Chapter 8: Inheritance
Lab Exercises

Topics	**Lab Exercises**
Inheritance	Exploring Inheritance
	A Sorted Integer List
	Test Questions
	Overriding the *equals* Method
Adapter Classes	Extending Adapter Classes
Animation	Rebound Revisited
	Count Down

Exploring Inheritance

File *Dog.java* contains a declaration for a Dog class. Save this file to your directory and study it—notice what instance variables and methods are provided. Files *Labrador.java* and *Yorkshire.java* contain declarations for classes that extend Dog. Save and study these files as well.

File *DogTest.java* contains a simple driver program that creates a dog and makes it speak. Study DogTest.java, save it to your directory, and compile and run it to see what it does. Now modify these files as follows:

1. Add statements in DogTest.java after you create and print the dog to create and print a Yorkshire and a Labrador. Note that the Labrador constructor takes two parameters: the name and color of the labrador, both strings. Don't change any files besides DogTest.java. Now recompile DogTest.java; you should get an error saying something like

   ```
   ./Labrador.java:18: Dog(java.lang.String) in Dog cannot be applied to ()
        {
        ^
   ```
   1 error

   If you look at line 18 of Labrador.java it's just a {, and the constructor the compiler can't find (Dog()) isn't called anywhere in this file.
   a. What's going on? (Hint: What call must be made in the constructor of a subclass?)
      =>

   b. Fix the problem (which really is in Labrador) so that DogTest.java creates and makes the Dog, Labrador, and Yorkshire all speak.

2. Add code to DogTest.java to print the average breed weight for both your Labrador and your Yorkshire. Use the avgBreedWeight() method for both. What error do you get? Why?

   =>

   Fix the problem by adding the needed code to the Yorkshire class.

3. Add an abstract *int avgBreedWeight()* method to the Dog class. Remember that this means that the word *abstract* appears in the method header after *public*, and that the method does not have a body (just a semicolon after the parameter list). It makes sense for this to be abstract, since Dog has no idea what breed it is. Now any subclass of Dog must have an avgBreedWeight method; since both Yorkshire and Laborador do, you should be all set.

   Save these changes and recompile DogTest.java. You should get an error in Dog.java (unless you made more changes than described above). Figure out what's wrong and fix this error, then recompile DogTest.java. You should get another error, this time in DogTest.java. Read the error message carefully; it tells you exactly what the problem is. Fix this by changing DogTest (which will mean taking some things out).

```java
// ****************************************************************
// Dog.java
//
// A class that holds a dog's name and can make it speak.
//
// ****************************************************************
public class Dog
{
    protected String name;

    // ------------------------------------------------------------
    // Constructor -- store name
    // ------------------------------------------------------------
    public Dog(String name)
    {
      this.name = name;
    }

    // ------------------------------------------------------------
    // Returns the dog's name
    // ------------------------------------------------------------
    public String getName()
    {
      return name;
    }

    // ------------------------------------------------------------
    // Returns a string with the dog's comments
    // ------------------------------------------------------------
    public String speak()
    {
      return "Woof";
    }
}
```

```
// ****************************************************************
// Labrador.java
//
// A class derived from Dog that holds information about
// a labrador retriever.  Overrides Dog speak method and includes
// information about avg weight for this breed.
//
// ****************************************************************

public class Labrador extends Dog
{
    private String color; //black, yellow, or chocolate?
    private int breedWeight = 75;

    public Labrador(String name,  String color)
    {
      this.color = color;
    }

    // ------------------------------------------------------------
    // Big bark -- overrides speak method in Dog
    // ------------------------------------------------------------
    public String speak()
    {
      return "WOOF";
    }

    // ------------------------------------------------------------
    // Returns weight
    // ------------------------------------------------------------
    public static int avgBreedWeight()
    {
      return breedWeight;
    }
}
```

```
// ****************************************************************
// Yorkshire.java
//
// A class derived from Dog that holds information about
// a Yorkshire terrier. Overrides Dog speak method.
//
// ****************************************************************

public class Yorkshire extends Dog
{

    public Yorkshire(String name)
    {
      super(name);
    }

    // ------------------------------------------------------------
    // Small bark -- overrides speak method in Dog
    // ------------------------------------------------------------
    public String speak()
    {
      return "woof";
    }

}



// ****************************************************************
// DogTest.java
//
// A simple test class that creates a Dog and makes it speak.
//
// ****************************************************************

public class DogTest
{
    public static void main(String[] args)
    {
      Dog dog = new Dog("Spike");
      System.out.println(dog.getName() + " says " + dog.speak());

    }
}
```

A Sorted Integer List

File *IntList.java* contains code for an integer list class. Save it to your directory and study it; notice that the only things you can do are create a list of a fixed size and add an element to a list. If the list is already full, a message will be printed. File *ListTest.java* contains code for a class that creates an IntList, puts some values in it, and prints it. Save this to your directory and compile and run it to see how it works.

Now write a class SortedIntList that extends IntList. SortedIntList should be just like IntList except that its elements should always be in sorted order from smallest to largest. This means that when an element is inserted into a SortedIntList it should be put into its sorted place, not just at the end of the array. To do this you'll need to do two things when you add a new element:

- Walk down the array until you find the place where the new element should go. Since the list is already sorted you can just keep looking at elements until you find one that is at least as big as the one to be inserted.
- Move down every element that will go after the new element, that is, everything from the one you stop on to the end. This creates a slot in which you can put the new element. Be careful about the order in which you move them or you'll overwrite your data!

Now you can insert the new element in the location you originally stopped on.

All of this will go into your *add* method, which will override the *add* method for the IntList class. (Be sure to also check to see if you need to expand the array, just as in the IntList *add* method.) What other methods, if any, do you need to override?

To test your class, modify ListTest.java so that after it creates and prints the IntList, it creates and prints a SortedIntList containing the same elements (inserted in the same order). When the list is printed, they should come out in sorted order.

```
// ****************************************************************
// IntList.java
//
// An (unsorted) integer list class with a method to add an
// integer to the list and a toString method that returns the contents
// of the list with indices.
//
// ****************************************************************
public class IntList
{

    protected int[] list;
    protected int numElements = 0;

    //-------------------------------------------------------------
    // Constructor -- creates an integer list of a given size.
    //-------------------------------------------------------------
    public IntList(int size)
    {
      list = new int[size];
    }

    //-------------------------------------------------------------
    // Adds an integer to the list.  If the list is full,
    // prints a message and does nothing.
    //-------------------------------------------------------------
    public void add(int value)
    {
      if (numElements == list.length)
          System.out.println("Can't add, list is full");
      else
          {
            list[numElements] = value;
            numElements++;
          }
    }

    //-----------------------------------------------------------
    // Returns a string containing the elements of the list with their
    // indices.
    //-----------------------------------------------------------
    public String toString()
    {
      String returnString = "";
      for (int i=0; i<numElements; i++)
          returnString += i + ": " + list[i] + "\n";
      return returnString;
    }
}



// ***********************************************************
// ListTest.java
//
// A simple test program that creates an IntList, puts some
// ints in it, and prints the list.
//
// ***********************************************************

public class ListTest
{
    public static void main(String[] args)
    {
      IntList myList = new IntList(10);
      myList.add(100);
      myList.add(50);
      myList.add(200);
      myList.add(25);
      System.out.println(myList);
    }
}
```

Test Questions

In this exercise you will use inheritance to read, store, and print questions for a test. First, write an abstract class TestQuestion that contains the following:

- A protected String variable that holds the test question.
- An abstract method *protected abstract void readQuestion()* to read the question.

  Now define two subclasses of TestQuestion, Essay and MultChoice. Essay will need an instance variable to store the number of blank lines needed after the question (answering space). MultChoice will not need this variable, but it will need an array of Strings to hold the choices along with the main question. Assume that the input is provided from the standard input as follows, with each item on its own line:
  - type of question (character, m=multiple choice, e=essay)
  - number of blank lines for essay, number of blank lines for multiple choice (integer)
  - choice 1 (multiple choice only)
  - choice 2 (multiple choice only) ...

  The very first item of input, before any questions, is an integer indicating how many questions will be entered. So the following input represents three questions: an essay question requiring 5 blank lines, a multiple choice question with 4 choices, and another essay question requiring 10 blank lines:

```
3
e
5
Why does the constructor of a derived class have to call the constructor
of its parent class?
m
4
Which of the following is not a legal identifier in Java?
guess2
2ndGuess
_guess2_
Guess
e
5
What does the “final” modifier do?
```

You will need to write *readQuestion* methods for the MultChoice and Essay classes that read information in this format. (Presumably the character that identifies what kind of question it is will be read by a driver.) You will also need to write *toString* methods for the MultChoice and Essay classes that return nicely formatted versions of the questions (e.g., the choices should be lined up, labeled a), b), etc, and indented in MultChioce).

Now define a class WriteTest that creates an array of TestQuestion objects. It should read the questions from the standard input as follows in the format above, first reading an integer that indicates how many questions are coming. It should create a MultChoice object for each multiple choice question and an Essay object for each essay question and store each object in the array. (Since it's an array of TestQuestion and both Essay and MultChoice are subclasses of TestQuestion, objects of both types can be stored in the array.) When all of the data has been read, it should use a loop to print the questions, numbered, in order.

Use the data in *testbank.dat* to test your program.

```
testbank.dat

5
e
5
Why does the constructor of a subclass class have to call the constructor of its
parent class?
m
4
Which of the following is not a legal identifier in Java?
```

```
guess2
2ndGuess
_guess2_
Guess
e
5
What does the “final” modifier do?
e
3
Java does not support multiple inheritance.  This means that a class cannot do
what?
m
3
A JPanel has an addMouseListener method because JPanel is a subclass of
JComponent
JApplet
Object
```

Overriding the *equals* Method

File *Player.java* contains a class that holds information about an athlete: name, team, and uniform number. File *ComparePlayers.java* contains a skeletal program that uses the Player class to read in information about two baseball players and determine whether or not they are the same player.

1. Fill in the missing code in ComparePlayers so that it reads in two players and prints "Same player" if they are the same, "Different players" if they are different. Use the *equals* method, which Player inherits from the Object class, to determine whether two players are the same. Are the results what you expect?

2. The problem above is that as defined in the Object class, *equals* does an address comparison. It says that two objects are the same if they live at the same memory location, that is, if the variables that hold references to them are aliases. The two Player objects in this program are not aliases, so even if they contain exactly the same information they will be "not equal." To make *equals* compare the actual information in the object, you can override it with a definition specific to the class. It might make sense to say that two players are "equal" (the same player) if they are on the same team and have the same uniform number.
   - Use this strategy to define an *equals* method for the Player class. Your method should take a Player object and return true if it is equal to the current object, false otherwise.
   - Test your ComparePlayers program using your modified Player class. It should give the results you would expect.

```
// **********************************************************
// Player.java
//
// Defines a Player class that holds information about an athlete.
// **********************************************************

import java.util.Scanner;

public class Player
{
    private String name;
    private String team;
    private int jerseyNumber;


    //-----------------------------------------------------------
    // Prompts for and reads in the player's name, team, and
    // jersey number.
    //-----------------------------------------------------------

    public void readPlayer()
    {
      Scanner scan = new Scanner(System.in);
      System.out.print("Name: ");
      name = scan.nextLine();
      System.out.print("Team: ");
      team = scan.nextLine();
      System.out.print("Jersey number: ");
      jerseyNumber = Scan.nextInt();
    }

}
```

```
// ***************************************************************
// ComparePlayers
//
// Reads in two Player objects and tells whether they represent
// the same player.
// ***************************************************************
import java.util.Scanner;
public class ComparePlayers
{
    public static void main(String[] args)
    {
      Player player1 = new Player();
      Player player2 = new Player();

      Scanner scan = new Scanner();

      //Prompt for and read in information for player 1

      //Prompt for and read in information for player 2

      //Compare player1 to player 2 and print a message saying
      //whether they are equal

    }
}
```

Extending Adapter Classes

Files *Dots.java* and *DotsPanel.java* contain the code in Listings 7.18 and 7.19 of the text. This program draws dots where the user clicks and counts the number of dots that have been drawn. Save these files to your directory and compile and run Dots to see how it works.

Now study the code in DotsPanel.java. (Dots just creates an instance of DotsPanel and adds it to its content pane.) DotsPanel defines an inner class, DotsListener, that implements the MouseListener interface. Notice that it defines the *mousePressed* method to draw a dot at the click point and gives empty bodies for the rest of the MouseListener methods.

1. Modify the DotsListener class so that instead of implementing the MouseListener interface, it extends the MouseAdapter class. What other code does this let you eliminate? For now, just comment out this code. Test your modified program.

2. We have been using inner classes to define event listeners. Another common strategy is to make the panel itself be a MouseListener, eliminating the need for the inner class. Do this as follows:
   - Modify the header to the DotsPanel class to indicate that it implements the MouseListener interface (it still extends JPanel).
   - Delete the DotsListener class entirely, moving the five MouseListener methods into the DotsPanel class.
   - The DotsPanel constructor contains the following statement:

      ```
      addMouseListener (new DotsListener());
      ```

      This creates an instance of the DotsListener class and makes it listen for mouse events on the panel. This will have to change, since the DotsListener class has gone away. The listener is now the panel itself, so the argument to *addMouseListener* should change to *this*, that is, the current object. So the panel is listening for its own mouse events! Compile and run your modified program; it should behave just as before.

3. When we were using the DotsListener class we saw that it could either implement the MouseListener interface or extend the MouseAdapter class. With the new approach in #2, where the DotsPanel is also a MouseListener, can we do the same thing – that is, can DotsPanel extend MouseAdapter instead of implementing MouseListener? Why or why not? If you're not sure, try it and explain what happens.

```
//********************************************************************
//  Dots.java       Author: Lewis/Loftus
//
//  Demonstrates mouse events.
//********************************************************************

import javax.swing.JFrame;

public class Dots
{
   //-----------------------------------------------------------------
   //  Creates and displays the application frame.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Dots");
      frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add (new DotsPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  DotsPanel.java       Author: Lewis/Loftus
//
//  Represents the primary panel for the Dots program.
//********************************************************************

import java.util.ArrayList;
import javax.swing.JPanel;
import java.awt.*;
import java.awt.event.*;

public class DotsPanel extends JPanel
{
   private final int SIZE = 6;  // radius of each dot

   private ArrayList<Point> pointList;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this panel to listen for mouse events.
   //-----------------------------------------------------------------
   public DotsPanel()
   {
      pointList = new ArrayList<Point>();

      addMouseListener (new DotsListener());

      setBackground (Color.black);
      setPreferredSize (new Dimension(300, 200));
   }

   //-----------------------------------------------------------------
   //  Draws all of the dots stored in the list.
   //-----------------------------------------------------------------
   public void paintComponent (Graphics page)
   {
      super.paintComponent(page);

      page.setColor (Color.green);

      for (Point spot : pointList)
         page.fillOval (spot.x-SIZE, spot.y-SIZE, SIZE*2, SIZE*2);

      page.drawString ("Count: " + pointList.size(), 5, 15);
   }

   //*****************************************************************
   //  Represents the listener for mouse events.
   //*****************************************************************
   private class DotsListener implements MouseListener
   {
      //--------------------------------------------------------------
      //  Adds the current point to the list of points and redraws
      //  the panel whenever the mouse button is pressed.
      //--------------------------------------------------------------
      public void mousePressed (MouseEvent event)
      {
         pointList.add(event.getPoint());
         repaint();
      }
      //-----------------------------------------------------------------
      //  Provide empty definitions for unused event methods.
      //-----------------------------------------------------------------
      public void mouseClicked (MouseEvent event) {}
      public void mouseReleased (MouseEvent event) {}
      public void mouseEntered (MouseEvent event) {}
      public void mouseExited (MouseEvent event) {}
   }
}
```

Rebound Revisited

The files *Rebound.java* and *ReboundPanel.java* contain the program are in Listings 8.15 and 8.16 of the text. This program has an image that moves around the screen, bouncing back when it hits the sides (you can use any GIF or JPEG you like). Save these files to your directory, then open *ReboundPanel.java* in the editor and observe the following:

- The constructor instantializes a Timer object with a delay of 20 (the time, in milliseconds, between generation of action events). The Timer object is started with its *start* method.
- The constructor also gives the initial position of the ball (*x* and *y*) and the distance it will move at each interval (*moveX* and *moveY*).
- The *actionPerformed* method in the *ReboundListener* inner class "moves" the image by adding the value of *moveX* to *x* and of *moveY* to *y*. This has the effect of moving the image to the right when *moveX* is positive and to the left when *moveX* is negative, and similarly (with down and up) with *moveY*. The *actionPerformed* method then checks to see if the ball has hit one of the sides of the panel—if it hits the left side (x <= 0) or the right side (x >= WIDTH-IMAGE_SIZE) the sign of *moveX* is changed. This has the effect of changing the direction of the ball. Similarly the method checks to see if the ball hit the top or bottom.

Now do the following:

1. First experiment with the speed of the animation. This is affected by two things—the value of the DELAY constant and the amount the ball is moved each time.
   - Change DELAY to 100. Save, compile, and run the program. How does the speed compare to the original?
   - Change DELAY back to 20 and change *moveX* and *moveY* to 15. Save, compile, and run the program. Compare the motion to that in the original program.
   - Experiment with other combinations of values.
   - Change *moveX* and *moveY* back to 3. Use any value of DELAY you wish.
2. Now add a second image to the program by doing the following:
   - Declare a second ImageIcon object as an instance variable. You can use the same image as before or a new one.
   - Declare integer variables *x2* and *y2* to represent the location of the second image, and *moveX2* and *moveY2* to control the amount the second ball moves each time.
   - Initialize *moveX2* to 5 and *moveY2* to 8 (note—this image will have a different trajectory than the first—no longer a 45 degree angle).
   - In *actionPerformed*, move the second image by the amount given in the *moveX2* and *moveY2* variable, and add code to check to see if the second image has hit a side.
   - In *paintComponent*, draw the second image as well as the first.

2. Compile and run the program. Make sure it is working correctly.

```
//********************************************************************
//  Rebound.java       Author: Lewis/Loftus
//
//  Demonstrates an animation and the use of the Timer class.
//********************************************************************
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Rebound
{
   //-----------------------------------------------------------------
   //  Displays the main frame of the program.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Rebound");
      frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new ReboundPanel());
      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//********************************************************************
//  ReboundPanel.java       Author: Lewis/Loftus
//
//  Represents the primary panel for the Rebound program.
//********************************************************************
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ReboundPanel extends JPanel
{
   private final int WIDTH = 300, HEIGHT = 100;
   private final int DELAY = 20, IMAGE_SIZE = 35;

   private ImageIcon image;
   private Timer timer;
   private int x, y, moveX, moveY;

   //-----------------------------------------------------------------
   //  Sets up the panel, including the timer for the animation.
   //-----------------------------------------------------------------
   public ReboundPanel()
   {
      timer = new Timer(DELAY, new ReboundListener());
      image = new ImageIcon ("happyFace.gif");

      x = 0;
      y = 40;
      moveX = moveY = 3;

      setPreferredSize (new Dimension(WIDTH, HEIGHT));
      setBackground (Color.black);
      timer.start();
   }

   //-----------------------------------------------------------------
   //  Draws the image in the current location.
   //-----------------------------------------------------------------
   public void paintComponent (Graphics page)
   {
      super.paintComponent (page);
      image.paintIcon (this, page, x, y);
   }

   //*****************************************************************
   //  Represents the action listener for the timer.
   //*****************************************************************
   private class ReboundListener implements ActionListener
   {
      //--------------------------------------------------------------
      //  Updates the position of the image and possibly the direction
      //  of movement whenever the timer fires an action event.
      //--------------------------------------------------------------
      public void actionPerformed (ActionEvent event)
      {
         x += moveX;
         y += moveY;

         if (x <= 0 || x >= WIDTH-IMAGE_SIZE)
            moveX = moveX * -1;

         if (y <= 0 || y >= HEIGHT-IMAGE_SIZE)
            moveY = moveY * -1;

         repaint();
      }
   }
}
```

Count Down

Clocks are a standard thing to animate—they change at regular intervals. In this exercise, you will write an applet that displays a simple "clock" that just counts down from 10. The clock will be a DigitalDisplay object. The file *DigitalDisplay.java* contains the code for the class. The file *CountDown.java* contains the program and *CountDownPanel.java* contains a skeleton for the panel for the animation. Copy these files to your directory, compile and run the program. It should just display the clock with the number 10. Now do the following to make the clock count down, stop when it hits 0, and reset to 10 if the user clicks the mouse.

1. Add an inner class named *CountListener* that implements ActionListener. In actionPerformed,
   - Decrement the clock value. To do this use the decrement method in the DigitalDisplay class. (Look at its signature to see how to use it.)
   - If the display value of the clock is negative, stop the timer; otherwise repaint the panel.

2. In the constructor, set up the timer.

3. Compile and run the program to see if it works.

4. Now add code to let a mouse click have some control over the clock. In particular, if the clock is running when the mouse is clicked the clock should be stopped (stop the timer). If the clock is not running, it should be reset to 10 and started. To add the ability of the panel to respond to mouse clicks, do the following:
   - Add an inner class that implements the MouseListener interface. In mouseClicked, if the timer is running, stop it; otherwise reset the clock to 10 (there is a method in the DigitalDisplay class to do this), start the timer, and repaint the applet. The bodies of the other methods in the MouseListener interface will be empty.
   - Add a second parameter of type JApplet to the constructor for CountDownPanel. In your constructor, add the mouse listener to the applet.
   - Modify the program CountDown.java to send the applet (*this*) as the second parameter for the CountDownPanel constructor.

5. Compile and run the program to make sure everything works right.

```
// ***************************************************
//   DigitalDisplay.java
//
//   A simple rectangular display of a single number
// ***************************************************

import java.awt.*;

public class DigitalDisplay
{
    private int displayVal;        // value to be displayed
    private int x, y;              // position
    private int width, height;     // size
    private Font displayFont;      // font for the number

    // -------------------------------------------------------
    // construct a DigitalDisplay object with the given values
    // and New Century Schoolbook font in 40 point bold
    // -------------------------------------------------------
    public DigitalDisplay(int start, int x, int y, int w, int h)
    {
      this.x = x;
      this.y = y;
      width = w;
      height = h;
      displayVal = start;
      displayFont = new Font ("New Century Schoolbook", Font.BOLD, 40);
    }

    // ---------------------------
    // decrement the display value
    // ---------------------------
    public void decrement()
    {
      displayVal--;
    }

    // ---------------------------
    // increment the display value
    // ---------------------------
    public void increment()
    {
      displayVal++;
    }

    // ---------------------------
    // return the display value
    // ---------------------------
    public int getVal()
    {
      return displayVal;
    }

    // ---------------------------------------------------
    // reset the display value to that given in the parameter
    // ---------------------------------------------------
    public void reset (int val)
    {
      displayVal = val;
    }

    // ----------------
    // draw the display
    // ----------------
    public void draw (Graphics page)
    {
      // draw a black border
      page.setColor (Color.black);
      page.fillRect (x, y, width, height);

      // a white inside
      page.setColor (Color.white);
      page.fillRect (x + 5, y + 5, width - 10, height - 10);

      // display the number centered
      page.setColor (Color.black);
      page.setFont (displayFont);
      int fontHeight = page.getFontMetrics().getHeight();
      int strWidth = page.getFontMetrics().stringWidth(""+displayVal);
      page.drawString (""+displayVal, x + width/2 - strWidth/2,
                   y + fontHeight/2 + height/2);
    }
}
```

```
// **********************************************************
//   CountDown.java
//
//   Draws a digital display that counts down from 10.  The
//   display can be stopped or reset with a mouse click.
// **********************************************************


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

public class CountDown extends JApplet
{
    private final int DELAY = 200;
    private Timer timer;


    // ---------------------------------------------------------
    //   Initialize the applet, including the animation.
    // ---------------------------------------------------------
    public void init()
    {
      timer = new Timer (DELAY, null);

      getContentPane().add (new CountDownPanel(timer));
    }


    // ---------------------------------------------------------
    //   Start the animation when the applet is started.
    // ---------------------------------------------------------
    public void start()
    {
      timer.start();
    }

    // ---------------------------------------------------------
    //   Stop the animation when the applet is stopped.
    // ---------------------------------------------------------
    public void stop()
    {
      timer.stop();
    }


}
```

```
// ********************************************************
//   CountDownPanel.java
//
//   Panel for a digital display that counts down from 10.
//   The display can be stopped or reset with a mouse click.
// ********************************************************

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

public class CountDownPanel extends JPanel
{
    private final int WIDTH = 600;
    private final int HEIGHT = 400;
    private final int DISPLAY_WIDTH = 150;
    private final int DISPLAY_HEIGHT = 100;
    private final int DELAY = 200;
    private final int COUNT_START = 10;

    private DigitalDisplay clock;
    private Timer timer;


    // -------------------------------------------------------
    //   Set up the applet.
    // -------------------------------------------------------
    public CountDownPanel (Timer countdown)
    {
      // Set up the timer



      setBackground (Color.blue);
      setPreferredSize (new Dimension (WIDTH, HEIGHT));


      clock = new DigitalDisplay(COUNT_START, WIDTH/2 - DISPLAY_WIDTH,
                          HEIGHT/2 - DISPLAY_HEIGHT,
                          DISPLAY_WIDTH, DISPLAY_HEIGHT);
    }


    // ----------------
    // draw the clock
    // ----------------
    public void paintComponent (Graphics page)
    {
      super.paintComponent (page);
      clock.draw(page);
    }

}
```