Lecture Sixteen – Threads and Animation

- Introduction
- Threads
- Animation
- AnimatedCar program
- AnimatedCar walkthrough
- Thread issues
- Timers
- Drill

Read Chapter 15
(only pages 170-179)
So far, we have written programs that have a single path of execution.

Writing code in this manner is definitely simple, but occasionally we are faced with situations when it is necessary to split that path in order to do two things at once.

We typically refer to this as multitasking.

For example, suppose we are writing software for a media player. We want our media player to play music, but at the same time we want to be able to get input from the user (like if they press the “Stop” button). This requires our program to multitask. It must play music AND be prepared for user input.
In order to perform several tasks at once, we use **Threads** (short for thread of execution).

You can think of **Threads** as being independent processes performing individual tasks that appear to be working in parallel (a split in our path of execution).

Many of the concepts we have already covered in this class (like listeners and event-driven programming) are actually internally implemented in Java using threading.

There are many issues that most programmers must deal with when using **Threads** (like the sharing of common resources), but those issues are beyond the scope of this class.
Animation

- Because **Threads** allow us to perform multiple tasks at the same time, they are a logical choice to use when creating animation.

- Typically, when we create animation for our programs we still want our program to operate normally – we don’t want our program to focus solely on making our animation work.

- So we create a **Thread** that has the task of maintaining our animation while our program continues to execute.
Animation

- Animation is typically created in one of the following ways:

  1. The illusion of animation is created by picture swapping. That is, replacing a picture with another picture (with slight modifications) quickly enough that it looks like the picture is animated (like a cartoon strip). The Java Juggler program in your book is an example of this method.

  2. An image is rapidly redrawn to the screen, but in a different orientation, location, or with other slight modifications. We will see an example of this in our next few slides.
AnimatedCar program

- The following program will show an example of how we can create animation by using the Graphics objects from Lesson 14 along with Threads.

- This program will simply draw a car to the screen (using basic shapes like rectangles and ellipses) and will contain buttons that can start and stop the animation.

- Our car will be animated to appear as though it is driving across the screen.
AnimatedCar program

- Here is a screenshot of what the program will look like once you get it working (isn’t that a beautiful car!)
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.lang.*;

public class AnimatedCar extends JFrame implements ActionListener, Runnable
{
    private JPanel buttonPanel;
    private JButton startButton;
    private JButton stopButton;
    private CarPanel car;
    Thread carThread;
}
public static void main(String[] args) {
    AnimatedCar c = new AnimatedCar();
    c.setVisible(true);
}

public AnimatedCar() {
    // application properties
    super("My Animated Car");
    setSize(400, 400);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// create components

car = new CarPanel();
buttonPanel = new JPanel();
startButton = new JButton("Start");
stopButton = new JButton("Stop");
buttonPanel.add(startButton);
buttonPanel.add(stopButton);

// create container and add components

Container window = getContentPane();
BorderLayout layout = new BorderLayout();
window.add(car, BorderLayout.CENTER);
window.add(buttonPanel, BorderLayout.SOUTH);
// add listeners to our buttons
startButton.addActionListener(this);
stopButton.addActionListener(this);
}

class CarPanel extends JPanel
{

private float changeX = 0f;
private final float carWidth = 200f;

public void paintComponent(Graphics g)
{
  // used to draw the car to the screen
  Graphics2D g2 = (Graphics2D) g;
g2.setColor(Color.red);
Rectangle2D.Float back = new Rectangle2D.Float(0f+changeX, 125f, carWidth/4, 50f);
g2.fill(back);
Rectangle2D.Float middle = new Rectangle2D.Float(50f+changeX, 75f, carWidth/2, 100f);
g2.fill(middle);
Rectangle2D.Float front = new Rectangle2D.Float(150f+changeX, 125f, carWidth/4, 50f);
g2.fill(front);
g2.setColor(Color.black);
Ellipse2D.Float backWheel = new Ellipse2D.Float(25f+changeX, 150f, 50f, 50f);
AnimatedCar code

g2.fill(backWheel);
Ellipse2D.Float frontWheel = new
Ellipse2D.Float(125f+changeX, 150f, 50f, 50f);
g2.fill(frontWheel);
}

// method used to move the car forward
public void moveForward()
{
    changeX = changeX + 5;
    // make sure the car hasn't gone past the
    // screen
    if (changeX > getWidth())
        changeX = -carWidth;
}
}
AnimatedCar code

// method used to start the thread
public void start()
{
    // if thread does not already exist then create and start it
    if (carThread == null) {
        carThread = new Thread(this);
        carThread.start();
    }
}

// method used to stop the thread
public void stop()
{
    if (carThread != null)
    {
        carThread = null;
    }
}
// method needed to implement Runnable
public void run()
{
    while (carThread == Thread.currentThread())
    {
        car.moveForward();
        repaint();
        try {
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            JOptionPane.showMessageDialog(null, e.getMessage());
        }
    }
}
// method needed to implement ActionListener
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == startButton) {
        start();
    } else if (e.getSource() == stopButton) {
        stop();
    }
}
AnimatedCar walkthrough

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.lang.*;

- Here we have most of the same imports that we typically need to create our GUI and the Graphics

- We added java.lang.* since it is needed for the Thread class and Runnable interface.
Private JPanel buttonPanel;
private JButton startButton;
private JButton stopButton;
private CarPanel car;
Thread carThread;

1. Next, we create the member variables for our class, including a Thread named carThread.

2. We will not go over main() and the constructor for AnimatedCar – both of these should be familiar from previous lessons.
AnimatedCar walkthrough

class CarPanel extends JPanel {
    private float changeX = 0f;
    private final float carWidth = 200f;

    Our **CarPanel** is the panel we want to be animated.

    We first create two variables – one will be used to update the change in X position (since our animation will consist of only moving horizontally) and the other is a constant used to denote the width of the entire car we are drawing.
public void paintComponent(Graphics g)
{
    // used to draw the car to the screen
    Graphics2D g2 = (Graphics2D) g;
    g2.setColor(Color.red);
    Rectangle2D.Float back = new Rectangle2D.Float(0f+changeX, 125f,
    carWidth/4, 50f);
    g2.fill(back);
}

- Next we create our `paintComponent()` method that will be called each time the panel is drawn or redrawn.

- Most of this method should be familiar from Lesson 14.

- Notice that when we specify the X coordinate for each shape that we translate each one by `changeX`. This ensures that all the shapes will be translated equally (which makes it appear that the entire car is being translated).
AnimatedCar walkthrough

// method used to move the car forward
public void moveForward()
{
    changeX = changeX + 5;
    // make sure the car hasn't gone past the screen
    if (changeX > getWidth())
        changeX = -carWidth;
}

- Now we will create a member method named moveForward which is simply used to update our changeX variable.

- We use our if statement to make sure our car has not driven off our panel. If it has then we change our translation variable to the negative size of the car’s width (so it looks like the car is driving back into the screen from the opposite side).

- One way we could make our car look like it is going faster (or slower) is to change the number of pixels we translate by in changeX. In this case, we will translate our car by 5 pixels everytime moveForward() is called.
public void start()
{
    // if thread does not already exist then create and start it
    if (carThread == null) {
        carThread = new Thread(this);
        carThread.start();
    }
}

- The `start()` method will be a member of `AnimatedCar` and is used to simply start our animation `Thread`.

- We first check to make sure the `Thread` does not already exist (because if we try to start a `Thread` that already exists then we will get an exception).

- If the `Thread` does not already exist then we create it by calling the constructor which will take `this` as a parameter (since our class implements `Runnable`) and will start the `Thread` by calling the `start()` method.
public void stop()
{
    if (carThread != null)
        carThread = null;
}

Likewise, the \textbf{stop()} method will be used to stop a \textit{Thread}.

In order to stop a \textit{Thread} from running we simply set it equal to \textbf{null}.

This may not seem very logical at first, especially if you consult the API for \textit{Thread} and notice that it has a \textbf{stop()} method. However, this method is deprecated and is no longer used. We will talk about this more later.
AnimatedCar walkthrough

```java
public void run() {
    while (carThread == Thread.currentThread()) {
        ...
    }
}
```

- It’s now time to write our `run()` method which will be called once our `Thread` is started.

- We continue to do our animation tasks while our `Thread` is equal to `Thread.currentThread()` (a method that returns the `Thread` currently being executed).

- Notice if we set our `Thread` equal to `null` (like we do in our `stop()` method) then this expression becomes false and animation stops.
AnimatedCar walkthrough

car.moveForward();
repaint();
try {
    Thread.sleep(100);
}
catch (InterruptedException e) {
    JOptionPane.showMessageDialog(null, e.getMessage);
}

To perform animation, we move our car forward and then call repaint() which will repaint our entire GUI (including the CarPanel).

We then call Thread.sleep() which causes the current Thread to pause. It takes an argument that specifies the length of that pause in milliseconds. This is another way we can make our animation appear faster or slower.

If the sleep command fails then an InterruptedException occurs, so a try/catch block is needed.
AnimatedCar walkthrough

// method needed to implement ActionListener
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == startButton) {
        start();
    }
    else if (e.getSource() == stopButton) {
        stop();
    }
}

- Finally we have our **actionPerformed** method which simply calls our **start()** and **stop()** methods depending on which button is selected.
Thread issues

- Recently, there has been some debate over the use of Threads in Java.

- Many of the Thread methods in Java have been declared deprecated and are no longer used because they are “inherently unsafe”.

- In addition, there are various scaling and performance issues that are encountered when multiple Threads are spawned.

- We will not get into these issues for this class, but you should be aware that they exist.
Timers

- Because of the issues surrounding Threads, many newer textbooks focus on using **Timers** to do animation.

- **Timer** is a class in the `javax.swing.*` package that utilizes **ActionListeners** to perform a task.

- You can find out more about how to use **Timers** by consulting the API (note: there are 3 **Timer** classes, be sure you look at the one in the `javax.swing.*` package) or by visiting this web tutorial.
  
  [http://java.sun.com/docs/books/tutorial/uiswing/misc/timer.html](http://java.sun.com/docs/books/tutorial/uiswing/misc/timer.html)

- You can also read this article on Java’s website which contains a good tutorial and also explains the benefits of a **Timer**.
  
Drill Sixteen

- Create an animated GUI much like AnimatedCar but that contains a single animated ball instead. Unlike the car though, the ball should move in both the X (horizontal) and Y (vertical) directions. Once you have that working try one of the following (or both!)
  1. Create a JSlider and add it to your button panel. The user should be able to use this slider to control the speed of animation.
  2. Make the ball a “bouncing ball”. That is, if the ball hits the side of the panel it will change direction instead of continuing past the panel. This one will require a bit more work and creativity. (HINT: It may be best to keep track of not only X and Y positions but also X and Y directions – although there are other ways of doing it)

- *BONUS* This one is not required but will give you some good practice. Try using Timers instead of Threads in your program.