A Jump Start to Java 2 using Swing for C++ Programmers

Frank Ducrest

These examples were created with Sun Microsystems Java 2 developers kit JDK1.4.1_02. At this writing, JDK1.4.1_02 is the latest stable release of the developers kit and is available as a free download for most operating systems on the Sun Microsystems site at **http://java.sun.com/**.

1. Preface

This tutorial introduces the ideas used in creating an interactive Java program that uses Swing to create an interactive game that can be run as an application program. The resulting game is very close to the resulting game in A Jump Start to Java 2 Applets for C++ Programmers. However, the code has been extensively revised and classes have been deleted and added to accommodate the development sequence of the game using Swing.

You may wish to review A Jump Start to Java 2 Applets for C++ Programmers after completing this one for a quick introduction to Applets.

If find this tutorial of value, I would appreciate being told about typos and mistakes so that they can be corrected.

Frank Ducrest
fdd@louisiana.edu

2. 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 compile in original release Java or any other available Java compilers.

Whereas C++ was an object-oriented extension of the C programming language, Java is natively object-oriented, with strong support for abstraction, inheritance, polymorphism and encapsulation. 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.

3. 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, called the *Java run time environment* or *Java virtual machine*, 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 releases 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 (i.e. they are initialized with *new*).

4. What this Tutorial Covers

This example project will go through the steps of creating a series of classes that will come together to create a Java program that can be run in the Java run time environment. Once compiled into byte code, the program will be able to be run on any operating system on any machine that has the Java run time environment. 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 an interface. 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 procedural programming language such as C++.

There are a number of things not included in this example project. Some of these are creation and running applets and serverlets, user input other than with buttons or touch-and-go, file I/O, loading and displaying image files, loading and using sound files, and so on *ad infinitum*. Some of these are covered in a series of short examples attached to the end of this example project as an appendix, but a progression to a full text is recommend after completing this tutorial.

5. 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 - carnivorous - bugs, whose sole purpose is to catch and eat the user’s bug, will prowl the game. The user’s goal is to have his bug survive.

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

6. Basic Facts about Java Classes

Classes in 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, the same applies with the addition that 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 sub class 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.]]class-name*;

where an asterisk (*) may be substitued 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 [extends class-name]
                                        [implements interface[,interface][,...] {
        access type global-instance-variable-1;
        access type global- instance-variable-2;
                :
        access type global-instance-variable-n;

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

        access type method-name-2(parameter-list) {
        }
                ... etc. ..
}
```

where *access* is *public*, *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)

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

Setting *public* access means that the item described is accessible from outside of the class. Setting *private* access means that the item described may be accessed only inside of an enclosing class. (Note: Classes may be created inside of other classes in Java 2. They are essentially local in scope.) Setting *protected* access means much the same as *private*, but *private* members cannot be inherited while *protected* members can be inherited.

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* value 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 with lots of space let for additions. 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 ACSII collating sequence.

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

an integer number :	45
floating point numbers:	-3.245, 0.2E-11
a char:	‘a’
a String:	“Hello, world!”
boolean:	true

A literal integer is always a long. A literal floating point is always a double. (Note: *String* is a class, although *String* literals are supported.)

Instance variables of the basic data types are declared in the usual C++ fashion, but the access is generally also specified. For example:

```
private int j;
```

Assignment is via the familiar “=” sign, with C++ syntax of the assignment statement 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. 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, as is typical 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 name and parameter list of a *method* is referred to as its *signature*. Overloaded methods must have different *signatures* despite having the same name.

The *type* of a method is its *return type*. This may be *void* (for no return), nothing (for constructors only) or any *type* or *class* within the scope of the containing *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 created in the same manner 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.

7. Class JavaByBugs

The first class that we will create is the class that will contain the code to initiate the window that will be used to contain the game. This class will eventually contain an instance variable of a class called *MainPanel* (described in section #8) that will contain the game itself.

Our class will be called *JavaByBugs*. This means that it must be placed in a file called *JavaByBugs.java*.

Note: If you are creating this program via a Java programming environment, you may have to place all code in a package. If so, make sure that the same package statement is the first line in each file in the program.

In this file, the first lines of code will be imports of the *Abstract Windows Toolkit* (which will be used to provide some basic window objects) and *Swing*, which will be used to create the window itself and most window components.

```
import java.awt.*;
import javax.swing.*;
```

After the imports, the class JavaByBugs is created.

```
public class JavaByBugs {
}
```

Since the class *JavaByBugs* is to be used to create a window for the game, it must have access to the methods of class *JFrame*, a part of *Swing*. The simplest way to do this is to inherit *JFrame* into *JavaByBugs* via the *extends* option on the class header. This way, the methods of *JFrame* become part of *JavaByBugs* and can be used as if they had been written in *JavaByBugs*. Here is the revised class header:

```
public class JavaByBugs extends JFrame {
}
```

Inside of class *JavaByBugs*, the first lines of code will be two *constants* that will be *global* throughout the class and will hold the number of pixels in the dimensions of the window. Constants in Java are created by placing the descriptor *final* between the access and type in an object declaration. The actual assignment of the object can be performed immediately or later in the constructor. However, assignment may be performed only once in the logic flow of a program. If the assignment is performed in a constructor, it must be performed in all constructors. (*final* has other uses. *final* methods can not be overwritten by inheriting classes, *final* method parameters cannot have values changed in the method.) For these two constants, the use of *private* access insures that they cannot be accessed from outside of the class.

```
public class JavaByBugs extends JFrame {

    private final int HEIGHT = 400;
    private final int WIDTH  = 500;
```

Next to be created in the class JavaByBugs, will be the method *main*. All Java programs that are not applets must have a method *main*. As in C++, the first code executed is the code in *main*. The syntax of the header and body for function main is as follows:

```
    public static void main(String args[]) {
    }
```

where *args[ ]* is the array that gives access to command line arguments (as in C++)
static is the descriptor that indicates that method *main* can be accessed without an instance variable of the class being created. (This is necessary to allow the program to be run!)

Besides letting a class member be directly accessed without an instance variable – a very handy thing such as when a member of class *Math* is needed [example: Math.sin(angle)], the descriptor *static* creates several problems, the worst of which is that *static* methods can only directly call other *static* methods in the class. While this does help the process of encapsulation, it means that class *JavaByBugs* needs to contain a *constructor* in order that an instance variable of itself that is not *static* may be created in *main*. The code of the *constructor* will take over the basic steps of setting up the window that the game will use and passing control on to the actual game code.

The basic form of the *constructor* will be:

```
public JavaByBugs() {
}
```

In the *constructor*, the size, title, closing operation, initial location and contents of the window will be set.

To set the size of the window, the method *void setSize(int width, int height)* that was inherited from *JFrame* must be called. To set the title of the window, the method *void setTitle(String)* inherited from *JFrame* must be called. To set the closing operation of the window, the method *void setDefaultCloseOperation(JFrame.closing-op)* inherited from *JFrame* must be called. To set the initial location on the screen of the upper left hand corner of the winodow, the method *void setLocation(int across, int down)* inherited from *JFrame* must be called.

```
public JavaByBugs() {
     setSize(WIDTH, HEIGHT);
     setTitle("Java by Bugs");
     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     setLocation(200,50);
```

Once the basic features of the window are set, a container object must be created to hold all of the components of the game that will be displayed in the window. Java contains a number of container classes. This tutorial will use the class *Container* and the class *JPanel*. An object of class *Container* will be used to encompass all components of that will appear in the window.

To create an instance object variable of a class, the object must be declared and created. This can be all in one step:

access class *identifier* = new constructor(*parameter-list*);

or in separate actions:

access class *identifier*;
identifier = new constructor(*parameter-list*);

Since all class instance variables in Java are dynamic, an instance variable simply points to the memory location created by the call to the constructor using *new*. [Note: You can look at this in two ways. (1) There are no pointers in Java or (2) all class instance variables are pointers in Java, just no special syntax is needed.] Additionally, the constructor called does not have to be the constructor of the class. It can be a method of a class that returns a compatible type that can be called (having indirectly called a constructor).

To create an instance variable object of class *Container*, the method *getContentPanel* is called, which is a method from *JFrame*. This is written in the constructor:

```
Container c = getContentPane();
```

Now that a container object exists, it needs to have its basic properties set. First, its layout needs to be set with a call to the *setLayout* method of class *Container*. This has the following syntax:

void setLayout(*layout-class-object*)

There are a number of layout classes. This tutorial will use the simplest, class *BorderLayout*, which has the relative container locations *NORTH*, *SOUTH*, *EAST*, WEST and *CENTER*. They are relative because if a location is not used, its neighbor fills it in as if that location does not exist. This tutorial will use only *CENTER*. *(A useful exercise is to populate other locations when the tutorial has been completed and to see what adjustments must be made to the code in order to accommodate the new location.)* The *Container* object *c* will have its layout set by the code:

```
c.setLayout(new BorderLayout());
```

Other basic properties of a *Container* class object that often get set are its default font and its background color. This tutorial will set them, but they are not directly used by the rest of the code and can be left out.

To set the default font, the *Container* class method *setFont* will be called. This method has the syntax:

void setFont(*font-class-object*)

The constructor for class *Font* that will be used has the syntax:

Font (*String font*, *font-property*, *int size*)

The example of setting a font is:

```
c.setFont(new Font("SansSerif", Font.BOLD, 12));
```

Setting the background color requires a call to the *Container* method *setBackground*, which has the syntax:

void setBackground(*Color*)

where *Color* is a color as defined in class color or a *Color* object created by a call to the *Color* class constructor;

The *Color* constructor syntax is:

Color(int *red*, int green, int *blue*)

where *red*, *green* and *blue* are int expressions that can have the values 0-255

The background color of the *Container* object *c* will be set to white by:

```
c.setBackground(Color.white);
```

Thus far the constructor for class JavaByBugs looks like this:

```
public JavaByBugs() {
     setSize(WIDTH, HEIGHT);
     setTitle("Java by Bugs");
     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     setLocation(200,50);

     Container c = getContentPane();

     c.setLayout(new BorderLayout());
     c.setFont(new Font("SansSerif", Font.BOLD, 12));
     c.setBackground(Color.white);
}
```

Now that the constructor exists, the method *main* can be completed. First in *main*, an instance variable object of class *JavaByBugs* can be created using the constructor:

```
JavaByBugs mainFrame = new JavaByBugs();
```

At this point, all that remains to complete method *main* is to make the window visible. This is done by a call to the method *show*, which was inherited from *JFrame*.

```
mainFrame.show();
```

As of now, method *main* is complete and has the final form:

```
public static void main(String args[]) {
     JavaByBugs mainFrame = new JavaByBugs();
     mainFrame.show();
}
```

The class JavaByBugs now has the form:

```
import java.awt.*;
import javax.swing.*;

public class JavaByBugs extends JFrame {

   private final int HEIGHT = 400;
   private final int WIDTH  = 500;

   public JavaByBugs() {
        setSize(WIDTH, HEIGHT);
        setTitle("Java by Bugs");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocation(200,50);

        Container c = getContentPane();

         c.setLayout(new BorderLayout());
         c.setFont(new Font("SansSerif", Font.BOLD, 12));
         c.setBackground(Color.white);
   }

   public static void main(String args[]) {
        JavaByBugs mainFrame = new JavaByBugs();
        mainFrame.show();
   }

}
```

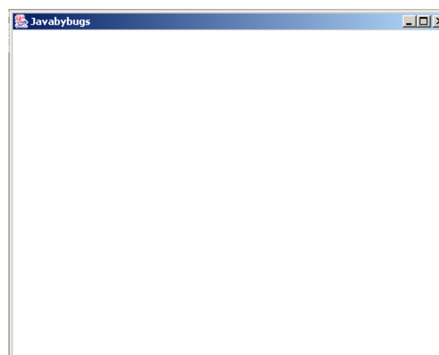
JavaByBugs is now ready to be tested. First compile the file that contains the class:

```
javac JavaByBugs.java
```

This will create the file *JavaByBugs.class*, which can be run by:

```
java JavaByBugs
```

This will produce the window:



8. Class MainPanel

A number of paths could be taken at this point. Rather than over-stuff class *JavaByBugs*, a separate class will be created that will inherit the *Swing* class *JPanel*. This class will be *MainPanel*. By inheriting *JPanel*, *MainPanel* will be used to assemble and update the components of the game. The constructor in *JavaByBugs* will declare an object of class *MainPanel* and will add that object to the *Container* object *c*. This will place the game in the window created in *JavaByBugs*.

Class *MainPanel* will be created in a file called *MainPanel.java*. It will need to import the same code as class *JavaByBugs*. In addition, it will also need to import the *event* package in order to listen for user input. This will be written:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

Class *JPanel* is a container class. That is, like objects of class *Container*, objects of class *JPanel* are able to have objects added to them. Objects of class *JPanel* can be added to regions of a *Container* object, based on the *Container* class object's layout. (Think of boxing a set of dishes for moving, then putting the box on the moving van.)

Class *MainPanel* will inherit class *JPanel*.

```
public class MainPanel extends JPanel{
}
```

Like class *JavaByBugs*, class *MainPanel* will need to know what its boundaries will be in pixels. To that end two global constants will be needed to hold that information and a constructor that will receive values for those constants will need to be created. This will be:

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

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

Two more methods will be needed to complete a minimal version of class *MainPanel*. The first will be *paintComponent*, which was inherited from *JPanel* and allows the class to place graphical components in the panel. The class *MainPanel* must overwrite it in order to place the components of the game in the window. The second will be the programmer created method *display*, which will exist only to call method *repaint*. Method *repaint* is an inherited method that exists to allow *paintComponent* to be called by any other method. This is necessary because method *paintComponent* must be passed an argument of class *Graphics*, which will not be available to all methods.

These two methods have the basic form:

```
public void paintComponent(Graphics g) {
}

public void display() {
     repaint();
}
```

The keyword *super* is used to call methods in the inherited (super) class that have otherwise been replaced. This is necessary in order to access the super class constructor or any super class methods that have been rewritten. The syntax of a *super* constructor call is:

super(*parameter-list*);

The syntax of a *super* method call is:

super.*method*(*parameter-list*);

Method *paintComponent* will start off by calling the method *paintComponent* of JPanel via the keyword *super*. This is done in order to pass the *Graphics* object *g* to that original method so the graphical properties of the *JPanel* object can be returned and used in *paintComponent* of class *MainPanel*. If this call were not to be made, there would be nothing to draw with! This is:

```
super.paintComponent(g);
```

For now, two more statements will be added to *paintComponent*. These will be two calls to methods of class *Graphics*. The first sets the drawing color and has the syntax:

void setColor(*Color*);

The second draws a filled rectangle and has the syntax:

void fillRect(*int x*, *int y*, *int across*, *int down*);

where *x* is the upper left across start of the rectangle
y is the upper left down start of the rectangle
across is the width of the rectangle in pixels
down is the height of the rectangle in pixels(starting from the top)

These two calls will be

```
g.setColor(Color.black);
g.fillRect(0, 0, WIDTH, HEIGHT);
```

The second of these calls creates a black background that will overwrite any objects previously drawn.

At this point class MainPanel has the form:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MainPanel extends JPanel{

   private final int WIDTH;
   private final int HEIGHT;

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

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

   public void display() {
        repaint();
   }

}
```

The file containing class *MainPanel* can be compiled to test its syntax:

```
javac MainPanel.java
```

9. Class JavaByBugs Redux

An object of class *MainPanel* can now be added to class *JavaByBugs*. This could be as a global variable or a variable that is local to the constructor. For no specific reason, the tutorial will declare global variable:

```
private MainPanel gamePanel;
```

Then, in the constructor of *JavaByBugs*, the *MainPanel* object must be created. This is done by:

```
gamePanel = new MainPanel(WIDTH, HEIGHT);
```

The *Container* class method *add* allows other containers and components to be added to container objects. It has the syntax:

void add(*container-or-component*, *layout-region*)

The *MainPanel* object *gamePanel* must be added to the *Container* object *c*. This is done by:

```
c.add(gamePanel, BorderLayout.CENTER);
```

Class J*avaByBugs* now looks like this:

```
import java.awt.*;
import javax.swing.*;

public class JavaByBugs extends JFrame {

  private final int HEIGHT = 400;
  private final int WIDTH  = 500;

  private MainPanel gamePanel;

  public JavaByBugs() {
       setSize(WIDTH, HEIGHT);
       setTitle("Java by Bugs");
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       setLocation(200,50);

       Container c = getContentPane();

        c.setLayout(new BorderLayout());
        c.setFont(new Font("SansSerif", Font.BOLD, 12));
        c.setBackground(Color.white);

       gamePanel = new MainPanel(WIDTH, HEIGHT);
       c.add(gamePanel, BorderLayout.CENTER);
  }

  public static void main(String args[]) {
       JavaByBugs mainFrame = new JavaByBugs();
       mainFrame.show();
  }

}
```

At this point, class *JavaByBugs* can be compiled and, since class *MainPanel* has already been compiled, the program can be run.

```
javac JavaByBugs.java
java JavabyBugs
```

When run, the program produces: