Class Handouts for CS 1B

Program Guidelines

Before you turn in a program, reread it with the following points in mind. You will receive points off for not following these Program Guidelines for every program this quarter:

1. Every Application program you turn in must be accompanied by a printout of the RUN as well as the program listing. The RUN must have been created with the program it accompanies. Only the exact source code file can be submitted, never copy the contents of a source code file to some other type of file.

2 Programs must be submitted through the Assignments tool in Etudes. Programs will not be accepted by email.

3. You must write a clear, concise comment at the top of every class telling what one object of the class represents, stores and manipulates. This comment must be in JavaDoc format. Also, you must write a comment for every method telling what that method does with the object and with its parameters, whether it changes the object or the parameters, and whether any input or output is done in the method. All of these comments must be in the JavaDoc format.

4. Choose identifiers for your variables that remind the reader what that variable stands for (e.g. Use "Price" or "Tax", instead of "X" or "Y").

5. **Indent** the body of a class or method inside { and }. The book shows one example of an indentation scheme, whichever you choose, it is imperative that you are consistent.

6. Keep revising your program until you have the **simplest** solution you can find (e.g. do not use any more variables than you need.)

7. Keep numbers (except 0, 1, 10, 100) out of your reusable functions. Define "final" variables instead, named with all CAPITALS, as described on page 137 of the text.

8. Do not use **break** or **continue**, because they create unstructured code. Exception: you can use **break** with the **switch** statement ONLY.

9. In some textbooks, instance variables are not declared private. This practice defeats the purpose of object-oriented programming, which was designed to insulate the calling program from the details of an object. In the programs you write for this class, you are required to declare each and every instance variable **private**.

10. **Test** your program to make sure that every feature works in many different situations.

11. It is always important to test the class you are developing by writing test code that calls your development class from the outside. Software engineers have found that it saves a LOT of time to write this test code FIRST. Therefore, before you develop a new class, you need to write the test code containing the main() that does nothing but test the development class. Do not include the main() that tests your development class inside the new class itself. You know that your development class is complete when the test code works.

12. Every class that you write must be in its own file. The name of the file will be the same as the name of the class with ".java" at the end of it. You will always write a test class and a development class (see 11. above), so you will always have a minimum of 2 files to upload to Etudes for each assignment.

12. No method should be longer than 30 lines.

class Employee

This source code file (Employee.java) shows the appropriate programming style for this class. It is also the conventional style used in industry because it makes writing, calling, debugging and modifying code easy.

```
/**
 *  One object of class Employee stores the personal information for one Employee.
 */
public class Employee {

   private String name;
   /**
    * sets the value of name to "newName"
    */
   public void setName(String newName) {
      name = newName;
   }
   /**
    * returns the current value of name
    */
   public String getName() {
      return name;
   }

   private String ssn;
   /**
    * sets the value of ssn to "newSsn"
    */
   public void setSsn (String newSsn) {
      ssn = newSsn;
   }
   /**
    * returns the current value of ssn
    */
   public String getSsn() {
      return ssn;
   }

   private double salary;
   /**
    * sets the value of salary to "newSalary"
    */
   public void setSalary(double newSalary) {
      salary = newSalary;
   }
   /**
    * returns the current value of salary
    */
   public double getSalary() {
      return salary;
   }

   /**
    • default constructor for Employee class object which assigns default values
    • for all class properties
    */
   public Employee() {
      this.setName("null");
      this.setSsn("000-00-0000");
      this.setSalary(0);
   }
   /**
    * parameterized constructor for Employee class object which assigns values for all
    * class properties, based on the values sent into the parameters.
    */
   public Employee(String name, String ssn, double salary) {
      this.setName(name);
      this.setSsn(ssn);
      this.setSalary(salary);
   }

   /**
    * Returns a String containing all the data stored in this object.
    */
   public String toString() {
      String result = "Name: " + this.getName()
            + "\nSSN#: " + this.getSsn()
            + "\nSalary: $" + this.getSalary() + "\n";
      return result;
   }
}
```

Reference vs. Value

A **reference** stores the address of a variable, a **value** stores the value of the variable itself.

When you declare a variable of one of the primitive data types (like int) or when you declare a String variable, you get a variable itself, where you can store a value immediately. On the other hand, when you declare a variable of a class type, you get only a reference, and you must call **new** to get space to store an actual value.

```
class Main
{
   public static void main(String args[])
   {
      int i;                                  /* allocates space for an int value */
      i = 25;

      String str;                          /* allocates space for a string value */
      str = "hello";

      Jabberwock monster;
                           /* Only allocates space for a reference to a Jabberwock */

      monster = new Jabberwock();          /* allocates space for a Jabberwock value */
      monster.color = "orange";
   }
}

// class Jabberwock would have to be defined here
```

Regarding Parameter Passing

You may have contrasted the terms *reference* and *value* regarding function parameters in other programming languages (including C++).

In Java, all objects are passed by reference and all primitive types are passed by value. The programmer has no choices to make when declaring parameters to methods. Therefore, for example, there is no way to write a method in Java that modifies an int variable. Also, there is no way to pass an object by value to method.

A String goes both ways

```
/**
 *  Shows that you can declare and use String objects either as values or as
 *  references to Objects that you call new on. They work fine both ways.
 */
class Main
{
   public static void main(String args[])
   {
      int i;                                     /* allocates space for an int value */
      i = 25;

      String str;                           /* declares a variable of type String */ */
      str = "hello";                    /* Treating a String like a primitive int */
      System.out.println(str);

      String strObject;                /* Declares a reference to a String object */
      strObject = new String ("HELLO");     /* Explicitly calling new to allocate */
      System.out.println(strObject);
   }
}
```

Assigning References

```
/**
 *  Uses the Point class defined in the standard library to show that assigning
 *  references does NOT copy objects, but instead makes the two references point
 *  to exactly the same object. Note that only one new object is allocated.
 */

import java.awt.Point;

public class Main
{  public static void main (String args[])
   {  Point pt1, pt2;
      pt1 = new Point (100, 100);
      pt2 = pt1;

      pt1.x = 200;
      pt1.y = 300;

      System.out.println("Point1: " + pt1.x + ", " + pt1.y);
      System.out.println("Point2: " + pt2.x + ", " + pt2.y);
   }
}

/****************** OUTPUT ******************/
Point1: 200, 300
Point2: 200, 300
**********************************************/
```

Equality of References

/**
 * **This program shows that references are only equal if they point to the exact same object.**
 * **Two references are NOT equal if they merely point to two different, but equivalent, objects.**
 */

```
import java.awt.Point;

public class working
{  public static void main(String args[])
   {  Point pt1;
      Point pt2;
      pt1 = new Point ( 100, 200);
      pt2 = new Point (100, 200);
      if ( pt1 == pt2)
         System.out.println("They are equal!");
      else
         System.out.println("They aren't equal!");
      // doesn't work because "==" compares references

      if ( pt1.equals( pt2))
         System.out.println("They are equal!");
      else
         System.out.println("They aren't equal!");
      // works because class Point defines equals() method to compare the
      // contents of the objects.

      pt1= pt2;
      if ( pt1 == pt2)
         System.out.println("After assignment, they are equal!");
      else
         System.out.println("After assignment, they still aren't equal!");
   }
}

/************* OUTPUT ******************
They aren't equal!
They are equal!
After assignment, they are equal!
******************************************/
```

Inheriting Constructors

```
public class Main
{
   public static void main(String args[])
   {  Mammal m;
      m = new Mammal();
      System.out.println(m);
      System.out.println("----------------------");
      m = new Mammal(3);
      System.out.println(m);
      System.out.println("----------------------");

      Dog d;
      d = new Dog();
      System.out.println(d);
      System.out.println("----------------------");
      d = new Dog(4,"Fido");
      System.out.println(d);
      System.out.println("----------------------");
   }
}

public class Mammal
{  private int age;
   private int getAge() { return age; }
   private void setAge( int newAge ) { age = newAge; }

   private void initialize( int a )
   {  this.setAge(a);
   }

   public Mammal()
   {  this.initialize(0);
      System.out.println("Mammal() constructor called");
   }
   public Mammal(int a)
   {  this.initialize(a);
      System.out.println("Mammal(int) constructor called");
   }
   public String toString()
   {  return "Mammal's age is "+ this.getAge();
   }
}
```

```
public class Dog extends Mammal
{  private String name;
   private String getName() { return name; }
   private void setName ( String newName ) { name = newName; }

   private void initialize( String n )
   {  this.setName(n);
   }

   public Dog()                       //Implicitly calls the default Mammal constructor
   {  this.initialize("Fido");
      System.out.println ("Dog() constructor called");
   }
   public Dog(int a, String n)
   {  super(a);                               //Explicitly calls the Mammal constructor
                                                                          // with an int
      this.initialize (n);
      System.out.println("Dog(int, String) constructor called");
   }

   public String toString()
   {  return super.toString() + ", and name is " + this.getName();
   }
}

/***********************************

Mammal() constructor called
Mammal's age is 0
----------------------
Mammal(int) constructor called
Mammal's age is 3
----------------------
Mammal() constructor called
Dog() constructor called
Mammal's age is 0, and name is Fido
----------------------
Mammal(int) constructor called
Dog(int, String) constructor called
Mammal's age is 4, and name is Fido
----------------------

***********************************/
```

The length of an array vs. the length of a String

```
/**
 * length is a variable in the Array class and is a method for the class String.
 * Note the two different syntaxes that result.
 */

class Main
{
   public static void main(String args[])
   {
      String strArray[];
      strArray = new String [5];              /* allocates 5 string values */
      strArray [0] = "hi!";

      System.out.println ("Number of strings in strArray:"+ strArray.length );

      System.out.println ("Length of the 0th string is "+ strArray[0].length() );

   }
}

/****************************
Number of strings in strArray:5
Length of the 0th string is 3
****************************/
```

Casting primitive types

```
class Main
{
   public static void main(String args[])
   {  int i = 15;
      double d = 1.7;
//    i = d;                           "Explicit cast needed to convert double to int"
      d = i;                       // no problem here because all integers ARE doubles
//    same goes for parameter passing
      System.out.println ("d= " + d);

//    i = 34.7;                        "Explicit cast needed to convert double to int"
      i = (int) 34.7;
      System.out.println ("i= " + i);

      char ch = 'A';
      i = ch;
      System.out.println ("i= " + i);
      System.out.println ("ch= " + ch);

      i = 66;
//    ch = i;                                                    "Explicit cast needed"
      ch = (char) i;
      System.out.println ("i= " + i);
      System.out.println ("ch= " + ch);
   }
}

/************* OUTPUT **************
d= 15
i= 34
i= 65
ch= A
i= 66
ch= B
```

OBJECTS IN THE SUBCLASS INHERIT EVERY METHOD IN THE SUPERCLASS

```
/**
 * Therefore, you needn't cast an object of a subclass in order to call a method of
 * the superclass. When one class is subclassed from another, it is the child of the
 * other, and as such it is created after the parent. When the child class is
 * designed the parent class is known. Just as software version 2.0 knows about
 * everything and so it fully compatible with software version 1.0, the child class
 * (subclass) knows everything about and is fully compatible with the public methods
 * of the parent class (superclass).
 */

class Main
{    public static void main(String args[])
     {    Dog d;
          d = new Dog();
          d.printDog();
          d.printMammal();            // d is a Dog object, so it gets acess to all Mammal methods.
          Mammal m;
          m = new Mammal();
          m.printMammal();
//        m.printDog();               Compiler ERROR- "printDog" can't be called with just any Mammal object.
//        ((Dog)m).printDog();        Runtime ERROR –"m.name" doesn't exist.
//                                    Because it is possible the "m" might be pointing to a Dog (SOME Mammals ARE
     }    //                          Dogs) this compiles but it doesn't run. We'll see more on this later.
}

class Mammal
{    private int age;
     private int getAge() { return age; }
     private void setAge( int newAge) { age = newAge; }

     public void printMammal()
     {    System.out.println ("doing the Mammal thang at age = " + getAge());
     }
}

class Dog extends Mammal
{    private String  name;
     private String getName() {   return name; }
     private void setName ( String  newName )  {   name = newName; }

     public void printDog()
     {    System.out.println ( getName() + " is a dog");
     }
}

/****** OUTPUT
null is a dog
doing the Mammal thang at age = 0
doing the Mammal thang at age = 0
*******/
```

A SUPERCLASS REFERENCE CAN REFER TO A SUBCLASS OBJECT

```
class Main
{  public static void main(String args[])
   {  Mammal m;                           // A Mammal reference must point to a Mammal.
      m = new Dog();                        // A Dog IS a Mammal, so a Mammal reference
//                                                                   CAN point to a Dog.
      m.printMammal();         // Dogs can call Mammal methods because Dogs ARE Mammals
//    m.printDog();                Compiler ERROR-"printDog" not found in class Mammal.

      ((Dog)m).printDog();                 // Works because in this case "m" IS referring
//                                          to a Dog. If "m" were referring to a Mammal
//                                that wasn't a dog, this would cause a run-time error.
//                                                   Note: precedence of cast operator,
//                                                      member operator require parens.
      Dog d;
//    d = new Mammal();                   Won't work because there are Mammals that are
//                                                                            NOT Dogs.
   }
}

class Mammal
{  private int age;
   public void printMammal()
   {  System.out.println ("doing the Mammal thang at age = " + age);
   }
}

class Dog extends Mammal
{  private String name;
   public void printDog()
   {  System.out.println ( name + " is a dog");
   }
}

/****** OUTPUT
doing the Mammal thang at age = 0
null is a dog
*******/
```

Method Overriding

```
/**
 *   Shows method overriding: Once you override a method, there is no way for the
 *   main() to get back to the superclass's version of it, even by casting.
 */

class Main
{  public static void main(String args[])
   {  Mammal m;
      m = new Mammal();
      m.print();

      Dog d = new Dog();
      ((Mammal)d).print();                                 // Still calls Dog::print()

      m = new Dog();
      m.print();                                  // At runtime, the VM sees that m is
                                                                // referring to a Dog.
      ((Mammal)m).print();                                 // there's no way to get back
                                                        // to Mammal's version of print().
   }
}
class Mammal
{  public void print()
   {  System.out.println("Inside Mammal's version of print()");
   }
}

class Dog extends Mammal
{  public void print()
   {  System.out.println("Inside Dog's version of print()");
   }
}

/****** OUTPUT
Inside Mammal's version of print()
Inside Dog's version of print()
Inside Dog's version of print()
Inside Dog's version of print()
*******/
```

==>> Moral: When you override a class' method, make sure that your new method performs the same logical task as the superclass' method.

Inheritance Quiz

1) If class Beta extends class Alpha, class Beta is called the _________class and class Alpha is call the ___________class.

2) An object of a ______class can be treated as an object of its corresponding ______class.

3) (true/false) A subclass object is also an object of its superclass.

4) (true/false) A subclass object may call any public method in its superclass.

5) Circle the lines in the main that would not compile:

```
class Main
{  public static void main(String args[])
   {  Rectangle r;
      r = new Rectangle();
      r.printRect();
      r.printBillBoard();

      BillBoard b;
      b = new BillBoard();
      b.printBillBoard();
      b.printRect();

      b = new Rectangle();
      r = new BillBoard();
      r.printRect();
      r.printBillBoard();
      ((BillBoard)r).printBillBoard();
   }
}

class Rectangle
{  private int length;
   private int width;

   public void printRect()
   {  System.out.println ("Length = " + length + ", Width = " + width);
   }
}

class BillBoard extends Rectangle
{  private String message;

   public void printBillBoard()
   {  System.out.println (message);
      super.printRect();
   }
}
```

Inheritance Quiz, cont'd

6) What is the output of the following program?

```
class Main
{  public static void main(String args[])
   {  Mammal m;
      m = new Mammal();
      m.print();

      Dog d;
      d = new Dog();
      d.print();

      m = new Dog();
      m.print();

      ((Mammal)m).print();
   }
}
class Mammal
{  private int age;
   public void doMammal()
   {  System.out.println ("doing the Mammal thang at age = " + age);
   }
   public void print()
   {  System.out.println("Inside Mammal's version of print()");
   }
}

class Dog extends Mammal
{  private String name;
   public void doDog()
   {  System.out.println ( name + " is a dog");
   }
   public void print()
   {  System.out.println("Inside Dog's version of print()");
   }
}
```

Arrays

How does the Reference vs. Value concept apply to arrays? When you declare an array of **int**s or **String**s, you get an array of values. But when you declare an array of objects, you only get an array of references to those objects, and you must call **new** to get space to store an actual object's value.

```
class Main
{
   public static void main(String args[])
   {  int intArray[];
      intArray = new int [5];                     //allocates space for 5 int values
      intArray[0] = 25;

      String strArray[];
      strArray = new String [5];                //allocates space for 5 string values
      strArray [0] = "hello";

      Jabberwock monsterArray[];
      monsterArray = new Jabberwock [5];                    //Only allocates space for
                                                         //5 references to Jabberwocks
      monsterArray[0] = new Jabberwock();       // Allocates space for one Jabberwock
                                                               // value (e.g. object)
      monsterArray[0].color = "orange";
      monsterArray[1] = new Jabberwock();  // Allocates space for another Jabberwock

      monsterArray[1].color = "blue";
   }
}
```

A METHOD THAT RETURNS THE OBJECT THAT CALLS IT

```
class Main
{  public static void main(String args[])
   {  Person minnie = new Person("Minnie", 41);
      minnie.older();
      minnie.print();

      (minnie.older()).print();            // the method's returned value allows 1 step
   }
}

class Person
{  private String name;
   private String getName() { return name; }
   private void setName ( String n ) { name = n; }

   private int age;
   private int getAge() { return age; }
   private void setAge( int newAge ) { age = newAge ; }

   public Person ()
   {  initialize ("default", 0 );
   }

   public Person ( String n, int a )
   {  initialize (n, a);
   }

   public void initialize(String n, int a)
   {  setName(n);
      setAge (a);
   }

   public void marry (Person bride)
   {  bride.setName (bride.getName()+ " " +this.getName());
   }

   /**
    * Adds 1 to the age of the Person that calls it, and returns this Person
    * that calls it
    */
   public Person older ()
   {  setAge(getAge()+1);
      return this;
   }

   public void print()
   {  System.out.println(name() + "'s age is " + getAge());
   }
}
```

Starry Night

```
/**
 *  An Application that draws randomly colored stars at random locations on
 *  the screen.
 */
import java.awt.event.*;
import java.awt.*;
import java.util.*;]
import javax.swing.JFrame;

public class GUIApp extends JFrame
{     private final int HEIGHT = 200;   // the height of the Frame
      private final int WIDTH = 200;   // the width of the Frame
      private final int MAXRGB = 255;
      private final int NUMTIMES = 1000;

      private Random randomObject;  // Holds the object that calculates
                                   // and returns our pseudorandom numbers.
      private Random getRandomObject() { return randomObject; }
      private void setRandomObject( Random newR ) { r = newR; }

      private void initialize()
      {     setRandomObject (new Random());
            this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
            this.setSize(WIDTH, HEIGHT);
            this.setVisible(true);
      }

      public GUIApp()
      {    this.initialize();
      }

      public void paint( Graphics g )
      {     for ( int count = 0; count < NUMTIMES; count ++ )
            {     g.setColor (new Color (Math.abs(getRandomObject().nextInt())% MAXRGB,
                                   Math.abs(getRandomObject().nextInt())% MAXRGB,
                                  Math.abs(getRandomObject().nextInt())% MAXRGB) );
                  g.drawString( "*", Math.abs(getRandomObject().nextInt())% WIDTH,
                                    Math.abs(getRandomObject().nextInt())% HEIGHT );
            }
      }
}

public class Main
{     public static void main(String args[])
      {     GUIApp g = new GUIApp();
      }
}
```

Displaying an Image

```
/**
 *  Displays a gif file from disk onto a Frame. Only works for an Application.
 *  For an Applet you don't need Toolkit because you can use Applet.getImage()
 */
import java.awt.*;
import java.awt.event.*;

public class GUIApp extends JFrame
{  private Image eye;
   private Image getEye() { return eye; }
   private void setEye( Image newEye ) { eye = newEye; }

   public void paint( Graphics screen )
   {  screen.drawImage( getEye(), 20, 20, this );
   }
   private void loadImage()
   {  MediaTracker mt = new MediaTracker(this);
      Toolkit tk = Toolkit.getDefaultToolkit();
      setEye( tk.getImage("eye.gif"));
      mt.addImage(getEye(),1);
      try
      {  mt.waitForAll();
      }   catch (Exception e)
         { }
   }
   private void initialize()
   {  loadImage();
      this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
      this.setVisible(true);
   }
   public GUIApp()
   {    this.initialize();
   }

}
```

The Image could take a few seconds to load, because it is coming from disk. **getImage()** starts the loading process in its own Thread, and therefore the main Thread must wait until the **getImage()** Thread is finished. You can just copy this **loadImage()** method into your own Application if you want.

If you have problems getting the image to display, the problem could be:
1) The external image file is in a format that Java can't recognize.
Use Photoshop to open the image file and save it in either **jpg** or **gif.**

2) The external image file is in the wrong directory.
Copy the file into the same directory as your .java file.

3) The external image file has a different name.
Check the file name by looking at the file with your operating system directory listing to see if it has a n extension or a different spelling, or a different capitalization.

A Reusable Graphical Object

```
/**
 * Tests class Circles, whose objects display a variable number of concentric circles
 * on a variable size canvas.
 */
public class Main extends Applet
{  private final int NUMCANVASES = 16;                // this must be a perfect square

   public void init()
   {  Circles c;
      c = new Circles( 8, 100, 100);                      // Creates a Circles object
      this.add(c);                                           // Adds it to the Applet
   }
}

/**
 *  An object of this class displays a number of concentric Circles on a Canvas.
 *  The calling program specifies the number of concentric Circles,
 *  and the height and width of the Canvas, when it calls the constructor.
 */
public class Circles extends Canvas
{  private int numTimes;                                   // # of concentric circles
   private int getNumTimes() { return numTimes; }
   private void setNumTimes (int newTimes) { numTimes = newTimes; }

   private int xCenter;                 // Center of the circles relative to the Canvas
   private int getXCenter() { return xCenter; }
   private void setXCenter( int newCenter ) { xCenter = newCenter; }

   private int yCenter;
   private int getYCenter() { return yCenter; }
   private void setYCenter( int newCenter ) { yCenter = newCenter; }

   private final int GAP = 5;// Gap between the concentric Circles on 1 Canvas

   /* Constructs a new Circle object, whose "numCircles" concentric Circles
   are displayed in the center of a Canvas of Dimension "h" x "w"  */
   public Circles (int numCircles, int h, int w)
   {  initialize(numCircles, h, w);
   }
   public Circles()
   {  initialize( 1, 10, 10 );
   }
   private void initialize(int numCircles, int h, int w)
   {  setNumTimes( numCircles);
      setSize( h, w);                                       // sizes the canvas h x w
      setXCenter (w/2);
      setYCenter (h/2);
   }

   /* Elegant algorithm by Konstantin Svist. */
   public void paint( Graphics g )
   {  int radius;
      int counter;
      for (counter = getNumTimes(); counter > 0; counter --)
      {  radius = counter * GAP;
         g.drawOval(getXCenter() - radius, getYCenter()-radius, radius*2, radius * 2);
      }
   }
}
```

GUI Application

```
/**
 * This simple JFrame just displays the message "Hello World!". The only way to quit
 * the application is to click the Frame’s close box.
 */

import javax.swing.*;

public class GUIApp extends JFrame
{    private JLabel prompt;
     private JLabel getPrompt() { return prompt; }
     private void setPrompt( JLabel newPrompt ) { prompt = newPrompt; }

     private void initialize()
     {    setPrompt( new JLabel("Hello World! ") );
          setSize(100,50);
          this.add(getPrompt());
          this.setVisible(true);
          this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
     }

     public GUIApp()
     {    this.initialize();
     }
}


/* This main just tests the class GUIApp, above. */

public class Main
{  public static void main(String args[])
   {
      GUIApp g = new GUIApp();
   }
}
```

ActionListener

```
/**
 * A GUI Application that interacts with the user via push Buttons and ActionListener
 * interface.
 */
import java.awt.event.*;
import javax.swing.*;
import java.awt.FlowLayout;

public class GUIApp extends JFrame implements ActionListener
{  private JLabel prompt;
   private JButton hello;
   private JButton goodbye;

   private void initialize()
   {  this.setLayout(new FlowLayout());
      this.setSize(300, 200);

      hello = new JButton ("Hello");
      this.add(hello);
      hello.addActionListener(this);

      goodbye = new JButton ("Goodbye");
      this.add(goodbye);
      goodbye.addActionListener(this);

      this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
      this.setVisible(true);
}

   public GUIApp()
   {    this.initialize();
   }

   public void actionPerformed (ActionEvent evt)
   {  if (prompt != null)
         this.remove(prompt);
      if (evt.getSource() == hello )
      {  prompt = new JLabel("Hello, world! ") ;
         this.add(prompt);
      }
      else
      {  prompt = new JLabel("Goodbye, world!");
         this.add(prompt);
      }
      validate();
   }
}
```

// Note: All 3 of these Listener examples are tested with the same main(), that is on the previous page.

ItemListener

```
/* You must handle radio button events with ItemListener interface. */
import java.awt.FlowLayout;
import java.awt.event.*;
import javax.swing.*;
public class GUIApp extends JFrame implements ItemListener
{       private JLabel prompt;
        private ButtonGroup choices;
        private JRadioButton hello;
        private JRadioButton goodbye;

        private void initialize()
        {       this.setLayout(new FlowLayout());
                this.setSize(300, 200);

                hello = new JRadioButton ("Hello");
                this.add(hello);
                hello.addItemListener(this);

                goodbye = new JRadioButton ("Goodbye");
                this.add(goodbye);
                goodbye.addItemListener(this);

                choices = new ButtonGroup();
                choices.add(hello);
                choices.add(goodbye);

                prompt = new JLabel("                              ");
                this.add(prompt);

                this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
                this.setVisible(true);
        }

        public GUIApp()
        {   this.initialize();
        }

        public void itemStateChanged (ItemEvent ie)
        {       if (ie.getSource() == hello )
                {       prompt.setText( "Hello, world! " );
                }
                else
                {       prompt.setText("Goodbye, world!" );
                }
                validate();
        }
}
```

MouseListener

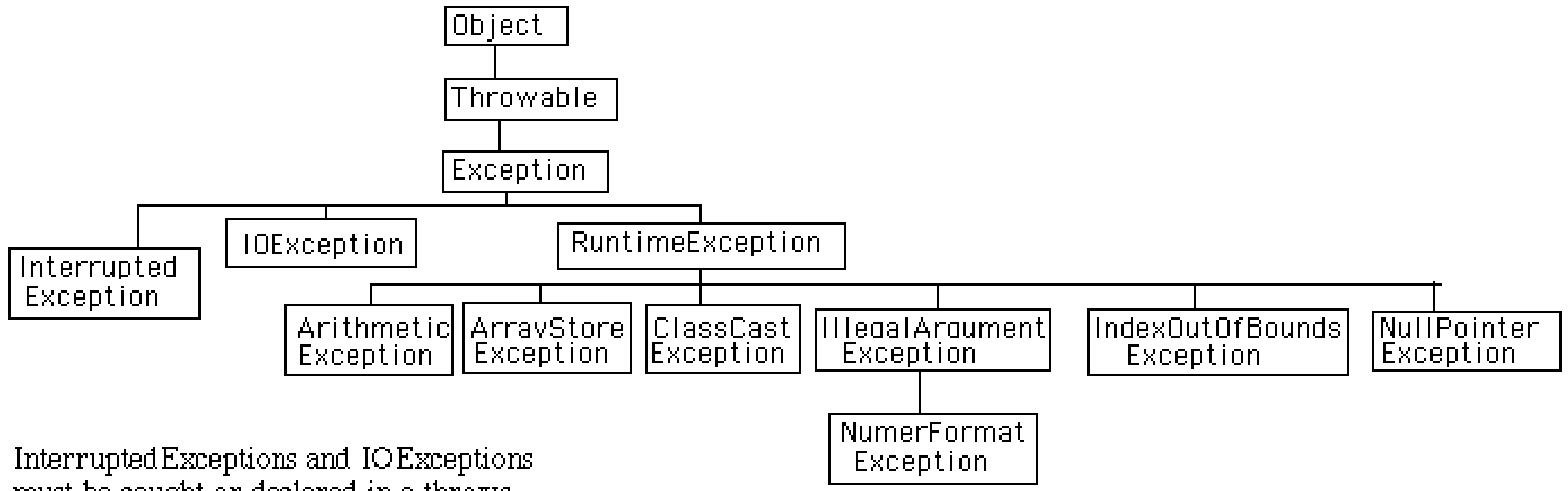
```java
/**
 * An App that reacts to the user's mouse clicks. When the user clicks on a
 * point on the App's Frame, the App displays the coordinates where the user clicked.
 */

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

public class GUIApp extends JFrame implements MouseListener
{       private JLabel coordinates;

        private void initialize()
        {  coordinates = new JLabel("Click the mouse");
                this.setLayout(new FlowLayout());
                this.setSize(200, 200);
                this.add(coordinates);

                this.setVisible(true);
                this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
                addMouseListener(this);
        }

        public GUIApp()
        {    this.initialize();
        }

        public void mouseClicked (MouseEvent me)
        {       coordinates.setText("("+ me.getPoint().x + ", " + me.getPoint().y+ ")");
        }
        public void mouseEntered (MouseEvent me)
        { }
        public void mouseExited (MouseEvent me)
        { }
        public void mousePressed (MouseEvent me)
        { }
        public void mouseReleased (MouseEvent me)
        { }
}
```

Steps to react to Mouse movement:

1) Implement MouseListener Interface
2) Implement mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased
3) addMouseListener(this)

Getting User Input

```java
/** Gets user input using the ActionListener interface in an Applet. Prompts the user
 *  to type a number of circles, reads the user's response from a TextField, and
 *  displays the chosen number of circles on the Applet window using the Canvas
 *  Circles object from the handout called "A Reusable Graphical Object"
 */

import java.awt.*;
import java.awt.event.*;                                   // Required for ActionListener
import java.applet.Applet;

public class Main extends Applet implements ActionListener
{  private Circles currentCircles;
   private Label prompt;
   private TextField numCircles;
   private int userNum;

   public void init()
   {
      prompt = new Label("Type the number of circles you want and press Return:");
      add(prompt);

      userNum = 5;                                   // default value for number of circles

      numCircles = new TextField ( Integer.toString(userNum), 3);
      add(numCircles);                          // Adds the textField to the Applet window
      numCircles.addActionListener(this);
      // The call to addActionListener makes the actionPerformed() method
     // be called automatically when the user presses <return> in the TextField.
   }

   /**
    * This method is called because the user typed <return> into the numCircles
    * TextField. numCircles is the only component that has an actionListener.
    */
   public void actionPerformed (ActionEvent evt)
   {     userNum =Integer.parseInt(numCircles.getText());     // converts String to int
      if (currentCircles != null)
         remove (currentCircles);
      currentCircles = new Circles
                        (userNum ,this.getSize().width, this.getSize().height);
      add (currentCircles);// adds the new canvas "currentCircles" to this Applet
      currentCircles.repaint();
   }
}
```



InterruptedExceptions and IOExceptions
must be caught or declared in a throws

RuntimeExceptions don't need to be caught

Throwing an Exception

```
/** Asks the user for a numerator and a denominator and waits for the user to press
 *  the "Divide" button. When the user clicks the "Divide" button, this Applet calls
 *  a method which throws an Exception if the denominator is 0. The call to this
 *  method is inside a try/catch block.
 */

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

public class Main extends Applet implements ActionListener
{  Label promptForNum;
   Label promptForDen;
   TextField numerator;
   TextField denominator;
   Button go;
   Label result;

   public void init()
   {  promptForNum = new Label ("Please type the numerator:");
      add(promptForNum);

      numerator = new TextField (10));
      add (numerator);

      promptForDen = new Label ("Please type the denominator:");
      add (promptForDen);

      denominator = new TextField (10);
      add (denominator);

      go = new Button ("Divide"));
      add (go);
      go.addActionListener(this);// The call to addActionListener makes actionPerformed()
                           // method be called automatically when the user presses the go Button.
   }
```

```
public void actionPerformed ( ActionEvent evt )
   {  if (evt.getSource() == go)
      {  double num = Double.valueOf (numerator.getText()).doubleValue();
         double den = Double.valueOf (denominator.getText()).doubleValue();
         if (result != null)
            remove (result);
         try
         {  in.readLine(str);
            result = new Label(Double.toString (divide(num, den)));
         } catch (IOException ioe)
           {   result = new Label (ioe.getMessage());
            }
           catch (Exception e)
            {  result = new Label (e.getMessage());
            }
         add (result);
         validate();                       // Asks the Applet to validate, => repaint()
      }
   }


   /**
    * returns n/d
    */
   double divide (double n, double d) throws Exception
   {  if ( d == 0)
         throw ( new Exception ("Attempt to divide by zero."));
      return n/d;
   }
}                                                     // end of definition of class Main
```

Notes:

- Inside the method "actionPerformed()" if the call to "divide()" does not throw an Exception, then the catch block is skipped.

- Inside the method "divide()", the throw statement causes:
  1) a new Exception object to be created, which stores the message inside of it.
  2) execution of the method "divide()" to terminate.
  3) control to be passed back to the caller, to the catch block inside the method "actionPerformed()".
  4) the new Exception object is passed back to the catch block, where e.getMessage() is used to get the message that was stored in the Exception when it was created back in the "divide()" method.

- Use Exception handling only when a method is unable to complete its task for reasons it cannot control.

- It is advantageous to separate the error handling code from the main line of the program.

- If your code has to perform an operation that may encounter an error:
  1) Put the operation in a separate method.
  2) Define that method to throw an Exception of some type.
  3) Put the call to that method in a try/catch block.

DEFINING AN EXCEPTION CLASS

```
/** Defines an Exception class called "DivideByZeroException", a subclass of
 *  ArithmeticException. The class Main tests the new Exception class by creating an
 *  Applet with a numerator and denominator to be filled in by the user. The Applet
 *  then prints out the Exception's message if the user types in 0 for the
 * denominator.
 */

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

public class Main extends Applet implements ActionListener
{  /*~~~~~~~~~ This class is all the same as on the previous page, except
      ~~~~~~~~ method divide(), below, throws a different type of Exception:

   double divide (double n, double d) throws DivideByZeroException
   {  if ( d == 0)
         throw new MyDivideByZeroException ("Attempt to divide by zero.");
      return n/d;
   }
}                                                     // end of definition of class Main



/**
 * An object of this class represents an Exception. There are two
 * constructors, both of which call the constructor for the super class
 * with an error message.
 */
class MyDivideByZeroException extends ArithmeticException
{
   public MyDivideByZeroException ()
   {
      super ("Attempted to divide by zero");
   }

   public MyDivideByZeroException (String message)
   {
      super (message);
   }
}
```

Exception Quiz

What is the output of the following program? Don't use your compiler to get the answer! This is a good sample midterm exam question, and you will not be able to use your compiler on the midterm.

```
public class ExceptionReview
{
  public static void main(String args[])
  {  int x;
     try
     {   x = foo (10);
     } catch (Exception e)
       {   System.out.println ("Caught an exception: " + e);
           x = 99;
       }
     System.out.println (x);
  }

  public static int foo (int x) throws Exception
  {
    System.out.println ("foo started with " + x);
    int temp = bar (x);
    System.out.println ("foo returning " + temp);
    return temp;
  }

  public static int bar (int y) throws Exception
  {
    System.out.println ("bar started with " + y);
    if (y>0)
       throw new Exception ("just a test");
    System.out.println ("when is this executed?");
    return y;
  }
}
```

Animated Applet

```
/**
 *  Animates an asterisk by moving it diagonally across the Applet window
 */
import java.awt.*;
import java.applet.Applet;

public class Main extends Applet implements Runnable
{  private int height;                             // The height of the Applet window
   private int width;                               // The width of the Applet window

   private int x;                              // The current x position on the Applet
   private int y;                              // The current y position on the Applet

   private Thread runner;                        // Holds the thread for the animation

   public void start()
   {  System.out.println("start() called");
      if (runner==null)
      {  runner = new Thread(this);        // Connects Main with the Thread "runner"
         runner.start();                                         // Calls Main.run()
      }
   }

   public void stop()
   {  System.out.println("stop() called");
   }

   public void run()
   {  System.out.println("run() called");
      while( runner!=null )                // Causes the animation to be continuous
      {  repaint();                                                // Calls paint()
         try
         {  Thread.sleep(500);                // allows the image to stay on screen
         } catch (InterruptedException e) {}
      }
   }
   public void init()
   {  System.out.println("init() called");
      width = this.getSize().width;
      height = this.getSize().height;
      x = 10;  y = 10;
   }

   public void paint( Graphics screen )
   {  System.out.println("paint() called");
      screen.drawString( "*", x % width, y % height );
      x++;    // this will eventually overflow the ints
      y++;    // what is a better solution?
   }
}

/**** OUTPUT ON NEXT PAGE  *****/
```

```
/**** OUTPUT *************   explanation  *****
init() called               because Applet is loaded
paint() called              because Applet is loaded
start() called              because Applet is loaded
run() called                because Thread is started
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
stop() called               because I chose Applet - Suspend
paint() called              because I chose Applet - Resume
start() called              because Thread is started
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
stop() called               because I chose Applet - Restart
init() called               because I chose Applet - Restart
paint() called              because I chose Applet - Restart
start() called              because I chose Applet - Restart
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
paint() called              called by repaint()
stop() called               because I closed the Applet window
```

Creating Threads

There are two ways to implement Threads in Java:

1) Define your class to implement the Runnable interface,
as in "Animated Applet" on the previous page..
This means that your class must include in the class header:
" implements Runnable"
and your class must include a definition for:
public void run() { }

An alternate way that only works with Applications, not Applets, is to:

2) Define your class to extend the Thread class,
as in "A Thread that Prints" on the next page

A Thread that Prints

```
/**
 * Tests the class PrintThread by creating and running a number of Thread objects.
 */
public class Main
{  public static void main (String args[])
   {  PrintThread thread1;
      PrintThread thread2;
      PrintThread thread3;
      PrintThread thread4;

      thread1 = new PrintThread();
      thread2 = new PrintThread();
      thread3 = new PrintThread();
      thread4 = new PrintThread();

      thread1.start();    // DOESN'T HAVE TO RETURN BEFORE CONTINUING THIS SEQUENCE
      thread2.start();
      thread3.start();
      thread4.start();
   }
}

/**
 *  A Thread class. Each object created from this class starts a process running
 *  and then puts that process to sleep.
 */
class PrintThread extends Thread
{
   private int sleepTime;

   public PrintThread ()
   /* Constructor */
   {  sleepTime = (int) (Math.random() * 5000);
      System.out.println("Just constructed Name: " + getName() +
                        sleep: " + sleepTime);
   }

   /** Executes the Thread, which just puts the Thread to sleep
    *  and then prints its name.
    */
   public void run()
   {  try
      {  Thread.sleep (sleepTime );
         //causes currently executing Thread to sleep
      }
      catch (Exception e)
      {  System.out.println ("Error in " + getName());
      }
      System.out.println ( getName() + " just woke up!" );
   }
}
```

See output on next page

```
/***************  OUTPUT
Name: Thread-3; sleep: 4146
Name: Thread-4; sleep: 2041
Name: Thread-5; sleep: 4047
Name: Thread-6; sleep: 4010
Thread-4
Thread-6
Thread-5
Thread-3
**************************/
```

Multiple Threads from an Applet

```
/**
 * Moves an asterisk ‘*’ and an at sign ‘@’ across the window at the same time.
 * Demonstrates multiple Threads controlled by a single Applet.
 */
import java.awt.*;
import java.applet.Applet;

public class MyApplet extends Applet
{  private CharMover star_;                       // the instance of Thread that moves
   private CharMover star() { return star_; }             //the star across the screen
   private void star( CharMover chmv ) { star_ = chmv; }

   private CharMover at_;                         // the instance of Thread that moves
   private CharMover at() { return at_; }             // the at sign across the screen
   private void at( CharMover chmv ) { at_ = chmv; }

   public void init()
   {  star(new CharMover('*', 0, 0, this, 1));
      at (new CharMover('@', this.getSize().width, 0, this, -1));
   }

   public void start()
   {  star().start();                                            // Calls CharMover.run()
      at().start();                                                   // for each Thread
   }

   public void paint( Graphics screen )          // Displays one frame of the animation
   {  screen.drawString( at().ch()+"", at().x() , at().y());
      screen.drawString( star().ch()+"", star().x() % this.getSize().width,
                         star().y() % this.getSize().height);
   }
}

/**
 * One object of class CharMover stores a Thread that repeatedly paints a
 * character "cha", starting at "initX", "initY", in a diagonal left to right or
 * right to left ("newDir"=1 for for left to right, "newDir"=-1 for right to
 * left) on the Applet window "own" that instantiated it.
 */
public class CharMover extends Thread
{  private char ch_;
   public char ch() {return ch_; }
   private void ch (char newch) { ch_ = newch; }

   private int x_;
   public int x() {return x_; }
   private void x (int newX) { x_ = newX; }

   private int y_;
   public int y() {return y_; }
   private void y (int newY) { y_ = newY; }

   private Applet owner_;
   private Applet owner() { return owner_; }
   private void owner(Applet newApp ) { owner_ = newApp; }
   private int direction_; // 1 means left to right, -1 means right to left
   private int direction() { return direction_ ; }
   private void direction( int newDirection ) { direction_ = newDirection;}

   /**
    *  Constructs an object that, starting at (initX, initY), moves cha across
    *  the window of the Applet own. If newDir == 1, then "cha" moves diagonally
    *  left to right, if newDir == -1, then "cha" moves diagonally right to left.
    */
   public CharMover(char cha, int initX, int initY, Applet own, int newDir)
   {  ch(cha);
      x(initX);
      y(initY);
      owner(own);
      direction(newDir);
   }

   /**
    * Repeatedly moves the character across the screen.
    */
   public void run()
   {  while( true )                            // Causes the animation to be continuous
      {  if (direction() == 1)
         {  x(x()+1);
            x( x()% owner().getSize().width);
            y(y() + 1);
         }
         else
         {  x(x() - 1);
            if ( x() <= 0 )
            {  x(owner().getSize().width);
            }
            y(y() + 1);
         }
         y( y()% owner().getSize().height);
         try
         {  Thread.sleep(50);
         } catch (InterruptedException e) {}
         owner().repaint();                        // Calls the Applet's paint() method
      }
   }
}
```

Copying a File

```
/** Copies a text file from "data" to "data.copy". Both files reside
 *  in the same directory as the ".class" file.
 */

import java.io.*;                    // Required for FileInputStream and FileOutputStream

class Main
{  public static void main(String args[])
   {  FileInputStream inFile;
      FileOutputStream outFile;

      int i;
      try
      {  inFile = new FileInputStream("data");          // throws FileNotFoundException
         outFile = new FileOutputStream ("data.copy");             //throws IOException

         i = inFile.read();
         while (i != -1)
         {  outFile.write(i);                                      //throws IOException
            i = inFile.read();                                     //throws IOException
         }

         inFile.close();                                           //throws IOException
         outFile.close();                                          //throws IOException
      } catch (FileNotFoundException fnfe)
         {  System.out.println ("Error opening input file: " + fnfe.getMessage());
         }
        catch (IOException ioe)
         {  System.out.println ("Error opening output file, reading or writing file: "
                                                                   + ioe.getMessage());
         }
   }
}
```

Notes:

- To read or write you must have a Stream Object. To read and write a file, you must have a FileInputStream object and a FileOutputStream object.
- You can tell which methods throw which Exceptions by looking up the method in your Java reference.
- Even though I could have placed each method call in its own try/catch block, it breaks up the code less and makes it more readable to place all method calls in one large try/catch block, then catch the specific Exceptions separately.
- The Exceptions had to be caught in the order given, because
  FileNotFoundException extends (is more specific than) IOException
  and IOException extends Exception

class FileCopyTask

```
/** Contains a class 'FileCopyTask' for objects whose methods can then:
 *  1) getSource(): Displays an open file dialog box for the user to choose a file.
 *  2) makeCopy(): Makes a copy of the chosen file, and saves it to the disk right
 *  next to the original, with the same name + "copy" appended to it.
 */

import java.awt.*;                                         // for class FileDialog
import java.io.*;                        // for class FileInputStream, FileOutputStream

class FileCopyTask
{  private File fileObject_;             // contains the source for the copy operation.
   private File fileObject() { return fileObject_; }
   private void fileObject( File fo ) { fileObject_ = fo; }

   /** Displays a file dialog, sets "fileObject_" to the user's chosen file.
    *  If the user types a filename that doesn't exist, or cancels out of the dialog
    *  box, this returns false. Otherwise, it returns true.
    */
   public boolean getSource()
   {  Frame frameObject;
      FileDialog dialogObject;

      frameObject = new Frame();
      dialogObject = new FileDialog ( frameObject,
                                      "Which file would you like to copy?",
                                       FileDialog.LOAD);

      dialogObject.setVisible();
      // This modal dialog pauses execution until the user chooses a file.

      if (dialogObject.getFile() == null )
      {  System.out.println ("You hit Cancel!");
         return false;
      }

      // Set the fileObject with the user's chosen directory and filename.
      fileObject( new File(dialogObject.getDirectory(), dialogObject.getFile()));

      if ( ! fileObject().exists() )
      {  System.out.println ("You typed a filename that doesn't exist!");
         return false;
      }
      return true;
   }
```

```
  /**
   *  Makes a copy of the "fileObject_" and saves it to disk in the same
   *  directory as the original, with "copy" appended to the filename.
   */
   public boolean makeCopy()
   {
      FileInputStream inFile;
      FileOutputStream outFile;

      int i = 0;
      try
      {  inFile = new FileInputStream (fileObject());
         outFile = new FileOutputStream (fileObject().toString() + "copy");
         i = inFile.read();
         while (i != -1)
         {  outFile.write(i);
            i = inFile.read();
         }
         inFile.close();
         outFile.close();
      } catch (Exception e)
         {   System.out.println("Exception while opening,reading or writing files. ");
           return false;
         }
      return true;
   }
}
```

Linked List

```
/**
 * One object of this class represents one node in the linked list, class List,
 * below.
 */
class ListNode
{  private Object data;                          // Holds whatever data is in each node
   private Object getData() { return data; }
   private void setData ( Object d ) { data = d; }

   // don't want these to be public!!
   private ListNode next; // Holds the reference to the next node in the List
   public ListNode getNext() { return next; }
   public void setNext( ListNode ln ) { next = ln; }

   public ListNode (Object d)
   {  initialize (d, null);
   }
   public ListNode (Object d, ListNode n)
   {  initialize(d, n);
   }
   private void initialize(Object d, ListNode n)
   {  setData(d);
      setNext(n);
   }
   public String toString ()                              // Returns the data in one node
   {  return ""+getData();             // this automatically calls getData().toString()
   }
}

/**
 * One object of this class represents a linked list of ListNode objects, above.
 */
public class List
{  private ListNode first;                 // A reference to the first node in the list
   private ListNode getFirst() { return first ; }
   private void setFirst( ListNode newFirst) { first = newFirst; }

   public List ()
   {  setFirst( null);
   }

   /**
    * Makes a new node containing "d" and links it to the end of List.
    */
   public void addAtEnd (Object d)
   {  if (getFirst() == null)
         setFirst (new ListNode ( d ));
      else if (getFirst().getNext() == null)
         getFirst().setNext(new ListNode ( d ));
      else
      {  ListNode temp = getFirst().getNext();
         while (temp.getNext() != null)   // We know that (temp != null)
            temp = temp.getNext();
         temp.setNext( new ListNode ( d ));
      }
   }
   public String toString()
   {  ListNode temp = getFirst();
      String value = "";
      while (temp != null)
      {  value = value + " " + temp;
         temp = temp.getNext();
      }
      return value;
   }
}
```

class System

/* The following is defined in the standard Java class library java.lang. I have copied just a few of the variables and functions in class System. All methods and variables in class System are static and class System cannot be instantiated. */

```
public final class System extends Object
{
        public static final PrintStream out;
        /* println() is a member of class PrintStream */

        public static void exit(int status);
        /* Terminates the currently running Java Virtual Machine. The argument serves as a status
        code; by convention, a nonzero status code indicates abnormal termination. */

        public static void gc();
        /* Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling
        unused objects in order to make the memory they currently occupy available for quick reuse.
        When control returns from the method call, the Java Virtual Machine has made a best effort to
        reclaim space from all discarded objects.*/

        public static String getProperty(String key);
        /* See next page for (key, returned value) pairs */
}
```

```
/**
 * Shows how to tell the operating system, API implementation, and JVM version.
 */

public class Main
{  public static void main(String args[])
   {
      System.out.println("os.name = " + System.getProperty("os.name"));
      System.out.println("API implementation:");
      System.out.println("java.version = " + System.getProperty("java.version"));
      System.out.println("java.vendor = " + System.getProperty("java.vendor"));
      System.out.println("Java Virtual Machine:");
      System.out.println("java.vm.version = " +System.getProperty("java.vm.version"));
      System.out.println("java.vm.vendor = " + System.getProperty("java.vm.vendor"));
   }
}

/******* OUTPUT **********
os.name = Mac OS X
API implementation:
java.version = 1.4.2_05
java.vendor = Apple Computer, Inc.
Java Virtual Machine:
java.vm.version = 1.4.2_38
java.vm.vendor = "Apple Computer, Inc."
***************************/
```

```
public static String getProperty(String KEY);
```

KEY	value returned from getProperty(KEY)
`java.version`	Java Runtime Environment version
`java.vendor`	Java Runtime Environment vendor
`java.vendor.url`	Java vendor URL
`java.home`	Java installation directory
`java.vm.specification.version`	Java Virtual Machine specification version
`java.vm.specification.vendor`	Java Virtual Machine specification vendor
`java.vm.specification.name`	Java Virtual Machine specification name
`java.vm.version`	Java Virtual Machine implementation version
`java.vm.vendor`	Java Virtual Machine implementation vendor
`java.vm.name`	Java Virtual Machine implementation name
`java.specification.version`	Java Runtime Environment specification version
`java.specification.vendor`	Java Runtime Environment specification vendor
`java.specification.name`	Java Runtime Environment specification name
`java.class.version`	Java class format version number
`java.class.path`	Java class path
`java.ext.dirs`	Path of extension directory or directories
`os.name`	Operating system name
`os.arch`	Operating system architecture
`os.version`	Operating system version
`file.separator`	File separator ("/" on UNIX)
`path.separator`	Path separator (":" on UNIX)
`line.separator`	Line separator ("\n" on UNIX)
`user.name`	User's account name
`user.home`	User's home directory
`user.dir`	User's current working directory

Serializable

```
/* If a class implements the interface Serializable, then its objects can be written
to a file and read back with no conversion necessary. Interface Serializable is an
empty Interface, with no methods to implement. The only addition I made to the simple
class Rectangle is to add "implements Serializable". This program works even if class
Rectangle were to have a member variable that was a reference to another class. */

import java.io.*;

public class Main
{  public static void main(String args[])
   {  Rectangle r = new Rectangle(3,4);
      Rectangle s = new Rectangle(5,6);

      // 1) Write the 2 Rectangle objects to the file
      FileOutputStream outFile;
      ObjectOutputStream outObject;
      try
      {  outFile = new FileOutputStream ("data");
         outObject = new ObjectOutputStream(outFile);
         outObject.writeObject(r);
         outObject.writeObject(s);
         outFile.close();
         outObject.close();
      } catch (IOException ioe)
      {  System.out.println ("Error writing objects to the file: "+ ioe.getMessage());
      }

      // 2) Set the Rectangle references to null
      r = s = null;

      // 3) Read the 2 Rectangle objects back in from the file
      FileInputStream inFile;
      ObjectInputStream inObject;
      try
      {  inFile = new FileInputStream("data");
         inObject = new ObjectInputStream(inFile);
         r = (Rectangle)inObject.readObject();
         s = (Rectangle)inObject.readObject();
         inFile.close();
         inObject.close();
      } catch(IOException ioe)
      {  System.out.println ("Error reading from the file: " + ioe.getMessage());
      }
      catch (ClassNotFoundException cnfe)
      {  System.out.println ("Error in casting to Rectangle: " + cnfe);
      }

      // 4) Print the 2 Rectangle objects to see if they were read in correctly
      System.out.println (r);
      System.out.println (s);
   }
}
```

Vector and Iterator

```
import java.util.Vector;
import java.util.Iterator;

class TrivialApplication
{  public static void main(String args[])
   {  Vector v = new Vector();
      System.out.println ("Size of vector = " + v.size());
      System.out.println ("Capacity of vector = " + v.capacity());

      Rectangle r;
      //  v.add(r);    Error! Java won't let you add null references to a vector
      r = new Rectangle(1,2);
      v.add(r);
      System.out.println ("Size of vector = " + v.size());
      System.out.println ("Capacity of vector = " + v.capacity());

      r = new Rectangle (2,3);
      v.add(r);

      System.out.println("The vector now looks like:");
      Iterator i = v.iterator();
      while (i.hasNext())
      {  System.out.println(i.next() );
      }

      r.setRectangle (4,5);
      System.out.println("\nAfter changing r, the vector looks like:");
      i = v.iterator();       // you must get another, not redeclare, the iterator
      while (i.hasNext())
      {  System.out.println(i.next() );
      }

      r = new Rectangle(7,8);
      v.add(r);
      System.out.println("\nUsing another way to print the vector:");
      for (int j= 0; j < 3; j++)    // this only works if we know how many elements
      {  System.out.println(v.get(j) );  // use get() instead of []
      }

      r = (Rectangle)v.firstElement();
      System.out.println("First element is: " + v.firstElement());

      v.remove(0);
      System.out.println("\nAfter removing the 0th Object, the vector looks like:");
      i = v.iterator();       // you must reset, not redeclare, the iterator
      while (i.hasNext())
      {  System.out.println(i.next() );
      }
      boolean isThere;
      isThere = v.contains(r);
      if (isThere)
         System.out.println("Is there");
      else
         System.out.println("isn't there");
   }
}

/* OUTPUT on next page */
```

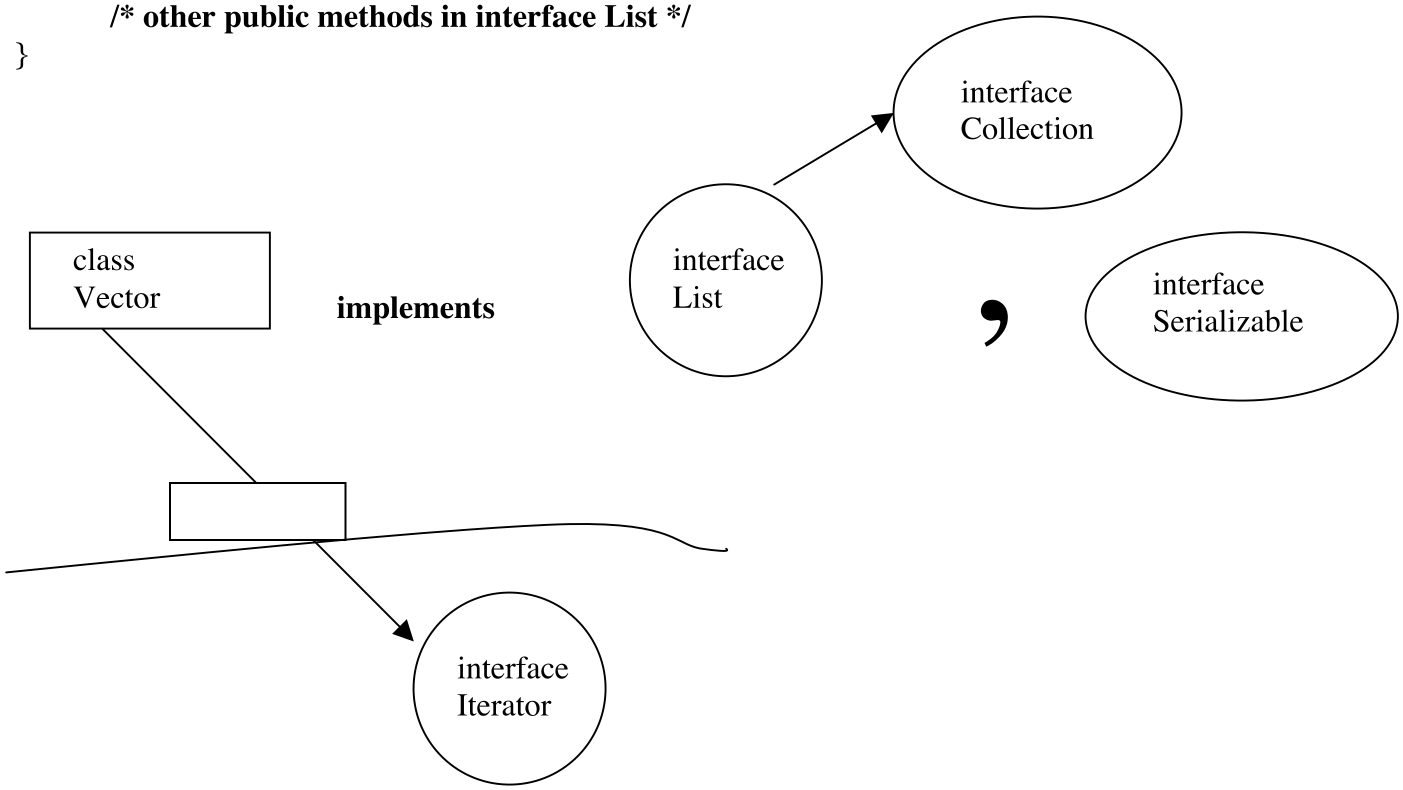
```
/************ OUTPUT ****************
Size of vector = 0
Capacity of vector = 10
Size of vector = 1
Capacity of vector = 10
The vector now looks like:
length=1.0, and width=2.0
length=2.0, and width=3.0

After changing r, the vector looks like:
length=1.0, and width=2.0
length=4.0, and width=5.0

Using another way to print the vector:
length=1.0, and width=2.0
length=4.0, and width=5.0
length=7.0, and width=8.0
First element is: length=1.0, and width=2.0

After removing the 0th Object, the vector looks like:
length=4.0, and width=5.0
length=7.0, and width=8.0
isn't there
*************************************/
```

Java Collections Framework

**~~~~~~~~~~~~~~~~~~~~~~~~ in java.util**

```
class Vector implements List, Serializable
{       public Iterator iterator();          // returns an Iterator for accessing the Vector.
        /* other public methods in class Vector */
}
```

```
interface List extends /*interface*/ Collection
/* class Vector has these methods because Vector implements interface List */
{       public boolean add (Object o);   // returns true if the add() was successful
        /* other public methods in interface List */
}
```



**~~~~~~~~~~~~~~~~~~~~~~~~ in your code:**

```
Vector v = new Vector();
v.add(...);
Iterator i = v.iterator();
```

**~~~~~~~~~~~~~~~~~~~~~~~~ in java.util:**

```
interface Iterator
{       public boolean hasNext();
        public Object next();
}
```

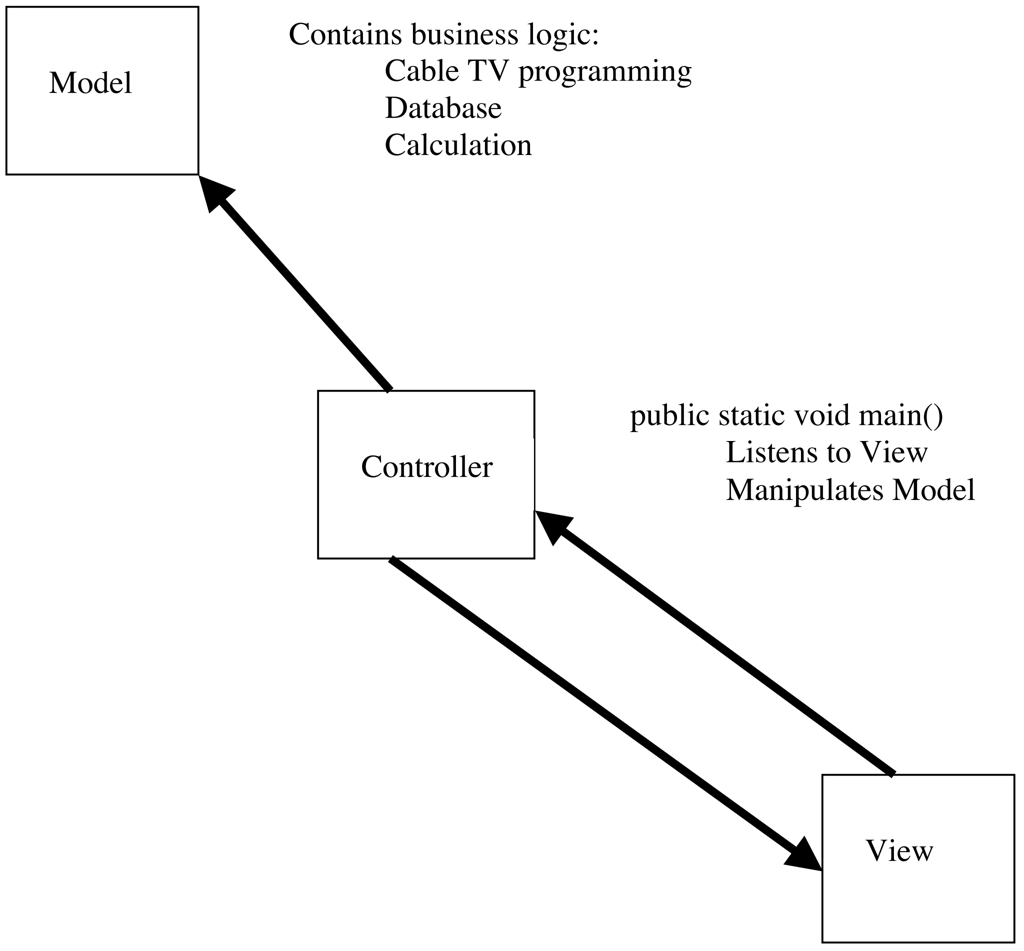
Implementing an Interface

```
/**
 * This interface definition is inside java.lang
 */
public interface Comparable
{  public abstract int compareTo( Object o);
}
```

```
/** This is the class Rectangle from "Program Guidelines". I have omitted
 *  all the private and public members for brevity. I have added "implements
 *  Comparable", and a full method definition for compareTo().
 */
public class Rectangle implements Comparable
{
   . . .

   /** Returns >0 if this object's area is > obj's area. Returns <0 if this
    * object's area is < obj's area. Returns 0 if this object's area == obj's area.
    */
   public int compareTo( Object obj)
   {  return (int)(this.calculateArea() - ((Rectangle)obj).calculateArea());
   }
}
```

```
/** Demonstrates the benfit of using an interface.
 *  Here, because class Rectangle implements Comparable, we can use Arrays.sort()
 *  to sort an array of Rectangles in ascending order.
 */

import java.util.Arrays;

class Main
{  public static void main(String args[])
   {  Rectangle list[];
      list = new Rectangle[10];
      list [0] = new Rectangle (11,2);
      list [1] = new Rectangle (3,4);
      list [2] = new Rectangle (25, 30);
      list [3] = new Rectangle (1, 2);

      System.out.println("before sort():");
      for ( int i = 0; i< 4; i++)
         System.out.println(list[i]);
      Arrays.sort(list, 0 ,4);    // static method inside class Arrays
      System.out.println("--------------");
      System.out.println("after sort():");
      for ( int i = 0; i< 4; i++)
         System.out.println(list[i]);
   }
}
```

Model/View/Controller - diagram



Contains business logic:
- Cable TV programming
- Database
- Calculation

public static void main()
- Listens to View
- Manipulates Model

User Interface
- GUI Frame
- System console
- touch screen
- television set

1) the Controller (class Main, for example) instantiates a View object
2) the View registers the Controller as a Listener
3) the Controller waits, listening
4) when the Controller.actionPerformed() method is called, Controller manipulates the Model

Model/View/Controller - code

Example of the Model/View/Controller organization. The Controller instantiates a Frame and Listens for the user to cause an Event. The Controller calls a method in the Model every time the user clicks a Button in the View

```
/**
 * An object of this class counts the number of times its beep() method is called, and
 * returns a String containing that number each time beep() is called. This usually contains some
 * sort of business model, calculations and data.
 */
class Model
{         private int count;

          public Model ()
          {         count = 0;
          }
          public String beep()
          {         return "beep " + count++;
          }
}

/**
 *  This contain the main(). It conains a reference to both the Model and the View because it
 * controls them both.
 */
import java.awt.*;
import java.awt.event.*;
class Controller implements ActionListener
{         private Model model;
          private View view;

          public Controller()
          {         view = new View(this);
                    model = new Model();
          }

          public static void main(String args[])
          {         // instantiates itself to get its own constructor going
                    Controller ctr = new Controller();
          }

          public void actionPerformed (ActionEvent evt)
          {         // tell the View to show something from the Model
                    view.show(model.beep());
          }
}
```

```
/**
 * One object of this class is a JFrame that contains the GUI Components through which
 * the user interacts.
 */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class View extends JFrame
{       private JButton go;
        private JLabel label;
        private ActionListener controller;

        /**
         * displays "message" on label.
         */
        public void show(String message)
        {       label.setText(message);
                validate();
        }

        /**
         * Creates Components and places them onto the JFrame.
         * Stores the reference to "control" so that actionPerformed() method in class "control"
         * will be called when the user interacts with the Components in this class.
         */
        public View(ActionListener control)
        {       controller = control;
                go = new JButton ("go");
                add(go);
                go.addActionListener(controller);

                label = new JLabel();
                add(label);

                /* call methods in class JFrame */
                this.setLayout(new FlowLayout());
                this.setSize(200,100);
                setVisible(true);
                /* closing the JFrame quits the application */
                this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
        }
}
```

Read/Write Objects

If a class implements the interface Serializable, then its objects can be written to a file and read back with no conversion necessary. Interface Serializable is an empty Interface, with no methods to implement. The only addition I made to the simple class Employee (see page 3 of these handouts) is to add "implements Serializable" to the class header. This program works even if class Employee were to have a member variable that was a reference to another class.

```
import java.io.*;
public class Main
{  public static void main(String args[])
   {  Employee e1 = new Employee (3,4);
      Employee e2 = new Employee (5,6);

      // 1) Write the 2 Employee objects to the file
      FileOutputStream outFile;
      ObjectOutputStream outObject;
      try
      {  outFile = new FileOutputStream ("data");
         outObject = new ObjectOutputStream(outFile);
         outObject.writeObject(e1);
         outObject.writeObject(e2);
         outFile.close();
         outObject.close();
      } catch (IOException ioe)
      {  System.out.println ("Error writing objects to the file: "+
ioe.getMessage());
      }

      // 2) Set the Employee references to null
      e1 = e2 = null;

      // 3) Read the 2 Employee objects back in from the file
      FileInputStream inFile;
      ObjectInputStream inObject;
      try
      {  inFile = new FileInputStream("data");
         inObject = new ObjectInputStream(inFile);
         e1 = (Employee)inObject.readObject();
         e2 = (Employee)inObject.readObject();
         inFile.close();
         inObject.close();
      } catch(IOException ioe)
      {  System.out.println ("Error reading from the file: " +
ioe.getMessage());
      }
      catch (ClassNotFoundException cnfe)
      {  System.out.println ("Error in casting to Employee: " + cnfe);
      }

      // 4) Print the 2 Employee objects to see if they were read in correctly
      System.out.println (e1);
      System.out.println (e2);
   }
}
```

Cloning a Complex Object

An authentic Employee records application would also have to keep track of the address of each Employee. A class Address would be developed, where each Object of class Address would store one street address for an Employee.

In this case, each time a new Employee Object is constructed, the Employee constructor must make a new Address Object, to store the address of the new employee. If later you want to make a copy of that Employee object, should a copy be made of the Address Object inside of the Employee Object being copied also? Only the designer of class Employee can answer that question, i.e. making copies of Objects that contain references to other Objects cannot be done automatically by Java.

If your class contains a method called clone(), which does the right thing when it makes a copy of Objects of your class, then you can say that your class implements the Cloneable interface. The declaration that your class implements Cloneable tells any future class that contains references to Objects of your class, that Objects of your class can be copied correctly.

For example, let's say that you define a class Line where each Object of class Line contains two Point Objects (class Point is defined in the Java standard class library.) When a copy is made of a Line Object, the two Points inside of it should be copied too. We do NOT want the new Line Object to contain references to the same two Point objects that are inside of the original Line Object.

In order to accomplish this, we write a clone() method inside class Line. The clone() method calls the Line constructor, which constructs two new Point Objects. Here is the code:

```
/**
 * Tests class Line and its ability to Clone itself
 */
public class Main {
      public static void main(String[] args) {
            Line l1 = new Line(1,2,3,4);
            System.out.println("l1 = " + l1);
            Line l2;
            l2 = (Line)l1.clone();
    //notice that we are not calling "new", but a new Line Object is being constructed
            System.out.println("l2 = " + l2);
            l2.increment(); // does this change l1 also?
            System.out.println("l1 = " + l1);
            System.out.println("l2 = " + l2);
      }
}

/*
Output with the trivial definition of Line.clone():
l1 = p1 = java.awt.Point[x=1,y=2], and p2 = java.awt.Point[x=3,y=4]
l2 = p1 = java.awt.Point[x=1,y=2], and p2 = java.awt.Point[x=3,y=4]
l1 = p1 = java.awt.Point[x=2,y=3], and p2 = java.awt.Point[x=4,y=5]
l2 = p1 = java.awt.Point[x=2,y=3], and p2 = java.awt.Point[x=4,y=5]
```

```
Output with the non-trivial definition of Line.clone():
l1 = p1 = java.awt.Point[x=1,y=2], and p2 = java.awt.Point[x=3,y=4]
l2 = p1 = java.awt.Point[x=1,y=2], and p2 = java.awt.Point[x=3,y=4]
l1 = p1 = java.awt.Point[x=1,y=2], and p2 = java.awt.Point[x=3,y=4]
l2 = p1 = java.awt.Point[x=2,y=3], and p2 = java.awt.Point[x=4,y=5]
*/

/**
 *  One object of class Line contains two points on the (x,y) plane.
 */
import java.awt.Point;

public class Line implements Cloneable{
      private Point p1;
      private Point p2;

      /**
       * p1 = (x1,y1); p2 = (x2,y2)
       */
      public Line(int x1,int y1,int x2,int y2)
      {      p1 = new Point (x1,y1);
             p2 = new Point (x2,y2);
      }

      /**
       * Returns a String containing the values for p1 and p2
       */
      public String toString()
      {       return "p1 = "+p1 + ", and p2 = " +p2;      }

      /**
       * Returns a new Line object containing the same values as this.
       */
      protected Object clone()
      {
         /* trivial version of clone() doesn't work
               try
               {
                     return super.clone();
               } catch (CloneNotSupportedException c)
               {       return null;
               }
         */

         /* Nontrivial clone() version, calls the Line constructor which makes
            two new Point objects to be contained in this new Line object.
         */
         Object newCopy;
         try
         {     newCopy = super.clone();
         } catch (CloneNotSupportedException c)
         {     return null;
         }
         ((Line)newCopy).p1 = new Point(p1.x, p1.y);
         ((Line)newCopy).p2 = new Point(p2.x, p2.y);
         return newCopy;
      }
	/**
	 * adds 1 to the x and y inside p1 and p2.
	 */
	public void increment() // adds 1 to all x's and y's
	{	p1.x++; // strange how class Point has PUBLIC data fields x and y!
		p1.y++;
		p2.x++;
		p2.y++;
	}
} // end of class Line
```