Continuing with the SimpleTurtle Class: Multimedia Programming with Java

Learn how the World class and the Turtle class of the multimedia library implement a practical version of the Model-View-Control programming paradigm. Investigate the differences between placing a turtle in a world and placing a turtle in a picture.

Published: December 16, 2008
By Richard G. Baldwin

Java Programming Notes # 344

- Preface
  - General
  - Viewing tip
    - Figures
    - Listings
  - Supplementary material
- General background information
  - A multimedia class library
  - The DrJava IDE
  - Software installation and testing
- Preview
- Discussion and sample code
  - First sample program - Java344a
  - Second sample program - Java344b
  - The reason for the dead turtles
  - The drop method
  - Methods that cause the turtle to turn
    - The turn(int degrees) method
    - A model-view-control (MVC) programming paradigm
    - The turnLeft() and turnRight() methods
    - The turnToFace(int x,int y) method
    - The turnToFace(SimpleTurtle turtle) method
- Run the programs
- Summary
- What's next?
- Resources
- Complete program listings
- Copyright
- About the author

Preface

General

This is the third lesson in a series 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 in videos of live humans.

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

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

- Figure 1. Screen output from program named Java344a.
- Figure 2. Screen output from program named Java344b.

Listings

- Listing 1. Beginning of source code for program Java344a.
- Listing 2. Load the turtle image into a Picture object.
- Listing 3. Instantiate and manipulate a Turtle object in a World.
- Listing 4. Drop more pictures while turning and moving the turtle.
- Listing 5. Abbreviated listing of the program named Java344b.
- Listing 6. Abbreviated listing of the updateDisplay method.
- Listing 7. Beginning of the drop method.
- Listing 8. Construct a rotation and translation transform.
- Listing 9. Draw dropPicture on the graphics context.
- Listing 10. Remainder of the drop method.
- Listing 11. The turn(int degrees) method.
- Listing 12. The turnLeft and turnRight methods
- Listing 13. Beginning of the turnToFace(int x,int y) method.
- Listing 14. The turnToFace(SimpleTurtle turtle) method.
- Listing 15. Source code for the SimpleTurtle class.
- Listing 16. Source code for the program named Java344a.
- Listing 17. Source code for the program named Java344b.

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.

The DrJava IDE

In some cases, I will use a free lightweight Java IDE named **DrJava** *(see Resources)*. This IDE is useful because it provides an interactive Java programming mode. The interactive mode makes it easy to *"try things out"* without the requirement to write and compile a complete Java application. *(The IDE also provides a typical Java text editor, access to the Java compiler and runtime engine, a debugger, etc.)*

Even though I will sometimes use DrJava, you should be able to use any Java IDE *(for the non-interactive material)* to compile and execute my sample programs so long as you set the *classpath* to include the multimedia class library. You should also be able to avoid the use of a Java IDE altogether if you choose to do so. You can create the source code files using a simple text editor, and then compile and execute the sample programs from the command line using a batch file.

Software installation and testing

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

I also explained how to create a Windows batch file that you can use to set the classpath and run programs that use the multimedia library in the total absence of an IDE.

Preview

As I promised in the previous lesson *(see Resources)*, I will begin this lesson with an explanation of the **drop** method of the **SimpleTurtle** class. A complete listing of the **SimpleTurtle** class is provided in Listing 15 near the end of the lesson.

Then I will explain the following methods from the **SimpleTurtle** class. The methods in this group can be called to cause a turtle to face in a particular direction.

- turn(int degrees)
- turnLeft()
- turnRight()
- turnToFace(int x,int y)
- turnToFace(SimpleTurtle turtle)

Along the way, I will explain how the **World** class and the **Turtle** class implement a practical version of the *Model-View-Control* programming paradigm.

Discussion and sample code

I will begin by explaining two sample programs that illustrate the use of the **drop** method as well as some of the other methods that I will explain in this lesson.

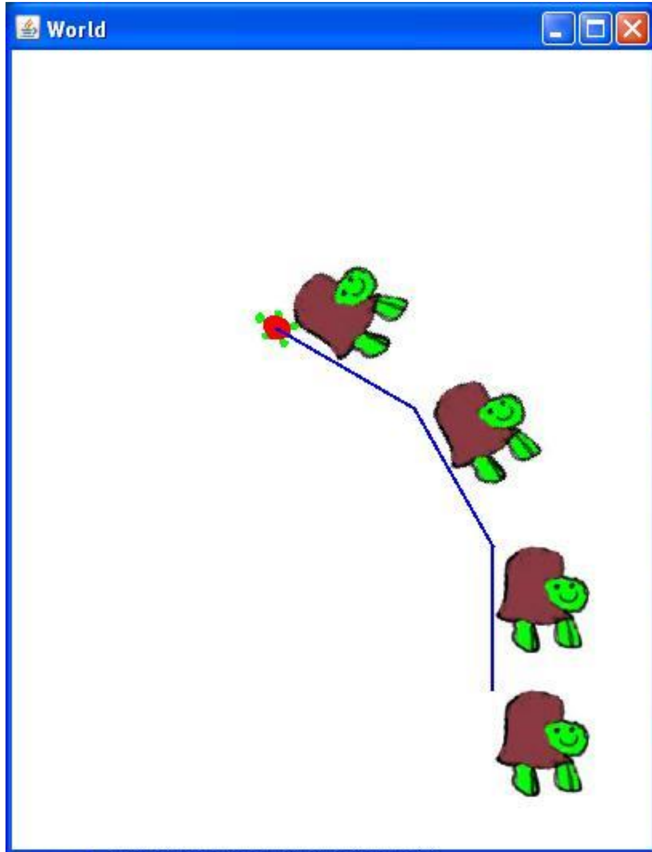
First sample program - Java344a

The purpose of this program is to illustrate several different methods of the **Turtle** class including the **drop** method. The program moves a **Turtle** object around on a **World** and drops several copies of a small **Picture** object along the way.

The screen output

The screen output produced by this program is shown in Figure 1. I will refer back to Figure 1 while discussing the program code.

Figure 1. Screen output from program named Java344a.



Set the mediapath property

As is my custom, I will explain this program in fragments. A complete listing of the program is provided in Listing 16 near the end of the lesson. The source code begins with the fragment shown in Listing 1.

Listing 1. Beginning of source code for program Java344a.

```
import java.awt.Color;
public class Main{
  public static void main(String[] args){

FileChooser.setMediaPath("M:/Ericson/mediasources/");
```

Set the mediapath

There are several ways to establish the location *(on the disk)* of an image file that will be loaded into a **Picture** object. One of those ways is shown in Listing 1. The code in Listing 1 calls the static **setMediaPath** method of the **FileChooser** class to set a property value named **mediapath**. The value is set to point to a folder containing an image file that will be loaded in Listing 2. *(Note that all the slash characters in the*

***mediapath** string are forward-leaning slash characters and the **mediapath** string is terminated with a forward slash.)*

Load the turtle image

Listing 2 instantiates a new object of the class **Picture** initializing its contents with the image of the smiling turtle shown in Figure 1. Note that after having set the **mediapath** property, only the name of the image file is required as type **String**.

Listing 2. Load the turtle image into a Picture object.

```
Picture p2 = new Picture("turtle.jpg");
```

Eliminating the need to set the mediapath

You can also access an image file by omitting the **mediapath** entirely and providing a full path to the image file in Listing 2. Just be sure to use forward-leaning slashes and not backward-leaning slashes in the string. For the special case where the image file is in the current directory, you can also eliminate the path and simply specify the name of the image file.

The Picture class

I will be explaining the **Picture** class in some detail in a future lesson so I won't go into much detail regarding that class in this lesson. Suffice it at this point to say that one of the overloaded constructors for the **Picture** class accepts an image file name as a **String** and uses that information to encapsulate the image in the new object.

Instantiate and manipulate a Turtle object in a World

Listing 3 begins by instantiating a new **World** object with dimensions of 400 by 500 pixels as shown in Figure 1.

Listing 3. Instantiate and manipulate a Turtle object in a World.

```
World mars = new World(400,500);
Turtle joe = new Turtle(300,400,mars);

joe.setShellColor(Color.RED);
joe.setPenColor(Color.BLUE);
joe.setPenWidth(2);

joe.drop(p2);//Draw a small picture
```

Then Listing 3 instantiates a new **Turtle** object and places it in the world at coordinates 300,400. This places the turtle at the bottom of the vertical blue line on the bottom right side of Figure 1. *(Recall that the origin is the upper left corner. Positive x is to the right and positive y is down.)*

Not related to the turtle image

It is also important to note that this **Turtle** object is completely independent of the turtle image that was loaded in Listing 2. Listing 2 places an image of a turtle in an object of the **Picture** class. This turtle is an object of the **Turtle** class.

Set Turtle object properties

Then Listing 3 calls three property setter methods to:

- Change the color of the turtle's shell from default green to red.
- Change the color of the pen from default green to blue.
- Change the width of the pen from the default of one pixel to two pixels.

Drop a picture

Finally, Listing 3 calls the **drop** method on the **Turtle** object to draw the image of the smiling turtle at the current location and orientation of the **Turtle** object. Note that at this point the turtle is facing north and is centered at the bottom of the blue line segment. Also note that when you drop a picture, the upper-left corner of the picture coincides with the current location of the turtle. Thus, the bulk of the turtle image is to the right of and below the end of the blue line in Figure 1.

Drop more pictures while turning and moving the turtle

Listing 4 begins by moving the turtle forward *(north)* by 90 pixels and then dropping another picture of the turtle image. This is the second turtle image from the bottom in Figure 1. Because the **Turtle** object is still facing north at this point, this turtle image is oriented the same as the turtle image at the bottom.

Listing 4. Drop more pictures while turning and moving the turtle.

```
    joe.forward(90);
    joe.drop(p2);

    joe.turn(-30);
    joe.forward();
    joe.drop(p2);

    joe.turn(-30);
    joe.forward();
    joe.drop(p2);
  }//end main
```

```
}//end class
```

Turn, move, and drop

Then Listing 4 turns the turtle by -30 degrees, moves it forward by the default 100 pixels, and drops another picture. This is the third turtle image from the bottom, and it has been rotated by 30 degrees counter-clockwise. *(Positive angles represent clockwise rotation.)*

That process is repeated one more time producing the turtle image at the top along with the visual manifestation of the **Turtle** object with the green body and the red shell.

Only one Turtle object is visible

Note that unlike the next sample program, the **Turtle** object in this program is visible only at the final location. However, if we were to slow the execution down significantly, we would see the **Turtle** object jumping from the end of one line segment to the end of the next line segment and dropping a picture each time it stops. We could do that by calling the **sleep** method of the **Thread** class to insert a time delay immediately following each call to the **drop** method.

Second sample program - Java344b

A complete listing of this program is shown in Listing 17 near the end of the lesson.

The purpose of this program is to illustrate the use of several different methods of the **Turtle** class, as well as to illustrate the placement of a **Turtle** object on a **Picture** object *(as opposed to a **World** object)*.

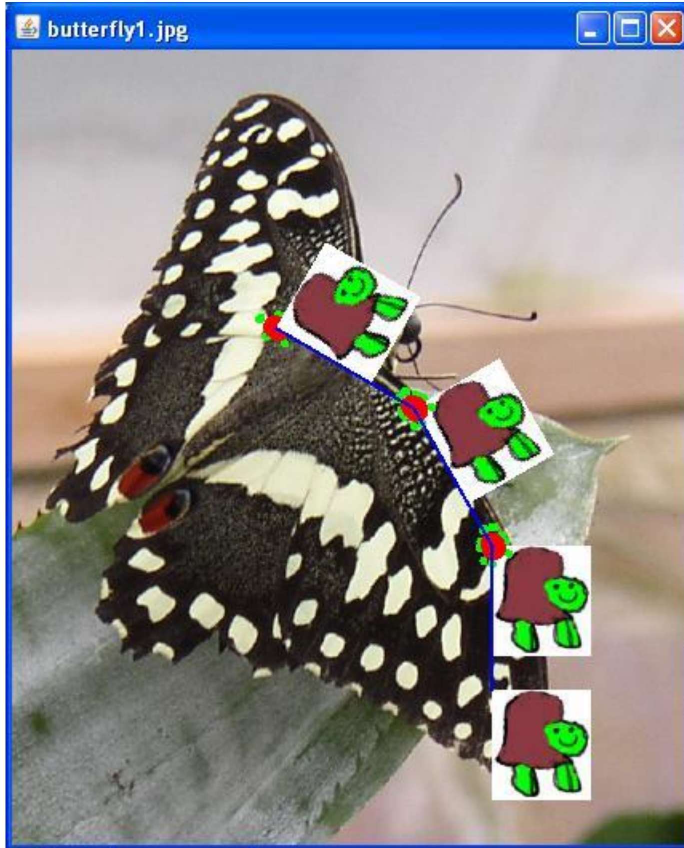
The program also illustrates dropping pictures at the current position and orientation of the **Turtle** object.

The program moves a **Turtle** object around on a Picture and drops several copies of a smaller **Picture** object along the way.

The screen output

The screen output produced by this program is shown in Figure 2. Except for the fact that this program places the Turtle object on a **Picture** object instead of a **World** object, the program is very similar to the previous program. There are, however, some significant differences in the behavior of the **Turtle** object in these two cases.

Figure 2. Screen output from program named Java344b.



Lots of dead turtles lying around

We learned in the previous lesson that when a **Turtle** object is placed on a **Picture** object, it is made invisible by default. Therefore, in order to make the turtle visible, it was necessary for me to set the turtle's *visible* property to true.

I was initially surprised to learn that unlike the case of the **World** in Figure 1, an old image of the **Turtle** object remains on the screen when it moves from one location to the next. Further complicating matters, when a turtle is simply rotated, two images of the turtle appear, one on top of the other. As a result, two of the images of the **Turtle** object in Figure 2 appear to have two heads and eight legs. *(In each these two cases, there are really two images of the turtle, one on top of the other.)*

I suppose that serves to illustrate the reason why the turtle is invisible by default. When you place a visible turtle on a picture and move it around, it leaves a trail of dead turtles along the way.

An abbreviated listing of program named Java344b

An abbreviated listing of this program is shown in Listing 5.

Listing 5. Abbreviated listing of the program named Java344b.

```
import java.awt.Color;
public class Main{
  public static void main(String[] args){

    //...code deleted
    //Instantiate two objects of Picture
class, each of
    // which encapsulates an image.
    Picture p1 = new
Picture("butterfly1.jpg");//large
    Picture p2 = new
Picture("turtle.jpg");//small

    //Note that the Turtle object is different
from the
    // image of the turtle in the Picture
object.
    Turtle joe = new Turtle(300,400,p1);

    //..code deleted
    joe.setVisible(true);

    //..code deleted
    p1.show();
  }//end main

}//end class
```

Listing 5 contains only the code that is different from the code that I explained earlier in Listing 1 through Listing 4. Hopefully the embedded comments in Listing 5 are sufficient to help you understand the difference between these two programs.

The reason for the dead turtles

Before getting into an explanation of the **drop** method, let's see if we can understand why Figure 1 shows only one image of the turtle with the red shell and Figure 2 shows five images of the turtle with the red shell. *(Some of the turtle images in Figure 2 are drawn on top of other turtle images.)* To understand this, we need to go back and take another look at the **updateDisplay** method that I explained in the previous lesson *(see Resources)*.

The updateDisplay method

An abbreviated listing of the **updateDisplay** method is shown in Listing 6.

Listing 6. Abbreviated listing of the updateDisplay method.

```
  public synchronized void updateDisplay(){
    //...code deleted
    if (picture != null){
```

```
      //...code deleted
      //Turtle was placed in a Picture object
      Graphics g = picture.getGraphics();
      paintComponent(g);
    }//end if
    else if (modelDisplay != null){
      //...code deleted
      //Turtle was placed in a World object
      modelDisplay.modelChanged();
    }//end else if
  }//end updateDisplay
```

I refer to this as an abbreviated listing because I deleted all of the code that is not germane to the question regarding dead turtles.

Called by many other methods

We learned in the previous lesson that the **updateDisplay** method is called by many other methods that make changes to the position or orientation of a turtle. In fact, it is called by all of the following methods:

- turnToFace
- setVisible
- forward
- moveTo
- turn

In addition, the methods in the above list are called by other methods resulting ultimately in a call to the **updateDisplay** method by many different methods. For example, the **forward** method is called by the **backward** method with a negative parameter value.

Behavior of the updateDisplay method

Now consider the behavior of the **updateDisplay** method shown in Listing 6. As you can see, the behavior is different depending on whether the turtle was placed in a **Picture** object or in a **World** object.

A turtle in a Picture object

For the case where the turtle was placed in a **Picture** object, the **paintComponent** method is called every time the **updateDisplay** method is called. Nothing is done to erase the old image of the turtle before drawing a new one at the same or at a different location. This results in the multiple images of the turtle, some on top of others, shown in Figure 2.

A turtle in a World object

For the case where the turtle was placed in a **World** object, the **updateDisplay** method does not call the **paintComponent** method directly. Instead it calls the **modelChanged** method on the **World** object. This sends a message to the **World** object indicating that the turtle has changed. The decision as to whether or not to repaint the turtle is made by the world.

As we will see when we study the **World** class later, if the **World** object decides to repaint the turtle as a result of that notification, it first draws a background image on the entire world, erasing everything that was previously displayed. Then it draws the turtle in its new location and/or orientation on the world's background image.

An important question

This raises the question as to why this process erases old images of the turtle with the red shell but doesn't erase the pictures that were previously dropped. The answer is rather complicated, but I will attempt to answer it in conjunction with my explanation of the **drop** method later.

What about the history of pen movements?

This display approach also has very significant ramifications for the display of the turtle's track created by the pen, but that is another topic for another lesson. In effect, this means that the pen must maintain a history of its movements so that every time the world is repainted, the entire history of pen movements can be redrawn. I will explain the **Pen** class in a future lesson.

The drop method

The purpose of this method is to draw an incoming **Picture** object at the current location of the turtle with the same orientation as the turtle in either a **ModelDisplay** object *(such as a **World** object)* or a **Picture** object.

Beginning of the drop method

The beginning of the **drop** method is shown in Listing 7.

Listing 7. Beginning of the drop method.

```
  public synchronized void drop(Picture 
dropPicture){
    Graphics2D g2 = null;

    if (picture != null)
      //Draw dropPicture on a Picture object
      g2 = (Graphics2D) picture.getGraphics();
    else if (modelDisplay != null)
```

```
      //Draw dropPicture on a ModelDisplay
(World) object
      g2 = (Graphics2D)
modelDisplay.getGraphics();
```

Figure 1 shows the result of dropping a picture of a smiling turtle four times in a **World** object. Figure 2 shows the result of dropping the same picture four times in a **Picture** object. As I explained earlier, the upper-left corner of the dropped picture is aligned with the location of the turtle and the orientation of the picture matches the orientation of the turtle. This is most obvious in Figure 2 where images of the turtle at the time of the drop are still showing.

Decision between World and Picture object

The code in Listing 7 determines whether **dropPicture** is to be drawn on a **Picture** object or on a **World** object. The case for a **Picture** object is relatively simple. The **getGraphics** method is called on the **Picture** object to get a reference to the graphics context for that object *(as type **Graphics**)*. The reference is cast to type **Graphics2D** and saved in the variable named **g2** for later use in drawing the image.

A much more complicated situation

The case for a **World** object in Listing 7 is much more complicated and provides the answer to the earlier question as to why this process erases old images of the turtle with the red shell but doesn't erase the pictures that have been dropped as shown in Figure 1.

Get a reference to a graphics context

Listing 7 calls the **getGraphics** method on the reference to the **World** object. Normally, you might expect this call to return a reference to the graphics context for the world, but that is not the case here.

Every World object contains a Picture object

By default, every **World** object contains a **Picture** object which is used to produce the background for the world. By default, this picture is simply a white image the same size as the world as shown by the background in Figure 1. *(You can easily replace the default picture with a different one.)*

An overridden getGraphics method

The **getGraphics** method of the **World** class is overridden so that it returns a reference to the graphics context for that background **Picture** object instead of returning a reference to the graphics context for the **World** object.

When the **getGraphics** method is called in Listing 7, the returned reference value is cast to type **Graphics2D** and stored in the variable named **g2** for later use. *(I will continue this explanation later.)*

Construct a rotation and translation transform

Listing 8 begins by confirming that the reference stored in **g2** is not null. If it is null, no attempt will be made to draw the picture referred to by the incoming parameter named **dropPicture**.

Listing 8. Construct a rotation and translation transform.

```
    if (g2 != null){

      // save the current transform
      AffineTransform oldTransform =
g2.getTransform();

      // rotate to turtle heading and
translate to xPos
      // and yPos

g2.rotate(Math.toRadians(heading),xPos,yPos);
```

When g2 is not null...

When **g2** is not null, Listing 8 saves the current affine transform for the graphics context referred to by **g2** and then modifies the current transform to match the location and orientation of the turtle. *(I explained the use of the affine transform in some detail in the previous lesson. See Resources for a link to that lesson.)* From this point forward, until the original transform is restored in Listing 10, the modified transform will be applied to any drawing commands that are issued against **g2**.

Draw dropPicture on the graphics context

Listing 9 calls the **drawImage** method to draw the image referred to by **dropPicture** at the specified location on the graphics context referenced by **g2**.

Listing 9. Draw dropPicture on the graphics context.

```

g2.drawImage(dropPicture.getImage(),xPos,yPos,null);
```

Where does that leave us?

> **ImageObserver**
> The last parameter to the **drawImage** method must either be null or must be a referenced to an **ImageObserver** object. In this case, we don't need such an object so we are passing null as the last parameter.

At this point, we have drawn **dropPicture** at a specified location with a specified orientation on another **Picture** object. For the case where the program placed the turtle in a **Picture** object *(as in Figure 2)*, that's pretty much the end of the story.

Not the end of the story for a World object

However, for the case where the program placed the turtle in a **World** object, there is much more to the story. In this case, **g2** is a reference to the **Picture** object owned by the **World** object AND IS not a reference to the **World** object itself.

When the statement in Listing 9 has been executed, the **Picture** object owned by the **World** object has been permanently modified such that the picture referred to by **dropPicture** has been drawn on that picture. *(Except in some very special circumstances, drawing one image on another image is a non-reversible process.)*

Because the **World** object uses its **Picture** object to create a background, **dropPicture** has now become a permanent part of that background.

Repainting the world

Sometime later, for a variety of reasons, the **World** object will need to be repainted. This can result from the execution of program code, or can result from user actions such as minimizing and later restoring the **World**. When the World object decides that it needs to be repainted, it will call its **repaint** method.

Speaking in broad terms, a call to the **repaint** method sends a message to the operating system telling the operating system that the world would like to be repainted as soon as possible. Sometime after that, the operating system will cause the world's overridden **paintComponent** method to be executed. *(Essentially the same thing happens when the need to repaint results from user actions except that there is no call to the **repaint** method by the program.)*

Behavior of the world's paintComponent method

Although the code from the **World** class isn't shown here, the first thing that the world's overridden **paintComponent** method does is to draw its **Picture** object on its own graphics context to create a background image. That completely overwrites or erases everything that was previously drawn there, *(including any Turtle objects that may have been drawn there)*.

At that point, the world's visual representation consists only of the background image contained in the world's **Picture** object. If that **Picture** object has been modified to

include the image referred to by **dropPicture**, the **dropPicture** image will be a permanent part of the background.

Cause the turtles to draw themselves

Then the world's **paintComponent** method cycles through a list of turtles that have been placed in the world, calling the **paintComponent** method belonging to each turtle and passing the graphics context for the world as a parameter to the **paintComponent** method. *(Note that the graphics context for the world's **Picture** object is not passed as a parameter.)*

Behavior of the turtle's paintComponent method

We learned in the previous lesson that the turtle's **paintComponent** method draws a shape that looks something like a turtle *(see Figure 1)* on the incoming graphics context by drawing five overlapping filled ovals in the correct location with the correct orientation. Then it may optionally draw some text on the same graphics context, following which it will call the **paintComponent** method belonging to its **Pen** object to draw all of the line segments in the pen's history on the same graphics context. *(The **Pen** class is very interesting, so I will explain it in a future lesson.)*

The important point

The important point is that the **drop** method draws an image on the graphics context belonging to the **Picture** object that belongs to the world, making a permanent change to that object. Later on, the world will draw the picture on its own graphics context to form a background for whatever else may be drawn there.

On the other hand, the turtle draws itself on the graphics context belonging to the world and does not draw itself on the graphics context belonging to the **Picture** object. Therefore, the next time the world repaints itself by drawing its **Picture** object as a background, all of the turtles that have been drawn on the world's graphics context are overwritten or erased. The result is that each time the **drop** method is called to draw a picture *"on the world"*, that picture actually becomes a permanent part of the world's background. However, the ovals that are drawn on the world's graphic context remain visible only until the next time the world repaints its background.

Remainder of the drop method

The remainder of the **drop** method is shown in Listing 10. Listing 10 restores the original affine transform for the graphics context referred to by **g2** and then causes the pen to draw its history of line segments on the graphics context stored in **g2**.

Listing 10. Remainder of the drop method.

```
      // reset the transformation matrix
```

```
      g2.setTransform(oldTransform);

      //  draw the pen
      pen.paintComponent(g2);
    }
  }//end drop
```

A possible redundancy

I believe that the call to the pen's **paintComponent** method in Listing 10 may be redundant with a similar call in the **paintComponent** method belonging to the turtle. However, there may be some circumstance in which it is necessary to have that call in both locations.

Nothing is really permanent
The World class contains a method named clearBackground that can be called to erase everything in the world's background image.

In any event, placing the call in Listing 10 causes the lines drawn by the pen to become a permanent part of the background for the world when a picture is dropped into the world. The similar call in the turtle's **paintComponent** method causes the lines to be drawn on the world's graphics context instead of the background picture's graphics context.

Methods that cause the turtle to turn

I have identified the following methods that cause the turtle to turn and face in a different direction:

- turn(int degrees)
- turnLeft()
- turnRight()
- turnToFace(int x,int y)
- turnToFace(SimpleTurtle turtle)

The turn(int degrees) method

The **turn(int degrees)** method, which is shown in its entirety in Listing 11, causes a turtle to rotate around its center by a specified number of degrees. A positive value causes the turtle to rotate clockwise and a negative value causes the turtle to rotate counter-clockwise.

Listing 11. The turn(int degrees) method.

```
  public void turn(int degrees){
    this.heading = (heading + degrees) % 360;
    this.updateDisplay();
  }//end turn
```

Not actually true

In truth, this method really doesn't cause the turtle to rotate. Instead, it simply changes the value of an instance variable belonging to the turtle named **heading**. This is one of many variables that maintain the current state of a turtle. This variable keeps track of the direction that the turtle is facing. The value for heading is ultimately used to modify the affine transform to implement a rotation as I explained in the previous lesson *(see Resources)*.

The modulus operation in Listing 11 constrains the angle to the range from -359 degrees to +359 degrees where 0 degrees, 360 degrees, and -360 degrees all mean the same thing.

A model-view-control (MVC) programming paradigm

In case you haven't already recognized it, the **Turtle** class and the **World** class work together to implement a practical form of the MVC programming paradigm. If you are unfamiliar with MVC, don't worry about it. It isn't necessary to understand MVC to understand the library. If you are familiar with MVC, however, you might want to think about how it is implemented here.

The view

In this case, the **World** object is the view. *(Perhaps that is why it implements an interface named **ModelDisplay**.)* Among other methods, the **ModelDisplay** interface declares a method named **modelChanged**, whereby a model *(turtle)* can notify the view *(world)* that the state of the model has changed. The world then has the discretion to decide whether or not it wants to update the display to reflect those changes in the model.

The model

The model is represented by the many state variables in a **Turtle** object *(such as **heading**, **visible**, **xPos**, and **yPos**)* that maintain the state of the object.

The control

The control is represented by many of the methods that can be called to modify the state variables, including but not limited to the following:

- turnToFace
- setVisible
- forward
- moveTo
- turn

Many of those methods modify one or more state variables and then cause the turtle's **updateDisplay** method to be called. The **updateDisplay** method in turn calls the

world's **modelChanged** method to notify the world that the model has changed. For example, the **turn** method in Listing 11 modifies the value of the **heading** variable belonging to the **Turtle** object and then calls the turtle's **updateDisplay** method to cause the **World** object to be notified that the state of the **Turtle** object has changed.

More information
If you would like to learn more about MVC, see *Implementing the Model-View-Controller Paradigm using Observer and Observable* in [Resources]

Updating the view

As explained earlier, when the world object decides to update the view of the turtles, it first creates a new background image for the view. Then it calls the **paintComponent** method on each turtle in sequence. This causes each turtle to draw itself in its current state on the new background.

The turnLeft() and turnRight() methods

These two methods, which are shown in Listing 12, are very simple.

Listing 12. The turnLeft and turnRight methods

```
//Method to turn left
public void turnLeft(){this.turn(-90);}

//Method to turn right
public void turnRight(){this.turn(90);}
```

Each method makes a call to the **turn** method from Listing 11 passing either -90 degrees or 90 degrees as a parameter value.

The turnToFace(int x,int y) method

The purpose of this method, which is shown in Listing 13, is to cause a turtle to turn to face a given point in 2D space specified by a pair of x and y coordinates. This method is much more complex than the other *turn* methods discussed above.

Behavior of the method

The behavior of this method is to:

- Consider a line that joins the turtle and the point in space to be the hypotenuse of a right triangle.
- Determine the length of the base and the length of the opposite side of the right triangle.
- Use trigonometry to compute the near angle in degrees for the right triangle.
- Adjust the angle to be within the range of -180 to +180 degrees based on the sign of the base of the right triangle.

- Store the resulting angle in the state variable named **heading**.
- Notify the world that the state of the turtle has changed.

Along the way, the method is careful to avoid division by zero.

Listing 13. Beginning of the turnToFace(int x,int y) method.

```
  public void turnToFace(int x, int y){
    double dx = x - this.xPos;
    double dy = y - this.yPos;
    double arcTan = 0.0;
    double angle = 0.0;

    // avoid a divide by 0
    if (dx == 0){
      // if below the current turtle
      if (dy > 0) heading = 180;

      // if above the current turtle
      else if (dy < 0) heading = 0;
    }
    // dx isn't 0 so can divide by it
    else{
      arcTan =
Math.toDegrees(Math.atan(dy/dx));
      if (dx < 0) heading = arcTan - 90;
      else heading = arcTan + 90;
    }//end else

    // notify the display that we need to
repaint
    updateDisplay();
  }//end turnToFace
```

If you already understand trigonometry, you should have no problem understanding this method. Otherwise, you will simply have to take it on faith that this method behaves as described above.

The turnToFace(SimpleTurtle turtle) method

This method can be called to cause one turtle to face another turtle.

Listing 14. The turnToFace(SimpleTurtle turtle) method.

```
  /**
   * Method to turn to face another simple
turtle
   */
  public void turnToFace(SimpleTurtle turtle){
    turnToFace(turtle.xPos,turtle.yPos);
```

```
  }//turnToFace
```

This method calls the method from Listing 13, passing the coordinates of the target turtle as parameters.

Run the programs

I encourage you to copy the code from Listing 16 and Listing 17, 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

I began this lesson with an explanation of the **drop** method of the **SimpleTurtle** class.

Then I explained the following methods that cause a turtle to face in a particular direction:

- turn(int degrees)
- turnLeft()
- turnRight()
- turnToFace(int x,int y)
- turnToFace(SimpleTurtle turtle)

Along the way, I explained how the **World** class and the **Turtle** class implement a practical version of the *Model-View-Control* programming paradigm.

What's next?

In the next lesson, I will explain the methods in the following list along with some of the other methods in the multimedia library:

- forward()
- forward(int pixels)
- backward()
- backward(int pixels)
- moveTo(int x,int y)

These are the methods that make it possible for a turtle to move around in either a **World** object or a **Picture** object.

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
- 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
- 340 Multimedia Programming with Java, Getting Started
- 342 Getting Started with the Turtle Class: Multimedia Programming with Java

Complete program listings

Complete listings of the programs discussed in this lesson are shown in Listing 15 through Listing 17 below.

Listing 15. Source code for the SimpleTurtle class.

```
import javax.swing.*;
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.util.Observer;
import java.util.Random;

/**
 * Class that represents a Logo-style turtle.  The
turtle
 * starts off facing north.
 * A turtle can have a name, has a starting x and y
 * position, has a heading, has a width, has a
height,
```

```
 * has a visible flag, has a body color, can have a
shell
 * color, and has a pen.
 * The turtle will not go beyond the model display
or
 * picture boundaries.
 *
 * You can display this turtle in either a picture
or in
 * a class that implements ModelDisplay.
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimpleTurtle{
  ///////////////// fields //////////////////////

  /** count of the number of turtles created */
  private static int numTurtles = 0;

  /** array of colors to use for the turtles */
  private static Color[] colorArray = {Color.green,
              Color.cyan,new
Color(204,0,204),Color.gray};

  /** who to notify about changes to this turtle */
  private ModelDisplay modelDisplay = null;

  /** picture to draw this turtle on */
  private Picture picture = null;

  /** width of turtle in pixels */
  private int width = 15;

  /** height of turtle in pixels */
  private int height = 18;

  /** current location in x (center) */
  private int xPos = 0;

  /** current location in y (center) */
  private int yPos = 0;

  /** heading angle */
  private double heading = 0;  // default is facing
north

  /** pen to use for this turtle */
  private Pen pen = new Pen();

  /** color to draw the body in */
  private Color bodyColor = null;

  /** color to draw the shell in */
  private Color shellColor = null;
```

```
  /** color of information string */
  private Color infoColor = Color.black;

  /** flag to say if this turtle is visible */
  private boolean visible = true;

  /** flag to say if should show turtle info */
  private boolean showInfo = false;

  /** the name of this turtle */
  private String name = "No name";

  ////////////////// constructors
///////////////////

  /**
   * Constructor that takes the x and y position for
the
   * turtle
   * @param x the x pos
   * @param y the y pos
   */
  public SimpleTurtle(int x, int y){
    xPos = x;
    yPos = y;
    bodyColor =
               colorArray[numTurtles %
colorArray.length];
    setPenColor(bodyColor);
    numTurtles++;
  }//end constructor

  /**
   * Constructor that takes the x and y position and
the
   * model displayer
   * @param x the x pos
   * @param y the y pos
   * @param display the model display
   */
  public SimpleTurtle(int x, int y, ModelDisplay
display){
    this(x,y); // invoke constructor that takes x
and y
    modelDisplay = display;
    display.addModel(this);
  }//end constructor

  /**
   * Constructor that takes a model display and adds
   * a turtle in the middle of it
   * @param display the model display
   */
  public SimpleTurtle(ModelDisplay display){
    // invoke constructor that takes x and y
    this((int) (display.getWidth() / 2),
```

```
         (int) (display.getHeight() / 2));
    modelDisplay = display;
    display.addModel(this);
  }//end constructor

  /**
   * Constructor that takes the x and y position and
the
   * picture to draw on
   * @param x the x pos
   * @param y the y pos
   * @param picture the picture to draw on
   */
  public SimpleTurtle(int x, int y, Picture
picture){
    this(x,y); // invoke constructor that takes x
and y
    this.picture = picture;
    this.visible = false;//default is not to see
turtle
  }//end constructor

  /**
   * Constructor that takes the
   * picture to draw on and will appear in the
middle
   * @param picture the picture to draw on
   */
  public SimpleTurtle(Picture picture){
    // invoke constructor that takes x and y
    this((int) (picture.getWidth() / 2),
         (int) (picture.getHeight() / 2));
    this.picture = picture;
    this.visible = false;//default is not to see
turtle
  }//end constructor

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

  /**
   * Get the distance from the passed x and y
location
   * @param x the x location
   * @param y the y location
   */
  public double getDistance(int x, int y){
    int xDiff = x - xPos;
    int yDiff = y - yPos;
    return (Math.sqrt((xDiff * xDiff) + (yDiff *
yDiff)));
  }//end getDistance

  /**
   * Method to turn to face another simple turtle
   */
```

```
  public void turnToFace(SimpleTurtle turtle){
    turnToFace(turtle.xPos,turtle.yPos);
  }//turnToFace

   /**
   * Method to turn towards the given x and y
   * @param x the x to turn towards
   * @param y the y to turn towards
   */
  public void turnToFace(int x, int y){
    double dx = x - this.xPos;
    double dy = y - this.yPos;
    double arcTan = 0.0;
    double angle = 0.0;

    // avoid a divide by 0
    if (dx == 0){
      // if below the current turtle
      if (dy > 0) heading = 180;

      // if above the current turtle
      else if (dy < 0) heading = 0;
    }
    // dx isn't 0 so can divide by it
    else{
      arcTan = Math.toDegrees(Math.atan(dy/dx));
      if (dx < 0) heading = arcTan - 90;
      else heading = arcTan + 90;
    }//end else

    // notify the display that we need to repaint
    updateDisplay();
  }//end turnToFace

  /**
   * Method to get the picture for this simple
turtle
   * @return the picture for this turtle (may be
null)
   */
  public Picture getPicture() { return this.picture;
}

  /**
   * Method to set the picture for this simple
turtle
   * @param pict the picture to use
   */
  public void setPicture(Picture pict){
    this.picture = pict;
  }//end setPicture

  /**
   * Method to get the model display for this simple
   * turtle.
   * @return the model display if there is one else
```

```
null
   */
  public ModelDisplay getModelDisplay(){
    return this.modelDisplay;
  }//end getModelDisplay

  /**
   * Method to set the model display for this simple
   * turtle.
   * @param theModelDisplay the model display to use
   */
  public void setModelDisplay(
                            ModelDisplay
theModelDisplay){
    this.modelDisplay = theModelDisplay;
  }//end setModelDisplay

  /**
   * Method to get value of show info
   * @return true if should show info, else false
   */
  public boolean getShowInfo(){return
this.showInfo;}

  /**
   * Method to show the turtle information string
   * @param value the value to set showInfo to
   */
  public void setShowInfo(boolean value){
    this.showInfo = value;
  }//end setShowInfo

  /**
   * Method to get the shell color
   * @return the shell color
   */
  public Color getShellColor(){
    Color color = null;
    if(this.shellColor == null && this.bodyColor !=
null)
      color = bodyColor.darker();
    else color = this.shellColor;
    return color;
  }//end getShellColor

  /**
   * Method to set the shell color
   * @param color the color to use
   */
  public void setShellColor(Color color){
    this.shellColor = color;
  }//setShellColor

  /**
   * Method to get the body color
   * @return the body color
```

```
   */
  public Color getBodyColor(){return
this.bodyColor;}

  /**
   * Method to set the body color which
   * will also set the pen color
   * @param color the color to use
   */
  public void setBodyColor(Color color){
    this.bodyColor = color;
    setPenColor(this.bodyColor);
  }//end setBodyColor

  /**
   * Method to set the color of the turtle.
   * This will set the body color
   * @param color the color to use
   */
  public void setColor(Color color){
    this.setBodyColor(color);
  }//end setColor

  /**
   * Method to get the information color
   * @return the color of the information string
   */
  public Color getInfoColor(){return
this.infoColor;}

  /**
   * Method to set the information color
   * @param color the new color to use
   */
  public void setInfoColor(Color color){
    this.infoColor = color;
  }//setInfoColor

  /**
   * Method to return the width of this object
   * @return the width in pixels
   */
  public int getWidth(){return this.width;}

  /**
   * Method to return the height of this object
   * @return the height in pixels
   */
  public int getHeight(){return this.height;}

  /**
   * Method to set the width of this object
   * @param theWidth in width in pixels
   */
  public void setWidth(int theWidth){
    this.width = theWidth;
```

```
  }//end setWidth

  /**
   * Method to set the height of this object
   * @param theHeight the height in pixels
   */
  public void setHeight(int theHeight){
    this.height = theHeight;
  }//end setHeight

  /**
   * Method to get the current x position
   * @return the x position (in pixels)
   */
  public int getXPos(){return this.xPos;}

  /**
   * Method to get the current y position
   * @return the y position (in pixels)
   */
  public int getYPos(){return this.yPos;}

  /**
   * Method to get the pen
   * @return the pen
   */
  public Pen getPen(){return this.pen;}

  /**
   * Method to set the pen
   * @param thePen the new pen to use
   */
  public void setPen(Pen thePen){this.pen = thePen;}

  /**
   * Method to check if the pen is down
   * @return true if down else false
   */
  public boolean isPenDown(){return
this.pen.isPenDown();}

  /**
   * Method to set the pen down boolean variable
   * @param value the value to set it to
   */
  public void setPenDown(boolean value){
    this.pen.setPenDown(value);
  }//end setPenDown

  /**
   * Method to lift the pen up
   */
  public void penUp(){this.pen.setPenDown(false);}

  /**
   * Method to set the pen down
```

```
   */
  public void penDown(){this.pen.setPenDown(true);}

  /**
   * Method to get the pen color
   * @return the pen color
   */
  public Color getPenColor(){return
this.pen.getColor();}

  /**
   * Method to set the pen color
   * @param color the color for the pen ink
   */
  public void setPenColor(Color color){
    this.pen.setColor(color);
  }//end setPenColor

  /**
   * Method to set the pen width
   * @param width the width to use in pixels
   */
  public void setPenWidth(int width){
    this.pen.setWidth(width);
  }//end setPenWidth

  /**
   * Method to get the pen width
   * @return the width of the pen in pixels
   */
  public int getPenWidth(){return
this.pen.getWidth();}

  /**
   * Method to clear the path (history of
   * where the turtle has been)
   */
  public void clearPath(){
    this.pen.clearPath();
  }//end clearPath

  /**
   * Method to get the current heading
   * @return the heading in degrees
   */
  public double getHeading(){return this.heading;}

  /**
   * Method to set the heading
   * @param heading the new heading to use
   */
  public void setHeading(double heading){
    this.heading = heading;
  }//end setHeading

  /**
```

```
   * Method to get the name of the turtle
   * @return the name of this turtle
   */
  public String getName(){return this.name;}

  /**
   * Method to set the name of the turtle
   * @param theName the new name to use
   */
  public void setName(String theName){
    this.name = theName;
  }//end setName

  /**
   * Method to get the value of the visible flag
   * @return true if visible else false
   */
  public boolean isVisible(){return this.visible;}

  /**
   * Method to hide the turtle (stop showing it)
   * This doesn't affect the pen status
   */
  public void hide(){this.setVisible(false);}

  /**
   * Method to show the turtle (doesn't affect
   * the pen status
   */
  public void show(){this.setVisible(true);}

  /**
   * Method to set the visible flag
   * @param value the value to set it to
   */
  public void setVisible(boolean value){
    // if the turtle wasn't visible and now is
    if (visible == false && value == true){
      // update the display
      this.updateDisplay();
    }//end if

    // set the visibile flag to the passed value
    this.visible = value;
  }//end setVisible

  /**
   * Method to update the display of this turtle and
   * also check that the turtle is in the bounds
   */
  public synchronized void updateDisplay(){
    // check that x and y are at least 0
    if (xPos < 0) xPos = 0;
    if (yPos < 0) yPos = 0;

    // if picture
```

```
    if (picture != null){
      if (xPos >= picture.getWidth())
        xPos = picture.getWidth() - 1;
      if (yPos >= picture.getHeight())
        yPos = picture.getHeight() - 1;
      Graphics g = picture.getGraphics();
      paintComponent(g);
    }//end if
    else if (modelDisplay != null){
      if (xPos >= modelDisplay.getWidth())
        xPos = modelDisplay.getWidth() - 1;
      if (yPos >= modelDisplay.getHeight())
        yPos = modelDisplay.getHeight() - 1;
      modelDisplay.modelChanged();
    }//end else if
  }//end updateDisplay

  /**
   * Method to move the turtle foward 100 pixels
   */
  public void forward(){forward(100);}

  /**
   * Method to move the turtle forward the given
number
   * of pixels
   * @param pixels the number of pixels to walk
forward in
   * the heading direction
   */
  public void forward(int pixels){
    int oldX = xPos;
    int oldY = yPos;

    // change the current position
    xPos = oldX + (int)(pixels *
Math.sin(Math.toRadians(

heading)));
    yPos = oldY + (int)(pixels * -
Math.cos(Math.toRadians(

heading)));

    // add a move from the old position to the new
    // position to the pen
    pen.addMove(oldX,oldY,xPos,yPos);

    // update the display to show the new line
    updateDisplay();
  }//end forward

  /**
   * Method to go backward by 100 pixels
   */
  public void backward(){backward(100);}
```

```
  /**
   * Method to go backward a given number of pixels
   * @param pixels the number of pixels to walk
backward
   */
  public void backward(int pixels){
    forward(-pixels);
  }//end backward

  /**
   * Method to move to turtle to the given x and y
   * location
   * @param x the x value to move to
   * @param y the y value to move to
   */
  public void moveTo(int x, int y){
    this.pen.addMove(xPos,yPos,x,y);
    this.xPos = x;
    this.yPos = y;
    this.updateDisplay();
  }//end moveTo

  /**
   * Method to turn left
   */
  public void turnLeft(){this.turn(-90);}

  /**
   * Method to turn right
   */
  public void turnRight(){this.turn(90);}

  /**
   * Method to turn the turtle the passed degrees
   * use negative to turn left and pos to turn right
   * @param degrees the amount to turn in degrees
   */
  public void turn(int degrees){
    this.heading = (heading + degrees) % 360;
    this.updateDisplay();
  }//end turn

  /**
   * Method to draw a passed picture at the current
turtle
   * location and rotation in a picture or model
display
   * @param dropPicture the picture to drop
   */
  public synchronized void drop(Picture
dropPicture){
    Graphics2D g2 = null;

    // only do this if drawing on a picture
    if (picture != null)
```

```
      g2 = (Graphics2D) picture.getGraphics();
    else if (modelDisplay != null)
      g2 = (Graphics2D) modelDisplay.getGraphics();

    // if g2 isn't null
    if (g2 != null){

      // save the current tranform
      AffineTransform oldTransform =
g2.getTransform();

      // rotate to turtle heading and translate to
xPos
      // and yPos
      g2.rotate(Math.toRadians(heading),xPos,yPos);

      // draw the passed picture

g2.drawImage(dropPicture.getImage(),xPos,yPos,null);

      // reset the tranformation matrix
      g2.setTransform(oldTransform);

      //  draw the pen
      pen.paintComponent(g2);
    }
  }//end drop

  /**
   * Method to paint the turtle
   * @param g the graphics context to paint on
   */
  public synchronized void paintComponent(Graphics
g){
    // cast to 2d object
    Graphics2D g2 = (Graphics2D) g;

    // if the turtle is visible
    if (visible){
      // save the current tranform
      AffineTransform oldTransform =
g2.getTransform();

      // rotate the turtle and translate to xPos and
yPos
      g2.rotate(Math.toRadians(heading),xPos,yPos);

      // determine the half width and height of the
shell
      int halfWidth = (int) (width/2); // of shell
      int halfHeight = (int) (height/2); // of shell
      int quarterWidth = (int) (width/4); // of
shell
      int thirdHeight = (int) (height/3); // of
shell
      int thirdWidth = (int) (width/3); // of shell
```

```
      // draw the body parts (head)
      g2.setColor(bodyColor);
      g2.fillOval(xPos - quarterWidth,
                  yPos - halfHeight - (int)
(height/3),
                  halfWidth, thirdHeight);
      g2.fillOval(xPos - (2 * thirdWidth),
                  yPos - thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos - (int) (1.6 * thirdWidth),
                  yPos + thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos + (int) (1.3 * thirdWidth),
                  yPos - thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos + (int) (0.9 * thirdWidth),
                  yPos + thirdHeight,
                  thirdWidth,thirdHeight);

      // draw the shell
      g2.setColor(getShellColor());
      g2.fillOval(xPos - halfWidth,
                  yPos - halfHeight, width, height);

      // draw the info string if the flag is true
      if (showInfo) drawInfoString(g2);

      // reset the tranformation matrix
      g2.setTransform(oldTransform);
    }//end if

    //  draw the pen
    pen.paintComponent(g);
  }//end paintComponent

  /**
   * Method to draw the information string
   * @param g the graphics context
   */
  public synchronized void drawInfoString(Graphics
g){
    g.setColor(infoColor);
    g.drawString(
             this.toString(),xPos + (int)
(width/2),yPos);
  }//end drawInfoString

  /**
   * Method to return a string with information
   * about this turtle
   * @return a string with information about this
object
   */
  public String toString(){
    return this.name + " turtle at " + this.xPos +
```

```
", " +
      this.yPos + " heading " + this.heading + ".";
  }//end toString

} // end of class
```

Listing 16. Source code for the program named Java344a.

```
/*Java344a
 * The purpose of this program is to illustrate the
use
 * of several different methods of the Turtle class.
 *
 * Also illustrates dropping pictures at the current
 * position and orientation of the turtle.
 *
 * Moves a Turtle object around on a World and drops
 * several copies of a small Picture along the way.
 */
import java.awt.Color;
public class Main{
  public static void main(String[] args){
    //Following statement eliminates necessity to
manually
    // establish location of media files. Modify this
to
    // point to the mediasources folder on your
machine.

FileChooser.setMediaPath("M:/Ericson/mediasources/");

    //Instantiate an object of a small picture of a
turtle
    // with a white background.
    Picture p2 = new Picture("turtle.jpg");

    //Note that the Turtle object is different from
the
    // image of the turtle in the Picture object.
    World mars = new World(400,500);
    Turtle joe = new Turtle(300,400,mars);
    joe.setShellColor(Color.RED);
    joe.setPenColor(Color.BLUE);
    joe.setPenWidth(2);
    joe.drop(p2);//Draw a small picture
    joe.forward(90);
    joe.drop(p2);
    joe.turn(-30);
    joe.forward();
    joe.drop(p2);
    joe.turn(-30);
    joe.forward();
```

```
    joe.drop(p2);
  }//end main

}//end class
```

Listing 17. Source code for the program named Java344b.

```
/*Java344b
 * The purpose of this program is to illustrate the
use
 * of several different methods of the Turtle class,
as
 * well as to illustrate the placement of a Turtle
object
 * on a Picture object.
 *
 * Also illustrates dropping pictures at the current
 * position and orientation of the turtle.
 *
 * Moves a Turtle object around on a Picture and
drops
 * several copies of a smaller Picture along the way.
 */
import java.awt.Color;
public class Main{
  public static void main(String[] args){
    //Following statement eliminates necessity to
manually
    // establish location of media files. Modify this
to
    // point to the mediasources folder on your
machine.

FileChooser.setMediaPath("M:/Ericson/mediasources/");

    //Instantiate two objects of Picture class, each
of
    // which encapsulates an image.
    Picture p1 = new
Picture("butterfly1.jpg");//large
    Picture p2 = new Picture("turtle.jpg");//small

    //Note that the Turtle object is different from
the
    // image of the turtle in the Picture object.
    Turtle joe = new Turtle(300,400,p1);
    joe.setShellColor(Color.RED);
    joe.setPenColor(Color.BLUE);
    joe.setPenWidth(2);
    joe.setVisible(true);
    joe.drop(p2);//Draw a small picture
    joe.forward(90);
```

```
    joe.drop(p2);
    joe.turn(-30);
    joe.forward();
    joe.drop(p2);
    joe.turn(-30);
    joe.forward();
    joe.drop(p2);

    p1.show();
  }//end main

}//end class
```

Copyright

Copyright 2008, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

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-