Introduction to Java

and

Object Oriented Programming

University of Bridgeport

Dr. Julius Dichter

Fall 2008

Table of Contents

Introduction to Java Programming 6

Procedural Programming 6

OO Programming 6

Object 6

Class 6

Java Application 6

Java Applet 6

Creating, compiling and executing a Java application/applet using command prompt. 7

Java Application 7

Simple HelloWorld Example Java Application 8

Java Applet 9

Applet Viewer Output for HelloWorldApplet.java 11

The Java Programming Language 11

Algorithm 11

Pseudocode 11

Control Structures 11

If/Else Conditional Logic 11

Repetition (The while loop) 13

Program Example: Average 14

Java class Members and Instance Members 15

Java Data Member 16

Sentinel Controlled loop 17

Basic Java Syntax 18

Assignment Operators 18

Increment / Decrement Operators 19

Pre / Post increment (decrement) operators 19

Basic Java Operators 22

Java Built-in types 22

Escape Sequences 22

Counter Loops 23

while loop *24*
for loop *24*
Conditional Logic: switch statement *25*
SwitchTest Java program *26*
Discussion Points for SwitchTest.java *28*
Basic OO Terms *31*
Class *31*
Derived class *31*
Base class *31*
Object Reference *31*
Object *31*
Basics of Java Applications *32*
Multiple line comments *32*
Single line comment *32*
Doc comment *32*
Defining a variable *32*
Defining and initializing a variable *32*
*Using class **boolean*** ***33***
Type Casting *33*
Java Operator List *34*
The conditional operator *34*
break statement *37*
continue statement *37*
Logical Operators *39*
Logical Examples *39*
De Morgan's Laws *40*
Java API **40**
Java API packages **41**
Methods **43**
Casting in Java **45**
Basic properties of Variables and Methods **47**
Instance and Static Objects **47**
Scope rules **47**
Class scope *47*

Block scope 48
Program Example: Scoping 49
Recursion **50**
Implementation of Recursion 51
Method Overloading **54**
Applet Class Methods **54**
Arrays in Java **55**
Declaring and Allocating Java Arrays 56
Example of a Simple Array Usage 57
Initialization of Java arrays 59
Array Size 59
Recursions and arrays **61**
Parameter passing (in arrays and in general) **64**
Quicksort and Binary Sort Methods **65**
Quicksort method 65
Recursive Binary Search method 66
Higher Dimensioned Arrays **66**
Defining Multidimensional Arrays **67**
3-D Array Example 68
Example of ragged 3-D Example 69
Arrays and Objects **70**
Copying Arrays **71**
Designing Classes **72**
Public and Private keywords **73**
Private Data Members **74**
Class *composition* **76**
The *finalize* method **80**
Static Members (Data and Methods) **81**
The *final* keyword **83**
Object Oriented Programming **84**
Types of members in a class **85**
How Protected is *protected?* **86**

Overriding the Defined Methods in Superclass 88
The Relationship between Superclass and Subclass Objects 88
Defining Java References 89
Instantiating Java Objects 90
Java Objects Assigned Across Inheritance Hierarchy 90
Multiple Inheritance 95
Polymorphism 100
Dynamic Behavior of Objects 100
Abstract Polymorphism Example 101
Dynamic Binding 102
Abstract and Concrete Classes 103

Introduction to Java Programming

Procedural Programming – also called imperative programming, requires the programmer to make a series of calls to functions to accomplish the task.

OO Programming – A set of classes are defined which interact and produce the desired program behavior

Object – A collection of data and operations on the data: this is an instance
An object knows how to manipulate itself (i.e. print, for example)

Class – The template for a class defining the data and operations components

A class contains members:

1. *fields –* data which belongs either to the class or to a particular object
2. *methods –* operations which belong either to the class or to a particular object. Methods are similar to C language functions.

Java Application – a stand-alone application which can run on the target machine

Java Applet – a program which can run in a GUI environment such as a World Wide Web browser or the JDK ***appletviewer***

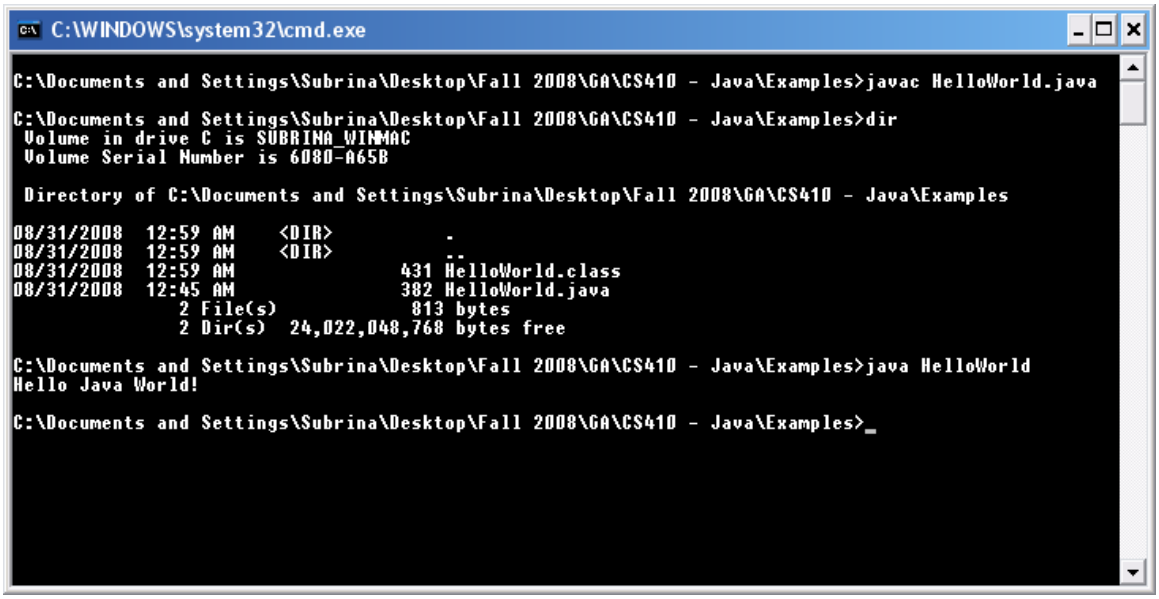
Creating, compiling and executing a Java application/applet using command prompt.

Java Application

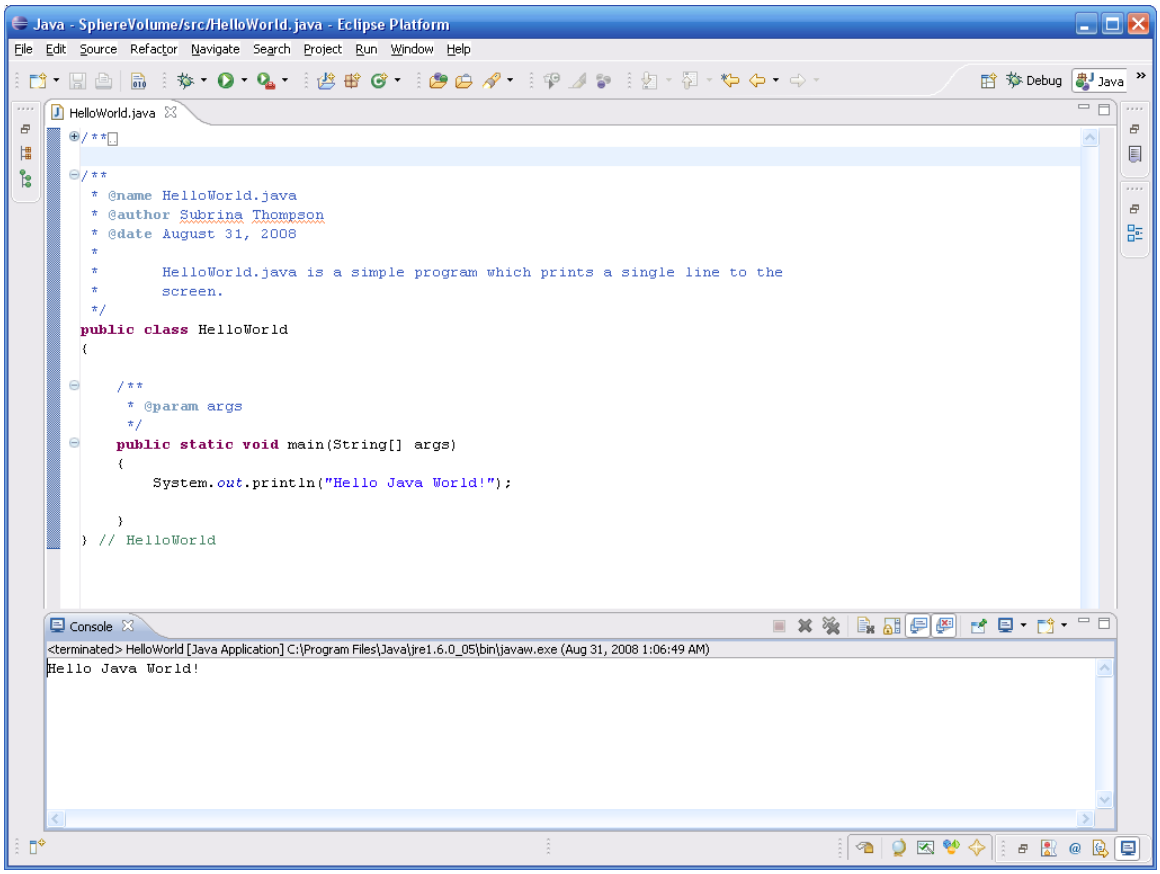
Step 1: Create the java file in any editor such as EditPlus,Notepad, Eclipse, etc.

```
/**
 * @name HelloWorld.java
 * @author Subrina Thompson
 * @date August 31, 2008
 *
 *       HelloWorld.java is a simple program which prints
 *       a single line to the screen.
 */
public class HelloWorld
{

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

      }
} // HelloWorld
```

Step 2: Compile and Execute the program using Windows command Prompt



Simple HelloWorld Example Java Application

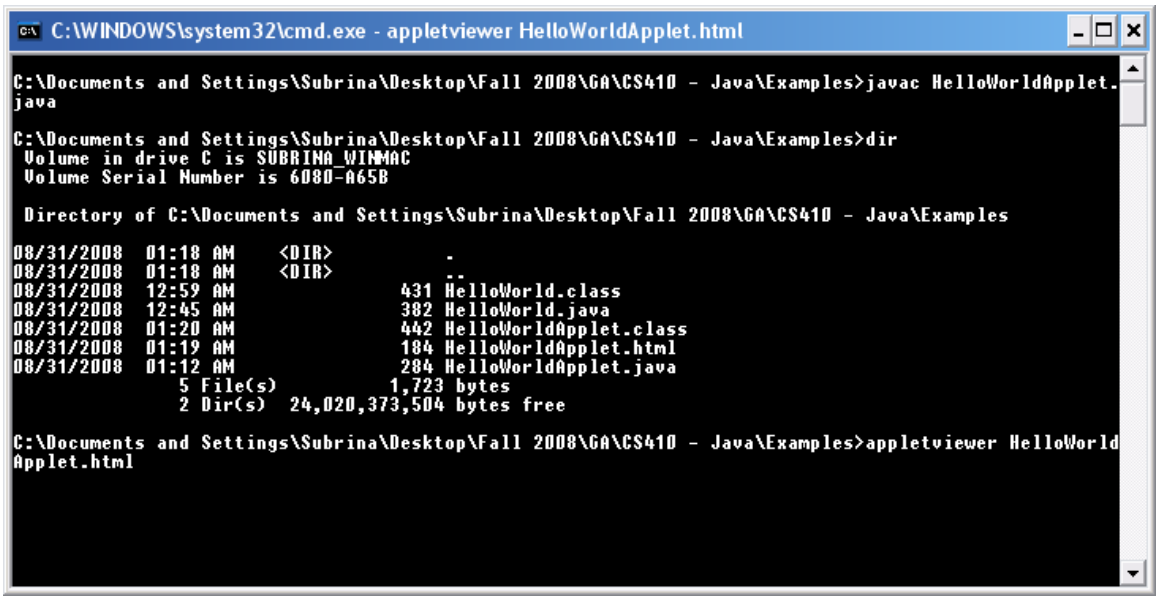


Java Applet

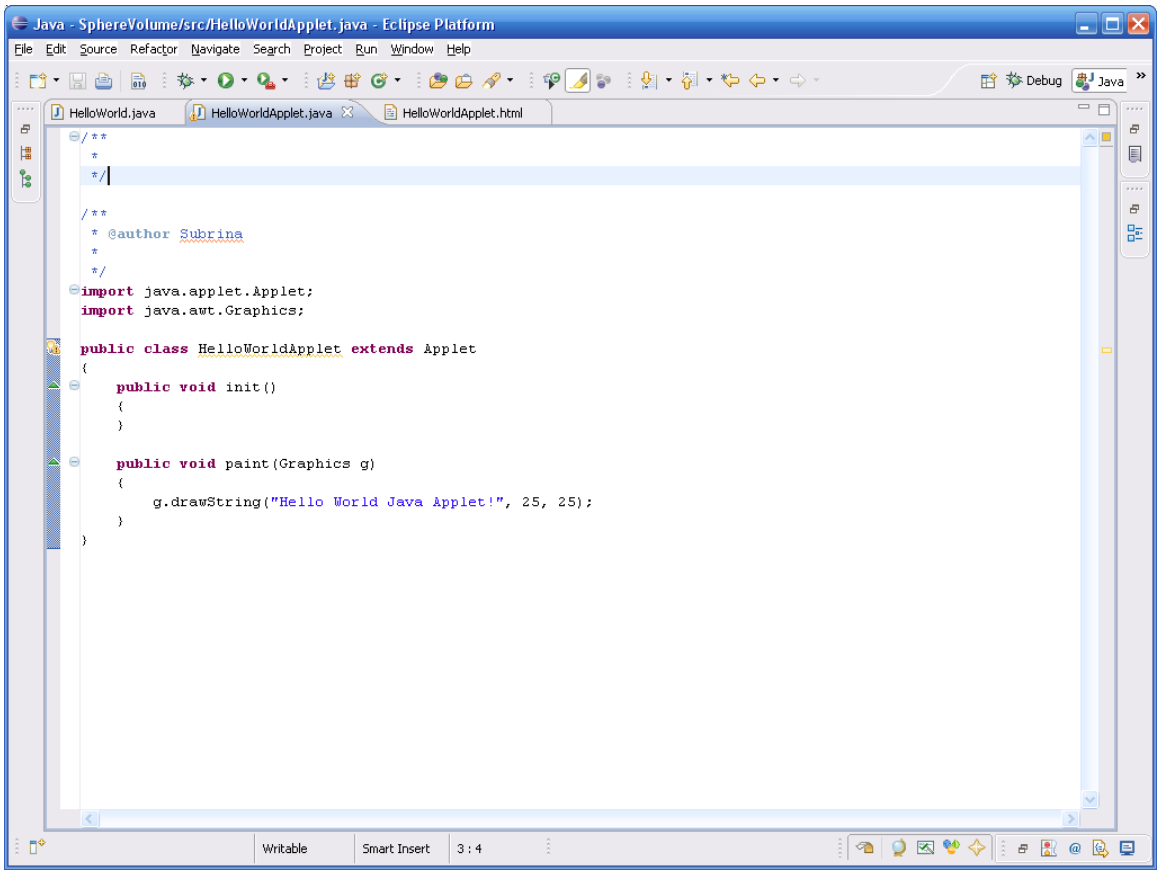
Step 1: Create the java file in any editor such as EditPlus,Notepad, Eclipse, etc.

```
/**
 * @name HelloWorld.java
 * @author Subrina Thompson
 * @date August 31, 2008
 *
 *       HelloWorld.java is a simple program which prints
 *       a single line to the screen.
 */
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorldApplet extends Applet
{
      public void init()
      {
      }

      public void paint(Graphics g)
      {
            g.drawString("Hello World Java Applet!", 25, 25);
      }
}
```

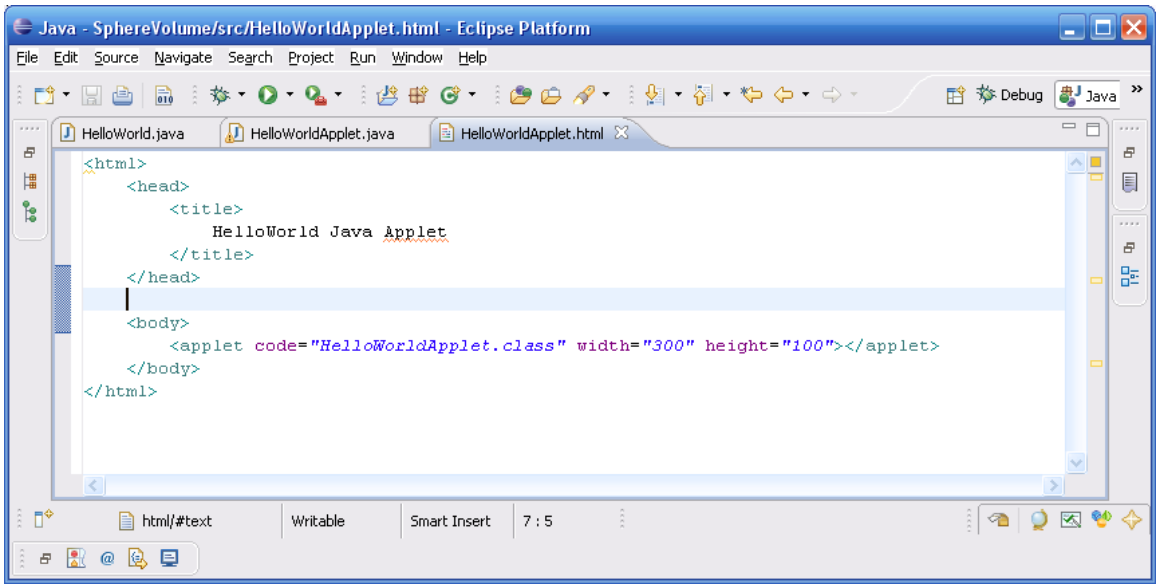
Step 2: Compile and Execute the program using Windows command Prompt



Simple HelloWorldApplet Example Java Applet



The HelloWorldApplet.html file is a HTML web page which looks like the following



Applet Viewer Output for HelloWorldApplet.java

Note: Java Applet may be run from a Browser (Firefox or IE). It is convenient to develop Applets in the **appletviewer.** It is the JDK *testbed* for Applets.

The Java Programming Language

Algorithm – A sequence of steps which achieves a particular goal.

Pseudocode – A non-language specific set of instructions to implement the algorithm. It is a program design aid to assist the programmer in getting to the actual solution.

Control Structures – Structured programs require only three control constructs: *sequence, selection, and repetition.*

If / Else Conditional Logic

```
if (grade >= 60)
      System.out.println("Passed");
else
      System.out.println("Fail");
```

```
if (grade >= 60)
      System.out.println("Passed");

System.out.println("Fail");
```

Nested if statement

```
if (x > 0)
      if (y > 0)
            System.out.println("x and y are both positive");
```

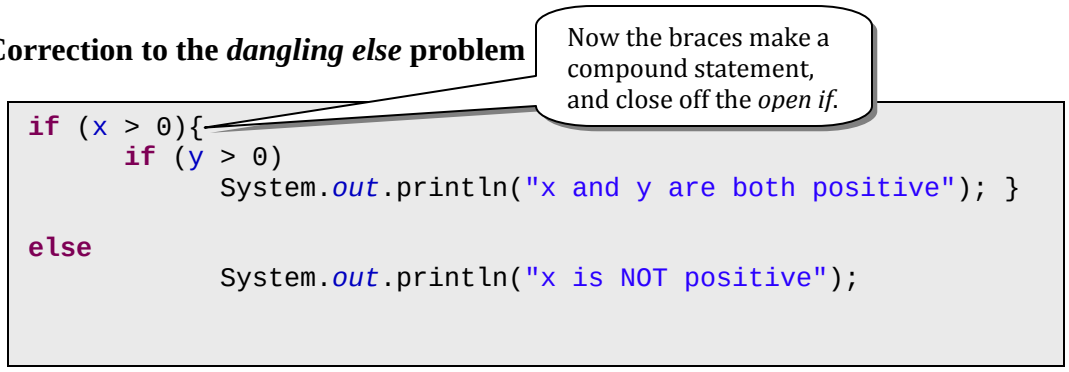
```
if (x > 0)
      if (y > 0)
            System.out.println("x and y are both positive");

else
      System.out.println("x is NOT positive");
```

else matches to last *open* **if** producing a logical *error!*

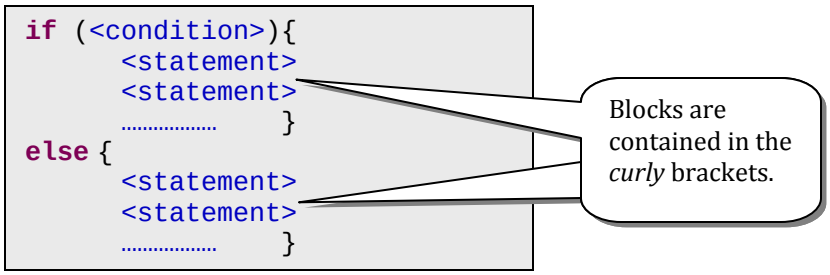
The interpretation of the code above is as follows. Note that there is no matching *else* for the first *if*.

```
if (x > 0)
      if (y > 0)
            System.out.println("x and y are both positive");

      else
            System.out.println("x is NOT positive");
```

Correction to the *dangling else* problem

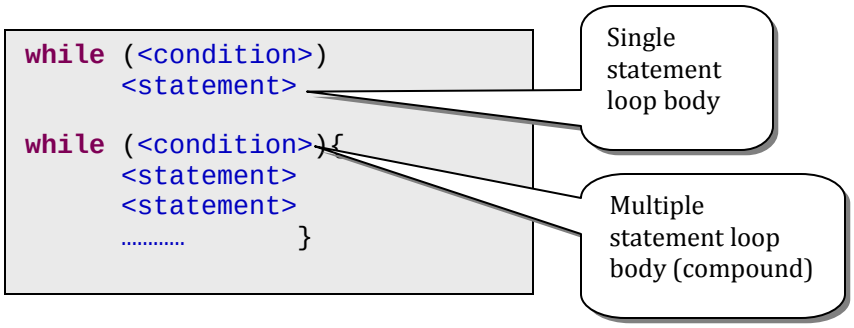


Compound statements can be used to make multiple statements execute as part of the *if* or the *else* block.



Repetition (The *while* loop)

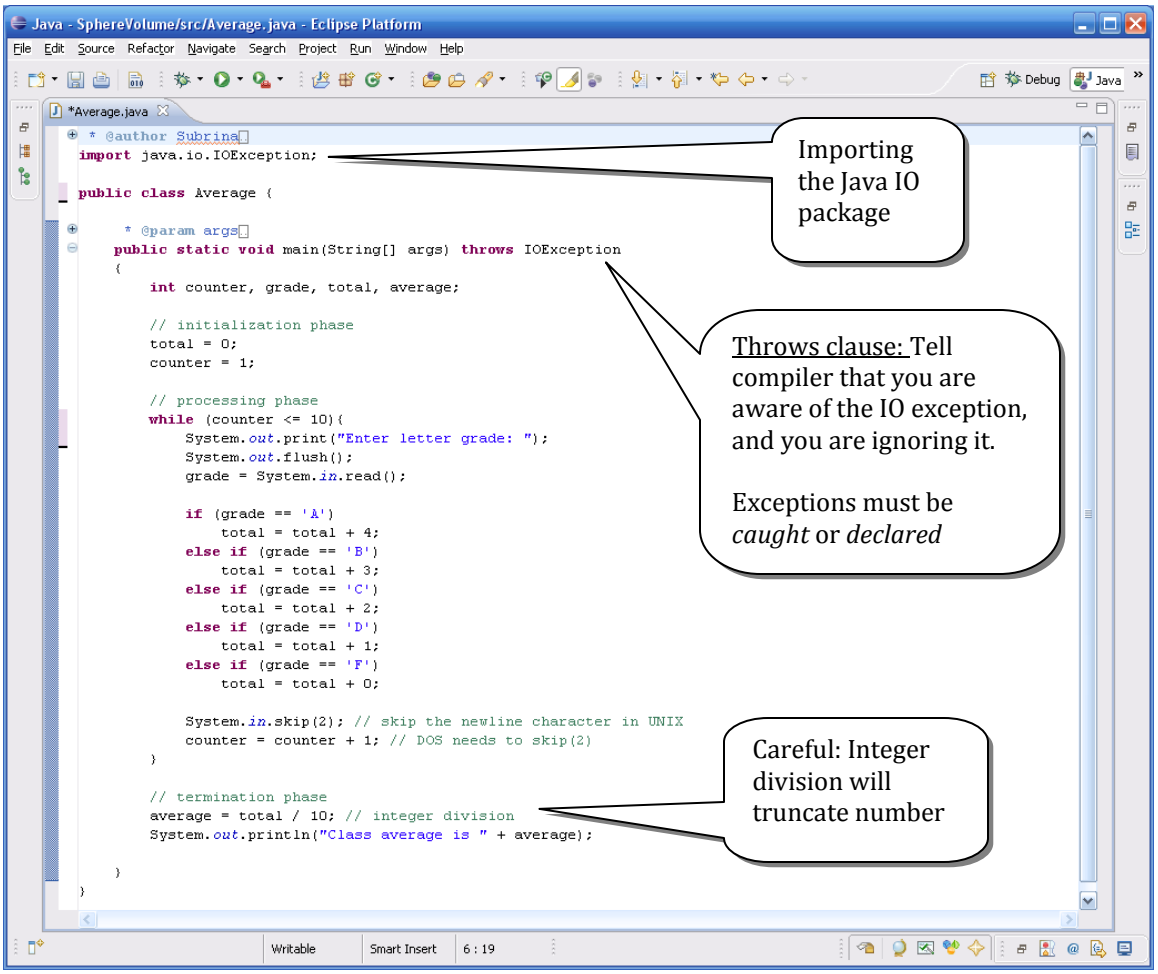
While loop – Repeats a statement (or a sequence of statements) *while* a condition remains *true*.



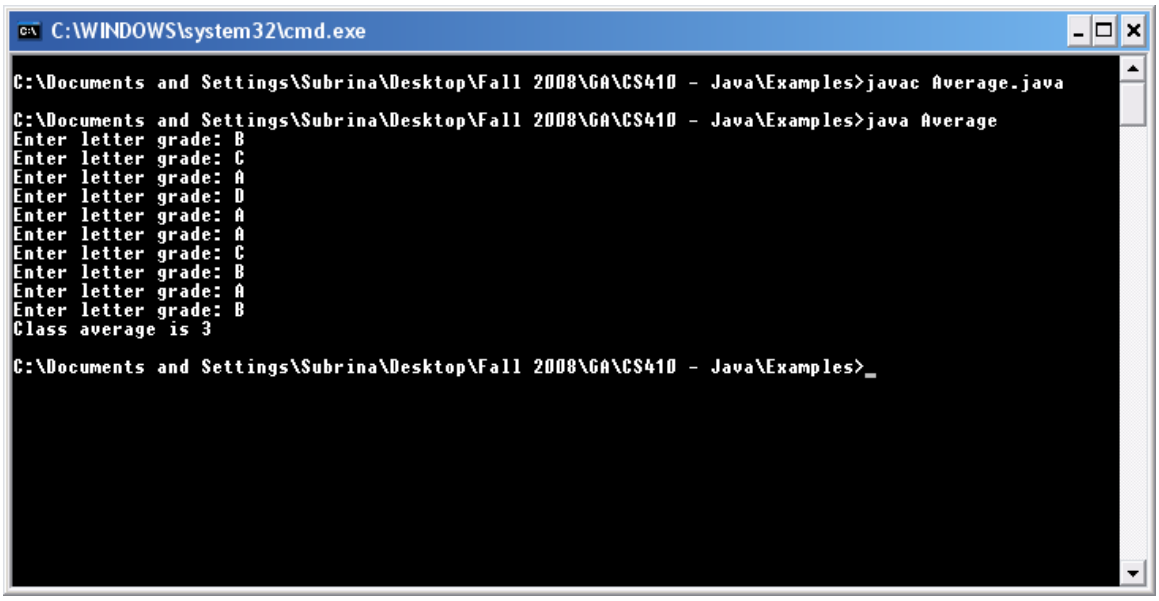
```
int k = 2;

while (k < 50){
      System.out.println("k = " + k);
      k *= 2;
}
```

```
k = 2
k = 4
k = 8
…
k = 32
```

Complete Program Example



Output



Java stores characters as integers. Java can determine the type of parameter to *print* by its type or type cast.

Java stores characters in two bytes in a code format called *Unicode*. This is a superset of the traditional ASCII character encoding used in the past (and today)

```
System.out.println("The character " + 'B' + " has value " + (int) 'B');
```

Java determines the type to output by the type of the expression

```
→ The character B has value
```

Java has *class* members as well as *instance* members

Class members exist as a part of the class and are independent of any object of that class. No object need ever be instantiated to access a class member. Access is always through the **class** itself. Java refers to these members as *static.*

Example: **double** ***d*** **= Math**.random( ) or **Math**.PI

Instance members exist as a part of an object, that is, an instance of the class. Referencing instance members is always through the **object**.

Example: **Fraction** f = new **Fraction( )**
f.*setValue*(3,4);

A **class** *data member* (**System**.*out*)

The class System never *instantiates* an object. Therefore we note that the that *out* must be a class variable. The dot (**.**) is how we *tie-in* the variable to its class/object.

The *out* variable is also an object itself. It is an instance of the **PrintStream** class. Then *println*( ) is an instance method of the *out* object.

Both *method* and *variable* members are accessed by the dot (**.**) operator

Example of a *sentinel* controlled loop. A *double cast* is used to get a floating point result

```
/**
 * @author Subrina
 *
 */
import java.io.IOException;

public class Average
{

	/**
	 * @param args
	 */
	public static void main(String[] args) throws IOException
	{
		double average;
		int counter, grade = ' ', total;

		total = 0;
		counter = 0;

		while (grade != 'Z')
		{
			System.out.print("Enter letter grade: ");
			System.out.flush();
			grade = System.in.read();

			if (grade == 'A')
				total = total + 4;
			else if (grade == 'B')
				total = total + 3;
			else if (grade == 'C')
				total = total + 2;
			else if (grade == 'D')
				total = total + 1;
			else if (grade == 'F')
				total = total + 0;

			System.in.skip(2); // skip the newline character, in this case
			counter = counter + 1;
			System.out.print("Enter Letter grade, Z to end: ");
			System.out.flush();
			grade = System.in.read();
		}

		// getting the results
		if (counter != 0)
		{
			average = (double) total / counter;
			System.out.println("Class average is " + average);
		} else
			System.out.println("No grades were entered");

	}
}
```

Program using a while loop to count the number of students who passed a class. Outputs a message if 6 or more have passed.

```
/**
 * @author Subrina
 *
 */
public class Analysis
{

	/**
	 * @param args
	 */
	public static void main(String[] args) throws IOException
	{
		// initializing variables in declarations
		int passes = 0, failures = 0, student = 1, result;

		// process 10 students; counter-controlled loop
		while (student <= 10)
		{
			System.out.print("Enter result (1 = pass, 2 = fail): ");
			System.out.flush();
			result = System.in.read();

			if (result == '1') // if/else nested in while
				passes = passes + 1;
			else
				failures = failures + 1;

			student = student + 1;
			System.in.skip(1); // Use skip(2) on Window Java system
		}

		System.out.println("Passed " + passes);
		System.out.println("Failed " + failures);

		if (passes > 5)
			System.out.println("Most People Passed!! ");
	} // main()
} // Analysis
```

Basic Java Syntax

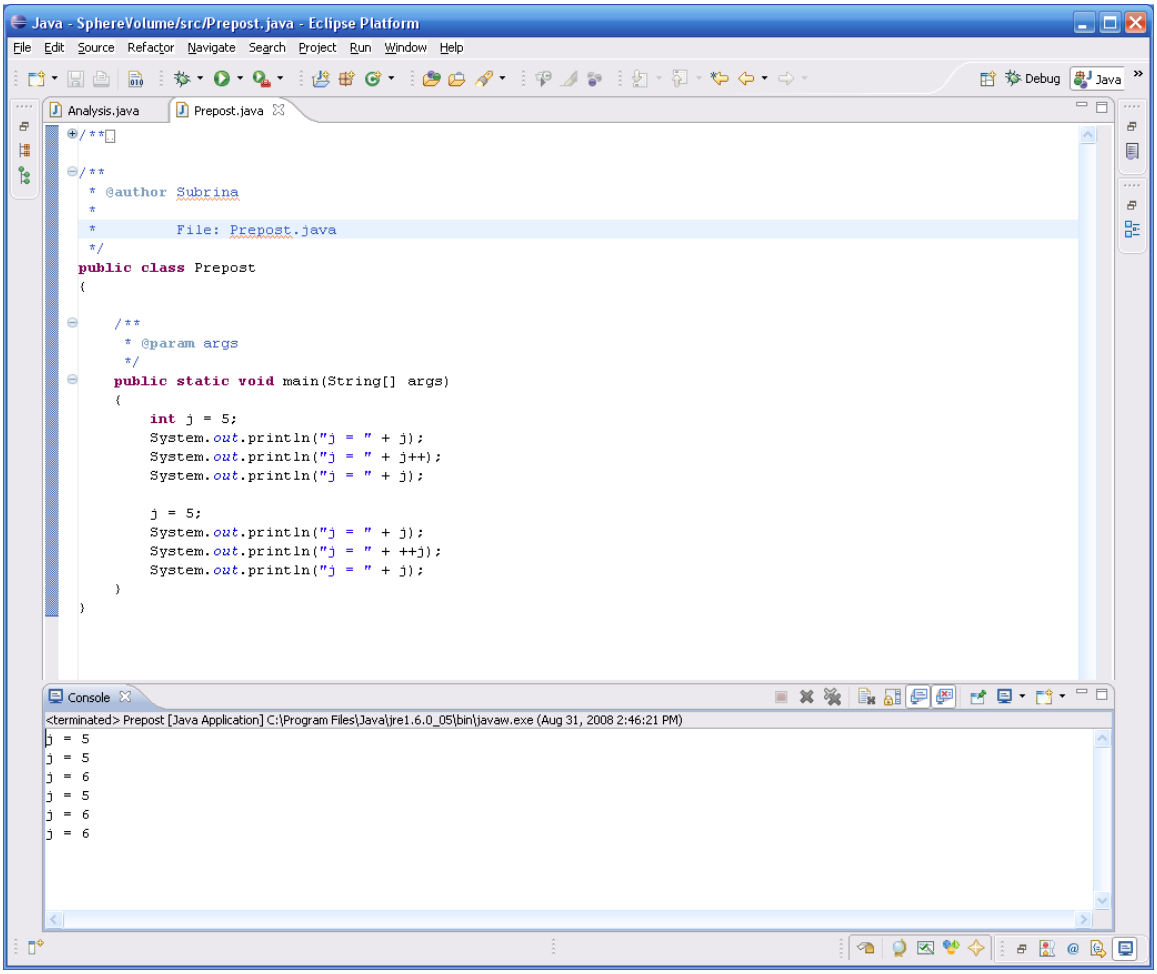
Assignment Operators

The *simple assignment* operator is **=**

The *compound assignment* operators are **+=** **-=** ***=** **/=** **%=**

Increment / Decrement Operators

This Java application demonstrates pre and post increment operators. The next page runs a similar program with an Applet



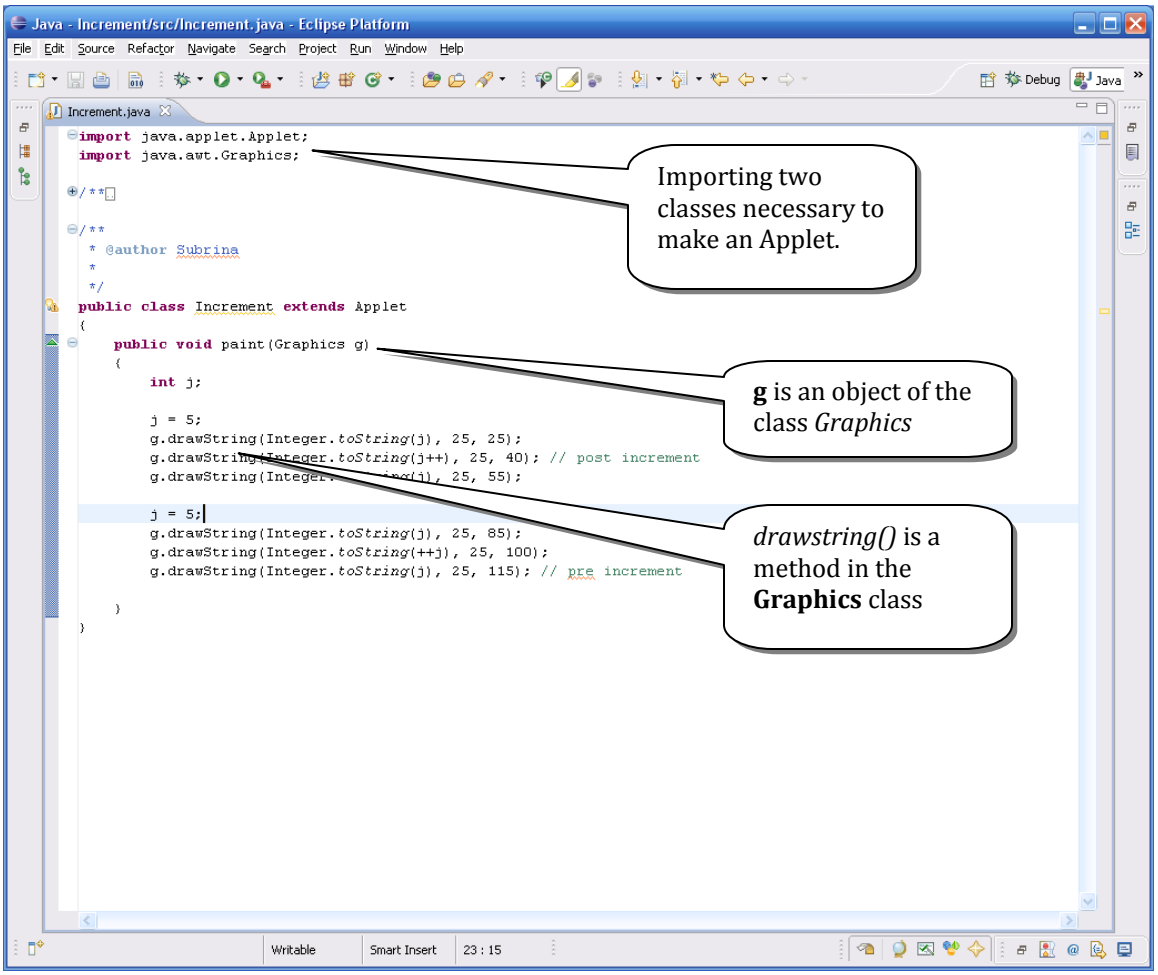
Pre / Post *increment* (*decrement*) operators

The *post-increment* uses the current value as the evaluation, then increments the expression by one

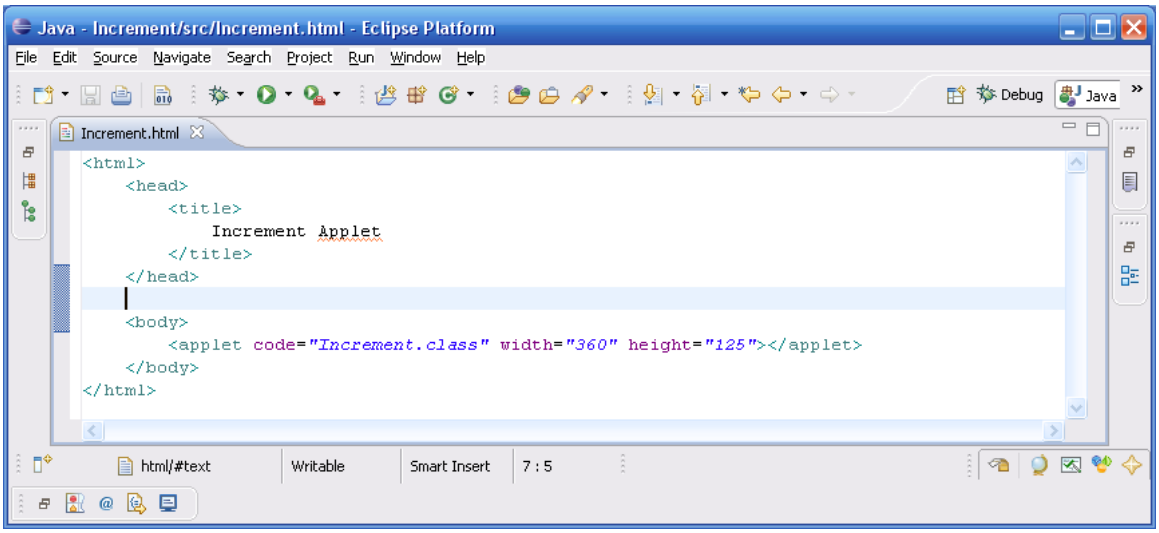
The *pre-increment* increments immediately, then uses the *new* value in the evaluation

Example of the same code using Java Applet

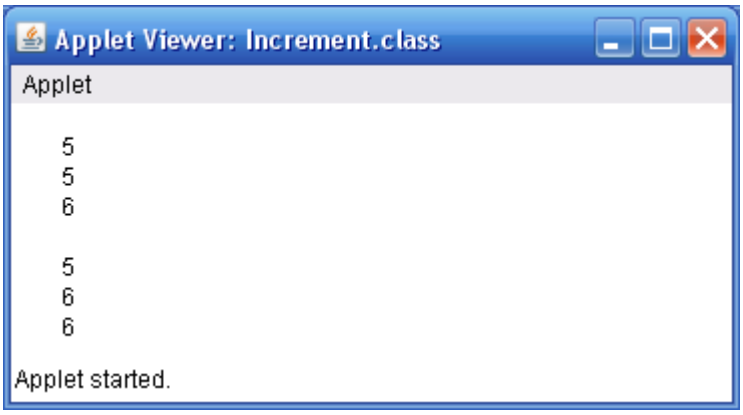
Increment.java



Increment.html



Output




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

/**
 *
 */
/**
 * @author Subrina
 *
 */
public class Increment extends Applet
{
        public void paint(Graphics g)
        {
                int j;

                j = 5;
                g.drawString(Integer.toString(j), 25, 25);
                g.drawString(Integer.toString(j++), 25, 40); // post increment
                g.drawString(Integer.toString(j), 25, 55);

                j = 5;
                g.drawString(Integer.toString(j), 25, 85);
                g.drawString(Integer.toString(++j), 25, 100);
                g.drawString(Integer.toString(j), 25, 115); // pre increment

        }
}
```

Integer is a class

toString() is a static method in the **Integer** class

Basic Java Operators

Operator	Associativity
()	L → R
++ -- + - (<type>)	R → L
/ % *	L → R
+-	L → R
< <= > >=	L → R
== !=	L → R
?:	R → L
= += -= *= /= %=	R → L

Java has built-in types (similar to C) as well as classes. Examples of built-in types are given below. All Java systems must have the exact match on the format (Unicode char and IEEE 754 floating point). The **dominating** types are shown lower:

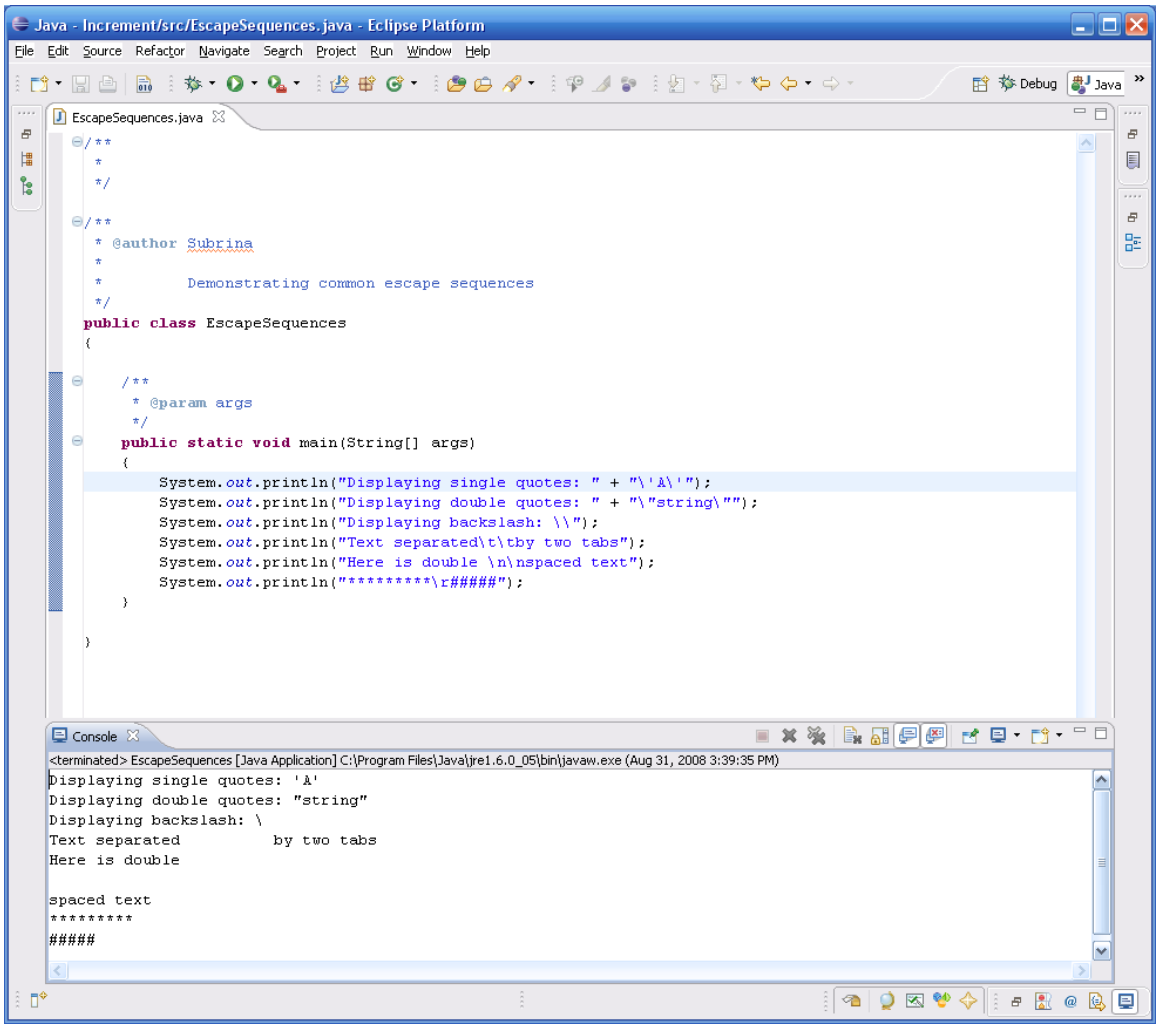
boolean	1 bit (values are *true* or *false*)
byte	1 byte
char	2 byte **ISO Unicode** character set
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes

Escape Sequences

In Java, Strings (a predefined class) can be embedded with escape sequence characters.

\n	**\t**	**\r**	****	**\'**	**\"**	**\0**	**\v**	**\u**xx	**\x**hh

Examples of the codes above in a program:



Counter Loops require the following sections:

1. Initialization of the counter (sometimes called priming)
2. Testing of the condition *
3. Loop body *

The counter is usually *incremented or decremented* in either of these sections so that the loop will tend toward the condition to terminate the loop.

Two examples of counter-controlled loops (*while* and *for*)

while loop example

```
/**
 * @author Subrina
 *
 * Counter-controlled repetition
 */

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

public class WhileCounter extends Applet
{
        public void paint(Graphics g)
        {
                int counter = 1; // initialization
                int yPos = 25;

                while (counter <= 10)
                { // repetition condition
                        g.drawString(Integer.toString(counter), 25, yPos);
                        ++counter; // increment
                        yPos += 15;
                }
        }
}
```

Counter initialization

Loop condition

Counter increment

for loop example

```
/**
 * @author Subrina
 *
 * Counter-controlled repetition with the for structure
 */

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

public class ForCounter extends Applet
{
        public void paint(Graphics g)
        {
                int yPos = 25;

                // Initialization, repetition condition, and incrementing are all
                // included in the for
                // structure header.
                for (int counter = 1; counter <= 10; counter++)
                {
                        g.drawString(Integer.toString(counter), 25, yPos);
                        yPos += 15;
                }
        }
}
```

Initialization, condition, and increment are all done inside the control of the *for* loop

Conditional Logic using the *switch* statement

→ Simple example of the *switch* statement

```
switch (kidCount)
{
        case 0:
            System.out.println("Enjoy your freedom");
            break;
        case 1:
            System.out.println("Welcome to parenthood");
            break;
        case 2:
            System.out.println("Having fun yet?");
            break;
        default:
            System.out.println("Need a second job?");
}
```

```
@Override
public void actionPerformed(ActionEvent e)
{
        String val = e.getActionCommand();
        char grade = val.charAt(0);

        showStatus("");        // clear status bar area
        input.setText("");     // clear input text field

        switch (grade)
        {
                case 'A': case 'a':    // Grade was uppercase A
                    ++aCount;          // or lowercase a.
                    break;

                case 'B': case 'b':    // Grade was uppercase B
                    ++bCount;          // or lowercase b.
                    break;

                case 'C': case 'c':    // Grade was uppercase C
                    ++cCount;          // or lowercase c.
                    break;

                case 'D': case 'd':    // Grade was uppercase D
                    ++dCount;          // or lowercase d.
                    break;

                case 'F': case 'f':    // Grade was uppercase F
                    ++fCount;          // or lowercase f.
                    break;

                default:
                    showStatus("Incorrect grade. Enter new grade: ");
                    break;
                }
                repaint(); // display summary
        }
```

Integer expression

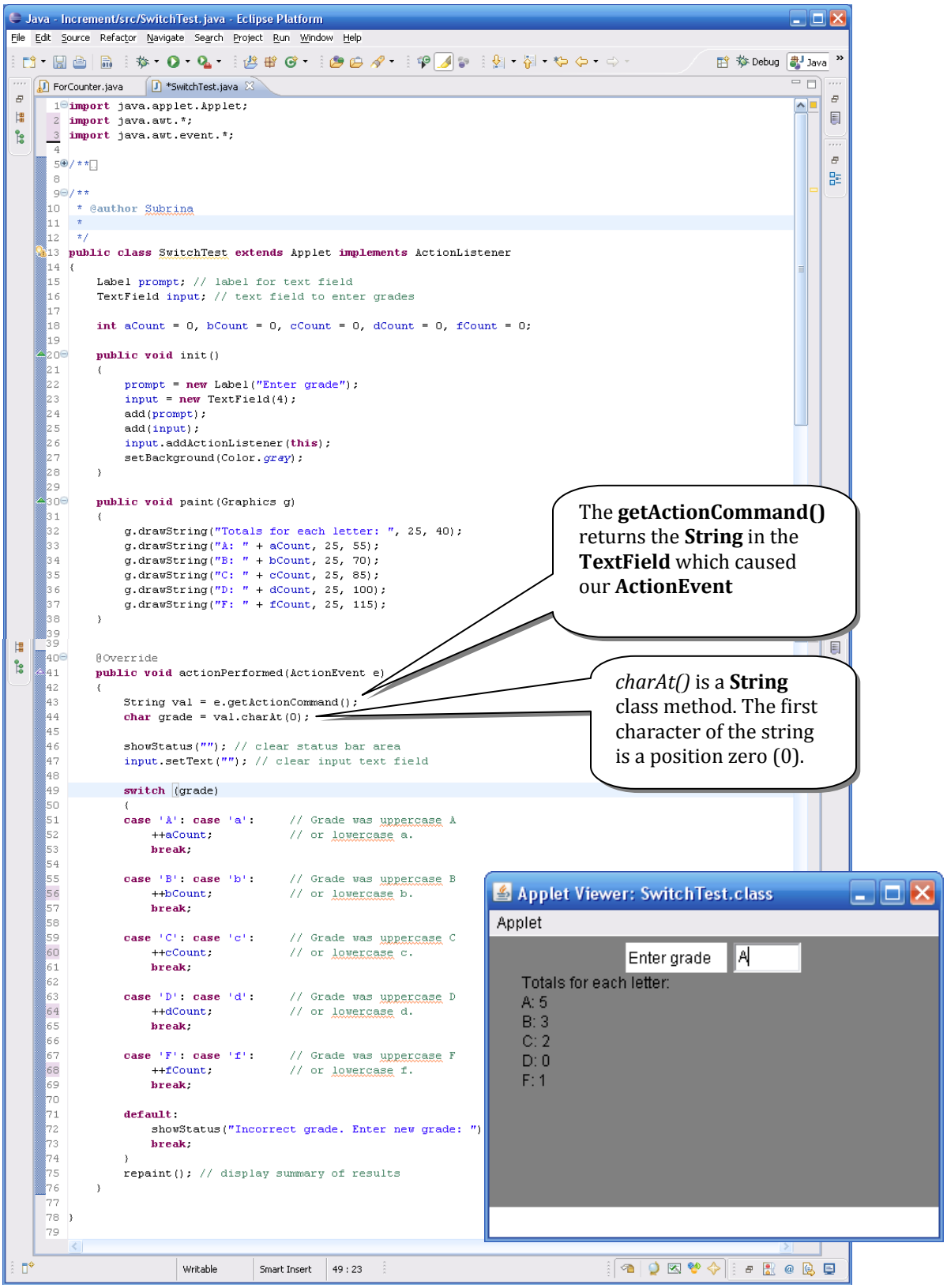
Actually capturing *two* cases

On matching case 'C' or 'c', we add one to the cCount variable, then *break* out of the *switch* construct

Default action if none of the cases match

showStatus() method outputs the string at the bottom of the **Applet**

The Entire **SwitchTest** program



Discussion Points on program SwitchTest

Various classes are introduced here. These are used in creating a nice GUI environment. Classes included here are

Label – a class which creates a write-only text label

TextField – a class which has a field which can be used as an input from the user into the application

Whenever we use a predefined class, we need to be aware of its behavior, that is its data and methods. Only through its methods and public interface can we manipulate the actual object

Note:
To create new **Label** and **TextField** objects, we use the code

Label prompt

TextField input;

This code defines two object *references*. *prompt* is a **Label** object, while *input* is a **TextField** object

prompt = new **Label**("Enter grade");
input = new **TextField**(2);

In this code we use the newly defined reference to create actual objects of their respective types. We are actually calling a *constructor* method for the appropriate class

Discussion Points on program SwitchTest (continued)

To make the objects (TextField input, and Label prompt) visible in the Applet, we must call the method *add( )* to add them. The following code was used.

```
add( prompt );
add( input );
```

Question: *Of which class do you think that the add method is a member*?

Answer: _____

A number of methods are in this program which will be used in future programs

init() – init( ) is called once when the Applet is about to begin its execution. Any preparatory initializations must be done here. Variables are initialized, objects can be constructed, files are opened, and the GUI is created. Also, it is called automatically once.

paint() – paint( ) is called after init( ), to create the graphic rendition of the Applet. Paint is called automatically once, but can be *re*called to regenerate the new look to the Applet. paint( ) is passed a Graphics object, which is the Graphics context for the applet. Basically, drawing using the corresponding Graphics object draws graphics on your Applet.

Graphics are *not* text. Therefore the **println**( ) method is not useful. Instead, a Graphics class method, **drawString**() can be used to output a string onto the Applet. Method *drawString*( ) takes a string and *x* and *y* pixel coordinates as parameters.

Discussion Points on program SwitchTest (continued)

actionPerformed() – *actionPerformed*( ) is called automatically when there is an *action event* that needs be handled. In this program, you can type in whatever you want in the **TextField** object. However, when you press <CR> (Enter) that triggers and *event*. This triggers an automatic call to the *actionPerformed ( )* method which has to deal with the event. The calling method passes to *actionPerformed ( )* the **ActionEvent** which was the target of the event. Since TextField event is the only type of event that can occur in this program, we assume that a TextField object (*input*, in our case) was the target.

We create a **String** object *val*, and assign into it the *string* value of the event target object. A **String** class handles strings (sequences of characters). Class **String** has its own methods. One method is called *charAt*( ). Given a **String** object, the *charAt*( ) method returns the character at the position of the parameter passed to it. Since in our program the user will enter a single character into the **TextField** imput, we access the character at position 0 (zero), i.e. the first character.

repaint() – The method *repaint( )* is called from method *paint()*. What *repaint*( ) does is basically to redraw the Applet's graphics content with the new program values. In this case, each time *actionPerformed( )* is called the count of A,B,C,D, or F is increased, or the "Incorrect grade…" message is displayed.

Note: This program is an example of an **event-driven** program. That means that the method which is called is based on the event that is triggered from the GUI interface. Such events can be **TextField**, **Button**, **Choice**, etc… Often, we have to discriminate as to which GUI component triggered the call to *actionPerformed*().

Basic OO Terms

Class - An object type description specifying the instance methods and data, as well as class methods and data

note: instance variables, member variables, properties all refer to the same
note: methods, and member methods refer to the same

Derived class - A class which is defined in terms of another class. As such, it inherits all the propertied and methods of its parent class.

note: subclass, child class refer to the same

Base class - A class from which other classes are derived.

note: parent class, superclass refer to the same

Object Reference - This is where we store the actual location of the object (once it has been instantiated)

note: objects are *not* created immediately, they must be created with the *new* operator
note: two objects may be equivalent objects, *or*, in fact, be the same object

Object - Once an object reference has been defined, we can instantiate the object. Instantiation causes memory to be allocated for the data and methods

Basics of Java Applications

Comments - There are 3 kinds of comments in Java:

Multiple line comment

```
/*      werwerwer
        Erterterterter
        dfgdfgdfgdfg
*/
```

Single line comment

```
//      This is a single line comment
/*      This is also a single line comment    */
```

Doc comment

```
/**     dfgdfgdfgdfg
        Dfgdfgdfgdfgdfgdfg
        dfgdfgdfgdfgdfg
*/
```

The *doc* comment will be extracted by the tool *javadoc* as an aid in documentation preparation. It creates html documentation. See next page for the html *tree.html* file

Defining a variable

```
byte smallval;
int i;
boolean amIDone;
char ch1,  ch2;
```

Defining and initializing a variable

```
int k = 6;
```

Java Application showing initialization and display of string variable

```
public class Junk1
{
        public static void main(String s[])
        {
                String str = "Professor";

                System.out.println("Hello " + str);
        }
}
```

Using class *boolean*

```
public class Junk1
{
        public static void main(String s[])
        {
                String str = " Professor ";
                boolean success = true;

                System.out.println("Hello " + str);
                System.out.println(success);
        }
}
```

➔ *Hello* Professor

true

Type Casting – As in C (C++), we can cast one type to e treated as another type.

Typically a compiler will warn if a *cast* operator is not used

```
int i = 7;
byte b;
…
…
b = (byte) i;
```

Casting the *int* to a *byte* type

Java Operator List (highest to lowest priority)

()	[]					
~	++	--	instanceof	() cast operator		
*	/	%				
+	-					
<<	>>	>>>				
<=	<	>=	>			
==	!=					
&						
^						
?:						
=	+=	-=	*=	/=	%=	&=
^=	|=	<<=	>>=	>>>=		

Sign propagate right

Zero fill in right shift

The conditional operator

Can be used to simplify the use of if-else constructs

```
int max, i = 6, j = 7;
max = i > j ? i : j;
```

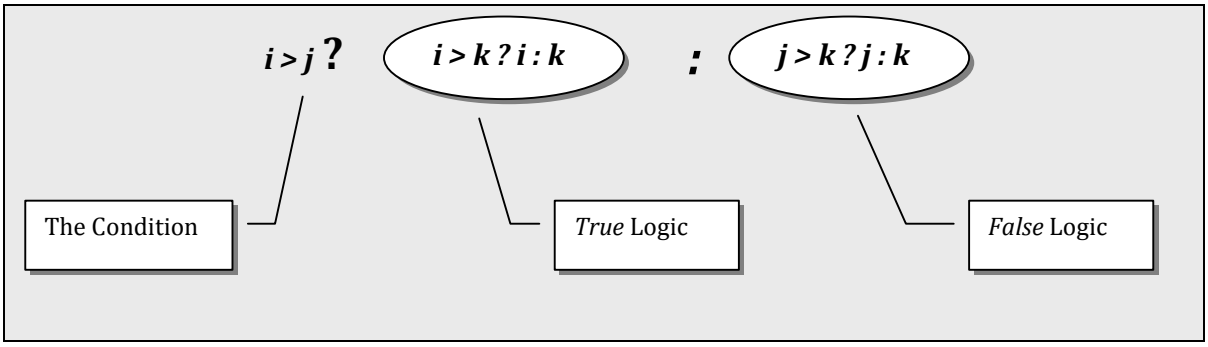
max is assigned the greater of *i* and *j.*

```
int max, i = 5, j = 4, k = 12;
max = i > j ? i > k ? i : k: j > k ? j : k
System.out.println("k = " + k);
```

max assigned the maximum of three int values

Alternatively, we can write:

```
System.out.println("k = " + (i > j ? i > k ? i : k : j > k ? j : k));
```



Example of Using a Nested Conditional Operator

Java Program example of persistent while loop with switch statement

Java - Increment/src/Junk2.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run Window Help

*Junk2.java

```
1  import java.io.IOException;
2
3  /**
6
7  /**
8   * @author Subrina
9   *
10  */
11 public class Junk2
12 {
13     public static void main(String s[]) throws IOException
14     {
15         char ch;
16         boolean done = false;
17
18         while (!done)
19         {
20             System.out.print("Please enter A, B, or C Only: ");
21             ch = (char) System.in.read();
22             switch (ch)
23             {
24             case 'a': case 'A':
25                 System.out.println("You typed in 'A'");
26                 done = true;
27                 break;
28             case 'b': case 'B':
29                 System.out.println("You typed in 'B'");
30                 done = true;
31                 break;
32             case 'c': case 'C':
33                 System.out.println("You typed in 'C'");
34                 done = true;
35                 break;
36
37             default:
38                 System.out.print("Incorrect Input. Again, ");
39                 System.in.skip(2);
40             } // end of switch
41         } // end of while
42     } // end of main()
43 } // end of class
```

Example of *declare-or-handle* code for possible IOException

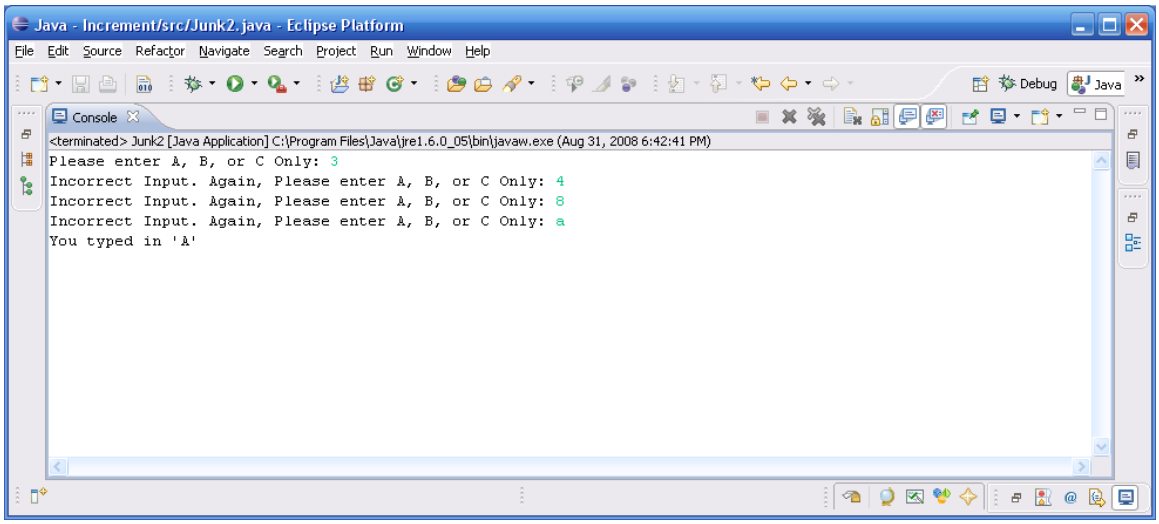
cast example on standard input stream I/O

Writable | Smart Insert | 10 : 4

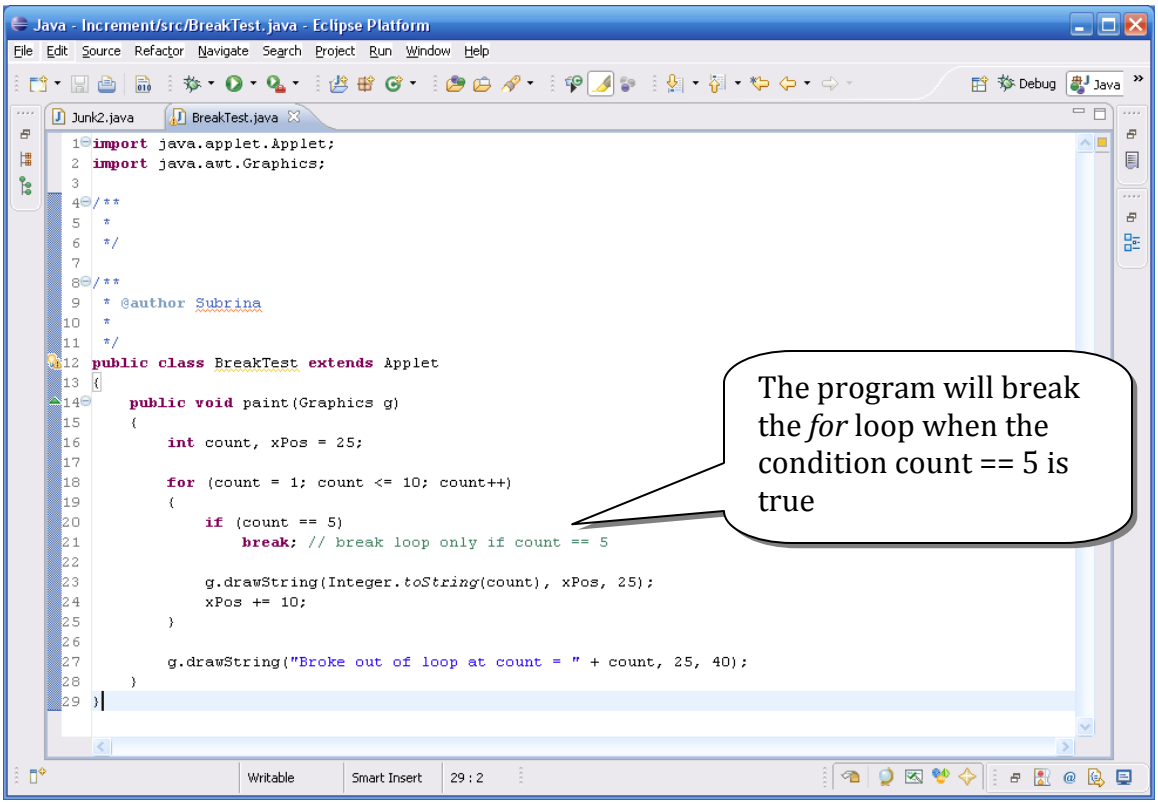
Junk2.java program

```
import java.io.IOException;

/**
 *
 */
/**
 * @author Subrina
 *
 */
public class Junk2
{
        public static void main(String s[]) throws IOException
        {
                char ch;
                boolean done = false;

                while (!done)
                {
                        System.out.print("Please enter A, B, or C Only: ");
                        ch = (char) System.in.read();
                        switch (ch)
                        {
                        case 'a': case 'A':
                                System.out.println("You typed in 'A'");
                                done = true;
                                break;
                        case 'b': case 'B':
                                System.out.println("You typed in 'B'");
                                done = true;
                                break;
                        case 'c': case 'C':
                                System.out.println("You typed in 'C'");
                                done = true;
                                break;

                        default:
                                System.out.print("Incorrect Input. Again, ");
                                System.in.skip(2);
                        } // end of switch
                } // end of while
        } // end of main()
} // end of class
```

Output of the Junk2 Java Application

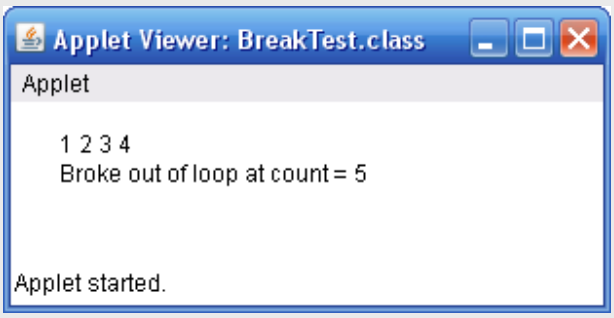


Break statement – the *break* statement takes control out of the loop, such as *for*, *do-while*, and *while*. It also sends control out of the *switch* statement.

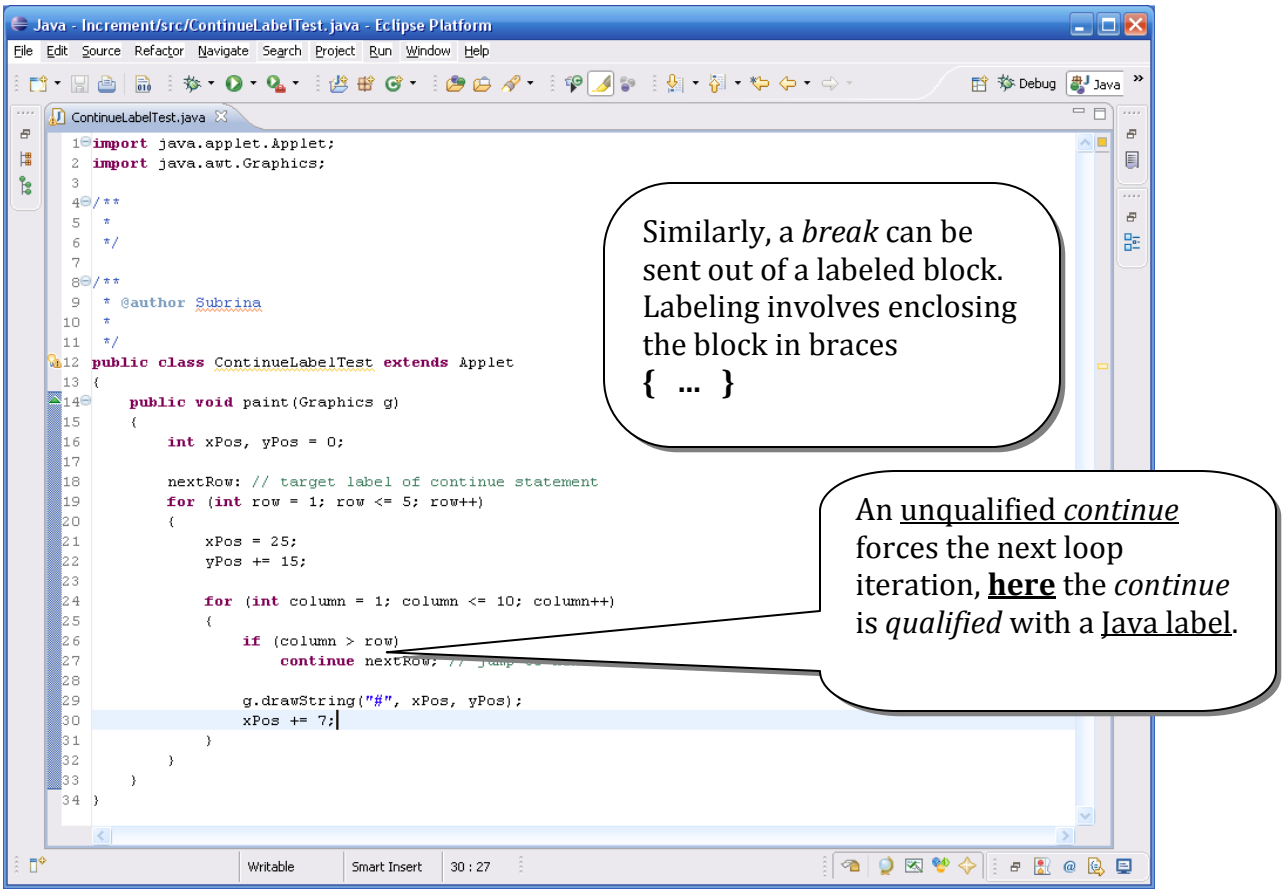


BreakTest.java program

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

/**
 *
 */

/**
 * @author Subrina
 *
 */
public class BreakTest extends Applet
{
        public void paint(Graphics g)
        {
                int count, xPos = 25;

                for (count = 1; count <= 10; count++)
                {
                        if (count == 5)
                                break; // break loop only if count == 5

                        g.drawString(Integer.toString(count), xPos, 25);
                        xPos += 10;
                }

                g.drawString("Broke out of loop at count = " + count, 25, 40);
        }
}
```



The *continue* statement forces the next loop iteration to occur (useful in as *for*, *do-while*, and *while* loops). It is a way to short-circuit the *body* of the loop.




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

/**
 * @author Subrina
 *
 */
public class ContinueLabelTest extends Applet
{
	public void paint(Graphics g)
	{
		int xPos, yPos = 0;

		nextRow: // target label of continue statement
		for (int row = 1; row <= 5; row++)
		{
			xPos = 25;
			yPos += 15;

			for (int column = 1; column <= 10; column++)
			{
				if (column > row)
					continue nextRow; // jump to nextRow label

				g.drawString("#", xPos, yPos);
				xPos += 7;
			}
		}
	}
}
```

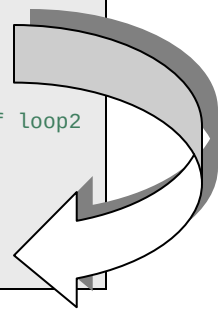
Main points on the *break and continue* commands:

break

This is as close to a *goto* as Java supports. The construct is structured because it is not a jump to an arbitrary location; rather it is a jump to the statement past the labeled block.

example:

```
myLabel:
{

while (condition1) {
        <beginning of loop1 body>
        ……
        while ( <condition2> )  {
        < beginning of loop2 body>
                ……
                if  (<condition>)  break myLabel;
                ……
                <end of loop2 body>  }  // end of loop2
        ……
        <end of loop1 body>   }  // end loop1

}  // end of labeled block

<next statement after the labeled break>
```



continue

The labeled *continue* statement needs sends control to a label which is written just before the control of a loop.

- ***note*:** if the label is improperly placed, the Java compiler will not find it. This disallows sending control to an arbitrary location

Logical operators

Java has typical **AND** *and* **OR** Logical and Bitwise operators as well as.
There is also an **XOR** operator.

There are two **AND** operators and two **OR** operators

1. AND
   - && Logical AND, uses short-circuit evaluation
   - & Bitwise AND
2. OR
   - || Logical OR, uses short-circuit evaluation
   - | Bitwise OR
3. XOR
   - ^ Bitwise XOR

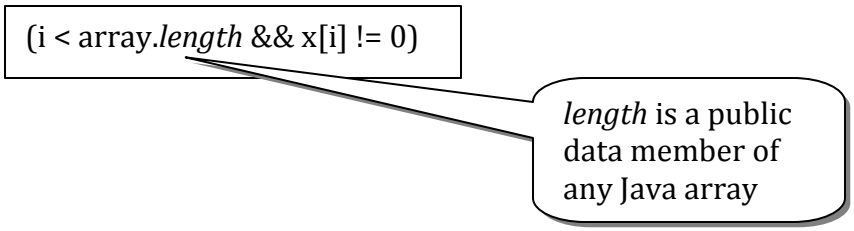
Precedence order is:
! & ^ | && ||

Logical Examples

boolean c1 = true, c2 = false, c3 = true;

(c1 && c3)	➔ true
(c1 & c3 && !c2)	➔ true
(c1 ^ c2 ^ c3)	➔ false
(!c3 | !c2)	➔ true
(!(!c3 | !c2))	➔ false *(negation of above)*

The following could cause an Exception <u>if the second condition were tested</u>!



Note:

Java will not test the second condition of a **Logical AND** if the first condition is *false*

De Morgan's Laws (similar for the OR)

! (<c1> && <c2>)

is same as

! <c1> || ! <c2>

Java API – the Java Application Program Interface provides classes and methods for peforming a variety of needed tasks such as

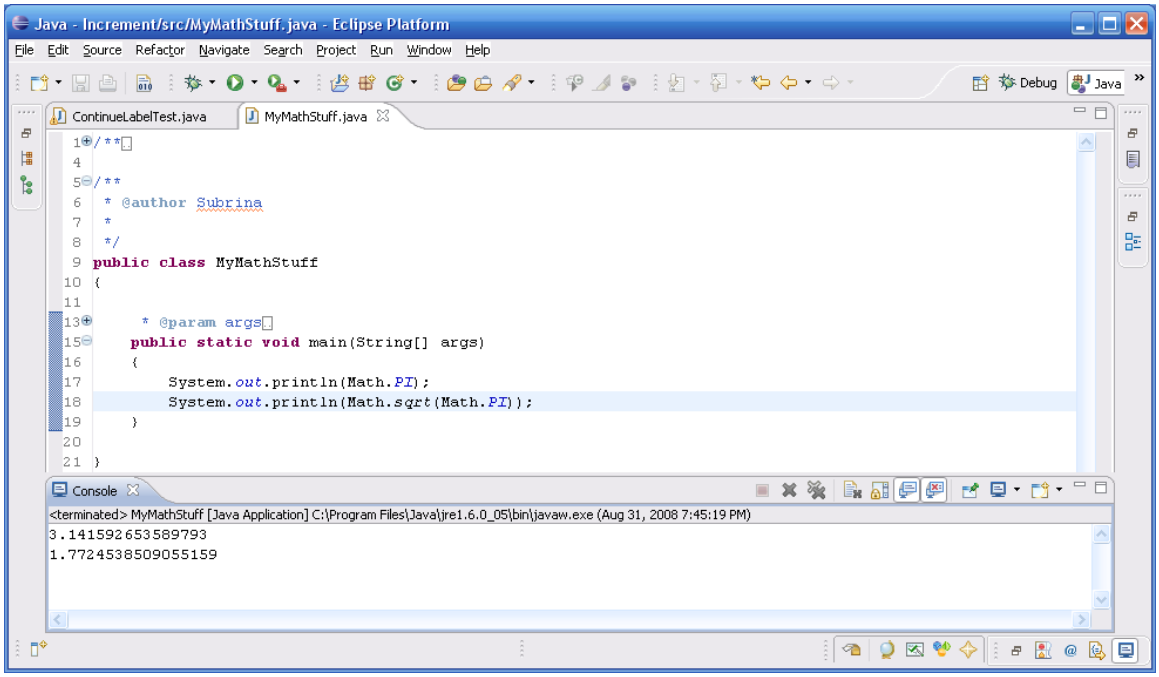
string mainpulations
character mainpulation
I/O
Date handling
error checking
mathematical calculations and constants
graphics
etc ...

Java API packages

java.applet	Applet class and other interfaces
java.awt	Abstract Windows Toolkit used for GUI management
javax.swing.*	New Java 1.2 GUI Support with LAF Control
java.awt.image	Store and minipulate images
java.awt.peer	Allows platform-specific interaction
java.io	Provides files and streams
java.sql	Allows database interaction through Java
java.lang	Supplies basic classes and interfaces, automatically brought in
java.net	Allows communication using socket-based networking
java.util	Provides utilities such as Date class, randomization, and String Tokenizer

Example: The **Math** class

The **Math** class provides constants and methods



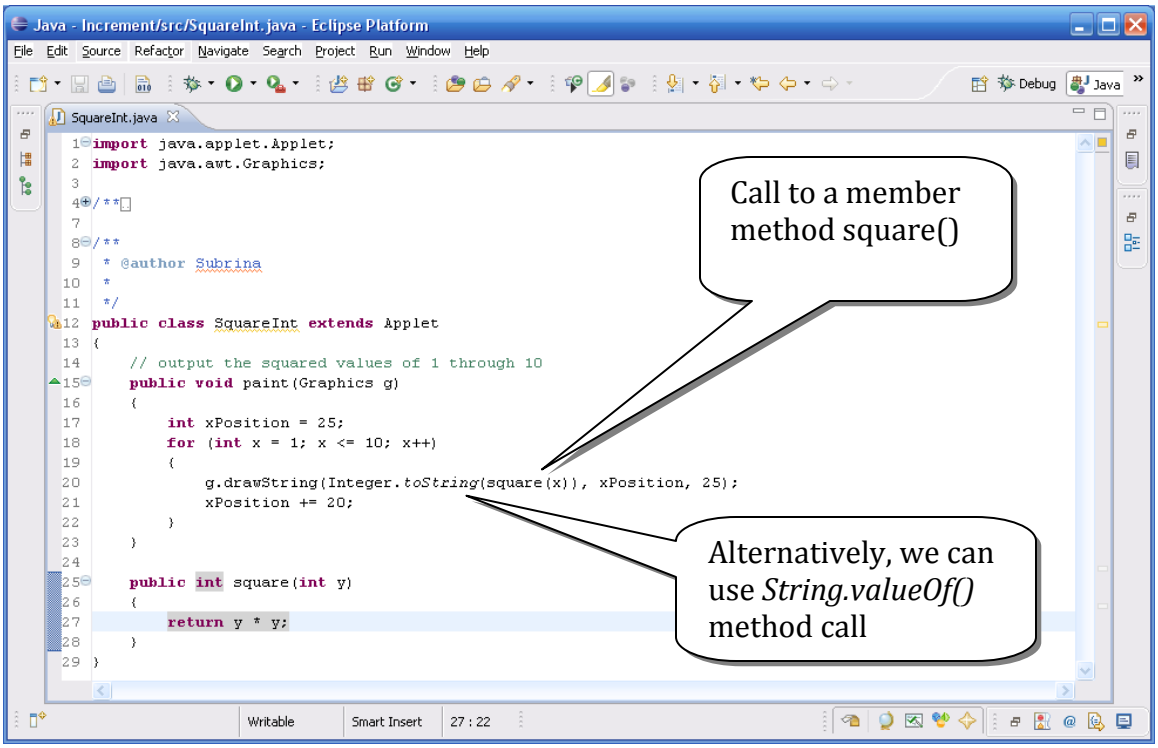
```java
/**
 * @author Subrina
 *
 */
public class MyMathStuff
{

	/**
	 * @param args
	 */
	public static void main(String[] args)
	{
		System.out.println(Math.PI);
		System.out.println(Math.sqrt(Math.PI));
	}

}
```

Other **Math** class methods include:

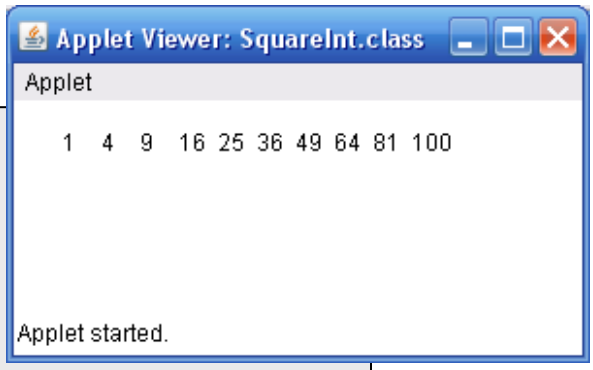
abs()
ceil()
cos()
exp()
floor()
log()
max()
min()
pow()
sqrt()
tan()

All calls must preface the method by its class name, **Math**

Methods can also be dfefined as a part of the class



SquareInt.java program




```
/**
 * @author Subrina
 *
 */
public class SquareInt extends Applet
{
        // output the squared values of 1 through 10
        public void paint(Graphics g)
        {
                int xPosition = 25;
                for (int x = 1; x <= 10; x++)
                {
                        g.drawString(Integer.toString(square(x)), xPosition, 25);
                        xPosition += 20;
                }
        }

        public int square(int y)
        {
                return y * y;
        }
}
```

Methods can call other methods by passing as paramameters, return values from other methods. The outer call is **bold**, the inner are in *italics*.

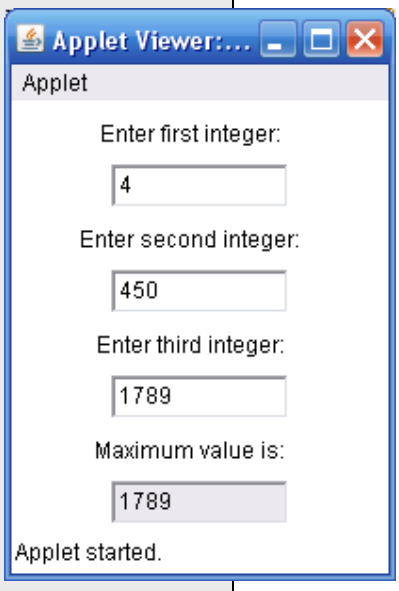
int val1 = 5, val2 = 6, val3 = 3, val4 = -3, max;

max = **Math.max** (*Math.max*(val1, val2), *Math.max*(val3, val4));

Program Example of Finding the Maximum of Three Numbers:

note: Method **maximum**() is called from **action**() to calculate result *max*

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/**
 * @author Subrina
 *
 */
public class Maximum extends Applet implements ActionListener
{
        Label label1, label2, label3, resultLabel;
        TextField number1, number2, number3, result;
        int num1, num2, num3, max;

        public void init()
        {
                label1 = new Label("Enter first integer:");
                number1 = new TextField("0", 10);
                label2 = new Label("Enter second integer:");
                number2 = new TextField("0", 10);
                label3 = new Label("Enter third integer:");
                number3 = new TextField("0", 10);
                resultLabel = new Label("Maximum value is:");
                result = new TextField("0", 10);
                result.setEditable(false);

                add(label1);
                add(number1);
                add(label2);
                add(number2);
                add(label3);
                add(number3);
                add(resultLabel);
                add(result);
                number2.addActionListener(this);
        }

        public int maximum(int x, int y, int z)
        {
                return Math.max(x, Math.max(y, z));
        }

        public void actionPerformed(ActionEvent e)
        {
                num1 = Integer.parseInt(number1.getText());
                num2 = Integer.parseInt(number2.getText());
                num3 = Integer.parseInt(number3.getText());
                max = maximum(num1, num2, num3);
                result.setText(Integer.toString(max));
        }
}
```



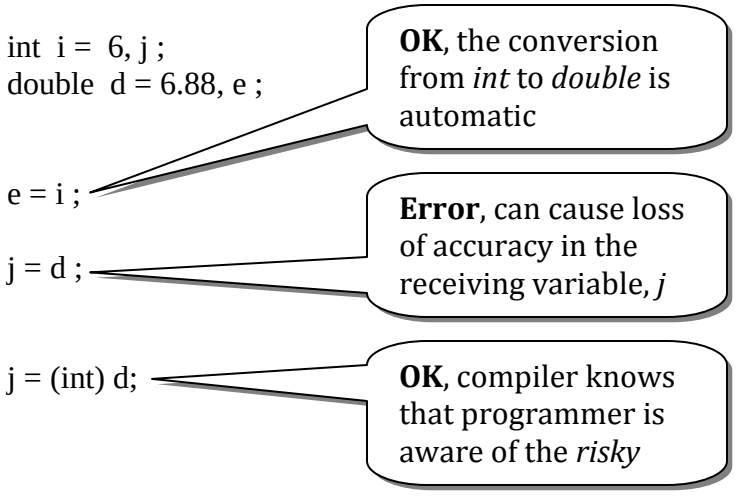
Casting in Java

The *dominating types* in high-to-low order in Java are as follows

double
float
long
int
char
short
byte
boolean

Assignment to an *inferior* type, requires an explicit *cast*

examples:



Example of the *random( )* method

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

/**
 *
 */

/**
 * @author Subrina
 *
 */
public class RandomInt extends Applet
{
        public void paint(Graphics g)
        {
                int xPosition = 25;
                int yPosition = 25;
                int value;

                for (int i = 1; i <= 20; i++)
                {
                        value = 1 + (int) (Math.random() * 6);

                        g.drawString(Integer.toString(value), xPosition, yPosition);

                        if (i % 5 != 0)
                                xPosition += 40;
                        else
                        {
                                xPosition = 25;
                                yPosition += 15;
                        }
                }
        }
}
```

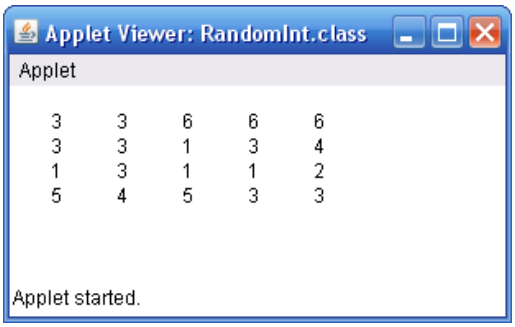
Accessing the **Math** method **random**()

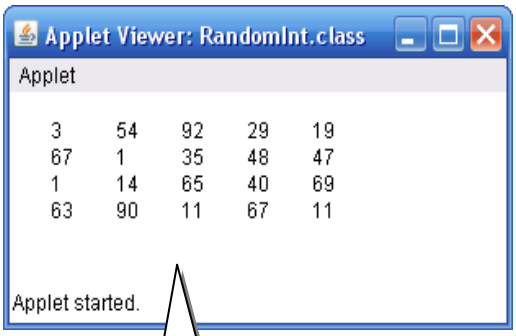
Scaling the result by a factor of 6

Is this *cast* needed?

Still in *same* row

Starting a *new* row, rest x-coordinate, and move y-coordinate down





What was changed here?

Basic properties of Variables and Methods (identifiers)

Lifetime – the period during which the identifier exists in memory

Scope – the places in which the identifier can be referenced in the program

Method Objects -

- These identifiers are created as we enter their creating block, and are destroyed as we exit the block. These are *local* to their **methods**.

- These are *not* initialized

examples: local objects in a method
formal parameters in a method

Instance and Static Objects -

- These are initialized by the compiler, given that the programmer has not given initial values
- Primitive numeric types are initialized to zero, boolean to *false*, and references to *null*

examples: class data member (static)
instance data member (non-static)

Scope rules

class scope – allows member methods to access other member methods, member data, as well all member methods and data inherited from superclass (es)

method and instance variables have *class* scope

static methods are an exception to the rule, as they cannot access non-*static* data members

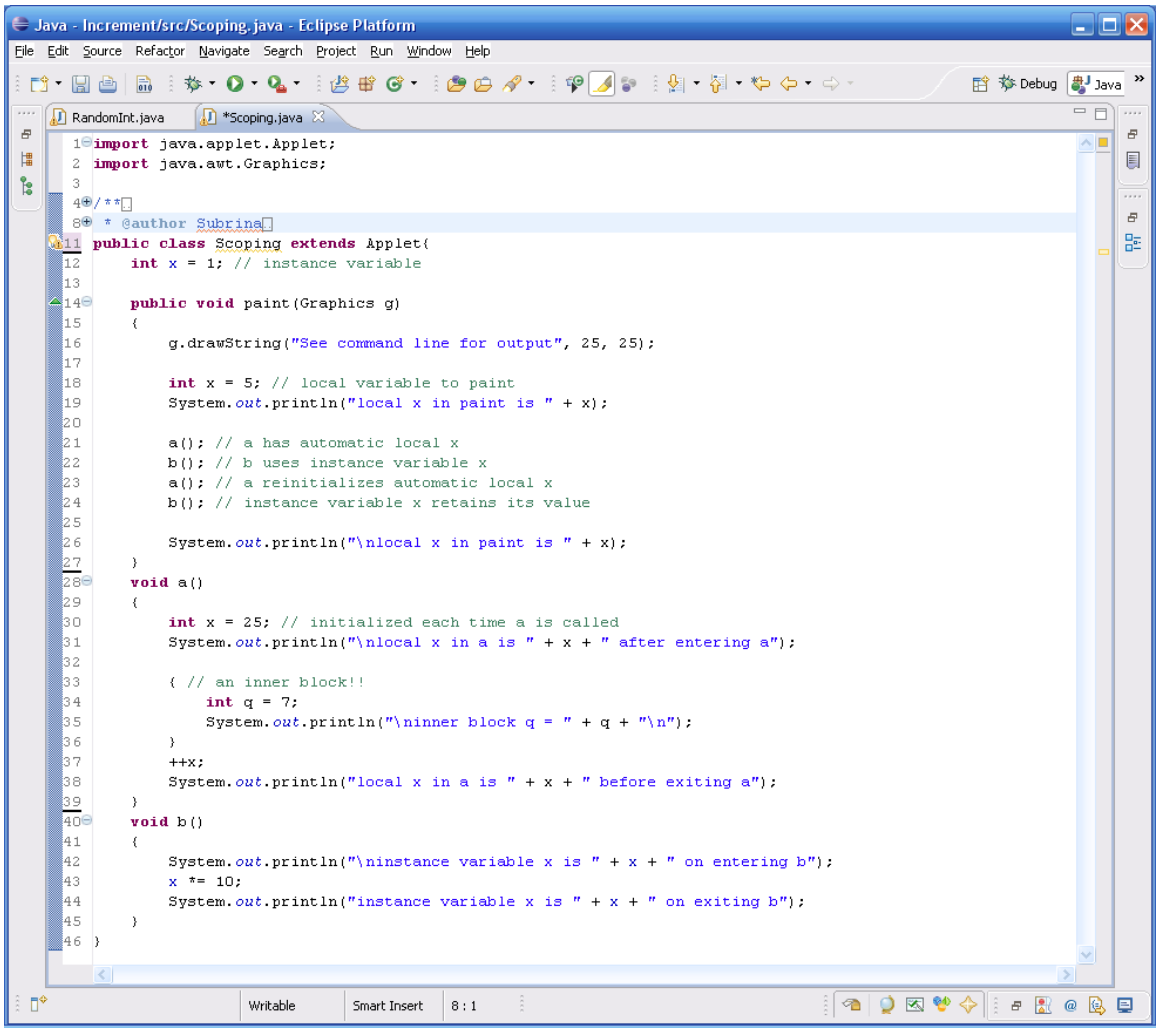
block scope – identifiers defined inside a block have *block* scope. This is typically a local method variable. Also nested blocks **are** permitted in Java. But, an inner block may not reuse an identifier name from an outer block

```
...
public void someMethod ( ) {
int num;
.....
.....
{
        int  val;
        int num;
}
.....
.....    }
```

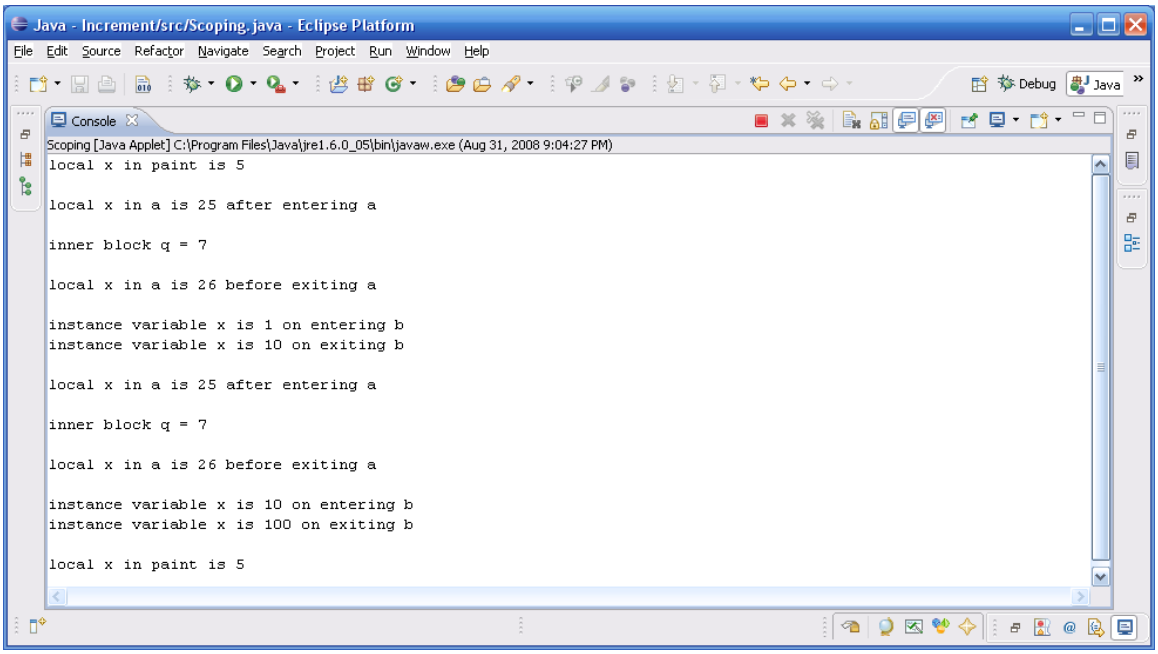
OK, *val* is a automatic variable with block scope

Error, *num* already exists in the block scope!

Program Example of Scope



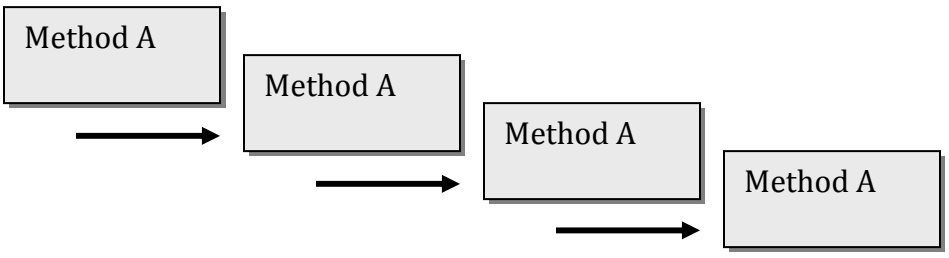
Output of Scoping program



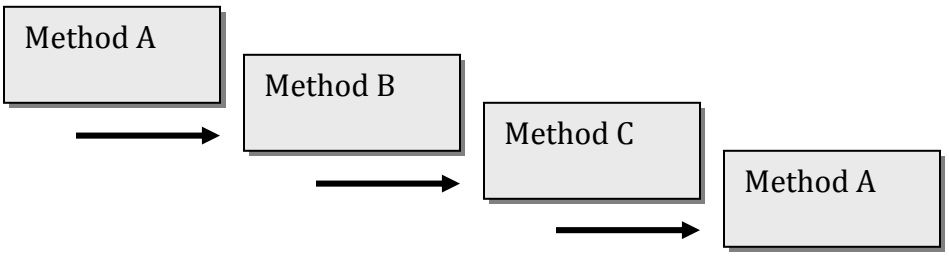
* Recursion

Recursion occurs when a method either calls *itself*, or calls *another* method such that in the chain of calls the **original method** gets called.

1. In the first case, *direct recursion*, the method is suspended and another version of the same method begins execution. This process may repeat itself many times leading to a deep recursion.
2. In the second, *indirect recursion*, the original method is suspended for another method until some method calls another version of the original method.



Direct recursion



Indirect recursion

Implementation of Recursion

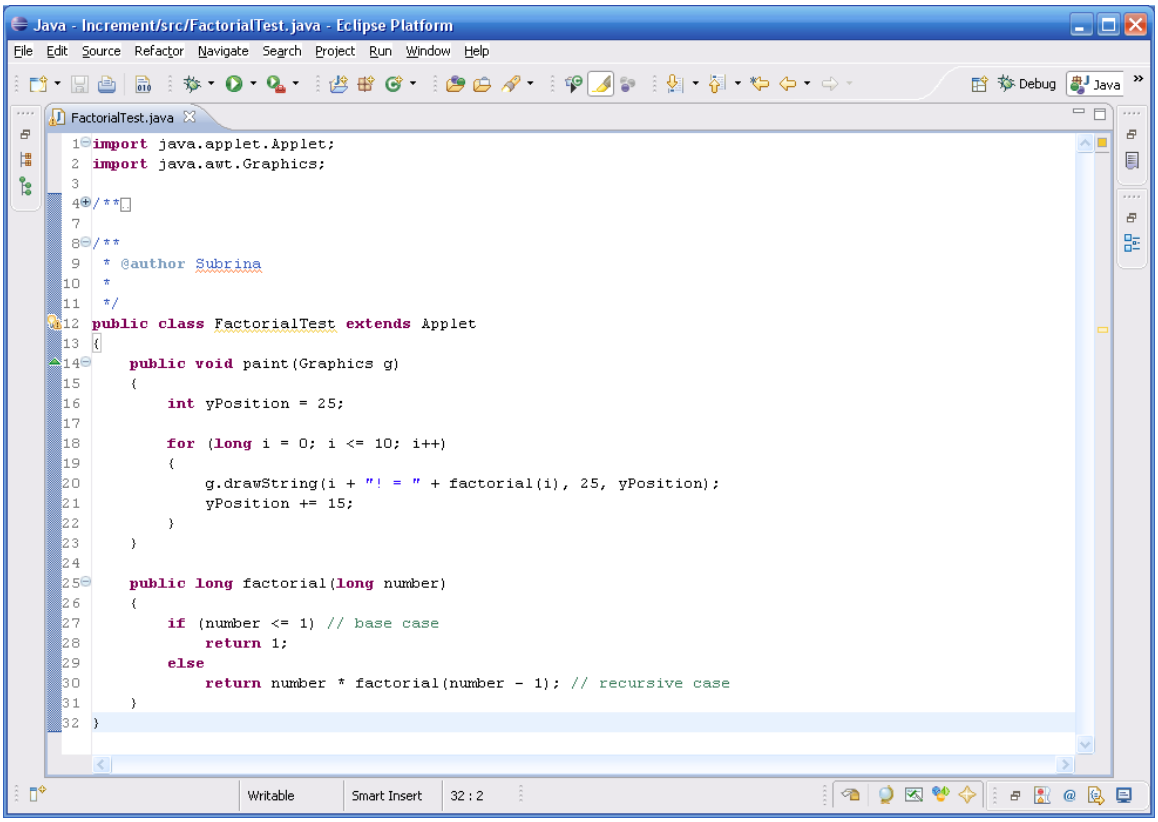
Each recursive method must have at least one *base case* and at least one *recursive case*.

The goal of recursion is to solve a *hard* problem by reducing it to a simpler problem of the same type.

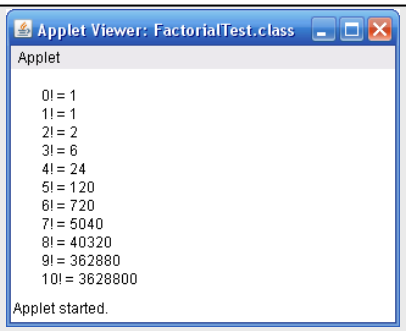
As the hard problem is posed, it is reduced in complexity by the recursive case such that it is tending to go to the base case

The base case is a trivial solution

A simple example: the *factorial* method

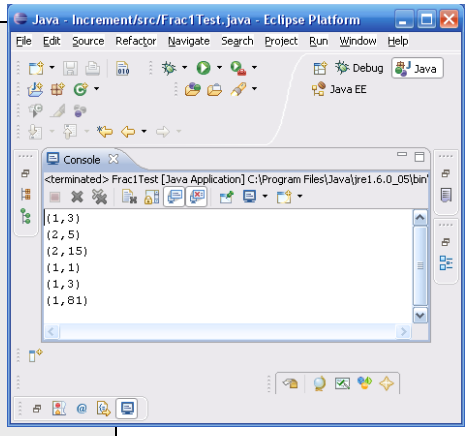



```
import java.applet.Applet;
import java.awt.Graphics;

/**
 * @author Subrina
 *
 */
public class FactorialTest extends Applet
{
        public void paint(Graphics g)
        {
                int yPosition = 25;

                for (long i = 0; i <= 10; i++)
                {
                        g.drawString(i + "! = " + factorial(i), 25, yPosition);
                        yPosition += 15;
                }
        }

        public long factorial(long number)
        {
                if (number <= 1) // base case
                        return 1;
                else
                        return number * factorial(number - 1); // recursive case
        }
}
```



Another Recursive example: *The Fraction Class*

```java
/**
 * @author Subrina
 *
 */
public class Frac1Test
{
	public static void main(String s[])
	{
		Fraction f = new Fraction();
		f.setValue(1, 3);
		f.display();

		Fraction g = new Fraction(), h;
		g.setValue(2, 5);
		g.display();

		h = g.mult(f);
		h.display();

		h = f.power(0); // utilizing the base case only
		h.display();

		h = f.power(1); // 1 recursive call
		h.display();

		h = f.power(4); // 4 recursive calls
		h.display();
	}
}
```




```java
/**
 * @author Subrina
 *
 */
public class Fraction
{
	private int numerator, denominator;

	boolean setValue(int x, int y)
	{
		boolean ok = y != 0; // set the return value to true or false
		numerator = x;
		denominator = (ok ? y : 1); // assign 1 if denominator is zero
		return ok;
	}

	void display()
	{
		System.out.println("(" + numerator + "," + denominator + ")");
	}

	Fraction mult(Fraction val)
	{
		Fraction temp = new Fraction();

		temp.numerator = numerator * val.numerator;
		temp.denominator = denominator * val.denominator;
		return temp;
	}

	Fraction power(int pow)
	{
		Fraction temp = new Fraction();
		temp.setValue(1, 1);

		if (pow == 0)
			return temp;
		else
			return this.mult(power(pow - 1));
	}
}
```

Method Overloading

Methods names can be reused in the same class as long as the method *signature* differs from another method of the same name.

The *signature* is the name of the method along with its number, order and types of parameters in its list

Method names are *mangled* to allow the compiler to differentiate between method calls bearing the same name.

The following methods can be defined in the same class:

```
public void  methodAbc (int x, int y, double d, Double D) { ...
public void methodAbc ( ) { ...
public double methodAbc ( int value, Object o) { ...
public double methodAbc( Fraction f ) { ...
public void methodAbc (int x, int y, Double d, Double D) { ...
```

Note: Methods *cannot* be distinguished by their return type!

Note: Overloaded methods *may* have different return types, but their parameter lists *must* differ

Applet Class methods

Examples of Applet class methods are *resize*() and *repaint*()

setSize() – this method allows the applet to alter its pixel width and height form the original specification in the HTML code APPLET tag

repaint() – this method passes the Graphics object to the **paint**() method. Before the call to **paint**(), there is a call to Applet method **update**() which erases the any drawing that was previously done on the applet, before calling **paint**() to redraw the screen.

Other important Applet class methods which are called automatically when particular events happen

public void init() – method is called once upon initial entry into the Applet code

public void start() – called after **init**(), and recalled every time the HTML page on which the applet resides is (re)visited. Typically animations and threads are started here

public void paint(Graphics g) – called after **init**() and after **start**() has begun its execution.. It is (re)called every time the applet has been out of view, or hidden by another frame or window

public void stop() – called automatically when the browser leaves the applet HTML page. This is a place to stop animations and suspend threads.

public void destroy() – called when the applet is being closed or terminated. Typically we return no longer needed resources as well as stop threads that may be in progress.

Arrays in Java

Array – a contiguous block of memory locations which are accessible under the same name and are accessed through the use of a subscript or index.

Arrays of primitive types are allocated memory to hold a particular number of elements via the *new* operator. Such an array is ready for use as soon as *new* has been called

Arrays of *class* elements are also allocated in the same manner, *but* the resulting reserved memory is big enough to hold references to the objects. The objects must also be allocated memory by using the *new* operator.

Arrays are accessed by an index which *always* begins at **zero.** Any *expression* reference to an invalid index triggers an ArrayIndexOutOfBoundsException. Any constant reference to an invalid index will be caught be the compiler.

Example array X, with 10 elements:

12	**3**	**5**	**-5**	**22**	**11**	**56**	**41**	**18**	**4**
X[0]	X[1]	X[2]	X[3]	...	...	...	...	...	X[8]	X[9]

X[3] ➔ -5	X[1] += X[9]; *adds 4 to X[1]*
X[X[1]] ➔ -5	X[X[3]]=5; *causes Exception*
X[5] + X[9] * X[3] ➔ -9	X[-5] *Compiler error*
X[7] / 2 ➔ 20	((X[2])++)++ Compiler error
X[8]++; *increases X[8] by one*	X[9] += ++ X[4] *ok*

Declaring and Allocating Java arrays

int myArray[ ];

This declares an array. It does not allocate any memory for the elements. You *cannot* specify the number of elements in the array at this time!

int myArray[25];

This is *illegal* code in Java.

Two ways of creating a usable array

One way:

```
int myArray[];
myArray = new int [25];
```

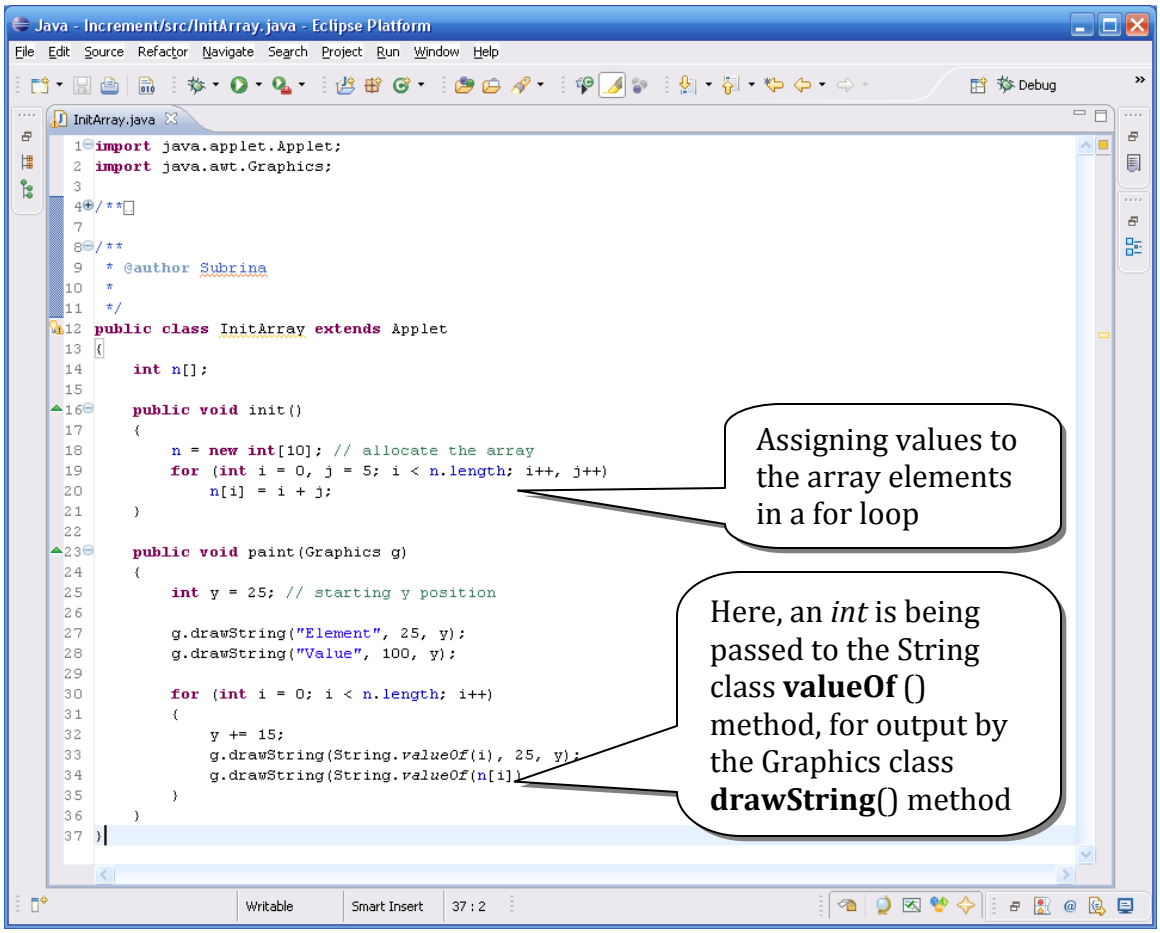
Another way:

```
int myArray[] = new int [25];
```

Note: Arrays are *always* initialized to the following values

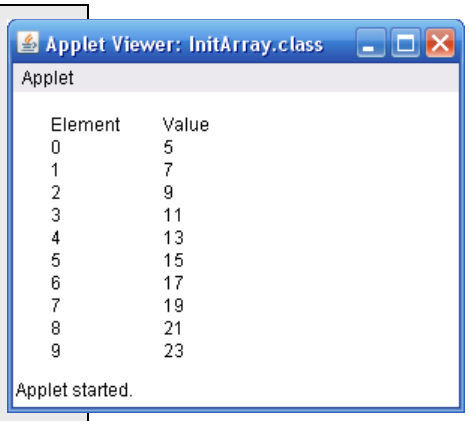
1. zero for any numeric primitive type
2. *false* for boolean type
3. **null** for object arrays

Example of a simple array usage



InitArray.java program Code

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

/**
 * @author Subrina
 *
 */
public class InitArray extends Applet
{
        int n[];

        public void init()
        {
                n = new int[10]; // allocate the array
                for (int i = 0, j = 5; i < n.length; i++, j++)
                        n[i] = i + j;
        }

        public void paint(Graphics g)
        {
                int y = 25; // starting y position

                g.drawString("Element", 25, y);
                g.drawString("Value", 100, y);

                for (int i = 0; i < n.length; i++)
                {
                        y += 15;
                        g.drawString(String.valueOf(i), 25, y);
                        g.drawString(String.valueOf(n[i]), 100, y);
                }
        }
}
```



Initialization of Java arrays

Arrays can be initialized as follows:

int X [ ] = { 3, 45, 1, 3, 2, 11, 35, 67, 12, 34, 11, 1, 8, -4, 3 };

Above, a 15-element int array has been declared, allocated, and initialized

Array Size

All Java arrays have a well-defined size which can be determined by accessing the expression **<array_name>.length**

Above, array X has a length which is equal to

X.length

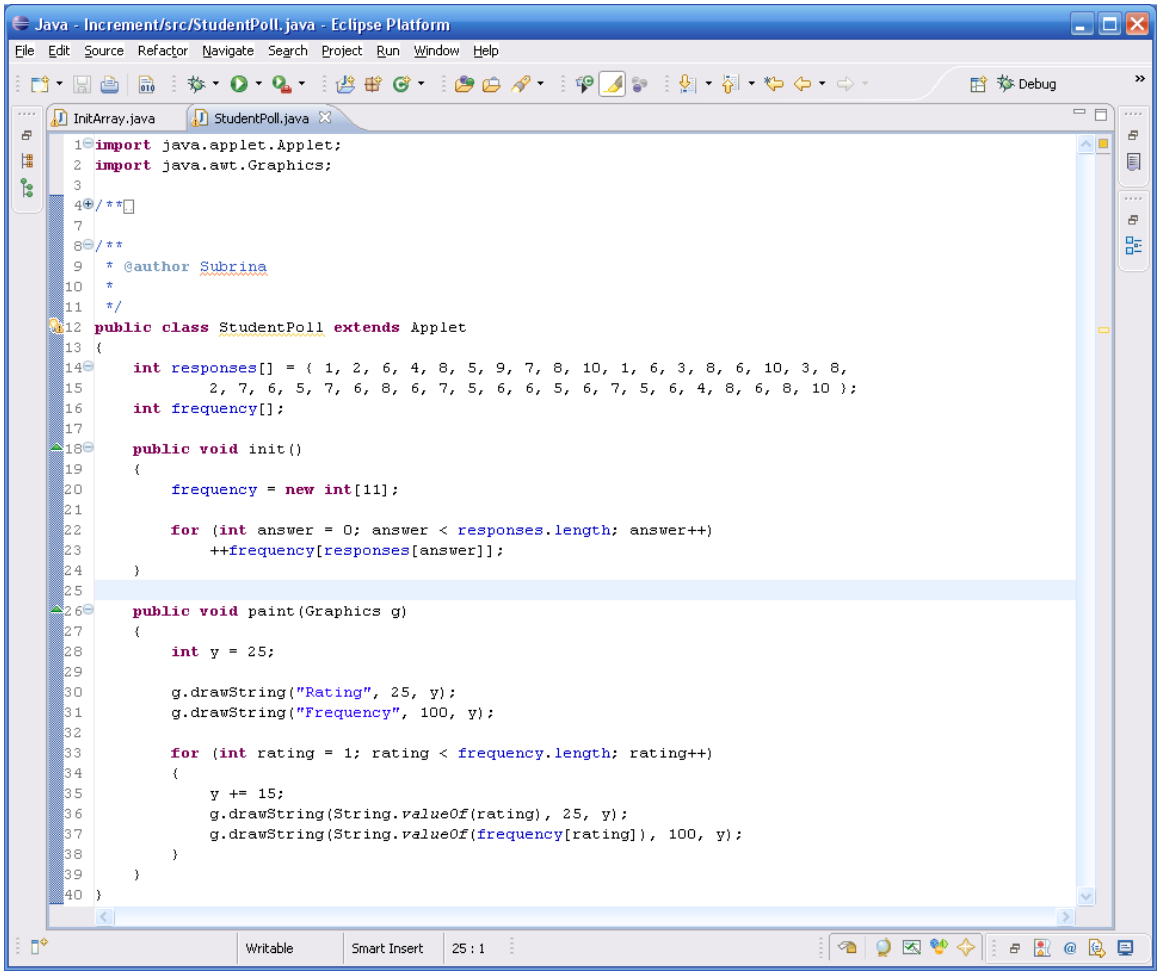
SumArray.java Java program

```
public class SumArray extends Applet
        {
                int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                int total;

                public void init()
                {
                        total = 0;

                        for (int i = 0; i < a.length; i++)
                                total += a[i];
                }

                public void paint(Graphics g)
                {
                        g.drawString("Total of elements: " + total, 25, 25);
                }
        }
```

Applet sums the elements of its array

```
Int
Int
Int
Integer val4 = new Integer(23);
```

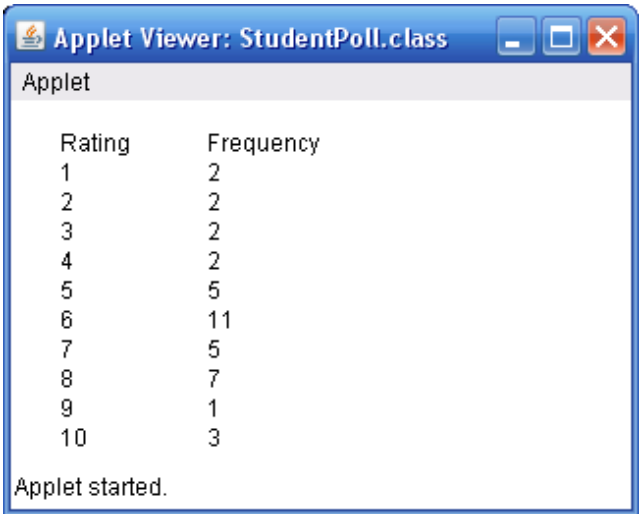
Example Summing Frequency Distribution of an array




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

/**

/**
 * @author Subrina
 *
 */
public class StudentPoll extends Applet
{
    int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6, 10, 3, 8,
            2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
    int frequency[];

    public void init()
    {
        frequency = new int[11];

        for (int answer = 0; answer < responses.length; answer++)
            ++frequency[responses[answer]];
    }

    public void paint(Graphics g)
    {
        int y = 25;

        g.drawString("Rating", 25, y);
        g.drawString("Frequency", 100, y);

        for (int rating = 1; rating < frequency.length; rating++)
        {
            y += 15;
            g.drawString(String.valueOf(rating), 25, y);
            g.drawString(String.valueOf(frequency[rating]), 100, y);
        }
    }
}
```

StudentPoll.java Java Program

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

/**
 * @author Subrina
 *
 */
public class StudentPoll extends Applet
{
        int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6, 10, 3, 8,
                            2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
        int frequency[];

        public void init()
        {
                frequency = new int[11];

                for (int answer = 0; answer < responses.length; answer++)
                        ++frequency[responses[answer]];
        }

        public void paint(Graphics g)
        {
                int y = 25;

                g.drawString("Rating", 25, y);
                g.drawString("Frequency", 100, y);

                for (int rating = 1; rating < frequency.length; rating++)
                {
                        y += 15;
                        g.drawString(String.valueOf(rating), 25, y);
                        g.drawString(String.valueOf(frequency[rating]), 100, y);
                }
        }
}
```



Recursion and arrays

Below, we write the array elements backwards *recursively*

Java - Increment/src/Recurs1.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run Window Help

Debug

Recurs1.java

```
2  import java.awt.Graphics;
3
4  /**
7
9   * @author Subrina
12 public class Recurs1 extends Applet
13 {
14     int myArray[];
15
16     public void init()
17     {
18         myArray = new int[16];
19         for (int i = 0; i < myArray.length; i++)
20             myArray[i] = (int) (Math.random() * 26);
21     }
22
23     public void paint(Graphics g)
24     {
25         int x = 25, y = 30;
26         g.drawString("Array elements: ", 25, 10);
27         for (int i = 0; i < myArray.length; i++)
28         {
29             g.drawString(String.valueOf(myArray[i]), x, y);
30             x += 25;
31         }
32         printBackwards();
33     }
34
35     public void printBackwards()
36     {
37         System.out.println("\nHere are the elements backwards:");
38         prBack(0);
39     }
40
41     private void prBack(int elNo)
42     {
43         if (elNo == myArray.length)
44             return; // do nothing base case
45         prBack(elNo + 1); // the recursive case
46         System.out.print(myArray[elNo] + "  ");
47     }
48 }
```

printBackwards() is an assisting method to the real recursive **prBack**() method

Writable Smart Insert 26 : 50

Applet Viewer: Recurs1.class

Applet

Array elements:

11 11 1 2 11 6 14 12 4 4 24 17 13 11 0 8

Applet started.

Java - Increment/src/Recurs1.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run Window Help

Debug

Console

Recurs1 [Java Applet] C:\Program Files\Java\jre1.6.0_05\bin\javaw.exe (Aug 31, 2008 10:15:48 PM)

```
Here are the elements backwards:
8  0  11  13  17  24  4  4  12  14  6  11  2  1  11  11
```

Recurs1.java Java program

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

/**
 * @author Subrina
 *
 */
public class Recurs1 extends Applet
{
	int myArray[];

	public void init()
	{
		myArray = new int[16];
		for (int i = 0; i < myArray.length; i++)
			myArray[i] = (int) (Math.random() * 26);
	}

	public void paint(Graphics g)
	{
		int x = 25, y = 30;
		g.drawString("Array elements: ", 25, 10);
		for (int i = 0; i < myArray.length; i++)
		{
			g.drawString(String.valueOf(myArray[i]), x, y);
			x += 25;
		}
		printBackwards();
	}

	public void printBackwards()
	{
		System.out.println("\nHere are the elements backwards:");
		prBack(0);
	}

	private void prBack(int elNo)
	{
		if (elNo == myArray.length)
			return; // do nothing base case
		prBack(elNo + 1); // the recursive case
		System.out.print(myArray[elNo] + "  ");
	}
}
```

Using a *nested* loop to print out a histogram from an array of frequencies

Java - Increment/src/Histogram.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run Window Help

Debug

Histogram.java

```
1  import java.applet.Applet;
2  import java.awt.Graphics;
3
4  /**
7
8  /**
9   * @author Subrina
10  *
11  */
12 public class Histogram extends Applet
13 {
14     int n[] = { 9, 13, 11, 5, 1, 19, 34, 21, 16, 8 };
15
16     public void paint(Graphics g)
17     {
18         int x; // position of * in histogram
19         int y = 25; // vertical position in applet
20
21         g.drawString("Element", 25, y);
22         g.drawString("Value", 100, y);
23         g.drawString("Histogram", 175, y);
24
25         for (int i = 0; i < n.length; i++)
26         { // the outer loop
27             y += 15;
28             g.drawString(String.valueOf(i), 25, y);
29             g.drawString(String.valueOf(n[i]), 100, y);
30             x = 175;
31
32             for (int j = 1; j <= n[i]; j++)
33             { // the inner loop
34                 g.drawString("*", x, y);
35                 x += 7;
36             }
37         }
38     }
39 }
```

Writable | Smart Insert | 39 : 2

Applet Viewer: Histogram.class

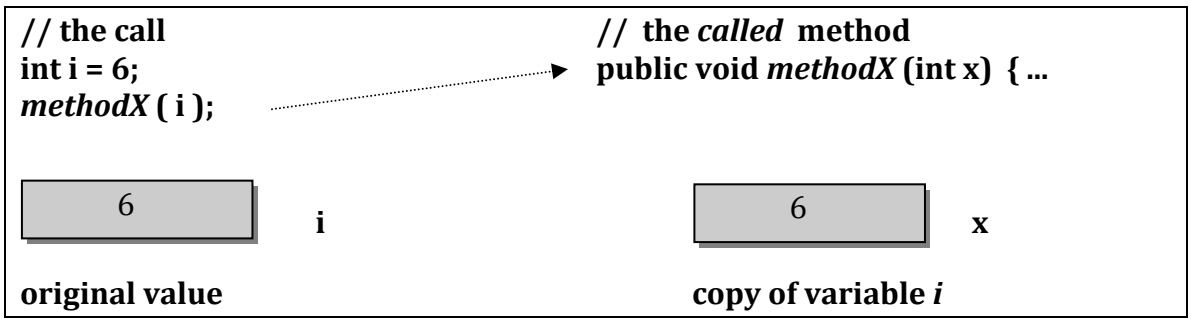
Applet

Element	Value	Histogram
0	9	*********
1	13	*************
2	11	***********
3	5	*****
4	1	*
5	19	*******************
6	34	**********************************
7	21	**********************
8	16	****************
9	8	********

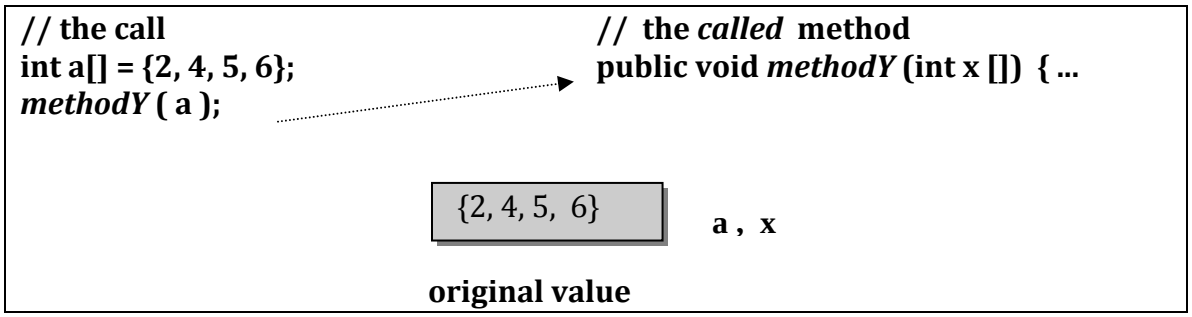
Applet started.

Passing *parameters* (in arrays and in general)

1. Primitive data types are always passed by value. In this way the *called* method has its own copy of the parameter. This implies that a method can never change its formal parameter.



2. Objects are *always* passed by reference. In this case, the *called* method is always able to access the actual data. The formal parameter behaves like an *alias* for the original object.



Arrays are treated like objects in Java, therefore the *called* method can always modify the elements of the array.

When a method returns a value, the values are returned by *value* for primitive data type and by *reference* for objects

Example of Passing Whole Arrays and Individual Array Elements

```
int X[] = {2, 5, 7, 3, 4, 1}      // declaring and defining the array
modifyArray (X);                  //  this effectively doubles all the array values
modifyElement(X[2])               // this does nothing to element X[2]
```

```
public void modifyArray(int b[]){ // b is a reference to actual array X
        for (int j = 0; j < b.length; j++)
                b[j] *= 2;
}

public void modifyElement(int e){ // e is a copy of actual array element X[2]
        e *= 2;
}
```

Quicksort and Binary Search methods (*optional*)

These algorithms are often used to implement efficient sorting and searching methods in programs and systems

Quicksort method code

Note: parameters *low* and *hi* need to be passed in, even within the same class definition. These are the local variables for each recursion level

```
private static void qsort(short[] A, int low, int hi){
        int k;

        if (hi - low > 1)
        {
                k = partition(A, low, hi);
                qsort(A, low, k - 1);
                qsort(A, k + 1, hi);
        }
}
private static int partition(short[] A, int low, int hi){
        int i = low, j = hi;
        short pivot = A[low];
        while (i <= j){

                while (i <= hi && A[i] <= pivot)
                        i++;

                while (j >= low && A[j] > pivot)
                        j--;

                if (i < j)
                        swap(A, i, j);
        }

        swap(A, low, j);
        return j;
        }
private static void swap(short[] A, int x, int y){
        short temp = A[x];
        A[x] = A[y];
        A[y] = temp;
}
```

Recursive Binary Search Method Code

Note: The method returns the index of the array X where a match is found, or the value -1 if a match does not exist.

```
private static int binarysearch(short[] X, short key, int low, int hi)
{
        if (low <= hi)
        {

                if (low == hi)
                        return (X[low] == key ? low : -1);
                else
                {
                        int mid = (low + hi) / 2;
                        if (X[mid] == key)
                                return mid;
                        else if (key < X[mid])
                                return binarysearch(X, key, low, mid - 1);
                        else
                                return binarysearch(X, key, mid + 1, hi);
                }
        }
        return -1;
}
```

Higher Dimensioned Arrays

2-D arrays are actually arrays of arrays. That is in a 2-D array, the first element is a 1-D array. Java uses double subscripting to access the elements.

Arrays can be defined to be multidimensional with a multiple number of brackets. For example

```
int x[ ][ ];
```

declares a 2-D array of **int** elements

```
char ch [ ][ ][ ];
```

declares a 3-D array of **char**

Alternative way to declare arrays are:

```
int  [][] x;

and

char [][] ch;
```

Defining Multidimensional Arrays

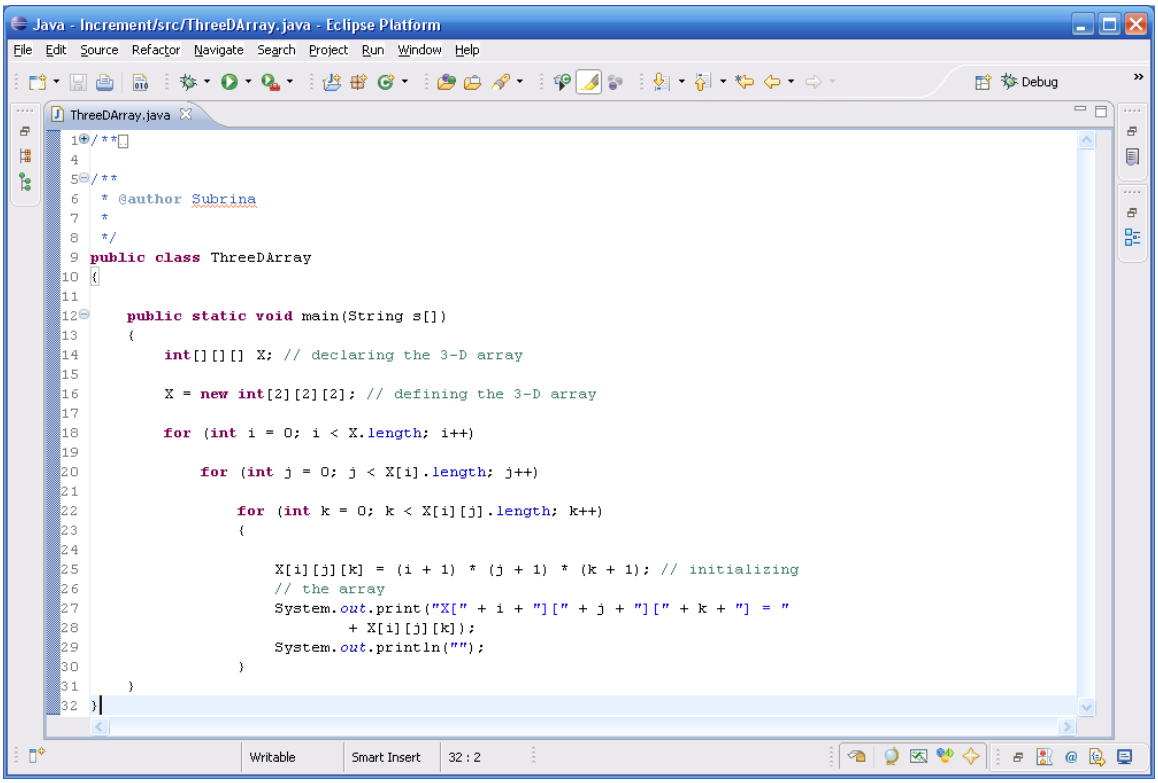
```
char [][][] ch;
ch = new char[3][4][2];

defines a 3-D array three deep by 4 rows, two columns
```

```
double [][] X = {{3,5,7}, {4,6,8}, {3,1,1}};

defines a 2-D array three rows by three columns
```

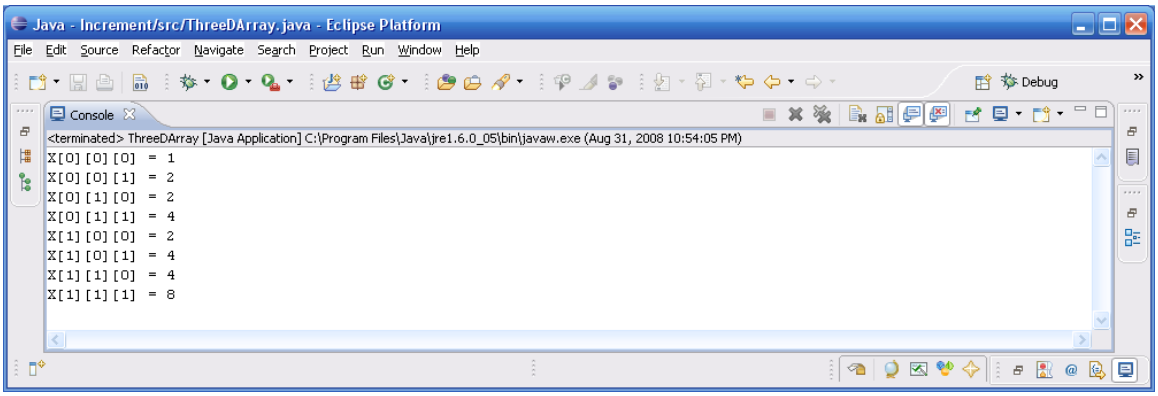
```
int [][] Y = {{2,4,6,8,10}, {3,4,1}, {4,0}};

defines a 2-D array with variable sized array subelements

Y[0].length ➔ 5
Y[1].length ➔ 3
Y[2].length ➔ 2
```

```
short  [][] S;

S = new short[5][];

defines the first dimension of the array

S[0] = new short[3];        // allocates 2nd dimension
S[1] = new short[2];
S[2] = new short[4];
```

3-D Array Example



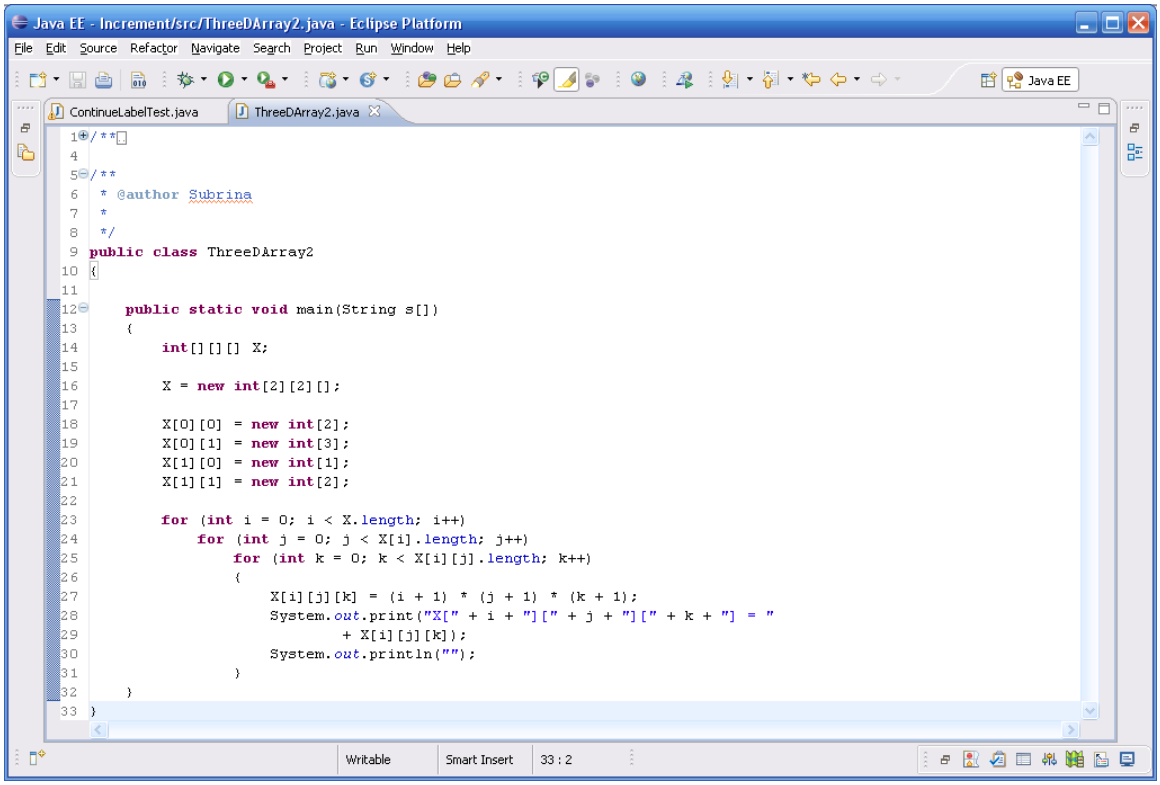
Output



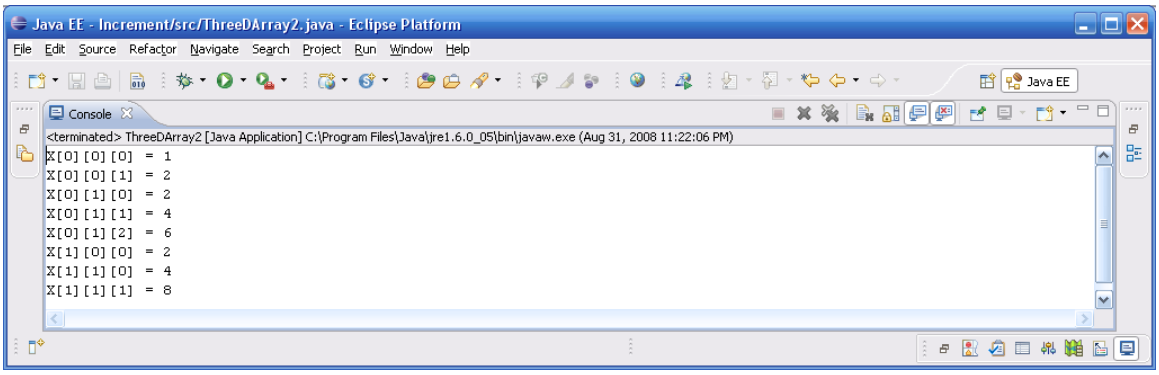
ThreeDArray.java Java program

```
/**
 * @author Subrina
 *
 */
public class ThreeDArray
{

        public static void main(String s[])
        {
                int[][][] X; // declaring the 3-D array

                X = new int[2][2][2]; // defining the 3-D array

                for (int i = 0; i < X.length; i++)

                        for (int j = 0; j < X[i].length; j++)

                                for (int k = 0; k < X[i][j].length; k++)
                                {

                                        X[i][j][k] = (i + 1) * (j + 1) * (k + 1); // initializing
                                        // the array
                                        System.out.print("X[" + i + "][" + j + "][" + k + "] = "
                                                        + X[i][j][k]);
                                        System.out.println("");
                                }
        }
}
```

Example of *ragged* 3-D array



Output



ThreeDArrays2.java Java Program

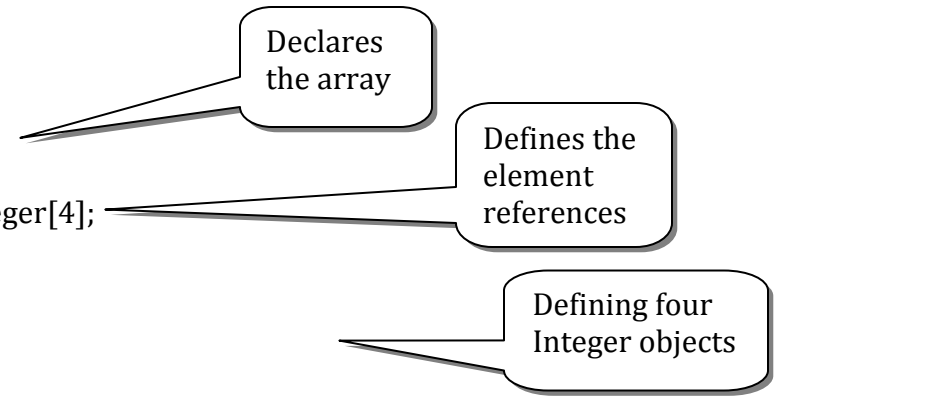
```
/**
 * @author Subrina
 *
 */
public class ThreeDArray2
{
        public static void main(String s[])
        {
                int[][][] X;

                X = new int[2][2][];

                X[0][0] = new int[2];
                X[0][1] = new int[3];
                X[1][0] = new int[1];
                X[1][1] = new int[2];

                for (int i = 0; i < X.length; i++)
                        for (int j = 0; j < X[i].length; j++)
                                for (int k = 0; k < X[i][j].length; k++)
                                {
                                        X[i][j][k] = (i + 1) * (j + 1) * (k + 1);
                                        System.out.print("X[" + i + "][" + j + "][" + k + "] = "
                                                        + X[i][j][k]);
                                        System.out.println("");
                                }
        }
}
```

Arrays and *Objects*

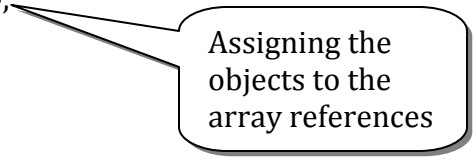
When we create an array of *objects*, we need to do two things

1. define the array references
2. define the object elements

Example:



X[0] = val1; X[1] = val2; X[2] = val3; X[3] = val4;



alternatively, we can write

Copying Arrays

There is a System class method **arraycopy**() which copies elements from the source array to the target array. The call matches the

System.**arraycopy** (<source>, source_index, <target>, target_index, el_count);

Java provides this method for built-in type such as boolean, byte, short, int, long, float, double, *as well as* for the class **Object**

Example:

```
public static void main(String s[])
{
        Integer[] X, Y = new Integer[3];
        X = new Integer[3];

        X[0] = new Integer(5);
        X[1] = new Integer(-3);
        X[2] = new Integer(2);

        System.out.println("X[0] = " + X[0]);
        System.out.println("X[1] = " + X[1]);
        System.out.println("X[2] = " + X[2]);

        System.arraycopy(X, 1, Y, 0, 2);

        System.out.println("Y[0] = " + Y[0]);
        System.out.println("Y[1] = " + Y[1]);
}
```

Output

```
X[0] = 5
X[1] = -3
X[2] = 2
Y[0] = -3
Y[1] = 2
```

Designing Classes

To create an object, we always are invoking special methods called *constructors*. There are a number of important features about constructors

1. *constructors* are called whenever an object is being instantiated
2. Any class without a constructor, has a default one defined automatically. The default constructor does two things
    i. calls the constructor method for the immediate superclass
    ii. initializes numerics, booleans, and references
3. *constructors* are usually overloaded. This allows building objects in a variety of ways
4. *constructors* are always public and do not return a value (not even *void*)

The name of a *constructor* is always the same as the name of the class of which they are a member

```
public class Fraction {
private int numerator, denominator;
Fraction () { numerator = 0; denominator = 1; }
Fraction (int top) { numerator = top; denominator = 1; }
Fraction (int top, int bottom) { numerator = top; denominator = bottom; }
....
....    }
```

Three constructors for the class Fraction

Classes define behavior for the object by actions which are in the class methods. Methods often take *no parameters* because they are being invoked by a particular object. As such, the instance data members are an inherent part of the object itself.

Public and Private keywords

The keywords **private** and **public** are *member access modifiers*. Any member tagged as **private** implies that the member cannot be accessed outside the methods of the immediate class.

public data and methods are those we want to present as accessible to the clients of the class. It is part of the class public interface, that is, visible to all class clients. A client is any other class of method in another class which defines an object.

Example:

```
public class MathProbs {
       Fraction f1, f2, F[];
       ...
       ...    }
```

Class MathProbs is a client of the Fraction class, as are all methods in the MathProbs class. This is because the Fraction objects have class scope

Because Fraction class members, *numerator* and *denominator* are **private** references in methods of class MathProbs such as

```
f1.numerator ++
        or
(F[i].denominator != 0)
```

are *illegal!*

Private Data Members

Classes usually provide *accessor* (get) and *mutator* (set) methods to allow clients to *get* and *set* private data members.

Note: **private** methods are often used to modularize the class method code. For example, a private method **reduce**() would be very useful in the Fraction class so that the following sequence would work as shown below:

```
Fraction f;
f = new Fraction (2,6);
f.display()
```

➔ (1/3)

The private method **reduce( )** put the Fraction object in lowest terms. This is very useful when multiplying fractions, for instance

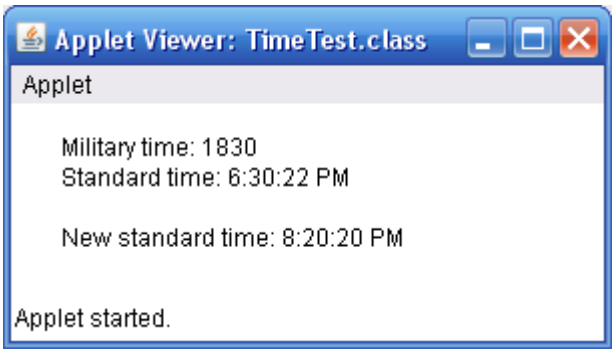
$(2/3) * (4/8) = (1/3)$ *not* $(8/24)$

Note: The accessor and mutator methods are usually public. The do not violate the OO paradigm of data encapsulation, since they can perform considerable error checking and validation prior to actual modification of the private data members

Example of overload constructors, *this* reference, and chaining

```
/**
 * @author Subrina
 *
 */
public class Time
{
	private int hour; // 0 - 23
	private int minute; // 0 - 59
	private int second; // 0 - 59

	public Time(){
		setTime(0, 0, 0);
	}

	public Time(int h){
		setTime(h, 0, 0);
	}

	public Time(int h, int m){
		setTime(h, m, 0);
	}

	public Time(int h, int m, int s){
		setTime(h, m, s);
	}

	public Time setTime(int h, int m, int s){
		setHour(h); // set the hour
		setMinute(m); // set the minute
		setSecond(s); // set the second

		return this;
	} // enables chaining

	public Time setHour(int h){
		hour = ((h >= 0 && h < 24) ? h : 0);
		return this;
	} // enables chaining

	public Time setMinute(int m){
		minute = ((m >= 0 && m < 60) ? m : 0);
		return this;
	}

	public Time setSecond(int s){
		second = ((s >= 0 && s < 60) ? s : 0);
		return this;
	}

	public int getHour(){
		return hour;
	}

	public int getMinute(){
		return minute;
	}

	public int getSecond(){
		return second;
	}

	public String toMilitaryString(){
		return (hour < 10 ? "0" : "") + hour + (minute < 10 ? "0" : "")
				+ minute;
	}

	public String toString(){
		return ((hour == 12 || hour == 0) ? 12 : hour % 12) + ":"
				+ (minute < 10 ? "0" : "") + minute + ":"
				+ (second < 10 ? "0" : "") + second
				+ (hour < 12 ? " AM" : " PM");
	}
}
```

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

/**
 * @author Subrina
 *
 */
public class TimeTest extends Applet
{
        private Time t;

        public void init()
        {
                t = new Time();
        }

        public void paint(Graphics g)
        {

                t.setHour(18).setMinute(30).setSecond(22);
                g.drawString("Military time: " + t.toMilitaryString(), 25, 25);
                g.drawString("Standard time: " + t.toString(), 25, 40);

                g.drawString("New standard time: " + t.setTime(20, 20, 20).toString(),
                                25, 70);
        }
}
```



Class *composition*

When an object of one class is a member of another class, that is referred to as class composition. The class is *composed* of instances of other classes

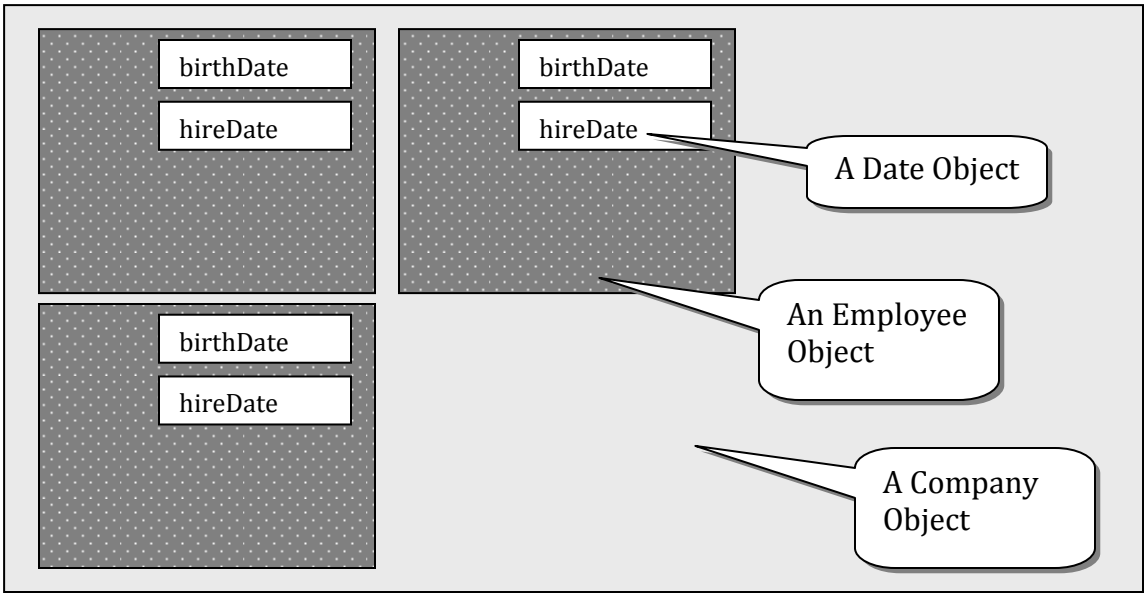
This facilitates the design process as components can be used (and reused) in a variety of new classes and programs.

Example of Composition:

Date is a class which can store, validate and retrieve date information

*Employee*s have hire Dates and birth dates

A *Company* has Employees



Code Implementing the *Company*, *Employee*, *Date* Composition

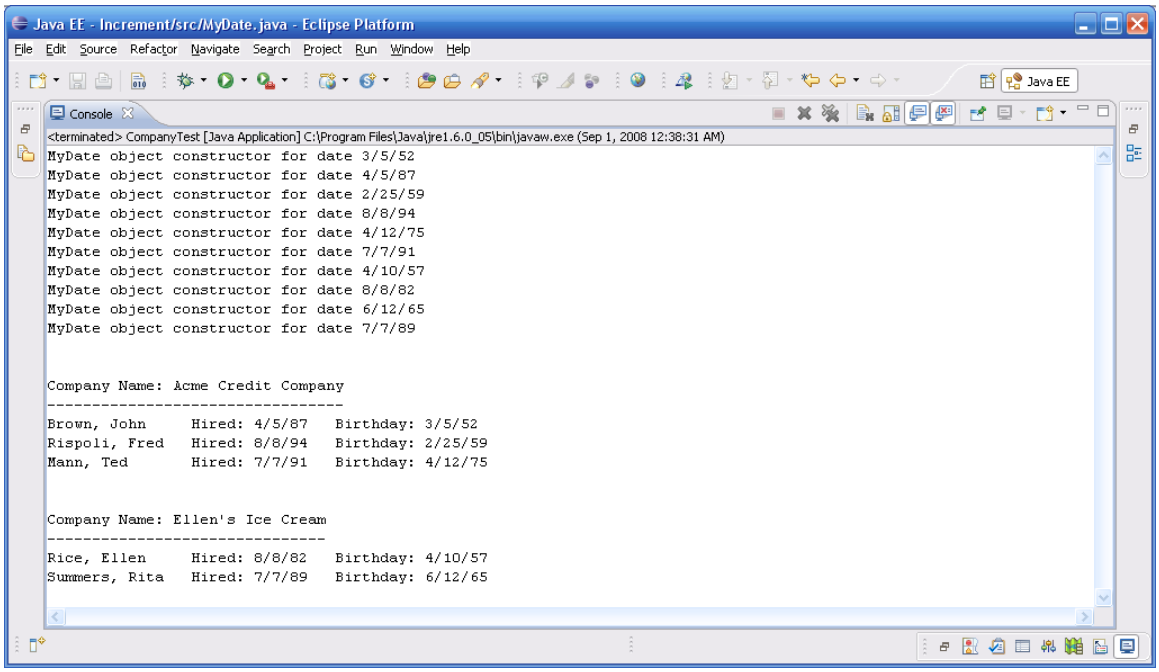
```
public class CompanyTest
{
        public static void main(String s[])
        {

                MyEmployee emps1[], emp1, emp2, emp3, emp4, emp5, emp6, emps2[];
                Company company1, company2;

                emp1 = new MyEmployee("John", "Brown", 3, 5, 52, 4, 5, 87);
                emp2 = new MyEmployee("Fred", "Rispoli", 2, 25, 59, 8, 8, 94);
                emp3 = new MyEmployee("Ted", "Mann", 4, 12, 75, 7, 7, 91);
                emp4 = new MyEmployee("Ellen", "Rice", 4, 10, 57, 8, 8, 82);
                emp5 = new MyEmployee("Rita", "Summers", 6, 12, 65, 7, 7, 89);

                emps1 = new MyEmployee[3];
                emps2 = new MyEmployee[2];

                emps1[0] = emp1;
                emps1[1] = emp2;
                emps1[2] = emp3;

                emps2[0] = emp4;
                emps2[1] = emp5;

                company1 = new Company(emps1, "Acme Credit Company");
                company2 = new Company(emps2, "Ellen's Ice Cream");

                company1.display();
                company2.display();
        }
}
```

```
public class Company
{
        MyEmployee[] emp;
        String name;
        int empNo;

        Company(MyEmployee[] emp, String name)
        {
                this.emp = emp;
                empNo = emp.length;
                this.name = name;
        }

        public void display()
        {
                System.out.println("\n\nCompany Name: " + name);
                for (int i = 0; i < name.length() + 14; i++)
                        System.out.print("-");
                System.out.println("");

                for (int i = 0; i < emp.length; i++)
                {
                        System.out.println(emp[i].toString());
                }
        }
}
```

```
public class MyEmployee
{

	private String firstName, lastName;
	private MyDate birthDate, hireDate;

	public MyEmployee(String fName, String lName, int bMonth, int bDay,
			int bYear, int hMonth, int hDay, int hYear)
	{
		firstName = fName;
		lastName = lName;
		birthDate = new MyDate(bMonth, bDay, bYear);
		hireDate = new MyDate(hMonth, hDay, hYear);
	}

	public String toString()
	{
		return lastName + ", " + firstName + "\tHired: " + hireDate.toString()
				+ "\tBirthday: " + birthDate.toString();
	}
}
```

```
public class MyDate
{
	private int month; // 1-12
	private int day; // 1-31 based on month
	private int year; // any year

	public MyDate(int mn, int dy, int yr)
	{
		if (mn > 0 && mn <= 12) // validate the month
			month = mn;
		else
		{
			month = 1;
			System.out.println("Month " + mn + " invalid. Set to month 1.");
		}

		year = yr; // could also check
		day = checkDay(dy); // validate the day

		System.out.println("MyDate object constructor for date " + toString());
	}

	private int checkDay(int testDay)
	{
		int daysPerMonth[] = { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30,
				31 };

		if (testDay > 0 && testDay <= daysPerMonth[month])
			return testDay;

		System.out.println("Day " + testDay + " invalid. Returning 1.");
		return 1; // leave object in consistent state
	}

	public String toString()
	{
		return month + "/" + day + "/" + year;
	}
}
```

Output



The *finalize* method

The complement of the object *constructor* method is the *finalize* method. It has a number of important properties

1. The name of the method is always *finalize*
2. *finalize* cannot be overloaded
3. *finalize* is called automatically every time the object is destroyed
    4. Object data space is automatically returned to the system but in finalize()we can
        i. free file resources
        ii. free socket communication resources
        iii. null out references to other objects
5. *finalize* does not take any parameters and returns void
6. The finalize method may not be called immediately, but it is guaranteed to be called as soon as possible, and *always* before garbage collection is performed

Example:

```
import java.io.FileInputStream;

/**
 * @author Subrina
 *
 */
public class FileOpen
{
        FileInputStream myFile = null;

        FileOpen(String fname)
        {
                try
                {
                        myFile = new FileInputStream(fname);
                } catch (java.io.FileNotFoundException e)
                {
                        System.err.println("No such file called " + fname);
                }
        }

        protected void finalize() throws Throwable
        {
                if (myFile != null)
                {
                        myFile.close();
                        myFile = null;
                }
        }
}
```

The *finalize* method return file resources to the system

Static Members (Data and Methods)

Static data members exist in a class independent of any object instances. *Static* class variables have class scope. *Static* data may be public or private.

Static methods in a class cannot access instance variables, but can access any static data members.

To access static members we access through *class name.static member name*

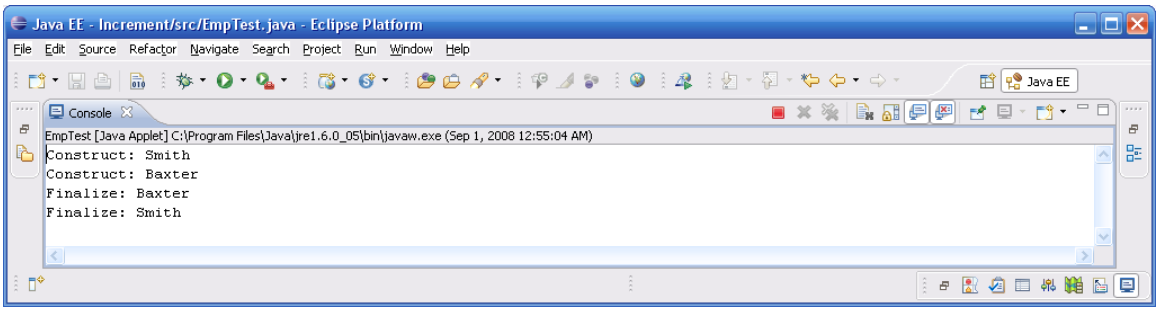
Any private static member data must be accessed through a public static method, or a friendly* static method in the class

```
public class Employee
{
        private String firstName;
        private String lastName;
        private static int count;

        public Employee(String first, String last){
                firstName = first;
                lastName = last;
                ++count;
                System.out.println("Construct: " + lastName);
        }

        public void finalize(){
                --count;
                System.out.println("Finalize: " + lastName);
        }

        public String getFirstName(){
                return new String(firstName);
        }

        public String getLastName(){
                return new String(lastName);
        }

        public static int getCount(){
                return count;
        }
}
```

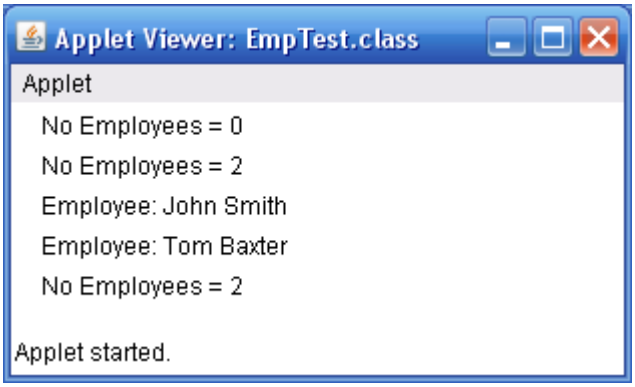
Code Example of *static* members and *finalize* method

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

public class EmpTest extends Applet{
        public void paint(Graphics g){
                g.drawString("No Employees = " + Employee.getCount(), 15, 15);
                Employee e1 = new Employee("John", "Smith");
                Employee e2 = new Employee("Tom", "Baxter");
                g.drawString("No Employees = " + Employee.getCount(), 15, 35);
                g.drawString("Employee: " + e1.getFirstName() + " " + e1.getLastName(),
                                15, 55);
                g.drawString("Employee: " + e2.getFirstName() + " " + e2.getLastName(),
                                15, 75);
                e1 = e2 = null;
                System.gc();
                g.drawString("No Employees = " + Employee.getCount(), 15, 95);
        }

        public boolean mouseDown(Event e, int x, int y){
                System.out.println("No Employees = " + Employee.getCount());
                return true;
        }
}
```

Console Output



GUI Output



* The *final* Keyword

We can define variables to be *final*. Final variables are, in effect, constants. They cannot be modified.

```
public class SomeClass {
public final static int dozen = 12;
...
```

The following reference gives the following compiler error message:

dozen ++ ;

C:\...\>javac SomeClass.java

SomeClass.java:9: **Can't assign a value to a final variable: dozen**

dozen ++;

^

1 error

final public static variables are often used as Class constants. This is used throughout Java's standard API classes to provide clear values used in methods in those classes.

An example of a constant Employee object can be placed into the Employee class code:

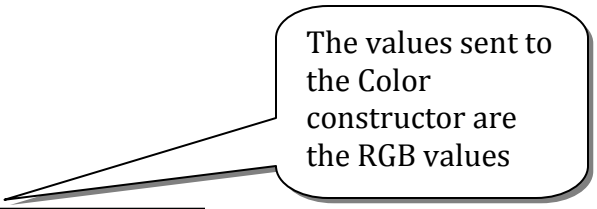
```
public final static Employee Kons = new Employee("Larry","Jones");
```

The **awt** package **Color** class provides colors and methods for manipulating screen colors

Java provides a **Color** class with the following constants

Color.green
Color.red
Color.lightGray
etc...

These are defined as follows:




```
public final static Color orange = new Color(255, 200, 0);
```

These can be used to set colors as in the following:

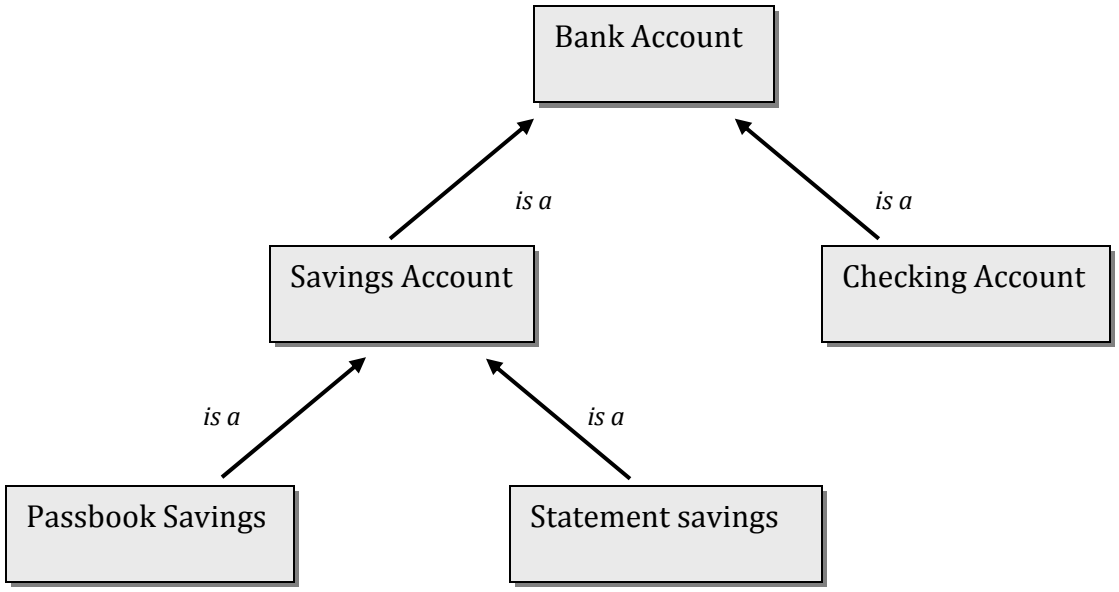
```
setColor (Color.orange);
setBackground (Color.lightGray);
```

Object Oriented Programming

While Object-based programming uses program design with classes and object composition, it does not fully exploit the possibilities of OOP. It lacks two important features which are regarded by many as true OO features

1. *polymorphism* – allows a more general approach to writing software, allowing the handling of future behavior in yet-to-be-designed classes. It also allows a dynamic behavior of objects where object *messages* activate dispatching of methods at run time, *not* at compile time.
2. *inheritance* – encourages and enables software *reusability*, as new classes *extend* existing ones, and add on to their capabilities.

These features help to deal with the complexity inherent in large-scale computer systems



Typical Class Hierarchy

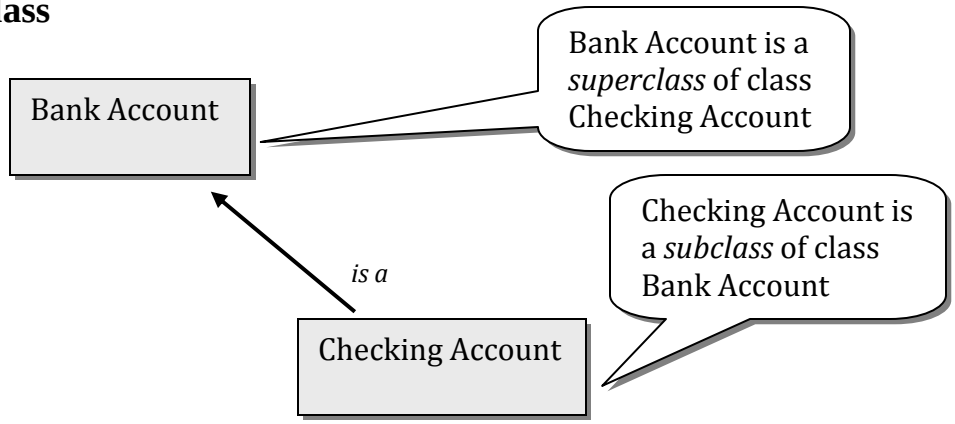
Types of members in a class

There are four types of class members that can be defined as part of a class

1. *private* – class members cannot be accessed outside the member methods
2. *public* – class members can be accessed by anyone
3. *package* – class members can be accessed on a directory (*package*) level

4. *protected* – class members can be accessed *only* by methods in direct or indirect subclasses

In *inheritance* a class is either a direct subclass or a direct superclass of another class



How Protected is *protected?*

Protected members can be accessed by

1. the class itself
2. a subclass method (and its descendants)

methods in the same package as the subclass method

```
class Super{
        protected int k = 5;
}
class Sub1 extends Super{
        public void print(){
                System.out.println("k = " + k);
        }
}
class Sub2 extends Sub1{
        void print2(){
                System.out.println("k = " + k);
        }
}
public class MultInherit{
        public static void main(String s[]){
                Sub2 obj = new Sub2();
                System.out.println("From MultInherit:  k = " + obj.k);
        }
}
➔ From MultInherit: k = 5
```

Example of *multiple inheritance*

Example of Multiple Inheritance with Calls to Superclass Methods

```
class Super
{
        protected int k = 5;
}

class Sub1 extends Super
{
        public void print()
        {
                System.out.println("From Sub1:  k = " + k);
        }
}

class Sub2 extends Sub1
{
        void print2()
        {
                super.print();
                System.out.println("From Sub2:  k = " + k);
        }
}

public class MultInherit
{
        public static void main(String s[])
        {
                Sub2 obj = new Sub2();
                obj.print2();
                System.out.println("From MultInherit:  k = " + obj.k);
        }
}
```

***Note**:*

1. method **print2**() calls super method **print**(). A call to **print**() rather than **super.print**() would also work. Why?
2. we need to call **print2**() with **obj.print2**(). Why would a call to **print2( )** not work?
3. *k* is available to **class MultInherit** because an instance variable, *obj*, is a sub-sub-class of the class Super, and *k* is a protected member of the Super class.

Overriding the Defined Methods in a Superclass

```
class Super{
        protected int k = 5;
}
class Sub1 extends Super{
        public void print(){
                System.out.println("From Sub1:  k = " + k);
        }
}
class Sub2 extends Sub1{
        public void print(){
                super.print(); // without "super" it would be a recursive call
                System.out.println("From Sub2:  k = " + k);
        }
}
public class MultInherit{
        public static void main(String s[]){
                Sub2 obj = new Sub2();
                obj.print();
                System.out.println("From MultInherit:  k = " + obj.k);
        }
}
```

Class Sub2 overrides the method **print**()

➔

```
From Sub1:  k = 5
From Sub2:  k = 5
From MultInherit:  k = 5
```

The Relationship Between Superclass and Subclass Objects

A subclass object, or even a sub-subclass object can be treated as an object of the type of the (super) superclass. This is due to the generalization-specification or *isa* inheritance.

```
SuperClass objSuper;
SubClass    objSub1, objSub2;

objSuper = new SuperClass();
objSub1 = new SubClass();

...
...

objSuper = objSub1;
```

By the assignment, we are treating the SubClass object as a SuperClass object

We can create an array of objects from various classes (given they have a *common base class*, or implement a *common interface*) as long as the array is of the common *superclass* or *interface* objects

Note: A superclass object *cannot* be treated as a subclass object. This is because the subclass may have additional data and methods defined which are not part of the superclass.

Note: we can explicitly cast the superclass object as a subclass object, If the superclass reference *actually* refers to a subclass object.

objSub2 = (SubClass) objSuper ;

If objSuper is not a reference to a SubClass object, the result is a **ClassCastException**

Defining Java References

SuperClass

```
void m1()
void m2()
void m3()
```

objSuper

```
SuperClass objSuper;
```

SubClass

```
void m4()
void m5()
void m6()
```

objSub1

objSub2

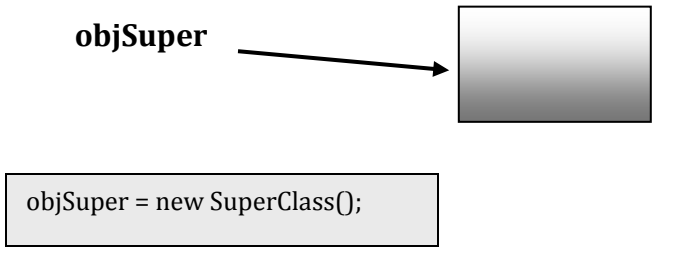
```
SubClass objSub1, objSub2;
```

Note:

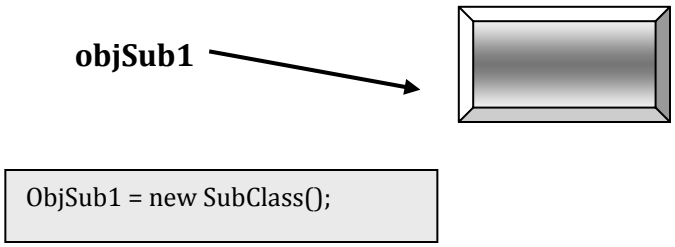
Compiler allows references to ObjSuper **class** objects to invoke **`m1()`**,**`m2()`**,and **`m3()`**.

Compiler allows references to ObjSub **class** objects to invoke **`m1()`**,**`m2()`**, and **`m3()`**,as well as **`m4()`**, **`m5()`**,and **`m6()`**.

Instantiating Java Objects



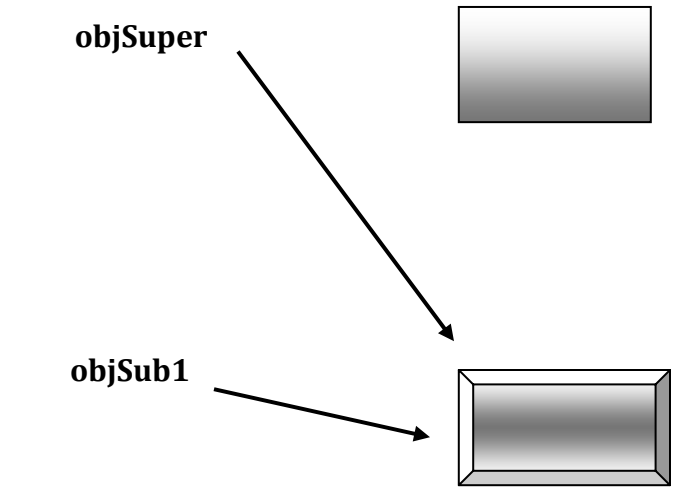

```
objSuper = new SuperClass();
```




```
ObjSub1 = new SubClass();
```

Java Objects Assigned Across Inheritance Hierarchy

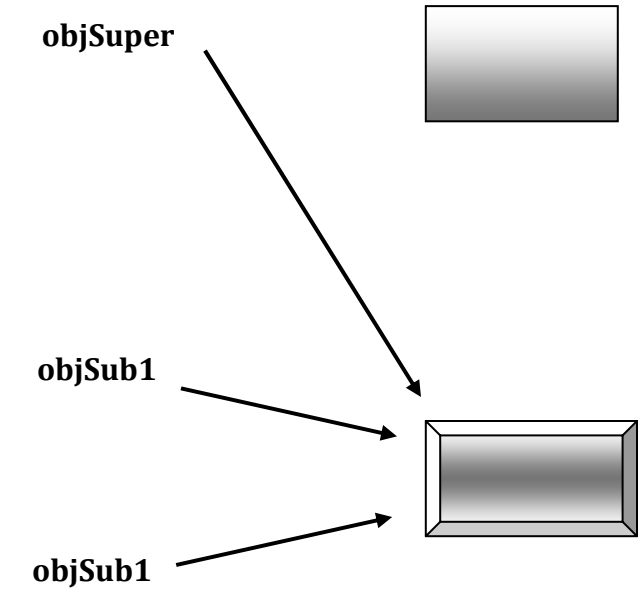
(This is OK)




```
objSuper = objSub1;
```

Java Objects Assigned Across Inheritance Hierarchy

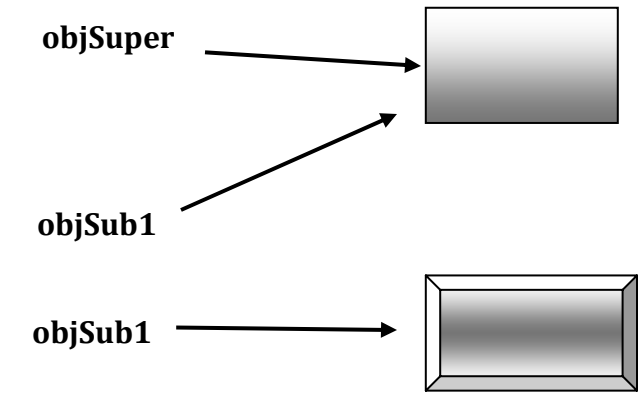
(This is OK, due to cast)




```
objSub2 = (SubClass) objSuper;
```

Java Objects Assigned Across Inheritance Hierarchy

(This is <u>N-O-T</u> OK, despite the cast)




```
objSub2 = (SubClass) objSuper;
```

Example of Inheritance and Super/Sub Class Relationships

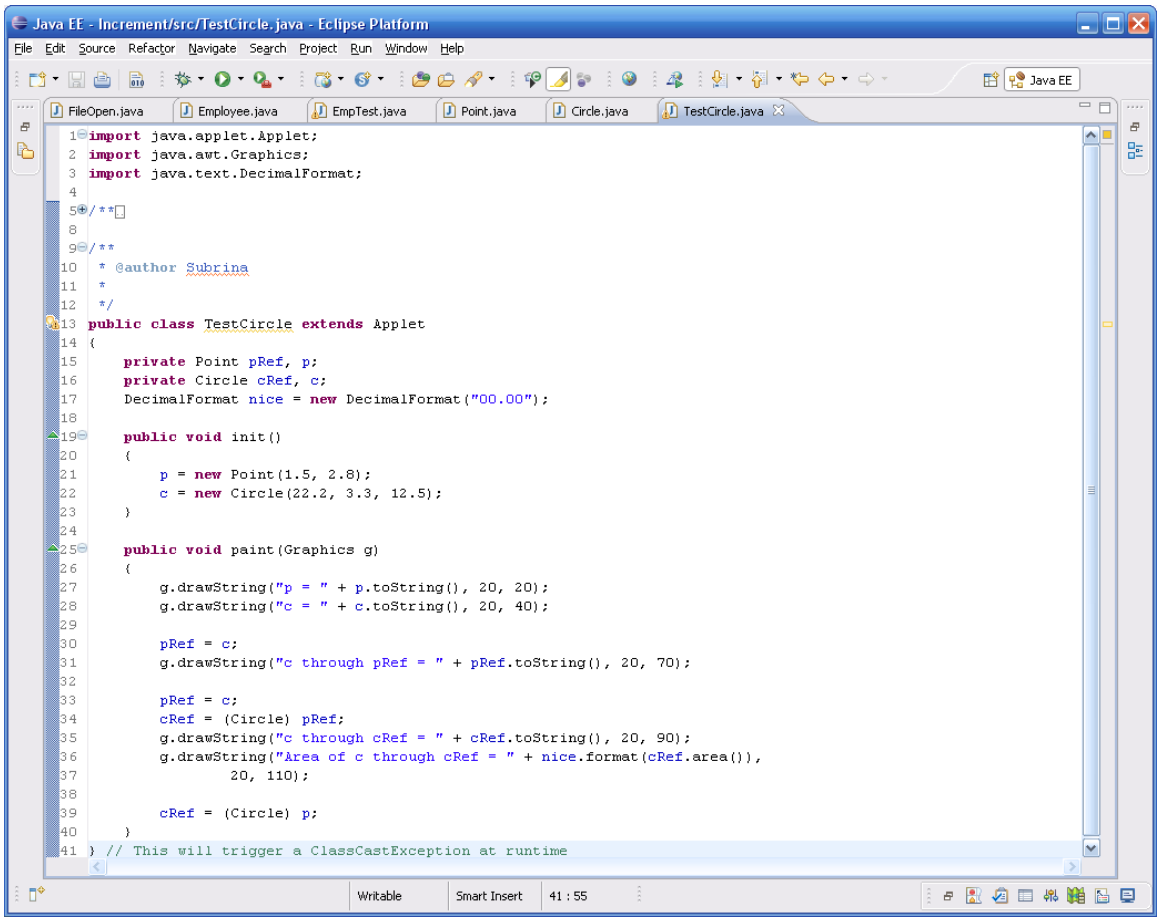
// File: Point.java

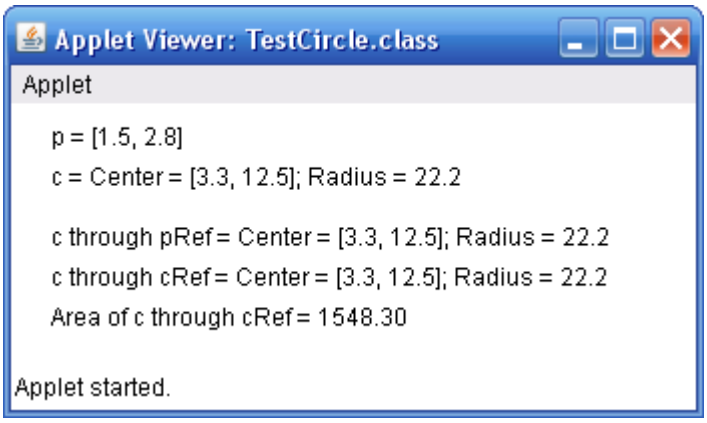
```
public class Point{
        protected double x, y;

        public Point(double a, double b){
                setPoint(a, b);
        }
        public void setPoint(double a, double b){
                x = a;
                y = b;
        }
        public double getX(){
                return x;
        }
        public double getY(){
                return y;
        }
        public String toString(){
                return "[" + x + ", " + y + "]";
                }
        }
```

// File: Circle.java

```
public class Circle extends Point{
        protected double radius;

        public Circle(){
                super(0, 0);
                setRadius(0);
        }
        public Circle(double r, double a, double b){
                super(a, b); // call the base class constructor
                setRadius(r);
        }
        public void setRadius(double r){
                radius = (r >= 0.0 ? r : 0.0);
        }
        public double getRadius(){
                return radius;
        }
        public double area(){
                return 3.14159 * radius * radius;
        }
        public String toString(){
                return "Center = " + "[" + x + ", " + y + "]" + "; Radius = "
                                        + radius;
                }
        }
```

Example of Inheritance and Super/Sub Class Relationships





Order of the Constructor and Finalize Methods

```
//File: Point.java (abbreviated)

public class Point{
        protected double x, y;

        public Point(double a, double b){
                setPoint(a, b);
                System.out.println("Point Constructor for " + this.toString());
        }

        // finalizer method
        public void finalize(){
                System.out.println("Point Finalizer for " + this.toString());
        }
          ........
          ........
}
```

This call actually invokes the **Circle** class toString() method! Note the output on next page.

```
// File: Circle.java (abbreviated)

public class Circle extends Point{ // inherits from Point
        protected double radius;

        public Circle(){
                super(0, 0); // call the base class constructor
                setRadius(0);
        }

        public Circle(double r, double a, double b){
                super(a, b); // call the base class constructor
                System.out.println("Circle Constructor for " + this.toString());
                setRadius(r);
        }

        public void finalize(){
                System.out.println("Circle Finalizer for " + this.toString());
                super.finalize();
        }
        .........
        ......... }
```

Calling the **finalize**() method for the superclass

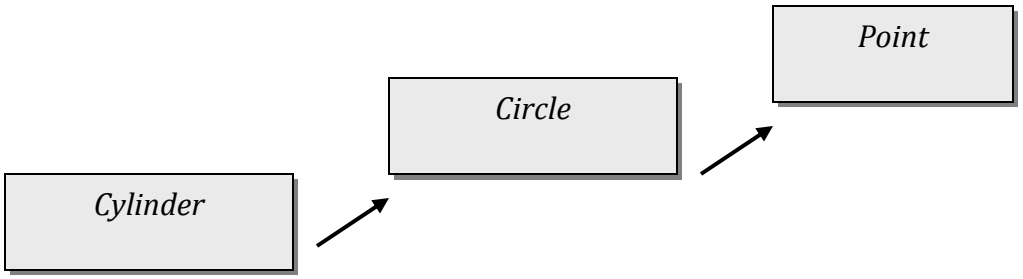
```
// File: TestConsFin.java

public class TestConsFin{
        public static void main(String s[]){
                Circle c1, c2;

                c1 = new Circle(2, 5, 4);
                c2 = new Circle(6, 6, 6);

                c1 = null;
                c2 = null;

                System.gc();
                while (true);
        }
}
```

Explicit call to the Java garbage collector

Output ➔

```
Point Constructor for Center = [5.0, 4.0]; Radius = 0.0
Circle Constructor for Center = [5.0, 4.0]; Radius = 0.0
Point Constructor for Center = [6.0, 6.0]; Radius = 0.0
Circle Constructor for Center = [6.0, 6.0]; Radius = 0.0
Circle Finalizer for Center = [5.0, 4.0]; Radius = 2.0
Point Finalizer for Center = [5.0, 4.0]; Radius = 2.0
Circle Finalizer for Center = [6.0, 6.0]; Radius = 6.0
Point Finalizer for Center = [6.0, 6.0]; Radius = 6.0
```

Multiple Inheritance



A Cylinder object has the behavior and data of a

1. Cylinder
2. Circle
3. Point

A Circle object has the behavior and data of a

1. Circle
2. Point

A Point object has the behavior and data of a

1. Point

Often, when we have a two-level, or more *ISA* relationship, a behavior ***B*** of the lower class object, executed only the part of the behavior specific to the lower class. To get the *complete* behavior, the lower class objects calls the superclass ***B*** behavior, and this process propagates up the inheritance chain.

The *total **B*** behavior for the lowest class object is simple the composite behavior of all the classes up the inheritance chain.

An lower class object calls the *immediate* class behavior with the syntax **super.*B*( )**

A lower class object *cannot* call a method 2 levels up with the syntax **super.super.*B*( )**

Class design with Inheritance

Inheritance is a powerful design tool. One method is to develop a new class by extending one which already exists. Then we do the following

1. Define new data specific to the new class
2. Define additional behavior specific to the new class
3. Override the superclass behaviors to make it specific to the new class

Inheritance is a good software engineering because we can reuse existing classes on an *as is* basis, or reuse existing classes in developing newly extended ones.

Software engineering is improved as we build more reliable systems with quicker turn-around time. Reliability is achieved because we are reusing tried and tested components

Inheritance can be implemented in top-down or bottom-up fashion

top-down

Extend already useful and existing classes in ***isa*** manner to form new classes

This allows *generalization-specialization* of a superclass to a new subclass

bottom-up

Factor out the common behaviors of existing classes to generate a superclass, which contains the common behavior

This allows the uniform treatment of objects of different classes

Implementing *protected* members

```
// File: Point.java

public class Point{
        protected double x, y;

        public Point(double a, double b){
                setPoint(a, b);
        }

        // System.out.println ("Point Constructor for " + this.toString());

        public void setPoint(double x, double y){
                this.x = x;
                this.y = y;
        }

        public void finalize(){
        }

        // System.out.println ("Point Finalizer for " + this.toString());

        public String toString(){
                return "[" + x + ", " + y + "]";
        }

        public double getX(){
                return x;
        }

        public double getY(){
                return y;
        }
}

import ....

public class Example extends Applet{
        private Point p;

        public void init(){
                p = new Point(6, 1.5);
        }

        public void paint(Graphics g){
                g.drawString("Point p coordinates are : " + p.getX() + ", " + p.getY());
        }
}
```

Accessor methods to the x and y data members of point objects

Note that the **paint**() method of class Example, needs to use the accesor methods, **getX**() and **getY**(). The member methods access data members *x* and *y* directly

Example of Three-level class hierarchy (*Cylinder* ➔ *Circle* ➔ *Point*)

```
// File: Circle.java

public class Circle extends Point
{

        protected double radius;
        public static final double PI = 3.14159;

        public Circle()
        {
                super(0, 0);
                setRadius(0);
        }

        public Circle(double r, double a, double b)
        {
                super(a, b);
                System.out.println("Circle Constructor for " + this.toString());
                setRadius(r);
        }

        public void finalize()
        {
                System.out.println("Circle Finalizer for " + this.toString());
                super.finalize();
        }

        public void setRadius(double r)
        {
                radius = (r >= 0.0 ? r : 0.0);
        }

        public double getRadius()
        {
                return radius;
        }

        public double area()
        {
                return PI * radius * radius;
        }

        public String toString()
        {
                return "Center = " + super.toString() + " ; Radius = " + radius;
        }
}
```

Calling the superclass **toString**() method

Example of Three-level class hierarchy (*Cylinder* ➔ *Circle* ➔ *Point*)

```
public class Cylinder extends Circle
{
        protected double height; // height of Cylinder

        public Cylinder(double h, double r, double a, double b)
        {
                super(r, a, b);
                setHeight(h);
        }

        public void setHeight(double h)
        {
                height = (h >= 0 ? h : 0);
        }

        public double getHeight()
        {
                return height;
        }

        public double area()
        {
                return 2 * super.area() + 2 * Circle.PI * radius * height;
        }

        public double volume()
        {
                return super.area() * height;
        }

        public String toString()
        {
                return super.toString() + "; Height = " + height;
        }
}
```

Call to Circle **area**() method

Using a Circle class *constant*

Call to **Circle** toString() method which, in turn, calls **Point** tostring() method!

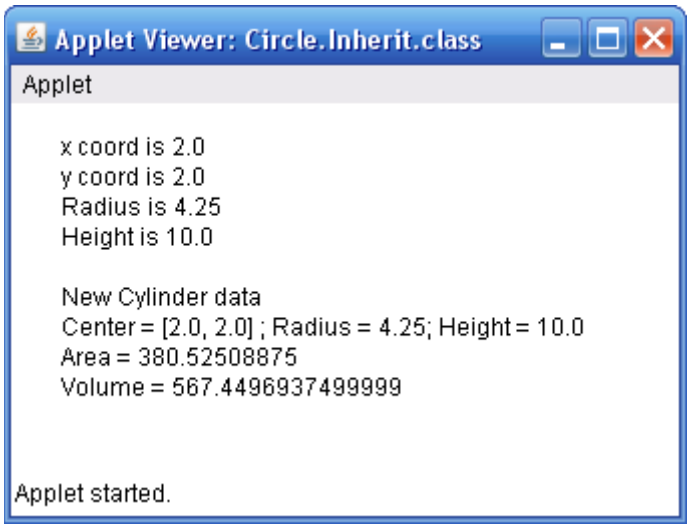
```
public class Inherit extends Applet
{
        private Cylinder c;

        public void init()
        {
                c = new Cylinder(5.7, 2.5, 1.2, 2.3);
        }

        public void paint()
        {
                g.drawString("x coord is " + c.getX(), 25, 25);
                g.drawString("y coord is " + c.getY(), 25, 40);
                g.drawString("Radius is " + c.getRadius(), 25, 55);
                g.drawString("Height is " + c.getHeight(), 25, 70);

                c.setHeight(10);
                c.setradius(4.25);
                c.setPoint(2, 2);

                g.drawString("New Cylinder data", 25, 100);
                g.drawString(c.toString(), 25, 115);
                g.drawString("Area = " + c.area(), 25, 130);
                g.drawString("Volume = " + c.volume(), 25, 145);
        }
}
```

Accessing Point class methods

Accessing Circle class method

Accessing Cylinder class method

Output



Polymorphism: Allows Dynamic Behavior of Objects

We can process *generically* (as a superclass objects) objects which are derived from a common superclass.

Adding new classes, which can be processed *generically*, can be done with little or no modification of the superclass or the client code

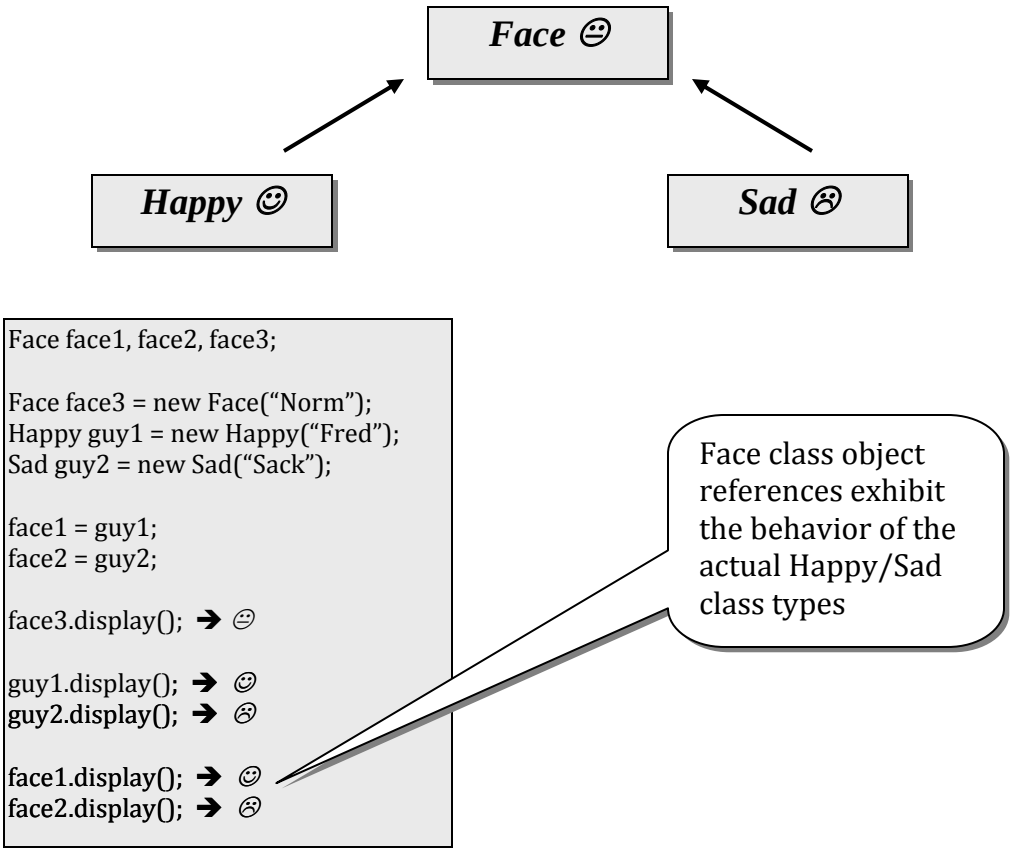
Client code may need modification only in areas where specific references to the *new* data/behaviors of the subclass are made

If a superclass and a subclass have defined a method with the same profile, and a superclass reference is attached to a subclass instance (object), then the reference to the *superclass* method will invoke the overridden method in the subclass.

This approach requires the *common denominator* of behaviors to be implemented as overriding methods down the chain of inheritance.

Suppose, the superclass, Face has a method **display**(), which is overridden in each of the subclasses,
then we can expect the following behavior (as shown on the following page).

Abstract Polymorphism Example



Example of how *Polymorphism* appears

Dynamic Binding

Any time the compiler cannot determine the code which will execute on a particular method call, the connection of the cal to the code (*binding*) will be performed at runtime. This is referred to as *dynamic method binding*, or *late binding*.

Example: Both face1 and face2 objects references are of class Face. However the method calls invoke Happy and Sad class display() methods.

Below, the following in *not* considered late binding because there is no question as to which method will execute.

final Methods and Classes

There are three applications for the keyword *final*

1. final variables or objects behave like constants
2. final methods cannot be overridden in subclasses
3. final classes cannot be extended by further subclasses

Abstract and Concrete Classes

Abstract classes are classes which cannot be instantiated. No objects of this type can be defined. Such classes are useful for treating related, but different classes generically.

Abstract classes are defined using the keyword *abstract*

Abstract classes can have methods which are defined with code. Later, subclasses of the abstract class may make calls to the defined method, as long as this method is not overridden in the subclass.

```
public abstract class Person { .....

        ......  }
```

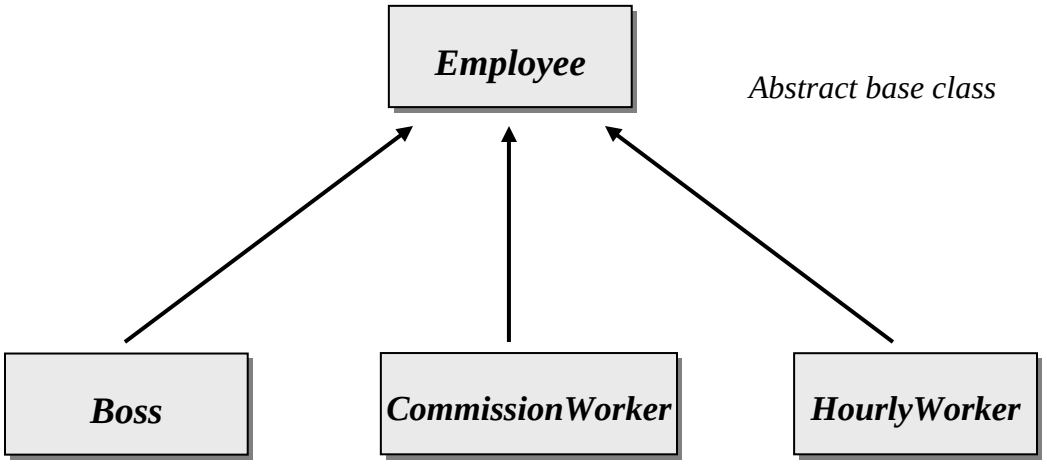
Abstract methods are created also by adding the keyword *abstract* to the header. The abstract keyword tells the compiler that no code will follow for the method.

```
public abstract class Person { .....
        .......

        public abstract void display();
        ......  }
```

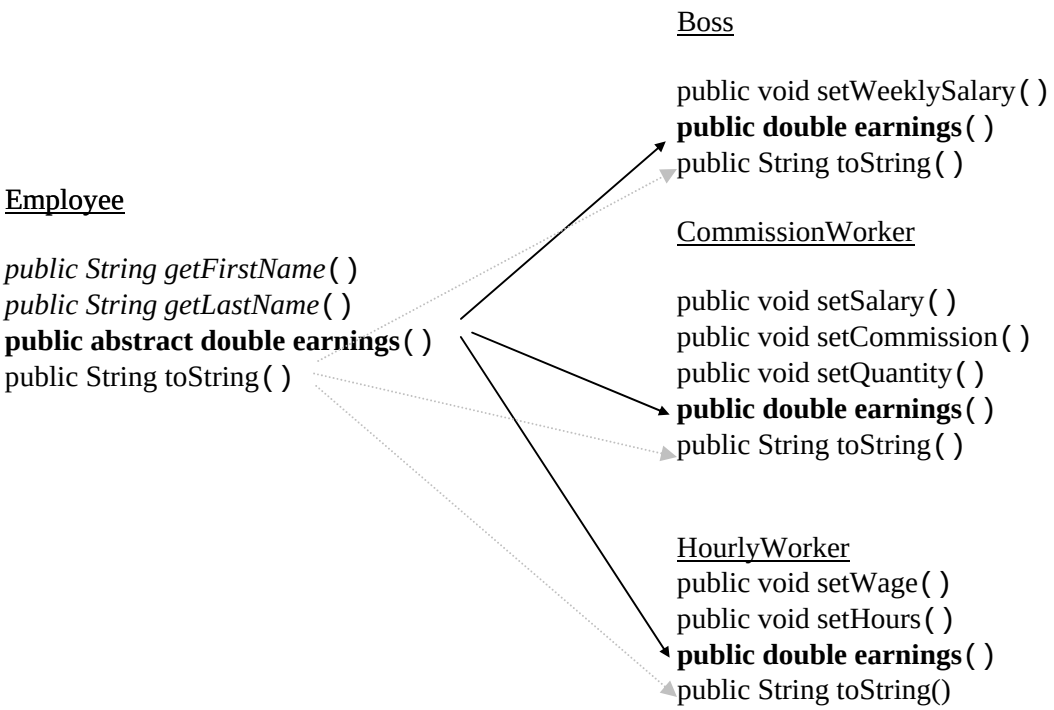
Useful class hierarchies often can have an abstract base class, even several layers of abstract base classes, before instantiable *concrete* classes are derived.

> ***Concrete*** classes are those classes which *can* be instantiated with objects. All classes we have defined so far have been *concrete* classes.

Example of a Class Hierarchy with an Abstract Base Class



Concrete classes



Arrows indicate overriding of the abstract method **earnings( )**

Full Example of the Employee Polymorphic Code

```
// File: Employee.java
public abstract class Employee
{
        private String firstName;
        private String lastName;

        public Employee(String first, String last){
                firstName = new String(first);
                lastName = new String(last);
        }

        public String getFirstName(){
                return new String(firstName);
        }

        public String getLastName(){
                return new String(lastName);
        }

        public abstract double earnings();
}
```

```java
// File: Boss.java
public final class Boss extends Employee
{
        private double weeklySalary;

        public Boss(String first, String last, double s){
                super(first, last);
                setWeeklySalary(s);
        }

        public void setWeeklySalary(double s){
                weeklySalary = (s > 0 ? s : 0);
        }

        public double earnings(){
                return weeklySalary;
        }

        public String toString(){
                return "Boss: " + getFirstName() + ' ' + getLastName();
        }
}
```

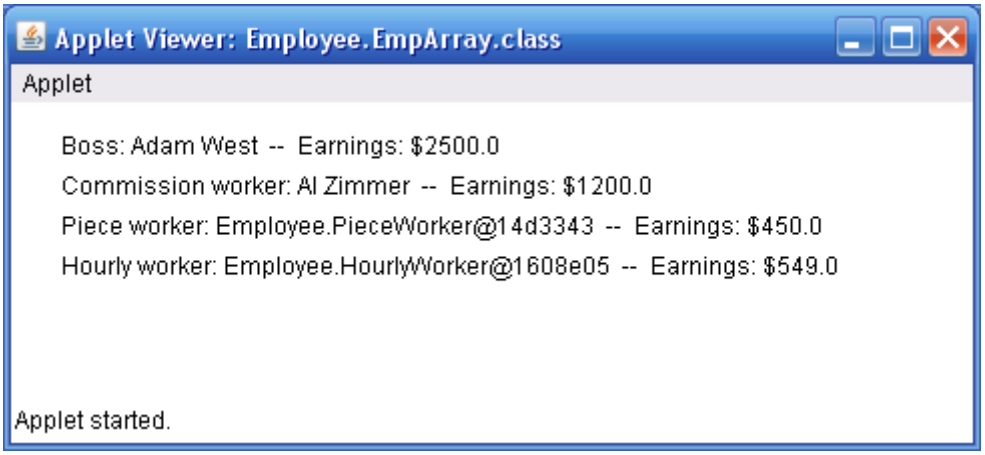
```java
// File: CommissionWorker.java
public final class CommissionWorker extends Employee
{
        private double salary;
        private double commission;
        private int quantity;

        public CommissionWorker(String first, String last, double s, double c, int q)
        {
                super(first, last); // call base-class constructor
                setSalary(s);
                setCommission(c);
                setQuantity(q);
        }

        public void setSalary(double s)
        {
                salary = (s > 0 ? s : 0);
        }

        public void setCommission(double c)
        {
                commission = (c > 0 ? c : 0);
        }

        public void setQuantity(int q)
        {
                quantity = (q > 0 ? q : 0);
        }

        public double earnings()
        {
                return salary + commission * quantity;
        }

        public String toString()
        {
                return "Commission worker: " + getFirstName() + ' ' + getLastName();
        }
}
```

```java
public final class PieceWorker extends Employee
{
	private double wagePerPiece; // wage per piece output
	private int quantity; // output for week

	// Constructor for class PieceWorker
	public PieceWorker(String first, String last, double w, int q){
		super(first, last); // call superclass constructor
		setWage(w);
		setQuantity(q);
	}

	// Set the wage
	public void setWage(double w){
		wagePerPiece = (w > 0 ? w : 0);
	}

	// set the number of items output
	public void setQuantity(int q){
		quantity = (q > 0 ? q : 0);
	}

	// Determine the PieceWorker's earnings
	public double earnings(){
		return quantity * wagePerPiece;
	}

	public String toString(){
		return "Piece worker: " + super.toString();
	}
}
```

```java
public final class HourlyWorker extends Employee
{
	private double wage; // wage per hour
	private double hours; // hours worked for week

	// Constructor for class HourlyWorker
	public HourlyWorker(String first, String last, double w, double h){
		super(first, last); // call supercalss constructor
		setWage(w);
		setHours(h);
	}

	// Set the wage
	public void setWage(double w){
		wage = (w > 0 ? w : 0);
	}

	// Set the hours worked
	public void setHours(double h){
		hours = (h >= 0 && h < 168 ? h : 0);
	}

	// Get the HourlyWorker's pay
	public double earnings(){
		return wage * hours;
	}

	public String toString(){
		return "Hourly worker: " + super.toString();
	}
}
```

Driver Program and Output of the Abstract Employee Class Program

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

public class EmpArray extends Applet
{
        private final int max = 4;
        private Employee emp, worker[];
        private Boss boss;
        private CommissionWorker comm;
        private PieceWorker piece;
        private HourlyWorker hourly;

        public void init(){
                boss = new Boss("Adam", "West", 2500);
                comm = new CommissionWorker("Al", "Zimmer", 600, 3, 200);
                piece = new PieceWorker("Ellen", "Jones", 3.75, 120);
                hourly = new HourlyWorker("Sam", "Adams", 15.25, 36);
                worker = new Employee[max];
                worker[0] = boss;
                worker[1] = comm;
                worker[2] = piece;
                worker[3] = hourly;
        }

        public void paint(Graphics g){
                int y = 25;
                for (int i = 0; i < max; i++, y += 20)
                        g.drawString(worker[i].toString() + "  --  Earnings: $"
                                        + worker[i].earnings(), 25, y);
        }
}
```



Output of the polymorphic program