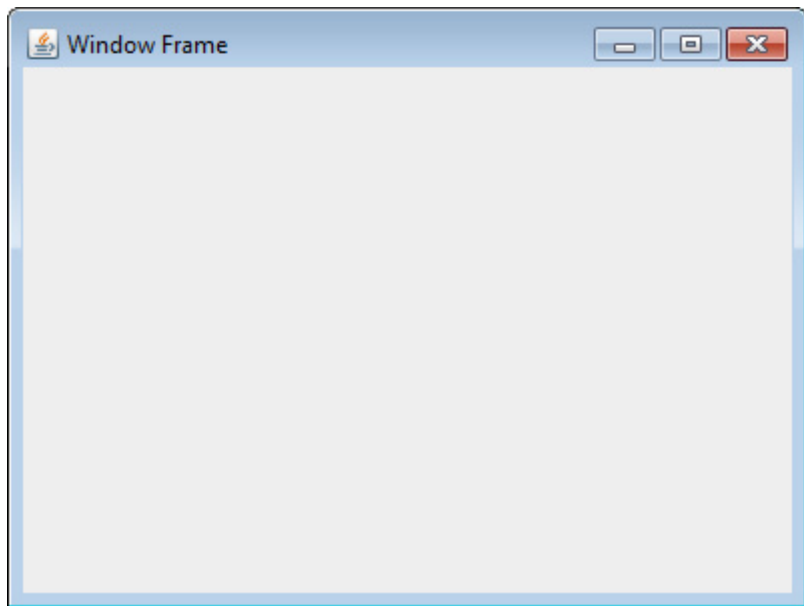


# Chapter 2: Looking at a Java Window

## Looking at a Java Window

A computer window looks different in different operating systems. Since Windows is the dominant system, we'll use Windows illustrations here. Here's a snapshot of a Windows frame and its visible parts:

### Introduction



A Java frame in Windows

Let me describe the various parts of the window, which Java programmers usually call a *frame*. Then we'll look at the code that created it.

This frame has three parts that are visible and more that aren't. The first visible part is the title bar across the top, which contains a small logo, the frame's name, and three buttons provided by the operating system that minimize, maximize, and close the window.

The second visible part of the frame is the border—the blue edge around the other three sides.

The last visible part is the remainder of the frame itself, inside the border. The title bar and the border cover part of the frame. (Just a reminder: The title bar and border sizes will vary between operating systems, so what you see on your computer may not look exactly like mine.) Java lets us figure out the actual visible part of the frame we can use, though. For example, the program that produced the above window also gave me this output with information about it.

---

Text equivalent start.

The Terminal Window with program output:

Window Size is 400 by 300

Window top inset is 30

Window bottom inset is 8

Window left inset is 8

Window right inset is 8

Text equivalent stop.

---

You can see from this that the size of the window in figure 1 was 400 pixels wide by 300 pixels tall. The title bar is 30 pixels tall in my operating system, and the three borders are each eight pixels thick.

Here's the code that produced that window and the text output:

```

import java.awt.Dimension;
import java.awt.Insets;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class WindowFrame extends JFrame {

    public WindowFrame() {
        setTitle("Window Frame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 300);
        setLocationRelativeTo(null); // centers window in display screen
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                WindowFrame dw = new WindowFrame();
                dw.setVisible(true);
                Dimension dim = dw.getSize();
                Insets ins = dw.getInsets();
                System.out.println("Window Size is " + dim.width + " by " +
dim.height);

                System.out.println("Window top inset is " + ins.top);
                System.out.println("Window bottom inset is " + ins.bottom);
                System.out.println("Window left inset is " + ins.left);
                System.out.println("Window right inset is " + ins.right);
            }
        });
    }
}

```

Let's go through this code piece by piece so you understand it.

## Java's GUI Tool Packages: AWT and Swing

The imports for this program are from two packages: `java.awt` (AWT or Abstract Window Toolkit) and `javax.swing`. These are Java's two main GUI tool packages.

AWT dates from the beginning of Java's history. It contains many of the basic GUI and graphics classes that Java still uses, and it provides a Java interface to many of the operating system's GUI objects.

You'll see many GUI programs that use wild cards to import the complete packages, like this:

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

I chose to import the classes we needed for this example individually so it would be clear which classes we're using from each package.

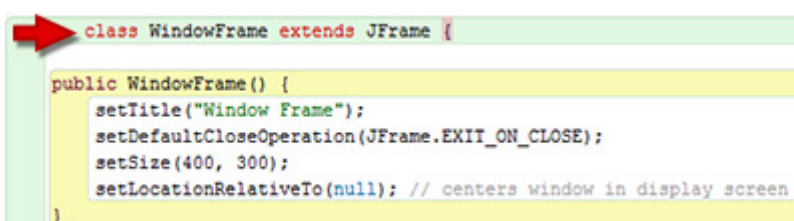
The Swing package is an update to Java's GUI tools introduced in Java 1.2 that provides more consistent-looking GUIs across platforms. It uses Java GUI objects instead of those that the various operating systems provide. Some designers prefer AWT tools because the GUI applications developed that way look more like those developed for the specific operating system. So a Java GUI app in Windows looks more like a native Windows app, and the same app in Mac OS X looks more like a native Mac app.

Other designers prefer the tools in Swing because they're more consistent across platforms, with the same app having the same Java look on both Windows and Mac platforms. The Swing toolkit also provides more built-in options for designers to use.

I'm going to take a standard approach in this course and use Swing for our GUI lessons. Many Swing classes have class names that start with "J" to distinguish them from their AWT counterparts.

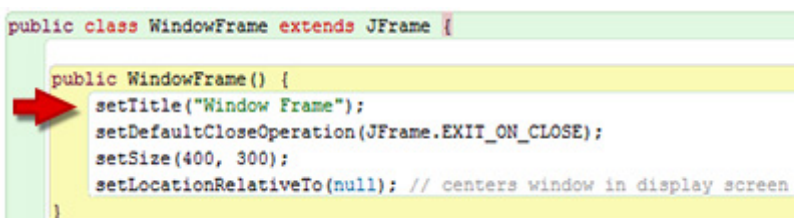
## The JFrame Class and Constructor

The class declaration in our code names our class JFrame and extends the JFrame class, which is Swing's frame class. As I mentioned in Lesson 8, extending a class means that our class will inherit all the data elements and methods from JFrame in addition to what we add to it in JFrame. So we'll already have everything included that we need to set up our window. Our basic job will be to set its parameters so it looks and behaves the way we want it to.



```
class JFrame extends JFrame {
    public JFrame() {
        setTitle("Window Frame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 300);
        setLocationRelativeTo(null); // centers window in display screen
    }
}
```

The class constructor takes care of most of that. (As I mentioned in Lessons 3 and 4, a constructor is a special method that gives objects a value to start with, even though that value may change later.) The first line calls JFrame's setTitle() method to set the text we want to appear in the window's title bar.



```
public class JFrame extends JFrame {
    public JFrame() {
        setTitle("Window Frame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 300);
        setLocationRelativeTo(null); // centers window in display screen
    }
}
```

The next line sets the behavior we want our window to take when a user clicks its close button in the title bar (a red rectangle in Windows and a red circle on a Mac). The default action is to hide (minimize) the window but leave the program running. By using the Java constant JFrame.EXIT\_ON\_CLOSE, we're telling Java we want the application to shut down when the user clicks that button.

```
public class WindowFrame extends JFrame {  
    public WindowFrame() {  
        setTitle("Window Frame");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setSize(400, 300);  
        setLocationRelativeTo(null); // centers window in display screen  
    }  
}
```

The constructor's third line sets the size we want for our window using JFrame's `setSize()` method. I set this example window to 400 pixels wide by 300 pixels tall.

```
public class WindowFrame extends JFrame {  
    public WindowFrame() {  
        setTitle("Window Frame");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setSize(400, 300);  
        setLocationRelativeTo(null); // centers window in display screen  
    }  
}
```

The last line in the constructor tells Java where to put the window when it opens. The default is to put it at the top left of the screen. You can set its absolute location using the `setLocation()` method. The option I chose, calling `setLocationRelativeTo(null)`, tells Java to center the window in the display screen.

```
public class WindowFrame extends JFrame {  
    public WindowFrame() {  
        setTitle("Window Frame");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setSize(400, 300);  
        setLocationRelativeTo(null); // centers window in display screen  
    }  
}
```

Those are all the settings we need in our window class, so that's the end of our class constructor.

## WindowFrame's main() Method

The only thing left to do is set up a `main()` method that creates our `WindowFrame` object and displays it on-screen. So that's what we'll look at next.

Probably the strangest and hardest part of this whole process to explain is the first two lines of the `main()` method, so we're going to talk about them for a bit. Just to save you a little scrolling, here's what they look like again:

```
SwingUtilities.invokeLater(new Runnable() {  
    public void run() {
```

These lines involve several topics that we haven't discussed yet, and you probably don't have the background yet to understand in detail. So I'm going to give you an option here: If you like, you can accept that this is Oracle's recommended way to start a GUI app and skip to Chapter 3.

Now if you're reading this paragraph I'm assuming you're a diehard "gotta-know-how-it-works" fanatic. So here goes . . .

Java has the ability to multitask, or in technical terms, run multiple *threads*. A thread is an independent execution path. If I start a new thread and run some code on it, it can execute in parallel with, and completely independently of, the program that started it. That's an efficient approach in the world of multicore computers.

Java maintains a thread it calls its AWT Event Dispatch Thread, and that thread should manage all GUI updates. The `SwingUtilities` class method named `invokeLater()` sets up an object to use that thread as soon as the thread is available. The only requirement is that the object be a `Runnable` object, which means it contains a `run()` method that the thread can call.

The parameter for the call to `invokeLater()` in this case is an anonymous `Runnable` object, which you can create with the code `new Runnable() { }`. Within those brackets, you'd define a `run()` method that tells Java what this object should do. The `run()` method contains the code that will create our new window. I'll describe that code in Chapter 3.

A full explanation of threads in general and the Event Dispatch Thread in particular would take more time and space than we have. But a couple of links in the Supplementary Material relate to threads and *concurrency* (the official term for running more than one thing in parallel).

Let's move on to creating a Java window and displaying it.