Adding Animated Movement to Your Java Application

Learn how to add animated movement into your program where multiple objects chase a lead object as it moves randomly in a given environment.

Published: February 24, 2009
By Richard G. Baldwin

Java Programming Notes # 352

Preface

General

This lesson is the next in a series *(see Resources)* designed to teach you how to write Java programs to do things like:

- Remove *redeye* from a photographic image.
- Distort the human voice.
- Display one image inside another image.
- Do edge detection, blurring, and other filtering operations on images.

- Insert animated cartoon characters into videos of live humans.

If you have ever wondered how to do these things, you've come to the right place.

The World class

If you have studied the earlier lessons in this series *(see Resources)*, you have learned all about the **Turtle** class, its superclass named **SimpleTurtle**, and the classes from which a turtle's contained objects are instantiated *(**Pen** and **PathSegment**)*. You have learned how to instantiate new **Turtle** objects, placing them in either a **World** object or a **Picture** object. You have learned how to manipulate the **Turtle** objects once you place them in their environment. The time has come for you to learn about the environment in which a turtle lives. You will learn about the **World** class in this lesson. You will learn about the **Picture** class in a future lesson.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com.

General background information

A multimedia class library

In this series of lessons, I will present and explain many of the classes in a multimedia class library that was developed and released under a **Creative Commons Attribution 3.0 United States License** *(see Resources)* by Mark Guzdial and Barbara Ericson at Georgia Institute of Technology. In doing this, I will also present some interesting sample programs that use the library.

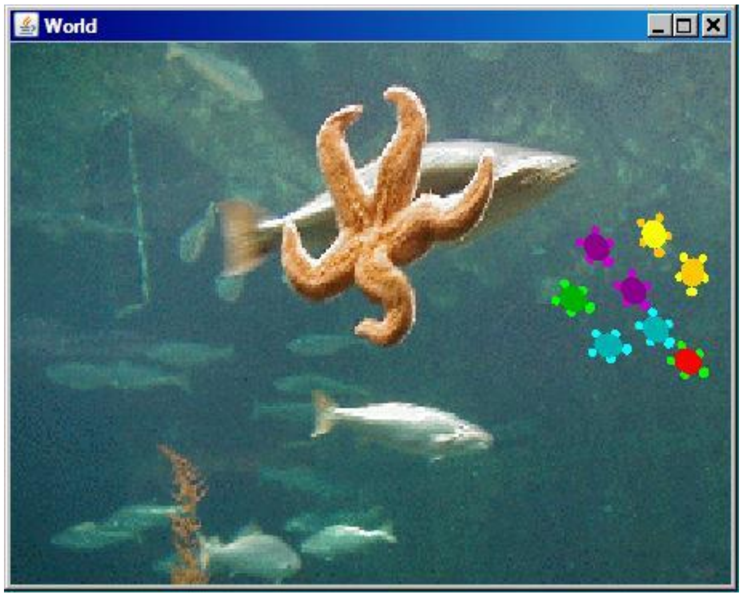
Software installation and testing

I explained how to download, install, and test the multimedia class library in an earlier lesson titled *Multimedia Programming with Java, Getting Started (see Resources)*.

Preview

I will explain Ericson's **World** class in this lesson. I will also present and explain a sample animation program in which seven **Turtle** objects chase another **Turtle** object around in an aquarium. A screen shot from the sample program, showing seven turtles in hot pursuit of a red and green turtle, is shown in Figure 1.

Figure 1. Screen output from program named TurtleWorld01.



Discussion and sample code

The World class

A complete listing of the **World** class is shown in Listing 23 near the end of the lesson. The only changes that were made to this listing were minor formatting changes that were required to make the source code fit into this narrow publication format.

Let's try something new

I'm going to try something in this lesson that I have never tried before. I'm going to explain the **World** class in the context of a sample program named **TurtleWorld01** that animates a group of turtles in a **World** object.

As usual, I will explain the code in fragments. However, to help you know whether a particular code fragment came from the **World** class or came from the sample program, I will use two different background colors for the listings of the code fragments. The two colors are shown in Listing 1 and Listing 2.

Listing 1. Background color for World class code fragments.

```
World class code fragments.
```

Listing 2. Background color for sample program code fragments.

```
Sample program code fragments.
```

Sample program named TurtleWorld01

This is an animated program that is designed to illustrate various features of the **World** class and the **Turtle** class. (*See resources for earlier lessons on the **Turtle** class.)* A complete listing of the program is provided in Listing 24 near the end of the lesson.

The program places eight **Turtle** objects in a **World** object referred to by a variable named **aquarium**. One turtle is designated as the leader and is given a red shell to make it highly visible. *(A screen shot from the running program is shown in Figure 1.)*

An Image from an aquarium containing a starfish and some other fish is used as a background picture for the aquarium. The lead turtle has a red shell and a green body. The body color and the shell color of two of the turtles are set to yellow and orange to make them stand out from the background. The remaining four turtles are presented in their default colors of green, cyan, and purple.

Follow the leader

All eight turtles are initially placed in random locations in the aquarium. The lead turtle swims around randomly. The other seven turtles converge rapidly on the leader and swim in formation following the leader while attempting to avoid collisions with one another.

Much of the time, the formation looks roughly like a hexagon with six turtles forming the perimeter and one turtle in the center. *(See Figure 1.)*

Once started, the program will run until it is manually terminated.

Beginning of the program named TurtleWorld01

The program begins in Listing 3 by instantiating an object of the **Runner** class and calling the **run** method on that object.

Listing 3. Beginning of the program named TurtleWorld01.

```
import java.util.Random;
import java.util.Date;
import java.util.List;
import java.awt.Color;

public class Main{
  public static void main(String[] args){
    new Runner().run();
  }//end main method
}//end class Main
```

Beginning of the class named Runner

The **Runner** class of the **TurtleWorld01** program begins in Listing 4.

Listing 4. Beginning of the class named Runner.

```
class Runner{
  //Instantiate a random number generator.
  Random randGen = new Random(new
Date().getTime());

  //Set the dimensions and instantiate a new
world.
  int aquariumWidth = 450;
  int aquariumHeight = 338;
  World aquarium = new World(

aquariumWidth,aquariumHeight);
```

The **Runner** class begins by instantiating a random-number generator object that will be used later to introduce randomness into the program.

One of the overloaded constructors for the World class

Then the program sets the dimensions to match the background image in Figure 1 and instantiates a new **World** object by calling the overloaded **World** constructor shown in Listing 5.

Listing 5. One of the overloaded constructors for the World class.

```

  public World(int w, int h){
    width = w;
    height = h;

    // set up the world and make it visible
    initWorld(true);
  }//end constructor
```

Listing 5, which is a code fragment from the **World** class, begins by saving the incoming width and height values in a pair of private instance variables. *(You can view all of the World's instance variables in Listing 23.)* Then it calls the **initWorld** method, *(shown in Listing 6)*, passing true as a parameter to complete the construction of the new **World** object.

The initWorld method

The incoming **boolean** parameter to the **initWorld** method is used to determine if the world will be visible when it is instantiated.

Listing 6. The initWorld method of the World class.

```
  private void initWorld(boolean visibleFlag){
    // set the preferred size
    this.setPreferredSize(new
Dimension(width,height));

    // create the background picture
    picture = new Picture(width,height);

    // add this panel to the frame
    frame.getContentPane().add(this);

    // pack the frame
    frame.pack();

    // show this world
    frame.setVisible(visibleFlag);
  }//end initWorld method
```

Listing 6 begins by calling the **setPreferredSize** method inherited from the **JComponent** class to set the preferred size of the **World** object.

The preferred size
I have discussed the *preferred size* in many earlier lessons.

Create an all-white background picture

Then Listing 6 instantiates a new default all-white **Picture** object and assigns the picture's reference to a private instance variable named **picture**.

Add the world to a JFrame object

One of the world's private instance variables *(see Listing 23)* is a reference variable named **frame** containing a reference to an empty **JFrame** object. Listing 6 adds the new **World** object to the **JFrame** object. Therefore, the **JFrame** object becomes the visual container for the world.

Then Listing 6 calls the **pack** method on the **JFrame** object. This causes the size of the **JFrame** object to be set to the preferred size of the world that it contains.

Finally, Listing 6 causes the **JFrame** object to be visible or not visible, depending on the boolean value received as an incoming parameter by the **initWorld** method.

Get reference to the list of turtles

If you examine the instance variables in Listing 23, you will see that the **World** object creates and maintains an **ArrayList** object containing references to all of the turtle objects that are added to the world. Listing 7, which is a fragment from the **TurtleWorld01** program, gets and saves a reference to that list. The reference will be used for manipulating the turtles later. The reference is saved as the interface type **List**.

Listing 7. Get a reference to the list of turtles.

```
  //Get a reference to the list of turtles
maintained by
  // the World object.
  List turtleList = aquarium.getTurtleList();
```

The getTurtleList method of the World class

As you can see in Listing 8, the **getTurtleList** method simply returns the contents of a private instance variable named **turtleList**, which is a reference to the list of turtles.

Listing 8. The getTurtleList method of the World class.

```
  public List getTurtleList(){
    return turtleList;
  }//end getTurtleList methodd
```

Beginning of the run method of the Runner class

The **run** method that was called in Listing 3 begins in Listing 9.

Listing 9. Beginning of the run method of the Runner class.

```
  void run(){
    aquarium.setPicture(new
Picture("aquarium.gif"));
```

Listing 9 calls the **setPicture** method on the **World** object to replace the all-white picture with the background image shown in Figure 1.

The size of the World
Note that the size of the picture of the aquarium was determined in advance and the dimensions of the world were set to the size of the picture Listing
4. Otherwise, either a portion of the world would be white, or some of the picture would not be visible.

The setPicture method of the World class

As you can see in Listing 10, the **setPicture** method simply assigns the incoming **Picture** object's reference to the instance variable named **picture**. *(See Listing 23.)*

Listing 10. The setPicture method of the World class.

```

  public void setPicture(Picture pict){
    picture = pict;
  }//end setPicture method
```

Instantiate eight Turtle objects

Listing 11 instantiates eight new **Turtle** objects and places them in random locations in the **World** object referred to by **aquarium**.

Listing 11. Instantiate eight Turtle objects.

```
    int numberTurtles = 8;

    //Place each turtle in a random location
in the
    // aquarium.
    for(int cnt=0;cnt < numberTurtles;cnt++){
      int xCoor =
              Math.abs(randGen.nextInt() %
aquariumWidth);
      int yCoor =
             Math.abs(randGen.nextInt() %
aquariumHeight);
      new Turtle(xCoor,yCoor,aquarium);
    }//end for loop
```

Turtle constructors

You learned in an earlier lesson *(see Resources)* that whenever you instantiate a new **Turtle** object, you can specify the container in which it will live: **Picture** object or **ModelDisplay** object. You also learned that you can optionally specify the coordinates at which the turtle will be placed in that container.

A ModelDisplay object
The **World** class implements the **ModelDisplay** interface. Therefore, a **World** object is also a **ModelDisplay** object.

When the container is a ModelDisplay object...

You learned that when the specified container is a **ModelDisplay** *(**World**)* object, the constructor for the **Turtle** class:

- Sets the initial position coordinates for the new turtle.
- Sets the initial body color of the new turtle to one of four default colors.
- Sets the initial pen color to match the body color.
- Increments a static turtle counter in the **Turtle** class.
- Calls the **addModel** method on the specified **ModelDisplay** *(**World**)* object.

The addModel method of the World class

The **addModel** method of the **World** class is shown in Listing 12.

Listing 12. The addModel method of the World class.

```
  public void addModel(Object model){
    turtleList.add((Turtle) model);
    if (autoRepaint)
      repaint();
  }//end addModel method
```

The method begins by adding the new turtle's reference to the world's **ArrayList** object that is used to maintain a list of all the turtles that belong to the world.

To paint or not to paint...

Then the **addModel** method checks the value of a **boolean** instance variable named **autoRepaint** *(which is true by default)* to determine whether or not to repaint the world containing the new turtle. The call to the **repaint** method causes the world's **paintComponent** method to be called. I will have more to say about this later.

Perform some housekeeping chores

Listing 13 performs some housekeeping chores in preparation for running the animation loop in the program named **TurtleWorld01**.

Listing 13. Perform some housekeeping chores.

```
    int angle = 0;//leader turning angle
    int leaderMove = 0;//leader move distance

    Turtle turtle = null;
    Turtle testTurtle = null;

    //First turtle in the list is the leader.
Color it red
    // and get its length.
    Turtle leader = (Turtle)turtleList.get(0);
    leader.setShellColor(Color.RED);
    int turtleLength = leader.getHeight();

    //Change the shell and body colors of two
of the other
    // turtles.
    turtle = (Turtle)turtleList.get(3);
    turtle.setBodyColor(Color.YELLOW);
    turtle.setShellColor(Color.ORANGE);
    turtle = (Turtle)turtleList.get(7);
    turtle.setBodyColor(Color.ORANGE);
    turtle.setShellColor(Color.YELLOW);
```

Most of the actions in Listing 13 involve extracting references to **Turtle** objects from the world's list of turtles and calling various methods of the **Turtle** and **SimpleTurtle** classes on those references. You learned about these methods in earlier lessons *(see Resources)*.

Beginning of the animation loop in TurtleWorld01

The animation loop begins in Listing 14.

Listing 14. Beginning of the animation loop in TurtleWorld01.

```
    while(true){//animation loop will run
forever
      //Leader will move a random distance
ranging fromm
      // half its length to 3/4 its length
during each
      // animation cycle.      leaderMove =
(int)(turtleLength/2
                    +
turtleLength*randGen.nextDouble()/4);
      //Leader will turn a random amount
ranging from
      // -22.5 degrees to +22.5 degrees during
each
      // animation cycle.
      angle = (int)(45*(randGen.nextDouble() -
0.5));
```

The animation loop will continue to run until the program is manually terminated.

Once during each animation cycle, the turtle with the red shell *(see Figure 1)* turns by a random amount and moves forward by a random distance. The variables named **leaderMove** and **angle** in Listing 14 are assigned random values that will be used later to control those actions.

Process each turtle during each animation cycle

Each turtle in the list is processed once during each animation cycle. Listing 15 shows the beginning of a **for** loop that iterates on the list of turtles to accomplish that.

Listing 15. Process each turtle during each animation cycle.

```
      for(int cnt = 0;cnt <
turtleList.size();cnt++){
        turtle = (Turtle)turtleList.get(cnt);
        turtle.penUp();//no turtle tracks
allowed
```

Control the distance between the turtles

Left strictly to their own devices, all seven of the turtles that are chasing the leader would attempt to occupy exactly the same space. Listing 16 contains a **for** loop that attempts to force those seven turtles to maintain some distance between them.

Listing 16. Control the distance between the turtles.

```
        for(int cntr = 1;cntr < 
turtleList.size();cntr++){
          testTurtle = 
(Turtle)turtleList.get(cntr);
          //Don't process leader or self.
          if((testTurtle != turtle) && (cnt != 
0)){
            int separation = 
(int)(turtle.getDistance(

testTurtle.getXPos(),testTurtle.getYPos()));
              //Try to keep them separated by 
at least
              // twice the turtleLength center 
to center
              if(separation < 2*turtleLength){
                //Turn and move away from test 
turtle.
                turtle.turnToFace(testTurtle);
                turtle.turn(180);

turtle.forward(turtleLength/3);
              }//end if
          }//end if
        }//end for loop on turtle separation
```

Maintain a decent separation

The code in Listing 16 computes the distance between each of those seven turtles and six other turtles, *(excluding itself and the leader)*.

If the distance between the current turtle and a test turtle is below a specified threshold, the current turtle turns and moves away from the test turtle by a specified amount. While this algorithm is not perfect, it does a pretty good job of keeping the turtles separated as you will see if you run the program.

Call the world's modelChanged method

You learned in an earlier lesson that if the heading, the visibility, or the position of a turtle is changed, the turtle object calls the world's **modelChanged** method, which is shown in Listing 17. The **World** object may, or may not cause itself to be repainted in response to that call.

A model-view-control paradigm
I explained in an earlier lesson *(see Resources)* that the call to the **modelChanged** method constitutes part of the use of a *model-view-control (MVC)* paradigm by the **World** class.

The modelChanged method of the World class

The world's **modelChanged** method checks the value of the **boolean** variable named **autoRepaint**. If the value is true, the **repaint** method is called, which in turn causes the world's **paintComponent** method to be called. *(I'll have more to say about the* ***paintComponent*** *method later.)* Otherwise, it does nothing in response to the call.

Listing 17. The modelChanged method of the World class.

```
  public void modelChanged(){
    if (autoRepaint)
       repaint();
  }//end modelChanged method
```

The leader makes his move

Continuing with the **for** loop that began in Listing 15, if the current turtle being processed is the first turtle in the list, it is the leader. Its behavior is different from the behavior of the other seven turtles.

Listing 18 shows the behavior of the leader during one iteration of the animation loop.

Listing 18. The leader makes his move.

```
        if(cnt == 0){
          //This is the leader

          //Force the leader to bounce off the
walls.
          int xPos = leader.getXPos();
          int yPos = leader.getYPos();

          if(xPos < turtleLength){
            leader.setHeading(90);
          }else if(xPos > aquariumWidth -
turtleLength -2){
            leader.setHeading(-90);
          }//end else

          if(yPos < turtleLength){
            leader.setHeading(180);
          }else if(
                 yPos > aquariumHeight -
turtleLength - 2){
            leader.setHeading(0);
          }//end else

          //Leader turns a random amount and
moves a
          // random distance during each
animation cycle.
          leader.turn(angle);
          leader.forward(leaderMove);
```

Has the leader collided with a wall?

Tests are performed in Listing 18 to determine if the leader has collided with one of the four walls of the world. If so, the leader's heading is set to the direction of the opposite wall.

Turn and move randomly

Regardless of whether or not a collision with a wall has occurred, the leader's heading is modified by a random amount ranging from -22.5 degrees to +22.5 degrees *(see Listing 14)* and the leader moves forward by a random distance ranging from half its length to three-fourths of its length *(see Listing 14)*.

The other seven turtles make their move

If the current turtle is not the leader but instead is one of the seven turtles that follow the leader, the code in Listing 19 is executed.

Listing 19. The other seven turtles make their move.

```
        }else{
          //This is not the leader.  Turn to
face the
          // leader and move toward the
leader.
          turtle.turnToFace(leader);
          int distanceToLeader =
(int)(turtle.getDistance(

leader.getXPos(),leader.getYPos()));
          turtle.forward(distanceToLeader/10);
        }//end else

      }//end for loop processing all turtles
```

Move towards the leader

Each of the seven turtles in the herd turn to face the new position of the leader and move toward the leader by one-tenth the distance to the leader. In theory, the members of the herd would never catch the leader. However, because the leader must turn back toward the herd when it collides with a wall, there are frequent collisions between the leader and the members of the herd. The leader simply blasts through the formation, colliding with other turtles along the way, and goes out the other side of the formation. This causes the other turtles to turn and give chase in the new direction.

Once again, every time any of the turtles moves or changes its heading, the **World** object is given an opportunity to repaint itself.

Listing 19 signals the end of the **for** loop that causes the processing of every turtle during each animation cycle.

Control the animation speed

At the end of each animation cycle, the program goes to sleep for 100 milliseconds to control the overall speed of the animation. This is shown in Listing 20.

Listing 20. Control the animation speed.

```
      //Control the animation speed.
      try{
        Thread.currentThread().sleep(100);
      }catch(InterruptedException ex){
      }//end catch
    }//end while loop

  }//end run
}//end class runner
```

Listing 20 also signals the end of the program named **TurtleWorld01**.

The remainder of the World class

Although that is the end of the program named **TurtleWorld01**, it is not the end of the explanation of the class named **World**. There are several other methods in the **World** class that need to be explained.

Two more overloaded constructors

There are two more overloaded constructors for the **World** class. One receives no arguments and constructs a world that is visible by default and has a default size. The other receives a boolean value and constructs a world that may or may not be visible, depending on the value of the incoming parameter and has a default size.

Both of these constructors are straightforward. You can view the code for these two constructors in Listing 23.

What is a graphics context?

According to *Java Graphics (see Resources)*:

> *"First and foremost, you should know that in Java graphics programming, one of the most important instantiated objects is the graphics context. This*

object is an instance of the java.awt.Graphics class, and it refers to an area of the screen such as an applet. A graphics context provides methods for all of the drawing operations on its area. It also holds "contextual" information about such things as the drawing area's clipping region, painting color, transfer mode, and text font."

Stated differently, if you want to draw on an object using methods of the **Graphics** class or the **Graphics2D** class, you must first get a *graphics context* on the object on which you want to draw.

A rectangular area...

I often tell my students that such an object of the **Graphics** class represents a rectangular area of the screen or an off-screen buffer in memory. Whatever you draw on the **Graphics** object will be drawn on the screen or in the off-screen buffer memory. *(As I recall, it is also possible to get a **Graphics** object that represents a rectangular area on a sheet of paper in a printer, but I haven't had a reason to think about that in a long time.)*

The graphics context for a World object

You can get a graphics context on a **World** object by calling the **getGraphics** method shown in Listing 21.

Listing 21. The getGraphics method of the World class.

```
public Graphics getGraphics(){
  return picture.getGraphics();
}//end getGraphics method
```

Interestingly, when you make that call, what you receive is a reference to the graphics context belonging to the **Picture** object that belongs to the **World** object.

Getting and clearing the world's Picture object

The following methods are available for working with the world's **Picture** object:

- **setPicture(Picture pict)** - replaces the world's default all-white picture with a picture of your choice *(see Listing 10)*.
- **getPicture()** - returns a reference to the world's current **Picture** object.
- **clearBackground()** - replaces the world's current **Picture** object with an all-white picture.

These methods are straightforward. You can view the code in Listing 23.

The paintComponent method of the World class

The paintComponent method of the World class is shown in its entirety in Listing 22.

Listing 22. The paintComponent method of the World class.

```
  public synchronized void
paintComponent(Graphics g){
    Turtle turtle = null;

    // draw the background image
    g.drawImage(picture.getImage(),0,0,null);

    // loop drawing each turtle on the
background image
    Iterator iterator = turtleList.iterator();
    while (iterator.hasNext())
    {
      turtle = (Turtle) iterator.next();
      turtle.paintComponent(g);
    }
  }//end paintComponent method
```

Update the world's visual representation

When an object of the **World** class decides, for whatever reason, that its visual representation on the screen needs to be updated *(see Listing 12 and Listing 17 for example)*, it makes a call to the **repaint** method inherited from the **Component** class. This ultimately results in a call to the world's **paintComponent** method.

Painting in AWT and Swing
For more information on the paint mechanisms utilized by AWT and Swing, including information on how to write the most efficient painting code, see Painting in AWT and Swing.

Can be triggered by the operating system

Similarly, when the operating system decides for whatever reason that the world's screen representation needs to be updated, *(such as when it is minimized and then restored)*, the operating system causes the world's **paintComponent** method to be called.

A reference to a graphics context

When the **paintComponent** method is called, it receives a reference to a graphics context that represents an area of the screen that is to be repainted.

Casting the graphics context
Casting the graphics context reference to type

Actually an object of the Graphics2D class

Graphics2D makes it possible to use the methods of the **Graphics2D** class to draw on the graphics context.

The graphics context is received as type **Graphics**, but it is actually a reference to an object of the **Graphics2D** subclass of **Graphics**.

Draw the current picture on the graphics context

Listing 22 begins by calling the **drawImage** method to draw the image contained in the world's current **Picture** object on the graphics context. This produced the aquarium background shown in Figure 1.

Call paintComponent on each of the turtles

Then Listing 22 uses an iterator to loop and call the **paintComponent** method on each **Turtle** object whose reference is stored in the list of turtles, passing the same graphics context received by the world's **paintComponent** method.

Draw each turtle and its track

We learned in an earlier lesson *(see Resources)* that a turtle's **paintComponent** method begins by casting the incoming reference of type **Graphics** to type **Graphics2D**. Then it draws the body parts of the turtle at the correct location, in the correct color with the correct heading. Then it calls the **paintComponent** method on the **Pen** object that belongs to the turtle, passing the same graphics context that it received.

Drawing the historical turtle track

As a turtle moves with the pen down, the turtle's **Pen** object maintains a list of historical turtle movements as type **PathSegment**. When the **paintComponent** method is called on the **Pen** object, it uses the contents of that list to reconstruct and draw line segments with the correct width and the correct color representing the turtle's historical path. Note, however, that the pen was never down in the sample program named TurtleWorld01 so no turtle tracks are produced. *(For an interesting effect, disable the call to the **penUp** method in Listing 15.)*

The remaining methods of the World class

The world class defines several additional methods that I haven't discussed in this lesson:

- **getLastTurtle** - a method to get a reference to the last turtle in the list of turtles.
- **containsTurtle** - a method to determine if the list contains a reference to a specific turtle object.
- **remove** - a method to remove a specific turtle from the list.

- **getWidth** - a method to get the width of the **World** object.
- **getHeight** - a method to get the height of the **World** object.
- **setAutoPaint** - a method to set the value of the **autoPaint** variable to true or false.
- **setVisible** - a method to make the **World** object visible or invisible.
- **getTurtleIterator** - a method to get an **Iterator** object on the list of turtles.
- **toString** - a method to return a string that describes a **World** object.

The code for all of the methods in the above list is straightforward. You can view that code in Listing 23.

That wraps it up

That brings us to the end of the explanation of the class named **World** and the end of the explanation of the program named **TurtleWorld01**.

Run the programs

I encourage you to copy the code from Listing 24, compile the code, and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this lesson, I have explained an animation program named **TurtleWorld01** along with an explanation of the class named **World**. The program named **TurtleWorld01** illustrates a form of *flocking behavior (see Wikipedia: Flocking behavior in Resources)* where a herd of turtles follow a lead turtle as it moves randomly in a **World** object while attempting to avoid collisions with one another.

What's next?

In the next lesson, you learn how to write programs using the methods defined in the Picture and SimplePicture classes that are declared in the DigitalPicture interface.

Resources

- Creative Commons Attribution 3.0 United States License
- Media Computation book in Java - numerous downloads available
- Introduction to Computing and Programming with Java: A Multimedia Approach
- DrJava download site
- DrJava, the JavaPLT group at Rice University
- DrJava Open Source License
- The Essence of OOP using Java, The this and super Keywords

- Threads of Control
- Painting in AWT and Swing
- Wikipedia Turtle Graphics
- IsA or HasA
- Vector Cad-Cam XI Lathe Tutorial
- Classification of 3D to 2D projections
- 200 Implementing the Model-View-Controller Paradigm using Observer and Observable
- 300 Java 2D Graphics, Nested Top-Level Classes and Interfaces
- 302 Java 2D Graphics, The Point2D Class
- 304 Java 2D Graphics, The Graphics2D Class
- 306 Java 2D Graphics, Simple Affine Transforms
- 308 Java 2D Graphics, The Shape Interface, Part 1
- 310 Java 2D Graphics, The Shape Interface, Part 2
- 312 Java 2D Graphics, Solid Color Fill
- 314 Java 2D Graphics, Gradient Color Fill
- 316 Java 2D Graphics, Texture Fill
- 318 Java 2D Graphics, The Stroke Interface
- 320 Java 2D Graphics, The Composite Interface and Transparency
- 322 Java 2D Graphics, The Composite Interface, GradientPaint, and Transparency
- 324 Java 2D Graphics, The Color Constructors and Transparency
- 400 Processing Image Pixels using Java, Getting Started
  402 Processing Image Pixels using Java, Creating a Spotlight
  404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
  406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
  408 Processing Image Pixels, Performing Convolution on Images
  410 Processing Image Pixels, Understanding Image Convolution in Java
  412 Processing Image Pixels, Applying Image Convolution in Java, Part 1
  414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
  416 Processing Image Pixels, An Improved Image-Processing Framework in Java
  418 Processing Image Pixels, Creating Visible Watermarks in Java
  450 A Framework for Experimenting with Java 2D Image-Processing Filters
  452 Using the Java 2D LookupOp Filter Class to Process Images
  454 Using the Java 2D AffineTransformOp Filter Class to Process Images
  456 Using the Java 2D LookupOp Filter Class to Scramble and Unscramble Images
  458 Using the Java 2D BandCombineOp Filter Class to Process Images
  460 Using the Java 2D ConvolveOp Filter Class to Process Images
  462 Using the Java 2D ColorConvertOp and RescaleOp Filter Classes to Process Images
- 506 JavaBeans, Introspection
- 2100 Understanding Properties in Java and C#
- 2300 Generics in J2SE, Getting Started
- 340 Multimedia Programming with Java, Getting Started

- 342 Getting Started with the Turtle Class: Multimedia Programming with Java
- 344 Continuing with the SimpleTurtle Class: Multimedia Programming with Java
- 346 Wrapping Up the SimpleTurtle Class: Multimedia Programming with Java
- 348 The Pen and PathSegment Classes: Multimedia Programming with Java
- 349 A Pixel Editor Program in Java: Multimedia Programming with Java
- 350 3D Displays, Color Distance, and Edge Detection
- 351 A Slider-Controlled Softening Program for Digital Photos

Complete program listings

Complete listings of the programs discussed in this lesson are shown in Listing 23 and Listing 24 below.

Listing 23. Source code for Ericson's class named World.

```
import javax.swing.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observer;
import java.awt.*;

/**
 * Class to represent a 2d world that can hold
turtles and
 * display them
 *
 * Copyright Georgia Institute of Technology
2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class World
               extends JComponent implements
ModelDisplay{
  ////////////////// fields
///////////////////////

  /** should automatically repaint when model
changed */
  private boolean autoRepaint = true;

  /** the background color for the world */
  private Color background = Color.white;

  /** the width of the world */
  private int width = 640;

  /** the height of the world */
  private int height = 480;

  /** the list of turtles in the world */
  private List<Turtle> turtleList =
                                  new
ArrayList<Turtle>();

  /** the JFrame to show this world in */
  private JFrame frame = new JFrame("World");

  /** background picture */
  private Picture picture = null;

  ////////////////// the constructors
///////////////

  /**
   * Constructor that takes no arguments
   */
  public World()
  {
    // set up the world and make it visible
    initWorld(true);
  }

  /**
   * Constructor that takes a boolean to
   * say if this world should be visible
   * or not
   * @param visibleFlag if true will be
visible
   * else if false will not be visible
   */
  public World(boolean visibleFlag)
  {
    initWorld(visibleFlag);
  }

  /**
   * Constructor that takes a width and height
for this
   * world
   * @param w the width for the world
   * @param h the height for the world
   */
  public World(int w, int h)
  {
    width = w;
    height = h;

    // set up the world and make it visible
    initWorld(true);
  }

  ///////////////// methods
///////////////////////////

  /**
   * Method to initialize the world
   * @param visibleFlag the flag to make the
```

```
world
   * visible or not
   */
  private void initWorld(boolean visibleFlag)
  {
    // set the preferred size
    this.setPreferredSize(new
Dimension(width,height));

    // create the background picture
    picture = new Picture(width,height);

    // add this panel to the frame
    frame.getContentPane().add(this);

    // pack the frame
    frame.pack();

    // show this world
    frame.setVisible(visibleFlag);
  }

  /**
   * Method to get the graphics context for
drawing on
   * @return the graphics context of the
background
   * picture
   */
  public Graphics getGraphics(){
    return picture.getGraphics();
  }

  /**
   * Method to clear the background picture
   */
  public void clearBackground(){
    picture = new Picture(width,height);
  }

  /**
   * Method to get the background picture
   * @return the background picture
   */
  public Picture getPicture() { return
picture; }

  /**
   * Method to set the background picture
   * @param pict the background picture to use
   */
  public void setPicture(Picture pict) {
picture = pict; }

  /**
   * Method to paint this component
   * @param g the graphics context
   */
  public synchronized void
paintComponent(Graphics g)
  {
    Turtle turtle = null;

    // draw the background image
    g.drawImage(picture.getImage(),0,0,null);

    // loop drawing each turtle on the
background image
    Iterator iterator = turtleList.iterator();
    while (iterator.hasNext())
    {
      turtle = (Turtle) iterator.next();
      turtle.paintComponent(g);
    }
  }

  /**
   * Method to get the last turtle in this
world
   * @return the last turtle added to this
world
   */
  public Turtle getLastTurtle()
  {
    return (Turtle)
turtleList.get(turtleList.size() - 1);
  }


  /**
   * Method to add a model to this model
displayer
   * @param model the model object to add
   */
  public void addModel(Object model)
  {
    turtleList.add((Turtle) model);
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to check if this world contains
the passed
   * turtle
   * @return true if there else false
   */
  public boolean containsTurtle(Turtle turtle)
  {
    return (turtleList.contains(turtle));
  }
  /**
   * Method to remove the passed object from
the world
   * @param model the model object to remove
   */
  public void remove(Object model)
  {
    turtleList.remove(model);
  }

  /**
   * Method to get the width in pixels
   * @return the width in pixels
   */
  public int getWidth() { return width; }

  /**
   * Method to get the height in pixels
   * @return the height in pixels
   */
  public int getHeight() { return height; }

  /**
   * Method that allows the model to notify
the display
   */
  public void modelChanged()
  {
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to set the automatically repaint
flag
   * @param value if true will auto repaint
   */
  public void setAutoRepaint(boolean value){
    autoRepaint = value;
  }

  /**
   * Method to hide the frame
   */
//  public void hide()
//  {
//    frame.setVisible(false);
//  }

  /**
   * Method to show the frame
   */
//  public void show()
//  {
//    frame.setVisible(true);
//  }
```

```
  /**
   * Method to set the visibility of the world
   * @param value a boolean value to say if
should show
   * or hide
   */
  public void setVisible(boolean value)
  {
    frame.setVisible(value);
  }

  /**
   * Method to get the list of turtles in the
world
   * @return a list of turtles in the world
   */
  public List getTurtleList()
  { return turtleList;}

  /**
   * Method to get an iterator on the list of
turtles
   * @return an iterator for the list of
turtles
   */
  public Iterator getTurtleIterator()
  { return turtleList.iterator();}

  /**
   * Method that returns information about
this world
   * in the form of a string
   * @return a string of information about
this world
   */
  public String toString()
  {
    return "A " + getWidth() + " by " +
getHeight() +
      " world with " + turtleList.size()
                                      + "
turtles in it.";
  }

} // end of World class
```

Listing 24. Source code for the program named TurtleWorld01.

```
/*Program TurtleWorld01
Copyright R.G.Baldwin 2009
```

```
This is an animated program that is designed to illustrate
various features of the Turtle and World classes.

The program places eight Turtle objects in a World object
known as the aquarium. One turtle is designated as the
leader and is given a red shell to make it highly visible.

An Image from an aquarium containing a starfish and some
other fish is used as a background picture for the
aquarium.

The body color and shell color of two other turtles are
set to yellow and orange to make them stand out from the
background.

The leader swims around randomly in the aquarium.

All eight turtles are initially placed in random locations
in the aquarium. However, the other seven turtles rapidly
converge on the leader and swim in formation following the
leader while attempting to avoid collisions with one
another.

Much of the time, the formation looks roughly like a
hexagon with six turtles forming the perimeter and one
turtle in the center.

Once started, the program will run until it is manually
terminated.

Tested using Windows Vista Premium Home edition and
Ericson's multimedia library.
*********************************************************/
import java.util.Random;
import java.util.Date;
import java.util.List;
import java.awt.Color;

public class Main{
  public static void main(String[] args){
    new Runner().run();
  }//end main method
}//end class Main
//-------------------------------------------------------//

class Runner{
  //Instantiate a random number generator.
  Random randGen = new Random(new Date().getTime());

  //Set the dimensions and instantiate a new world.
  int aquariumWidth = 450;
  int aquariumHeight = 338;
  World aquarium = new World(
                            aquariumWidth,aquariumHeight);

  //Get a reference to the list of turtles maintained by
```

```
// the World object.
List turtleList = aquarium.getTurtleList();
//-------------------------------------------------------//

void run(){
  aquarium.setPicture(new Picture("aquarium.gif"));

  int numberTurtles = 8;

  //Place each turtle in a random location in the
  // aquarium.
  for(int cnt=0;cnt < numberTurtles;cnt++){
    int xCoor =
            Math.abs(randGen.nextInt() % aquariumWidth);
    int yCoor =
           Math.abs(randGen.nextInt() % aquariumHeight);
    new Turtle(xCoor,yCoor,aquarium);
  }//end for loop

  int angle = 0;//leader turning angle
  int leaderMove = 0;//leader move distance

  Turtle turtle = null;
  Turtle testTurtle = null;

  //First turtle in the list is the leader. Color it red
  // and get its length.
  Turtle leader = (Turtle)turtleList.get(0);
  leader.setShellColor(Color.RED);
  int turtleLength = leader.getHeight();

  //Change the shell and body colors of two of the other
  // turtles.
  turtle = (Turtle)turtleList.get(3);
  turtle.setBodyColor(Color.YELLOW);
  turtle.setShellColor(Color.ORANGE);
  turtle = (Turtle)turtleList.get(7);
  turtle.setBodyColor(Color.ORANGE);
  turtle.setShellColor(Color.YELLOW);

  while(true){//animation loop will run forever
    //Leader will move a random distance ranging from
    // half its length to 3/4 its length during each
    // animation cycle.
    leaderMove = (int)(turtleLength/2
             + turtleLength*randGen.nextDouble()/4);
    //Leader will turn a random amount ranging from
    // -22.5 degrees to +22.5 degrees during each
    // animation cycle.
    angle = (int)(45*(randGen.nextDouble() - 0.5));

    //Process each turtle in the list during each
    // animation cycle.
    for(int cnt = 0;cnt < turtleList.size();cnt++){
      turtle = (Turtle)turtleList.get(cnt);
      turtle.penUp();//no turtle tracks allowed
```

```
//Force the turtles to maintain some distanced
// between them by comparing the distance from the
// current turtle to every other turtle (other
// than the leader) and making a correction when
// too close.
for(int cntr = 1;cntr < turtleList.size();cntr++){
  testTurtle = (Turtle)turtleList.get(cntr);
  //Don't process leader or self.
  if((testTurtle != turtle) && (cnt != 0)){
    int separation = (int)(turtle.getDistance(
      testTurtle.getXPos(),testTurtle.getYPos()));
      //Try to keep them separated by at least
      // twice the turtleLength center to center
      if(separation < 2*turtleLength){
        //Turn and move away from test turtle.
        turtle.turnToFace(testTurtle);
        turtle.turn(180);
        turtle.forward(turtleLength/3);
      }//end if
  }//end if
}//end for loop on turtle separation

if(cnt == 0){
  //This is the leader

  //Force the leader to bounce off the walls.
  int xPos = leader.getXPos();
  int yPos = leader.getYPos();

  if(xPos < turtleLength){
    leader.setHeading(90);
  }else if(xPos > aquariumWidth -turtleLength -2){
    leader.setHeading(-90);
  }//end else

  if(yPos < turtleLength){
    leader.setHeading(180);
  }else if(
         yPos > aquariumHeight -turtleLength - 2){
    leader.setHeading(0);
  }//end else

  //Leader turns a random amount and moves a
  // random distance during each animation cycle.
  leader.turn(angle);
  leader.forward(leaderMove);
}else{
  //This is not the leader.  Turn to face the
  // leader and move toward the leader.
  turtle.turnToFace(leader);
  int distanceToLeader = (int)(turtle.getDistance(
            leader.getXPos(),leader.getYPos()));
  turtle.forward(distanceToLeader/10);
}//end else
```

```
      }//end for loop processing all turtles

      //Control the animation speed.
      try{
        Thread.currentThread().sleep(100);
      }catch(InterruptedException ex){
      }//end catch
    }//end while loop

  }//end run
}//end class runner
```

Copyright

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is object-oriented programming using Java and other OOP languages.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-