Introduction to the Java^{TM} Language

Jan H. Meinke

July 1, 2000

1 Introduction

Since its introduction in 1995 Java has become one of the most popular programming language. At first powered by the popularity of interactive web applets it soon became apparent that its implementation of object oriented programming, its inbuilt GUI (Graphical User Interface) and easy networking made it an ideal candidate for general purpose programming in a more and more networked world. The biggest disadvantage of Java as an interpreted language was the slow speed of execution and many techniques were developed to remedy this.

1.1 History

Java was originally developed as a hardware independent general purpose language - then called Oak - for embedded devices by the "Green Project" at Sun. Looking for a market for their new system they developed a demo for a cable TV set-top box for pay-per-view. Unfortunately the cable industry was not ready for this new technology so they had to think of something new.

The Internet was starting to become more and more the network the "Green Project" team had envisioned as a home network when they developed Java and they started working on a new better browser that allowed animation and interactivity through Java applets.

Sun made its first Java Development Kit (JDK) available in 1994 and soon afterwards Netscape included a Java Virtual Machine (VM) in its Netscape Navigator.

2 Getting Started with Java

This section introduces some of the basic tools and language facilities for writing simple Java programs. After working through this section you will be able to write an animated applet, compile and run it with the standard tools provided in a Java Development Kit.

2.1 Getting Java for Your Computer

As Java becomes more and more popular it will be preinstalled on many systems, however if you need to get the JDK, it can be downloaded from Sun's web page (http://java.sun.com). It also provides links for many implementations of the JDK by third party vendors since Sun currently provides its JDK only for Windows, Solaris and Linux .

For writing the source code all you need is a text editor like Notepad, vi, nedit, Bbedit, or whatever you like to use.

2.2 Our First Applet

In general an applet is a program that can be run within another program. They are found in many different environments. Java applets have become popular, because they run within a web browser.

Java's java.applet.Applet class is the base class for all Java applets. Being an Applet guarantees that certain methods - in other languages they might be called functions or procedures - are available to the environment. These include the init, start, stop and destroy method which are important during the life cycle of an applet.

When an applet has finished loading the init method will be called. This is an excellent place to put one time initializations that do not need to be redone everytime the user comes back to the web page. After the init method has finished start is called. Start is called everytime the user comes back to the page containing the applet[1].

The stop method gets called if the user leaves the page. You might want to stop an animation or sound that you are playing and free up resources or people might not want to come back to your page.

Finally destroy is called just before the applet is garbage collected. If you are keeping network connections open this is the place to make sure they get closed.

After this long preamble let's look at a short applet:

```
/* HelloWorld.java
 *
 *(c)2000 Jan H. Meinke
 *
 */

import java.awt.*;
import java.applet.*;

/** A simple Applet that writes the words "Hello World!" on the screen.
 */
public class HelloWorld extends Applet{
    int xpos = 10;
    int ypos = 10;

    public void paint(Graphics g){
        g.drawString("Hello, World!", xpos, ypos);
    }
}
```

The first thing you might notice is that after talking about init, start, stop and destroy for several paragraphs, none of them appear instead there is a public void paint(Graphics g) method.

[1] This is not quite true. Some browsers do not run start if you use the back arrow button on the menu bar. The user might have to reload the page before the start method is called.

java.applet.Applet contains defaults for all its methods that often work just fine. In this case all the applet is doing is drawing something on the screen. It can do that because an Applet is a Panel which knows how to draw itself. That is also where the paint method comes from. Everytime our Applet needs to redraw itself the paint method is called and that is where the drawing has to take place.

To run the applet a very simple HTML-file is necessary:

```
<html>
  <body>
    <applet code=”HelloWorld.class” width=”240” height=”240”>
    </applet>
  </body>
</html>
```

Make sure that you store the source code for the HelloWorld Applet in a file called HelloWorld.java. Java is case sensitive, so upper and lower case letters are different. The name of the file is always equal to the name of the public class contained in that file plus the ending .java.

Now run the Java compiler: javac HelloWorld.java.

Create the HTML-file and save it under hello.html. Now you can run the applet using the appletviewer: appletviewer hello.html.

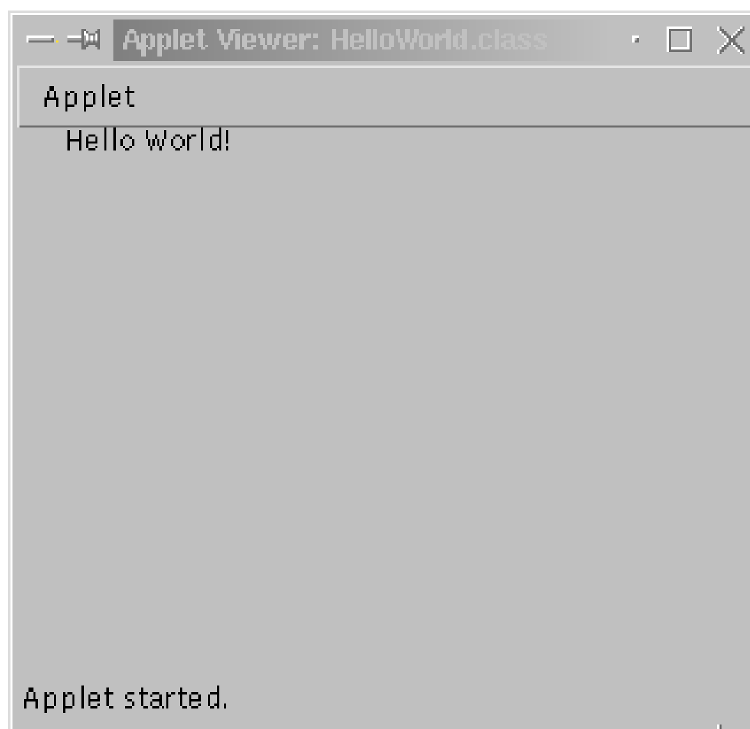

Figure 1: HelloWorld Applet in the appletviewer.

2.3 Our first animation

Why write an applet for something that could have been done with one line of code in HTML? In the following section we will animate our HelloWorld applet.

What is an animation? This question is not quite as trivial as it may seem. On TV or at the movies the illusion of motion is achieved by playing still pictures at a rate so high that the eye is incapable of resolving them. The brain therefore interprets them as motion if the difference between frames is reasonably small.

On a computer the principle is the same. Often however the goal is not just to replay a set of prefabricated frames, but create the next frame on the fly. This allows the user to interact with the animation.

There are therefore two distinct processes at work here:

1. Create/calculate the next frame.

2. Display the new frame on the screen.

To allow for simultaneous execution of more then one task Java provides threads. Creating threads is easy although programming with them, often referred to as "concurrent programming" can be tricky. Here we will only implement a straightforward animation thread and not worry about further subtleties for now.

There are two ways to create a thread. One can either subclass the Thread class directly and overwrite the run method or implement the Runnable interface and provide the run method.

Without going into more detail let us look at the changes that are necessary to animate the HelloWorld applet from section 2.2:

```
/* HelloWorldAnimation.java
 *
 *(c)2000 Jan H. Meinke
 *
 */

import java.awt.*;
import java.applet.*;

/** A simple Applet that scrolls the words "Hello World!" across the screen.
 */
public class HelloWorldAnimation extends Applet implements Runnable{
    /** Horizontal position of the Hello World! text. */
    int xpos = 10;

    /** Vertical position of the Hello World! text */
    int ypos = 10;

    /**Time in micro seconds between frames */
    int pause = 100;

    /** Flag to signal thread if it should continue. */
    private boolean animate = false;
```

```
    private Thread aniThread;

    public void start(){
        /* Create and start the animation thread */
        aniThread = new Thread(this);
        animate = true;
        aniThread.start();
    }

    public void stop(){
        animate = false;
    }

    public void run(){
        while(animate) {
            /* Do whatever is necessary to calculate the next frame in the animation */
            xpos++;

            /*Request repainting of the applet. */
            repaint();

            /* Put thread to sleep to give applet a chance to repaint itself. */
            try{
                Thread.sleep(pause);
            }catch(InterruptedException ie){}
        }
    }

    public void paint(Graphics g){
        g.drawString("Hello, World!", xpos, ypos);
    }
}
```

Here three more methods appear: start, stop and run. start and stop are part of the Applet methods as described earlier. The run method is part of the Runnable interface and needs to be implemented. Let us go through the applet part by part:

First comes the class definition followed by some more variables. In addition to extending the Applet class which provides default implementations for each method the Runnable interface is implemented. An interface declares some methods without giving any implementation. The runnable interface, for example, declares a method `public void run` but it does not provide any code. When you implement the Runnable interface you make a promise to provide this method and the compiler will make sure that you keep it.

The start method is used to create a Thread object, set the animate flag to true and start the thread. The stop method just sets the animate flag to false which will cause the thread to finish.

The run method is the heart of the animation. It will continue to execute as long as animate is set to

true. It first calculates the new frame - here the x position is increased by one pixel - and then requests a repaint of the applet. Notice that repainting cannot be forced. It will only take place if the JVM thinks it has time to do the paint job. After requesting a repaint the thread is put to sleep. This allows other threads, namely the paint thread, to run. Without implementing this pause the x position will be changed but it will not show on the screen. The try - catch structure will be discussed later.

If you have trouble compiling the program make sure that you named the source file HelloWorldAnimation.java. The HTML file looks just like the one used for the HelloWorld applet, except for the classname in the code attribute.

3 Copernicus' Solar System

For thousands of years people have watched the heavens. The movement of the stars through the seasons and the night and those strange wanderers that changed their place among the stars have fascinated and inspired much research and investigation. Many a model was suggested and intricate patterns of motion for the planets were deviced to keep the earth in the center of the universe.

Copernicus suggested a simpler model that could at least qualitatively explain the motion of the planets in the sky. He proposed that the Sun should be the center of motion and that the Earth with the other 5 known planets (Mercury, Venus, Mars, Jupiter and Saturn) followed a circular orbit around it.

In this section we will create an applet that shows a top view of the solar system as imagined by Copernicus.

3.1 Objects in the sky

The solar system consists of the Sun, the planets, moons and some smaller objects that we will ignore for the purpose of this model.

Exercise: Make a list of properties to describe

1. Sun
2. Planets
3. Moons

There are many possible answers to the previous exercise. Table presents one possibility.

All three have several things in common. They all have a mass and a radius and a color. Both planets and moons have an orbital radius and a period and a body in the center of this orbit. Object oriented programming tries to make use of these commonalities. First define a class CelestialBody that includes the features that all have in common.

CelestialBody:

- Member variables
  - Mass m
  - Radius r
  - Color c

- Member method
    - paint

You might wonder why the paint method is included. This way all CelestialBody objects can be drawn on the screen which after all is our final goal.

Next define a class Orbiter. An Orbiter is a CelestialBody that orbits around another CelestialBody. To define a circular orbit we need the center, the radius and period of the orbit and optionally an initial phase. The phase determines where the Orbiter is in its orbit at t=0. In summary:

Orbiter extends CelestialBody:

- Member variables
    - Center of orbit
    - Orbital period Period
    - Orbital radius Radius
    - Phase phi
- Member method
    - update

These two classes will take on the main workload. The system will be displayed and animated within an applet very similar to the one introduced earlier. Here is the implementation of the CelestialBody:

```
/* CelestialBody.java
 *
 * (c) 2000 Jan H. Meinke
 *
 */
import java.awt.*;

/** Represents a general body in space with certain properties like mass and
 *   color.
 */
public abstract class CelestialBody{

    double mass;
    double radius;

    /** The main color of a planet. This is not always very well defined, but
    *     since this is supposed to be a simple model it will suffice.
    */
    Color color;
```

```
    /** Since there is no such thing as a default mass, radius or color for
    *   a celestial body all three need to be given in the constructor
    */
    public CelestialBody(double mass, double radius, Color color{
        this.mass = mass;
        this.radius = radius;
        this.color = color;
    }
    /** The paint method is different for different sub classes of a
    *   CelestialBody. Here it is declared abstract to force a definition.
    */
    public abstract void paint(Graphics g);
}
```

Again the file started with the class declaration but there is an additional modifier, **abstract**. An abstract class cannot be used to create an object. This allows us to define common properties without having to implement some dummy procedures.

Imagine you wanted to write a program that simulates life on earth. All life needs to eat and you might want a method called eat, but it doesn't really make sense to describe exactly how this process of eating works since it will be very different for bacteria than humans.

Here the same is true for the paint method. We only know that every CelestialBody is capable of painting itself, but not how it is done. If you look further down to the declaration of the paint method notice that it too is declared abstract. This by itself would make CelestialBody an abstract class even without explicitely stating it at the beginning but it is good practice to do so anyway.

An Orbiter *is a* CelestialBody. This relation is made explicit by the **extends** statement in the class declaration

```
public class Orbiter extends CelestialBody{
```

All methods and variables except those declared priabte are available in any sub class, another name for the "is a" relationship.

In addition to mass, radius and color and Orbiter needs to store the properties of its orbit, its radius, period and phase.

The units au and year used for the radius and the period of the orbit respectively should be the same for all Orbiters, in other words there should only be one variable au for all instances of the class Orbiter. This is accomplished with the **static** modifier.

```
public static double au = 10.0;
public static double year = 15.0;
```

The constructor for an Orbiter is then

```
public Orbiter(double m, double r, Color c, double oR, double oP, double phase){

    /* Need to call the constructor of the super class explicitely since arguments disagree. */
```

```
    super(m, r, c);

    this.oR = oR;
    this.oP = oP;
    this.phase = phase;
}
```

Most of the work is done in the two upcoming methods *updatePosition* and *paint*. The *updatePosition* method brings the state of the orbiter up to the given time. For now this just means calculating the position of the orbiter:

$$x = R * \sin(\frac{t}{P} + \phi) + x_0$$

and

$$y = R * \cos(\frac{t}{P} + \phi) + y_0.$$

These are the x and y coordinate for a circular orbit around a center (x_0, y_0).

```
/** Calculates the position of the Orbiter at time t.
 *
 * @param t current time
 */
public void udatePosition(int t){

    xpos = oR * sin(t / oP + phase) + x0;
    ypos = oR * cos(t / oP + phase) + y0;
}
```

Finally the *paint* method has to draw the Orbiter. For simplicity we will just have it draw a disk with appropriate radius around x and y.

This is a little trickier than it sounds. The *fillOval* method, there is no *fillCircle* method available, takes an encompassing rectangle as argument (see Fig. 2). To get the disk properly centered takes a little effort. For convenience let us implement a *fillCircle* method.

A circular disk can be described by its center and radius. Internally we want to use the fillOval method. Therefore the coordinates need to transformed. This is not too hard.

$$\begin{aligned} x' &= x - r \\ y' &= y - r \end{aligned}$$

Our *fillCircle* method could then be written in the following way:

```
/** Draw a circular disk in the current foreground color
 *  around a point (x,y) with radius r.
 *
 *  @param g the graphics object used for drawing
 *  @param x x-coordinate of the center
```

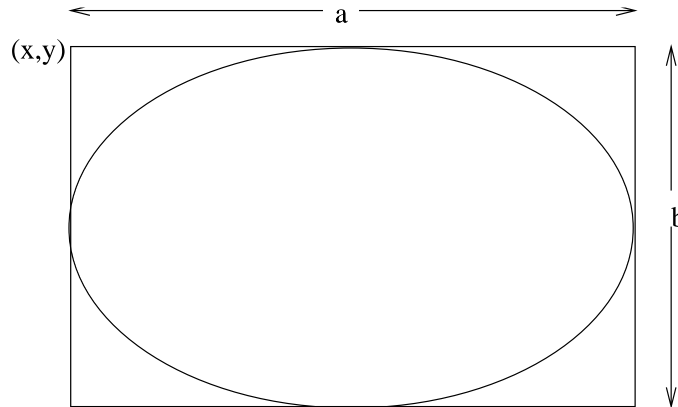

Figure 2: filledOval's parameters

```
 *   @param y y-coordinate of the center
 *   @param r radius of the disk in pixels
 */
public final void fillCircle(Graphics g, int x, int y, int r){
    g.fillOval(x - r, y - r, 2 * r, 2 * r);
}
```

The final modifier indicates that this method cannot be overwritten by a subclass. The advantage of this is that the compiler can inline the method, this means it can put the actual code for executing the function rather than a function call, which takes addtional time, in the byte code.

Now we are ready for the *paint* method.

```
public void paint(Graphics g){
     g.setForeground(color);
     fillCircle(g, xpos, ypos, radius);
}
```

One more closing } and the Orbiter class is done.

Now let us wrap things up. There are still four more classes left to build but thanks to the work we have already done they will be rather short ones. We still need to implement *Sun*, *Planet*, *Moon* and *SolarSystemApplet*. The Sun is just a CelestialBody and all we need to do is implement the paint method for it.

Planet is an *Orbiter*. However a planet always belongs to a solar system and the center of its orbit is the center of the solar system. Therefore the update method will be overwritten. Furthermore it can have moons. We need a way to add moons to the planet and we have to make sure they get updated and drawn when appropriate.

Moon is an *Orbiter* with a *Planet* in the center. The update method has to be modified in almost the same way as for *Planet*, except that this time a *Planet* is sitting in the center.

Finally the *SolarSystemApplet* provides our drawing surface. Within its init method the planets and moons are added and it contains the animation thread as in 2.3.

```
/* Sun.java
```

```
 *
 * (c)2000 Jan H. Meinke
 *
 */

/** The central star of a solar system.
   * The update and paint methods go recursively through the whole
   * solar system.
   */
public classSunextends CelestialBody{


}
```