to complete Statement**.

In the first error (**x cannot be resolved**) we see that just like with Objects, Java needs to be informed of the **type** of numbers to expect as well. In fact, whenever you use a word--or even a letter--that isn't one of Java's **reserved words** or specific syntax characters, you need to tell Java what it is. How will Java know what your variable names stand for unless you tell it?

Every piece of data in a Java program is classified and stored in memory according to its **data type**. If you took the Java 1 Course, you may recall that some properties of objects are not other objects (with properties and methods of their own), for example numbers. In most object-oriented languages, numbers are represented differently because they don't have "methods" of their own. Numbers are interpreted as things that **we** do things with; they do not have actions of their own. We call them **primitive data types**.

Thus, there are two basic categories of data types in Java:

- the various types of Classes of Objects
- eight different built-in **primitive data types**

Note Primitive data types have a variable name and a single value--no methods.

Primitive data types are not Classes; they are simply a place in computer memory with a value. In lesson three we'll go over the different primitive data types and their usage, but for now let's find out how these variables are **declared** and **named**.

Let's fix your HelloWorldApp. Edit the new line as shown (add the extra line to make it a little more interesting):

CODE TO EDIT: HelloWorldApp

```
public class HelloWorldApp {

    public static void main(String [] args) {
        System.out.println("Hello World");
        int x = 6;
        System.out.println("The value of x is " + x);
    }
}
```

Nice. The red goes away.

Look at the **System.out.println** statement, and at the **string concatenation** in the method parameter. Java will print everything just as it appears inside the quotation marks. In this statement, Java will then concatenate to the String the **VALUE** of the x. We included a space inside the quotes after "is"--if we didn't, the system would print "The value of x is6". If a variable is **not** contained within quotation marks, Java will print the value in memory, not the variable name.

Save and **Run** the program to verify.

Valid Identifiers (aka. Variable Names)

One important element of good programming design is the use of **meaningful names** for your Classes and variables. In addition to being meaningful, variable names must also adhere to the syntax rules that enable Java to parse your code:

- Variable names must begin with a letter and consist of a sequence of letters or digits (0-9). In this context, a letter is defined as A-Z, a-z, _, $.

- They are case-sensitive.
- They cannot contain spaces.
- They can be any length.

Here are a few examples of valid variable names. Since Java is case-sensitive, it will differentiate between the second and third entries on the list.

1. identifier
2. user_name
3. User_name
4. _sys_var1
5. $change

Usually, we do not use underscore; instead, standard Java style uses capital letters to begin words within variable names (beginWord, tryThis, userName). The convention is to always begin variable names with a letter, not another symbol like "$" or "_". In fact the dollar sign character, by convention, is never used. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "_", the practice is discouraged.

Let's create a Class in Java that we can use to test things in a Project. Make a new project called temp, and add an Application (including the main method stub) named Test to it.

Because it's a Java Application it will have the main method. Make sure to check the box (like we learned in the last lesson) so Eclipse will do the work for you. When the Editor window opens, remove the "ToDo" stub comment.

Type the **Test** class as shown:

CODE TO TYPE: Test

```
public class Test {

    public static void main(String[] args) {
        int 4tryme;
        int HowAboutThis&That;
    }
}
```

Both variables have Syntax errors. Since we didn't follow the rules, Java does not know what we want. **Sometimes**, an explanation of an error doesn't explain the error clearly. However, the message will usually at least tell us which lines contain the errors, and make them easier to find.

Edit the variable names until there are no errors. Don't worry if you get any warnings--we're still in testing mode.

Tip According to convention, Class names start with capitals (Applet, Graphics, MyApplet). Primitive data types, methods, and instance and class variable names start with lower case letters (paint, myVariable).

Some words are used by Java for its own language constructs and so they cannot be used by programmers for variables. These are called *reserved words*, or *keywords*. Here's a list of Java keywords.

Let's see what happens when we use a keyword as a variable name. Change your Test class as follows:

CODE TO EDIT: Test

```
public class Test {

   public static void main(String[] args) {
      int case;
   }
}
```

We got an error message: **Syntax error on token "case", invalid VariableDeclarator**

We'll demonstrate the **values** that primitive data types can have in the next lesson.

Primitive Data Types, Variables, and Expressions

Lesson Objectives

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

- use the str variables and the methods available in the String class.
- use classes to format numbers.

Bits and Pieces in Memory

Variables

The Java Programming Language defines the following kinds of variables:

- **Instance Variables (IVs): Fields** in **Classes** that do **not** have the **static** keyword. Each instance of the class will have its own value for its instance variables, so memory is needed for the value for **each** instance.

- **Class Variables (CVs): Fields** in **Classes** that **do** have the **static** keyword. Instances of a class share a common value for its class variables, so class variables take only one place in memory.

- **Local Variables:** Variables within **Class method** definition **code**. The values of *local variables* are only accessible within the methods where they are declared--between the opening and closing brackets { } of a method.

- **Parameters:** Used in **Class method** definition **signatures** (formal parameters) and invocations (actual parameters). The signatures of a method identify the formal parameter types required for proper invocation of the defined method by other methods.

Primitive Data Types

The **values** of a variable must be defined as a certain **type**. We saw this when we declared all of our variable types. So far we've declared these types of variables:

boolean
String
int
Color
Graphics

Some of them begin with capital letters and some with lower-case letters. Those **beginning with capital letters are *Objects***, and those **beginning with lower-case letters are *primitives***. We'll cover those later in this lesson.

A data type describes the representation, interpretation, and structure of values manipulated by algorithms or objects stored in computer memory. As introduced in the previous lesson, in Java there are two basic data type categories: *primitive data types* and *Classes*. Classes have both attributes (properties) and actions (methods). Primitive data types are "primitive" in that they are basic building blocks that do not have the complete object structure. The Java language (rather than the methods of a class), defines how the values can be manipulated.

Note Some languages have no primitive data types. In these languages, **everything** is an Object. These are called "*pure* object-oriented languages".

Java has eight basic (primitive) element types, because it provides four (4) integer types, two (2) decimal-point types, one (1) char type, and one (1) boolean type. But in practice, Java uses four basic element types: *integer, decimal point, boolean,* and *character*.

These types allow us to maximize speed and space. Each type of element requires a different amount of space in memory. Using smaller numbers or element types that occupy less memory space, allows for more speed because smaller elements require less time to travel from memory to the CPU. Selecting the appropriate element type allows us to avoid taking up more space than necessary so we can maximize speed. Check out the table below. It shows the four types of **integers** and the amount of space in memory each is allowed:

Primitive Data Type - Integer

Length	**Name**
8 bits	byte
16 bits	short
32 bits	int
64 bits	long

Because computer data is represented by zeros and ones (base 2), the number of bits (base 2 numbers) allowed will limit the range of numbers allowed. One of the bits will indicate whether the integer is positive or negative.

Data Type	**Range**
byte	-128 to 127
short	-32768 to 32767
int	-2,147,483,648 to 2,147,483,647
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

The most common integer type is: **int**

If you type 4, you get an **int** with value 4. Programmers usually use **int** as their default.
To use a longer integer, use the type **long** by adding an **L** (upper or lower case) to the end of the number, like this:

long myNumber = 12345678900**L**;

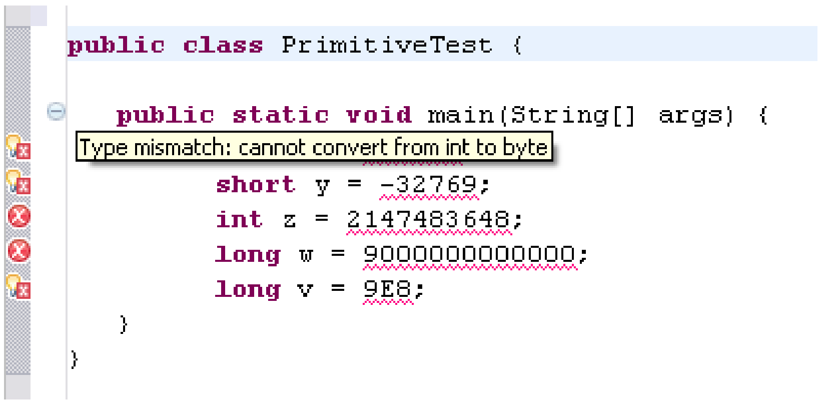
Let's see what happens when we declare an integer of a specific type and then give it a value that is not allowed.

In the Java2_Lessons folder, create a new Project named **java2_Lesson3**, and then create a Java Application class named **PrimitiveTest**.

CODE TO TYPE: testing types

```
 public class PrimitiveTest {

   public static void main(String[] args) {
     byte x = 344444;
     short y = -32769;
     int z = 2147483648;
     long w = 9000000000000;
     long v = 9E8;
     }
}
```

Whoops. It looks like we've got some errors. Actually, we had you create code that contains errors on purpose, to demonstrate a few concepts. Move your mouse over the first error indicator:



```
public class PrimitiveTest {

    public static void main(String[] args) {
Type mismatch: cannot convert from int to byte
            short y = -32769;
            int z = 2147483648;
            long w = 9000000000000;
            long v = 9E8;
    }
}
```

When you mouse over an error you'll see an error message; for example, the one on the line containing *byte x = 344444* has the error:

Type mismatch: cannot convert from int to byte

We got this error because we told Java that **x** was a byte, but we gave **x** a number that is in the **int** range. It's impossible for Java to take the integer value of 344444 and put it into the memory space allowed for a **byte**, so we get a **Type mismatch**.

The same logic applies to the error on the **short** variable we declared; the space for **short** is not big enough:

Type mismatch: cannot convert from int to short

To correct these errors in your code, make the changes shown in blue below:

CODE TO TYPE: testing types

```
public class PrimitiveTest {

    public static void main(String[] args) {
        int x = 344444;
        int y = -32769;
        int z = 2147483648;
        long w = 9000000000000;
        long v = 9E8;
    }
}
```

Now, mouse over the next error symbol. You see:

The literal 2147483648 of type int is out of range

The type is actually out of range by 1. You can verify this by referring back to the table of the ranges of values above.

Changing the number as shown below and the error should go away:

CODE TO TYPE: testing types

```
public class PrimitiveTest {

    public static void main(String[] args) {
        int x = 344444;
        int y = -32769;
        int z = 2147483647;
        long w = 9000000000000;
        long v = 9E8;
    }
}
```

Since we use commas when we write longer numbers, you may be curious about whether commas are used in programming using integers, especially the longer ones. Let's find out! Change the number as shown in blue below:

CODE TO TYPE: testing types

```
public class PrimitiveTest {

    public static void main(String[] args) {
        int x = 344444;
        int y = -32769;
        int z = 2,147,483,647;
        long w = 9000000000000;
        long v = 9E8;
    }
}
```

You get all kinds of errors now, but here's the one to pay attention to:

numbers cannot contain commas

Take the commas out and the errors will go away. Now, what's going on with the **long** declared variable? The error says:

The literal 9000000000 of type int is out of range

We declared **y** as a **long**; why did it say type **int**? Because 9000000000 is out of range for an **int**, we need to indicate that it's a **long** by adding the "L" to the end. Otherwise, the default for numbers is type **int**.

Change the code as shown in blue:

CODE TO TYPE: testing types

```
public class PrimitiveTest {

    public static void main(String[] args) {
        int x = 344444;
        int y = -32769;
        int z = 2147483647;
        long w = 9000000000000L;
        long v = 9E8;
    }
}
```

Finally, we have an error on the line that contains the code **long v = 9E8;**. The error message says:

Type mismatch: cannot convert from double to long

Change the type as shown in blue:

CODE TO TYPE: testing types

```
public class PrimitiveTest {

   public static void main(String[] args) {
       int x = 344444;
       int y = -32769;
       int z = 2147483647;
       long w = 9000000000000L;
       double v = 9E8;
   }
}
```

Everything should be fine now. Still, you might be wondering, "what's a double?" The next set of primitive data types represent numbers with decimal and exponent values. Java provides two types here: **float** and **double**.

Primitive Data Type - Floating Point (Decimal)

Decimal:

Length	**Name**
32 bits	float
64 bits	double

Here are some examples of numbers with decimal values:

- 3.14
- 3.1E12 means 3.1 X 10^{12}, which is shorthand for 3100000000000.
- 2e12 means 2 X 10^{12} (E or e makes no difference).
- 12.34f means 12.34 floating point.
- -32.0f means -32.0 floating point.

Similar to the way we used **L** at the end of integers to indicate the type **L**ong, you can use e or E (for **E**xponents or powers of 10), f or F (for **F**loat), d or D (for **D**ouble). **D**ouble is the default though, and usually omitted. You **must** use the f or F for **F**loat types. To find detailed information about the ranges for both of these types, visit the Java Language Specification. For our purposes, just keep in mind that float and double allow big numbers, so big that they aren't likely to come up for us much in our work. But it's still good to know about them. Here's a rough table of ranges just so you can get the idea:

Data Type	**Range**
float	+-1.4 * 10^{-45} to +-3.4 * 10^{38}
double	+-4.9 * 10^{-324} to +-1.7 * 10^{308}

A variable must be declared before it can be used. You can declare it when you first mention the variable, or at the same time its value is specified. For decimal point values, **d**ouble is the default if you omit the **f**. So, what would happen if you typed
`float myValue = 12.3;`? Give it a try.

Edit the **PrimitiveTest** class's **main** method as shown:

CODE TO TYPE: Can you guess what will happen?

```
public class PrimitiveTest {

    public static void main(String args[])
    {
        float myValue = 12.3;
    }
}
```

Look at the error column:

Type mismatch: cannot convert from double to float

Does that make sense? You're *declaring* the myValue variable to be a **f**loat, but because the 12.3 does not have an **f** at the end, Java reads it as a **d**ouble.

There are two ways to fix this:

- Declare myValue to be a float: `float myValue =12.3f;`
- Declare myValue to be a double: `double myValue = 12.3;`

Try both solutions in the Editor Window. Be sure to click the mouse somewhere after each change, so Eclipse accepts it.

CODE TO EDIT: PrimitiveTest

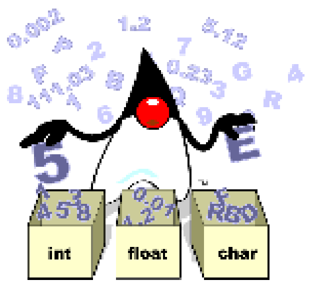
```
 public class PrimitiveTest {

    public static void main(String args[])
    {
        float myValue = 12.3f;
    }
}
```

OR

CODE TO EDIT: PrimitiveTest

```
 public class PrimitiveTest {

    public static void main(String args[])
    {
        double myValue = 12.3;
    }
}
```

Both work fine.



Even Duke has to sort out the right type!

Primitive Data Type - Boolean (boolean)

Booleans are used for logical reasoning and have only two possible values: true or false. **Not** T or F, or TRUE or FALSE, only **true** or **false**.

Let's check it out! Edit the body (inside the block {}) of the **PrimitiveTest** class's **main** method as shown:

CODE TO TYPE: booleans

```
public class PrimitiveTest {

    public static void main(String args[])
    {
        boolean testMe = false;    // note declaration of lower case boolean
        if (!testMe)
            System.out.println("testMe might be false but !testMe is " + !testMe
);
    }
}
```

Save and **run** it. The output should be in the console. It should say **testMe might be false but !testMe is true**. In Java the **!** means **not** (or *the opposite of*). So, if **testMe** is true, then **!testMe** is false. If **testMe** is false, then **!testMe** is true.

Primitive Data Type--Character (char):

Characters are just that--single characters. Think of them as the keys on your keyboard. Each key is a character. In Java, we enclose **char**acters inside single quotation marks: you type 'a' to get a **char** with value **a**:

char myCharacter = 'a';

Here are some special "escape sequences" or keystrokes represented in Java:

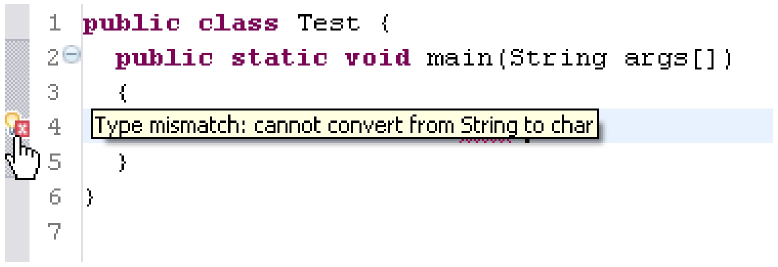
\t (tab)
\b (backspace)
\n (linefeed)
\r (carriage return)
\f (form feed)
\" (double quote)
\' (single quote)
\\ (backslash)

These escape sequences are used to allow Java to define a character being used. The backslash alerts Java that the next character typed will be used to perform the action shown in the parentheses.

To define the character **m**, you'd use `char myOtherCharacter = 'm';` So, why the single quotation marks, you ask? Edit the **Test** class's **main** method as shown to find out:

CODE TO EDIT: PrimitiveTest

```
public class Test {
    public static void main(String args[])
    {
        char myCharacter = "m";
    }
}
```



```
public class Test {
  public static void main(String args[])
  {
Type mismatch: cannot convert from String to char
  }
}
```

See the capital letter at the beginning of **String**? Double quotation marks are for a **Class** in Java called `String` in the java.lang package. That's why we use single quotation marks in this case.

Strings

Strings are not primitive data types--they are a Java class called **String** in the package **java.lang**. We'll cover Strings in the next lesson.

Literals

A literal is the source code representation of a fixed value. Literals are represented directly in your code without requiring computation. The examples below show how it is possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalD = 'D';
byte b = 42;
short s = 5280;
int i = 100000;
```

We have set values like this in previous examples, but now if you hear the word **literal**, you know it is just a fixed value (like a number) to which you can set primitive data type variables.

Memory: variable names and values

We need to look carefully at **Assignment statements**. Consider this line of code:

Type variableName = expression;

There are **three** important elements for you to observe:

1. The left side of the = sign is the **location** where the value of the variable is stored (variableName is a memory **location**).
Type variableName; is declaring the Type of variable that will be placed into the location of **variableName**. Java needs to know the Type in order to determine how much space to reserve in memory at that location.

2. In programming languages, a **location** or an **address** in memory is found on the left side of the equal sign **(=)**. It works kind of like an old-school card catalog in a library:

You look up a title of a book to get a number for where the book is located.

You don't go to the card catalog for the **book** itself, but for the book's location. The left side of an equal sign **(=)** in programming languages **always** indicates **only** the location of the value for the variable. Because the variableName tells Java the **location**, just like card catalog locations, you don't perform any manipulation (arithmetic) on these numbers.

Note The only things on the left side of an equal sign in Java are the variableName, which is an **address**, and possibly the **type** of variable that will be placed **into** that location.

3. On the right side of the equal sign **(=)** are expressions. Expressions calculate the **values** that will be put into those memory locations.
variableNames (left side) are memory locations where the **values of expressions (right side)** are stored once calculated.

When the computer sees a variableName on the **right** side of an equal sign **(=)**, it goes and **gets** the value at that location.
When it sees a variableName on the **left** side of an equal sign **(=)**, it **puts** the result of the right side into the address of the left side variableName.

What does the computer do when it reads `int length = 20;`? I'm so glad you asked. Actually, it does two things. This one line *could* have been written as two:

- Declaration: `int length;` At run time, the computer goes to a location of memory and leaves space for 32 bits. From then on, it has the address to this location through the variable name **length**. Any time that length is used (in the scope {} that it was declared), Java knows that it is of type **int**.
- Assignment: `length = 20;`. The computer goes to the address designated by **length**, and at that location, **puts** a value of 20.

Optional Reading: The Bus

Everything in the computer is **stored** in memory, but everything is **done** in the **Central Processing Unit's** (CPU's) **Arithmetic Logic Unit** (ALU). In order for the right information to get **to** the ALU at the right time, the CPU also has a **Control Unit** directing things by telling it what to do given your compiled code. The Control Unit is what gets the right things at the right time and makes sure they are in the right place. It tells the computer that a certain piece of information in **Memory** needs to be retrieved and put into a **register** for the ALU to use.

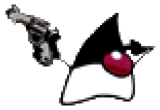
The wires that are used to transfer these bits of information (from memory to the CPU--or output devices, input devices, etc.) are called the **bus**. In very early computers, buses were as little as 8 bits wide. Now, suppose you had a variable that was of type **long**. How many times would the control unit have to go back and forth from memory to retrieve the whole variable? A variable of type `long` has 64 bits; with a bus that is 8 bits wide, we would need 64/8, or 8 trips, for one variable.

How many times would the control unit have to go back and forth from memory to get the whole variable if it was of type `byte`? A variable of type `byte` has 8 bits; with a bus that is 8 bits wide, we would need 8/8 or 1 trip for this one variable.

Which access would be faster--to get the **long** or the **byte**?

We have to make a trade-off. We want a smaller number of bits so that they can travel on the bus faster, but smaller numbers of bits do not represent numbers as large as we might want to use. Today, buses are most often 32 or 64 bits and hence the type **int** is commonly used for integers.

Alright. So now that we know what the primitive data types **are**, it's time to **use** them!



Output: Strings, Fonts, and Numbers

Lesson Objectives

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

- use classes to format numbers.
- allocate memory space.
- assign values to variables.

In the last lesson we discussed Primitive Types as well as Class types (objects). We used the Graphics object extensively in the Java 1 course. This lesson will be devoted to yet another useful type: the **String** class.

Strings

When you want to write text in your Applets and applications in Java, you use the **java.lang.String** class. Use Strings whenever you want to work with something more than one character in length.

Keep in mind that **Strings in Java are *immutable***, which means that, once a String is defined, the characters can't be changed. And **Strings are *constant*** -- once created, their values can't be changed. But despite these qualities, you can play with them all you want; you can create new Strings that are manipulations of an existing String; you just can't get into the memory location of a defined **String** and change it there.

API Find the String Class. As stated in the API, the String class includes methods for, among other things:

- examining individual characters of the sequence
- comparing strings
- searching strings
- extracting substrings
- creating a copy of a string with all characters translated to upper or lower case

Investigating Strings

String is a Java class in the **java.lang** package. When we use **String**s, we're using **instances** of the **String** Class.

Read the introduction to the **String** Class in the API. Also look at the **Method Summary** there, paying special attention to the method descriptions. The Constructor of the String Class is overloaded; it has **15** different possibilities. When we have an overloaded method, each constructor **must** have a different number and/or different types of parameters, so that Java can identify which unique constructor to use.

Strings allow us to put text into our output. In the HelloWorld Applet, you gave the command:

g.drawString("Hello World!", 5, 15);

In this code **g** indicates an instance of the **Graphics** class.

API Go to the Graphics class and find the **drawString()** method. There are two instances of the **drawString()** method here.

These are their **signatures**:

- drawString(AttributedCharacterIterator iterator, int x, int y)
- drawString(String str, int x, int y)

It appears that the **drawString()** method in **Graphics** is also **overloaded**.

Which instance of the **drawString()** method does Java use for our **g.drawString("Hello World!", 5, 15)** call?

The parameters inside the parentheses ( ) of the method call are of types: String, int, int. They direct Java to use the second instance of the **drawString()** method. Specifically, they tell Java to go to the instance **g** of Graphics and in that Graphics instance, call the method **drawString(String str, int x, int y)** by assigning the values: String str="Hello World", int x=5, and int y=15. And then, Java "draws" the text from the specified string at position (x, y) in this graphics context.

Using Strings

String is a class; therefore, we use the **new** keyword to create String objects. But, because Strings are used so much, Java provides special commands that make it easier to work with them. For example, to add text into your program, you could use the **new** command to make an instance of a **String** and then add your text. But Java has a special **myNewString** command that does the same thing:

```
String myNewString = "Java is fun";
```

This shortcut is equivilant to the Java statement:

```
String myNewString = new String("Java is fun");
```

It is important to note that **All** other rules about Java objects are the same for Strings as for other objects. There is one more special thing to know about Strings. The Java compiler is an optimizing compiler and knows that String objects are immutable. Therefore, when a String object is created with a String literal, it is inernalized with the String class **intern()** method. Any other String objects created with the same String literal will use that internalized object rather than another separate object.

For example:

```
String myFirstString = "Hello World";
```

and

```
String mySecondString = "Hello World";
```

will not create two objects. Both myFirstString and mySecondString will point to the same String object in memory.

When you print out strings, you often print out words mixed with variable values. In order to do this, you use **concatenation**, which means **appending** (adding) one String to another. Java allows us to use the plus (**+**) sign to do this.

Now, let's play with some Strings!

Make a new **java2_Lesson4** project. Create a new Class that extends Applet in this project and name it **DemoStrings**.

Note Notice that this is different from previous lessons where we created Classes that were Applications.

CODE TO TYPE: DemoStrings

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

public class DemoStrings extends Applet {

    public void start(){
        resize(400,200);
    }
    public void paint(Graphics g) {
        int length = 200;
        int width = 400;

        g.drawString("The area of this window is the length times the width", 10
, 20);
        g.drawString("Our width is " + width +  " pixels, and length is " + leng
th, 10, 40);
        g.drawString("The area of this rectangle is " + (width * length), 10, 70
);
    }
}
```

Save and **Run** it.

In drawString, everything that comes **before** the first comma must be part of a single String. That String may or may not include concatenation. Concatenation is indicated by the presence of the plus (**+**) sign. The quotation marks let Java know it's a string. The last two parameters of **drawString()** are the x and y coordinates that determine its location on the Applet. Here's an example:

g.drawString("Our width is " + width + " pixels, and length is " + length , 10, 40);

Again, we left a space after the word "is" before the closing quotation marks in the String to make our output easier to read. If we didn't leave that space, our output would look like this:

```
Our width is400pixels, and length is200
```

In Java you can use math expressions and they are automatically cast into Strings. The plus sign (**+**) concatenates the String output to the **result** of the arithmetic operation.

g.drawString("The area of this rectangle is " + (width * length) , 10, 70);

Let's try another example:

CODE TO TYPE: DemoStrings

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

public class DemoStrings extends Applet {

    public void start(){
        resize(400,200);
    }

    public void paint(Graphics g) {
        int length = 200;
        int width = 400;

        g.drawString("The area of a rectangle is the length times the width", 10
, 20);
        g.drawString("Our width is " + width +  " and length is " + length, 10,
40);
        g.drawString("The perimeter of this rectangle is " + 2*width + 2*length,
 10, 70);
    }
}
```

Save and **Run** it. Look at the output carefully. The answer should be **1200**, but it isn't. The first parameter of the **Graphics drawString()** method is a **String**. When Java sees a plus sign (**+**) within something that is supposed to be a **String**, it concatenates rather than adding.

Java did just what we told it to do--it multiplied the width by 2, and made the result (800) a **String**, because that is what the parameter required. It then multiplied length by 2, and made the result **(400)** a **String**, because that's what the parameter required. Then it concatenated these two Strings and got a result of **800400**. But that's not the result we want. Fortunately, Java understands the precedence rule of "parentheses first," so we can fix this error pretty easily if we change our code so it looks like this:

CODE TO TYPE: DemoStrings

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

public class DemoStrings extends Applet {

    public void start(){
        resize(400,200);
    }

    public void paint(Graphics g) {
        int length = 200;
        int width = 400;

        g.drawString("The area of a rectangle is the length times the width", 10
, 20);
        g.drawString("Our width is " + width +  " and length is " + length, 10,
40);
        g.drawString("The perimeter of this rectangle will be " + (2*width + 2*l
ength), 10, 70);
    }
}
```

Manipulating Strings

We can probably come up with something a little more interesting than arithmetic for examples of **Strings**.

Edit the **DemoStrings** Class's **paint()** method as shown in **blue**.

CODE TO EDIT: DemoStrings

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

public class DemoStrings extends Applet {

   public void paint(Graphics g) {
       int y=15;
       String str = "Java is Hot";              // normal String
       g.drawString(str, 10, y*1);
       String modify = str.replace("v","bb" ).replace('o', 'u').replace("is", "t
he");
       g.drawString(str.substring(0,7) + " still" + str.substring(7,11), 10, y*2
);
       g.drawString("But used to make: ", 10,y*4);
       g.drawString(modify + "t", 10, y*5);
   }
}
```

Save and **Run** it. Compare the methods in the code to the output. We can manipulate the defined instance **str** of **String**, but in doing so, we make a new String. The original **String** remains the same.

Take a look at the **replace()** and **substring()** methods of **String()** in the API. Let's try out some of these other methods. Edit DemoStrings as shown below:

CODE TO TYPE:

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

public class DemoStrings extends Applet {

   public void paint(Graphics g) {
       int y=15;
       String str = "Java is Hot";
       g.drawString(str, 10, y*1);

       String java = str.substring(0,6);
        g.drawString("The method substring - from index 0 to 6 gives:  " + java,
 10, y*2);

        int len = "HotJava".length();
        g.drawString("The length of the string HotJava is:  " + len, 10, y*4);

        String obj = new String("A String is an Object");
        g.drawString(obj, 10, y*6);              // seeing what it prints when it
 is accessed as the object

        String word = "Mississippi";
        g.drawString(word, 10, y*8);
        g.drawString(word.replace('i','a'), 10, y*9);
        g.drawString(word.toUpperCase(), 10, y*10);

        g.drawString("The original word is still: " + word, 10, y*12);
    }
}
```

Save and **Run** it. Compare the code to the output to see how methods in **String** work.

Fonts

When programming in Java, we want to be able to control the way our **String**s look. Java provides tools to change the appearance of our programs' output. The Graphics class has a setFont(Font font) method that enables us to set the font in our **Graphics** area. The Font class allows us to create the font that we want to use in its Constructor. Here's how you do that in general:

Font(String name, int style, int size)

And here's actual code we might use to create a new font object using the **Font** class's constructor:

Font bigFont = new Font("Helvetica", Font.BOLD, 20);

Let's try it. Start a new **JavaFont** Class in java2_Lesson4 (again using the java.applet.Applet superclass):

CODE TO TYPE:

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

public class JavaFont extends Applet {
 Font bigFont   = new Font("Serif", Font.BOLD, 56);
    Font smallFont = new Font("Helvetica", Font.PLAIN, 20);

    //Make a dark red for O'Reilly School of Technology
    private static final Color leafRed = new Color(115, 0, 0);

    String line1 = "JAVA";
    String line2 = "O'REILLY";

    public void paint(Graphics g)
    {
        // draw the text
        g.setColor(leafRed);
        g.setFont(bigFont);
        g.drawString(line1, 15, 100);
        g.setFont(smallFont);
        g.drawString(line2, 46, 140);
    }
}
```

Save and **Run** it and check out the results.

Now, let's change the parameter shown below in **blue**:

CODE TO TYPE: JavaFont

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

public class JavaFont extends Applet {
 Font bigFont    = new Font("Serif", Font.BOLD, 56);
    Font smallFont = new Font("Helvetica", 3, 20);

    //Make a dark red for O'Reilly School of Technology
    private static final Color leafRed = new Color(115, 0, 0);

    String line1 = "JAVA";
    String line2 = "O'REILLY";

    public void paint(Graphics g)
    {
        // draw the text
        g.setColor(leafRed);
        g.setFont(bigFont);
        g.drawString(line1, 15, 100);
        g.setFont(smallFont);
        g.drawString(line2, 46, 140);
    }
}
```

Save and **Run** it and check out the results.

API Go to the Font class in the API and look at its constructors. In our **JavaFont** class, go to the line:

Font smallfont = new Font("Helvetica", 3, 20);

Play around with it, change the int parameter values, run the code, and observe the results.

Numbers

Although Java has a **Number** class in the **java.lang** package, we won't be looking at it here. In this section we're actually interested in **formatting** numbers, so we'll investigate the **NumberFormat** class in the **java.text** package.

Start a new **NumbersDemo** class in java2_Lesson4 that again extends Applet:

CODE TO TYPE: NumbersDemo

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

public class NumbersDemo extends Applet {

    public void paint(Graphics g) {
        double area;
        double radius = 12;

        g.drawString("Area of a circle = (radius)^2*Pi", 10, 20);
        g.drawString("If radius = " + radius, 10, 40);
        g.drawString("The circle's area = " + Math.pow(radius,2)*Math.PI, 10, 70);
    }
}
```

Save and **Run** it.

Okay, but what if we don't need our results to be carried out that far, to be such long numbers? We can fix that, using the java.text.NumberFormat class to limit our output to just two decimal places. Add the **blue** code shown below:

CODE TO EDIT: NumbersDemo

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

public class NumbersDemo extends Applet {

    public void paint(Graphics g) {
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        double area;
        double radius = 12;

        g.drawString("Area of a circle = (radius)^2*Pi", 10, 20);
        g.drawString("If radius = " + radius, 10, 40);
        g.drawString("The circle's Area = " + nf.format(Math.pow(radius,2) * Math.PI),
10, 70);
    }
}
```

Save and **Run** it.

Ahh, much better--only two decimal places. To learn more about NumberFormat, go to the NumberFormat class in the **java.text** package in the API.

You could accomplish the same task using the **DecimalFormat** Class, located in the **java.text** package. It's actually a **subclass** of **NumberFormat**, which indicates that it will perform more specific actions. A **DecimalFormat** comprises a pattern and a set of symbols that you define so that the number in your output appears just the way you intend it to be.

Edit the **NumbersDemo** class as shown here in **blue**:

CODE TO EDIT: NumbersDemo

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

public class NumbersDemo extends Applet {

    public void start(){
        resize(300,200);
    }

    public void paint(Graphics g) {
        DecimalFormat myFormat1 = new DecimalFormat("###,###.##");
        DecimalFormat myFormat2 = new DecimalFormat("###,###");
        double area;
        double radius = 12;

        g.drawString("Area of a circle = (radius)^2*Pi", 10, 20);
        g.drawString("If radius = " + radius, 10, 40);
        g.drawString("The circle's area = " + myFormat1.format(Math.pow(radius,2)*Math.
PI), 10, 70);
        g.drawString("An alternate formatting without decimals ", 10, 95);
        g.drawString("The circle's area= " + myFormat2.format(Math.pow(radius,2)*Math.P
I), 10, 110);
    }
}
```

Save and **Run** it. Compare the output to the code.

API Go to the DecimalFormat class in the **java.text** package in the API to find out more.

Although this looks pretty good, consider a simple edit. Let's say we want to represent money, so we want exactly two decimal places again. Edit the **NumbersDemo** class by adding the lines shown in **blue**:

CODE TO EDIT:

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

public class NumbersDemo extends Applet {

    public void paint(Graphics g) {
        DecimalFormat myFormat1 = new DecimalFormat("###,###.##");
        DecimalFormat myFormat2 = new DecimalFormat("####,###");
        double area;
        double radius = 12;
        double amountOwed = 12.00;

        g.drawString("Area of a circle = (radius)^2*Pi", 10, 20);
        g.drawString("If radius = " + radius, 10, 40);
        g.drawString("The Circle's Area = " + myFormat1.format(Math.pow(radius,2)*Math.
PI), 10, 70);
        g.drawString("An alternate formatting without decimals ", 10, 95);
        g.drawString("The Circle's Area = " + myFormat2.format(Math.pow(radius,2)*Math.
PI), 10, 110);
        g.drawString("The money owed is " + myFormat1.format(amountOwed), 10, 130);
    }
}
```

Now **Save** and **Run** it and compare the output to the code.

The number was formatted, but since there were no values after the decimal, the given format left the digits off entirely. But in some cases, we want the extra digits even if they are zeros.

We can make Java do **exactly** what we want. We can. We will. We have the power!

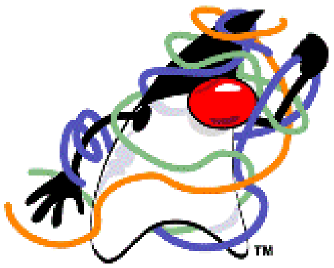
Edit the **NumbersDemo** Class by adding the two lines as shown below in **blue**:

CODE TO EDIT:

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

public class NumbersDemo extends Applet {

    public void paint(Graphics g) {
        DecimalFormat myFormat1 = new DecimalFormat("###,###.##");
        DecimalFormat myFormat2 = new DecimalFormat("####,###");
        DecimalFormat df1 =  new DecimalFormat("####.00");

        double area;
        double radius = 12;
        double amountOwed = 12.00;

        g.drawString("Area of a circle = (radius)^2*Pi", 10, 20);
        g.drawString("If radius = " + radius, 10, 40);
        g.drawString("The circle's area = " + myFormat1.format(Math.pow(radius,2)*Math.
PI), 10, 70);
        g.drawString("An alternate formatting without decimals ", 10, 95);
        g.drawString("The circle's area = " + myFormat2.format(Math.pow(radius,2)*Math.
PI), 10, 110);
        g.drawString("The money owed is " + df1.format(amountOwed), 10, 130);
    }
}
```

Save and **Run** it and compare the output to the code.

This time, rather than explain it to you ourselves, we'll let the API do it. You know where to go! Have fun and good luck!



Arithmetic Operations: The Basics

Lesson Objectives

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

- prompt the user for input and return information based on that input.
- use arithmetic operators.
- use increment statements.
- process operators in the proper order.

Calculations and Operators

In a previous lesson, we talked about **statements**. In this lesson, we'll look more closely at a particular type of statement known as an **expression**. We'll also look at the operators that Java provides and use valid syntax to perform desired calculations. Java is a *high-level language* (a language that's closer to human language than computer language). Because of this, it enables us to write expressions in a simple form without having to consider everything the computer does to perform the calculations. We'll leave that to the *low-level languages.*

Variables and a Program to Dissect

Let's get started! Create a **temp** project, and within that project, create an Applet named **Calculation**. Now type in the following code:

CODE TO TYPE: Calculation

```
// calculate area of rectangle
import java.awt.*;
import java.applet.Applet;

public class Calculation extends Applet {
    public void paint(Graphics g) {
        int length;       //declarations
        int width;
        int area;

        length = 20;      // assignments
        width = 10;
        area = length * width;
        g.drawString("Area is " + area, 100, 100);
            // display the answer
    }
}
```

Save and **Run** it.

Let's trace the program to see exactly what's happening in the **paint()** method:

1. We declared three variables as type **int**:
   - int length;
   - int width;
   - in area;
2. We assigned a value of 20 to the variable **length**. The computer goes to location of "length" and, at that location, **puts** a value of 20.
3. We assigned a value of 10 to the int **width** (**put** 10 at location "width").

4. The computer actually performs several steps in the line **area = length * width;**:
    1. It goes to variable **length**'s location in memory, **get** its value (20), and send it back on the bus to the CPU.
    2. It goes to variable **width**'s location in memory, **get** its value (10), and send it back on the bus to the CPU.
    3. It Multiplies the values it just got (20 * 10) in the CPU's Arithmetic Logic Unit (ALU).
    4. It goes to variable **area**'s location in memory on the bus, and **put** its calculated value (200).
5. It tells the Graphics area **g** to apply **drawString()** to the String "Area is ".
6. It **gets** the value (200) at location area. It **concatenates** (+) this to the "Area is" String and gets: Area is 200.
7. It prints this result at the specified location (100,100) on the Graphics area of the Applet.

While we're here, let's try an alternative declaration of the variables. Change the class as shown below in blue:

CODE TO EDIT: Calculation

```
// calculate area of rectangle
 import java.awt.*;
 import java.applet.Applet;

 public class Calculation extends Applet {
     public void paint(Graphics g) {
        int length, width, area;         //declarations

        length= 20;      // assignments
        width = 10;
        area = length * width;
        g.drawString("Area is " + area, 100, 100);
             // display the answer
      }
 }
```

Save and **Run** it. It should work exactly the same way.

You can declare variables of the same type all at once like this, but it's good programming style to group related variables together, while keeping different variables on separate lines.

Before we leave this example, let's look at the multiplication line once more:

area = length * width;

Note To perform multiplication in Java, you *must* use *.

In Java, you **cannot** use **x** to indicate multiplication and you **cannot** simply omit the operator, as you would in mathematics, substituting **6y** for **6 times y**. Nor can you use parentheses, as in **y(6)**, to indicate multiplication. In Java, parentheses may indicate precedence, as in y * (y+1), where **y+1** would be calculated first and the result multiplied by **y**. Also, keep in mind that Java uses parentheses in methods as well.

Change the Calculation class as shown below:

CODE TO EDIT: Calculation

```
// calculate area of rectangle
import java.awt.*;
import java.applet.Applet;

public class Calculation extends Applet {
    public void paint(Graphics g) {
        int length, width, area;        //declarations

        length= 20;      // assignments
        width = 10;
        area = length(width);
        g.drawString("Area is " + area, 100, 100);
             // display the answer
    }
}
```

The method **length(int)** is undefined for the type **Calculation**. Java sees parentheses **( )** after the word **length**. The compiler knows only that which is allowed. Because the only allowed syntax that uses parentheses ( ) are Methods and decision comparisons, Java thinks **length(width)** is a Method call. Java knows that width is an int, so it interprets the line `area = length(width)` as a call to a Method named length, with an int parameter named width. But there is no such method, so Java tells you that.

Change the line **area = length * width;** to **area = length x width;**

Hmm, we've got lots of error messages. The most pressing error message is **x cannot be resolved**. Java thinks **x** is a variable name rather than your intended multiplication operator.

Change the line out to:

area = 6width;

There's an error marker on the right and red zigzags under width:

Syntax error on token "width", delete this token

You can see right away that we won't be using statements like this to indicate multiplication. Change the line back to **area = length * width;** again and **Save** the Calculation class.

I'm confident that you now know how to use the division (**/**), addition (**+**), and subtraction (**-**) operators. We'll go over the order in which they are performed in the next lesson.

Defaults

Sometimes when you're defining a class's Fields (IVs and CVs), you know what the variable's value is going to be before anyone uses the class. Usually this is because that particular variable's initial value is always the same. It is often called the **default value**. When the value is known, it is usually specified at the time the variable is declared. Here, the variable **pi** is being given a default **float** value of **3.14f**:

float pi = 3.14f;

Actually, though, the choice of the math constant **pi** is not an altogether great example for us to use to define default variables because it already **is** a default **Class Variable** in the Class **Math**, located in the **java.lang** package.

API Go to the java.lang package and look at the Field Summary of the Math class.

The Math class gives **PI** a default value, and also makes it a **constant**. We discussed **constants** in Java in the first course in the series, but let's discuss them a bit more now. Java defines constants by using the **modifiers** of **public static final**:

- **public**: anyone can access it.
- **static**: they can access it through the class name.
- **final**: it cannot be changed.

Because it's a **Class Variable**, it can be accessed through the **Math** Class; that is, we can access it by saying **Math.PI**. Because it's `public`, we can access it from any class. Because it's in `java.lang`, we don't need to `import` anything in order to use it.

Let's see the Math class in action! Edit the Calculation class as shown in **blue**:

CODE TO EDIT: Calculation

```
// calculate area of circle
import java.awt.*;
import java.applet.Applet;

public class Calculation extends Applet {
    // make room for display
    public void start(){
        resize(300,200);
    }

    public void paint(Graphics g) {
     int radius;                      //declarations
     double area;

     radius = 50;                     // assignments
     area = Math.PI * Math.pow(radius,2);

     g.drawOval(10,10,radius, radius);
     g.drawString("My circle's area is " + area, 10, 100);
    }
}
```

Save and **Run** it.

Also, look at the pow(double a, double b) method of the Math Class; check out our call of Math.pow(radius,2) and see what it does.

If you do not specify values in the definition of your class for your **Instance and Class Variables**, Java will give them default values. That is, **Fields** of a Class (its Instance and Class Variables) have default values on instantiation:

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

WARNING

Local variables differ from Instance and Class Variables when it comes to assigning default values. The compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable when it is declared, be sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error. (Don't worry, though; Eclipse will catch it for you.)

To illustrate this, change the **Calculation.java** class's **paint()** method by commenting out the local variable setting for radius. We can just comment it out because we know we will put it back.

// radius = 50; // assignment

This leaves **radius** without a value. Eclipse tells you that:

The local variable radius may not have been initialized

You can try, but you can't run the program correctly until you give radius a value--you get all kinds of errors in the console and an empty Applet.

Uncomment the local variable setting and **Save** the class. I hope you appreciate how Eclipse watches out for you!

Added Attractions

Let's try more expressions using our Test.java class again.

Increment Statements

Programmers like shortcuts. Here are a few examples of ways to increment a variable by 1.

Edit the **main ()** method as shown below:

CODE TO EDIT: Test

```
public class Test {
    public static void main(String args[])
    {
        int n = 10;
        System.out.println("After declaring and setting to 10, n is " + n);

        n = n + 1;
        System.out.println("After n = n + 1, n is " + n);

        n++;      // an increment expression - if the + is post (after the varia
ble) then it adds after it uses the variable.
        System.out.println("After n++, n is " + n);
        System.out.println("Seeing n++ WITHIN the println command, Java first US
ES it: " + n++);
        System.out.println("After Java uses it in the statement, it then increme
nts");
        System.out.println("So without doing anything to n, n is now " + n);

        n += 1;  // essentially the same as n = n + 1;
        System.out.println("After n += 1, n is " + n);
    }
}
```

Save and **Run** it as an application.

An **increment expression** has the side effect of incrementing the value of the variable by 1. That is, it "uses" the variable **and** increments it as well.

The location of the **++** operator matters:

- If the operator **++** is a suffix (i++), Java **uses the variable first** and then increments the value.
- If the operator **++** is a prefix (++i), Java **increments the variable value first** and then uses the incremented value.

Use the increment expression sparingly to reduce confusion (when it's used, it's often only to increment loop variables). Let's try more.

Edit the **Test.java** class's **main ()** method as shown.

CODE TO EDIT: Test

```
public class Test {

    public static void main(String[] args) {
     int i = 0;
     System.out.println("After declaring and setting to 0, i is " + i);
     System.out.println();

     int j = i++;
     System.out.println("After j = i++, i is " + i + " and j is " + j);
     System.out.println("  Java USED the i FIRST by setting j to its value of 0"
);
     System.out.println("  THEN it incremented i by 1");
     System.out.println();

     j = ++i;   // if the + is pre (before) then it adds before it uses
     System.out.println("After j = ++i, i is " + i + " and j is " + j);
     System.out.println("  Java incremented the i first by 1, making its value 2
");
     System.out.println("  THEN it USED it by giving its value to j");
    }
}
```

Save and **Run** it as an application.

In the console, compare the output and the code to see how the code works--I know you're curious!

Some other incrementing shortcuts:
i *= 3; is the same as i = i * 3;
i--; is the same as i = i - 1;

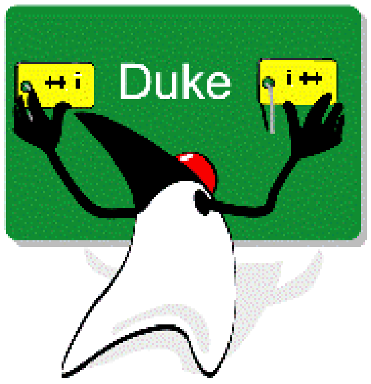
Add the following lines to see what happens (make sure you are between the braces **{}** in the **main()** method):

CODE TO EDIT: edit main

```
public static void main(String[] args) {
 int i = 1;
  System.out.println();
  System.out.println("i is " + i);
 i +=4;
  System.out.println("After i +=4, i is " + i);
  System.out.println();
 i *=3;
  System.out.println("After i *=3, i is " + i);
  System.out.println();
 i--;
  System.out.println("After i--, i is " + i);
  System.out.println();
 --i;
  System.out.println("After --i, i is " + i);
  System.out.println();

  System.out.println("When first using --i, we get " + --i + " and now that it w
as used i is  " + i);
  System.out.println("When first using i--, we get " + i-- + " and now that it w
as used i is  " + i);

}
```

Save and **Run** it.

Be careful with these; they can get you in trouble if used improperly.



You're doing great so far. See you in the next lesson!

Arithmetic Operations: Precedence and Operand Types

Lesson Objectives

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

- use concatenation and arithmetic operators through calls to drawString().
- use various integer operations in the proper order.

Executing Tasks in the Correct Order

Operators and Precedence

Operators and Precedence of operators are as important in computer languages as they are in mathematics. In elementary school you might have learned precedence using the mnemonic "**M**y **D**ear **A**unt **S**ally," which stands for **M**ultiplication, **D**ivision, **A**ddition, **S**ubtraction. The same order of precedence applies to computers, with a couple of additional operators; arithmetic operations are performed on computers in this order:

1. ( )
2. * / %
3. + -

First, operators in parentheses are executed:

4 + 3 * 2 = 10, because * has precedence over +, **BUT**
(4 + 3) * 2 = 14, because () has precedence over *, **so** (4 + 3) * 2 = (7) * 2 = 14

then, *, /, and % operators are done left to right:

4 % 3 * 2 = 2 = (4 % 3) * 2 = 1 * 2 = 2, **BUT**
4 % (3 * 2) = 4, because () has precedence over %, **so** 4 % (3 * 2) = 4 % 6 = 4

then, + and - operators are done left to right:

6 + 3 * 4 / 2 - 9,

Let's apply precedence rules to the first expression below and follow its path:

((6 + ((3 * 4) / 2)) - 9) =

((6 + ((12) / 2)) - 9) =

((6 + ((6))) - 9) =

((((12))) - 9) = 3

There are actually more precedence rules than those we've mentioned so far, but we'll just work with these for now.

So, can you figure out the result of this expression?: (6 + 3) * 4 / 2 - 9

Remainder

"%" stands for **remainder**, that is, what is left after you divide:
7 % 2 = 1, because 7/2 is 3, with a remainder of 1
6 % 2 = 0, because 6/2 is 3, with a remainder of 0
16 % 6 = 4, because 16/6 is 2, with a remainder of 4

Let's try using **%** in Java. Open the **Test.java** class in our **temp** testing project, and edit it as shown in **blue** below:

CODE TO TYPE: testing %

```
public class Test {
 public static void main(String[] args) {
     int rem = 2;
     int i = 21;
     System.out.println("After declaring and setting to 21, i is " + i);
     System.out.println();
        int j = i % rem;
     System.out.println("When i is " + i + " and j is i % " + rem + ", the value
 of j is " + j);
     --i;
     j = i % rem;
     System.out.println("When i is " + i + " and j is i % " + rem + ", the value
 of j is " + j);
     --i;
     j = i % rem;
     System.out.println("When i is " + i + " and j is i % " + rem + ", the value
 of j is " + j);
     --i;
     j = i % rem;
     System.out.println("When i is " + i + " and j is i % " + rem + ", the value
 of j is " + j);
     --i;
     j = i % rem;
     System.out.println("When i is " + i + " and j is i % " + rem + ", the value
 of j is " + j);
   }
}
```

Save and **Run** it.

In the Test.java class, change the value of rem to 3 like this: **int rem = 3;**.

Save and **Run** it. When you use **%** 2, the remainders are always 0 or 1.. When you use **%** 3, the remainders are 0, 1, or 2.

What will the remainders be with **%** 4? Test it by changing the value of rem to 4: **int rem = 4;**. **Save** and **Run** it. The result makes perfect sense. If you divide by a number, the remainder will be all numbers **between 0 and the number - 1**, because a remainder can never be larger than the number by which you are dividing.

Suppose you want to calculate: **a number % n**. The possible results for different values of **n** would look like this:

n	Possible Remainders
2	0,1
3	0,1,2
4	0,1,2,3
5	0,1,2,3,4
...	...
n	0,1,2,3,4, ... , n-1

The % function is also called modulo, or **mod**.

Precedence

Let's verify our precedence examples from above, while Java helps you remember some algebra in the process.

Edit the **Test** class as shown in **blue** below:

CODE TO TYPE: precedence verification

```
public class Test {
    public static void main(String[] args) {
        int x;
        x = 4 + 3 * 2;
        System.out.println("When x = 4 + 3 * 2 the answer is " + x);
        System.out.println();
        x = (4 + 3) * 2;
        System.out.println("When x = (4 + 3) * 2 the answer is " + x);
        System.out.println();
        x = 4 % 3 * 2;
        System.out.println("When x = 4 % 3 * 2  the answer is " + x);
        System.out.println();
        x = 4 % (3 * 2);
        System.out.println("When x = 4 % (3 * 2) the answer is " + x);
        System.out.println();
        x = 6 + 3 * 4 / 2 - 9;
        System.out.println("When x = 6 + 3 * 4 / 2 - 9 the answer is " + x);
        System.out.println();
        x = (6 + 3) * 4 / 2 - 9;
        System.out.println("When x = (6 + 3) * 4 / 2 - 9 the answer is " + x);
    }
}
```

Save and **Run** it.

Tip When you aren't sure about the effect a particular Java function will have on your program, you can test it in a small program like we did here.

Operand Types

When you divide integers by integers, you **get** integers, not fractions. Java **truncates** the result of integer division so that the results are always integers as well. For example:

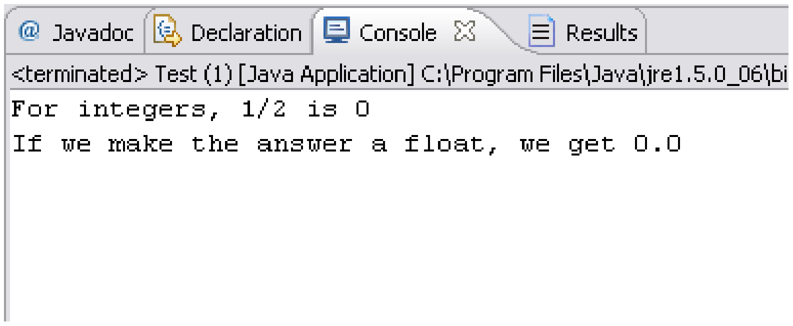
- 1 / 2 is 0--the answer (0.5) is **truncated** to 0.
- 16 / 5 is 3--the answer (3.2) is **truncated** to 3.
- 11 / 2 is 5--the answer (5.5) is **truncated** to 5.

Let's test the above examples. Edit the **Test** class as shown below:

CODE TO TYPE: Integer Division

```
public class Test {
    public static void main (String args[]) {
        int b = 1;
        int c = 2;
        float d = b/c;
        System.out.println("For integers, " + b + "/" + c + " is " + b/c);
        System.out.println("If we make the answer a float, we get " + d);
    }
}
```

Save and **Run** it.



```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.5.0_06\bi
For integers, 1/2 is 0
If we make the answer a float, we get 0.0
```

Zero? What the heck?

You told Java that **b** was **1** (an int) and **c** was **2** (an int), and Java thinks you mean it. Then, in the first **System.out.println**, you give **b/c**, which is **1/2**, which is 0.5--since both are ints, Java truncates the result to **0**.

In the statement **float d = b/c;**, Java sees that **d** is declared to be a float (decimal number) so Java says to itself, "Ok, I will leave space for a float." Then, on the right side, it sees **b/c** (that is, **b** divided by **c**), so Java performs that task. Since **b** is **1** and **c** is **2**, the result is **1/2** and since both are integers, Java truncates to **0**. **But** since you told Java to enter the answer in a place that is a **float**, Java **casts** the integer answer to a float. That means it took the integer answer and gave it a float (decimal) value. **But**, the integer answer it sees is **0**, so it gives you **0.0** Java always does exactly what you tell it to do (so long as it can understand your instructions).

Let's experiment in our Applet testing class. In the **temp** Project, edit the **Calculation.java** class as shown below. Also remove the **radius = 50;** line.

CODE TO TYPE:

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

public class Calculation extends Applet {

    public void paint(Graphics g) {
     int diameter = 50;
     int radius = 1/2 * diameter;
     double area;

     area = Math.PI * Math.pow(radius,2);

     g.drawOval(10,10,radius, radius);
     g.drawString("My circle's area is " + area, 10, 100);
 }
}
```

Save and **Run** it.

So what happened? Our result is an area of 0.0 and nothing is drawn. It seems we made a very common mistake. It's located on this line:

int radius = 1/2 * diameter;

In Java, 1/2 times **anything** will be 0, because 1/2 is integer division and 0.0 times anything is 0. So in our experiment, nothing is drawn, and we get an area of 0.0. So what does the following yield?:

int half = 20000*(1/2)

We can fix it. There are a couple of ways to make it work; here's one of them:

Comment out the line **int radius = 1/2 * diameter;**, so Java doesn't see it when it tries to run. Everywhere

radius appears in the code, change it to **diameter/2** as shown below:

CODE TO EDIT:

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

public class Calculation extends Applet {

    public void paint(Graphics g) {
        int diameter = 50;
        // int radius = 1/2 * diameter;
        double area;

        area = Math.PI * Math.pow(diameter/2,2);

     g.drawOval(10,10,diameter/2, diameter/2);
        g.drawString("My circle's area is " + area, 10, 100);
    }
}
```

Save and **Run** it.

Whoopee! It works. That's because we used **diameter/2** *within* the actual parameters for **Math.pow** and **g.drawOval**.

Tip **The Role of Expressions**: Anywhere you can put an integer, you can put an integer expression.

This means that if I can put an integer somewhere, I can also put something that **evaluates** to an integer. The same is true for other types as well. Expressions always evaluate to something in Java, so to Java, typing in an expression is just like typing the result of the expression in the same space.

Type Conversion/Casting

Java can temporarily *convert an integer into a String*. This is called **casting**. Java automatically casts for us when we call methods with different parameters from those specified.

Consider the method **Math.pow()**. We invoked it with Math.pow(radius,2) where radius and 2 are **int**s in our code earlier:

public static double pow(double a,double b)

It can also can be called with Math.pow(4,2).

The idea here is that no information is "added"--that is, if we have the number 2 then it does not hurt to make it 2.0 for computation in the method. Temporarily using a number with decreased precision for the purpose of performing the method or operation, then converting it to a more precise number, does not alter the expression's accuracy.

Programmers can perform **type conversion** or **casting** too.

When an operator manipulates a *mixture* of **int** and **float** (or **double**) values, any integers are temporarily converted to **float** for the purpose of the calculation. For example:

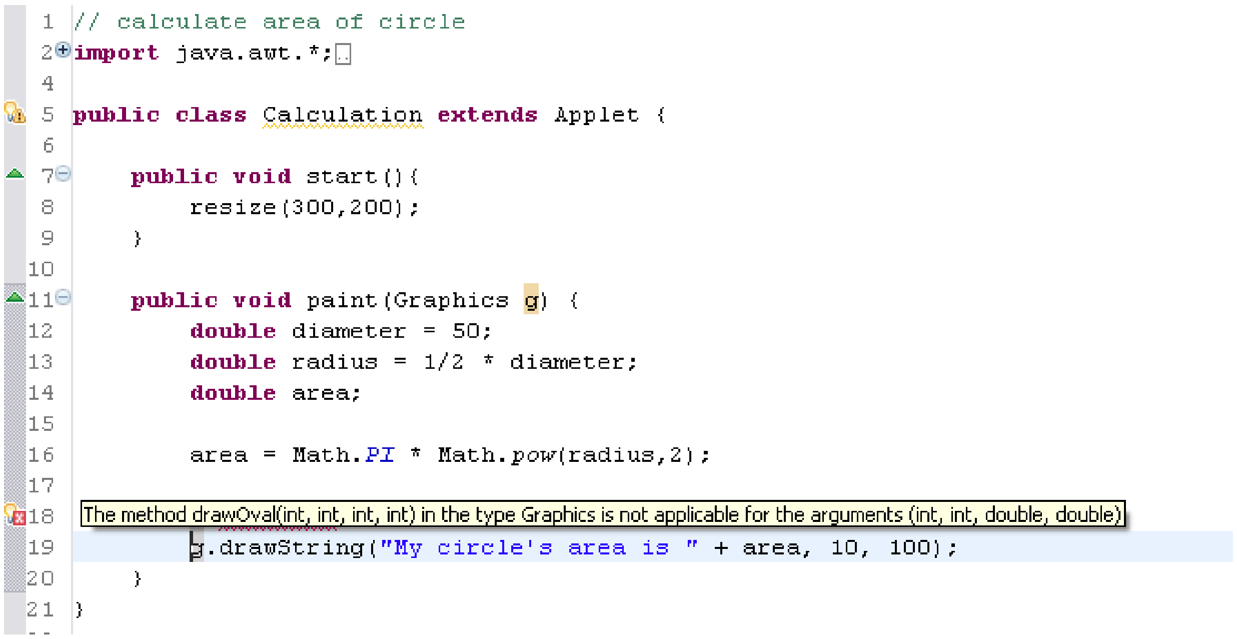
7 / 2.0 => 7.0 / 2.0 => 3.5

In Java, if **one** of the variables in an expression is a decimal (float or double), then it performs **automatic casting** and makes the whole expression a float or double.

Now let's change the **Calculation.java** class **back** to what it was (when it did not work), but let's suppose that we need more precise variables, so we're going to set them to **doubles**:

CODE TO EDIT: Calculation

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

public class Calculation extends Applet {

    public void paint(Graphics g) {
     double diameter = 50;
     double radius = 1/2 * diameter;
     double area;

     area = Math.PI * Math.pow(radius,2);

     g.drawOval(10, 10, radius, radius);
     g.drawString("My circle's area is " + area, 10, 100);
    }
}
```



```
// calculate area of circle
import java.awt.*;

public class Calculation extends Applet {

    public void start(){
        resize(300,200);
    }

    public void paint(Graphics g) {
        double diameter = 50;
        double radius = 1/2 * diameter;
        double area;

        area = Math.PI * Math.pow(radius,2);

The method drawOval(int, int, int, int) in the type Graphics is not applicable for the arguments (int, int, double, double)
        g.drawString("My circle's area is " + area, 10, 100);
    }
}
```

Let's fix the first error we see at our call to drawOval. Our call to drawOval is not happy because our **actual parameters** do not match our **formal parameters** in the actual call: g.drawOval(10, 10, radius, radius); formal params: drawOval(int x, int y, int width, int height).

x and y are set to 10, and 10 is an int, so that's OK; width and height are set to the value of radius--but radius is a **double**, and the method specification states **int**.

No problem. We'll do our own "forced" conversion. We'll tell Java that, just for now, we want it to look at radius as if it were an int:

CODE TO EDIT: Calculation

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

public class Calculation extends Applet {

    public void paint(Graphics g) {
        double diameter = 50;
        double radius = 1/2 * diameter;
        double area;

        area = Math.PI * Math.pow(radius,2);

        g.drawOval(10, 10, (int)radius, (int)radius);
        g.drawString("My circle's area is " + area, 10, 100);
    }
}
```

Save and **Run** it. **Rats.** We got all zeroes. It looks like we still have the 1/2 giving us the result of 0.

Another way to "force" a cast is to take a specific number and change it to the desired type. For example, make the 2 in 1/2 a 2.0 (or the 1 a 1.0). If Java sees a decimal in the expression, it will perform the whole operation as a decimal.

Change the method as shown in **blue**:

CODE TO EDIT: Calculation

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

public class Calculation extends Applet {

    public void paint(Graphics g) {
        double diameter = 50;
        double radius = 1/2.0 * diameter;
        double area;

        area = Math.PI * Math.pow(radius,2);

        g.drawOval(10, 10, (int)radius, (int)radius);
        g.drawString("My circle's area is " + area, 10, 100);
    }
}
```

Save and **Run** it. Sweet!

There are rules governing which types can be converted to which other types. The general idea is to prevent any **loss** of accuracy. For example, you wouldn't take an **int** (which can have 32 bits in memory) and cast it "down" to a **byte** (which has only 8 bits in memory). If we did cast an int to a byte, all but the lowest bits of data would be discarded in the conversion. By dropping the higher bits of data, you might lose valuable information and the precision of the original value would be diminished.

WARNING Larger types cannot be cast to smaller types without potential information loss.

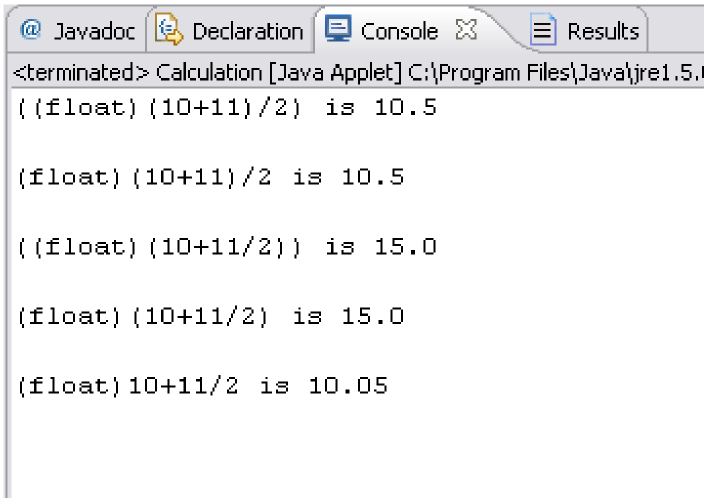
For more information, see Legal Conversions.

Edit the **Test.java** class as shown below:

CODE TO EDIT: Test

```
public class Test {

    public static void main (String args[]) {
        System.out.println("((float)(10+11)/2) is " + ((float)(10+11)/2));
        System.out.println();
        System.out.println("(float)(10+11)/2 is " +  (float)(10+11)/2);
        System.out.println();
        System.out.println("((float)(10+11/2)) is " +  ((float)(10+11/2)));
        System.out.println();
        System.out.println("(float)(10+11/2) is " +  (float)(10+11/2));
        System.out.println();
        System.out.println("(float)10+11/2 is " +  (float)10+11/2);
    }
}
```

Save and **Run** it. Compare the code to the results to see if it is what you expected:



```
<terminated> Calculation [Java Applet] C:\Program Files\Java\jre1.5.
((float)(10+11)/2) is 10.5

(float)(10+11)/2 is 10.5

((float)(10+11/2)) is 15.0

(float)(10+11/2) is 15.0

(float)10+11/2 is 10.05
```

In the first two expressions we can see that the outermost parentheses are not necessary. But comparing the second and third expressions, we see that the parentheses indeed **do** make a difference there: (float)(10+11)/2 makes 21 a float and then divides 21.0 by 2 and gets 10.5--**but** (float)(10+11/2) does all the int arithmetic (note precedence, so parentheses first and in the parentheses, division first), gets a result of 10+5 or 15, and **then** makes it a float--15.0.

Since the parameter is a String for the drawString method, Java sees the plus sign **+** as concatenation. Thus, it casts the integers as Strings to concatenate them into one String for the parameter.

(float)10+11/2 first makes 10 a float (following parentheses and left-to-right precedence), and so we get 10.0, then it does the integer arithmetic for 11/2 and gets 5; **Then**, it **concatenates** the two numbers (10.0 and 5) to get 10.05, because the plus sign **+** is in a place where the parameter is a **String**.

The following code results in only 3 possibilities: 0, 1 or 2:

```
rint = (int)(Math.random() * 3);
```

Look up **java.lang.Math** and its method **random()** in the API. Using information you find there, try to figure out why the only possible results in this case are 0, 1 or 2.

Math.random( ) returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. If we multiply this by an integer (n), we get a number greater than or equal to 0.0 and **less than n**. So if the number is 3, we get a number greater than or equal to 0.0 and **less than** 3.0 (for example 2.999999). Now, we cast that result to an (int) and ultimately we get 0, 1, or 2, because the cast will truncate the number and leave us with only those three possibilities.

If we had multiplied the random number by 16, we would get a number from 0 to 15. How would we change it to get a number from 1 to 16?

Easy: **rint = (int)(Math.random() * 16) + 1 ;**

More on String Concatenation

We learned earlier in the lesson that, when the parameter is a **String**, the plus sign **+** signifies concatenation. The same happens in the following line:

g.drawString("Area is " + area, 100, 100);

but Java first converts the area number into a String, then concatenates it.

Consider the differences between these lines:

g.drawString("Answer is " + 1 + 2, 100, 120);

and

g.drawString("Answer is " +(1+2), 100, 130);

Put them into your Calculation.java applet and run it.

Formats

Here's an example that demonstrates a way to control the number of decimal places your numbers display:

Edit the **Calculation.java** class to make the circle again. Use the code below so that it shows only 2 decimal places:

CODE TO TYPE: java.text.NumberFormat

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

public class Calculation extends Applet {

   public void paint(Graphics g) {
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);

        double diameter = 52;
        double radius = 1/2.0 * diameter;
        double area;

        area = Math.PI * Math.pow(radius,2);

        g.drawOval(10,10, (int)radius, (int)radius);
        g.drawString("My circle's area is " + nf.format(area), 10, 100);
   }
}
```

Save and **Run** it.

The class **NumberFormat** in the **java.text** package also puts commas in the number.

Another formatting class is **java.text.DecimalFormat**. Feel free to go to the API and check it out!

For reference, here is a link to Oracle's online tutorial and the Java Language Specification book chapter on Types, Values and Variables.

Decisions: Logic

Lesson Objectives

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

- evaluate each dice roll and use logic statements and flow control properly.
- use logic operators and expressions.

In the Java 1 course we learned about **if statements**, which use booleans (true or false) to make decisions. In this lesson, we'll dig a little deeper and use logic to decide when expressions are true or false.

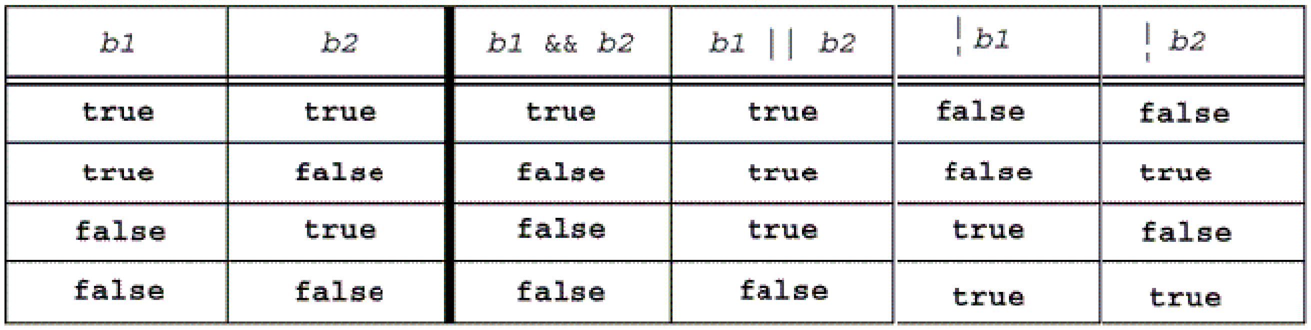
Logic: and, or, not

Java provides these three Logic Operators:

&& - *and*
|| - *or*
! - not

While other languages may allow results other than **boolean** in comparisons, Java *only* allows the result of **true** or **false**. The only allowable data type that can result from the condition of an **if** statement is a **boolean**, so all comparisons in Java must result in either **true** or **false**.

In the figure below, suppose that b1 and b2 are statements. The truth values of the logic expressions using **and**, **or**, and **not** are as shown in the table below:

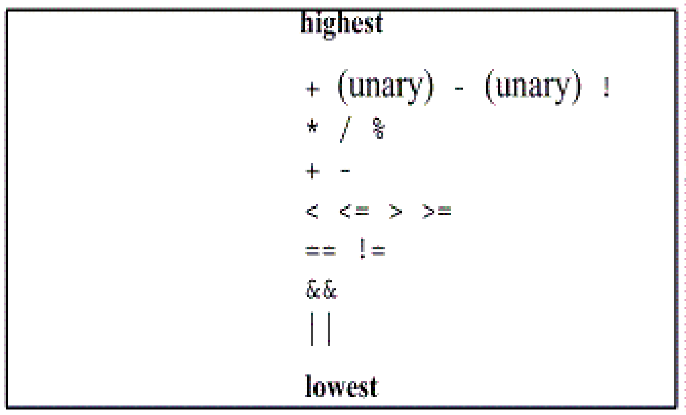


b1	b2	b1 && b2	b1 || b2	! b1	! b2
true	true	true	true	false	false
true	false	false	true	false	true
false	true	false	true	true	false
false	false	false	false	true	true

To read this table, select a row and read the values. For instance, if b1 is false and b2 is true, then b1 || b2 is true.

Here's a demo applet we made for you to use for practice.

It's interesting to compare **&&** and **||** to **true** and **false**, and even more so to use complex expressions that **evaluate** to **true** and **false**. These conditional statements are a tad more complicated to work with, so let's revisit the concept of **precedence** and make sure we have a clear plan for what to do when we encounter **Value1 || Value1 && Value3**. Below is the order in which operators are performed. If operators are on the same line in the chart below, then the compiler simply operates left to right.



highest

`+ (unary) - (unary) !`
`* / %`
`+ -`
`< <= > >=`
`== !=`
`&&`
`||`

lowest

More precedence rules exist, but we're going to cover just the most commonly used ones for now.

Examples Using Logic

OBSERVE: Getting an A

```
if (score > 90 && score < 93)
   grade = "A-";
```

Condition **operators** other than **! (not)** are binary, that is, they have two operands (something) on either side of the operator:

- something is **<** something else
- something is **==** to something else
- something is **&&** or **||** with something else

In the example above, we see that the variable **score** must be **both** > 90 **and** < 93. Thus, depending on the **type** of **score**, the values that yield **true** would be 91 and 92 if **score** is a **byte, short, int, long**, and anything **between** 90 and 93 (not including 90 or 93) if **score** is a **float or double**.

Let's test this with a simple main in an application. Open the **Test.java** class in your **temp** project folder. Edit it to hold the code shown below in **blue**.

CODE TO TYPE: Getting an A

```
public class Test {
 public static void main (String args[]) {
  int score = 96;
  String grade = "unknown";

  if (score > 90 && score < 93)
      grade = "A-";
     System.out.println("Given the input, your grade is " + grade);
 }
}
```

Save and **Run** it.

Now, change the score from 96 to 93:

CODE TO TYPE: Getting an A

```
public class Test {
    public static void main (String args[]) {
    int score = 93;
        String grade = "unknown";

        if (score > 90 && score < 93)
            grade = "A-";
        System.out.println("Given the input, your grade is " + grade);
    }
}
```

Save and **Run** it.

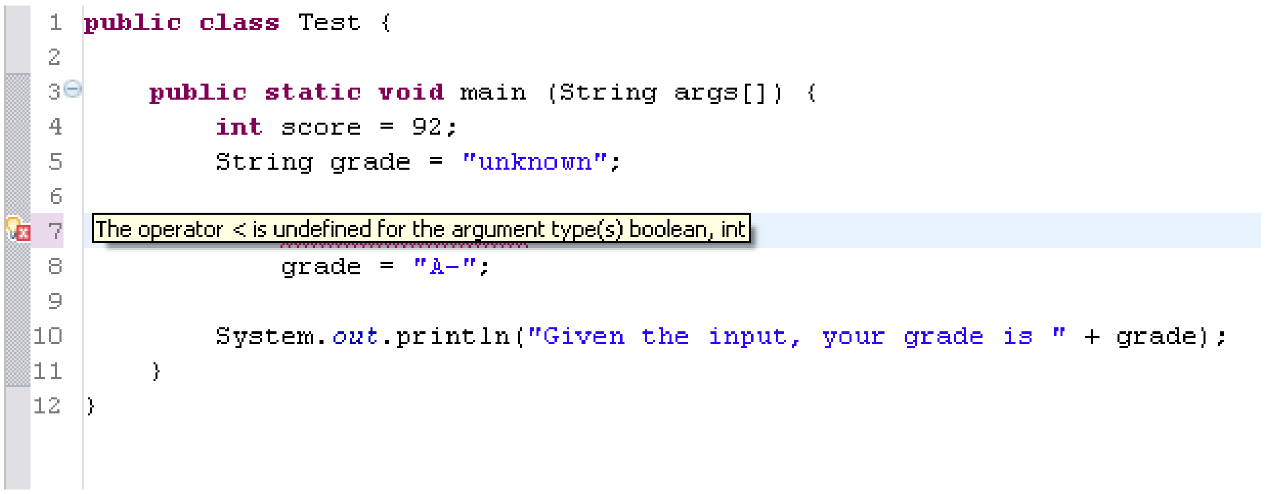
Now, change the score from **93** to **92** and run it again. Were those the results you expected?

Let's try working with the same semantics (meaning), but using the syntax we normally see in mathematics. Change the main as shown below:

CODE TO TYPE: Test

```
public class Test {
    public static void main (String args[]) {
          int score = 92;
          String grade = "unknown";

          if (90 < score < 93)
              grade = "A-";
          System.out.println("Given the input, your grade is " + grade);
     }
}
```

Move your cursor over the error; you'll see this:



```
public class Test {

    public static void main (String args[]) {
        int score = 92;
        String grade = "unknown";

The operator < is undefined for the argument type(s) boolean, int
            grade = "A-";

        System.out.println("Given the input, your grade is " + grade);
    }
}
```

We got that particular error message as a result of **precedence** rules. Java evaluates logic expressions from left to right. Consider the expression: (90 < score < 93); with score = 92, which becomes (90 < 92 < 93). It evaluates 90 < 92 and gets the result **true**. Java then sees: **true < 93**, which has no meaning.

The error message "**The operator < is undefined for the argument type(s) boolean, int**" makes sense now. We were trying to compare a boolean with an integer using "<".

Note Because logic operators are binary, expressions such as (a < b < c) must be expressed this way: (**a < b && b < c**)

Let's roll the dice and get more experience with logic.

Create a **java2_Lesson7 project**. In it, create a new Applet class called **Craps**. Enter the code as shown below:

CODE TO TYPE: Craps

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

public class Craps extends Applet implements ActionListener {
    private Button die1, die2;
    private int value1 = 0, value2 = 0;

    public void init() {
        die1 = new Button("Die1");
        add(die1);
        die1.addActionListener(this);
        die2 = new Button("Die2");
        add(die2);
        die2.addActionListener(this);
    }

    public void paint(Graphics g) {
        int total;
        total = value1 + value2;
        g.drawString("Die 1 is " + value1 + "     Die 2 is " + value2, 40, 50);
        g.drawString("Total is " + total, 40, 65);
        if (total==7)
            g.drawString ("You won!" , 50, 90);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getActionCommand().toString() == "Die1")
            value1 = (int)(Math.random() * 6) + 1;
        else
            value2 = (int)(Math.random() * 6) + 1;
        repaint();
    }
}
```

Save and **Run** it (click on the different die buttons to "roll"). Observe the code as you run it to see the different uses of **if**:

1. The **actionPerformed()** method is used to determine which **button** was clicked (Die1 **else** Die2).
2. The **paint()** method is used to determine if you hit the lucky number: **(total==7)**.

Add a line at the end of the **paint()** method as shown below:

CODE TO EDIT: Craps

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

public class Craps extends Applet implements ActionListener {
    private Button die1, die2;
    private int value1 = 0, value2 = 0;

    public void init() {
        die1 = new Button("Die1");
        add(die1);
        die1.addActionListener(this);
        die2 = new Button("Die2");
        add(die2);
        die2.addActionListener(this);
    }

    public void paint(Graphics g) {
        int total;
        total = value1 + value2;
        g.drawString("Die 1 is " + value1 + "     Die 2 is " + value2, 40, 50);
        g.drawString("Total is " + total, 40, 65);
        if (total==7)
            g.drawString ("You won!" , 50, 90);
        if (die1 == die2)
            g.drawString ("You won again!" , 50, 100);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getActionCommand().toString() == "Die1")
            value1 = (int)(Math.random() * 6) + 1;
        else
            value2 = (int)(Math.random() * 6) + 1;
        repaint();
    }
}
```

Save and **Run** it. Keep clicking on one button until it has the same value as the other (you can never roll a 0, so you have to roll both at least once).

How many times will you have to roll until you see the *You have won again!* message? Are there any probabilities experts out there? Unfortunately, expertise in probabilities won't help us here, and I'll lay down a million dollars that you will **never** see the *You have won again!* message with the code above. Why, you ask? Because we've made an error--in our reasoning. **die1** can **never** be **==** **die2**, because **they are different objects**!

Let's fix that problem. Instead of **instances** on either side of the equals signs **==**, let's get our program to use the **values** of those instances. Edit your code as shown in **blue** below:

CODE TO EDIT: Craps

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

public class Craps extends Applet implements ActionListener {
    private Button die1, die2;
    private int value1 = 0, value2 = 0;

    public void init() {
        die1 = new Button("Die1");
        add(die1);
        die1.addActionListener(this);
        die2 = new Button("Die2");
        add(die2);
        die2.addActionListener(this);
    }

    public void paint(Graphics g) {
        int total;
        total = value1 + value2;
        g.drawString("Die 1 is " + value1 + "     Die 2 is " + value2, 40, 50);
        g.drawString("Total is " + total, 40, 65);
        if (total==7)
            g.drawString ("You won!" , 50, 90);
        if (value1 == value2)
            g.drawString ("You won again!", 50, 100);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getActionCommand().toString() == "Die1")
            value1 = (int)(Math.random() * 6) + 1;
        else
            value2 = (int)(Math.random() * 6) + 1;
        repaint();
    }
}
```

Save and **Run** it. Hmm, it looks like we'll need an additional **if** in case both values are 0 **or** an additional logic operator such as:

if ((value1 == value2) && (value1 !=0))

Change the **paint( )** method as shown below:

CODE TO EDIT: Craps

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

public class Craps extends Applet implements ActionListener {
    private Button die1, die2;
    private int value1 = 0, value2 = 0;

    public void init() {
        die1 = new Button("Die1");
        add(die1);
        die1.addActionListener(this);
        die2 = new Button("Die2");
        add(die2);
        die2.addActionListener(this);
    }

    public void paint(Graphics g) {
        int total;
        total = value1 + value2;
        g.drawString("Die 1 is " + value1 + "     Die 2 is " + value2, 40, 50);
        g.drawString("Total is " + total, 40, 65);
        if (total==7)
            g.drawString ("You won!" , 50, 90);
        if ((value1 == value2) && (value1 !=0))
            g.drawString ("You won again!" , 50, 100);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getActionCommand().toString() == "Die1")
            value1 = (int)(Math.random() * 6) + 1;
        else
            value2 = (int)(Math.random() * 6) + 1;
        repaint();
    }
}
```

Save and **Run** it. Keep clicking on one button until it has the same value as the other. Much better!

Tip

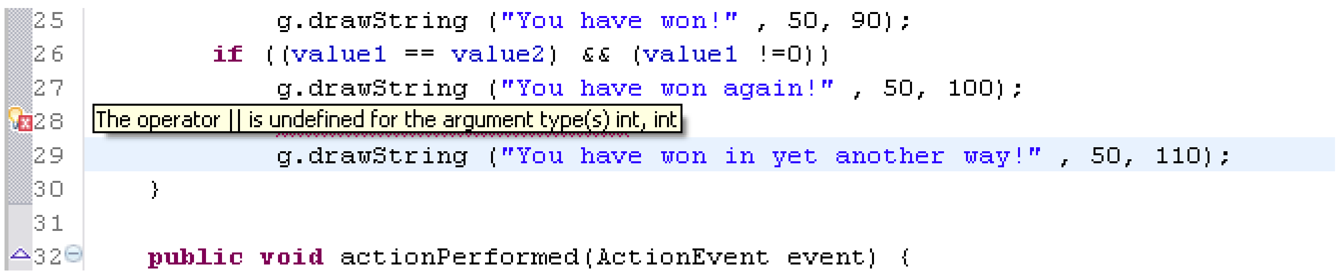
&& and **||** are considered **Lazy operators** because they do as little work as possible. In an **&&**, both operators must be **true** for the expression to be **true**, if the first is **false**, Java will not even bother to look at the second. In an **||**, only one operator needs to be **true** for the expression to be **true**, so if the first is **true**, Java will not even look at the other. So, if we put the expression most likely to be **false** first in **&&** and the expression most likely to be **true** first in **||**, we might save processing time.

Let's try another example. Add the lines in **blue** to the end of the **paint( )** method as shown:

CODE TO TYPE: Craps code

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

public class Craps extends Applet implements ActionListener {
    private Button die1, die2;
    private int value1 = 0, value2 = 0;

    public void init() {
        die1 = new Button("Die1");
        add(die1);
        die1.addActionListener(this);
        die2 = new Button("Die2");
        add(die2);
        die2.addActionListener(this);
    }

    public void paint(Graphics g) {
        int total;
        total = value1 + value2;
        g.drawString("Die 1 is " + value1 + "     Die 2 is " + value2, 40, 50);
        g.drawString("Total is " + total, 40, 65);
        if (total==7)
            g.drawString ("You won!" , 50, 90);
        if ((value1 == value2) && (value1 !=0))
            g.drawString ("You won again!" , 50, 100);
        if ((total=2) || (total=7))
            g.drawString ("You won in yet another way!" , 50, 110);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getActionCommand().toString() == "Die1")
            value1 = (int)(Math.random() * 6) + 1;
        else
            value2 = (int)(Math.random() * 6) + 1;
        repaint();
    }
}
```

Save and **Run** it. This time it doesn't compile:



```
            g.drawString ("You have won!" , 50, 90);
        if ((value1 == value2) && (value1 !=0))
            g.drawString ("You have won again!" , 50, 100);
The operator || is undefined for the argument type(s) int, int
            g.drawString ("You have won in yet another way!" , 50, 110);
    }

    public void actionPerformed(ActionEvent event) {
```

Check out either side of the **||**. On the left, we have **total = 2**. This is not a comparison statement that will result in **true** or **false**. Because we used a single equals sign**=** instead of two of them **==**, it's an **assignment statement**, so it shouldn't be inside of the **if** condition. But actually, we want Java to compare rather than to assign.

We can fix that. Change the **=** on both sides to **==** as shown below:

CODE TO TYPE: Craps

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

public class Craps extends Applet implements ActionListener {
    private Button die1, die2;
    private int value1 = 0, value2 = 0;

    public void init() {
        die1 = new Button("Die1");
        add(die1);
        die1.addActionListener(this);
        die2 = new Button("Die2");
        add(die2);
        die2.addActionListener(this);
    }

    public void paint(Graphics g) {
        int total;
        total = value1+value2;
        g.drawString("Die 1 is " + value1 + "     Die 2 is " + value2, 40, 50);
        g.drawString("Total is " + total, 40, 65);
        if (total==7)
            g.drawString ("You won!" , 50, 90);
        if ((value1 == value2) && (value1 !=0))
            g.drawString ("You won again!" , 50, 100);
        if ((total==2) || (total==7))
            g.drawString ("You won in yet another way!" , 50, 110);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getActionCommand().toString() == "Die1")
            value1= (int)(Math.random() * 6) + 1;
        else
            value2= (int)(Math.random() * 6) + 1;
        repaint();
    }
}
```

Save and **Run** it.

We've sure got a lot of parentheses in our code, but that's okay. Even though they're not all necessary, they help us make sense of the code when we read it, so we may as well keep them there.

Tip If precedence standards already dictate that a command will be executed, parentheses are not necessary.

Go into the editor and experiment; take out some of sets of parentheses to see what happens.

Your assignment for this lesson will be to roll the dice and then define the various combinations of dice rolls as they are used in the craps game. Keep this in mind as we go through the lesson. Do you know what it means to roll "snake eyes" (1 and 1)? In craps, if you roll a 1 and 1 on your first roll, you've "crapped out" (you lose). Can you tell someone using our Java program if they have crapped out?

For more information on Craps, here's a Wikipedia article . For an enhanced version of our craps game, with sound and images, click here .

Nested Ifs Versus Logic Statements

Nested **ifs** are nice when you want to reduce the number of if statments that Java has to go through. In other words, Java stops going through the nested **ifs** as soon as it finds a *true* statement. Let's try an example.

Most people are familiar with this grading scale:

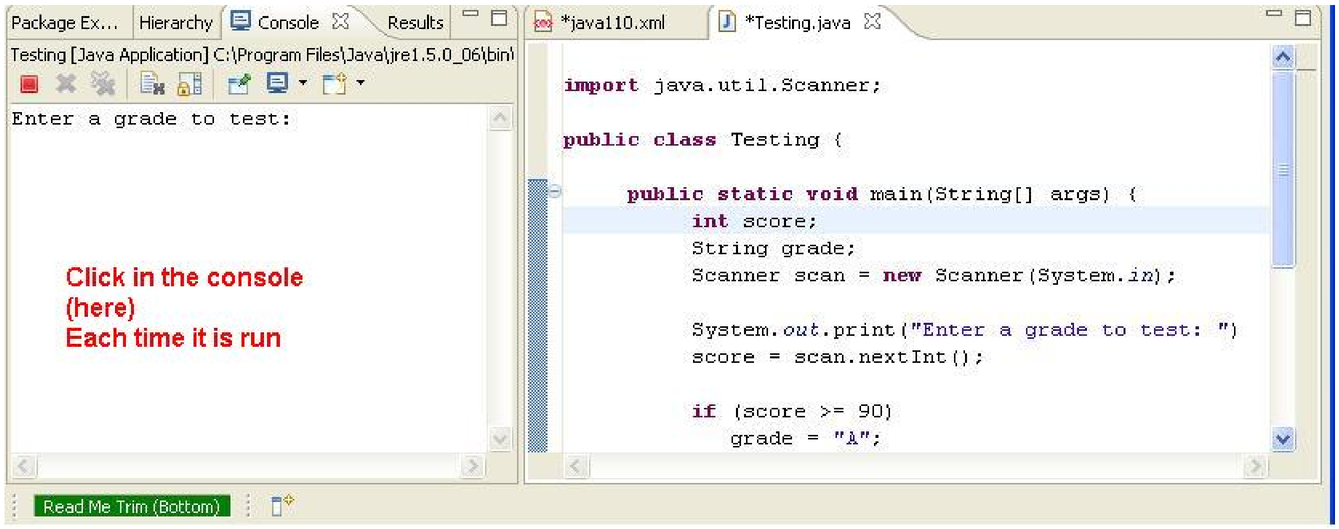
- A: 100-90
- B: 89-80
- C: 79-70
- D: 69-60
- F: below 60

We have two ways to implement such a grade scale. Let's make a quick little main method to test them. Create a new class in java2_Lesson7 (or java2_Lesson7/src) called Testing. Type the code in **blue** below:

CODE TO TYPE: Testing.java

```
import java.util.Scanner;

public class Testing {

   public static void main(String[] args)   {

       int score;
       String grade;
       Scanner scan = new Scanner(System.in);

       System.out.print("Enter a grade to test: ");
       score = scan.nextInt();

       if (score >= 90)
           grade = "A";
       else if (score >= 80)
           grade = "B";
       else if (score >= 70)
           grade = "C";
       else if (score >= 60)
           grade = "D";
       else
           grade = "F";
       System.out.println("For a score of " + score + ", the grade is " + grade);
    }
}
```

Save and **Run** it. The class will prompt you for a score in the console to the left:



Testing [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\

```
Enter a grade to test:
```

```
import java.util.Scanner;

public class Testing {

    public static void main(String[] args) {
        int score;
        String grade;
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a grade to test: ")
        score = scan.nextInt();

        if (score >= 90)
           grade = "A";
```

Click in the console, and then **enter** a number when it prompts. (For this class, you'll need to **Run As** again to test a different number.)

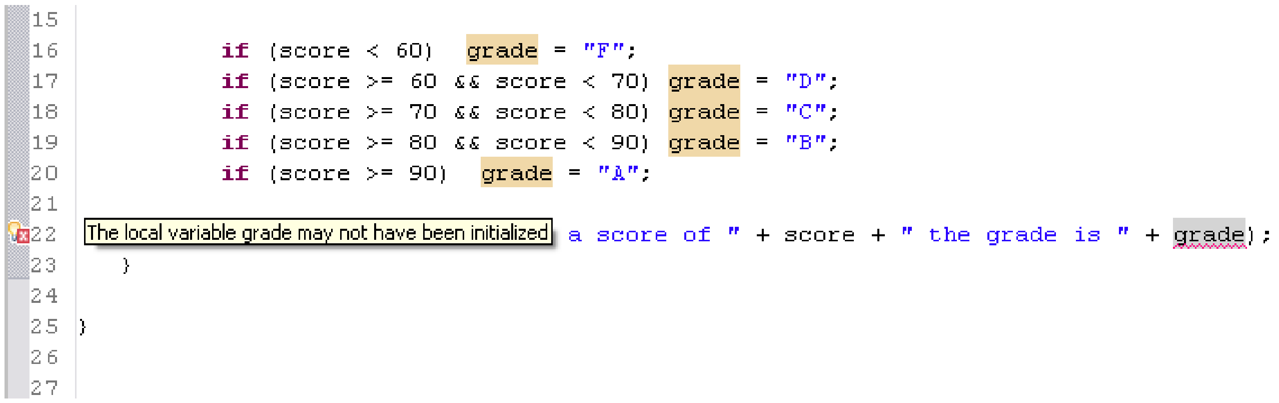
Sometimes code is easier to read if you use logic instead of lots of nested **if**s. Let's try an example using logic. Replace the nested **if** in **Testing.java** as shown below:

CODE TO EDIT: Testing.java

```
import java.util.Scanner;

public class Testing {

    public static void main(String[] args)  {

        int score;
        String grade;
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a grade to test: ");
        score = scan.nextInt();

        if (score < 60)  grade = "F";
        if (score >= 60 && score < 70) grade = "D";
        if (score >= 70 && score < 80) grade = "C";
        if (score >= 80 && score < 90) grade = "B";
        if (score >= 90)  grade = "A";
        System.out.println("For a score of " + score + ", the grade is " + grade);
    }
}
```

Save and **Run** it.

Whoops. We got an error:



```

            if (score < 60)  grade = "F";
            if (score >= 60 && score < 70) grade = "D";
            if (score >= 70 && score < 80) grade = "C";
            if (score >= 80 && score < 90) grade = "B";
            if (score >= 90)  grade = "A";

The local variable grade may not have been initialized a score of " + score + " the grade is " + grade);
    }

}


```

Why didn't we get this same error when we used the nested **if** instead of logic? Because our nested **if** version ended in an **else** that initialized the variable. When a program ends in **else**, even when a score isn't defined, we still get a value. In this case, after all the **else** statements were exhausted, the default grade of **F** was assigned.

Add the blue code below:

CODE TO EDIT: Testing.java

```
import java.util.Scanner;

public class Testing {

    public static void main(String[] args)  {

        int score;
        String grade;
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a grade to test: ");
        score = scan.nextInt();

        if (score < 60)  grade = "F"; else grade = "unknown";
        if (score >= 60 && score < 70) grade = "D";
        if (score >= 70 && score < 80) grade = "C";
        if (score >= 80 && score < 90) grade = "B";
        if (score >= 90)  grade = "A";
        System.out.println("For a score of " + score + ", the grade is " + grade);
    }
}
```

Save and **run** it.

Now our program works. We are assured of a value for **grade**; if the score is known, Java overwrites the "unknown" string. Although this method works, it isn't preferred. Java has to check too many things even though it might already have the answer it needs. This can slow the computer down. We try to avoid making the computer do more work than is necessary.

There are a lot of ways to use **if** statements. The statement that gets executed when the **if**'s conditional is true, can be another **if** condition. Here's another way you might write your grading program;

OBSERVE: If condition as statement

```
if (score < 80)
    if (score < 60)
        grade = "F";
    else if (score < 70)
        grade = "D";
    else
       grade = "C";
else if (score <= 100)
    if (score < 90)
        grade = "B";
    else
        grade = "A";
```

Logic Statements Versus If Statements

Consider a method that determines whether a year is a leap year. The Gregorian calendar stipulates that a year is a leap year if it is divisible by 4, unless it is also divisible by 100. If it is divisible by 100 then it is a leap year if and only if it is also divisible by 400. For example, 1900 was not a leap year, but 2000 was.

Here is the algorithm for this using just one logic statement:

Leap Year Logic

```
    (year % 100 != 0 && year % 4 == 0) || (year % 100 == 0 && year % 400 == 0)
```

Let's use it!

Edit the **Testing** class's **main** method as follows:

CODE TO TYPE: leap logic

```
import java.util.Scanner;

public class Testing {

    public static void main(String[] args)  {

        int year;
        boolean leap;
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a year to test: ");
        year = scan.nextInt();

        leap = (year % 4 == 0 && year % 100 != 0) || (year % 100 == 0 && year % 400 ==
0);

        System.out.println("It is " + leap + " that " + year + " is a leap year.");
    }
}
```

Save and **Run** it. Try running it with different values for the year to test the expression.

The code works, but it's not easy for others to read. Let's break up the **&&**s and **||**s to get an algorithm that's easier to read. People generally know that a leap year is divisable by 4, so let's begin our algorithm there:

1. Divide the year by 4. The remainder should be 0 (year % 4 == 0).
2. **If** it is divisible by 4, then see **if** it is divisible by 100.

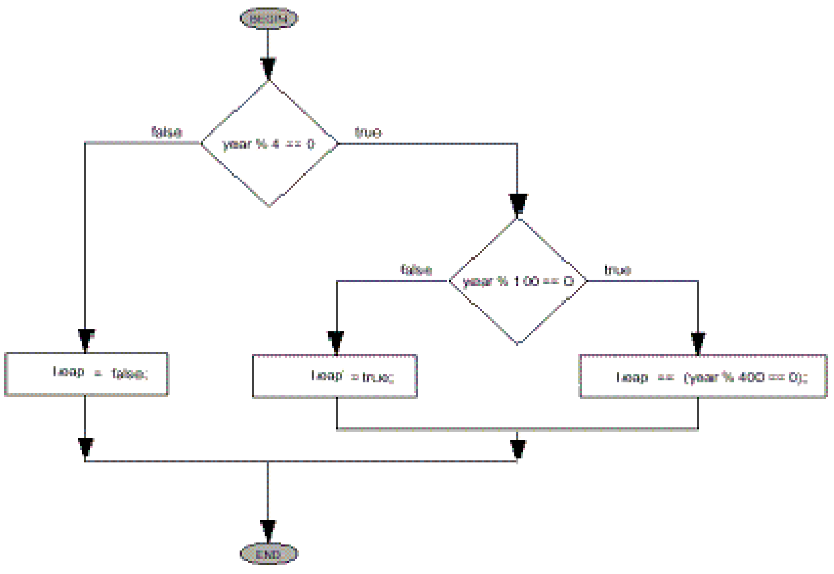
3. **If** it is divisible by 100, then see if it is divisible by 400--in which case it **is** a leap year.
4. **If** it is **not** divisible by 100, but it was divisible by 4, it **is** a leap year.
5. **If** it was **not** even divisible by 4 at all, it was **not** a leap year.

Let's make this into an algorithm using **if**s. Edit the main method to use **if**s rather than the single logic statement, like this:

CODE TO EDIT: Testing

```
import java.util.Scanner;

public class Testing {

    public static void main(String[] args)  {

        int year;
        boolean leap;
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a year to test: ");
        year = scan.nextInt();

        if (year % 4 == 0)
            if (year % 100 == 0)    // if divisible by 100 must be divisible by 400
                leap = (year % 400 == 0);  // i.e., if year is divisible by 400 then th
is is true
            else
                leap = true;
        else
            leap = false;

        System.out.println("It is " + leap + " that " + year + " is a leap year.");
    }
}
```

Save and **Run** it. Try it a few more times to test the expression, using different values for the year. Here's the **if**'s flowchart:



Our code could have looked like this:

A very tidy alternative

```
if (year % 4 == 0)
if (year % 100 == 0)
if (year % 400 == 0)
  leap = true;
else

   leap = false;
else
   leap = true;
else

   leap = false;
```

Can you match the **if**s with the proper **else**s?

We even have Duke leaping!

Decisions: Dangling and Switching

Lesson Objectives

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

- use nested if or case statements to return different results based on user input.
- use if, else, and switch statements seperately and in conjunction with one another when appropriate.

Matching Who Gets What

Dangling Elses

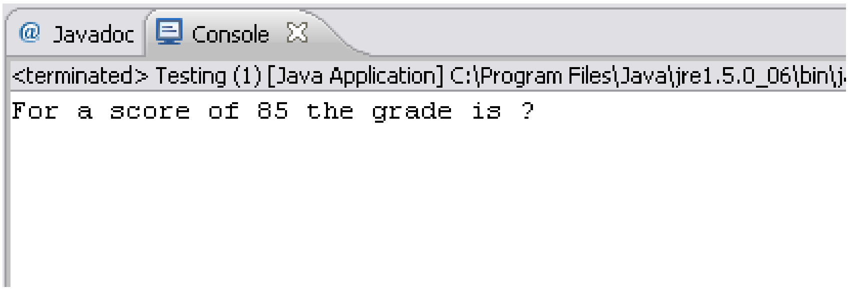
In the code from the previous lesson, we initally used two **if**s and two matching **else**s Later we used three **if**s and three matching **else**s. But **If**s do not require **else**s. It's possible to have nested **if**s without having the same number of **else**s. Such a situation is referred to as a **dangling else**. When a **dangling else** occurs and the brackets are not used, running that code may give the wrong results.

Create a new **java2_Lesson8** project with a new **Testing** class application in it. Enter the code as shown below:

CODE TO TYPE: dangling else

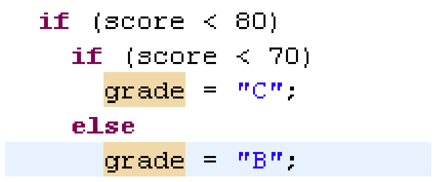
```
public class Testing {

    public static void main(String[] args)  {

        int score = 85;
        String grade = "?";

        if (score < 80)
            if (score < 70)
                grade = "C";
            else
                grade = "B";

        System.out.println("For a score of " + score + ", the grade is " + grade
);
    }
}
```

Save and **Run** it.



```
<terminated> Testing (1) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\j
For a score of 85 the grade is ?
```

Wait a minute. This score should result in a different grade. We want our program to give a B if the score is greater than 80, but Java is showing it did not execute the **else** condition. The **else** was executed with the *second* **if**, and Java did not give a B for **score > 80**. Even though Java doesn't recognize indentation, we can still look at the way the our code is indented to get a clear idea of what happened:



```
if (score < 80)
  if (score < 70)
    grade = "C";
  else
    grade = "B";
```

The **else** is attached to the indented **if** and is not applied to the first **if**. So Java has nothing to do if the score is less than 80.

Tip An **else** will always match with the **nearest if**, unless the use of brackets specifies otherwise.

Indentation does not matter to Java. Braces (**{}**) and semicolons (**;**) tell Java where things start and stop.

Change the code as shown below in blue:

CODE TO TYPE: dangling else

```
public class Testing {

    public static void main(String[] args)  {

        int score = 85;
        String grade = "?";

        if (score < 80)
            { if (score < 70)
                grade = "C"; }
        else
            grade = "B";

        System.out.println("For a score of " + score + ", the grade is " + grade
);
    }
}
```

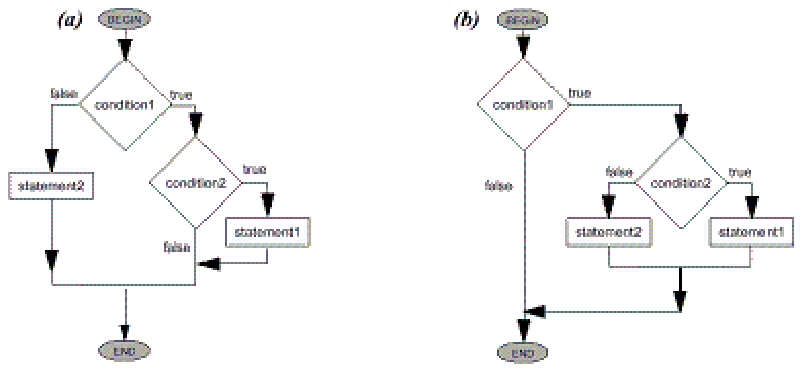
Save and **Run** it. That's much better.

See if you can determine which **if** statement corresponds to the **else** statement in the following code:

Which IF does ELSE belong to?

```
if (condition1)
    if (condition2)
        statement1
else
    statement2
```

In the code above, the **else** is associated with the second **if**, despite its position directly below the first one. The flow chart below gives you a more detailed look:



To associate the **else** with the first **if**, add braces around all the statements contained between the first **if** and the **else**.

Using Braces to Specify Flow

```
if (condition1)
  {
  if (condition2)
     statement1
  }
else
   statement2
```

Object equivalence

Another common error novice programmers make when using conditional statements is the use of **==** with **Strings**. Let's edit the **Testing** class again to see how that happens:

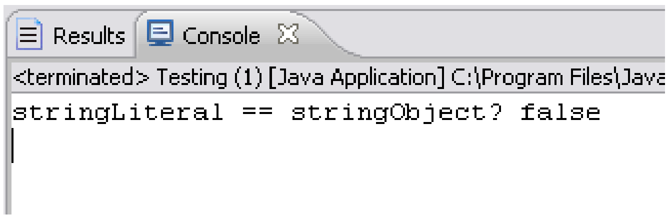
CODE TO EDIT: Testing

```
public class Testing {

    public static void main(String[] args)  {

     String stringLiteral = "are they the same";
     String stringObject = new String ("are they the same");
     System.out.println("stringLiteral == stringObject? " + (stringLiteral == st
ringObject));
    }
}
```

The boolean expression **(stringLiteral == stringObject)** is embedded in the **System.out.println**. If you only need to use a value once and don't need to save it in memory, this is a common technique.

Tip In Java, anywhere you can put a **variable** of a certain type, you can put an **expression** that results in that type (except on the left side of an equal sign, because that is a memory **location**).

Save and **Run** it.



```
<terminated> Testing (1) [Java Application] C:\Program Files\Java
stringLiteral == stringObject? false
```

Java tells us they are not the same because **==** is **object equivalence** and the two **Objects stringLiteral** and **stringObject** are not the same objects. They do not access the **same place in memory**.

Add the line of code shown below in **blue**:

CODE TO EDIT:

```
public class Testing {

    public static void main(String[] args)  {
         String stringLiteral = "are they the same";
         String stringObject = new String ("are they the same");
         System.out.println("stringLiteral == stringObject? " + (stringLiteral =
= stringObject));
         System.out.println("The sequence of characters is the same is " + strin
gLiteral.equals(stringObject));
    }
}
```

Save and **Run** it.

Go to the **java.lang** package and look up the **String** class. Check out the methods there--we discussed them in the first course, but they're really important in Java, so it's good to refresh your memory.

The Instance of Operator

Before we leave this discussion of **if**s, and switch to **switch**, let's look at another handy operator that Java provides.

In Java sometimes a method receives a collection of objects. You know the objects are an instance of a Class, but you don't necessarily know which Class. For example, in event handling, both **TextFields** and **Buttons** use the **ActionListener** interface. When an **ActionEvent** occurs, how does the listener know which one triggered the event?

Let's edit our **Driving** class in the Driving.java file. We used this example in the Java 1 course, but here it is again, just in case:

Click here to download The Driving Project

Open Driving.java to allow a user to **reset the age to 0** with the click of a Button. Add the code shown in

blue below:

CODE TO EDIT: Driving.java

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

public class Driving extends Applet
    implements ActionListener {

    private TextField ageField;
    private int age = 0;

    public void init() {

        Label l = new Label("Enter your age"); // the "l" is a lower-case "L," n
ot a One.
        add(l);                              // Same with this "l."

        ageField = new TextField(3);
     add(ageField);
     ageField.addActionListener(this);

        Button reset = new Button("Reset");
        add(reset);
        reset.addActionListener(this);
    }

   public void paint(Graphics g) {
     if ( ! (age < 100) )
            {
              g.drawString ("Are you kidding me!", 30, 70);
              g.drawString ("Delete some of those numbers." , 30, 80);
             }
     else if (age > 15)
      {
        g.drawString ("Congratulations", 50, 50);
        g.drawString ("You may drive", 50, 70);
      }
     else
      {
           g.drawString ("Wait a few years", 50, 50);
           g.drawString ("You may not drive yet", 50, 70);
      }
     g.drawString("Age is " + age, 50, 90);
    }

   public void actionPerformed(ActionEvent event) {
     if (event.getSource() instanceof Button)
            {
       age = 0;
       ageField.setText("");
            }
            else if (event.getSource() instanceof TextField)
            if (ageField.getText().length() == 0)
          age = 0;
        else
          age = Integer.parseInt(ageField.getText());
        repaint();

    }
}
```

Save and **Run** it.

Now **if** the event.getSource() is an *instance of* the Button Class, then (event.getSource() *instanceof* Button) will be true. We also put in an **else** that tests to see if event.getSource is an instance of the TextField class. If it is, then it retrieves the number we typed.

We'll see more of **instanceof** as the course progresses.

Switch Statements

The **if** statement is a powerful and commonly used control construct in programming. A similar construct is the **switch statement**. Switch statements can provide clarity and allow for a greater number of execution paths.

In general, the format for **switch** statements is:

OBSERVE: switch statements

```
switch (expr1) {
    case expr2:
        statements;
        break;
    case expr3:
        statements;
        break;
    default:
        statements;
        break;
}
```

In this example, expressions (**expr1, expr2, expr3**, etc.) must be constants: **int, byte, short** or **char**. (These can be convertible via casting to **int** (automatic promotion).)

Let's look at an example using **if**s, in order to compare the two constructs. In the **java2_Lesson8** Project, make a new class named **Days**, and type in the code below:

CODE TO TYPE: Days.java

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

public class Days extends Applet implements ActionListener{

    private TextField dayField;
    private int day;

    public void init() {
        dayField=new TextField(2);
        add(dayField);
        dayField.addActionListener(this);
    }

    public void paint(Graphics g) {

        g.drawString("Give a number from 1 to 7", 5, 80);

        if (day == 1)
           g.drawString("Monday", 50, 50);
        else
         if (day == 2)
           g.drawString("Tuesday", 50, 50);
        else
         if (day == 3)
           g.drawString("Wednesday", 50, 50);
        else
         if (day == 4)
           g.drawString("Thursday", 50, 50);
        else
         if (day == 5)
           g.drawString("Friday", 50, 50);
        else
         if (day == 6)
           g.drawString("Saturday", 50, 50);
        else
         if (day == 7)
           g.drawString("Sunday", 50, 50);
        else
           g.drawString("Please follow directions", 5, 50);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource()==dayField)
            day=Integer.parseInt(dayField.getText());
        repaint();
    }
}
```

Save and **Run** it.

Now, let's do the same thing in the next example using a **switch** statement instead of the nested **if**s and **else**s. Edit **Days.java** as shown in **blue** below:

CODE TO EDIT: Days

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

public class Days extends Applet implements ActionListener{

    private TextField dayField;
    private int day;

    public void init() {
        dayField=new TextField(2);
        add(dayField);
        dayField.addActionListener(this);
    }

    public void paint(Graphics g) {

        g.drawString("Give a number from 1 to 7", 5, 80);

        switch (day) {
              case 1: g.drawString("Monday", 50, 50); break;
              case 2: g.drawString("Tuesday", 50, 50); break;
              case 3: g.drawString("Wednesday", 50, 50); break;
              case 4: g.drawString("Thursday", 50, 50); break;
              case 5: g.drawString("Friday", 50, 50); break;
              case 6: g.drawString("Saturday", 50, 50); break;
              case 7: g.drawString("Sunday", 50, 50); break;
              default: g.drawString("Please follow directions", 5, 50); break;
              }
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource()==dayField)
            day=Integer.parseInt(dayField.getText());
        repaint();
    }
}
```

Save and **Run** it.

Both examples should produce the same output. Notice that after each **case**, the line ends in a semi-colon, and then has a **break;**:

Case Syntax

```
switch (day) {
    case 1: g.drawString("Monday", 50, 50); break;
    ...
    default: g.drawString("Please follow directions", 5, 50); break;
    }
next code
```

Here's what's happening:

- The **break** statement tells Java to escape from the current control construct; that is, to leave the **switch** statement and continue with the **next code**.
- The control statement **switch** goes down the list of cases until if finds a case that matches the number that was entered (or sees the **default**).
- The instructions for the particular case are executed, and then if it sees **break**, it exits the **switch**.
- If it does **not** see a **break**, it continues down the list and does the rest of the cases as well.

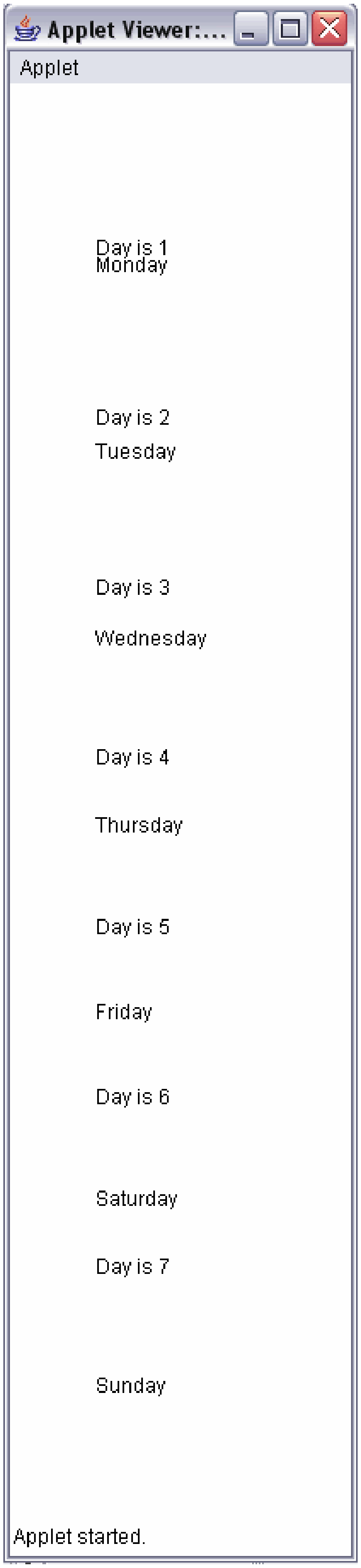
If we edit the Days applet, we can observe what's happening from a different angle. First we'll change the applet so that it moves through each day without user input. And since we won't have user input, we'll remove the **Listener** and its method **actionPerformed( )**.

Edit your program as shown below in **blue**:

CODE TO EDIT: loop through the days

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

public class Days extends Applet {
    private int day;

    public void start(){
        setSize(200,900);
    }

    public void paint(Graphics g) {
        for (day = 1; day < 8; day++)
        {
           g.drawString("Day is " + day, 50, 100*day);
           switch (day) {
                   case 1: g.drawString("Monday", 50, 100*day + 10); break;
                   case 2: g.drawString("Tuesday", 50, 100*day + 20); break;
                   case 3: g.drawString("Wednesday", 50, 100*day + 30); break;
                   case 4: g.drawString("Thursday", 50, 100*day + 40); break;
                   case 5: g.drawString("Friday", 50, 100*day + 50); break;
                   case 6: g.drawString("Saturday", 50, 100*day + 60); break;
                   case 7: g.drawString("Sunday", 50, 100*day + 70); break;
                   default:g.drawString("Not a day", 50, 790);
                   }
        }
    }
}
```

Save and **Run** it.



```
Day is 1
Monday

Day is 2
Tuesday

Day is 3
Wednesday

Day is 4
Thursday

Day is 5
Friday

Day is 6
Saturday

Day is 7
Sunday
```

Applet started.

To get this output we used a **for loop**. We will cover **for loops** in more detail later; for now you just need to understand that the for loop REPEATS actions over and over again *for* numbers running from some initial number to some final number. The initial number in this case is **1**, defined by **day = 1**, and the final number is day < 8, or **7**, with the day incrementing by 1 each time.

Here we went through the days and set them one at a time--day = 1, then day = 2, then day =3, etc.

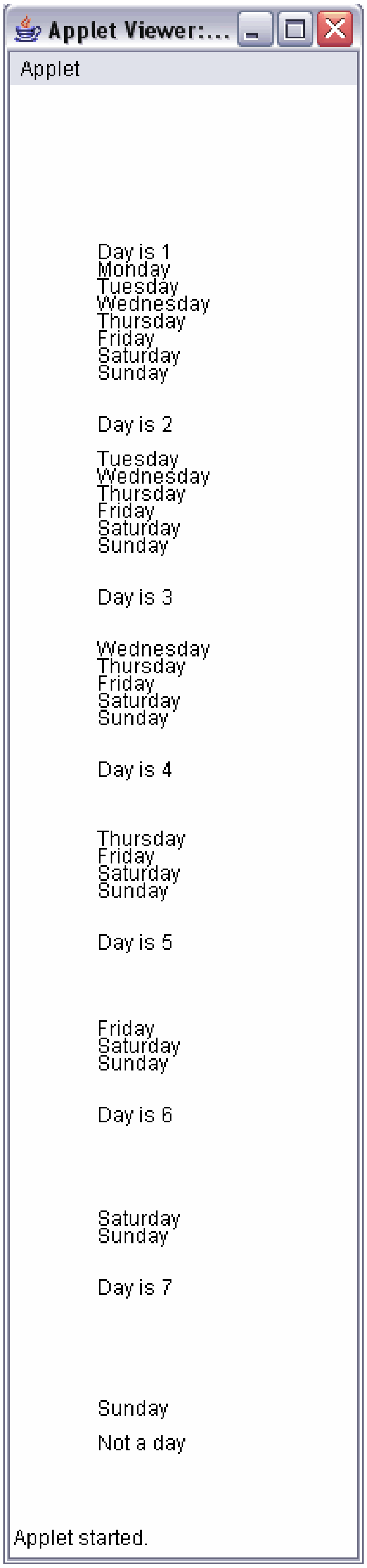
If you do not use the **break** statement, Java will continue down through **all** of the cases after the first true one.

Comment out the **break** statements as shown:

CODE TO TYPE:

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

public class Days extends Applet {

    private int day;

    public void paint(Graphics g) {

        for (day = 1; day < 8; day++)
        {
        g.drawString("Day is " + day, 50, 100*day);
        switch (day) {
                   case 1: g.drawString("Monday", 50, 100*day + 10); // break;
                   case 2: g.drawString("Tuesday", 50, 100*day + 20); // break;
                   case 3: g.drawString("Wednesday", 50, 100*day + 30); // break;
                   case 4: g.drawString("Thursday", 50, 100*day + 40); // break;
                   case 5: g.drawString("Friday", 50, 100*day + 50); // break;
                   case 6: g.drawString("Saturday", 50, 100*day + 60); //break;
                   case 7: g.drawString("Sunday", 50, 100*day + 70); // break;
                   default: g.drawString("Not a day", 50, 790);
                   }
        }
    }
}
```

Save and **Run** it.



```
Day is 1
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday

Day is 2

Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday

Day is 3

Wednesday
Thursday
Friday
Saturday
Sunday

Day is 4


Thursday
Friday
Saturday
Sunday

Day is 5



Friday
Saturday
Sunday

Day is 6




Saturday
Sunday

Day is 7





Sunday

Not a day



Applet started.
```

Normally Java would output only one day at a time with something like:

Day is 2
Tuesday

But here we have a loop to do each day, and **no** break, so Java prints all the days that come **after** as well.

Sometimes a particular case is valid *and* all of the following cases are valid as well. Using the **break** can allow programmers to have many options.

The java tutorial contains two examples using the break. When you click on the link, the first tutorial example you see has something to print at each `case`. We'll be going over the second tutorial example here in this lesson. It illustrates a situation when it is useful to "fall through". In this sense, various inputs can activate a situation.

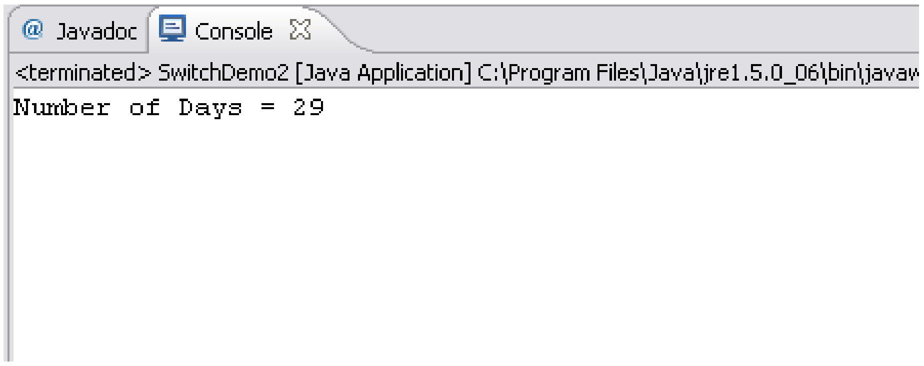
In java2_Lesson8, create a new application class named **SwitchDemo2**. **Type** **SwitchDemo2** as shown in **blue** below.

CODE TO TYPE: SwitchDemo2.java

```
public class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;
                break;
            case 2:
                if ( ((year % 4 == 0) && !(year % 100 == 0))
                     || (year % 400 == 0) )
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                numDays = 0;
                break;
        }
        System.out.println("Number of Days = " + numDays);
    }
}
```

Save and **Run** it.

Since this is an application, output will be seen in the console:



```
<terminated> SwitchDemo2 [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw
Number of Days = 29
```

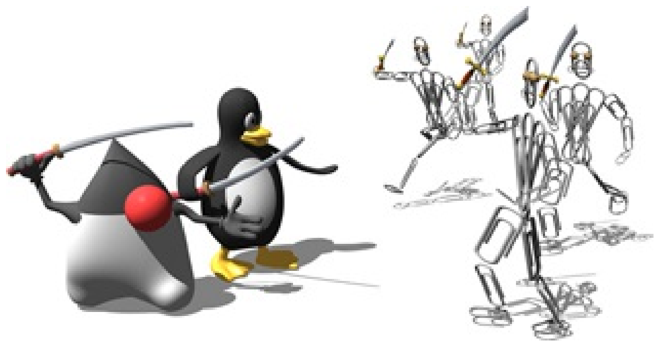
If you change **month = ?;** to different numbers, then Save and Run, it will give you the number of days for various months.

The final **break** is not required, because flow would fall out of the **switch** block anyway. However (as is seen on the tutorial page), the **break** is recommended to make modifying the code easier and less error-prone.

What's next?

The remaining control construct to learn about is **loops**. But before discussing them, we'll look at a data structure called **arrays**. Arrays are used in loops frequently, and will become a familiar tool you'll use to keep track of multiple pieces of data.

Phew! We've covered a lot of material so far. When we add arrays and looping to our arsenal, we'll be able to battle whatever comes our way!



Introduction to Arrays

Lesson Objectives

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

- declare arrays.
- identify various types of arrays.
- create arrays and assign them values.
- store dtat types and values within arrays.
- access array values.

What are Arrays?

We've learned a couple of ways to represent information in memory in object-oriented languages. We can use objects such as **classes** with specific **instances**, as well as **primitive data types** like **int**, **double**, and **char**.

But suppose we wanted to keep a collection of such objects together as a group? Well, there are a number of ways to do this. We could use the **Java Collections Framework**, but the most commonly used repository for a collection of pieces of information in Computer Science has long been the **array** construct. An array is a "container" that holds a fixed number of values of a single type; let's look at arrays now.

Some Fab Examples

A basic array is structured as follows:

INDICES	0	1	2	3	4
ARRAY beatles:	"John Lennon"	"Paul McCartney"	"George Harrison"	"Ringo Starr"	"George Martin"

For those of you who are not familiar with the history of the Beatles, *George Martin* produced most of the Beatles' music in the recording studio, and is considered by many to be the "fifth Beatle."

The items in the array are called *elements* and the "handles" to them are *indices*--each item is accessed by its numerical *index*. The array is named **beatles**. It's an array of elements, all of which are of type **String**:

beatles[3] = "Ringo Starr"

beatles[0] = "John Lennon"

beatles[4] = "George Martin"

After looking at the above elements, can you guess which one is beatles[2]? In other words, which element in the beatles array is located at index 2? I hope your answer was "George Harrison"!

Arrays always begin with index 0. The array's length (the number of elements that an array can hold) is always equal to the sum of the highest index in the array plus one. So what is the length (or **size**) of the **beatles** array? Let's check by using actual code.

Make a new **java2_Lesson9** project. Add an application class named **BeatleJuice**. **Type BeatleJuice** as shown below:

CODE TO TYPE: BeatleJuice

```
public class BeatleJuice {

    public static void main(String[] args) {
        String[] beatles;                // declares an array of Strings

        beatles = new String[5];         // allocates memory for 5 Strings

        beatles[0] = "John Lennon";      // initialize first element at index 0
        beatles[1] = "Paul McCartney";   // initialize second element
        beatles[2] = "George Harrison";  // etc.
        beatles[3] = "Ringo Starr";
        beatles[4] = "George Martin";

        System.out.println("Element at index 0: " + beatles[0]);
        System.out.println("Element at index 1: " + beatles[1]);
        System.out.println("Element at index 2: " + beatles[2]);
        System.out.println("Element at index 3: " + beatles[3]);
        System.out.println("Element at index 4: " + beatles[4]);

        System.out.println("\nSize of the beatles array is " + beatles.length);
    }
}
```

Save and **Run** it.

Did you notice what **\n** did in the System.out.println output? Try putting a few more **\n**s in there to see more.

Index Possibilities

Let's try another example:
If x = 1, What is beatles[x], beatles[2*x], beatles[x + 1]? We can find out by tweaking the code from our first example.

Edit the code as shown below:

CODE TO EDIT: BeatleJuice

```
public class BeatleJuice {
    public static void main(String[] args) {
        String[] beatles;              // declares an array of Strings

        beatles = new String[5];      // allocates memory for 5 Strings

        beatles[0] = "John Lennon"; // initialize first element
        beatles[1] = "Paul McCartney"; // initialize second element
        beatles[2] = "George Harrison"; // etc.
        beatles[3] = "Ringo Starr";
        beatles[4] = "George Martin";

        System.out.println("Element at index 0: " + beatles[0]);
        System.out.println("Element at index 1: " + beatles[1]);
        System.out.println("Element at index 2: " + beatles[2]);
        System.out.println("Element at index 3: " + beatles[3]);
        System.out.println("Element at index 4: " + beatles[4]);
        System.out.println("Element at index 5: " + beatles[5]);

        System.out.println("\nSize of the beatles array is " + beatles.length);
        int x = 1;
        System.out.println();
        System.out.println("When x=" + x + ", the element at beatles[x] is " + b
eatles[x]);
        System.out.println("When x=" + x + ", the element at beatles[2*x] is " +
 beatles[2*x]);
        System.out.println("When x=" + x + ", the element at beatles[x+1] is " +
 beatles[x+1]);
    }
}
```

Save and **Run** it.

Now let's check out a slick (and dangerous) use of increments (i++):
If x = 1
What is beatles[x++]?
What is beatles[++x]?
Let's try putting increments into our existing program. Edit the code as shown below:

CODE TO EDIT: BeatleJuice

```
public class BeatleJuice {
    public static void main(String[] args) {
        String[] beatles;              // declares an array of Strings

        beatles = new String[5];

        beatles[0] = "John Lennon";
        beatles[1] = "Paul McCartney";
        beatles[2] = "George Harrison";
        beatles[3] = "Ringo Starr";
        beatles[4] = "George Martin";

        System.out.println("Element at index 0: " + beatles[0]);
        System.out.println("Element at index 1: " + beatles[1]);
        System.out.println("Element at index 2: " + beatles[2]);
        System.out.println("Element at index 3: " + beatles[3]);
        System.out.println("Element at index 4: " + beatles[4]);

        System.out.println("\nSize of the beatles array is " + beatles.length);
        int x = 1;
        System.out.println();
        System.out.println("When x=" + x + ", the element at beatles[x] is " + b
eatles[x]);
        System.out.println("With x=" + x + ", the element at beatles[x++] is " +
 beatles[x++]);
        System.out.println("After use and then increment, x is now " + x);
        System.out.println("With x=" + x + ", Element at beatles[++x] is " + bea
tles[++x]);
        System.out.println("After increment and then use, x is now " + x);
    }
}
```

Save and **Run** it.

What is beatles[x++]?

Tricky--the variable x is used first to get the element located at beatles[x], and then x is incremented.

What is beatles[++x]?

The variable x was incremented first to x+1 and then used to get the element located at beatles[x] (where the value of x has been increased by 1).

Let's look at another example. This time we have an array named **array1** that consists of `int` (integer values):

INDICES	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ARRAY array1:	42	15	74	6	32	150	724	66

In this array, array1[1] = 15

What is array1[4]?

What is array1[6]?

What is array1[8]?

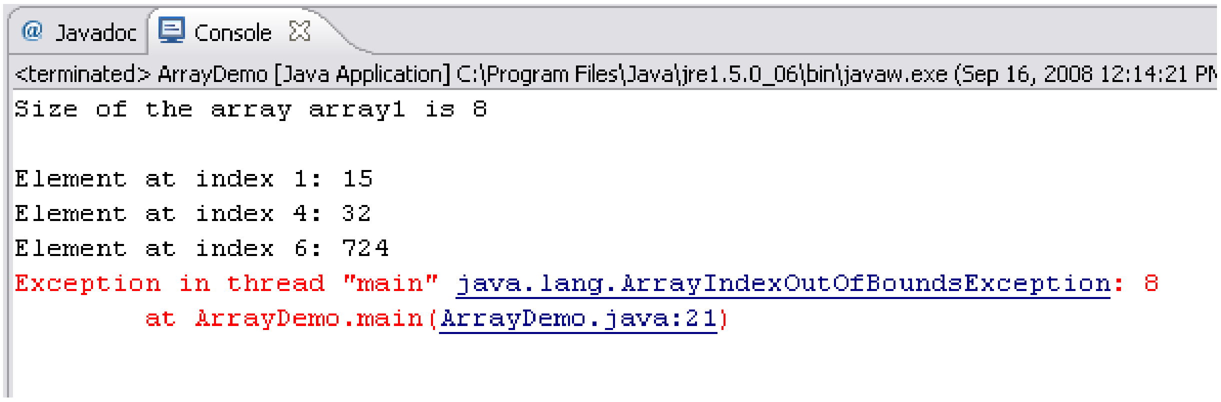
What is array1's size?

Let's test your answers. In java2_Lesson9, create a new application class named ArrayDemo. **Type ArrayDemo** as shown below:

CODE TO TYPE: ArrayDemo

```
class ArrayDemo {
     public static void main(String[] args) {
          int[] array1;              // declares an array of integers

          array1 = new int[8];      // allocates memory for 8 integers

          array1[0] = 42; // initialize first element
          array1[1] = 15; // initialize second element
          array1[2] = 74; // etc.
          array1[3] = 6;
          array1[4] = 32;
          array1[5] = 150;
          array1[6] = 724;
          array1[7] = 66;

          System.out.println("Size of the array array1 is " + array1.length);
          System.out.println();
          System.out.println("Element at index 1: " + array1[1]);
          System.out.println("Element at index 4: " + array1[4]);
          System.out.println("Element at index 6: " + array1[6]);
          System.out.println("Element at index 8: " + array1[8]);
     }
}
```

Save and **Run** it. You'll see this:



```
<terminated> ArrayDemo [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Sep 16, 2008 12:14:21 PM
Size of the array array1 is 8

Element at index 1: 15
Element at index 4: 32
Element at index 6: 724
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 8
        at ArrayDemo.main(ArrayDemo.java:21)
```

Cool. The two most common exceptions any programmer ever sees are:

1. ArrayIndexOutOfBoundsException
2. NullPointerException

In fact, these kinds of programming mistakes happen so often that Java has a whole class for them!

API Click on the API link, go to the **java.lang** package, scroll down to the Exception Summary, find **ArrayIndexOutOfBoundsException**, and click on it. Read some of its description--I know you're interested! We'll discuss **Exception** classes in detail in the next course. For now, it's enough to know that we don't want **ArrayIndexOutOfBoundsExceptions**, so we should make sure we provide proper indices (smaller than the array size).

Note An array's (*arrayName*) last possible element is **always** *arrayName*.length -1

A **NullPointerException** is most commonly encountered when dealing with arrays because an element of an array is **null** instead of holding a valid reference to an object. If we tried to access **arrayName[0]** and **arrayName[0]** was **null**, rather than having a valid reference, we would get the runtime exception, **NullPointerException** thrown by the Java Virtual Machine. The way to avoid this is to always preface accessing array elements that are object references, with something like: if(arrayName[0] != null) ... Substituting as needed to match your code of course.

Syntax and Java Implementation for Arrays

Declaring a Variable to Refer to an Array

You can declare arrays of any type, **but** all of the elements in an array must be of the **same** type.

Find the lines of code that **declare** the arrays in your **BeatleJuice.java** and **ArrayDemo.java** classes:

EXAMPLE: Array Declarations

```
int [] array1; //this is an array of ints called array1
String [] beatles; //this is an array of Strings called beatles
// The placement of the square brackets [ ] to denote an array is optional--afte
r the type (int []) or after the arrayName (char s[])
// However, convention discourages this form in favor of the brackets after the
array type so the array designation appears with the type designation.
```

Open your **ArrayDemo** class and change the code as shown below:

CODE TO TYPE:

```
class ArrayDemo {
    public static void main(String[] args) {
        int array1[];                // declares an array of integers

        array1 = new int[8];        // allocates memory for 8 integers

        array1[0] = 42; // initialize first element
        array1[1] = 15; // initialize second element
        array1[2] = 74; // etc.
        array1[3] = 6;
        array1[4] = 32;
        array1[5] = 150;
        array1[6] = 724;
        array1[7] = 66;

        System.out.println("Size of the array array1 is " + array1.length);
        System.out.println();
        System.out.println("Element at index 1: " + array1[1]);
        System.out.println("Element at index 4: " + array1[4]);
        System.out.println("Element at index 6: " + array1[6]);
        System.out.println("Element at index 8: " + array1[8]);
    }
}
```

Save and **Run** it. Now, just for fun, change that line so that there's a space between the type (**int array1**) and the brackets **[ ]**, like this:

```
int array1 [];
```

Save and **Run** it again. You can see that having a space before the brackets [] doesn't affect your result. You didn't get any errors and all is well.

Tip In Java, you cannot specify the size of an array when you **declare** it, only when you **create** it.

If you have not done so already, go to **ArrayDemo** and **remove** the line that caused the **ArrayIndexOutOfBoundsException**.

Change this line: `int array1 [];` so it reads: `int [8] array1;`

It looks like Java doesn't like that. Let's change it back to: `int array1 [];`

Ahh, all better now.

Memory

Why did Java complain? Because Java was made to be **cross-platform**, it can't actually set up memory on machines at **compile-time** because it doesn't know what kind of machine it will be running on. When you **declare** a variable to be of a certain type, the Java compiler can check for syntax problems associated with the declared variable. But Java cannot reserve space in memory when it doesn't know what machine it will be run on! The length of an array is established when the array is actually created--at **run-time**--instead.

Find these two important lines at the beginning of your **ArrayDemo** class:

`int [] array1 ;` declares that **array1** is an array of **int**s.

`array1 = new int[8];` **creates** array1 and leaves space for 8 **int**s **at run-time**.

After creation, the array's length is fixed. So it makes sense that the indices of arrays start at 0. To make our numbers easier to work with, we'll give Java the line **array1 = new int[10];**. We know that **int** variables each take up 32 bits of space. So for 10 **int**s, we'll need 320 bits. The table below is a representation of an array and the memory location and bits used in memory for an array that holds 10 integers. In the table, X represents the memory location of the array:

Indices:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ARRAY array1:	42	15	74	6	32	150	724	66	53	100
Memory Locations	x = x+ (0*32)	x+ (1*32)	x+ (2*32)	x+ (3*32)	x+ (4*32)	x+ (5*32)	x+ (6*32)	x+ (7*32)	x+ (8*32)	x+ (9*32)
Uses bit locations:	x to (x+32)	(x+32) to (x+64)	(x+64) to (x+96)	(x+96) to (x+128)	(x+128) to (x+160)	(x+160) to (x+192)	(x+192) to (x+224)	(x+224) to (x+256)	(x+256) to (x+288)	(x+288) to (x+320)

When Java sees that we want **array1[6]**, it locates **array1** and then moves over **6*32** (or 192) more bits. Most operating systems are 32 bit, although newer 64-bit versions are becoming available. But until Java is available in a **64-bit** version, it will continue to process arrays this way. So all of the objects in an array must be the same type, to indicate to Java that they're all the same size. And we need to specify the size of the array when we create it, so Java knows how much space to allow.

Understanding this process also explains why we got the **ArrayIndexOutOfBoundsException** exception in our previous example; we created the array to be of a certain fixed size and we asked Java to get something with an index bigger than the array allowed.

This error cannot be identified by the compiler while you are writing the code. When you actually **run** the code, you give it the bit locations. This is called **dynamic allocation** of memory.

Array boundaries are checked at run time to avoid overflowing a stack and corrupting memory. Again, because Java is cross-platform, it cannot allocate memory until it is actually running on a specific machine. Therefore, all objects in Java are **dynamically allocated**
and ArrayIndexOutOfBoundsException, **NullPointerException** and such are **run-time exceptions** rather than compilation (or syntax) errors.

Note

```
int [] a = new int [5] /* leaves space for 5 ints (32* 5 bits) */
```
however
```
T [] b = new T [5] /*leaves space for 5 references to Ts--not 5 Ts*/
```

Primitive data types always have exactly the same known size, but different classes may occupy different amounts of space. References to memory locations, however, are standardized.

Loops

Lesson Objectives

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

- write a for loop.

Repetition: for Loops

Introduction to Loops

Most humans don't enjoy performing repetitive tasks and will go to great lengths to avoid doing them. But that's not always such a bad thing. In fact, the desire to avoid such drudgery has provided us with motivation to invent all kinds of useful tools.

Early on, programmers recognized the computer's ability to execute repetitive tasks. Rather than writing the same piece of code over and over again and incoprating whatever small changes they wanted (even cutting and pasting gets old quickly), they provided control constructs that allowed them to write code to repeat a process...to **loop** around.

There are three ways in Java to do **loops** or **repetition of code**:

1. **for** statements
2. **while** statements
3. **do-while** statements

Let's go back to the example we used in arrays and check out the program first **without** using a loop. Eventually you'll feel the power of loop constructs!

Make a new **java2_Lesson10** project in the **Java2_Lessons** working set, and create a new **Beatlejuice.java** class. **Edit Beatlejuice.java** to look like the code below:

CODE TO TYPE: Beatlejuice

```
class Beatlejuice {
    public static void main(String[] args) {
        String[] beatles;
        beatles = new String[5];

        beatles[0] = "John Lennon";
        beatles[1] = "Paul McCartney";
        beatles[2] = "George Harrison";
        beatles[3] = "Ringo Starr";
        beatles[4] = "George Martin";

        System.out.println("Element at index 0: " + beatles[0]);
        System.out.println("Element at index 1: " + beatles[1]);
        System.out.println("Element at index 2: " + beatles[2]);
        System.out.println("Element at index 3: " + beatles[3]);
        System.out.println("Element at index 4: " + beatles[4]);

        System.out.println("\nSize of the beatles array is " + beatles.length);
    }
}
```

Save and **Run** it.

Of course we want to give each element in the array its specific name, but how many times do we have to type this:

System.out.println("Element at index *someNumber*: " + beatles[*someNumber*]);

Te-di-ous! I think we can do better. Edit it as shown by making the multiple **System.out.println** calls into a loop:

CODE TO EDIT: Beatlejuice

```
class Beatlejuice {
    public static void main(String[] args) {
         String[] beatles;
         beatles = new String[5];

         beatles[0] = "John Lennon";
         beatles[1] = "Paul McCartney";
         beatles[2] = "George Harrison";
         beatles[3] = "Ringo Starr";
         beatles[4] = "George Martin";

         for (int i=0; i < beatles.length; i++){
           System.out.println("Element at index " + i + ": " + beatles[i]);
         }

         System.out.println("\nSize of the beatles array is " + beatles.length);
    }
}
```

Save and **Run** it.

Java gives us exactly the same output as the original, but we did much less work. We just reduced five lines of code and to three. Cool!

Now imagine if everyone who WANTED to be a Beatle suddenly WAS a Beatle and we wanted to include all Beatles in our code. We would have to write thousands of new lines of code to do that. Or we could use the handy dandy **for** loop to do it in just two lines. **DO NOT** do this right now, because if you did, the loop would go on forever and crash our server, and we'd all be really unhappy. Just take a look:

DON'T DO THIS: Code to List the Names of thousands of beatles

```
beatles = new String[9999];

for (int i=0; i < beatles.length; i++){
    System.out.println("Element at index " + i + ": " + beatles[i]);

}
```

No matter **how many** items there are in an array, one or a billion, we would still only need to have one line **for (int i=0; i < *item*.length; i++)**. Now that's a powerful tool.

So how does it work specifically? Let's look again at the code:

OBSERVE: For Loop Code:

```
for (int i=0; i <
beatles.length; i++)
    System.out.println("Element at index " + i + ": " + beatles[i]);
```

Of course every **for statement** starts with **for**. The starting and ending points are defined in the parentheses. The **increment** in this example is **i** (i, j, and k are traditionally used for increments). The starting point is **0**. So **i** starts at **0** and the **i++** increases **i** by **one** every step of the loop, until it gets to **beatles.length** (which in this case is **5**).

So, first **i**=**0** and it prints "Element at index 0: John Lennon" Then i=1 and it prints "Element at index 1: Paul McCartney" and so on.

That's the way all for loops work. No matter which loop construct you choose to use, **all** loops consist of three major parts:

1. Initialize variable to begin.
2. Increment (to get to the next step in the loop).
3. Condition to check to end the loop.

In Java, the conditions to check in order to end the loop must return **boolean** values (similar to decision statements). All three loop constructs in Java (do-while, while, for) must contain these three parts, but each construct executes them at different points in the code. If those three major parts are missing, it may result in infinite loops.

for

for loops have all the three components in one construct:

for(initialize; condition; increment} {
statements;
...
}

It works like this:

1. The **for** loop starts by initializing the "loop variable." This is done once at the beginning of the loop.
2. The loop terminates when the condition becomes **false**.
3. The increment expression is executed after each iteration of the loop body.

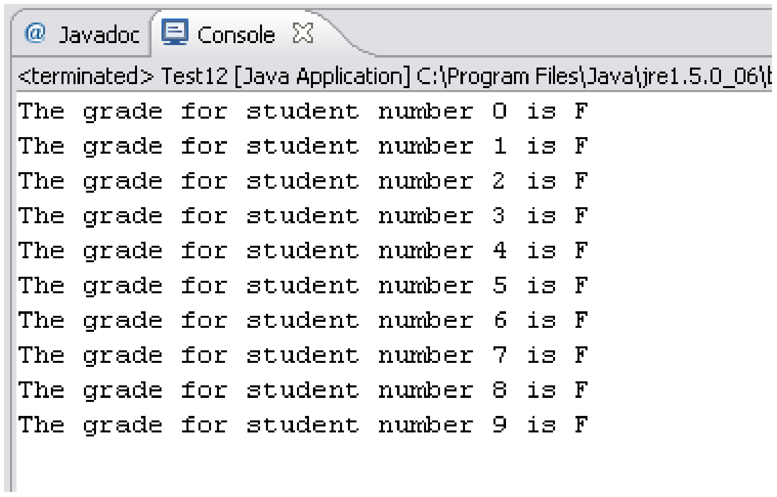
for loops may be the most commonly used loop construct because they make working with arrays easier. Let's try one out by creating an example of my favorite grading program. Have you ever heard that teachers grade by throwing the submitted papers down a flight of steps and each paper's grade is determined by the step it lands on? This grading example is something like that.

Go to the **temp** Project's default package, and edit the **Test.java** class as shown below:

CODE TO TYPE: Test

```
class Test {
    public static void main(String[] args){
        String [] studentGrade = new String[30];
        for (int i=0; i < studentGrade.length; i++){  // begin for body (or bloc
k)
            switch (i % 5) {                                // begin switch bloc
k
                case 0: studentGrade[i] = "A";
                case 1: studentGrade[i] = "B";
                case 2: studentGrade[i] = "C";
                case 3: studentGrade[i] = "D";
                case 4: studentGrade[i] = "F";
            }                                            // end switch
            System.out.println("The grade for student number " + i + " is " + st
udentGrade[i]);
        }                                                // end for
    }                                                    // end main method
}                                                     // end Test class
```

Save and **Run** it.


Javadoc Console

```
<terminated> Test12 [Java Application] C:\Program Files\Java\jre1.5.0_06\
The grade for student number 0 is F
The grade for student number 1 is F
The grade for student number 2 is F
The grade for student number 3 is F
The grade for student number 4 is F
The grade for student number 5 is F
The grade for student number 6 is F
The grade for student number 7 is F
The grade for student number 8 is F
The grade for student number 9 is F
```

Wow, that's a pretty rough grading scale. Don't panic--you don't even get those kinds of grade in this course. I only wanted to draw your attention to the importance of **break** in **switch** statements.

Edit each of the **case**s to have a **break;** after it, like you see in the code below:

CODE TO EDIT: Test

```
class Test {
    public static void main(String[] args) {
    String [] studentGrade = new String[30];
        for (int i=0; i < studentGrade.length; i++){  // begin for body (or bloc
k)
            switch (i % 5) {                              // begin switch block
                case 0: studentGrade[i] = "A"; break;
                case 1: studentGrade[i] = "B"; break;
                case 2: studentGrade[i] = "C"; break;
                case 3: studentGrade[i] = "D"; break;
                case 4: studentGrade[i] = "F"; break;
            }                                             // end switch
            System.out.println("The grade for student number " + i + " is " + st
udentGrade[i]);
        }                                                 // end for
    }                                                    // end main method
}                                                       // end Test class
```

Save and **Run** it.

Let's trace this code:

OBSERVE: Test Class

```
class Test {
    public static void main(String[] args){
        String [] studentGrade = new String[30];
        for (int i=0; i < studentGrade.length; i++){      // begin for body (or b
lock)

            switch (i % 5) {                               // begin switch block
                case 0: studentGrade[i] = "A"; break;
                case 1: studentGrade[i] = "B"; break;
                case 2: studentGrade[i] = "C"; break;
                case 3: studentGrade[i] = "D"; break;
                case 4: studentGrade[i] = "F"; break;
            }                                              // end switch
            System.out.println("The grade for student number " + i + " is " + st
udentGrade[i]);
        }                                                 // end for
    }                                                    // end main method
}                                                       // end Test class
```

Here we define an array **studentGrade** that is a string array of length **30**. For each number between 0 and 29 it enters the switch statement. For each number **i** it calculates **i%5**. i%5 can only be 0 or 1 or 2 or 3 or 4, and for each of these it sets studentGrade[i] = A or B or C or D or F. For instance, when i=7, i%5 is 2, and so studentGrade[7] = C.

Grading is a lot easier with a tool like this. In a "jumbo" class of 150 students, the only thing you need to change to give that many more grades is the length value of the array:

CODE TO EDIT: Test

```
class Test {
    public static void main(String[] args){
        String [] studentGrade = new String[150];
        for (int i=0; i < studentGrade.length; i++){      // begin for body (or b
lock)
            switch (i % 5) {                                 // begin switch block
                case 0: studentGrade[i] = "A"; break;
                case 1: studentGrade[i] = "B"; break;
                case 2: studentGrade[i] = "C"; break;
                case 3: studentGrade[i] = "D"; break;
                case 4: studentGrade[i] = "F"; break;
            }                                                // end switch
            System.out.println("The grade for student number " + i + " is " + st
udentGrade[i]);
        }                                                   // end for
    }                                                      // end main method
}                                                         // end Test class
```

Save and **Run** it.

That was pretty fast, huh? Though this is kind of a silly example (a good teacher would never do this!), it gave us a chance to see the **for** in action in an array.

Let's look at the **main()** method of our Test class to see how many times the loop variable **i** is used within the loop, to identify which element in the array (**studentGrade[i]**) it's handling.

Programs usually get their **input** from some outside source; it can vary in length. You can get your input using a loop, process the input using a loop, or give the output using a loop. Using loops means you don't have to worry about input and output issues, only what goes on inside the loop.

Tip Arrays and **for** loops make a good team.

Alright. Time to make our first slinky and investigate more of the particulars of the **for** construct. In the java2_Lesson10 project, create a new Applet class named **Slinky**. **Type** **Slinky** as shown below:

CODE TO TYPE: Slinky

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

    public class Slinky extends Applet implements ActionListener {
        private TextField countInput;
        private int howManyCircles = 0;

    public void init() {

        Label l = new Label("How many circles?");
        add(l);

        countInput = new TextField(3);
     add(countInput);
     countInput.addActionListener(this);

        Button reset = new Button("Reset");
        add(reset);
        reset.addActionListener(this);
    }

     public void paint(Graphics g) {
        int x = 20;
        int y = 20;
        for (int count=1; count <= howManyCircles; count ++)
           g.drawOval(x+count*5, y+count*5, 50, 50);
     }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() instanceof Button)
            {
         howManyCircles = 0;
       countInput.setText("");
            }
         else if (event.getSource() instanceof TextField)
            if (countInput.getText().length() == 0)
             howManyCircles = 0;
         else
         howManyCircles = Integer.parseInt(countInput.getText());
      repaint();
    }
}
```

Save and **Run** it.

Experiment and test lots of numbers of varying sizes. Make sure you throw in some big ones--you'll probably want to expand the output window!

We used quite a few different tools in this code:

1. The **init()** method of the **Applet** was called first and made the GUI.
    - *Control constructs:* Some sequential programming and method invocations.
2. The **actionPerformed(ActionEvent event)** method implemented the **ActionListener** and waited for us to give input.
    - *Control constructs:* Some decision programming with nested **if**s and method invocations.
3. The **paint(Graphics g)** method is called from the **actionPerformed(ActionEvent event)** method (via **repaint()**), and does the work.
    - *Control constructs:* A **for** loop ... **for (int count=1; count < = howManyCircles; count ++)**.

In this **for** loop, where did we: initialize the variable, increment it, and check for the end of looping? Which variables served which purpose?

The loop uses one outside variable (**howManyCircles**) to determine how many circles to make (from user input), but once that input is set, all of the information for the loop is within the **paint** method.

```
public void paint(Graphics g) {
    int x = 20;
    int y = 20;
    for (int count=1; count <= howManyCircles; count ++)
        g.drawOval(x+count*5, y+count*5, 50, 50);
}
```

The x and y variables are declared as **int** within the **paint** method. Since they are used to indicate where to draw the circles, they are made local to the **paint** method. No other methods need this information, so they do not need to be **instance variables**.

Can we make the code shorter by making the two lines into one? Edit the **Slinky** class's **paint** method as shown:

CODE TO TYPE: Slinky.paint()

```
public void paint(Graphics g) {
    int x , y = 20;
    for (int count=1; count <= howManyCircles; count ++)
        g.drawOval (x+count*5, y+count*5, 50, 50);
}
```

Save and **Run** it.

Okay, now edit it again so it looks like this:

CODE TO TYPE: Slinky.paint()

```
public void paint(Graphics g) {
    int x = 20, y = 20;
    for (int count=1; count <= howManyCircles; count ++)
        g.drawOval (x+count*5, y+count*5, 50, 50);
}
```

Save and **Run** it. Looks fine, runs fine. Good.

start

The beginning of the **for** statement initializes the loop counter variable, like this:

```
for(int i=0; i < arrayName.length; i++)
```

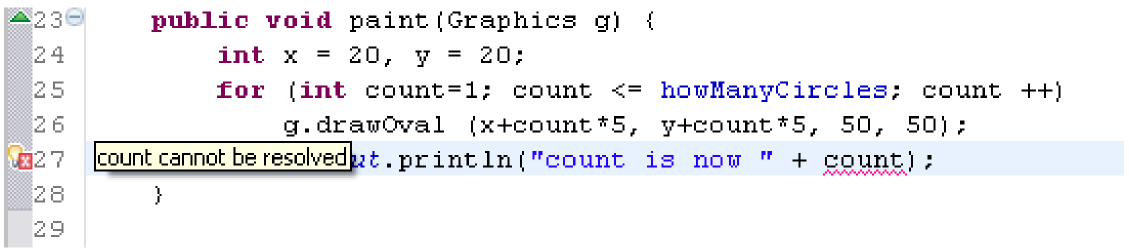
OBSERVE: Slinky paint method

```
public void paint(Graphics g) {
    int x = 20, y = 20;
    for (int count=1; count <= howManyCircles; count ++)
        g.drawOval (x+count*5, y+count*5, 50, 50);
}
```

The variable **count** used in this **for** is called a **loop variable** or **loop counter**. It is declared **inside** the loop initialization expression. Because of this, the scope of the variable **count** is **only** within the block of the loop. In this example, the block of the loop is actually only one line that is repeated. When the loop is over, Java will have no clue what **count** is. People often forget this and want to print things about the loop variable outside of a **for** loop. It might help to see an example of this common mistake. Edit the **Slinky** class's **paint** method by adding a line of code:

CODE TO TYPE: Slinky.paint()

```
public void paint(Graphics g) {
    int x = 20, y = 20;
    for (int count=1; count <= howManyCircles; count ++)
        g.drawOval (x+count*5, y+count*5, 50, 50);
    System.out.println("count is now " + count);
}
```



```
public void paint(Graphics g) {
    int x = 20, y = 20;
    for (int count=1; count <= howManyCircles; count ++)
        g.drawOval (x+count*5, y+count*5, 50, 50);
count cannot be resolved ut.println("count is now " + count);
}
```

You never believe me, do you! Okay, edit the code so the loop variable is declared **outside** of the loop:

CODE TO EDIT: Slinky.paint()

```
public void paint(Graphics g) {
    int count;
    int x = 20, y = 20;
    for (count=1; count <= howManyCircles; count ++)
        g.drawOval (x+count*5, y+count*5, 50, 50);
    System.out.println("count is now " + count);
}
```

Save and **Run** it. Looks fine, runs fine.

The value of **count** printed is 1 more than we input, which makes sense because the loop stops **after** count <= howManyCircles.

This information could be handy when we **need** to know something about the loop counter after we exit the loop. However, it is usually good programming practice to limit the scope of a variable to only what is needed. So, if the loop variable in a **for** statement is not needed outside of the loop, it's best to declare the variable in the initialization expression.

Because the scope of a loop variable is limited to within the loop and is being used specifically as a counter for the loop, loop variables are often named *i, j, k,* or *count*.

stop

-

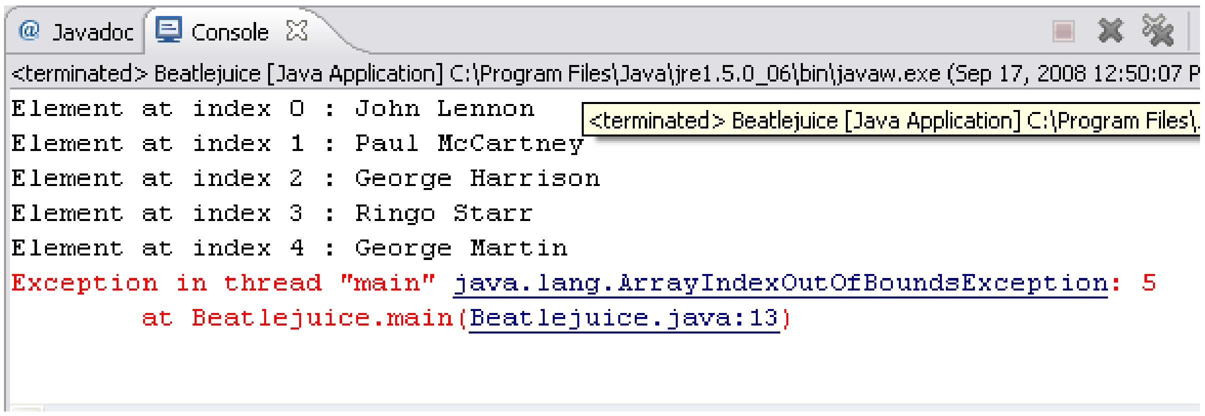
The second part of the for statement controls the extent of the loop, as in
for(int i=0; **i < arrayName.length**; i++).
To stop the loop, this must return a type **boolean**. The most common error programmers make in the *stop* expression is using the `arrayName.length` when they want to stop after the last element in the array.

Since the **Slinky** class did not use an array, we'll demonstrate the **stop** with **Beatlejuice.java**. Edit the **Beatlejuice.java** (in java2_Lesson10) class as shown:

CODE TO EDIT: BeatleJuice

```
class Beatlejuice {
    public static void main(String[] args) {
        String[] beatles;
        beatles = new String[5];

        beatles[0] = "John Lennon";
        beatles[1] = "Paul McCartney";
        beatles[2] = "George Harrison";
        beatles[3] = "Ringo Starr";
        beatles[4] = "George Martin";

        for (int i=0; i <= beatles.length; i++)
            System.out.println("Element at index " + i + " : " + beatles[i]);
        System.out.println("Size of the beatles array is " + beatles.length);
    }
}
```

Save and **Run** it.



```
<terminated> Beatlejuice [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Sep 17, 2008 12:50:07 P
Element at index 0 : John Lennon
Element at index 1 : Paul McCartney
Element at index 2 : George Harrison
Element at index 3 : Ringo Starr
Element at index 4 : George Martin
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Beatlejuice.main(Beatlejuice.java:13)
```

The program runs until it tries to access beatles[beatles.length]. Nasty. You never want your programs to have errors or exceptions.

Take that **=** sign back out (and save).

We did that because the **length** of an array is one more than the last index. (Because the first index is 0, the last must be length-1.)

increment

The last part of the **for** statement controls the size of the "steps" between each occurrence of the loop, as in for(int i=0; i < arrayName.length; **i++**).

We want to iterate through all of the elements. Normally, we go through an array one at a time. This **increment** expression doesn't have to be **i++**; it can be any math expression. If there is not an array, we need to increment according to the loop's needs (for example, moving over a certain number of pixels each time).

Edit the **Beatlejuice** class and try changing the **main()** method's **for** statement to reflect each of the examples shown. You do not need to Save and Run after each, but if you want to run them make sure there are no reported errors first. Experiment with the variety of expressions that are allowed by typing them in and seeing there are no errors:

CODE TO TEST in BeatleJuice

```
for (int i=0; i < beatles.length; i = i+1)

for (int i=0; i < beatles.length; i *= 3)

for (int i=0; i < beatles.length; i = 5*i+1)

for (int i=beatles.length; i > 0; i--)

for (int i=beatles.length;; i > beatles.length;  i = beatles.length - 2*i)

for (int i=0; i < beatles.length; i = (int)Math.PI * i)
```

As shown in the examples, it is also fine to **decrement** at the "*increment*" component (meaning decreasing), as long as you make sure you approach the **stop** state with your decrement. If you don't, you could get an infiniite loop!

Creating Arrays

Lesson Objectives

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

- create an array.
- number the values in an array.
- use class headers.
- use loops and logic expressions to evaluate data held in arrays.
-

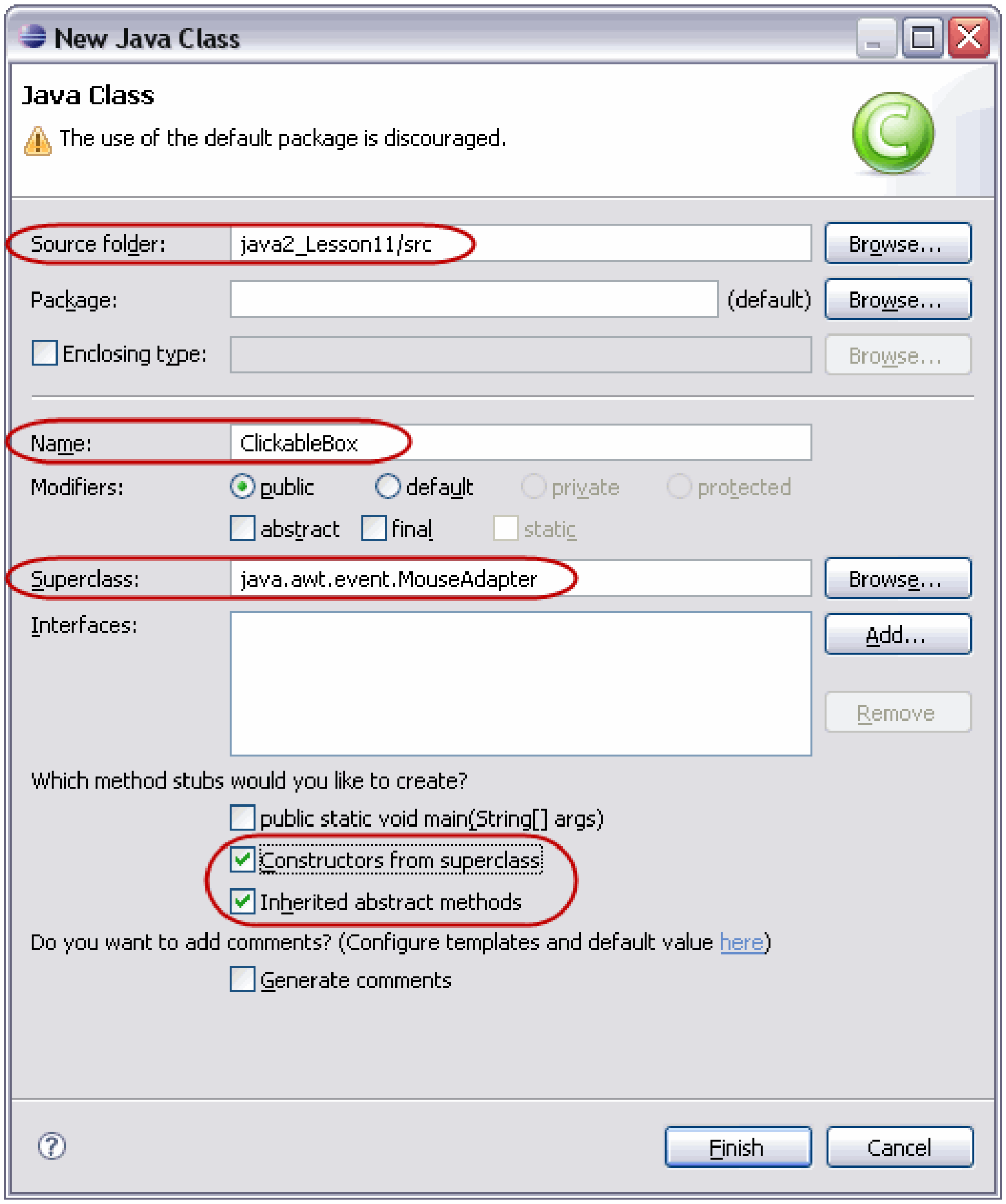
Creating, Initializing, and Accessing

Let's look at values being set in an array. Create an example using a simple graphical display of boxes, and pay special attention to desiging the classes just right.

Ideally, an object should know everything about itself and do as much for itself as possible. We'll be using an object to represent the box in our example. so we need to consider the properties of a box.

A box is, of course, a rectangle. and ultimately, we want the box to tell us if it's been clicked. We'll name the first class **ClickableBox** (even though it won't be clickable just yet). We want this class to implement **MouseListener**. However, we don't really need to implement all of the methods in the MouseListener, so rather than using the default **implement Interface**, use an **Adapter**. Every Listener interface in the Java API library that has more than one method to be implemented, has a corresponding **Adapter class** that can be *extended*. These Adapter classes implement all of the methods of the listener that have empty bodies (which means that they haven't been implemented yet).

Make a new **java2_Lesson11** project in the Package Explorer. Put it in your **Java2_Lessons** working set. Create a new class in the project using the settings in the image below:



The use of the default package is discouraged.

Source folder: java2_Lesson11/src

Package: (default)

Enclosing type:

Name: ClickableBox

Modifiers: [✓ public] default private protected

abstract final static

Superclass: java.awt.event.MouseAdapter

Interfaces:

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

The resulting class will look like this:

```
import java.awt.event.MouseAdapter;

public class ClickableBox extends MouseAdapter {

    public ClickableBox() {
        // TODO Auto-generated constructor stub
    }
}
```

So, what variables will we need? Well, a box is a rectangle, so at the very least we'll need x, y, height, and width. We are dealing with screen coordinates (as opposed to float or long), so **int** variables will work just fine. Add the code in blue and delete the code in red as shown below:

CODE TO TYPE: ClickableBox

```
import java.awt.event.MouseAdapter;

public class ClickableBox extends MouseAdapter {

    private int x, y, width, height;

    public ClickableBox(int x, int y, int width, int height) {
        // TODO Auto-generated constructor stub
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

The code we added gives the box definition and allows the constructor **Clickablebox( )** to pass information along to the instance variables.

Currently, we have the basic definition of a box. We want this box to be able to draw itself on the screen. There are various ways that Java can draw a Graphics object; two methods we might use are **java.awt.Graphics.drawRect()** and **java.awt.Graphics.fillRect()**. Both of these methods will give us functionality. We also need to know what color to draw in. Type the code as shown in **blue** below:

CODE TO EDIT: ClickableBox

```
import java.awt.Color;
import java.awt.event.MouseAdapter;

public class ClickableBox extends MouseAdapter {

    private int x, y, width, height;
    private Color borderColor, backColor;

    public ClickableBox(int x, int y, int width, int height, Color borderColor, Color b
ackColor) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.borderColor = borderColor;
        this.backColor = backColor;
    }
}
```

Later, we'll change the color of a Graphics object that's passed to us and set it back when we're done. Also, while we're at it, we'll add something that will give us some control over what is drawn. Edit your code so it looks like this:

CODE TO EDIT: ClickableBox

```
import java.awt.Color;
import java.awt.event.MouseAdapter;

public class ClickableBox extends MouseAdapter {

    private int x, y, width, height;
    private Color borderColor, backColor, oldColor;
    private boolean drawBorder;

    public ClickableBox(int x, int y, int width, int height, Color borderColor,
        Color backColor, boolean drawBorder) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.borderColor = borderColor;
        this.backColor = backColor;
        this.drawBorder = drawBorder;
    }
}
```

We'll add more later but right now, let's make this class do something. Add a **draw()** method so that this class can draw itself on a Graphics object. Edit the code as shown in **blue** below:

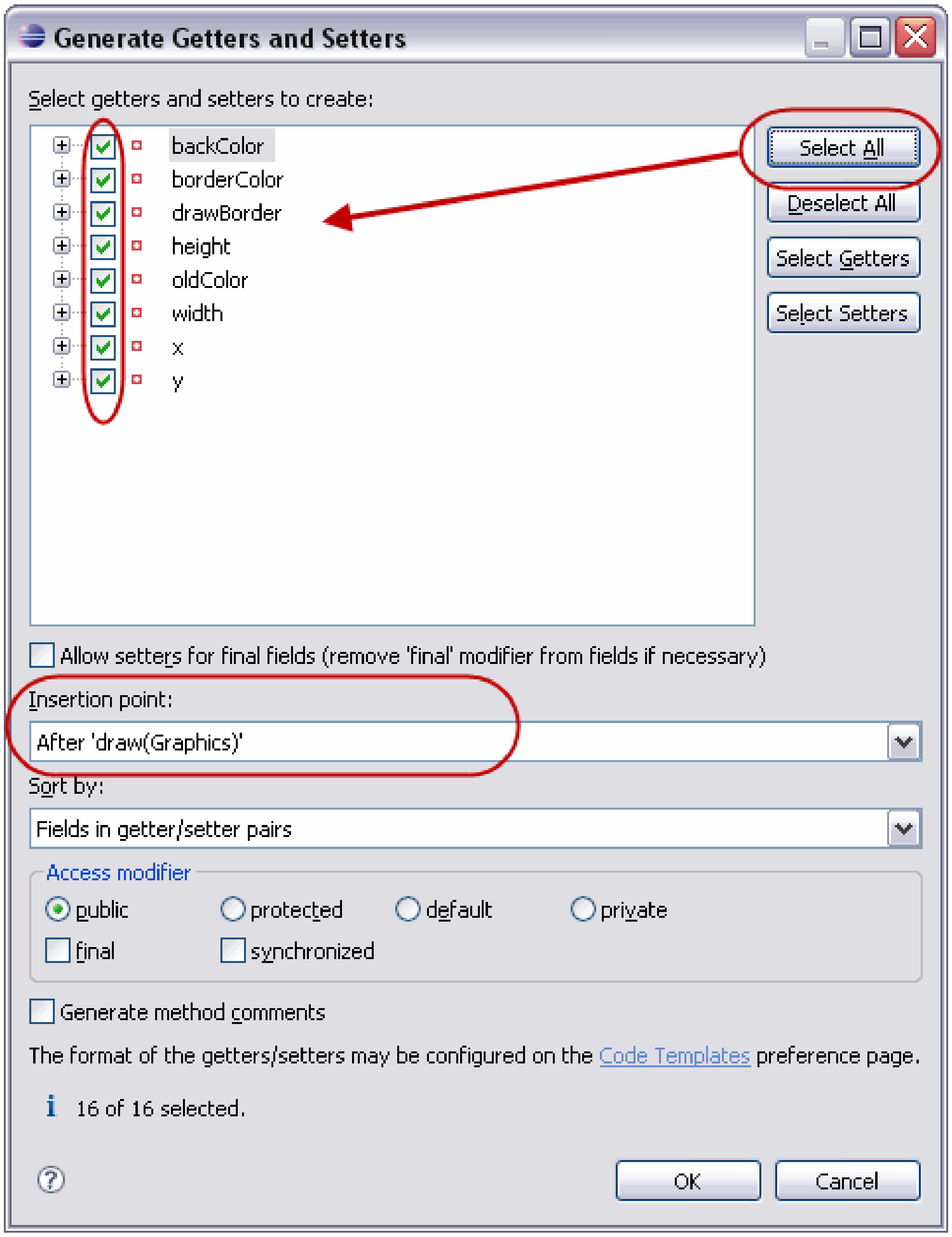
CODE TO TYPE: ClickableBox

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;

public class ClickableBox extends MouseAdapter {

    private int x, y, width, height;
    private Color borderColor, backColor, oldColor;
    private boolean drawBorder;

    public ClickableBox(int x, int y, int width, int height, Color borderColor,
        Color backColor, boolean drawBorder) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.borderColor = borderColor;
        this.backColor = backColor;
        this.drawBorder = drawBorder;
    }

    public void draw(Graphics g) {
        oldColor = g.getColor();
        g.setColor(backColor);
        g.fillRect(x, y, width, height);
        if(drawBorder) {
            g.setColor(borderColor);
            g.drawRect(x, y, width, height);
        }
        g.setColor(oldColor);
    }
}
```

Save it.

In the **draw()** method, we save the current color of the Graphics object. Then we fill a rectangle with the background color, then *if* drawBorder flag is set (by the constructor), draw the border on top of the filled rectangle. When done, we put the Graphics object color back to the way we found it using g.setColor(**oldColor**).

Now that we have most of our instance data, we can generate **get** and **set** methods (getters and setters) for each of the fields. Eclipse will do this for you. Right-click the ClickableBox.java class in the Package Explorer, select **Source**, and choose **Generate Getters and Setters**.


Generate Getters and Setters

Select getters and setters to create:

- [✓] backColor
- [✓] borderColor
- [✓] drawBorder
- [✓] height
- [✓] oldColor
- [✓] width
- [✓] x
- [✓] y

[ ] Allow setters for final fields (remove 'final' modifier from fields if necessary)

Insertion point: After 'draw(Graphics)'

Sort by: Fields in getter/setter pairs

Access modifier: [✓ public] protected default private

[ ] final [ ] synchronized

[ ] Generate method comments

The format of the getters/setters may be configured on the Code Templates preference page.

16 of 16 selected.

This places **get** and **set** methods for each instance field selected after the **draw()** method in your class.

We have enough of this project done to start testing it. So let's create an Applet. Create a new class named **OneDArrayApplet** in your applet. This class should use **java.applet.Applet** as its Superclass.

The resulting class looks like this:

```
import java.applet.Applet;

public class OneDArrayApplet extends Applet {

}
```

Hmm. It's not much of a class yet, but we'll fix that. Since this is ultimately a lesson on Arrays, let's use one now. We're about to hit you with a lot of stuff right here. Just stay with it and we'll break it down afterward.

CODE TO TYPE: ADD The Code In Blue

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

public class OneDArrayApplet extends Applet {
    private final int START_X = 25;
    private final int START_Y = 70;
    private final int BOX_WIDTH = 60;
    private final int BOX_HEIGHT = 20;

    private ClickableBox[] boxes = new ClickableBox[6];

    public void init() {
        for (int i = 0; i < boxes.length; i++) {
            boxes[i] = new ClickableBox(START_X, START_Y + i * BOX_HEIGHT, BOX_WIDTH,
               BOX_HEIGHT, Color.black, Color.red, true);
        }
    }

    public void paint(Graphics g) {
        for(int i = 0; i < boxes.length; i++) {
            boxes[i].draw(g);
        }
    }
}
```

Let's look at the code in detail:

Instance Fields:

```
private final int START_X = 25;
private final int START_Y = 70;
private final int BOX_WIDTH = 60;
private final int BOX_HEIGHT = 20;

ClickableBox[] boxes = new ClickableBox[6];
```

The fields marked **final** are **constants**, which means they will *never* change once they're set. In Java, a field marked final may only have its value set one time. It doesn't matter where in the class it is set, but it can only be set once. The final fields in this class represent the starting location and size of the boxes we are going to draw.

The colored line in the code is where we are declaring an array. **ClickableBox[ ] boxes** indicates that we have declared a variable named **boxes** and that it is going to contain references to **ClickableBox** objects. The brackets **[ ]** indicate that this is an array variable. As such, it can contain more than one value, using indexes starting with zero [0] up to the length of the array n, minus one [n-1].

We set the boxes variable **= new ClickableBox[6];**, which means that we are creating a new instance of an array object and that the array can only contain up to 6 references to **ClickableBox** objects. The index values we can use in accessing this array are **boxes[0]** through **boxes[5]**, because all array indexes start with zero, and this array can hold a total of six object references:

OBSERVE: The init() Method:

```
public void init() {
    for (int i = 0; i < boxes.length; i++) {
        boxes[i] = new ClickableBox(START_X, START_Y + i * BOX_HEIGHT, BOX_WIDTH,
             BOX_HEIGHT, Color.black, Color.red, true);
    }
}
```

Note Each element of the array is an Instance of ClickableBox.

In the **init()** method above, we created some ClickableBox objects. The **for** loop will loop from zero to the size of the array minus one. The section of code **boxes.length** indicates that we are accessing the **length** field of the **boxes** object. All arrays have a **length** field indicating the maximum capacity of the array. This is not the last index in the array, but its total capacity; therefore, the last index in the array is **length - 1**.

The **for loop** loops from **0** through **5** (< length) and on each pass, **i** increments by one. On each pass, we assign **boxes[i] = new ClickableBox(...)**. We pass **START_X** as the **x** location of all of the boxes. This means the boxes will line up vertically. We pass **START_Y + i * BOX_HEIGHT** as the **y** location of the box. On each pass, **i** increments. When **i** is zero, the value passed will be **70**. When **i** is one, the value passed will be **70 + 20**. In successive passes, **i** will be **70 + 40**, then **70 + 60**, and so on. This will give the effect of stacking the boxes vertically.

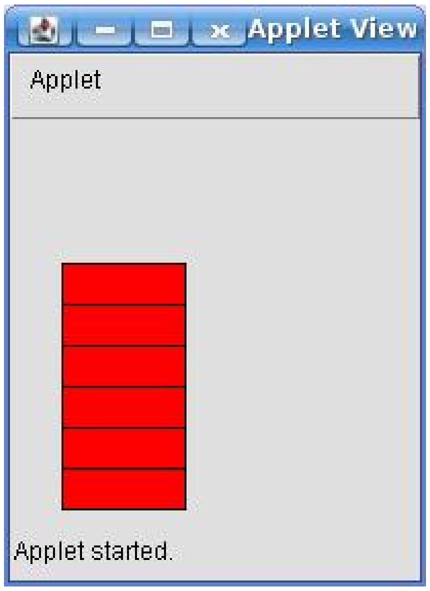
The other arguments being passed are the size of the boxes, the borderColor (black), and the backColor (red). Finally, we tell the ClickableBox objects to draw their border by passing **true** as the drawBorder argument.

Observe: The paint() Method:

```
public void paint(Graphics g) {
    for(int i = 0; i < boxes.length; i++) {
        boxes[i].draw(g);
    }
}
```

All of the work in this method is done in **boxes[i].draw(g)**. We loop through each **ClickableBox** reference and tell that object to draw itself on the Graphics object we are passing to it--the Graphics object we received in the **paint()** method formal argument.

Save and **Run** the Applet to check out your results:



Red boxes are cool, but wouldn't it be nice to get some other colors in there? Let's create another array. This one will hold some Color objects for us. While we're at it, let's create a new method to change the colors of the boxes. Edit your code as shown below:

CODE TO EDIT: OneDArrayApplet

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

public class OneDArrayApplet extends Applet {
    private final int START_X = 25;
    private final int START_Y = 70;
    private final int BOX_WIDTH = 60;
    private final int BOX_HEIGHT = 20;

    private ClickableBox[] boxes = new ClickableBox[6];

    private Color[] boxColors = { Color.blue, Color.red, Color.green, Color.cyan,
         Color.magenta, Color.yellow };

    public void init() {
        for (int i = 0; i < boxes.length; i++) {
            boxes[i] = new ClickableBox(START_X, START_Y + i * BOX_HEIGHT, BOX_WIDTH,
             BOX_HEIGHT, Color.black, Color.red, true);
        }
        defaultBoxColors();
    }

    public void paint(Graphics g) {
        for(int i = 0; i < boxes.length; i++) {
            boxes[i].draw(g);
        }
    }

    public void defaultBoxColors() {
        for(int i = 0; i < boxes.length; i++) {
            boxes[i].setBackColor(boxColors[i]);
        }
    }
}
```

Let's look at the code we added.

Observe: The boxColors[] Array

```
    private Color[] boxColors = { Color.blue, Color.red, Color.green, Color.cyan,
        Color.magenta, Color.yellow };
```

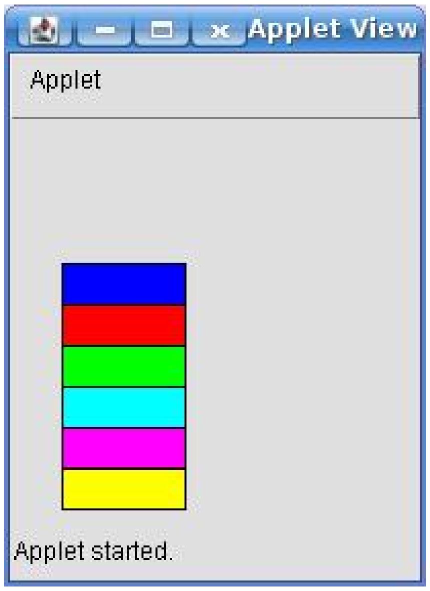
Here we're using a different way to create an array. The list of Color objects inside the **{ }** fills the array as it is made. This is called an **array initializer** or **array intialization list**. The array is created, instantiated, and set to the size of the list. Then, it is filled with the values in the list. Note that the semi-colon **;** after the last **}** is required. The result is that **boxColors[0]** holds a reference to the **Color.blue** object and **boxColors[5]** holds a reference to the **Color.yellow** object.

Observe: The defaultBoxColors() Method

```
    public void defaultBoxColors() {
        for(int i = 0; i < boxes.length; i++) {
            boxes[i].setBackColor(boxColors[i]);
        }
    }
```

This is virtually identical to the **paint()** method we made earlier, but instead of drawing each box, we set the **background color** of each box. The key is that we used the colors from the corresponding index in the boxColors array.

Save and **Run** the Applet to see your results:



So far we've learned that arrays can save us time and effort when we need several of the same kind of object. It's much easier to create an array of 1000 int values than it is to create 1000 separate int variables, right?

As promised, we're going to make the **ClickableBox** clickable, so here we go! Edit your code as shown below:

CODE TO EDIT: ADD The Code In Blue to the ClickableBox class:

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.Container;

    public class ClickableBox extends MouseAdapter {
    private int x, y, width, height;
    private Color borderColor, backColor, oldColor;
    private boolean drawBorder, clicked;
    private Container parent;

    public ClickableBox(int x, int y, int width, int height, Color borderColor,
           Color backColor, boolean drawBorder, Container parent) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.borderColor = borderColor;
        this.backColor = backColor;
        this.drawBorder = drawBorder;
        this.parent = parent;
    }

    public void draw(Graphics g) {
        oldColor = g.getColor();
        g.setColor(backColor);
        g.fillRect(x, y, width, height);
        if(drawBorder) {
            g.setColor(borderColor);
            g.drawRect(x, y, width, height);
        }
        g.setColor(oldColor);
    }

    public void mouseReleased(MouseEvent e) {
        if(x < e.getX() && e.getX() < x + width &&
             y < e.getY() && e.getY() < y + height) {
            clicked = true;
            parent.repaint();
        }
    }

    public boolean isClicked() {
        return clicked;
    }

    public void setClicked(boolean clicked) {
        this.clicked = clicked;
    }


    // Remaining Getters and Setters left out of listing for brevity.
```

Save it. Whoa! There is a lot going on here. Let's break it down.

Observe: Containers

```
private Container parent;
```

We need a way to tell our applet to **repaint()** after the mouse is released on a box. The only way to do that is to have a handle that we can grab to call **repaint()**. The Applet class and the Frame class are both descendents of the Container class, which is itself a descendent of the Component class. Component is where the **repaint()** method is defined.

API Click the API icon in the Eclipse menu. Go to the **java.awt** package. Scroll down to the **Container** class and select it. In the hierarchy, click **java.awt.Component**. Scroll down to the **repaint()** methods.

Since graphical Containers are designed to have Graphics areas and to present graphical content, we want to be able to display our class on any graphical Container. If we had just passed an **Applet** to the constructor, then only Applets would be able to use this class. Even worse would be passing a **OneDArrayApplet** to the constructor, because then only **Our** Applet would be able to use it. By using the polymorphism capability of Java, we have a class that can be used by **any** class that descends from **Container**.

Observe: The Constructor

```
public ClickableBox(int x, int y, int width, int height, Color borderColor,
   Color backColor, boolean drawBorder, Container parent) {
```

In the constructor of **ClickableBox**, we pass a reference to a **Container**. Since our applet **is** a **Container**, we will be able to pass its **this** constant to the method.

Observe: The mouseReleased() Method:

```
public void mouseReleased(MouseEvent e) {
    if(x < e.getX() && e.getX() < x + width &&
         y < e.getY() && e.getY() < y + height) {
        clicked = true;
        parent.repaint();
    }
}
```

The **mouseReleased()** method will be called any time the mouse button is released on this box--but only if this class is added to the applet's list of **MouseListeners**. This method gets the coordinates of the mouse when its button is released, through the MouseEvent methods **e.getX()** and **e.getY()**, then checks to see if those coordinates are within this **ClickableBox** instance's area. If they are, we set the **clicked** variable to **true** and tell the Applet to **repaint()** itself.

Observe: getters and setters for 'clicked' variable

```
public boolean isClicked() {
    return clicked;
}

public void setClicked(boolean clicked) {
    this.clicked = clicked;
}
```

Here we just round out the class by adding the extra Getter and Setter for the clicked variable. Now we'll **fix the Applet** so that it can use the new changes in the ClickableBox class. Edit your code as shown below:

CODE TO TYPE: ADD The Code In Blue To The OneDArrayApplet Class

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

public class OneDArrayApplet extends Applet {
    private final int START_X = 25;
    private final int START_Y = 70;
    private final int BOX_WIDTH = 60;
    private final int BOX_HEIGHT = 20;

    private ClickableBox[] boxes = new ClickableBox[6];
    private Color[] boxColors = { Color.blue, Color.red, Color.green, Color.cyan,
          Color.magenta, Color.yellow };

    public void init() {
        for (int i = 0; i < boxes.length; i++) {
            boxes[i] = new ClickableBox(START_X, START_Y + i * BOX_HEIGHT, BOX_WIDTH,
                  BOX_HEIGHT, Color.black, Color.red, true, this);
            this.addMouseListener(boxes[i]);
        }
        defaultBoxColors();
    }

    public void paint(Graphics g) {
        for(int i = 0; i < boxes.length; i++) {
            if(boxes[i].isClicked()) {
                boxes[i].setBackColor(new Color(
                  (int)(Math.random() * 256),
                  (int)(Math.random() * 256),
                  (int)(Math.random() * 256)));
                boxes[i].setClicked(false);
            }
            boxes[i].draw(g);
        }
    }

    public void defaultBoxColors() {
        for(int i = 0; i < boxes.length; i++) {
            boxes[i].setBackColor(boxColors[i]);
        }
    }
}
```

Save and **Run** your OneDArray applet. Click in different colored boxes to see what happens. Now let's look at how we did it:

Observe: The init() Method

```
    public void init() {
        for (int i = 0; i < boxes.length; i++) {
            boxes[i] = new ClickableBox(START_X, START_Y + i * BOX_HEIGHT, BOX_WIDTH,
                 BOX_HEIGHT, Color.black, Color.red, true, this);
            this.addMouseListener(boxes[i]);
        }
        defaultBoxColors();
    }
```

In the **init()** method, we have to modify the creation of the boxes so that we can pass the **this** constant to the new constructor of the **ClickableBox** class. We also need to add each box to the list of MouseListeners for this Applet. Each listener added to the list will be sent any MouseEvents that are fired on the applet. Each box is a MouseListener and will run its **mouseReleased()** method when the mouse button is released on the applet.

Observe: The paint() Method

```
    public void paint(Graphics g) {
        for(int i = 0; i < boxes.length; i++) {
        if(boxes[i].isClicked()) {
            boxes[i].setBackColor(new Color(
            (int)(Math.random() * 256),
            (int)(Math.random() * 256),
            (int)(Math.random() * 256)));
            boxes[i].setClicked(false);
        }
        boxes[i].draw(g);
    }
```

We add an *if* statement to the **paint()** method to see if the box about to be painted is clicked. If it is, we **change its color to a random color** and then set its **clicked** variable to false.

That's pretty much it for this example; if you want to have a little fun, just modify your **OneDArray** class like this, re-run it and click the Button. See if you can figure out how it works. Edit your code so it looks like this:

CODE TO TYPE: ADD The Code In Blue

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class OneDArrayApplet extends Applet {
    private final int START_X = 25;
    private final int START_Y = 70;
    private final int BOX_WIDTH = 60;
    private final int BOX_HEIGHT = 20;

    private ClickableBox[] boxes = new ClickableBox[6];
    private Color[] boxColors = { Color.blue, Color.red, Color.green, Color.cyan,
              Color.magenta, Color.yellow };

    private Button resetColors = new Button("Reset Colors");

    public void init() {
        for (int i = 0; i < boxes.length; i++) {
            boxes[i] = new ClickableBox(START_X, START_Y + i * BOX_HEIGHT, BOX_WIDTH,
                    BOX_HEIGHT, Color.black, Color.red, true, this);
            this.addMouseListener(boxes[i]);
        }
        defaultBoxColors();
        resetColors.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                defaultBoxColors();
                repaint();
            }
         });
         this.add(resetColors);
    }

    public void paint(Graphics g) {
        for(int i = 0; i < boxes.length; i++) {
            if(boxes[i].isClicked()) {
                boxes[i].setBackColor(new Color(
                    (int)(Math.random() * 256),
                    (int)(Math.random() * 256),
                    (int)(Math.random() * 256)));
                boxes[i].setClicked(false);
            }
            boxes[i].draw(g);
        }
    }

    public void defaultBoxColors() {
        for(int i = 0; i < boxes.length; i++) {
            boxes[i].setBackColor(boxColors[i]);
        }
    }
}
```

Save and **Run** it. Click in a few of the colored boxes, and then click the Reset button.

We could have put all of our classes into a single .java file. While this might be tempting, you must resist. When a single .java file for Java has more than one class, only **one** can be **public**, and that class must be the applet or application class if they exist. It also must be the class that has the same name as the .java file. The compiler will make separate .class files from a single .java file if it contains more than one class. The only time it makes any real sense to put multiple classes in a single file is if they are helper classes that are helping the main class do something. We are using multiple files, because we want our code to be reusable.

What are Arrays again?

An array is an object (sort of).

API Look in the API; you'll see that there is an **interface** named **Array** in the **java.sql** package, as well as a class named **Array** in the **java.lang.reflect** package. But when we made the arrays, we did not use a classname **Array**--rather, we used the square brackets []. (There is also a class called **ArrayList** in the **java.util** package that we will discuss in a future course.)

The array type that we are using here (and in most programming languages) is not a **class**, but rather a language-provided data structure to hold a specified number of a certain type of objects. It does not have a page in the API, and does not have its own set of methods. For practical use, you might consider arrays to be a separate kind of reference type from objects; think of them as memory addresses for collections of things.

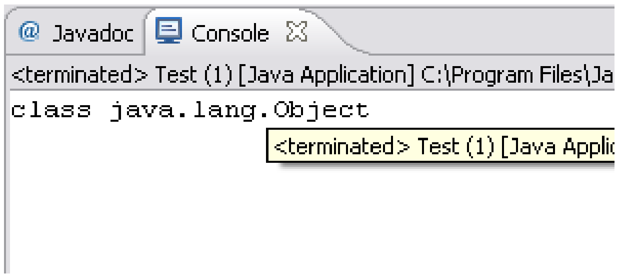
Arrays may be assigned to variables of type `Object` and all methods of the Object class may be invoked for arrays. See: the Java Language Specification.

In the **temp** project folder, edit your **Test.java** class as shown with the code in **blue**:

CODE TO TYPE: Test.java

```
class Test {
 public static void main(String[] args) {
  int[] testArray = new int[4];
  System.out.println(testArray.getClass().getSuperclass());
 }
}
```

Save and **Run** it and look at the output in the Console window:



```
<terminated> Test (1) [Java Application] C:\Program Files\Ja
class java.lang.Object
```

So, the **super** Class of a variable defined as an array of `int`s is **Object**. As we have seen, arrays have a built-in `length` property that's useful in loops and if statements.

Creating Arrays (revisited)

Thanks to Java shortcuts, there are a couple of ways to create arrays. We've already seen the first method (using **new**), in **OneDArrayApplet**:

Observe: Creating An Array

```
ClickableBox [] boxes = new ClickableBox[6];
```

This line is doing two things:

1. **ClickableBox[ ] boxes** is **declaring** that **boxes** will be the name of an array holding **ClickableBox** objects.
2. **boxes = new ClickableBox[6];** is **creating** the array and leaving space in memory for 6 **ClickableBox** object references.

This example of creating an array is similar to creating **instances** of **objects** through the use of **new**. However, after the **new** keyword, we do not see a **constructor** for a class, but rather a specification of the type of things being put into the array (in this example, **ClickableBox** object references). We could have made an array of ints or doubles just as easily.

Arrays can hold **primitive data types** or **object references**. Let's try it. Edit the Test.java class in the temp folder as follows:

CODE TO EDIT: Test.java

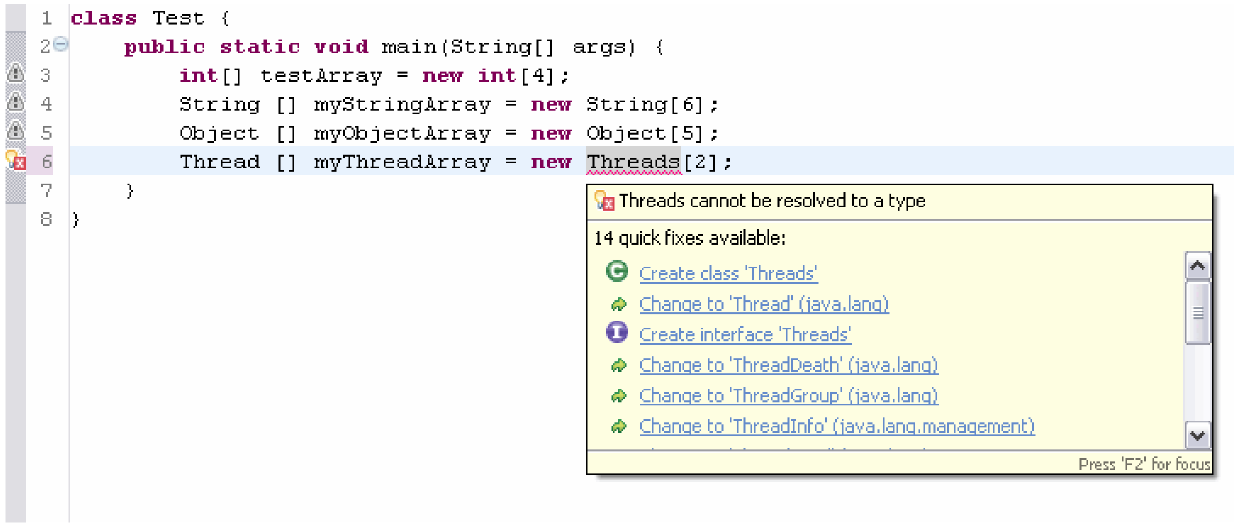
```
class Test {
 public static void main(String[] args) {
     int[] testArray = new int[4];
  String [] myStringArray = new String[6];
  Object [] myObjectArray = new Object[5];
  Thread [] myThreadArray = new Thread[2];
 }
}
```

See how the declaration of the type of array matches on the left and right side?

Change **Thread [ ] myThreadArray = new Thread[2];** to **Thread [ ] myThreadArray = new Threads[2];** as shown:

CODE TO EDIT: Test.java

```
class Test {
    public static void main(String[] args) {
        int[] testArray = new int[4];
        String [] myStringArray = new String[6];
        Object [] myObjectArray = new Object[5];
        Thread [] myThreadArray = new Threads[2];
    }
}
```



```
class Test {
    public static void main(String[] args) {
        int[] testArray = new int[4];
        String [] myStringArray = new String[6];
        Object [] myObjectArray = new Object[5];
        Thread [] myThreadArray = new Threads[2];
    }
}
```

Threads cannot be resolved to a type

14 quick fixes available:

- Create class 'Threads'
- Change to 'Thread' (java.lang)
- Create interface 'Threads'
- Change to 'ThreadDeath' (java.lang)
- Change to 'ThreadGroup' (java.lang)
- Change to 'ThreadInfo' (java.lang.management)

Press 'F2' for focus

The compiler doesn't like it! Change it back.

These lines of code are creating **empty** arrays. That is, the lines only declare that I have an array of `int`s, `String`s, `Object`s, or `Thread`s. The lines do not put element values into each location. **Except** for **primitive data types**, Java does put default values in each place. Let's check. Edit the Test.java class in the temp folder as shown below:

CODE TO EDIT: Test.java

```
class Test {
 public static void main(String[] args) {
     int[] myIntArray = new int[4];
   double [] myDoubleArray = new double[2];
   char [] myCharArray = new char[2];
   boolean [] myBooleanArray = new boolean[2];
   String [] myStringArray = new String[6];
   Object [] myObjectArray = new Object[5];
   Thread [] myThreadArray = new Thread[2];
   char showMe = '\u0000';  // this is here so you can see what the null characte
r looks like
                         // I have created the arrays, what does the first eleme
nt in each look like?
   System.out.println("The value of myIntArray[0] is " + myIntArray[0]);
   System.out.println("The value of myDoubleArray[0] is " + myDoubleArray[0]);
   System.out.println("The value of myCharArray[0] is " + myCharArray[0]);
   System.out.println("The value of myBooleanArray[0] is " + myBooleanArray[0]);
   System.out.println("The value of myStringArray[0] is " + myStringArray[0]);
   System.out.println("The value of myObjectArray[0] is " + myObjectArray[0]);
   System.out.println("The value of myThreadArray[0] is " + myThreadArray[0]);
   System.out.println("The value of showMe is " + showMe);  // to compare with th
e char array element
    }
}
```

Save and **Run** it.

The **showMe** line was put there so you could see an example of a **null** character (the default for **char**). Be aware that the default for any element in an array that holds objects rather than primitive data types is null.

For **Objects**, you used the `new` command to create an array, but you have not put anything into the array.

Shortcut for Creating Arrays

Another way to create an array is to declare it and simultaneously fill it with its initial values:

Creating and Filling an Array Simultaneously

```
String names [] = {"Joe","Sue","Molly","Maggie","Taj"};
```

The above is equivalent to:

Creating an Array the Long Way

```
String names [];
names = new String [5];
names[0] = new String ("Joe");
names[1] = new String ("Sue");
names[2] = new String ("Molly");
names[3] = new String ("Maggie");
names[4] = new String ("Taj");
```

Edit the Test.java class as follows:

CODE TO EDIT: Test.java

```
class Test {
    public static void main(String[] args) {
        String [] names = {"Joe","Sue","Molly","Maggie","Taj"};
        for (int i=0; i < names.length; i++)
            System.out.println("Name at index " + i + " is " + names[i]);
    }
}
```

Save and **Run** it.

This works for Objects as well. That is, you can list the Objects in **{}** to initialize the array of some type of object--as long as the Objects have been instantiated.

Two-Dimensional Arrays

Lesson Objectives

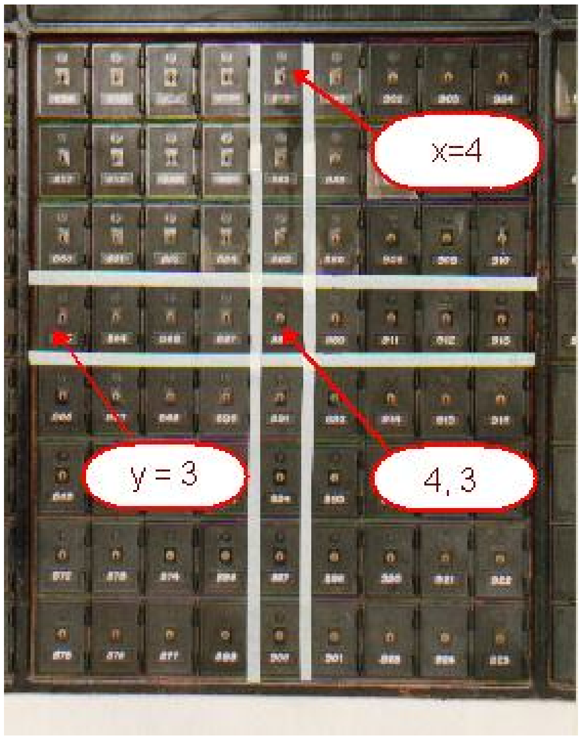
When you complete this course, you will be able to:

- translate the logic of the pseudo-code into working java code.
- use and evaluate two dimensional arrays.
- incorporate logic, flow control, two dimensional arrays, and loops.

More About Arrays

Working With Two-Dimensional Arrays

A two-dimensional array is like a grid of rows and columns, such as these Post Office boxes:



It is represented in Java as an array of arrays (that is, each entry of the array is an array!). The following snippet of code would create an array with **3 rows and 4 columns:**

OBSERVE:

```
int table[][] = new int[3][4];
```

Each cell in this array has a unique "row, column" address:

	Column 0	**Column 1**	**Column 2**	**Column 3**

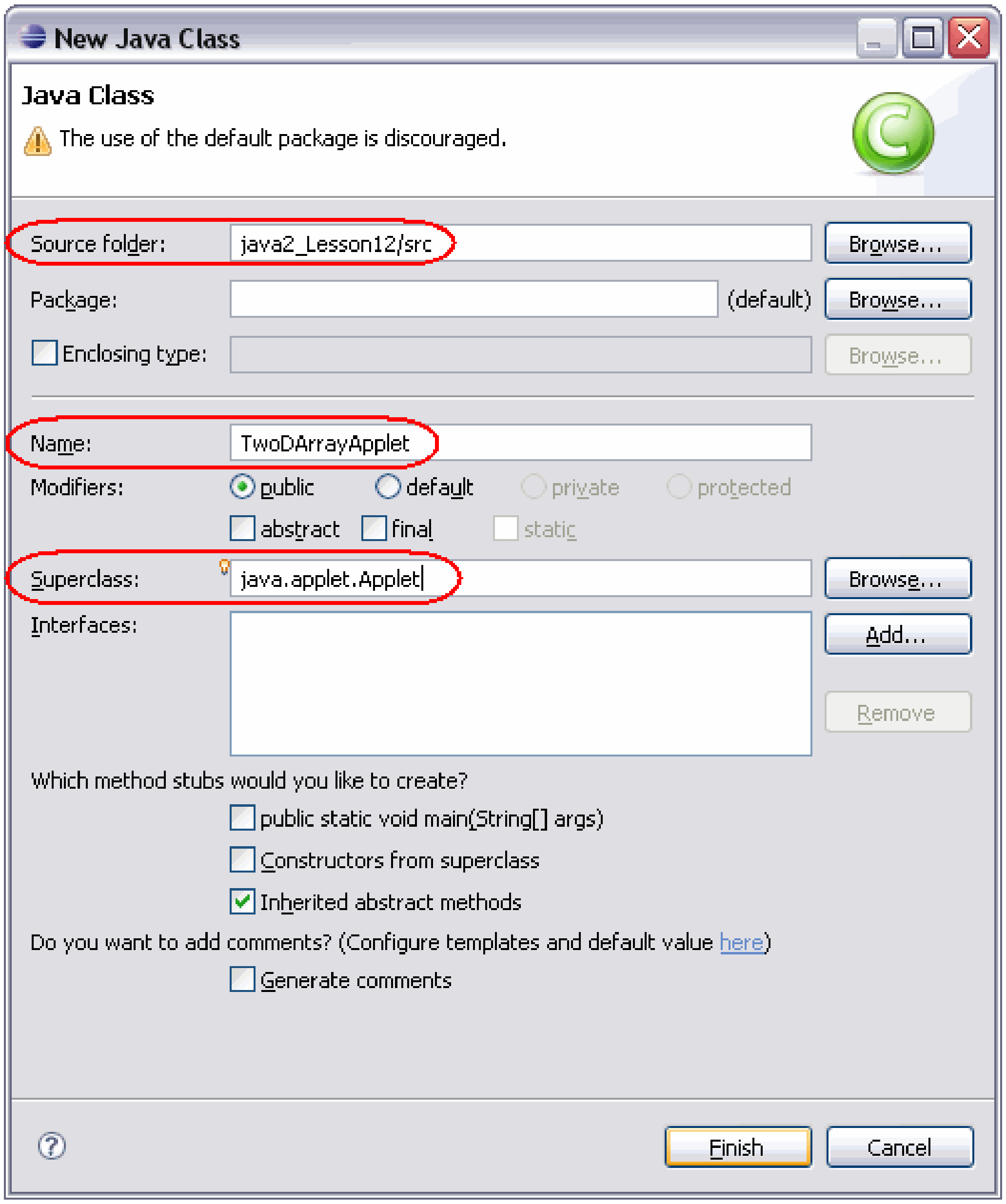
Row 0	0, 0	0, 1	0, 2	0, 3
Row 1	1, 0	1, 1	1, 2	1, 3
Row 2	2, 0	2, 1	2, 2	2, 3

Note In this lesson, we are preparing to make a "matching game," where players will take turns exposing two hidden boxes and try to match them from the memory of earlier turns.

Let's make a new class similar to the **OneDArrayApplet** from the last lesson. In fact, we'll **reuse** the **ClickableBox** class from that lesson, only this time we're going to use a two-dimensional array.

In your **Java2_Lessons** working set, make a new project named **java2_Lesson12**. Open your **java2_Lesson11** project and navigate to your **ClickableBox** class. Copy the **ClickableBox** class by right-clicking on the class and selecting **Copy**. Then, right-click on your **java2_Lesson12/src/** folder and select **Paste**.

Now, all we need to do is create a new class in your **java2_Lesson12** project. Right-click on the project and select **New | Class**. Name the new class **TwoDArrayApplet** and have it extend **java.applet.Applet**, as shown below:



Java Class

The use of the default package is discouraged.

Source folder: java2_Lesson12/src

Package: (default)

[ ] Enclosing type:

Name: TwoDArrayApplet

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.applet.Applet

Interfaces:

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

The new class will open in the editor. Make an Applet to display a 4x4 grid of **ClickableBox** objects. But this time, hide the color of the box when it's clicked and uncover it when it's clicked again. Start with some variables we know we're going to need.

The code we're going to use is similar to that in **OneDArrayApplet**, so we could copy that code and make the changes below to save time:

CODE TO TYPE: ADD the Code in Blue to the TwoDArrayApplet

```
import java.applet.Applet;

public class TwoDArrayApplet extends Applet {
    private final int START_X = 20;
    private final int START_Y = 40;
    private final int ROWS = 4;
    private final int COLS = 4;
    private final int BOX_WIDTH = 20;
    private final int BOX_HEIGHT = 20;
}
```

You may recognize some of this code from the previous lesson.

Now, we'll create the array variable:

CODE TO EDIT: ADD the Code in Blue to the TwoDArrayApplet

```
import java.applet.Applet;

public class TwoDArrayApplet extends Applet {
    private final int START_X = 20;
    private final int START_Y = 40;
    private final int ROWS = 4;
    private final int COLS = 4;
    private final int BOX_WIDTH = 20;
    private final int BOX_HEIGHT = 20;

    private ClickableBox boxes[][];
}
```

Observe: The Placement of the [ ][ ]

```
    private ClickableBox boxes[][];
```

Note In Java, *it doesn't matter* if you put the array declarators **[ ][ ]** after the type (**ClickableBox**) or after the variable name (**boxes**) *when declaring* the variable. *It does matter*, though, where you place them *when accessing* the variable. In that case, they must be placed after the variable name.

The double set of **[ ]** indicates that we are declaring a two-dimensional array. The first brackets denote the number of rows in the array and the second denotes the number of columns. Java allows **ragged** arrays, but only if we create the array using an initializer list. A **ragged array** is an array where rows have a varying number of columns. This information will be important when we discuss how to access the number of columns in each row.

We did not actually create the array instance in the example box above, but rather declared that we are going to use a variable named **boxes**, and that will be a two-dimensional array of **ClickableBox** objects. It's common in Java programs to **declare** a variable and then **define** it in the constructor, or in the case of an Applet, in the **init()** method. We **could** have declared it and defined it above by saying:

private ClickableBox boxes[ ][ ] = new ClickableBox[ROWS][COLS];

or

private ClickableBox[ ][ ] boxes = new ClickableBox[ROWS][COLS];

Okay, let's add some functionality to our Applet:

CODE TO EDIT: ADD the Code in Blue to the TwoDArrayApplet

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

public class TwoDArrayApplet extends Applet {
    private final int START_X = 20;
    private final int START_Y = 40;
    private final int ROWS = 4;
    private final int COLS = 4;
    private final int BOX_WIDTH = 20;
    private final int BOX_HEIGHT = 20;

    private ClickableBox boxes[][];
    private Color boxColors[][];

    public void init() {
        boxes = new ClickableBox[ROWS][COLS];
        boxColors = new Color[ROWS][COLS];
        randomizeColors();
        buildBoxes();
    }

    private void buildBoxes(){
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                boxes[row][col] =
                  new ClickableBox(START_X + col * BOX_WIDTH,
                                   START_Y + row * BOX_HEIGHT,
                                   BOX_WIDTH,
                                   BOX_HEIGHT,
                                   Color.gray,
                                   boxColors[row][col],
                                   true,
                                   this);
            }
        }
    }

    private void randomizeColors() {
        int[] chosenColors = {0, 0, 0, 0, 0, 0, 0, 0 };
        Color[] availableColors = { Color.red, Color.blue, Color.green,
            Color.yellow, Color.cyan, Color.magenta, Color.pink, Color.orange };
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                for(;;) {
                    int rnd = (int)(Math.random() * 8);
                    if(chosenColors[rnd] < 2) {
                        chosenColors[rnd]++;
                        boxColors[row][col] = availableColors[rnd];
                        break;
                    }
                }
            }
        }
    }
}
```

We added a boxColors array so that we can separate the colors from the boxes. This allows us to compact our code and use loops to create our arrays, and also allows us to re-randomize the colors later with minimal modifications to our code.

Observe: The init() Method:

```
    public void init() {
        boxes = new ClickableBox[ROWS][COLS];
        boxColors = new Color[ROWS][COLS];
        randomizeColors();
        buildBoxes()();
    }
```

In the **init()** method, we use **ROWS** to define our arrays and **COLS** to define the size of the arrays. Then we call the private helper methods **randomizeColors()** and **buildBoxes()** to set up our arrays.

Observe: The randomizeColors() Method:

```
    private void randomizeColors() {
        int[] chosenColors = {0, 0, 0, 0, 0, 0, 0, 0 };
        Color[] availableColors = { Color.red, Color.blue, Color.green,
            Color.yellow, Color.cyan, Color.magenta, Color.pink, Color.orange };
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                for(;;) {
                    int rnd = (int)(Math.random() * 8);
                    if(chosenColors[rnd] < 2) {
                        chosenColors[rnd]++;
                        boxColors[row][col] = availableColors[rnd];
                        break;
                    }
                }
            }
        }
    }
```

There is a lot going on in this method. First, we set up an array to keep track of the colors we have already used. We also set up an array of 8 colors. In the game we will create in the project, the object is to match two colors. Since we have 16 boxes, we need 8 colors, with each color used exactly twice. The outermost **for** loop iterates through each row; the next **for** loop will loop through each column, within each of the first loop's rows. The innermost **for(;;)** loop is an **infinite** loop. The only required parts of a **for** loop within its parentheses are the semicolons. A **for** loop with only two semicolons inside the parentheses is **infinite**.

Inside the infinite **for**, we create a random number, 0 through 7. We check to see if the **chosenColors** array at that index has been used twice. If it has not, we increment its value to indicate that it has been used. Then we set the **boxColors[row][col]** to the **availableColors[rnd]** value. Then, we must **break** out of the innermost loop. Without the **break** statement, this innermost loop would never end. Now we have code that will randomly fill our **boxColors** array with 8 colors.

Note We're using a nested **for** loop to loop through the array because there are two dimensions in the array. The **outside for loop** loops once after the **inside for loop** has completed **all** of its iterations.

Observe: Two Dimensional Array length Constant:

```
    for(int row = 0; row < boxes.length; row++) {
        for(int col = 0; col < boxes[row].length; col++) {
```

In the outer 'for' loop, we access **boxes.length** in the same way we would access the length of a one-dimensional array. This gives us the number of rows in the array.

The inner loop, however, it needs to know how many columns are in each row. So we'll modify how we access the **length** constant. Since Java stores multi-dimensional arrays as an array of arrays, each row actually represents an array. We can access the **length** constant of each row by using **boxes[row].length**, which says we want to look at the array stored at **boxes[row]** and get its **length** constant. Accessing the row and column of a two-dimensional array in this manner allows us to parse even ragged arrays. One way to read the above nested for loop: **for each row in the array boxes, loop through each column of that row**.

Observe: The buildBoxes() Method:

```
        private void buildBoxes(){
            for(int row = 0; row < boxes.length; row++) {
                for(int col = 0; col < boxes[row].length; col++) {
                    boxes[row][col] =
                        new ClickableBox(START_X + col * BOX_WIDTH,
                                         START_Y + row * BOX_HEIGHT,
                                         BOX_WIDTH,
                                         BOX_HEIGHT,
                                         Color.gray,
                                         boxColors[row][col],
                                         true,
                                         this);
                }
            }
        }
```

In the **buildBoxes()** method, we need to access the **row** and **column** of the array in order to find the element that needs a value assigned to it. In this case, we are assigning all of the column values before moving on to the next row, because the inner loop executes **all** of its iterations before allowing the outer loop to move to its next iteration. We create the columns of each row by assigning a new **ClickableBox** for each **[row][col]** of the array.

For the **x** value of the boxes, we take the **START_X**, and add to it the **col** times the **BOX_WIDTH** to place the boxes next to each other on the row automatically. For the **y** location of the boxes, we take the **START_Y**, add the **row**, and multiply the sum by the **BOX_HEIGHT** to place the boxes in rows.

CODE TO EDIT: Add the Code in Blue to the TwoDArrayApplet

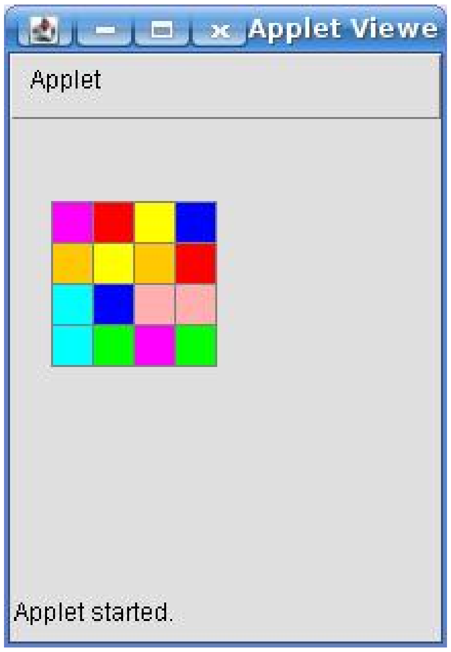
```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;

public class TwoDArrayApplet extends Applet {
    private final int START_X = 20;
    private final int START_Y = 40;
    private final int ROWS = 4;
    private final int COLS = 4;
    private final int BOX_WIDTH = 20;
    private final int BOX_HEIGHT = 20;

    private ClickableBox boxes[][];
    private Color boxColors[][];

    public void init() {
        boxes = new ClickableBox[ROWS][COLS];
        boxColors = new Color[ROWS][COLS];
        //separate building colors so we can add a button later
        //to re-randomize them.
        randomizeColors();
        buildBoxes();
    }

    public void paint(Graphics g) {
        for(int row = 0; row < boxes.length; row ++) {
            for(int col = 0; col < boxes[row].length; col++) {
                boxes[row][col].draw(g);
            }
        }
    }

    private void buildBoxes() {
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                boxes[row][col] =
                    new ClickableBox(START_X + col * BOX_WIDTH,
                                     START_Y + row * BOX_HEIGHT,
                                     BOX_WIDTH,
                                     BOX_HEIGHT,
                                     Color.gray,
                                     boxColors[row][col],
                                     true,
                                     this);
            }
        }
    }

    private void randomizeColors() {
        int[] chosenColors = {0, 0, 0, 0, 0, 0, 0, 0 };
        Color[] availableColors = { Color.red, Color.blue, Color.green,
             Color.yellow, Color.cyan, Color.magenta, Color.pink, Color.orange }
;
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                for(;;) {
                    int rnd = (int)(Math.random() * 8);
                    if(chosenColors[rnd] < 2) {
                        chosenColors[rnd]++;
                        boxColors[row][col] = availableColors[rnd];
                        break;
                    }
                }
            }
        }
    }
}
```

Observe: Two Dimensional Array Assignment

```
        public void paint(Graphics g) {
            for(int row = 0; row < boxes.length; row ++) {
                for(int col = 0; col < boxes[row].length; col++) {
                    boxes[row][col].draw(g);
                }
            }
        }
```

Here, we're looping through the array and telling each box to draw itself on the **Graphics** object **g**, that we received in the **paint()** method's parameters.

Save and **Run** it. It should look something like the image below, but remember, the colors are random:



Now, let's add a Button we can click to re-randomize the colors:

CODE TO EDIT: Add The Code In Blue To The TwoDArrayApplet

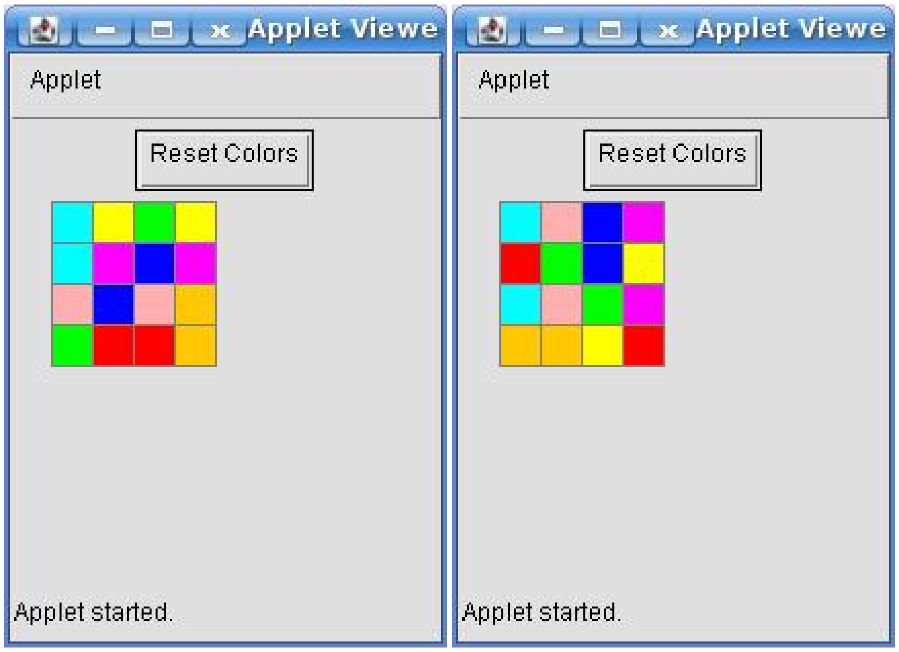
```
import java.applet.Applet;
import java.awt.Button;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TwoDArrayApplet extends Applet {
    private final int START_X = 20;
    private final int START_Y = 40;
    private final int ROWS = 4;
    private final int COLS = 4;
    private final int BOX_WIDTH = 20;
    private final int BOX_HEIGHT = 20;

    private ClickableBox boxes[][];
    private Color boxColors[][];

    private Button resetButton;

    public void init() {
        boxes = new ClickableBox[ROWS][COLS];
        boxColors = new Color[ROWS][COLS];
        resetButton = new Button("Reset Colors");
        resetButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                randomizeColors();
                buildBoxes();
                repaint();
            }
        });
        add(resetButton);
        //separate building colors so we can add a button later
        //to re-randomize them.
        randomizeColors();
        buildBoxes();
    }

    public void paint(Graphics g) {
        for(int row = 0; row < boxes.length; row ++) {
            for(int col = 0; col < boxes[row].length; col++) {
                boxes[row][col].draw(g);
            }
        }
    }

    private void buildBoxes() {
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                boxes[row][col] =
                    new ClickableBox(START_X + col * BOX_WIDTH,
                                     START_Y + row * BOX_HEIGHT,
                                     BOX_WIDTH,
                                     BOX_HEIGHT,
                                     Color.gray,
                                     boxColors[row][col],
                                     true,
                                     this);
            }
        }
    }

    private void randomizeColors() {
        int[] chosenColors = {0, 0, 0, 0, 0, 0, 0, 0 };
        Color[] availableColors = { Color.red, Color.blue, Color.green,
            Color.yellow, Color.cyan, Color.magenta, Color.pink, Color.orange };
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                for(;;) {
                    int rnd = (int)(Math.random() * 8);
                    if(chosenColors[rnd] < 2) {
                        chosenColors[rnd]++;
                        boxColors[row][col] = availableColors[rnd];
                        break;
                    }
                }
            }
        }
    }
}
```

Observe: resetButton Anonymous Inner Class

```
    resetButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            randomizeColors();
            buildBoxes();
            repaint();
        }
    });
```

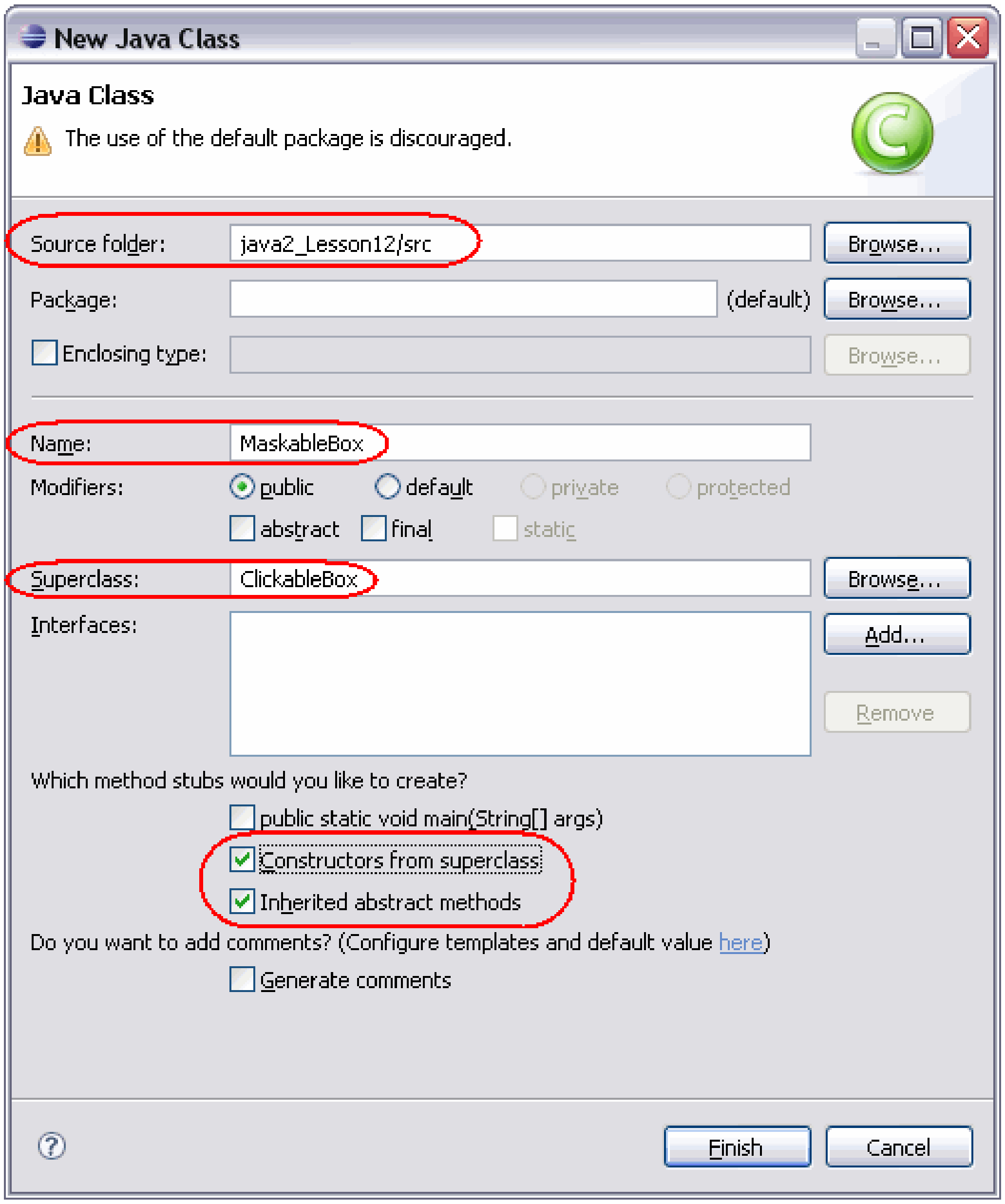
We saw this in an earlier lesson. It's an **anonymous inner class** that handles the event when the button is pressed. In the **actionPerformed()** method, we randomize the colors, build the boxes, and then repaint the applet. We need to build the boxes again so that they pick up the new colors.

Save and **Run** the applet now. You should be able to click the **Reset Colors** button and see results similar to the images below. The first image is from before the button was pressed and the second image is after it was pressed. Again, remember that the colors are random.



Since we're **making a match game**, we're going to want to **hide or mask the colors from the user**. Then we'll be able to expose specific boxes on each player's turn. We want to be able to mask the color of a box; unfortunately, the **ClickableBox** doesn't have that functionality. We can either add that functionality or extend it with another class and give **that** class the functionality. Good design dictates that we not muck around with a class we might already be using in another situation, so let's build another class.

Create a new class named **MaskableBox** in your project, as shown:


New Java Class

Java Class

The use of the default package is discouraged.

Source folder: java2_Lesson12/src

Package: (default)

[ ] Enclosing type:

Name: MaskableBox

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: ClickableBox

Interfaces:

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

CODE TO TYPE: MaskableBox

```
import java.awt.Color;
import java.awt.Container;
import java.awt.Graphics;

public class MaskableBox extends ClickableBox {
    private boolean mask;
    private Color maskColor;
    Container parent;

    public MaskableBox(int x, int y, int width, int height, Color borderColor,
        Color backColor, boolean drawBorder, Container parent) {
        super(x, y, width, height, borderColor, backColor, drawBorder, parent);
        this.parent = parent;
    }

    public void draw(Graphics g) {
        super.draw(g);
        if(mask) {
            setOldColor(g.getColor());
            g.setColor(maskColor);
            g.fillRect(getX(), getY(), getWidth(), getHeight());
            if(isDrawBorder()) {
                g.setColor(getBorderColor());
                g.drawRect(getX(), getY(), getWidth(), getHeight());
            }
            g.setColor(getOldColor());
        }
    }

    public boolean isMask() {
        return mask;
    }

    public void setMask(boolean mask) {
        this.mask = mask;
    }

    public Color getMaskColor() {
        return maskColor;
    }

    public void setMaskColor(Color maskColor) {
        this.maskColor = maskColor;
    }
}
```

Save the new class.

Observe: variables in MaskableBox

```
    private boolean mask;
    private Color maskColor;
    Container parent;
```

We need this class to keep track of when to mask the box, and which color to use to mask it. Since we failed to place a getter and setter for the **parent** variable in **ClickableBox**, we need an instance variable here to keep track of it. If you want, you can modify your **ClickableBox** class to add the getter (no need for a setter) for this variable and then access it later using the getter.

Observe: draw() method in MaskableBox

```
    public void draw(Graphics g) {
        super.draw(g);
        if(mask) {
            setOldColor(g.getColor());
            g.setColor(maskColor);
            g.fillRect(getX(), getY(), getWidth(), getHeight());
            if(isDrawBorder()) {
                g.setColor(getBorderColor());
                g.drawRect(getX(), getY(), getWidth(), getHeight());
            }
            g.setColor(getOldColor());
        }
    }
```

The **draw()** method first calls **super.draw(g)**. If you call **super** class in a method, it must be the first thing in the method body. We allow the **ClickableBox** to draw itself. If the **mask** variable is set true, we then draw over the box. This keeps us from having to keep track of the original color of the box, because we draw over it with the **maskColor** only if the mask is set. Notice how we had to use the getters from the **ClickableBox** class? They are **private** in the **ClickableBox** class, so this is the only way to access them.

We're almost there! Now we need to modify our TwoDArrayApplet class. Change all references to ClickableBox and make them references to MaskableBox.

Note We omitted some of the code to make the listing smaller; don't remove it from your class!

CODE TO EDIT: In TwoDArrayApplet, Change All ClickableBox References to MaskableBox (IN BLUE)

```
//import statements removed for brevity.
public class TwoDArrayApplet extends Applet {
    //variables removed for brevity.
    private MaskableBox boxes[][];

    public void init() {
        boxes = new MaskableBox[ROWS][COLS];
        //code removed for brevity.
    }

    private void buildBoxes() {
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                boxes[row][col] =
                  new MaskableBox(START_X + col * BOX_WIDTH,
                                            START_Y + row * BOX_HEIGHT,
                                            BOX_WIDTH,
                                            BOX_HEIGHT,
                                            Color.gray,
                                            boxColors[row][col],
                                            true,
                                            this);
            }
        }
    }
    //randomizeColors() method removed for brevity.
}
```

Now modify the TwoDArrayApplet to give it the functionality. There's quite a bit going on here, so we will break it down after you enter the code:

CODE TO EDIT: Add The Code in Blue to the TwoDArrayApplet

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TwoDArrayApplet extends Applet {
    private final int START_X = 20;
    private final int START_Y = 40;
    private final int ROWS = 4;
    private final int COLS = 4;
    private final int BOX_WIDTH = 20;
    private final int BOX_HEIGHT = 20;

    private MaskableBox boxes[][];
    private Color boxColors[][];

    private Button resetButton;

    public void init() {
        boxes = new MaskableBox[ROWS][COLS];
        boxColors = new Color[ROWS][COLS];
        resetButton = new Button("Reset Colors");
        resetButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                randomizeColors();
                buildBoxes();
                repaint();
            }
        });
        add(resetButton);
        //separate building colors so we can add a button later
        //to re-randomize them.
        randomizeColors();
        buildBoxes();
    }

    public void paint(Graphics g) {
        for(int row = 0; row < boxes.length; row ++) {
            for(int col = 0; col < boxes[row].length; col++) {
                if(boxes[row][col].isClicked()) {
                    boxes[row][col].setMaskColor(Color.black);
                    boxes[row][col].setMask(!boxes[row][col].isMask());
                    boxes[row][col].setClicked(false);
                }
                boxes[row][col].draw(g);
            }
        }
    }

    private void removeMouseListeners() {
        for(int row = 0; row < boxes.length; row ++) {
            for(int col = 0; col < boxes[row].length; col++) {
                removeMouseListener(boxes[row][col]);
            }
        }
    }

    private void buildBoxes() {
        removeMouseListeners();
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                boxes[row][col] =
                  new MaskableBox(START_X + col * BOX_WIDTH,
                                  START_Y + row * BOX_HEIGHT,
                                    BOX_WIDTH,
                                    BOX_HEIGHT,
                                    Color.gray,
                                    boxColors[row][col],
                                    true,
                                    this);
                addMouseListener(boxes[row][col]);
            }
        }
    }

    private void randomizeColors() {
        int[] chosenColors = {0, 0, 0, 0, 0, 0, 0, 0 };
        Color[] availableColors = { Color.red, Color.blue, Color.green,
          Color.yellow, Color.cyan, Color.magenta, Color.pink, Color.orange };
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                for(;;) {
                    int rnd = (int)(Math.random() * 8);
                    if(chosenColors[rnd] < 2) {
                        chosenColors[rnd]++;
                        boxColors[row][col] = availableColors[rnd];
                       break;
                    }
                }
            }
        }
    }
}
```

Observe: The paint() Method

```
    public void paint(Graphics g) {
        for(int row = 0; row < boxes.length; row ++) {
            for(int col = 0; col < boxes[row].length; col++) {
                if(boxes[row][col].isClicked()) {
                    boxes[row][col].setMaskColor(Color.black);
                    boxes[row][col].setMask(!boxes[row][col].isMask());
                    boxes[row][col].setClicked(false);
                }
                boxes[row][col].draw(g);
            }
        }
    }
```

In our **paint()** method, we added code similar to our **OneDArrayApplet** from the previous lesson, to determine **if the box we are looking at has been clicked**. If it has been clicked, we **set its maskColor to black**. Next, we perform a little trick with booleans--we set its **mask** variable to the opposite of what it was before. **!boxes[row][col].isMask()** simply gets the value of the box's **mask** variable and takes its **not** value. So, if **mask** is **true**, we get false as an answer. Then we **set the box's clicked variable to false**, and **have the box draw itself**.

Observe: The removeMouseListeners() Method

```
    private void removeMouseListeners() {
        for(int row = 0; row < boxes.length; row ++) {
            for(int col = 0; col < boxes[row].length; col++) {
                removeMouseListener(boxes[row][col]);
            }
        }
    }
```

We need to add the boxes to the applet as **MouseListeners**. Listeners are stored in the applet as a list. When we run the **buildBoxes()** method, we need to add new boxes to this list, because when **buildBoxes()** creates new instances of the **MaskableBox** class, handles to the old versions of those objects are lost. The **Applet** class provides a method (from one of its ancestors) to remove a **MouseListener** from the list. Our method simply loops through the array of boxes and calls the **removeMouseListener()** method, removing each one from the list.

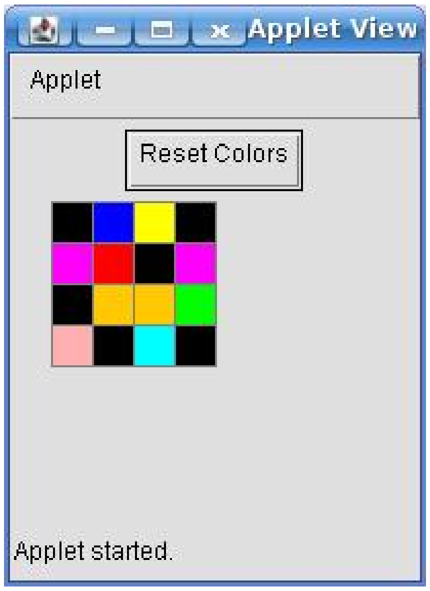
Observe: The buildBoxes() Method

```
    private void buildBoxes() {
        removeMouseListeners();
        for(int row = 0; row < boxes.length; row++) {
            for(int col = 0; col < boxes[row].length; col++) {
                boxes[row][col] =
                  new MaskableBox(START_X + col * BOX_WIDTH,
                                  START_Y + row * BOX_HEIGHT,
                                  BOX_WIDTH,
                                  BOX_HEIGHT,
                                  Color.gray,
                                  boxColors[row][col],
                                  true,
                                  this);
                addMouseListener(boxes[row][col]);
            }
        }
    }
```

In the **buildBoxes()** method, we remove all of the MouseListeners we might have added before. The **removeMouseListener()** method provided by **Applet** throws no exceptions and does nothing if the **MouseListener** is **null**. It's a safe method to call.

After we **create a box** inside the **inner for loop**, we **add it to the applet as a MouseListener**.

Save and **Run** it. Click a few boxes. It should look something like the image below, but remember, the colors are random:



Ragged Arrays

You can also declare and initialize **ragged** arrays with nested initializations. Here we have an array made from three arrays:

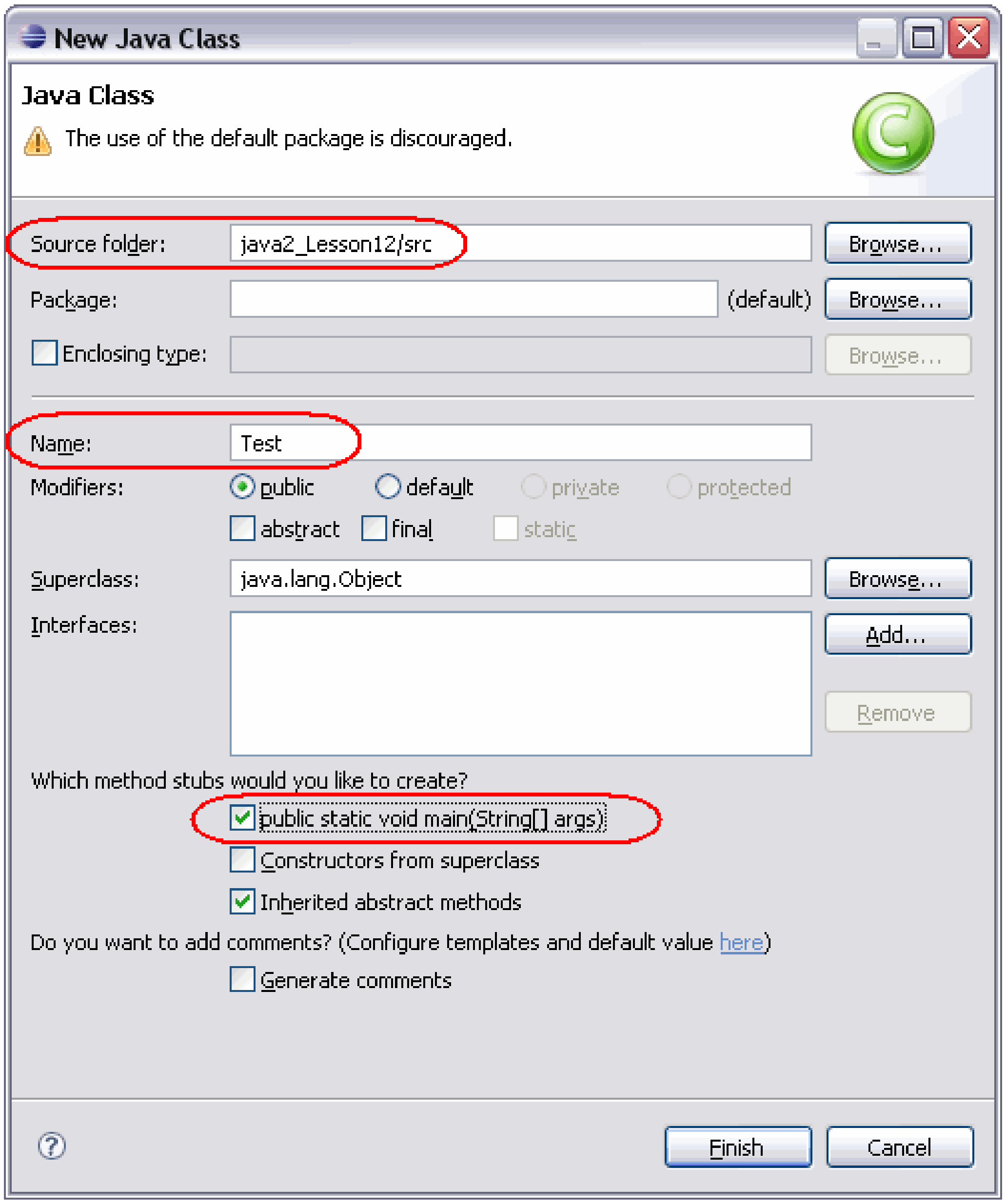
1. {2,1}
2. {5,4,3}
3. {9,8,7,6}

Observe: twoDimArray Declaration

```
    int[][] twoDimArray = { {2,1},
                            {5,4,3},
                            {9,8,7,6} };
```

The initialization list has changed in this declaration. We have the **outer set of { } braces**, which represents the overall array. The comma-delimited lists of array initializers in the inner sets of **{ }** braces build each row of the array using the comma-delimited numbers inside of them, to form the columns of each row. **row 0** of the array will have a length of 2 and will contain 2 and 1 as its elements. **Row 1** will have a length of 3 and will contain 5, 4, and 3 as its elements. **Row 2** will have a length of 4 and will contain 9, 8, 7, and 6 as its elements.

Rather than explain, let's work through it with an experient in Java. Create a class named **Test.java** in your project, as shown below:



New Java Class

Java Class

The use of the default package is discouraged.

Source folder: java2_Lesson12/src

Package: (default)

[ ] Enclosing type:

Name: Test

Modifiers: [● public] ○ default ○ private ○ protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

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

CODE TO TYPE: Add The Code In Blue

```
class Test {
    public static void main(String[] args) {
        int[][] twoDimArray = { {2,1},{5,4,3},{9,8,7,6} };

        System.out.println("The length of the array");
        System.out.println("  (or the number of rows) = " + twoDimArray.length);
        System.out.println("The width of the array");
        System.out.println("  (or the number of columns)");
        System.out.println("    in twodim[0].length is " + twoDimArray[0].length
);
        System.out.println("Which is not the same as");
        System.out.println("    in twodim[1].length is " + twoDimArray[1].length
);
        System.out.println("Which is not the same as");
        System.out.println("    in twodim[2].length is " + twoDimArray[2].length
);
        System.out.println("Printing out the array results in:");
        for(int row = 0; row < twoDimArray.length; row ++) {
            for(int col = 0; col < twoDimArray[row].length; col ++) {
                System.out.print(twoDimArray[row][col] + " ");
            }
            System.out.println();
        }
    }
}
```

Save and **Run** it.

Observe: Printing Out The Array

```
    for(int row = 0; row < twoDimArray.length; row ++) {
        for(int col = 0; col < twoDimArray[row].length; col ++) {
            System.out.print(twoDimArray[row][col] + " ");
        }
        System.out.println();
    }
```

Again, Java goes through all of the iterations (columns) in the **inner loop** for each iteration (row) of the **outer loop**. Java does **not** even out the arrays, therefore they hold the same number of columns. Let's test this:

CODE TO TYPE: Change the Code in Blue

```
public class Test {
    public static void main(String[] args) {
        int[][] twoDimArray = { {2,1},{5,4,3},{9,8,7,6} };

        System.out.println("The length of the array");
        System.out.println(" (or the number of rows) = " + twoDimArray.length);
        System.out.println("The width of the array");
        System.out.println(" (or the number of columns)");
        System.out.println("    in twodim[0].length is " + twoDimArray[0].length
);
        System.out.println("Which is not the same as");
        System.out.println("    in twodim[1].length is " + twoDimArray[1].length
);
        System.out.println("Which is not the same as");
        System.out.println("    in twodim[2].length is " + twoDimArray[2].length
);
        System.out.println("Printing out the array results in:");
        for(int row = 0; row < twoDimArray.length; row ++) {
            for(int col = 0; col < twoDimArray[0].length; col ++) {
                System.out.print(twoDimArray[row][col] + " ");
            }
            System.out.println();
        }
    }
}
```

Save and **Run** it.

Did you notice a problem? In the inner **for** loop, we addressed a particular row's length in twoDimArray[] instead of the general row length for the array. Try making this change:

CODE TO TYPE: Change the Code in Blue

```
public class Test {
    public static void main(String[] args) {
        int[][] twoDimArray = { {1,2},{3,4,5},{5,6,7,8} };

        System.out.println("The length of the array");
        System.out.println(" (or the number of rows) = " + twoDimArray.length);
        System.out.println("The width of the array");
        System.out.println(" (or the number of columns)");
        System.out.println("    in twoDimArray[0].length is " + twoDimArray[0].l
ength);
        System.out.println("Which is not the same as");
        System.out.println("    in twoDimArray[1].length is " + twoDimArray[1].l
ength);
        System.out.println("Which is not the same as");
        System.out.println("    in twoDimArray[2].length is " + twoDimArray[2].l
ength);
        System.out.println("Printing out the array results in:");
        for(int row = 0; row < twoDimArray.length; row ++) {
            for(int col = 0; col < twoDimArray[row].length; col ++) {
                System.out.print(twoDimArray[row][col] + " ");
            }
            System.out.println();
        }
    }
}
```

The Oracle tutorial on arrays has an interesting example of this usage as well. Check it out.

Note Implementation information: for memory storage, Java uses **row-major order** (the row is filled and stored first). For further discussion, see this Wikipedia entry.

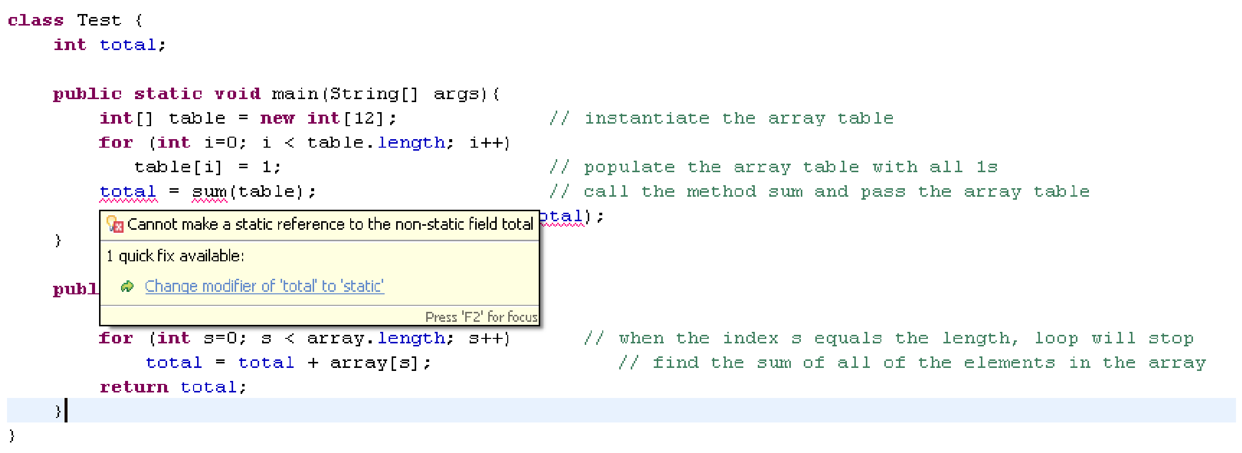
Using Arrays

We need to declare and create arrays just like other primitive data types and objects, but we also want to **pass** them to other classes via **methods**. Let's make an example that illustrates passing arrays as parameters. It will be very similar to the *accumulator* **add()** method of our **SetMyArray** class, except we will **pass** the array rather than having the array be accessible as an **instance variable**. Go ahead and edit Test.java as shown:

CODE TO TYPE: passing arrays

```
public class Test {
    int total;

    public static void main(String[] args){
        int[] table = new int[12];             // instantiate the array table
        for (int i=0; i < table.length; i++)
           table[i] = 1;                       // populate the array table with 
all 1s
        total = sum(table);                    // call the method sum and pass t
he array table
        System.out.println("The total is " + total);
    }

    public int sum(int [] array) {
        total = 0;
        for (int s=0; s < array.length; s++)      // when the index s equals the
 length, loop will stop
            total = total + array[s];                 // find the sum of all of t
he elements in the array
        return total;
    }
}
```

Check out the error. We admit, we did this to you on purpose to remind you about the use of the **main()** method .



```
class Test {
    int total;

    public static void main(String[] args){
        int[] table = new int[12];           // instantiate the array table
        for (int i=0; i < table.length; i++)
            table[i] = 1;                    // populate the array table with all 1s
        total = sum(table);                  // call the method sum and pass the array table
                                             otal);
    }

    publ
        for (int s=0; s < array.length; s++)      // when the index s equals the length, loop will stop
            total = total + array[s];                 // find the sum of all of the elements in the array
        return total;
    }
}
```

Cannot make a static reference to the non-static field total

1 quick fix available:

Change modifier of 'total' to 'static'

Press 'F2' for focus

In an application, the **class** that **main ()** is located in is not instantiated unless you instantiate it specifically.

total is an instance variable of the Test class and is not marked static, and the main method is marked static; a static method can never access a non-static variable. **total** requires the class to be instantiated with the **new** keyword in order for total to exist.

CODE TO TYPE:

```
class Test {
    int total;

    public static void main(String[] args){
        Test myTest = new Test();     // instantiate the class Test
        myTest.demo();                      // start the class
    }

    public void demo(){
        int[] table = new int[12];          // instantiate the array table
        for (int i=0; i < table.length; i++)
        table[i] = 1;                   // populate the array table with all 1s
        total = sum(table);                 // call the method sum and pass the
array table
        System.out.println("The total is " + total);
    }

    public int sum(int [] array) {
        total = 0;
        for (int s=0; s < array.length; s++)      // when the index s equals the
 length, loop will stop
            total = total + array[s];               // find the sum of all of t
he elements in the array
        return total;
    }
}
```

Save and **Run** it.

It should work now, because the **main ()** method creates an instance of the Test class and then accesses the variable via the reference to the new instance of the class (myTest).

Tip Non-static instance methods can access static or non-static stuff. Static methods can only access static stuff.

The important things to see in this new code are:

1. Somewhere an array was instantiated:
   int[] table = new int[12];
2. In the method where the array is created--**demo()**--a method call is made and the array's variable is passed as an actual parameter:
   total = sum(table);
3. A method is defined with a formal parameter of an array:
   public int sum(int [] array)

Two other important aspects:

1. The name of the actual parameter does not need to match the name of the formal parameter.
2. The array **table** did not need to be an **instance variable**, because it could be seen by the other method when passed.

This last aspect is more important than you might think. Objects (and arrays) in Java are passed by passing a reference to the object (or array); objects (and arrays) in Java are not passed by passing the values of the elements of the object (or array). Phew! This means that:

- You are actually giving the memory location of the variable `table` to the method `sum`.
- If you change anything in the array while in the method `sum`, it changes that array content everywhere.

Some sources say that objects and arrays are passed "by reference" (because you are passing the pointer to a memory location) and not "by value."

Oracle's Java Tutorial uses this terminology:

"Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level."

Note Be aware when reading about how objects are passed, that the use of the terms "by value" and "by reference" are not always consistent.

Objects are arrays that pass the memory location and so they can be inadvertently changed.

Edit the Test.java class as shown:

CODE TO TYPE

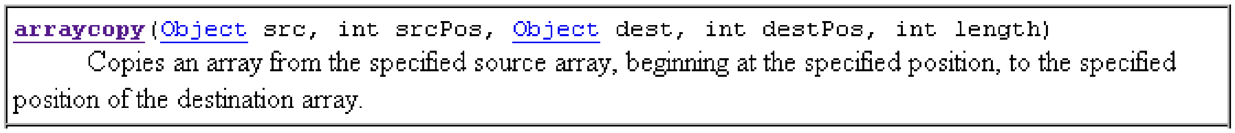
```
class Test {
    int total;

    public static void main(String[] args){
        Test myTest = new Test();     // instantiate the class Test
        myTest.demo();                      // start the class
    }

    public void demo(){
        int[] table = new int[12];          // instantiate the array table
        for (int i=0; i < table.length; i++)
        {
            table[i] = 1;                   // populate the array table with all
 1's
            System.out.print(table[i] + " ");
        }
        total = sum(table);                 // call the method sum and pass the
array table
        System.out.println("The total is " + total);
        System.out.println("After method invocation completes \n and control has
 returned, values are");
        for (int i=0; i < table.length; i++)
            System.out.print(table[i] + " ");
    }

    public int sum(int [] array) {
        total = 0;
        for (int s=0; s < array.length; s++)    // when the index s equals the l
ength, loop will stop
        {
            total = total + array[s];              // find the sum of all of the
 elements in the array
            array[s]= array[s] + 1;
        }
        return total;
    }
}
```

Save and **Run** it.

The copy method is useful when you want to pass an array without causing its elements to change.

Copying Arrays

Go to the API to the **java.lang** package. Go to the **System** class. Find the **arraycopy** method:



`arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`
Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

Try it out by editing the Test.java class as shown blow. (Add the code shown in **blue**, and delete the code shown in **red**):

CODE TO TYPE

```
class Test {
    int total;

    public static void main(String[] args){
        Test myTest = new Test();           // instantiate the class Test
        myTest.demo();                      // start the class
    }

    public void demo(){
        int[] table = new int[12];          // instantiate the array table
        for (int i=0; i < table.length; i++)
        {
            table[i] = 1;                   // populate the array table with all
 1's
            System.out.print(table[i] + " ");
        }

        int[] table2 = new int[12];
        System.arraycopy(table, 0, table2, 0, table.length);
        total = sum(table2);                // call the method sum and pass the
copy of array table

        total = sum(table);                 // call the method sum and pass the
array table
        System.out.println("The total is " + total);
        System.out.println("After method invocation completes \n and control has
 returned, values are");
        for (int i=0; i < table.length; i++)
            System.out.print(table[i] + " ");
    }

    public int sum(int [] array) {
        total = 0;
        for (int s=0; s < array.length; s++)      // when the index s equals the
 length, loop will stop
        {
            total = total + array[s];               // find the sum of all of th
e elements in the array
            array[s]= array[s] +1;
        }
        return total;
    }
}
```

Save and **Run** it.

It's nice to know you can do that!

Common Mistakes

Can you find the errors in the **defaultFillGrade()** method below? It gives two errors:

CODE TO TYPE

```
class Test {

    public static void main(String[] args){
        Test myTest = new Test();
        myTest.demo();
    }

    public void demo(){
        String [] grades = new String[4];
        defaultFillGrade(grades);
        for (int i=0; i < grades.length; i++) System.out.print(grades[i] + " ");
    }

    public void defaultFillGrade(String [] grades){
        for (int studentNum = 0; studentNum <= grades.length; studentNum++)
            grades(studentNum) = "A";
    }
}
```

- **studentNum <= grades.length**
  If studentNum **=** grades.length, then the array element with index grades[studentNum] causes a **run-time** *array out of bounds* exception. Remember, the indices start at 0, so they only go as far as the array's length which is -1.
- **grades(studentNum)**
  Java would think this is a call to a method **grades** with a parameter **studentNum**. But Java should be looking at indices in an array. Arrays use square brackets. So the correct syntax for putting the string value of "A" into the array **grades** at location **studentNum** is:
  grades[studentNum] = "A";
  Although this is the **real** problem, what Eclipse will tell you is: **The left-hand side of an assignment must be a variable**. Why? For the reason I said above--Java thinks that **grades(studentNum)** is a method call, and method calls cannot be on the left side of an assignment, because the left side is a memory location.

Edit your Test class as shown in **blue** below:

CODE TO TYPE

```
class Test {

 public static void main(String[] args){
     Test myTest = new Test();
     myTest.demo();
 }

 public void demo(){
   String [] grades = new String[4];
     defaultFillGrade(grades);
     for (int i=0; i < grades.length; i++) System.out.print(grades[i] + " ");
 }

 public void defaultFillGrade(String [] grades){
     for (int studentNum = 0; studentNum < grades.length; studentNum++)
      grades[studentNum] = "A";
 }
}
```

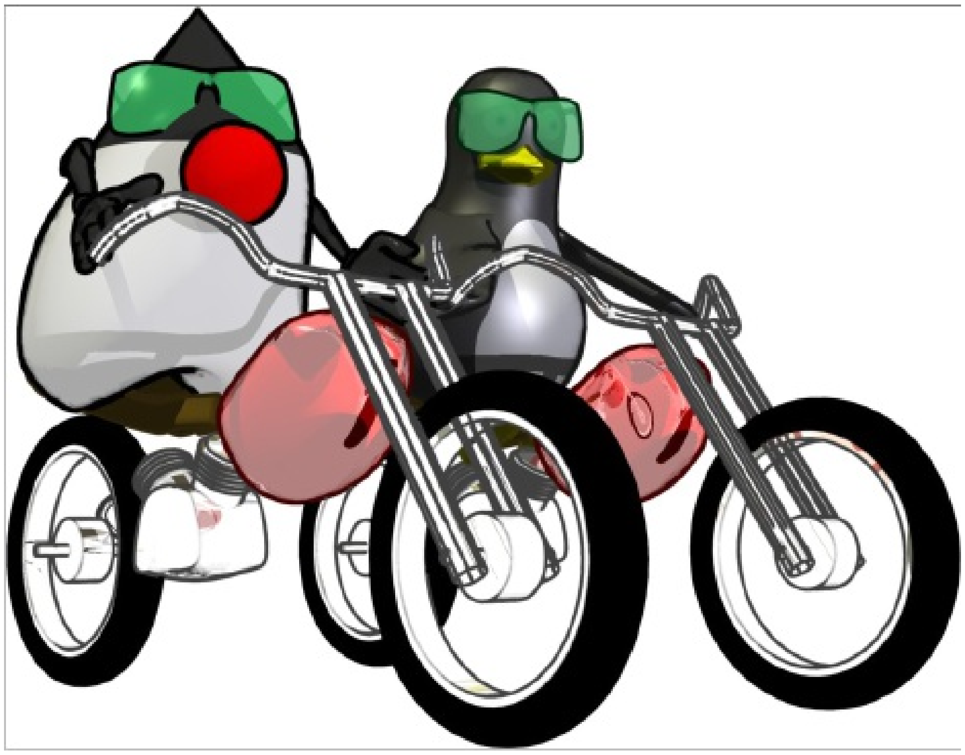
Save and **Run** it.

When this loop is done, the array looks like:

ARRAY grades:	A	A	A	A
INDICES	0	1	2	3

I like that grading scale!

You've worked hard; go take a ride with Duke and Tux to relax before the next lesson on loops.



The Other Loops

Lesson Objectives

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

- create an application that instantiates and tests a class.
- use while and do-while loops.
- change a while loop into an equivalent do-while loop.

Repetition: while, do-while

Enhanced For Loops

When Java added the **Java Collections Framework** to their **Development Kit**, they also added a new *enhanced for* statement to be used in **Generic** classes. Programmers often iterate through an entire set, or collection, or array of things, so Java created some constructs that make doing that easier. The Collections Framework and Generic classes will be studied in greater detail in the next course, but we can use the *enhanced for* right now.

Go to your **java2_Lesson10** Project, and edit the **Beatlejuice** class's **main** method as shown:

CODE TO TYPE:

```
class Beatlejuice {
    public static void main(String[] args) {
         String[] beatles;
         beatles = new String[5];

         beatles[0] = "John Lennon";
         beatles[1] = "Paul McCartney";
         beatles[2] = "George Harrison";
         beatles[3] = "Ringo Starr";
         beatles[4] = "George Martin";

         for(String item : beatles)
             System.out.println("Element is : " + item);

         System.out.println("\nSize of the beatles array is " + beatles.length);
    }
}
```

Save and **Run** it.

We changed this code:

OBSERVE:

```
for (int i=0;  i <= beatles.length; i++)
    System.out.println("Element is : " + beatles[i]);
```

into this code:

OBSERVE:

```
for (String item : beatles)
    System.out.println("Element is : " + item);
```

beatles could have been any array name.

There is no need to specify that the code should start at the beginning of the array and go through each index until the end (as in the standard "for" loop), the **enhanced for** allows you to do something to each **item** in the array.

We only have to specify the array name and the type of elements in the array. **item** is our chosen variable name that temporarily holds each value of the array for each step of the **enhanced for** loop. Let's work thorugh an example of one now.

In the temp project, edit the **Test.java** class as shown:

CODE TO TYPE:

```
class Test {
   public static void main(String[] args){
     int [] arrayValues = {5,6,6,44,6,7,34,4,9,89};
     int sum = 0;
     for (int e : arrayValues) // e is short for element
        sum += e;
     System.out.println("Sum of the array is " + sum);
   }
}
```

Save and **Run** it.

If we just want to iterate through an array, we can use this shortcut. Java recommends using this form of the **for statement** instead of the general form whenever possible.

while

It's true that **for** loops make it easy to manipulate arrays, but not all programming loops use arrays. The two other types of loop constructs, **while** and **do-while**, provide more flexibility.

The grammatical structure for a **while** loop is:

OBSERVE

```
while(condition) {
  statement1;
  statement2;
  ...
}
```

The **while** statement continues testing the condition expression and executing its block until the condition evaluates to `false`.

The three loop components are not part of the `while` construct itself. Only the stop condition is present within the parentheses. The initialization process and increment processes occur outside of them. **YOU**, the programmer, must remember to make and place the initialize and increment statements:

1. The `while` loop variable should be **initialized outside** of the loop.
2. The `while` loop variable should be **incremented inside** of the loop.

Let's create an example of the `while` loop by editing our **Slinky.java**.

Open the **Slinky.java** class in java2_Lesson10, and edit it as shown below:

CODE TO TYPE:

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

    public class Slinky extends Applet implements ActionListener {
        private TextField countInput;
        private int howManyCircles = 0;

    public void init() {

        Label l = new Label("How many circles?");
        add(l);

        countInput = new TextField(3);
     add(countInput);
     countInput.addActionListener(this);

        Button reset = new Button("Reset");
        add(reset);
        reset.addActionListener(this);
    }

     public void paint(Graphics g) {
        int x = 20, y = 20;
        int count=1;                              // declare and set loop varia
ble
        while (count <= howManyCircles) {          // check condition before ente
ring loop block
           g.drawOval (x+count*5, y+count*5, 50, 50);
           count++;                                // increment loop variable
        }
     }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() instanceof Button)
            {
         howManyCircles = 0;
       countInput.setText("");
            }
         else if (event.getSource() instanceof TextField)
            if (countInput.getText().length() == 0)
             howManyCircles = 0;
         else
         howManyCircles = Integer.parseInt(countInput.getText());
      repaint();
    }
}
```

Save and **Run** it. Try testing lots of different numbers (include some big ones). Results for the **while** should look the same as those from the **for** loop.

The while loop components

```
              int x = 20, y = 20;
              int count=1;                                // declare and set loop
 variable
              while (count <= howManyCircles) {          // check condition befor
e entering loop block
                  g.drawOval (x+count*5, y+count*5, 50, 50);
                  count++;                                  // increment loop vari
able
              }
```

In the code above, we can determine where we:

- **initialized the variable**
- **incremented it**
- **checked for the end of looping**

while loops are often used together with **flags**. A **flag** is simply a certain value for a variable that indicates the end (or to alert you to some special case). For example, you may want to set a flag in this case: "While I do not see police, I will drive over the speed limit":
(**while (!police) {speed = "fast"; watching = true;}**)

do-while

The **do-while** loop is *very* similar to the **while** loop. The only difference is that **do-while** loops evaluate their conditional expressions at the bottom of the loop instead of the beginning. So the statements within the **do-while** block are **always executed at least once**.

The structure of a **do-while** loop looks like this:

OBSERVE:

```
 do {
     statement1;
     statement2;
       ...
 } while(condition);
```

Note as in the **while**, the three loop components are not a part of the **do-while** construct. *You*, the programmer, must remember to do the initialize and increment statements.

We will demonstrate the **do-while loop** by editing our **Slinky.java** one more time.

Edit the class as shown below:

CODE TO TYPE:

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

    public class Slinky extends Applet implements ActionListener {
        private TextField countInput;
        private int howManyCircles = 0;

    public void init() {

        Label l = new Label("How many circles?");
        add(l);

        countInput = new TextField(3);
     add(countInput);
     countInput.addActionListener(this);

        Button reset = new Button("Reset");
        add(reset);
        reset.addActionListener(this);
    }

     public void paint(Graphics g) {
        int x = 20, y = 20;
        int count=1;                                   // declare and set loop vari
able
        do {
            g.drawOval (x+count*5, y+count*5, 50, 50);
            count++;                                   // increment loop variable
            } while (count <= howManyCircles);        // check condition after runn
ing loop block
     }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() instanceof Button)
            {
         howManyCircles = 0;
       countInput.setText("");
            }
         else if (event.getSource() instanceof TextField)
            if (countInput.getText().length() == 0)
             howManyCircles = 0;
         else
         howManyCircles = Integer.parseInt(countInput.getText());
      repaint();
    }
}
```

Save and **Run** it. Test it with lots of different numbers, and be sure to include some big ones.

Hey, something's different! Results for the **do-while** **should** be different from the **for** loop and the **while** loop in one important way: the **do-while** will **always** go through the loop body at least once.

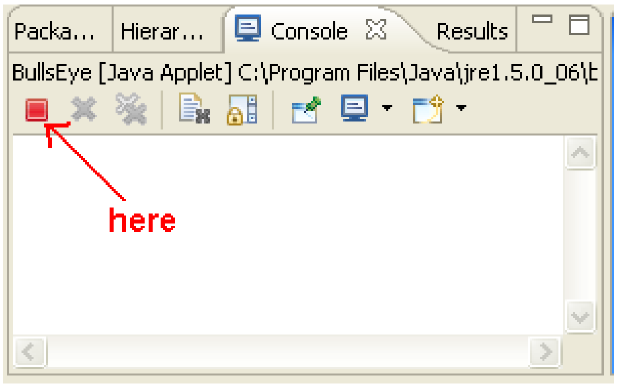
Look at the comments in the code we can determine where we:

- initialized the variable
- incremented it
- checked for the end of looping

Infinite Loops

Sometimes we **want** loops to go on forever. We can make that happen with any loop type. But other times we need to make loops **stop**, especially if we created one by mistake. The method you use to exit a Java program that is stuck in an infinite loop depends on whether you are running an **Applet** or an application.

- If you are running an **Applet** and you have an infinite loop, just close the browser window.
- If you are running an application and you have an infinite loop, use the console's **Terminate** button.



Let's put this concept to use!

Edit the **Test.java** class as shown:

CODE TO TYPE:

```
class Test {
  public static void main(String[] args){
    while (true){
      System.out.println("I am learning lots of Java");
      System.out.println("This course is wonderful! \n");
    }
  }
}
```

Save and **Run** it.

You'll see "flickering" or, if the Console window is open, you might see the println output displaying again and again. To stop it, click the **Terminate** button.

The *flicker* is actually the computer writing to the console very fast. See the scroll bar on the right of the **Console Window**? It will tell you many times your loop ran before you stopped it.

Optional For Loop

Including each of the three expressions of the **for** loop is optional. The most common reason for omitting these expression is to use a **for** to create an infinite loop.

OBSERVE:

```
for ( ; ; )
 {     // infinite loop
       // your code goes here
 }
```

Edit the **Test.java** class as shown here:

CODE TO TYPE:

```
class Test {
  public static void main(String[] args){
    for( ; ; ){
      System.out.println("I am learning lots of Java");
      System.out.println("This course is wonderful! \n");
    }
  }
}
```

Save and **Run** it. Click the **Terminate** button when you've had enough.

Branching Statements

General

We've already seen branching statements in methods that return values. For example, sometimes you want a method to return something before the end of the method is reached. In that case you can use the **return** statement. The **return** statement exits from the method it's in, and returns control flow to where the method was invoked. The **return** statement has two forms: it can return a value or not.

In addition to control changing due to method calls, with nested **if**s and nested loops, there are times when a programmer wants to jump out of a nested loop without going back into the loop in which it was enclosed. Java provides two additional **branching statements** to allow programmers to do this:

- break [label]
- continue [label]

[label] indicates these two statements can be labeled or not, where a label has syntax label:

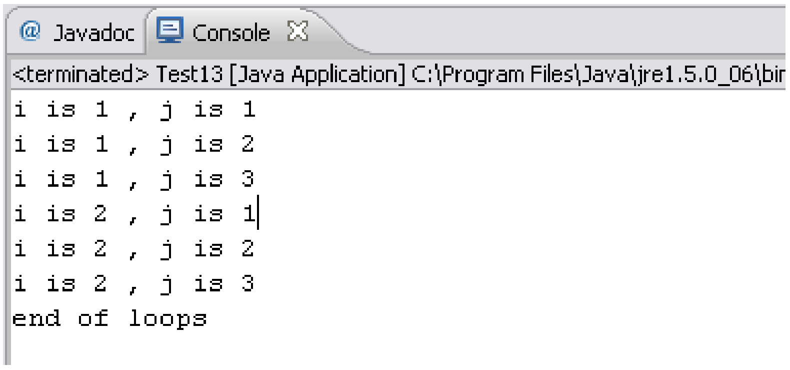
We have already seen unlabeled **break**s in **switch** statements. Here is an example to illustrate the use of **break** with a label.

Edit the **Test.java** class as shown:

CODE TO TYPE:

```
class Test {
    public static void main(String[] args){
        out:
        for (int i = 1; i <= 5; i++)
        {
            for (int j = 1; j <= 3; j++) {
                System.out.println("i is " + i + " , j is " + j);
                if ((i + j) > 4)
                    break out;         // jumps out of both loops
            }
        }
        System.out.println("end of loops");
    }
}
```

Save and **Run** it. Output should look like this:

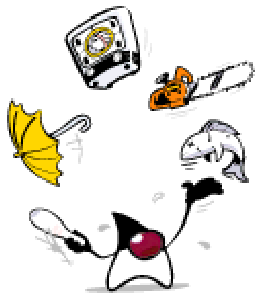


```
<terminated> Test13 [Java Application] C:\Program Files\Java\jre1.5.0_06\bir
i is 1 , j is 1
i is 1 , j is 2
i is 1 , j is 3
i is 2 , j is 1
i is 2 , j is 2
i is 2 , j is 3
end of loops
```

Trace the code to follow why and when the **break** was used.

For more, see the Java Tutorial on branching.

In the next lesson, we'll really put the power of loops to work!



Tracing Code

Lesson Objectives

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

- use nested while loops.
- design and implement the game logic for a tic tac toe game.

How Did I Get Here?

We've come a long way. So far we've covered all of the Java control constructs and quite a few Java Classes. As you continue to work with Java, you'll become familiar with even more of the Java classes and the best ways to use them in your programs.

The larger our programs become, or for that matter, the more loops we have, the harder can be to follow the flow of control. In this lesson, we will practice the technique of **tracing** code. It's the most common way to find logic bugs (Eclipse can help us with our syntax errors, but not with poor logic reasoning), and is an invaluable tool.

We've already seen arrays and **for** loops working together. In this lesson, we'll strengthen our knowledge of arrays in method calls and use 2-D arrays in **nested for loops**. In general, our goal is to be able to trace through code to figure out how it works, and if the output doesn't seem to match up with what we **thought** you programmed, figure out why.

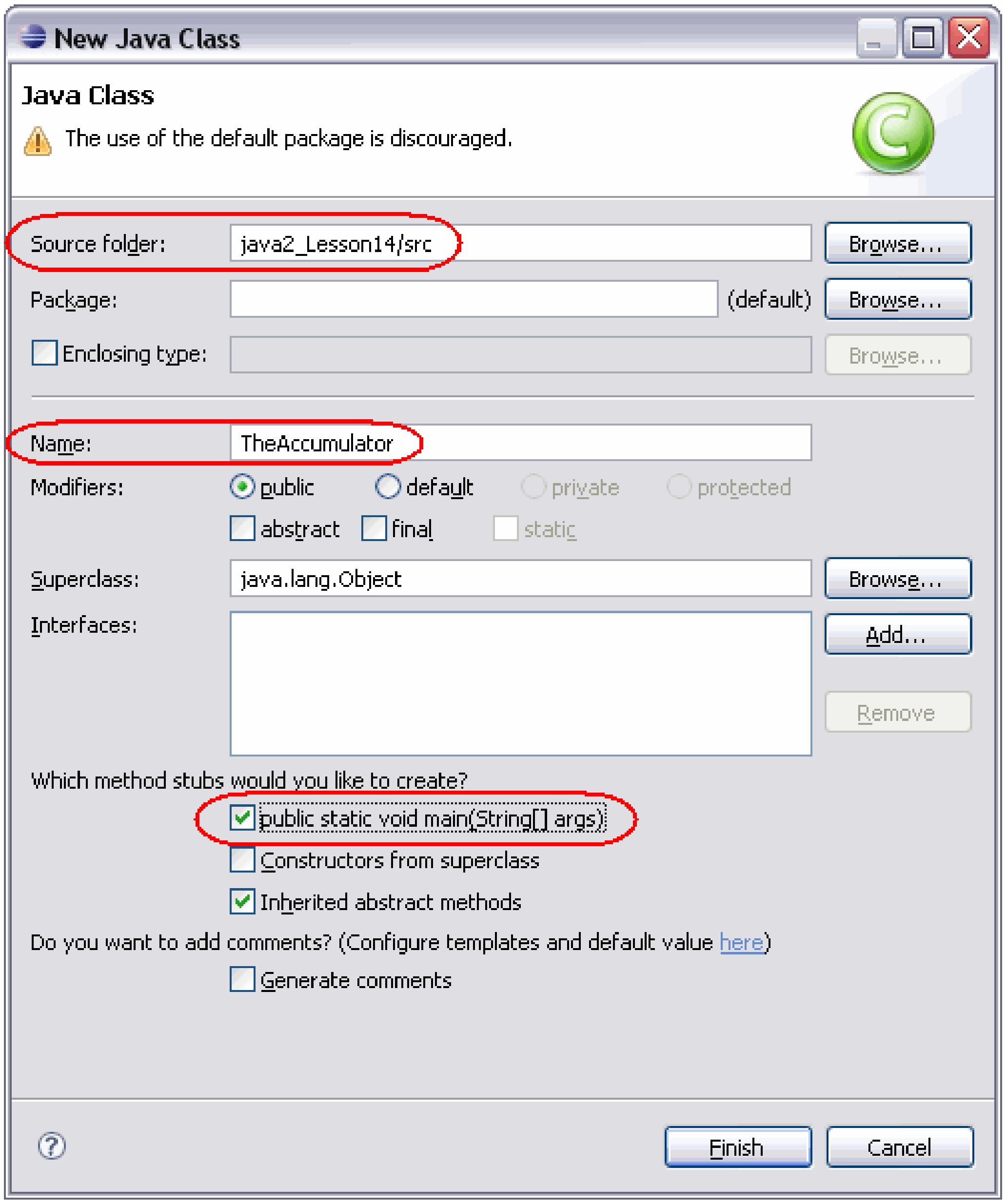
Arrays for Methods--Reminder

Let's review arrays:

1. In some classes, the array itself has to be declared and instantiated.
2. Elements of the array must be declared and instantiated.
3. The array must be filled with these elements.

Now let's try to pass the array as a parameter:

Make a new Java Project and name it **java2_Lesson14**. Inside your nee project, create a new application Class and name it **TheAccumulator**, as shown below:


New Java Class

Java Class

The use of the default package is discouraged.

Source folder: java2_Lesson14/src

Package: (default)

[ ] Enclosing type:

Name: TheAccumulator

Modifiers: [● public] ○ default ○ private ○ protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

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

Type the class named **TheAccumulator**, as shown below:

CODE TO TYPE:

```
class TheAccumulator {

    public static void main(String[] args){
        TheAccumulator myInstance = new TheAccumulator();
        int myArray [] = {1,2,3,4,5,6,7,8,9,10};
        int myArrayTotal = myInstance.sum(myArray);
        System.out.println("Total is : " + myArrayTotal);
    }

    // method definition

    public int sum( int [] array) {
        int total = 0;
        for (int s=0; s < array.length; s++)
            total = total + array[s];
                        // find the sum of all of the elements in the array
                        // total here is called an accumulator - it accumulates the sum
        return total;
    }
}
```

Save and **Run** it.

OBSERVE: Instantiating an Array and Passing it to a Method

```
    int myArray [] = {1,2,3,4,5,6,7,8,9,10};
    int myArrayTotal = myInstance.sum(myArray);
```

Once you've initialized **myArray** as an array of **int**, Java knows what it is. From then on, it is a variable like any other. When you pass it, you do not need to state again that it is an array—Java knows. So, when you call the **sum()** method, you don't need the square brackets or any other indicators; **myArray** is enough.

OBSERVE: Method Declaring an Array as a Formal Parameter

```
    public int sum( int [] array)
```

You do, however, need to specify which types the formal parameters are expected to be in a method definition. If you are defining a method and want it to be passed an array, you need to specify the parameter as **array**.

Objects in Java are passed by passing the reference to the Object (a handle or pointer to the place in memory where the Object/array is stored). This means that when you pass an array as a method parameter, you are not giving the method the content of the array, but a reference to it in memory.

This means programmers could inadvertently change or corrupt the array, because they might pass it to a method that alters it. Primitive data types are passed by value, so they are not as "dangerous" to pass. In the next section, we'll see an example to help us understand why we need to be careful of such **side effects**.

Always Look Before Passing (or Hiring)

Suppose a company has hired a student to do some programming. They want a Method that will take an Array of their sales numbers to find out which salesperson has the largest sale that week. The student needs to write a Class to test their existing code by creating the Array in a check method and sending the Array to the method **highest()**. The programmer needs the method to return the highest sale and also to print which salesperson made a sale in that amount. Let's take a look at an example to see how the student proceeded.

Make a new class called **Test**. Type the **Test** class as shown:

CODE TO TYPE:

```
class Test {

public static Test student;

    public static void main(String[] args){
     student = new Test();
     student.check();
    }

    public void check(){
        int s;
        int[] sales = {3400, 2233, 3433, 754, 5664, 42};       // instantiate an
d populate the array sales
        int winner = student.highest(sales);                    // call the meth
od highest and pass the array sales;

        s=0;                                                    // while loop to
 find the highest person
        while (sales[s] ==0)
         s++;

        System.out.println("The highest sale was salesperson " + (s+1));   // sa
lespeople do not like to be named 0 so add one to indices

        for (s=0; s < sales.length; s++)
            System.out.println("Salesperson " + (s+1) + " sold "+ sales[s]);
    }

    public int highest(int [] passedSales) {
        int currentHigh = passedSales[0];       // instantiate the high sale to
be the first salesperson's sales initially
        int highGuy = 0;                        // instantiate the high person

        for (int s=0; s < passedSales.length; s++)
            if (passedSales[s] > currentHigh)                // get highest valu
e
                {currentHigh=passedSales[s];                   // remember curre
nt high
                passedSales[highGuy] = 0;                     // this one isn't
highest anymore, set to 0
                highGuy = s;                                  // new highGuy
            }
            else passedSales[s]=0;                         // only keep value at
 highest so can retrieve

        return currentHigh;
    }
}
```

Save and **Run** it.

Note This is **not** good code, but we're going to follow it anyway and address the problems in it.

Because he could only pass back one thing, the programmer passed back the highest sale value and then created the array to be all zeros unless it was the top salesperson. Then, he searched through the array to find the value that was not 0 to determine which salesperson had the higest sale. (The method has an array of sales as a parameter.)

Of course, the array being passed back and forth *is* the information about sales, so using it this way destroys the original data.

You do not want to use an array that holds such valuable information as a method argument--make a copy instead. Also, design your code so that other methods are not reliant on strange manipulations you have made. As Albert Einstein once said, "Everything should be made as simple as possible, but not simpler."

Tip Return the *index* of the highest value. If you have the index, you have the value at that index (that is, knowing **i** allows you to get **sales[i]**).

Nested for

In our last project, we made heavy use of nested **for** loops. Considering the amount of code we write to create them, **for loops** give us a big bang for our buck. Let's look at some examples.

Edit the **Test** class as shown below:

CODE TO TYPE:

```
class Test {

    public static void main(String[] args){
        for (int i = 0; i < 4; i++)
            for (int k = 0 ; k < 5; k++)
                System.out.println("here I am again");
    }
}
```

Can you tell how many times it printed and why *without* the help of running it? Let's trace it.

OBSERVE:

```
for (int i = 0; i < 4; i++)
    for (int k = 0; k < 5; k++)
```

The outer loop starts with **i = 0**, iterates while **i < 4**; and increments by one (**i++**) for each iteration. It will iterate with i = 0, 1, 2, and 3—or 4 times. For each one of these outer loops, the inner loop starts with **k = 0**; interates while **k < 5**; and increments by one (**k++**) for each iteration. It will iterate with k = 0, 1, 2, 3, and 4—or 5 times. The inner loop iterates 5 times for each of 4 outer-loop iterations; 5 * 4 = 20.

Programmers often use **System.out.printlns** to help them trace their code and see what it's doing while it runs.

Edit the **Test** class's **main()** method as shown below:

CODE TO TYPE:

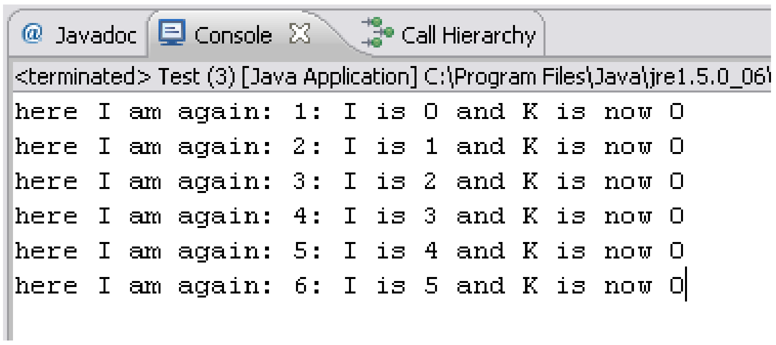
```
class Test {

    public static void main(String[] args){
        int n=0;

        for (int i = 0; i < 4; i++)
            for (int k = 0 ; k < 5; k++)
            {
                System.out.print("here I am again: ");
                n++;
                System.out.println(n + ": i is " + i + " and k is now " + k);
            }
    }
}
```

Save and **Run** it.

CODE TO TYPE:

```
class Test {

    public static void main(String[] args){
        int n=0;

        for (int i = 0; i < 6; i++)
            for (int k = 0 ; k < 1; k++)
            {
                System.out.print("here I am again: ");
                n++;
                System.out.println(n + ": i is " + i + " and k is now " + k);
  }
            }
}
```

Save and **Run** it.


@ Javadoc | Console | Call Hierarchy

```
<terminated> Test (3) [Java Application] C:\Program Files\Java\jre1.5.0_06\
here I am again: 1: I is 0 and K is now 0
here I am again: 2: I is 1 and K is now 0
here I am again: 3: I is 2 and K is now 0
here I am again: 4: I is 3 and K is now 0
here I am again: 5: I is 4 and K is now 0
here I am again: 6: I is 5 and K is now 0
```

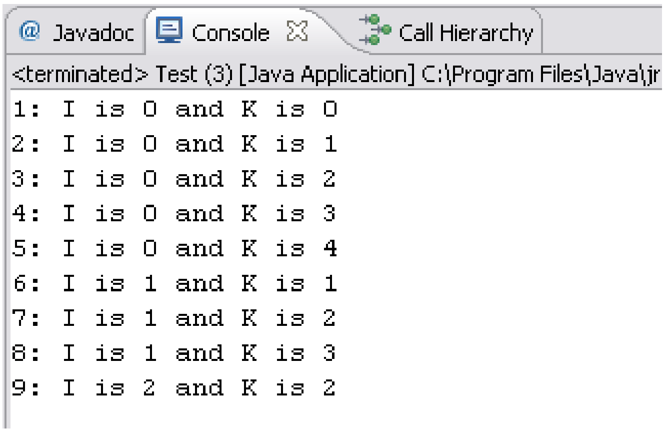
Edit the code as shown below. (Before running, trace it to determine how many times it will print this time.):

CODE TO TYPE:

```
class Test {

    public static void main(String[] args){
        int n = 0;
        for (int i = 0; i < 5; i++)
            for (int k = i ; k < 5-i; k++)
            {
                n++;
                System.out.println(n + ": i is " + i + " and k is " + k);
            }
    }
}
```

Notice the use of the variable **i** in the inside **for** loop.

Save and **Run** it.



```
<terminated> Test (3) [Java Application] C:\Program Files\Java\jr
1: I is 0 and K is 0
2: I is 0 and K is 1
3: I is 0 and K is 2
4: I is 0 and K is 3
5: I is 0 and K is 4
6: I is 1 and K is 1
7: I is 1 and K is 2
8: I is 1 and K is 3
9: I is 2 and K is 2
```

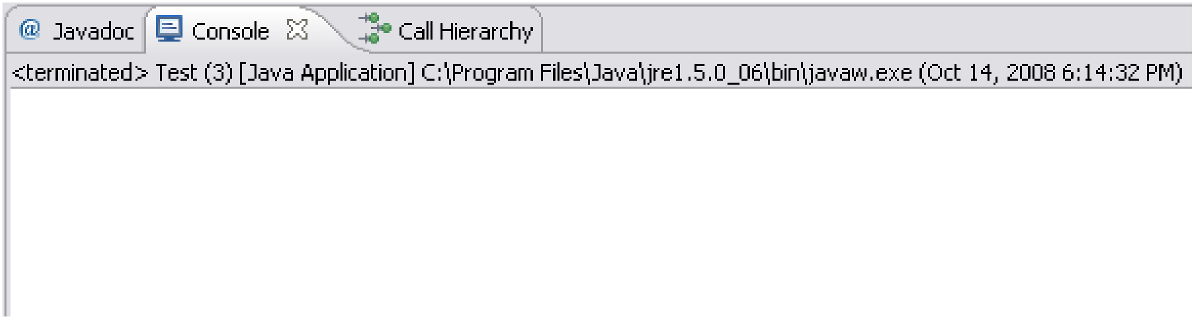
Why did it print nine times? Follow the interaction of the values of **i** and **k**.

Edit the code again as shown here:

CODE TO TYPE:

```
class Test {

    public static void main(String[] args){
       int n = 0;
       for (int i = 5; i < 0; i--)
           for (int k = 0 ; k < 1; k++)
           {
               System.out.print("here I am again: ");
               n++;
               System.out.println(n + ": i is " + i + " and k is " + k);
           }
    }
}
```

Save and **Run** it.



```
<terminated> Test (3) [Java Application] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Oct 14, 2008 6:14:32 PM)
```

Hey, there's nothing there! What happened? Well, there's no output because **i** is initially **5**. **5** is not less than **0**, so the loop is never entered.

Tracing do-while

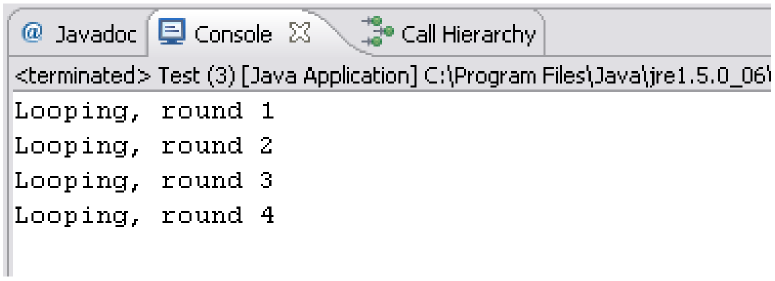
Let's trace some **do loops** (with the help of **System.out.println**).

Edit the **Test** class's **main()** method as shown:

CODE TO TYPE:

```
class Test {

    public static void main(String[] args){
        int x = 1;
        do {
            System.out.println("Looping, round " + x);
            x++;
        }  while (x <= 4);
    }
}
```

Save and **Run** it.



```
<terminated> Test (3) [Java Application] C:\Program Files\Java\jre1.5.0_06\
Looping, round 1
Looping, round 2
Looping, round 3
Looping, round 4
```

How about this one. Edit the **Test** class's **main()** method as shown below:

CODE TO TYPE:

```
class Test {
    public static void main(String[] args){
        int n = 6;
        int counter = 0;
        do {
            counter = counter + 2;
        } while (counter <= n);
    }
}
```

Save and **Run** it.

There's no output in this program, so you'll need to think!

Trace: Follow the assignment of the **counter** variable as the looping occurs. It's initialized as **0**. In the first iteration of the **do loop**, counter = counter + 2 makes it 2; 2 <= n (which is 6), so looping continues. In the second iteration, counter becomes 4 (still <=6). In the third iteration counter beomes 6 (still <=6). In the fourth iteration, it becomes 8, which is NOT <=6), so looping stops.

How can we be sure? Edit the **Test** class's **main( )** method as shown:

CODE TO TYPE:

```
class Test {
    public static void main(String[] args){
        int n = 6;
        int counter = 0;
        do {
            counter = counter + 2;
            System.out.println("counter is:  " + counter);
        } while (counter <= n);
    }
}
```

Save and **Run** it. System.out.println can, indeed, be handy!

Using a **for** loop, write code to generate the following output:

```
-1000
-996
-992
-988

...output deleted but you should have the whole sequence...

988
992
996
1000
```

Try it first yourself. Your answer should look something like this:

OBSERVE:

```
public class testMe {

    public static void main (String args[]){

         for(int c = -1000; c <= 1000; c=c+4)
           System.out.println(c);
    }
}
```

Tracing Decisions

Here's an **if** question to trace. Try to determine the exact range of values of the variable x that causes the code below to print the letter B:

CODE TO CONSIDER:

```
    if (x <= 200)
    if (x < 100)
    if (x <= 0)
    System.out.println("A");
    else
    System.out.println("B");
    else
    System.out.println("C");
    else
    System.out.println("D");
```

When you trace **if** statements, try an example of *every* possible case (remember that each **else** corresponds to the nearest **if**).

When you try all of these values of **x**:

- not less than 200
- equal to 200
- less than 200
- less than 200 but not less than 100
- less than 200 and less than 100
- less than 100
- less than 100 but not less than 0
- less than 100 and less than 0

You'll see that:

- for {x | x <= 0}, "A" will print
- for {x | 0 < x < 100}, "B" will print
- for {x | 100 <= x <= 200}, "C" will print
- for {x | x > 200}, "D" will print

Therefore, "B" will print for values of x in the range 1 through 99.

So, could you use the above forms in code rather than the nested **if**s? No. You need to use well-formed logic statements, such as (100 <= x && x <= 200).

Here's another loop (with a switch) to trace:

CODE TO CONSIDER:

```
int count;
for (count = 1; count <= 12; count++)
 {
    switch (count % 3)
    {              // mod 3
      case 0:
        g.setColor(Color.red);
        break;
      case 1:
        g.setColor(Color.blue);
        break;
      case 2:
        g.setColor(Color.green);
        break;
     }
    g.drawLine(10, count * 10 , 80, count * 10);
 }
```

In the java2_Lesson14 project, create a new class named **Lines**, with the Superclass **java.applet.Applet**. Type the **Lines** as shown below:

CODE TO TYPE

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

public class Lines extends Applet {

    public void paint(Graphics g){
        int count;
        for (count = 1; count <= 12; count++) {
            switch (count % 3) {              // mod 3
                     case 0:
                             g.setColor(Color.red);
                             break;
                     case 1:
                             g.setColor(Color.blue);
                             break;
                     case 2:
                             g.setColor(Color.green);
                             break;
            }
        g.drawLine(10, count * 10 , 80, count * 10);
        }
    }
}
```

Save and **Run** it. Compare the output with a trace of the code. Pay patricular attention to the **drawLine** parameters to see how count was used to move the lines.

More About Flow Control

You can download a Java application here that will help you visualize the flow of control through methods. It will automatically install itself into your learning environment. Place the **java2_FlowControl** project in your **java2_Lessons** working set. Run the **JavaFlowControlExample** class as a Java Application.

We've seen that all algorithms are made up of the following **control** constructs that direct the flow of the program:

- Sequences (assignment statements, IO calls)
- Decisions and selections (if/then, switch)
- Repetitions and loops (while, for, do-while)
- Method invocation

Summary of Control Constructs

Branching

if/else>

(Condition must result in boolean: not 0 or 1.)

OBSERVE:

```
if (Boolean)
{
    statements;
}
else
{
    statements;
}

if (a instanceof b) ...
```

switch

Expressions must be constants: int, byte, short, or char (convertible via casting to int (automatic promotion)).

OBSERVE:

```
switch (expr1)
{
    case expr2:
        statements;
        break;
    case expr3:
        statements;
        break;
    default:
        statements;
        break;
}
```

Consider for some graphics method specify Class Variables:

OBSERVE:

```
static int SQUARE = 1;
static int CIRCLE = 2;
  ...

int currShape = 0;
  ...
switch (currShape)
{
    case (SQUARE): statements; break;
    case (CIRCLE): statements; break;
}
  ...
```

Alternatively, sometimes **constant** variables are set:

public static final int YES = 0, NO = 1, CANCEL = 2;

These variables are constant because the **final** keyword specifies that they can't be changed. If there are no **break**s in a case, control drops through, which means that once there's a match, all code that follows is executed.

Repetition/Loops

(while, do and for--same as in C, C++)

while loops

OBSERVE:

```
while (boolean)
{
    statements;
}
```

do-while loops

OBSERVE:

```
do
{
    statements;
} while (boolean);
```

for loops

OBSERVE:

```
for (init expr; test expr2; increment expr3)
    { statements; }

for ( i=1 ; i < 10 ; i++)
    (start, stop, increment)
```

For reinforcement of the materials covered in this course, see the Java Tutorial on Language Basics. When you finish with this first Java series, you will be **hands-on familiar** with all of the topics in the Java Tutorial on Learning the Java Language.