Building a Java First-Person Shooter

Episode 1 – Windows [Last update: 5/03/2017]

Objectives

In this episode you are presented with enough information to:

- Using the Eclipse IDE
- Create a window
- Set it to a certain size
- Have it close cleanly
- Display a title in the window

Video URL

https://www.youtube.com/watch?v=iH1xpfOBN6M

Discussion

What is an IDE?

A hot topic of discussion among professional programmers is the tool or application they use for software development.

The top Java development IDEs are Eclipse, NetBeans, and IntelliJ. A good IDE is invaluable in the creation and maintenance of code. I highly recommend that you obtain and download Eclipse as you follow along the videos and these notes. Episode 0 lists URLs on the Internet to learn how to build Java projects with Eclipse.

Organizing our Workspace

I have decided to create an Eclipse workspace named CHERNO_01 to hold a Java project for each Episode. At the start of every episode I copy the previous episode's Java project into a new one so that we have continuity of our work product as we view the video episodes. There will be side-projects unrelated to the video episode but used to clarify a new concept.

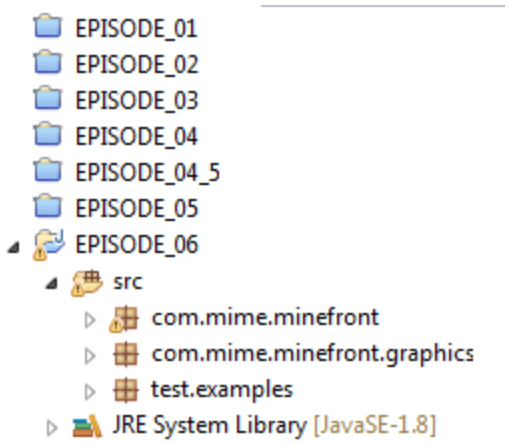

Figure 1 - Collection of EPISODE Java projects

Each episode Java project contains the video episode's final code. The only non-episode package will be test.examples. The test.examples will contain the additional programs we create to test concepts.

Creating our new Project

What version of Java?

Make sure you have Java 1.6 or greater installed.

The Cherno Workspace

STEP 1:

Create a new Java project named EPISODE_01.

STEP 2:

Create a new class `com.mime.minefront.Display` that extends the `Canvas` class. This class will be the "kickoff" class for our game, that is, it will contain a `main()` method that will be invoked by the platform operating system (OS).

STEP 3:

Enter the following code, compile and execute.

Table 1 – Display.java (version #1)

```
package com.mime.minefront;

import java.awt.Canvas;
import javax.swing.JFrame;

public class Display extends Canvas {

	private static final long serialVersionUID = 1L;

	public static final int WIDTH = 800;
	public static final int HEIGHT = 600;
	public static final String TITLE = "Minefront Pre-Alpha 0.01";
```

```
	public static void main(String[] args) {
		Display game = new Display();
		JFrame frame = new JFrame();
		frame.add(game);
		frame.setTitle(TITLE);
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.setSize(WIDTH, HEIGHT);
		frame.setLocationRelativeTo(null);
		frame.setResizable(false);
		frame.setVisible(true);

		System.out.println("Running...");
	}
}
```

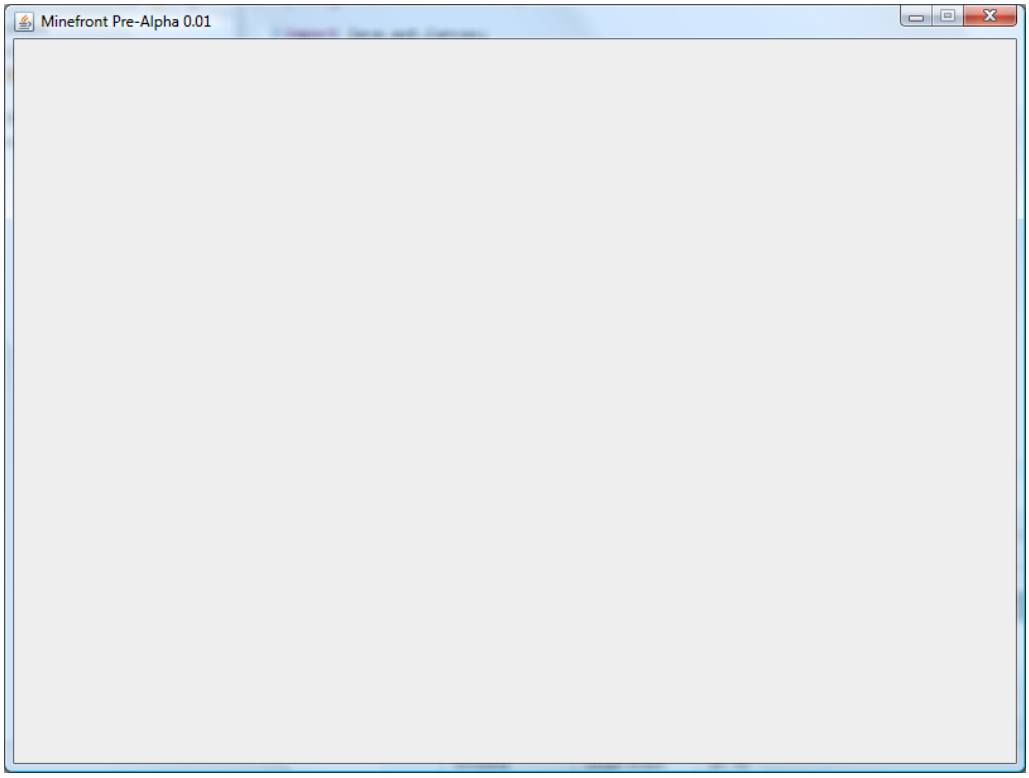

Figure 2 - The Minefront application window

How the code works

AWT and Swing

The code uses two classes from two different libraries. The first class `java.awt.Canvas` comes from what is referred to as the `AWT` library. `AWT` stands for `Abstract Window Toolkit`. This library came with the original or first version of Java (1995). `AWT` provided the "windowing, graphics and user-interface widget toolkit." So it is the library to use to create any GUI application that is, applications with windows, button, menus and textboxes. In order to come up with something very quickly that could

provide graphic functionality across many machines and platforms `AWT` was purposely designed as a thin layer between Java and the actual platform[1]'s graphical APIs. That is the graphic component (windows, buttons, menus, etc.) are rendered by the platform operating system graphics library. There were two major problems with this approach. The first is the lowest common denominator of graphics and window functionality was provided and second all the applications took on the look and feel of their native platform so you could not get applications to look (and even behave) the same across platforms since it was their underlying operating systems APIs that was being used to draw and manage the screen.

The class `javax.swing.JFrame` comes from the `Swing` library. In Java version 1.2, Sun Microsystems introduced the `Swing` toolkit. `Swing` was first developed by the then Netscape Communications Corporation in 1996. The goal was to develop sharper and more elegant looking GUI components than what was provided by `AWT`. `Swing` was developed so that applications would appear the same across different platforms. In addition, the look and feel was intended to be pluggable. The library provided a richer set of widgets that were implemented strictly in Java.

The current version of Java handles more easily the mixing of components from both toolkits (that was once a problem). The two libraries are not independent from each other as the class hierarchy diagram below illustrates:

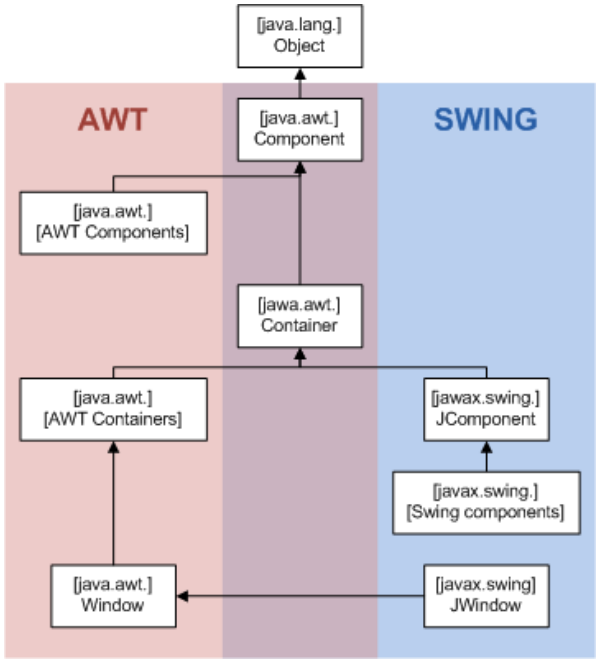

Figure 3 – AWT/Swing class hierarchy (from http://en.wikipedia.org/wiki/File:AWTSwingClassHierarchy.png)

Both `AWT` and `Swing` are now part of the **Java Foundations Classes** (JFC). The fundamental GUI object shared by both `AWT` and `Swing` is the `java.awt.Component` class. A component is an object having a graphical representation that can be displayed on the screen. So all the entities we see on a Java GUI

[1] The first version of Java ran on the following platforms: Windows 95 and NT, Sun Solaris and later Mac OS 7.5.

screen derives from the component class. There are two types of components – *lightweight* and *heavyweight*. A lightweight component is not associated with a native window (this is true for all `Swing` components) and a heavyweight component is associated with a native window (this is true for all `AWT` components). A container is just an object that can contain other components.

It is easy to tell the difference between classes that are associated with AWT from those that are associated with Swing – all Swing classes start with the letter J (e.g. `JComponent`, `JWindow`, `JFrame`, etc.). In addition, the `AWT` classes reside in the `java.awt` package and the `Swing` classes reside in the `javax.swing` package.

Canvas

Figure 4 - From http://mainline.brynmawr.edu/Courses/cs110/fall2003/Applets/CanvasExample/CanvasExample.html

The `Canvas` component represents the area on the screen where your applications draw images, buttons, text, missiles, bombs, and dogs! Applications (like our example) subclass the class `Canvas` in order to gain access to the functionality offered by that class.

Typical use:

Table 2 - Typical use of Canvas

```
public class MyGreatGameDisplay extends Canvas {
}
```

This episode does not involve any painting to the screen but typically you would use the `paint(Graphics g)` method to draw your blown up enemies on the screen (more on this in future episodes[2]) if you were using `AWT` Frame to hold your window. Since we are using `JFrame` a `Swing` class we should override `paintComponent()` method instead. In this version of the program we create the `Display` class:

[2] We will not be relying on paint() to do our actual drawing since games require we control when things are drawn – more later…

```
public class Display extends Canvas {
            :
            :
}
```

It should be obvious why we are calling this class "Display" since it will handle the game display screen.

The first thing we will need is a window to display our game – hence the use of `JFrame` to hold our `Canvas`.

JFrame

Figure 5 - JFrame class hierarchy

A Frame is a top-level window with a title and a border. The size of the frame includes an area designated for the border. The dimensions of the border area may be obtained using the `getInsets` method. Since the border area is included in the overall size of the frame, the border effectively obscures a portion of the frame, constraining the area available for rendering and/or displaying subcomponents to the rectangle which has an upper-left corner location of (`insets.left, insets.top`), and has a size of `width - (insets.left + insets.right)` by `height - (insets.top + insets.bottom)`.[3] What all this means is that even if you created the frame to be WIDTH x HEIGHT your Canvas object will have less space than that for drawing since the insets or window borders take room. The equations above detail how much less. In a future, episode CHERNO will make adjustments to ensure the Canvas or drawing area matches our desired WIDTH x HEIGHT specifications.

[3] http://docs.oracle.com/javase/tutorial/uiswing/components/frame.html This whole section is from this site.

A frame, implemented as an instance of the `JFrame` class, is a window that has decorations such as a border, a title, and supports button components that close or iconify the window. Applications with a GUI usually include at least one frame. Applets sometimes use frames, as well.

Here is a picture of the extremely plain window created by the FrameDemo demonstration application.

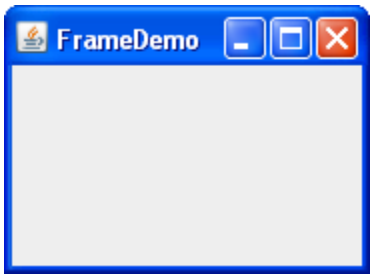

Figure 6 - FrameDemo window

The following FrameDemo code shows how to create and set up a frame.

```
//1. Create the frame.
JFrame frame = new JFrame("FrameDemo");

//2. Optional: What happens when the frame closes?
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//3. Create components and put them in the frame.
//...create emptyLabel...
frame.getContentPane().add(emptyLabel, BorderLayout.CENTER);

//4. Size the frame.
frame.pack();

//5. Show it.
frame.setVisible(true);
```

Here are some details about the code:

```
//1. Create the frame.
JFrame frame = new JFrame("FrameDemo");
```

1. The first line of code creates a frame using a constructor that lets you set the frame title. The other frequently used `JFrame` constructor is the no-argument constructor.

```
//2. Optional: What happens when the frame closes?
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

2. Next the code specifies what happens when your user closes the frame.
   The EXIT_ON_CLOSE operation exits the program when your user closes the frame. This behavior is appropriate for this program because the program has only one frame, and closing the frame makes the program useless.

```
    //3. Create components and put them in the frame.
    //...create emptyLabel...
    frame.getContentPane().add(emptyLabel, BorderLayout.CENTER);
```

3. The next bit of code adds a blank label to the frame content pane. If you're not already familiar with content panes and how to add components to them, please read Adding Components to the Content Pane.

```
//4. Size the frame.
frame.pack();
```

4. The `pack` method sizes the frame so that all its contents are at or above their preferred sizes. An alternative to `pack` is to establish a frame size explicitly by calling `setSize` or `setBounds` (which also sets the frame location). In general, using `pack` is preferable to calling `setSize`, since `pack` leaves the frame layout manager in charge of the frame size, and layout managers are good at adjusting to platform dependencies and other factors that affect component size.

   This example does not set the frame location, but it is easy to do so using either the `setLocationRelativeTo` or `setLocation` method. For example, the following code centers a frame onscreen:

   ```
   frame.setLocationRelativeTo(null);
   ```

5. Calling `setVisible(true)` makes the frame appear onscreen. Sometimes you might see the `show` method used instead. The two usages are equivalent, but we use `setVisible(true)` for consistency's sake.

The program below creates a JFrame object and adds our canvas to the frame:

```
            Display game = new Display();
            JFrame frame = new JFrame();
            frame.add(game);
```

The next couple of instructions just set certain characteristics of our window:

```
            frame.setTitle(TITLE);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(WIDTH, HEIGHT);
            frame.setLocationRelativeTo(null);
            frame.setResizable(false);
            frame.setVisible(true);
```

The code above sets the title of the window, what action to take when the user clicks on the "close window" icon, the size of our window, where to place the window on the screen, if our window is resizeable, and then to make is visible to the user.

The following text presents more details on the other methods in our first episode:

setLocationRelativeTo

`public void setLocationRelativeTo(Component c)`

Sets the location of the window relative to the specified component. If the component is not currently showing, or c is null, the window is placed at the center of the screen. The center point can be determined with GraphicsEnvironment.getCenterPoint.

setResizable

public void setResizable(Boolean resizable)

Sets whether this frame is resizable by the user.

add

public Component add(Component comp)

Appends the specified component to the end of this container.

serialVersionUID?

There will be times when you will need to 'save' the state of a game. This will require that we save the objects into tables or files and retrieve them later. The act of saving an object 'state' (the member variable values) is called serialization. The act of restoring or retrieving saved objects from a file or table is called deserialization.

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an `InvalidClassException`. A serializable

class can declare its own serialVersionUID explicitly by declaring a field named "`serialVersionUID`" that must be static, final, and of type `long`:

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;
```

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is *strongly recommended* that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected `InvalidClassExceptions` during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members.

FROM: http://stackoverflow.com/questions/285793/what-is-a-serialversionuid-and-why-should-i-use-it

If the objects you create from the class can be saved or serialized to a file for later processing (e.g. the user issues a “save game” operation you will want to save all the game objects) than you will want to create a unique serialVersionUID for the objects associated with the class. Doing so allows you to “check” if a future version of the class can handle the version saved to a file.

Improvements/Suggestions

You can skip this section. It discusses ways to ensure that the display area – the `Canvas` or `JPanel` being used to display game elements has the desired size.

Old Java Style

I would have recommended another name for this class rather than Display. It will get confusing later with the addition of a class called Render and Screen since they could all mean the same thing. The Display class will be responsible for displaying our “Screen” to the actual screen or monitor

We will now explore interesting aspects of having a JFrame and placing components to the frame.

STEP 1:

Create a new package `test.examples` where we will place all our example side classes that are unrelated to the video.

STEP 2:

Create the java class Board.java that holds our game drawing area – a JPanel.

Table 3 - Board.java

```
package test.examples;

import javax.swing.JPanel;

public class Board extends JPanel {

	public Board() {

	}
}
```

I prefer to use a `JPanel` with a `JFrame` but a `JPanel` is basically the same Swing equivalent to a `Canvas`.

STEP 2:

Create the class MyCoolGame that pretty much performs all the work shown in the video.

Table 4 - MyCoolGame.java

```
package test.examples;

import javax.swing.JFrame;

public class MyCoolGame extends JFrame {

	public final int WIDTH = 300;
	public final int HEIGHT = 280;

	Board displayArea;

	public MyCoolGame() {
		// Create our board or display area
		displayArea = new Board();

		// attach it to the window
		add(displayArea);

		// set the window title
		setTitle("My Cool Game");

		// ensure that closing the window - closes the application
		setDefaultCloseOperation(EXIT_ON_CLOSE);

		// set the size of our window
		setSize(WIDTH, HEIGHT);

		// center the window in the middle of our screen
		setLocationRelativeTo(null);

		// don't allow users to resize our window
```

```
                setResizable(false);

                // show the window
                setVisible(true);          }

        public static void main(String[] args) {
                MyCoolGame game = new MyCoolGame();
        }
}
```

STEP 3:

Build and run `MyCoolGame`.

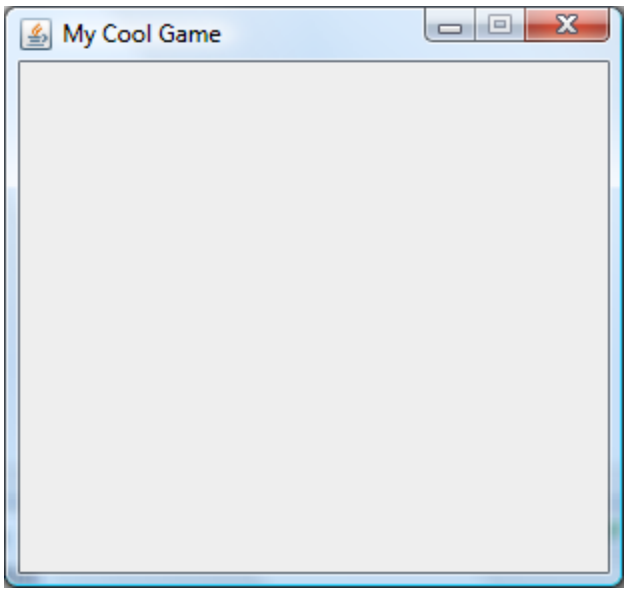

Figure 7 - My Cool Game window

Even though we defined the frame to be 300 x 280 the canvas/jpanel or displayable area will be less than this.

How do we know this to be true?

STEP 3:

Add the following new method to MyCoolGame class:

```
        public void printElementSizes() {
                System.out.println("Window size - width: " + this.getWidth()
                        + " ;height: " + this.getHeight());
                System.out.println("Board size - width: " + displayArea.getWidth()
                        + " ;height: " + displayArea.getHeight());
        }
```

```
```

The method `printElementSizes()` prints the width and height of the window and the board components.

STEP 4:

Add the following line to the main method:

```
	public static void main(String[] args) {
		MyCoolGame game = new MyCoolGame();
		game.printElementSizes();
	}
```

STEP 5:

Run the program again.

```
Window size - width: 300;height: 280
Board size - width: 284height: 242
```

As you can see above the window frame has our desired width and height as defined in the program:

```
public final int WIDTH = 300;
public final int HEIGHT = 280;
```

The display area is smaller because of the room taken up for the title and window borders.

What we should do is reset the window size to make our displayable area match our expectations of being WIDTH x HEIGHT. The way to do this is to obtain the border values and re-adjusting the window size. Here is how:

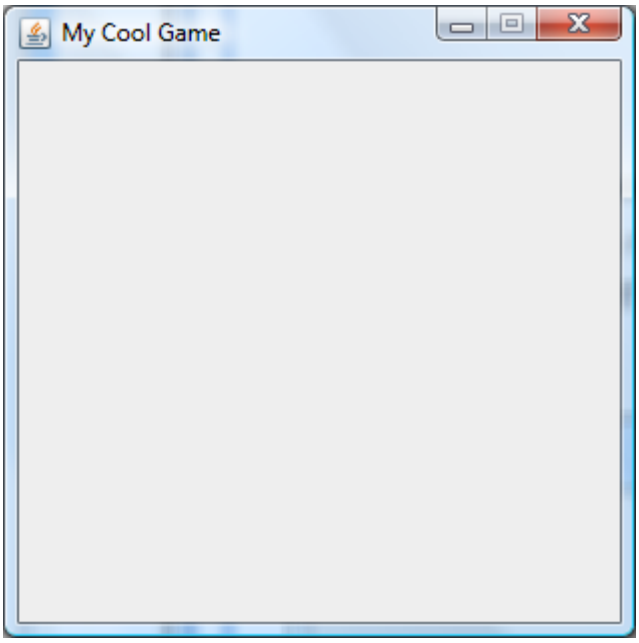

Figure 8 - Adjusted window

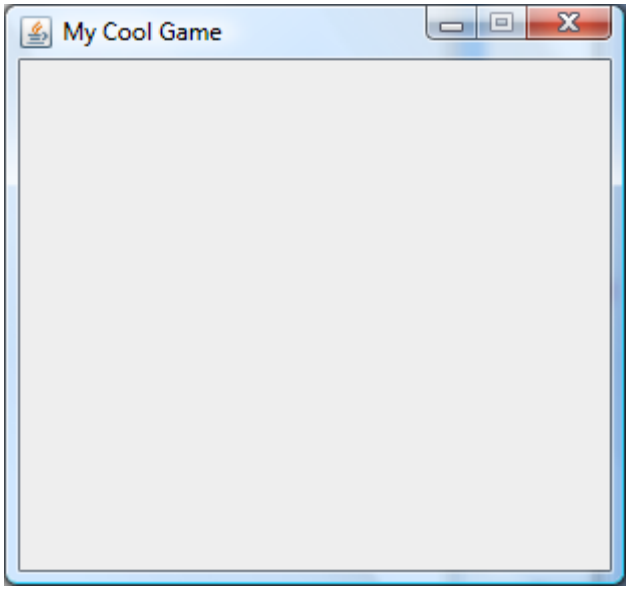

Figure 9 - Original window

STEP 6:

Add new code to MyCoolGame.java to ensure the window is sized to fit the display area we want.

Table 5 - New version of MyCoolGame.java

```
package test.examples;

import java.awt.Insets;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class MyCoolGame extends JFrame {

	private static final long serialVersionUID = 1L;

	public final int DISPLAY_WIDTH = 300;
	public final int DISPLAY_HEIGHT = 280;

	Board displayArea;

	public MyCoolGame() {
		// Create our board or display area
		displayArea = new Board();

		// attach it to the window
		add(displayArea);
```

```
		// set the window title
		setTitle("My Cool Game");

		// ensure that closing the window - closes the application
		setDefaultCloseOperation(EXIT_ON_CLOSE);

		// set the size of our window
		setSize(WIDTH, HEIGHT);

		// center the window in the middle of our screen
		setLocationRelativeTo(null);

		// don't allow users to resize our window
		setResizable(false);

		// show the window
		setVisible(true);

		// Adjust the window so that the board has the desired dimensions
		resizeToInternalSize(DISPLAY_WIDTH, DISPLAY_HEIGHT);

	}

	private void resizeToInternalSize(int internalWidth, int internalHeight) {
		Insets insets = getInsets();
		final int newWidth = internalWidth + insets.left + insets.right;
		final int newHeight = internalHeight + insets.top + insets.bottom;

		Runnable resize = new Runnable() {
			public void run() {
				setSize(newWidth, newHeight);
			}
		};

		if (SwingUtilities.isEventDispatchThread()) {
			try {
				SwingUtilities.invokeAndWait(resize);
			} catch (Exception e) {
				// ignore ...but will be no no if using Sonar!
			}
		} else {
			resize.run();
		}

		validate();

	}

	public void printElementSizes() {
		System.out.println("Window size - width: " + this.getWidth() + "
;height: " + this.getHeight());
		System.out.println("Board size - width: " + displayArea.getWidth() + "
;height: " + displayArea.getHeight());
	}
```

```
	public static void main(String[] args) {
		MyCoolGame game = new MyCoolGame();
		game.printElementSizes();
	}
}
```

The new version of MyCoolGame.java adds a new method resizeToInternalSize that makes the adjustment to the displayable area (or in this case our Canvas) by adjusting the windows width and height by taking into account the actual border values. How it actually gets this done looks a bit esoteric and complex since we are creating a thread to perform the adjustment. The next episode will discuss threads and the EventDispatcher in more detail so we will not discuss this code until then. Suffice it to say that when you create a GUI program that there is a thread that manages when the GUI elements are drawn (this is out of your hands!) and we are making adjustments to our window when that thread is running.

When you run the new version of the program the console will display:

```
Window size - width: 306 ;height: 308
Board size - width: 300 ;height: 280
```

As you see our "display area" has the desired width and height – not our window.

Note: The above is additional material not discussed in the video episodes. I learned this technique from the book *Java 1.4 Game Programming* by Andrew Mulholland and Glenn Murphy. The book did not do so well when released mostly because it made the same mistake most books and video tutorials do – try to teach java and game programming at the same time. If someone new to programming ever gets to the pages discussing the game and graphics they would have been totally clueless and I imagine ready to pull out their hair. The readers who already knew how to code would have been equally frustrated and ready to pull their hair having to go through two hundred pages before getting into anything related to creating a game. Having said that I enjoyed the book and found many of the chapters worthwhile in creating 2D games. It would have been a GREAT book if it would have immediately focused on games and actually have the users build a classic 2D game from beginning to end rather than just presenting the ideas and sample programs. You can actually find the book online. I should have a link somewhere on brainycode.com

What is going on?

The code above is best explained by the oracle documentation on Swing:

Initial Threads

Every program has a set of threads where the application logic begins. In standard programs, there's only one such thread: the thread that invokes the `main` method of the program class. In applets the initial threads are the ones that construct the applet object and invoke its `init` and `start` methods; these actions may occur on a single thread, or on two or three different threads, depending on the Java platform implementation. In this lesson, we call these threads the *initial threads*.

In Swing programs, the initial threads don't have a lot to do. Their most essential job is to create a `Runnable` object that initializes the GUI and schedule that object for execution on the event dispatch thread. Once the GUI is created, the program is primarily driven by GUI events, each of which causes the execution of a short task on the event dispatch thread. Application code can schedule additionals tasks on the event dispatch thread (if they complete quickly, so as not to interfere with event processing) or a worker thread (for long-running tasks).

An initial thread schedules the GUI creation task by invoking `javax.swing.SwingUtilities.invokeLater` or `javax.swing.SwingUtilities.invokeAndWait` . Both of these methods take a single argument: the `Runnable` that defines the new task. Their only difference is indicated by their names: `invokeLater` simply schedules the task and returns; `invokeAndWait` waits for the task to finish before returning.

You can see examples of this throughout the Swing tutorial:

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

In an applet, the GUI-creation task must be launched from the `init` method using `invokeAndWait`; otherwise, `init` may return before the GUI is created, which may cause problems for a web browser launching an applet. In any other kind of program, scheduling the GUI-creation task is usually the last thing the initial thread does, so it doesn't matter whether it uses `invokeLater` or `invokeAndWait`.

In summary, this episode created a simple GUI window that will be the foundation of our 3D java application as we add classes and functionality.

New Java Way (2017)

Today it would appear the need to calculate the insets is no longer required. If you adhere to the following guidelines you will get a drawing Canvas that is exactly the WIDTH and HEIGHT you desire without requiring too much work:

1. Create the JPanel and set it to the desired WIDTH and HEIGHT
2. Add to the JFrame setting the preferred dimensions to the WIDTH and HEIGHT of the JPanel component
3. Use SwingUtilities to create the GUI within the Swing thread.

STEP 7:

We will update Board.java to give it the desired size:

Table 6 - Board.java (final for this episode)

```
package test.examples;

import java.awt.Color;
import java.awt.Dimension;

import javax.swing.JPanel;

public class Board extends JPanel {

	private static final long serialVersionUID = 1L;

	public static final Color BACKGROUND_COLOR = Color.WHITE;
```

```
	public Board() {

	}

	public Board(int width, int height) {
		setBackground(BACKGROUND_COLOR);
		setPreferredSize(new Dimension(width, height));
	}
}
```

We add a new constructor where the client provides the desired width and height of the display area. We also create a default background color and set the Board class to use it. The preferred size of the Board is specified.

STEP 8:

Copy the class MyCoolGame → MyCoolGame2.

Table 7 - MyCoolGame2.java

```
package test.examples;


import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class MyCoolGame2 extends JFrame {

	private static final long serialVersionUID = 1L;

	public final int DISPLAY_WIDTH = 300;
	public final int DISPLAY_HEIGHT = 280;

	Board displayArea;

	public MyCoolGame2() {
		// empty constructor
	}

	public void setupGame() {
		// Create our board or display area
		displayArea = new Board(DISPLAY_WIDTH, DISPLAY_HEIGHT);

		// attach it to the window
		add(displayArea);

		// set the window title
		setTitle("My Cool Game 2");

		// ensure that closing the window - closes the application
```

```
                setDefaultCloseOperation(EXIT_ON_CLOSE);

                // Size the window to fit the preferred size of its components
                pack();

                // center the window in the middle of our screen
                setLocationRelativeTo(null);

                // show the window
                setVisible(true);
        }

        public void printElementSizes() {
                System.out.println("Window size - width: " + this.getWidth() + "
;height: " + this.getHeight());
                System.out.println("Board size - width: " + displayArea.getWidth() + "
;height: " + displayArea.getHeight());
        }

        public static void main(String[] args) {

                MyCoolGame2 game = new MyCoolGame2();
                // Start the GUI work in a thread since Swing is
                // not thread-safe
                SwingUtilities.invokeLater(new Runnable() {
                        public void run() {
                                game.setupGame();
                                game.printElementSizes();
                        }
                });
        }
}
```

Figure 10 - MyCoolGame2

The code is simpler in that the window insets are not calculated in order to adjust the window size, rather the creation of the window and its elements are packaged into its own method `setupGame()` and is invoked using `SwingUtilities`.

The `pack()` JFrame method instructs the window to size itself to fit the preferred size of all its components.

Finally in this final program we see the following being displayed:

```
Window size - width: 316 ;height: 318
Board size - width: 300 ;height: 280
```

References

- http://en.wikipedia.org/wiki/Abstract_Window_Toolkit
- http://home.cogeco.ca/~ve3ll/jatutorg.htm
- http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/Canvas.html
- http://en.wikibooks.org/wiki/Java_Programming/Canvas
- http://docs.oracle.com/javase/tutorial/uiswing/components/frame.html
- http://www.javablogging.com/what-is-serialversionuid/
- http://stackoverflow.com/questions/285793/what-is-a-serialversionuid-and-why-should-i-use-it
- Mullholland, Andrew. *Java 1.4 Game Programming*. Wordware Publishing. 2003. ISBN: 978-1556229633