

A Jump Start to Java 2 Applets for C++ Programmers

Frank Ducrest
fdd@louisiana.edu

These examples were created with Sun Microsystems Java 2 developers kit JDK1.3 for Windows. They have also been tested in a Solaris UNIX environment. At this writing, Java 2 developers JDK1.4 for most platforms is available on the Sun site at <http://java.sun.com/j2se/1.4/index.html>. The developers kit contains the javac compiler, and the interpreters java and appletviewer. Sun also distributes a freely available Java editor called Forte, which is also available for most platforms at <http://www.sun.com/software/Developer-products/ffj/buy.html>. If you wish to use the applets created in this tutorial in a browser, you will need Microsoft Internet Explorer 5.01 or greater or Netscape Communicator 6.01 or greater.

1. Java

Java is a language that was created by fair sized group of people at Sun Microsystems in the early to mid 1990s. Originally called Oak, it was renamed Java when it was released in 1995. Java 2, the release used in this introduction, contains many improvements over the original release. These examples may have to be modified to run in original release Java or any of the existing versions of Microsoft's J++ compiler.

Whereas C++ was an object-oriented extension of the C programming language, Java is natively object-oriented, with strong support for inheritance and polymorphism. Java is so completely object oriented that even minimal programs must be created of a class that transparently inherits minimal functions from classes belonging to the Java system. Since programs are classes, objects do not exist in Java until they are created at run time. In fact, the class of an object may not be known until run time. This is a departure from C++ where programmers are used to specifying every object instance of a class or classes, then having them created at compile time.

2. Why Java?

Java compiles into *byte code*, which is a binary code that is not specific to any of the usual processors. To be run, a Java interpreter must interpret Java byte code. Since Java byte code is binary, it is much more efficient than interpreted high-level language code. Since Java byte code is not specific to any processor, it may be run on any computer that contains a Java interpreter.

Java programs execute in an encapsulated space. That is, the Java run-time environment protects the system running the program against accidental or malicious damage from poorly written code or viruses. In addition, the run-time environment additionally has the virtue of performing all of the necessary garbage collection when a running Java program no longer needs portions of memory. Encapsulation of a running Java program is supported further by the lack of pointers that a programmer can directly

manipulate, although all programmer created instances of class based objects are dynamic (they are initialized with *new*).

3. What this Tutorial Covers

This example Java project will go through the steps of creating a series of classes that will come together to create an applet that can be run in a browser or the free standing applet interpreter, *appletviewer*. It will include basic Java classes and methods, object initialization and use (including simple types, class objects and arrays of class objects), inheritance (including inheritance of user created classes, system classes and other classes) and implementation of various interfaces. It is not necessary for you to know what these things are in Java, as they will be explained in the examples. All that is assumed is that you have some background in an object-oriented programming language such as C++ and, since Java was created around the syntax of C++, a working knowledge of C++ syntax.

There are things too numerous to mention not included in this example project. Except for a few tutorials-examples included at the end of this example, no attempt has been made to comprehensively cover all aspects of Java. The sole purpose of this tutorial is to make C++ programmers comfortable enough with the basics of Java in short order, then to be able to move on to a comprehensive Java reference book.

4. The Example Project

This example project will take the reader through the steps necessary to create an interactive Java game where a bug of heroic bent (controlled by the user) will have to avoid crashing into bug zappers and randomly moving bugs. In addition, other bugs will hunt the user's bug. These "evil" bugs have the sole purpose of catching the user's bug. (The user's bug is apparently very tasty.) The user's goal is to have his bug survive.

The fundamental components needed for the game are the background, bug zappers, a controllable bug, randomly moving bugs and bugs capable of targeting and pursuing the user's bug.

5. Basic Facts about Java Classes

Classes in Java are more fundamental to Java programs than to C++ programs. In fact, it is impossible to write a Java program without at least one class. In C++, classes are factory like structures that are used to create objects within a program. In Java, a program is a class that is instantiated at run time.

In most class files, the first line in the file is an *import* statement. The *import* statement allows existing code to be introduced into a Java file. This is very similar in function to the *#include* preprocessor instruction in C++. However, *import* is a Java keyword and the *import* statement will be translated directly by the Java compiler. The general form of the *import* statement is:

```
import packages-and-classes;
```

where *packages-and-classes* refers to the code to be imported. (*Note: A Java package can act as a library of classes, a subset of a Java program or a complete Java program.*)

There isn't much difference in Java between these things.) In general, the *import* statement ends up looking something like this:

```
import package1. [package2.[package3.]] classname;
```

where an asterisk (*) may be substituted for *classname*. This has the effect of importing all classes of a package into the file.

After importing, the class is described. The typical general form of a Java class is:

```
access class class-name {
    access type instance-variable-1;
    access type instance-variable-2;
    :
    access type instance-variable-n;

    access type method-name-1(parameter-list) {
    }

    access type method-name-2(parameter-list) {
    }
    :
    :
    access type method-name-n(parameter-list) {
    }
}
```

where *access* is *public*, *default* (none), *private*, or *protected*
class-name is the name of the class
type refers to one of the basic Java data types or a class
{ } is a code block (compound statement)

Notice that Java classes do not end with a semicolon (;).

Public access means that the item described is accessible from outside of the class. The Java default access is similar to public, but it is recommended that public access always be specified when public access is intended. Private access means that the item described may be access only inside of an enclosing class. (Note: Classes may be created inside of other classes in Java 2. They are essentially local in scope.) Protected access means that only an inheriting class (to be dealt with later) may access the described item

Java names, called identifiers, consist of letters, numbers, the underscore or dollar sign. They must not begin with a number or include a space. Java is case sensitive.

Basic Java types are byte (8 bit integer), short (16 bit integer), int (32 bit integer), long (64 bit integer), float (32 bit floating point), double (64 bit floating point), char (16 bit character), boolean (true or false – note that a boolean is never 1 or 0).

Type char does not represent a special class of integer from 0-255 as in C++. Instead, it represents a character set called Unicode, which represents a fully international character set. The range of char is 0 to 65,536. The good news for C++ programmers is that the range from 0 to 255 corresponds to the familiar ASCII collating sequence.

Literals, including strings are supported in the familiar fashion. For example:

an integer number :	45
a floating point number:	3.245
a character:	'a'
a string:	"Hello, world!"

A literal integer is always a long. A literal floating point is always a double.

Instance variables are declared in the usual C++ fashion. For example:

```
private int j;
```

The access may be specified or default.

Assignment is via the familiar "=" sign, with C++ syntax of the assignment statement being preserved:

```
variable = expression;
```

Expressions may include the usual mix of variables and literals. Problems may occur when mixing literal numbers and variables in an expression, as an expression will always be converted to the largest type present. Integers will always be converted to floating point numbers if both types are present in the same expression. For this reason, C++ style type casting is supported:

```
double h;  
int z;  
h = 35.4;  
z = (int)(h * 4);
```

Instance variables may be assigned a value when they are created. Since Java programs are not instantiated until run time, variable values may be declared dynamically with values that are not known until the program executes. In the following example, variable *a* receives its value as a method parameter.

```
double f = a * 3.14;
```

Functions found in classes are referred to as class methods. These may be constructors (or not) and maintain C++ syntax, with the exception that they are almost always developed immediately in the class definition rather than being defined in a separate file or later in the same file as in C++. Constructors must have the same name and access as the class. Since Java supports polymorphism, methods in a class may be

overloaded, i.e. have the same name so long as the number or type of parameters in the parameter list is unique to each method.

The type of a method is its return type. This may be void (for no return), nothing (for constructors) or any type or class within the scope of the class. The return statement in Java is very similar to the C++ return statement and is syntactically the same.

One last note before launching into the example, comments are the same as in C++. // is used to indicate that the portion of the line to the right is a comment. /* */ is used to enclose what is often a multi-line comment.

6. Class Background

The first class to be intruded in the example is the class *Background*. Java is happiest when a class is placed in a file of the same name, ending in “.java”. Class *Background* is created in a file called *Background.java*. (Note: In the Windows environment, capitalization in file names is not usually required or even particularly noticed. In other environments, file name capitalization must match class name capitalization exactly.)

The file containing class *Background* begins with an import statement. This is:

```
import java.awt.*;
```

This imports all of the classes that are part of the package *awt*, which is a package that is part of package *java*. (Almost all Java programs that use the windowing features include package *awt*, which stands for *Abstract Window Toolkit*.) The import statement, like all Java statements, ends with a semicolon (;).

Next comes the declaration of class *Background*.

```
public class Background {  
}
```

Background is declared as access type *public* so that it can be used by other classes. In fact, class *Background* is being created solely to be used by other classes and will not be capable of running as a program. Class *Background* is being created only to create a background for another Java class to operate over in an applet.

Within class *Background*, two instance variables of type *int* need to be created. These will be used to hold the size of the background to be produced on demand. These are:

```
private int WIDTH;  
private int HEIGHT;
```

Both *WIDTH* and *HEIGHT* will not be directly accessible outside of class *Background* because they are *private*. Since they are *private*, a method inside of *Background* will have to assign them a value. This falls to the constructor method:

```
public Background(int w, int h) {  
    WIDTH = w;  
    HEIGHT = h;  
}
```

When an instance variable of class `Background` is created in some other class, this constructor will have to be called.

The second and final method of class `Background` is *paint*, which receives a parameter of class `Graphics` (part of the package *awt* and vital to producing an applet). When method *paint* is called by other classes, each call will have to furnish an object of class `Graphics` as a parameter. *setColor* and *fillRect* are methods of class `Graphics`.

```
public void paint(Graphics g) {
    g.setColor(Color.black);
    g.fillRect(0, 0, WIDTH, HEIGHT);
}
```

Here, *setColor* receives an argument of class `Color`. In this case it is a named constant defined in class `Color` called *black*. Since we have not imported `Color`, it is necessary to refer to the constant *black* via the name of its class, separated by a period. (Note: Other color constants created in class `Color` are *blue*, *cyan*, *darkGray*, *gray*, *green*, *lightGray*, *magenta*, *orange*, *pink*, *red*, *white* and *yellow*. Creating colors will be dealt with later.)

The `Graphics` method *fillRect* has four parameters. The first two are used to set the starting X and Y values of the rectangle. X is the starting pixel across. Y is the starting pixel down. Point 0,0 is in the upper left corner. The third and fourth parameters give the width (to the right) from the starting X and height (down) from the starting Y of the rectangle.

Here is the full listing of class `Background`:

```
import java.awt.*;

public class Background {
    private int WIDTH;
    private int HEIGHT;

    public Background(int w, int h) {
        WIDTH = w;
        HEIGHT = h;
    }

    public void paint(Graphics g) {
        g.setColor(Color.black);
        g.fillRect(0, 0, WIDTH, HEIGHT);
    }
}
```

7. Test Class Driver

Since class *Background* cannot be used as a standalone program, a simple driver class will be constructed to test it. This is a common and highly recommended technique that will also allow several other Java techniques to be introduced in a simple setting.

First, start a new file and name it *driver.java*. In it import all AWT classes and Applet class. This is as follows:

```
import java.awt.*;
import java.applet.Applet;
```

Since class *driver* will constitute a Java applet, it must inherit class *Applet*. This means that all methods and instance variables of class *Applet* will become part of class *driver*. Inheritance is a large portion of Java's support for polymorphism. In Java, classes inherit by using *extends* after the class name. This has the form:

```
access class class-name extends inherited-class {
```

To inherit class *Applet*, the declaration of class *driver* is as follows:

```
public class driver extends Applet {
```

Class *Applet* comes with a number of methods, one of which is the method *paint*. (*This is not to be confused with the method *paint* in class *Background*.*) When a program created from a class that has inherited class *Applet* is run, an instance variable of class *Graphics* will be created and the method *paint* will be called with the *Graphics* instance variable as the sole parameter. Java programmers who wish to use this automatically created variable and the call to *paint* are expected to provide an overriding version of *paint*. Class *driver* will provide an overriding version of method *paint* to take advantage of this build-in logic.

```
    public void paint(Graphics g) {
        :
        :
    }
```

Since the whole point of class *driver* is to test the class *Background*, an instance variable of class *Background* is created in it, the *Background* constructor is called and the newly created variable is used to call the *Background* method *paint*. This leads to this full version of class *driver*:

```
import java.awt.*;
import java.applet.Applet;

public class driver extends Applet {
    private Background bg = new Background(500,400);

    public void paint(Graphics g) {
        bg.paint(g);
    }
}
```

(Note: In Java, constructors are always called with *new*. This is because all variables not of fundamental data types are dynamic. Since Java instance variables are not created until run time, this gives maximum flexibility in how these variables can be created and used, even to the point of dynamically allocating the type or class. Additionally, Java garbage collection automatically returns previously allocated memory to the system when a variable passes out of scope or is otherwise no longer used in the running program.)

All that need be added to *driver* before we compile and run the test program is an HTML applet tag in comments after the import statements and before the class. In practice, this often does not work in the Windows environment. Instead, this example will use an independent HTML program named *driver.html* to call the driver applet once the Java files have been compiled. This is as follows:

```
<HTML>
<HEAD>
<TITLE>A Test Driver</TITLE>
</HEAD>
<BODY>
<APPLET CODE="driver.class" WIDTH=500 HEIGHT=400>
</APPLET>
</BODY>
</HTML>
```

8. Compiling and Running the Driver Applet

Each Java file must be compiled into a *byte code* file with the Java compiler. This is done at the DOS prompt as follows:

```
javac file-name.java
```

Each byte code file that is created will be named

```
class-name.class
```

Once `Background.java` and `driver.java` have been compiled into `.class` byte code files, the resulting applet `driver.class` is ready to be invoked from the HTML file by loading into a browser (Microsoft Internet Explorer 5.01 or higher, Netscape Communicator 6.01 or higher) or may be viewed via the appletviewer utility. To run the driver applet via the appletviewer, type this command at the prompt:

```
appletviewer driver.html
```

All this typing can become very tedious, especially with long and expressive file class names. To overcome this, the lost art of the DOS batch file (a text file of commands with a name ending in `.bat`) can be resurrected on Windows systems. Once a batch file is created, all that need be typed to compile and run the applet is the name of the batch file. For example, the DOS batch file to compile and run the driver applet would look like the following:

```
del *.class
javac Background.java
pause
javac driver.java
pause
appletviewer driver.html
```

By deleting all class files (`del *.class`) before compilation, the programmer is assured that only the latest compilations will be used. By including the pause command after every call to the Java compiler, the process can be halted to fix any problems by typing CTRL-C at the pause. Having named this batch file `go.bat`, running it produces the following:

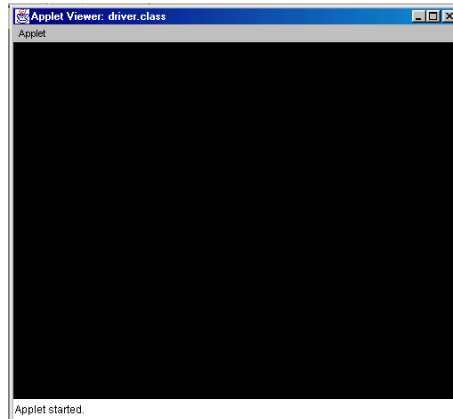
```
C:\jdk1.3\projects\gameexample>go
C:\jdk1.3\projects\gameexample>del *.class
C:\jdk1.3\projects\gameexample>javac Background.java
C:\jdk1.3\projects\gameexample>pause
Press any key to continue . . .

C:\jdk1.3\projects\gameexample>javac driver.java
C:\jdk1.3\projects\gameexample>pause
Press any key to continue . . .

C:\jdk1.3\projects\gameexample>appletviewer driver.html
```

UNIX users can create a script file to serve the same purpose as the above batch file by creating a file of commands in an editor, giving the file a name, then using the `chmod` command to make the script file executable.

Here is a view of the running applet:



9. Class *protoBugZapper*

The next class to be tackled is the generic class *protoBugZapper*. This class will allow us to devise and test our basic bug zapper class without having to worry about things that do not yet exist, such as the bugs that will some day seek out our bug zappers and collide with them. In the future, *protoBugZapper* will be inherited by an ever so slightly more sophisticated class that will fit more precisely into the game project. (Sorry, this tutorial suffers from the usual desire to show a technique and strives to do so by jamming the technique in even if it is not strictly necessary. It does, however, strive to keep this to a minimum.)

Class *protoBugZapper* will be housed inside of a file called *protoBugZapper.java*. It will begin with the usual include of all AWT classes and will include three protected variables. Instance variables *x* and *y* will give the initial location of the zapper while *scale* will give its size.

```
import java.awt.*;

public class protoBugZapper{
    protected int x;
    protected int y;
    protected int scale;
    :
    :
}
```

protoBugZapper is meant to be the foundation class (called a super class) of the actual bug class that will eventually be used in the game and it will have a constructor. As you would expect, the constructor will receive as parameters the initial values for *x*, *y* and *scale*.

```

protoBugZapper(int initX, int initY, int initScale) {
    x = initX;
    y = initY;
    scale = initScale;
}

```

Lastly, a *paint* method will be created to draw the actual bug zappers. Since the necessary drawing function that will be used is part of class *Graphics*, to work, *paint* will receive a variable of class *Graphics* as its single parameter.

Three functions will be used. Two are of class *Graphics* and one of class *Color*. First, the constructor of class *Color* will be used to create two new colors stored in two instance variables of class *Color*. These calls are:

```

Color runnyGray = new Color(128,96,64);
Color runnyBrown = new Color(128,84,64);

```

Notice that the constructor of class *Color* requires three parameters. The first parameter represents the intensity of red, the second green and the third blue (RGB). Values allowed range from 0 (off) to 255 (brightest).

The *Graphics* method *setColor* is called twice to set the color to be used to draw with. Once the color is set to *runnyGray*, the next time it is set to *runnyBrown*.

```

    :
    g.setColor(runnyGray);
    :
    :
    g.setColor(runnyBrown);
    :

```

To finish drawing our bug zapper, the *Graphics* method *fillArc* is called liberally after each *setColor* call. *fillArc* requires 6 parameters. These are, from left to right:

- *x, y* – a pair that gives the upper left point of the rectangle that will enclose the arc
- *width, height* – a pair that will give the width (right) from *x* and the height (down) from *y* of the rectangle that will enclose the arc
- *start angle* – gives the starting angle of the arc in degrees; 0 degrees is located on the horizontal at 3 o'clock
- *sweep angle* – angular distance from the start angle in degrees; negative draws clockwise, positive draws counter-clockwise

```

public void paint(Graphics g)
    {
        Color runnyGray = new Color(128,96,64);
        g.setColor(runnyGray);
        g.fillArc(x,y,scale/2,scale/2,0,45);
        g.fillArc(x,y,scale/2,scale/2,90,45);
        g.fillArc(x,y,scale/2,scale/2,180,45);
        g.fillArc(x,y,scale/2,scale/2,270,45);
        Color runnyBrown = new Color(128,84,64);
        g.setColor(runnyBrown);
        g.fillArc(x,y,scale/2,scale/2,45,45);
        g.fillArc(x,y,scale/2,scale/2,135,45);
        g.fillArc(x,y,scale/2,scale/2,225,45);
        g.fillArc(x,y,scale/2,scale/2,315,45);
    }

```

Here is the complete listing of class *protoBugZapper*:

```

import java.awt.*;

public class protoBugZapper {
    protected int x;
    protected int y;
    protected int scale;

    protoBugZapper(int initX, int initY, int initScale) {
        x = initX;
        y = initY;
        scale = initScale;
    }

    public void paint(Graphics g) {
        Color runnyGray = new Color(128,96,64);
        g.setColor(runnyGray);
        g.fillArc(x,y,scale/2,scale/2,0,45);
        g.fillArc(x,y,scale/2,scale/2,90,45);
        g.fillArc(x,y,scale/2,scale/2,180,45);
        g.fillArc(x,y,scale/2,scale/2,270,45);
        Color runnyBrown = new Color(128,84,64);
        g.setColor(runnyBrown);
        g.fillArc(x,y,scale/2,scale/2,45,45);
        g.fillArc(x,y,scale/2,scale/2,135,45);
        g.fillArc(x,y,scale/2,scale/2,225,45);
        g.fillArc(x,y,scale/2,scale/2,315,45);
    }
}

```

Class driver can easily be rewritten to test class *protoBugZapper*, as can the batch file *go.bat*. Here is the revised *driver.java*. Notice that even though no constructors for *protoBugZapper* have been explicitly declared, the compiler has created a default constructor and it must be called. Notice also that the Background object has been left in, even though that is not strictly necessary.

```
import java.awt.*;
import java.applet.Applet;

public class driver extends Applet {
    private Background bg = new Background(500,400);
    private protoBugZapper one = new protoBugZapper(50,100,35);
    private protoBugZapper two = new protoBugZapper(300,250,100);

    public void paint(Graphics g) {
        bg.paint(g);
        one.paint(g);
        two.paint(g);
    }
}
```

Here is the revised *go.bat*. (Notice that since *Background.java* has previously been compiled and tested, notice that *Background.class* has not been deleted and so *Background.java* does not need to be compiled again, saving time.)

```
del driver.class
del protoBugZapper.class
javac protoBugZapper.java
pause
javac driver.java
pause
appletviewer driver.html
```

Running go.bat produces the following:

```
C:\jdk1.3\projects\bugs>go
C:\jdk1.3\projects\bugs>del driver.class
File not found
C:\jdk1.3\projects\bugs>del protoBugZapper.class
C:\jdk1.3\projects\bugs>javac protoBugZapper.java
C:\jdk1.3\projects\bugs>pause
Press any key to continue . . .

C:\jdk1.3\projects\bugs>javac driver.java
C:\jdk1.3\projects\bugs>pause
Press any key to continue . . .

C:\jdk1.3\projects\bugs>appletviewer driver.html
```

The running applet (shown here with colors reversed) looks like this:

