Building a Java First-Person Shooter

Episode 2 [Last update: 4/30/2017]

Objectives

This episode adds the basic elements of our game loop to the program. In addition, it introduces the concept of threads.

Video URL

https://www.youtube.com/watch?v=0zuVHDNYPQU

Discussion

STEP 1:

Copy the Episode_01 Java project to a new one titled EPISODE_02.

STEP 2:

Close the project EPISODE _01

The following instructions and programs will all be in EPISODE_02.

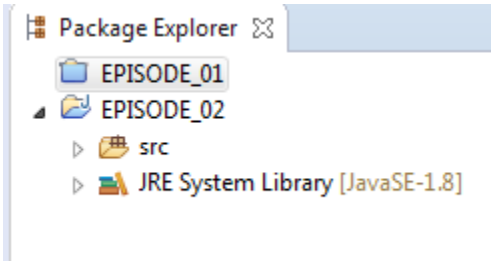

Figure 1 - Eclipse Workspace setup

What are Threads?

"The term thread is shorthand for thread of control, and a thread of control is, at its simplest, a section of code executed independently of other threads of control within a single program."[1]

The notion of threading is "so ingrained in Java" that even our most basic and simple programs use it without our being aware of it. We should be accustomed to starting our Java programs by having the operating systems invoke our `main()` method. This starts your typical single-threaded application. A Java program allows you to spawn off one or more threads that will run independently from the main thread. In a Java program each thread will have the following properties:

[1] This section is composed (stolen) from material in reference [1].

- Each thread will begin execution at a predefined well-known location. You are of course familiar with `main()` being the starting point of your application. You as a programmer get to decide the starting point of the other threads you start up.
- Each thread executes code from its starting location in an ordered, predefined sequence (the a given set of inputs)
- Each thread executes its code independently of the other threads in the program

Another thread that you know runs with your application is the garbage collector. But this thread is created and managed by the Java Virtual Machine (JVM). When you build GUI applications you know that the events (mouse click, keyboard entry, etc.) are all handled by another thread managed outside your application code - **Event Thread Dispatcher**. You have to perform special actions to inform the system that you want to handle certain events within your program. Another type of thread that you don't manage but work with is the `paint()` event. When your GUI application requires painting your application has to implement or override special methods (e.g. `paint()` or `paintComponent()`) in order to draw the game display. In these series of episodes we do not explicitly override the `paint()` method to draw on the screen but it essentially gets done because we subclass components that do invoke methods to refresh the screen.

Why do we need Threads in our game program?

Threading makes certain type of programs easier to construct. Dividing the program so that it can be split into separate tasks allows us to create conceptually easier programs that coordinate the job the application is trying to execute.

Threading Experience

My first experience with threading came up when I was asked to fix a problem a particular application was having with respect to a bad user experience. The user used the application to manage different router[2] devices. The job of the application was to download to the selected device a configuration file. The figure below shows you the setup.

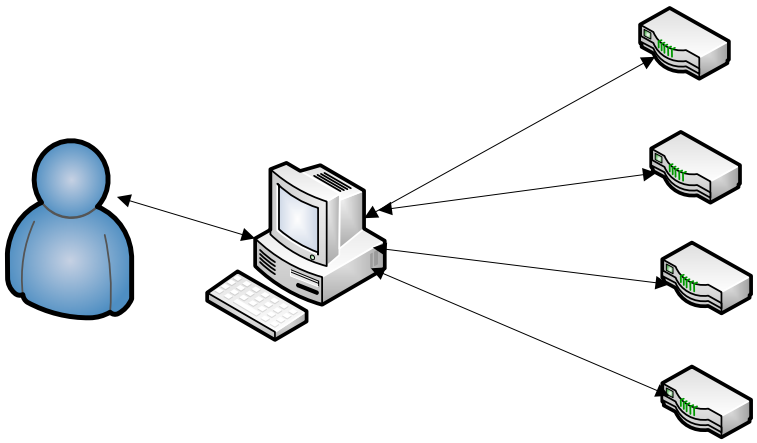

Figure 2 - User connecting to router devices

Very often the router device the user selected was not yet available and the application when reaching out to the device would wait several minutes (or at least the user reported it felt like several minutes) before the application would return from the "download request" operation to report back to the user that the operation failed because the router connection could not be established. The user knew that if the application did not get a response from the router within 15 seconds that the router was not "up" and available yet but the actual time was more like minutes because of the built-in timeout the network connection software was set at. The application was originally written as a single-threaded application its logic was something like this:

```
try {
      connectToRouter(ipAddressForRouter);
      downloadConfigurationFile();
} catch (ConnectionTimedOutException ctoe) {
      // give the user the bad news ….after waiting for several
      // minutes….what a waste of time!
}
```

The problem was in the `connectToRouter()` call. It would go off into the method trying to set up a connection to the router and the code waited and waited and waited until the connection timed out before returning with a connection timed out failure. The solution[3] that I came up with (for what was available at the time) was to execute the `connectToRouter()` in its own thread and if the thread did not complete within a time limit that I set in some property file (15 seconds) than I would assume the router was not available. I then continued program execution and reported the bad news to the

[2] If you don't know what a router is that is fine just imagine some black box that uses some file called configuration file to figure out what to do.

[3] There is a bit more about threads you will need to learn – I recommend the book "Java Threads" by Scott Oaks and Henry Wong.

user in a more timely fashion. This was a great use of threading and the user's time!

"The popularity of threading increased when graphical interfaces became the standard for desktop computers because the threading system allowed the user to perceive better program performance." In a game we have at least two things we want it to do at the same time "manage the screen display" and "execute the game loop." In order to do this will require that we learn to create and manage threads.

The goal for our games programming is to create as few threads as possible but implement them we must. The most important thread for our program at this time is the *Event Dispatcher Thread*. This is the thread that handles events (e.g. mouse click, keyboard entry, etc.) You dictate which events you want to handle in your program by specifying an event listener. This topic will be covered in a future episode.

Creating a Thread

There are two ways to create threads.

- Create a class that extends `java.lang.Thread`
- Create a class that implements the interface `java.lang.Runnable`

In **both** cases your class must define a method named `run`.

Table 1 - the Thread run() method

```
public void run() {
}
```

To terminate a thread just let the `run()` method exit.

Let's recall that the `main()` method is also a thread that the operating systems starts. Let's look at a simple example.

STEP 3:

Create the class SimpleThreadExample01 in the `test.examples` package and run.

Table 2 - SimpleThreadExample01.java

```
public class SimpleThreadExample01 extends Thread {

	public void run() {
		for (int i = 0; i < 1000; i++)
		System.out.println("New thread executing " + i);
	}

	public static void main(String[] args) {
		SimpleThreadExample01 myThread = new SimpleThreadExample01();
		myThread.start();
		for (int i = 0; i < 50; i++) {
```

```
                    System.out.println("Main thread executing " + i);
            }
      }

}
```

In the above code the `main()` method creates and starts its own thread `myThread` by invoking the method `start()`. Invoking the `start()` method actually starts the execution of the run() method. In the output you will see "New thread executing" messages mixed in with "Main thread executing" messages. This demonstrates that the `main()` thread and `myThread` are running at the same time. Since the machine is very fast each thread may print out many messages on the console before the second thread is given the CPU or maybe your output looks like the one below.

Table 3 - Output from SimpleThreadExample01

```
New thread executing 212
Main thread executing 247
New thread executing 213
Main thread executing 248
New thread executing 214
Main thread executing 249
New thread executing 215
Main thread executing 250
New thread executing 216
Main thread executing 251
New thread executing 217
Main thread executing 252
New thread executing 218
Main thread executing 253
```

In our program the main thread and the new thread (myThread) are two separate processes that are running. You can regard them as two separate programs running at the same time sharing the computer resources. The operating system is designed to give each program and each thread within each program a slice of the CPU to get work done. In our example, the work is being done through a for loop.

The second way to create and start a thread is by using the `Runnable` interface.

STEP 4:

Create and run the program SimpleThreadExample02.java shown below.

Table 4 - SimpleThreadExample02.java

```
public class SimpleThreadExample02 implements Runnable {

      @Override
      public void run() {
            for (int i = 0; i < 1000; i++)
            System.out.println("New thread executing " + i);
```

```
	}

	public static void main(String[] args) {
		SimpleThreadExample02 runner = new SimpleThreadExample02();
		Thread myThread2  = new Thread(runner);
		myThread2.start();
		for (int i = 0; i < 500; i++) {
			System.out.println("Main thread executing " + i);
		}
	}

}
```

The output for the program should be quite similar to the previous one. Note, that in the example above we create a "runner" object based on our `SimpleThreadExample02` class and provide the object as an argument to the Thread constructor for our thread object `myThread2`. We invoke the `run()` method of the thread the same way - by invoking the `start()` method of the thread.

To stop a thread you merely allow the `run()` method to exit. There are some deprecated methods in the Thread class that you should not use. One deprecated method is the `stop()` method and the other is the `suspend()` method.

In this episode we will use the second method. The video episode code is shown below:

Table 5 - Display.java (Episode 2)

```
package com.mime.minefront;

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

public class Display extends Canvas implements Runnable{

	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";

	private Thread thread;
	private boolean running = false; // indicates if the game is running or not

	private void start() {
		if (running) {
			return;
		}

		running = true;
		thread = new Thread(this);
		thread.start();
```

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

        private void stop() {
                if (!running) {
                        return;
                }
                running = false;
                try {
                        thread.join();
                } catch (Exception e) {
                        e.printStackTrace();
                        System.exit(0);
                }
        }

        @Override
        public void run() {
                while(running) {
                        // play the game
                }

        }
        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...");

                game.start();
        }


}
```

```
6  public class Display extends Canvas implements Runnable{
```

The first change we made is to change our `Display` class to implement the `Runnable` interface. This will require that we add a `run()` method. In the `run()` method will be our game loop.

What is the Game Loop?

The key area of your game engine or game program is the game loop. This is the where your core game logic resides. It can be summarized as having the following structure:

```
while (!gameOver) {
        check for user Input
        run AI
        move enemies
        resolve collisions
        draw graphics
        play sounds
}
```

This loop repeats over and over again until the game is over by either the player winning or exiting the game. In this episode we introduce the beginning of the game loop:

Table 6 – The game loop in our program

```
public void run() {
        while(running) {
                // play the game
        }

}
```

It is currently missing all the key elements we need for a real game but we plan on adding that later - the start is there. The first thing that must be done is to set this up to run in its own thread…that is what this episode is about.

At this time you don't need to know much more about Threads and the code in this episode does not use much more. When you start the main method of the program:

```
public static void main(String[] args) {
        Display game = new Display();
        JFrame frame = new JFrame();
        frame.add(game);
                :
                :
        System.out.println("Running...");

        game.start();
}
```

In the `main()` method the call to the `game.start()` method invokes the **actual** `start()` method defined in the `Display` class (not some Thread's `run()` method). How do we know this? Well, the `Display` class does not extend the `Thread` class but implements the `Runnable` interface. The example shown above shows that when a class implements the `Runnable` interface that somewhere in

the code that class is sent to the `Thread` constructor where a thread object is created. In our code that is done in the `Display` `start()` method.

```
	private void start() {
		if (running) {
			return;
		}

		running = true;
		thread = new Thread(this);
		thread.start(); // here is where we invoke the Display's run() method
		System.out.println("Working");
	}
```

It is in the `start()` method where the variable `running` is set to true (it used to indicate that the game loop thread is running). The thread object is created (note how **`this`** is equal to the Display object) and NOW the `thread.start()` will invoke `Display`'s `run()` method.

```
	@Override
	public void run() {
		while(running) {
			// play the game
		}

	}
```

It is worth repeating: The `run()` command is the heart and soul of your game and runs in its own thread. There are other threads that you may not be aware of in this type of program - one would be the garbage collector thread, the event dispatcher for when the user clicks the mouse, presses a key, etc. and another thread is the one responsible for keeping the screen updated as users move the window, minimize it, overlap it with other windows, or it gets updated in the `run()` method as missiles explode and enemies disappear into the ether.

The author added a `stop()` method to this program but if you add a print statement to it you will see that it is never called. So one would regard it at this time as being unnecessary[4] and it could be easily removed without changing existing functionality.

I think the `stop()` method was added in the case that the program is executed as a Java Applet. As a Java Applet the program would be invoked externally (the applet container's `main()` method). The convention for Java Applets is that you implement the methods `init()` and `start()` that are invoked by the browser when starting your applet. The browser will also call your applet `stop()` method when your applet should stop execution. It makes sense to have the code we see in that method:

[4] It becomes useful when we create an applet version of the program in a future Episode but at this time Applets are not easy or even possible to run on most Internet browsers.

```
		if (!running) {
			return;
		}
		running = false;
		try {
			thread.join();
		} catch (Exception e) {
			e.printStackTrace();
			System.exit(0);
		}
```

If the game loop is not running then it must have reached the end of the game loop or in our case exited the `run()` method. If not, the `stop()` method sets running to false which will eventually put an end to the `run()` method game loop since that what determines staying in the loop:

```
	public void run() {
		while(running) {
			// play the game
		}

	}
```

The `thread.join();` call is used to block execution at that point until the thread has completed its run() method. So even after setting `running = false;` we have to wait until the thread executing the game loop is given time on the cpu to actually determine it should stop! So we wait for it.

One thing not mentioned but used in this program is that threads share global memory. In our case the various threads running in our program can access and change the `running` variable. This is one way in which they communicate with each other.

In summary, in this episode the game loop was created and set running in a thread of its own.

Additional Information

In the previous episode I presented a common technique to force the displayable game area to match our desired WIDTH and HEIGHT using the method below:

```
	public 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();
      }
```

The insets.left and insets.right refer to the border windows size in pixels on the left and right hand side of the window used to hold the canvas component. The top and bottom refer to the top border and bottom border respectively. The code then creates an anonymous class that implements the `Runnable` interface and creates the object `resize`.

The Event Dispatch Thread is a thread that polls the system event queue for events. Possible events are the mouse clicks, mouse movements, keyboard events, repaint requests. When an event occurs it is added to the event queue and handled by the Event Dispatch Thread.

Changing the state (in our case the size) of a Swing component outside the Event Dispatch Thread is considered unsafe when the component is "being shown and is visible." We must now change aspects of our application (e.g. drawing or changing window size) in sync[5] with the Event Dispatch Thread.

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

The above checks to see if we are currently running in the **Event Dispatch Thread**, if we are then we execute the `SwingUtilitied.invokeAndWait(resize)` which "causes the resize.run() to be executed synchronously on the AWT event dispatching thread. This call will block until all pending AWT events have been processed and then resize.run() returns. The `validate()` method at the end of our `resizeToInternalSize()` method will just lay out the components in the window.

[55] There are exceptions but we will not discuss that in this episode.