Getting Started with the Turtle Class: Multimedia Programming with Java

*Learn about the behavior of the **Turtle** class, and start learning about the behavior of its superclass named **SimpleTurtle**.*

Published: December 2, 2008
By Richard G. Baldwin

Java Programming Notes # 342

Preface

General

This is the second 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

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.

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 the 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

In this lesson, I will explain the behavior of the **Turtle** class, and will partially explain the behavior of its superclass named **SimpleTurtle**.

In particular, I will explain the purpose of a number of the instance and class variables that are declared in the **SimpleTurtle** class. I will also explain all five overloaded constructors in the **SimpleTurtle** class.

There are more than fifty methods defined in the **SimpleTurtle** class. Arguably, the most complex method is the method named **paintComponent**. The complexity of most of the other methods pales in comparison with the **paintComponent** method.

The **paintComponent** method and the other two methods in the following list are the methods that are primarily responsible for displaying a turtle and its pen on either a **World** object or a **Picture** object. Thus, these three methods are critical to the **SimpleTurtle** class.

- updateDisplay()
- paintComponent(Graphics g)
- drop(Picture dropPicture)

I will explain the **updateDisplay** method and the **paintComponent** method in this lesson. I will explain the **drop** method and many of the remaining methods in the next lesson.

Discussion and sample code

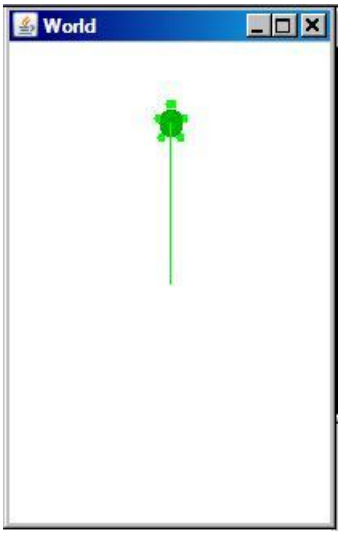
A Turtle class example

I'm going to begin by explaining a sample program named **StartupTest01** from the previous lesson. This program places a **Turtle** object in a **World** object and then causes the turtle to move forward and draw a line.

Source code for the program

The source code for this program is shown in its entirety in Listing 17 near the end of the lesson. The screen output produced by the program is shown in Figure 1.

Figure 1. Output from StartupTest01.



The interesting part of the source code

The interesting part of the source code is repeated for viewing convenience in Listing 1. As you can see, this is a very simple program consisting only of a **main** method that contains three statements.

Listing 1. Main class definition for the program named StartupTest01.

```
public class Main{
  public static void main(String[] args){
    World mars = new World(200,300);
    Turtle joe = new Turtle(mars);
    joe.forward();
  }//end main

}//end class
```

Create the world

The first statement in Listing 1 instantiates a new object of the class **World** *(from the multimedia library)* with dimensions of 200 pixels by 300 pixels. *(I will explain the **World** class in a future lesson.)* A reference to the **World** object is stored in the variable named **mars**. This **World** object is represented by the **JFrame** object *(without the turtle)* shown in Figure 1.

Create the turtle

You will see shortly that there are four overloaded constructors for the **Turtle** class. Two of those constructors require a reference to an object of the class **Picture** as an incoming parameter. *(I will explain the **Picture** class in a future lesson.)* The other two constructors require an incoming parameter that is a reference to an object that implements the **ModelDisplay** interface. According to the documentation for the version of the library that I am currently using, the only class that implements the **ModelDisplay** interface is the class named **World**. Therefore, as of this time, the constructors for the **Turtle** class require an incoming parameter that is either a reference to a **Picture** object or a **World** object.

The second statement in Listing 1 instantiates an object of the **Turtle** class, passing a reference to the **World** object as a parameter. This constructor causes the turtle to appear in the center of the world shown in Figure 1.

Move the turtle forward

Also as you will see later, the **Turtle** class inherits a method named **forward()** from its immediate superclass named **SimpleTurtle**. This method causes the turtle to move forward by 100 pixels. Thus, the third statement in Listing 1 causes the turtle to move from the center of the world to the position shown in Figure 1. As you can see from Figure 1, when a turtle moves forward, it draws a thin line by default. I will have more to say about this later.

The Turtle class

A complete listing of the **Turtle** class is provided in Listing 18 near the end of the lesson. This listing is essentially the same as the original class in the multimedia class library. The only changes that I made were cosmetic in nature, mainly to force the source code to fit into this narrow publication format.

Two overloaded constructors for the Turtle class

Listing 2 shows the beginning of the class definition and two of the four overloaded constructors for the **Turtle** class. *(Note that the **Turtle** class extends the **SimpleTurtle** class.)*

Listing 2. Two overloaded constructors for the Turtle class.

```
public class Turtle extends SimpleTurtle{

  public Turtle (ModelDisplay modelDisplay){
    //Let the parent constructor handle it
    super(modelDisplay);
  }//end constructor

  public Turtle(int x,int y,ModelDisplay
modelDisplayer){
    super(x,y,modelDisplayer);
  }//end constructor
```

The constructor with a single ModelDisplay parameter

The first constructor shown in Listing 2 receives a single incoming parameter of type **ModelDisplay** and uses the **super** keyword to pass it along to the corresponding constructor for the superclass named **SimpleTurtle**. We will understand more about what is going on here when we examine the constructors for the **SimpleTurtle** class.

Note that this is the constructor that was used to instantiate the **Turtle** object in Listing 1, which passed a reference to an object of the **World** class to the **Turtle** constructor. As you will see later, this constructor places the turtle in the center of the world by default.

The constructor with three parameters

The second constructor in Listing 2 requires a pair of horizontal and vertical coordinates in addition to the **ModelDisplay** object. All three parameters are simply passed to the corresponding constructor in the **SimpleTurtle** class. As you will see later, this constructor places the turtle at a location in the world specified by the coordinate values.

Turtle constructors that require a Picture object

It is also possible to place a turtle in a **Picture** object and use it to draw on the picture. The two constructors that are used to place a turtle in a picture are shown in Listing 3.

Listing 3. Turtle constructors that require a Picture object.

```
  public Turtle (int x, int y, Picture
picture){
    super(x,y,picture);
  }//end constructor

  public Turtle (Picture p){
    super(p);
  }//end constructor

}//end Turtle class definition
```

As before, both constructors pass their incoming parameters to the corresponding constructors for the superclass named **SimpleTurtle**. Also as before, one constructor causes the turtle to be placed in the center of the picture and the other constructor makes it possible for the programmer to specify the initial location of the turtle in the picture.

The end of the Turtle class definition

Listing 3 also signals the end of the class definition for the **Turtle** class.

The SimpleTurtle class

A complete listing of the **SimpleTurtle** class is provided in Listing 19. Once again, this listing is essentially the same as the original class in the multimedia class library. The only changes that I made were cosmetic in nature, mainly to force the source code to fit into this narrow publication format.

According to the author, this class represents a Logo-style turtle, which 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** object or in an object that implements **ModelDisplay** such as a **World** object.

Will discuss in fragments

As you can see from Listing 19, this is a large class. As is my custom, I will break the class down and discuss it in fragments.

Listing 4 shows the beginning of the **SimpleTurtle** class along with a large number of variable declarations.

Listing 4. Beginning of SimpleTurtle class with variable declarations.

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

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";
```

This class extends **Object** by default, thereby inheriting only the eleven methods that are defined in the **Object** class.

The purpose of most of the variables is adequately explained by the embedded comments. In addition, the purpose of the variables will become clearer in the following discussion of the behavior of an object of the **SimpleTurtle** class.

A common constructor

The first of five overloaded constructors is shown in Listing 5.

Listing 5. A common constructor.

```
  public SimpleTurtle(int x, int y){
    xPos = x;
    yPos = y;
    bodyColor =
                colorArray[numTurtles %
colorArray.length];
    setPenColor(bodyColor);
    numTurtles++;
  }//end constructor
```

This is a *common* constructor. By this, I mean that this constructor is called by each of the other four constructors.

This constructor sets the values of four properties of the new turtle and increments a static turtle counter variable by:

- Receiving and saving the initial x and y position coordinates for the new turtle.
- Setting the body color for the new turtle to one of four colors by cycling through the following four colors:
  - green
  - cyan
  - new Color(204,0,204)
  - gray
- Setting the color of the pen to the body color.
- Incrementing the count of the number of turtles.

Constructor for a specified position in a World object

As mentioned earlier, a turtle can be displayed either in an object of the class **Picture**, or in an object that implements the **ModelDisplay** interface *(World)*. For the version of the multimedia class library that I am using, the only class that implements the **ModelDisplay** interface is the class named **World**. *(Presumably, the authors are leaving open the option to display turtles in objects of currently undefined classes that implement the **ModelDisplay** interface.)*

The addModel method of a World object

I will explain both the **World** class and the **Picture** class in a future lesson. What you need to know for now is that an object of the **World** class has a method named **addModel**. This method receives an incoming parameter of type **Object** and adds it to an **ArrayList** collection that is constrained through generics to store only objects of type **Turtle**. Then, depending on the value of a **boolean** variable belonging to the **World** object, the **World** object may or may not repaint itself on the screen to display the turtle.

The constructor in Listing 6 receives the initial x and y position coordinates along with a reference to the **World** in which the turtle is to be displayed.

Listing 6. Constructor for a specified position in a World object.

```
  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
```

Call the common constructor

This constructor begins by using the **this** keyword to call the common constructor shown in Listing 5, passing the x and y position coordinates as parameters. If you are unfamiliar with this usage of the keyword **this**, see *The Essence of OOP using Java, The this and super Keywords* in Resources.

Save a reference to the world

When the common constructor returns, Listing 6 saves the incoming reference to the **World** object in an instance variable named **modelDisplay** for future reference. In other words, each turtle knows which world it belongs to.

Add the turtle to the world

Finally, Listing 6 calls the **addModel** method of the **World** object to cause this turtle object to be added to the list of turtles being displayed by that world.

Constructor for default position in a World object

The constructor in Listing 7 is similar to the constructor in Listing 6 with the difference being that instead of receiving the initial x and y position coordinates, they are computed to be the center of the **World** object.

Listing 7. Constructor for default position in a World object.

```
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 for a specified position in a Picture object

The default value for the variable named **visible** in Listing 4 is **true**. As a result, when you add a turtle to a world and repaint the world, the turtle is visible by default.

Apparently the primary purpose of the use of **Turtle** objects with **Picture** objects is not to display turtles in pictures, but rather is to use a turtle to make line drawings on a picture.

The constructor in Listing 8 is similar to the constructor in Listing 6, but there are differences.

Listing 8. Constructor for a specified position in a Picture object.

```
  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
```

Call the common constructor

As before, Listing 8 calls the common constructor shown in Listing 5 for the purpose discussed earlier. When the common constructor returns, Listing 8 saves a reference to the **Picture** object on which it is to draw.

Then unlike the case for the **World** object shown in Listing 6, Listing 8 causes the turtle to be invisible by default. Also, unlike the case for the **World** object, the constructor in Listing 8 does not add itself to the picture. *(We will learn a little more about how the turtle behaves relative to a picture later in this lesson and will learn even more when we discuss the **Picture** class in a future lesson.)*

Constructor for the default position in a Picture object

Finally, Listing 9 shows the constructor for an invisible turtle that is positioned in the center of a specified **Picture** object.

Listing 9. Constructor for the default position in a Picture object.

```
  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
```

By now, it shouldn't be necessary for me to provide a further explanation of Listing 9.

The painting and drawing methods

More than fifty methods are defined in the **SimpleTurtle** class. The code in some of the methods is very simple while the code in other methods is somewhat complex.

Fortunately, it is possible to group the methods into several groups of related methods in order to organize them in your mind. Perhaps the most basic group and arguably the

most complex group includes methods that cause the turtle and other images to be displayed on the world or on the picture. That group includes the following methods:

- updateDisplay()
- paintComponent(Graphics g)
- drop(Picture dropPicture)

The **updateDisplay** method is called by many other methods to cause the turtle or the line produced by the turtle's pen to be displayed if certain conditions are met. When it is determined that the display does need to be updated, the **paintComponent** method is called to control how the turtle or the line produced by the turtle's pen is displayed on the screen.

Apparently paintComponent is not an overridden method

Normally, I would refer to the **paintComponent** method as an *overridden* method because it is defined in many classes *(such as the JComponent class)* and inherited in and overridden in subclasses. However, the **SimpleTurtle** class extends the **Object** class and therefore does not inherit a method named **paintComponent**. In this case, the **paintComponent** method is a stand-alone method that is intended to do essentially the same thing as overridden versions of the method in other classes.

The drop method

The **drop** method is different from the other two. It doesn't draw a turtle. Instead it draws a **Picture** object on the primary **Picture** object or on the **World** object at the turtle's current location and with the same orientation as the turtle. This makes it possible to cause some other image to be displayed at the current location of a turtle. You could use this, for example to create a mosaic of images.

The updateDisplay method

The **updateDisplay** method, which begins in Listing 10, causes the display of the turtle to be updated under certain conditions and also makes certain that the turtle is within the bounds of the world or the picture.

The synchronized keyword
If you are unfamiliar with multi-threaded programming and the *synchronized* keyword, see *Threads of Control* in Resources.

Listing 10. Beginning of the updateDisplay method.

```
public synchronized void updateDisplay(){
  // check that x and y are at least 0
  if (xPos < 0) xPos = 0;
  if (yPos < 0) yPos = 0;
```

The **updateDisplay** method begins in Listing 10 by setting the position coordinate values to 0 if they are negative.

Update display for a picture

Listing 11 updates the display for the case where the turtle is being used to draw on a picture.

Listing 11. Update display for a 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
```

Listing 11 begins by modify the x and y position coordinates if the turtle is not within the bounds of the picture. This makes certain that the turtle remains within the bounds of the picture.

Get a Graphics object on the Picture

Then Listing 11 calls the **getGraphics** method on the **Picture** object to get a graphics object on which to draw the turtle.

Conceptually, you can think of the **Graphics** object as representing the portion of the screen that currently belongs to your program. Material that you draw on the **Graphics** object will appear on the screen.

Java screen painting methodology
If you are unfamiliar with the callback mechanism that Java uses to cause the screen to be repainted, go to Google and search for the following keywords.

baldwin java repaint

This will expose numerous tutorials that I have written on the topic.

Call paintComponent method

Finally, Listing 11 calls the **paintComponent** method to control how the turtle or the line produced by the turtle's pen is displayed on the screen. I will have much more to say about the **paintComponent** method later.

Update display for a world

The code in Listing 12 is executed for the case where the turtle has been added to a **World** object.

Listing 12. Update display for a world.

```
    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
```

As before, Listing 12 begins by guaranteeing that the position of the turtle is within the bounds of the world.

Make a callback to the world

Then, unlike the case in Listing 11, Listing 12 doesn't cause the turtle object to be repainted. Instead, Listing 12 calls the **modelChanged** method on the **World** object to notify the world that some aspect of the image of the turtle has changed. It is left up to the world object to decide whether or not to redraw the turtle at that point in time.

Either then, or sometime later, the world object causes the turtle's **paintComponent** method to be called to cause it to be redrawn in its then-current configuration.

The paintComponent method

The beginning of the **paintComponent** method is shown in Listing 13.

Listing 13. Beginning of the paintComponent method.

```
public synchronized void paintComponent(Graphics g){
  Graphics2D g2 = (Graphics2D) g;
```

This is arguably the most complex method in the entire **SimpleTurtle** class.

If you examine the code in this class and the class named **World**, you will find that every time the **paintComponent** method is called on a turtle, a reference to an object of type **Graphics** is passed as a parameter. *(Otherwise, a type mismatch error would occur.)* The **Graphics** object either represents the graphics context for an object of type **World** or the graphics context for an object of type **Picture**. *(See Listing 11 for example.)* As I described earlier, this **Graphics** object can be thought of as representing the object onto with material is to be drawn.

Cast the reference to type Graphics2D

Listing 13 immediately casts the reference from type **Graphics** to type **Graphics2D** and saves the reference in a variable named **g2**. What is this all about? For the whole story

on this, see the thirteen lessons beginning with lesson number 300 in Resources. Briefly, however, the **Graphics2D** class is a subclass of the Graphics class, which provides enhanced drawing capability that is not provided by the **Graphics** class.

The reference that is received by the **paintComponent** method is actually a reference to an object of the **Graphics2D** class, but in order to maintain backward compatibility with legacy code, it is passed as type **Graphics**. To access the expanded capability, you must cast a **Graphics** reference to a **Graphics2D** reference before calling methods from the **Graphics2D** class on that reference.

What is an affine transform?

You will need to understand *affine transforms* to really understand the code that we are about to encounter. You can learn all about *affine transforms* in my earlier lesson titled *Java 2D Graphics, Simple Affine Transforms (see Resources)*. I'm not going to repeat that information here, but I will attempt to describe the use of an affine transform to draw the turtle in a specific position with a specific orientation.

Although Figure 1 shows the turtle facing due north, we will explore methods of the **SimpleTurtle** class later that make it possible to cause a turtle to face in any direction while being located anywhere inside the world or the picture.

As you will see shortly, the **paintComponent** method actually draws the turtle as a set of six overlapping filled ovals. The shell is a large filled oval while the legs are constructed of smaller filled ovals. Maybe it would make it easier to understand what is going on to think of the turtle as being constructed as a set of six overlapping filled rectangles with a large rectangle for the shell and five smaller rectangles for the head and legs.

How would you...?

The question is, how would you draw a rectangle for which the sides are not parallel to the horizontal and vertical axes? The answer is that you would probably do something like the following:

- Pretend that the rectangle is located at the origin.
- Compute the four coordinate values for the corners that would be required to draw a rectangle that is centered on the origin with the sides parallel to the horizontal and vertical axes.
- Use trigonometry to re-compute the coordinate values for the corners that would be required to draw the rectangle if it were rotated by a specified number of degrees around the origin. *(This is often called rotation.)*
- Add appropriate x and y values to the coordinate values for the corners to cause the rectangle to be located somewhere other than at the origin. *(This is often called translation.)*

- Draw *(potentially sloping)* straight lines to connect the corners, resulting in an image that represents a rotated rectangle located somewhere other than at the origin.

This is a transform process

The process shown above is often called a *transform*. Although a lot more coordinate values must be dealt with to translate and rotate a filled oval than is required to translate and rotate a rectangle, the process is essentially the same regardless of the geometric shape.

Creating a transform

Using various methods of the **Graphics2D** class, you can create a *transform* that will automatically be applied to all geometric shapes that you draw on a graphics context *(a Graphics2D object)*. In this case, we are interested in *translation* and *rotation* only, but you can also incorporate *scaling* and *shear* into the transform if needed.

A default affine transform

Every **Graphics2D** object contains an **AffineTransform** object that automatically performs the following operations whenever you draw on the object:

- Scaling
- Translation
- Shear
- Rotation

By default, the transform is a *"do nothing"* transform. In other words, by default, your drawing data is not scaled, translated, sheared, or rotated. However, if you need to do any of those operations, you can modify the default transform to perform any or all of them. In this case, we will modify the default transform to apply rotation and translation to the coordinate values that are used to draw the filled ovals that represent the turtle.

Is the turtle visible?

Most of the remaining code in the **paintComponent** method is executed only if the turtle is intended to be visible. However, as you will see later, the code required to draw the line produced by the pen is executed even if the turtle is invisible.

Modify the current transform

Listing 14 begins by getting and saving a reference to the affine transform that currently belongs to the graphics context. The transform will be restored to this value later before the **paintComponent** method returns.

Listing 14. Modify the current transform.

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

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

Then Listing 14 calls the **rotate** method to cause the current transform to be modified to one that will apply a *rotation* equal to the number of degrees stored in the variable named **heading** and a *translation* to the current position of the turtle. Later on, we will see the methods that are called to cause the value stored in **heading** to change. For now, suffice it to say that this is an angle in degrees that specifies the direction that the turtle is facing. Therefore, it must be converted to radians in Listing 14 to satisfy the angle requirements of the **rotate** method.

Rotate, translate, and draw ovals that represent the turtle

Having established the required rotation transform, Listing 15 draws six filled ovals for the shell, head, and legs of the turtle.

Listing 15. Rotate, translate, and draw ovals that represent the turtle.

```
      // 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);
```

The fillOval method

The calls to the **fillOval** method in Listing 15 assume that each oval is contained in a bounding rectangle whose sides are parallel to the horizontal and vertical axis. It is the affine transform that causes the ovals to be rotated, immediately prior to drawing, when rotation is required.

You will find the **fillOval** method in the **Graphics** class. I'm going to leave it up to you to use the documentation and analyze the code in Listing 15 to satisfy yourself that these six filled ovals will produce a *(possibly rotated)* turtle that looks like the turtle shown in Figure 1. *(As you may already have guessed, the calls to the setColor method establish the colors in which the different parts of the turtle will be drawn.)* The color, width, and height variables used in the arithmetic are declared in Listing 4.

Remainder of the paintComponent method

The remainder of the **paintComponent** method is shown in Listing 16.

Listing 16. Remainder of the paintComponent method.

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

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

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

The code in Listing 16 performs the following operations:

- Calls the **drawInfoString** method to draw a text label next to the turtle if the variable named **showInfo** is true.
- Restores the affine transform to the value that it had when the **paintComponent** method began execution.
- Calls the **paintComponent** method on a reference to a **Pen** object to draw the trail left by the turtle during its journey.

I will have more to say about the **paintComponent** method belonging to a **Pen** object in a future lesson.

It's time for a break

I don't know about you, but my brain is just about saturated at this point. I think it is time to call a halt to this lesson and to pick up with the remaining methods of the **SimpleTurtle** class in the next lesson. Fortunately, most of those methods will be somewhat easier to understand than the **PaintComponent** method.

Run the program

I encourage you to copy the code from 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

In this lesson, I explained the behavior of the **Turtle** class, and partially explained the behavior of its superclass named **SimpleTurtle**.

I explained the purpose of a number of the instance and class variables that are declared in the **SimpleTurtle** class. I also explained all five overloaded constructors in the **SimpleTurtle** class.

The most complex method of more than 50 methods in the class is the method named **paintComponent**.

The **paintComponent** method and the other two methods in the following list are the methods that are primarily responsible for displaying a turtle and its pen on either a **World** object or a **Picture** object. Thus, these three methods are critical to the **SimpleTurtle** class.

- updateDisplay()
- paintComponent(Graphics g)
- drop(Picture dropPicture)

I explained the **updateDisplay** method and the **paintComponent** method.

What's next?

I will begin the next lesson with an explanation of the drop method. Then I will explain another fundamental group of methods, which includes the methods that cause a turtle to face in a particular direction. That group includes the following methods:

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

After that, I will probably explain the methods that are used to cause a turtle to move around in a world or a picture. This includes the following methods:

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

It remains to be seen where I will go from there.

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
- 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

Complete program listings

Complete listings of the programs discussed in this lesson are shown in Listing 17 through Listing 19 below.

Listing 17. Source code for the program named StartupTest01.

```
```

```
/**
 *StartupTest01
 * The purpose of this program is to test such items as
 * the classpath, the media path, etc.
 *
 * 10/10/08 Compiles and runs OK on my laptop computer.
 *
 * Displays a turtle in a world and moves it forward by
 * 100 pixels.
 *
 * Note that the program does not terminate when you
 * click the X button in the frame.
 *
 * Based on a program by Barbara Ericson that is:
 * Copyright Georgia Institute of Technology 2004-2005
 */
public class Main{
  public static void main(String[] args){
    World mars = new World(200,300);
    Turtle joe = new Turtle(mars);
    joe.forward();
  }//end main

}//end class
```

Listing 18. The multimedia library class named Turtle.

```
/**
 * Class that represents a turtle which is similar to a
 * Logo turtle.
 * This class inherits from SimpleTurtle and is for
 * students to add methods to.
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class Turtle extends SimpleTurtle{
  ////////////////// constructors ///////////////////////

  /** Constructor that takes the x and y and a picture to
   * draw on
   * @param x the starting x position
   * @param y the starting y position
   * @param picture the picture to draw on
   */
  public Turtle (int x, int y, Picture picture){
    // let the parent constructor handle it
    super(x,y,picture);
  }

  /** Constructor that takes the x and y and a model
   * display to draw it on
```

```
   * @param x the starting x position
   * @param y the starting y position
   * @param modelDisplayer the thing that displays the
   * model
   */
  public Turtle(int x,int y,ModelDisplay modelDisplayer){
    // let the parent constructor handle it
    super(x,y,modelDisplayer);
  }

  /** Constructor that takes the model display
   * @param modelDisplay the thing that displays the model
   */
  public Turtle (ModelDisplay modelDisplay){
    // let the parent constructor handle it
    super(modelDisplay);
  }

  /**
   * Constructor that takes a picture to draw on
   * @param p the picture to draw on
   */
  public Turtle (Picture p){
    // let the parent constructor handle it
    super(p);
  }

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


}
//This is the end of class Turtle, put all new methods
// before this
```

Listing 19. The multimedia library class named SimpleTurtle.

```
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
```

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-