LECTURE 15

ANIMATION & MULTI-THREADING

Slides by Rick Giles

Lecture Goals

- To learn about animated graphics
- Animation with Timer Events
- To understand how multiple threads can execute in parallel
- To learn to implement threads
- Animation with Threads
- To understand race conditions and deadlocks
- To avoid corruption of shared objects by using locks and conditions

Animation

- From the latin *anima* – to act, process, or impart life, interest, spirit, motion
- Sequence of pictures with small differences between them.
  - Use graphics to draw the pictures, or use a sequence of images.
- When shown in fairly rapid succession, it appears like motion.
  - Use events to signal the time for the next frame.

    or
  - Put the current thread to sleep to have some elapse time until showing the next frame

Ch. 11.5 Using Timer Events for Animation

- In this section, we will study timer events and use them to implement simple animations
- `javax.swing` provides a handy `Timer` class

A Swing timer notifies a listener with each “tick”.

javax.swing Timer class

- javax.swing.Timer
  - Can generate a series of events at even time intervals
  - Specify the frequency of the events and an object of a class that implements the ActionListener interface

```
class MyListener implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    // Listener action (executed at each timer event)
  }
}
MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

Timer events will begin after the start method is called

Animated Rectangle Example

```
 8  public class RectangleComponent extends JComponent
 9  {
10     private static final int RECTANGLE_WIDTH = 20;
11     private static final int RECTANGLE_HEIGHT = 30;
12
13     private int xLeft;
14     private int yTop;
15
16     public RectangleComponent()
17     {
18        xLeft = 0;
19        yTop = 0;
20     }
21
22     public void paintComponent(Graphics g)
23     {
24        g.fillRect(xLeft, yTop, RECTANGLE_WIDTH, RECTANGLE_HEIGHT);
25     }
```

```
32     public void moveRectangleBy(int dx, int dy)
33     {
34        xLeft = xLeft + dx;
35        yTop = yTop + dy;
36        repaint();
37     }
38  }
```

`repaint` method invokes the `paintComponent` method

RectangleFrame.java

```
 9  public class RectangleFrame extends JFrame
10  {
11     private static final int FRAME_WIDTH = 300;
12     private static final int FRAME_HEIGHT = 400;
13
14     private RectangleComponent scene;
15
16     class TimerListener implements ActionListener
17     {
18        public void actionPerformed(ActionEvent event)
19        {
20           scene.moveRectangleBy(1, 1);
21        }
22     }
23
24     public RectangleFrame()
25     {
26        scene = new RectangleComponent();
27        add(scene);
28
29        setSize(FRAME_WIDTH, FRAME_HEIGHT);
30
31        ActionListener listener = new TimerListener();
32
33        final int DELAY = 100; // Milliseconds between timer ticks
34        Timer t = new Timer(DELAY, listener);
35        t.start();
36     }
37  }
```

Inner class `TimerListener` moves the rectangle

`TimerListener` is called every 100ms by the `Timer` object

repaint and paintComponent

- When you make a change to the data, the component is not automatically painted with the new data
- You must call the `repaint` method of the component, either in the event handler or in the component's mutator methods
- Your component's `paintComponent` method will then be invoked with an appropriate `Graphics` object

Do not call the `paintComponent` method directly.

Bouncing Ball with Timer

- How many threads were used in the TimerAnimation program?
  - main thread
  - timer thread: the thread used by all swing timers to throw the actionEvent
  - event dispatch thread
- Is there conflict between the three threads?
- What if a time consuming task is done every tick?

  e.g., tick every 5 milliseconds, but task takes 10 milliseconds

Ch. 20.1 Running Threads

- **Thread:** a program unit that is executed independently of other parts of the program
- The Java Virtual Machine executes each thread in the program for a short amount of time
- This gives the impression of parallel execution

Running a Thread (1)

1. Implement a class that implements the `Runnable` interface:

```
public interface Runnable
{
   void run();
}
```

2. Place the code for your task into the `run` method of your class:

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      Task statements
      . . .
   }
}
```

Running a Thread (2)

3. Create an object of your subclass:

```
Runnable r = new MyRunnable();
```

4. Construct a `Thread` object from the `Runnable` object:

```
Thread t = new Thread(r);
```

5. Call the `start` method to start the thread:

```
t.start();
```

Running a Thread (3)

- A program to print a time stamp and “Hello World” once a second for ten seconds:

```
Fri Dec 28 23:12:03 PST 2012 Hello, World!
Fri Dec 28 23:12:04 PST 2012 Hello, World!
Fri Dec 28 23:12:05 PST 2012 Hello, World!
Fri Dec 28 23:12:06 PST 2012 Hello, World!
Fri Dec 28 23:12:07 PST 2012 Hello, World!
Fri Dec 28 23:12:08 PST 2012 Hello, World!
Fri Dec 28 23:12:09 PST 2012 Hello, World!
Fri Dec 28 23:12:10 PST 2012 Hello, World!
Fri Dec 28 23:12:11 PST 2012 Hello, World!
Fri Dec 28 23:12:12 PST 2012 Hello, World!
```

GreetingRunnable Class

```
public class GreetingRunnable implements Runnable
{
   private String greeting;

   public GreetingRunnable(String aGreeting)
   {
      greeting = aGreeting;
   }

   public void run()
   {
      Task statements
      . . .
   }
}
```

`run` Method (1)

- Loop 10 times through task actions:
  - Print a time stamp
  - Print the greeting
  - Wait a second

`run` Method (2)

- To get the date and time, construct a `Date` object:

```
Date now = new Date();
System.out.println(now + " " + greeting);
```

- To wait a second, use the sleep method of the `Thread` class:

```
Thread.sleep(milliseconds)
```

- Sleeping thread can generate an `InterruptedException`
  - Catch the exception
  - Terminate the thread

run Method (3)

```
public void run()
{
   try
   {
      Task statements
   }
   catch (InterruptedException exception)
   {
   }
   Clean up, if necessary
}
```

GreetingRunnable.java

```
1  import java.util.Date;
2
3  /**
4     A runnable that repeatedly prints a greeting.
5  */
6  public class GreetingRunnable implements Runnable
7  {
8     private static final int REPETITIONS = 10;
9     private static final int DELAY = 1000;
10
11    private String greeting;
12
13    /**
14       Constructs the runnable object.
15       @param aGreeting the greeting to display
16    */
17    public GreetingRunnable(String aGreeting)
18    {
19       greeting = aGreeting;
20    }
21
```

Continued

GreetingRunnable.java (cont.)

```
22      public void run()
23      {
24         try
25         {
26            for (int i = 1; i <= REPETITIONS; i++)
27            {
28               Date now = new Date();
29               System.out.println(now + " " + greeting);
30               Thread.sleep(DELAY);
31            }
32         }
33         catch (InterruptedException exception)
34         {
35         }
36      }
37   }
```

Start the Thread

- First construct an object of your `Runnable` class:

```
Runnable r = new GreetingRunnable("Hello World");
```

- Then construct a `Thread` and call its `start` method:

```
Thread t = new Thread(r);
t.start();
```

GreetingThreadRunner.java

```
1  /**
2     This program runs two greeting threads in parallel.
3  */
4  public class GreetingThreadRunner
5  {
6     public static void main(String[] args)
7     {
8        GreetingRunnable r1 = new GreetingRunnable("Hello, World!");
9        GreetingRunnable r2 = new GreetingRunnable("Goodbye, World!");
10       Thread t1 = new Thread(r1);
11       Thread t2 = new Thread(r2);
12       t1.start();
13       t2.start();
14    }
15 }
```

Continued

GreetingThreadRunner.java (cont.)

Program Run

```
Fri Dec 28 12:04:46 PST 2012 Hello, World!
Fri Dec 28 12:04:46 PST 2012 Goodbye, World!
Fri Dec 28 12:04:47 PST 2012 Hello, World!
Fri Dec 28 12:04:47 PST 2012 Goodbye, World!
Fri Dec 28 12:04:48 PST 2012 Hello, World!
Fri Dec 28 12:04:48 PST 2012 Goodbye, World!
Fri Dec 28 12:04:49 PST 2012 Hello, World!
Fri Dec 28 12:04:49 PST 2012 Goodbye, World!
Fri Dec 28 12:04:50 PST 2012 Hello, World!
Fri Dec 28 12:04:50 PST 2012 Goodbye, World!
Fri Dec 28 12:04:51 PST 2012 Hello, World!
Fri Dec 28 12:04:51 PST 2012 Goodbye, World!
Fri Dec 28 12:04:52 PST 2012 Goodbye, World!
Fri Dec 28 12:04:52 PST 2012 Hello, World!
Fri Dec 28 12:04:53 PST 2012 Hello, World!
Fri Dec 28 12:04:53 PST 2012 Goodbye, World!
Fri Dec 28 12:04:54 PST 2012 Hello, World!
Fri Dec 28 12:04:54 PST 2012 Goodbye, World!
Fri Dec 28 12:04:55 PST 2012 Hello, World!
Fri Dec 28 12:04:55 PST 2012 Goodbye, World!
```

Thread Scheduler

- **Thread scheduler:** runs each thread for a short amount of time (a **time slice**)
- Then the scheduler activates another thread
- There will always be slight variations in running times – especially when calling operating system services (e.g. input and output)
- There is no guarantee about the order in which threads are executed

20.2 Terminating Threads

- A thread terminates when its `run` method terminates
- Do not terminate a thread using the deprecated `stop` method
- Instead, notify a thread that it should terminate:

```
t.interrupt();
```

- `interrupt` does not cause the thread to terminate – it sets a `boolean` variable in the thread data structure

Terminating Threads (2)

- The `run` method should check occasionally whether it has been interrupted:
  - Use the `interrupted` method
  - An interrupted thread should release resources, clean up, and exit:

```
public void run()
{
   for (int i = 1;
        i <= REPETITIONS && !Thread.interrupted(); i++)
   {
      Do work
   }
   Clean up
}
```

Terminating Threads (3)

- The `sleep` method throws an `InterruptedException` when a sleeping thread is interrupted:
  - Catch the exception and terminate the thread:

```
public void run()
{
   try
   {
      for (int i = 1;
           i <= REPETITIONS i++;
      {
         Do work
         Sleep
      }
   }
   catch (InterruptedException exception)
   {
   }
   Clean up
}
```

Terminating Threads (4)

- Java does not force a thread to terminate when it is interrupted
- It is entirely up to the thread what it does when it is interrupted
- Interrupting is a general mechanism for getting the thread's attention

Bouncing Ball with Threads

- How many threads are used in the ThreadAnimation program?
  - main thread
  - event dispatch thread
  - new thread

- Is there conflict between the three threads?

  main

  event dispatch thread

  new thread

  updates the ball position

  issues repaint( ) instructions

  handled by placing an event in the event queue

Bouncing Ball with Mouse Click

- Extend the application by having a mouse click on the panel to move the ball to that location.
  - The mouse click generates a MouseEvent.
  - Need a MouseListener class with mousePressed( ) method.
  - What is the event source? the panel
  - Hence, need to add the MouseListener to the panel.
  - The event stores the location of the click in the panel.

Bouncing Ball with Mouse Clicks

- Now, how many threads are used in the ThreadAnimation program?
  - main thread
  - event dispatch thread
  - new thread

Is there conflict between the three threads?

main

event dispatch thread

handles mouse clicks

updates the ball position

issues repaint( ) instructions

new thread

updates the ball position

issues repaint( ) instructions

Problem:
•Two threads updating and accessing the same data.
•This is called a **Race Condition**: the results depend upon the scheduling of the threads.

Race Condition

- Suppose thread 1 is executing

```
x = 1;
y = 1;
a[x][y] = 1;
```

- Suppose thread 2 is executing

```
x = 2;
y = 2;
a[x][y] = 2;
```

Race Condition

- If one executes to completion before the other is started, there is no problem. However, suppose that the execution proceeds as follows:

Thread 1	Thread 2
x = 1;	
	x = 2;
	y = 2;
	a[x][y] = 2;
y = 1;	
a[x][y] = 1;	

The result is a[2][2] is set to 2, and a[2][1] is set to 1.

The race condition resulted in the wrong location being set.

Handling Race Conditions

- Prevent them by allowing only one thread to access the data at a time.
- When one finishes, the other can have access.
- This is called **synchronizing**.
- Access is controlled by having a lock.
  - When one thread has the lock for a resource, no other thread can access the resource until the thread holding the lock has released the lock.
- Java has two independent ways to handle locks:
  - Special lock objects
  - A built-in lock in every object

Special lock objects

```
private Lock ballAccessLock = new ReentrantLock();

…

ballAccessLock.lock();      // grab the lock

            …     // code to access and manipulate the shared resource

ballAccessLock.unlock();  // return the lock
```

The thread that grabbed a lock is said to own the lock.

If a thread A attempts to grab a lock, but it is currently owned by another thread,

then thread A is deactivated. From time-to-time, thread A is reactivated to try the lock again.

It is necessary to guarantee that any thread that grabs a lock eventually releases the lock, even if an exception occurs. The later situation is handed by using the **finally** clause of a try-catch.

synchronized

- Every object has a lock built into it.
- Two approaches to using this lock:
  1. Use the synchronized statement on the object.
  2. Place the synchronized modifier on a method.

synchronized (1)

- Use the synchronized statement on the object.

```
synchronized(ball)
{
        // the lock of the “ball” object is owned by this thread as long as
        //      the statements in the block are being executed
        // the lock is automatically released when the block is exited,
        //      whether normally or abnormally
        // no other thread can obtain the lock for this object
        //      when it is owned by this thread, but unsynchronized access is
        //      allowed by Java (the programmer should preclude this)
}
```

synchronized (2)

- Place the synchronized modifier on a method.

```
public synchronized void moveBallAndRepaint(Double x, Double y)
{
        // the lock for “this” object is held for the duration of the method
}
```

Note that this would obtain the lock of the panel, not the ball.

Bouncing Ball with Synchronization

```
public void shiftBallAndRepaint()
{
        synchronized(ball)
        {
                super.shiftBallAndRepaint();
        }
}

public void moveBallAndRepaint(Double x, Double y)
{
        Double newX;
                … (as before)
        Double newY;
                … (as before)
        synchronized(ball)
        {
                ball.setFrame(newX, newY, ball.getWidth(), ball.getHeight());
        }
        repaint();
}
```

Other Java tools for Threading

- Thread Thread.currentThread() // static function
  // returns the current thread

- t.interrupt( ); // sets a flag in the thread to true to indicate
  // an interrupt request
  // if the thread is waiting or sleeping
  // an InterruptedException is thrown

- boolean t.isInterrupted( ) // checks the interrupt flag for thread t

- boolean Thread.interrupted( ) // static method that checks the
  // interrupt flag for the current thread
  // sets the flag to false

Other Java tools for Threading

- Class Object has the method wait( ) that causes the current thread to wait (in this object) until another thread has issued a notify() or notifyAll() for the object.
- Class Object has methods notify( ) and notifyAll( ) to wake up one other (or all other) threads in the object specified as the target.
- A ReentrantLock can be temporarily released :
  - Suppose that there is a need to acquire a resource in a certain state
  - Acquire the resource lock
  - If not in the acceptable state
    - release the lock and enter a special “await” state
    - remains in the “await” state until some other thread issues a “signal” or a “signalAll”
    - try again

Review: Running Threads

- A thread is a program unit that is executed concurrently with other parts of the program.
- The `start` method of the `Thread` class starts a new thread that executes the `run` method of the associated `Runnable` object.
- The `sleep` method puts the current thread to sleep for a given number of milliseconds.
- When a thread is interrupted, the most common response is to terminate the `run` method.
- The thread scheduler runs each thread for a short amount of time, called a time slice.

Review: Terminating Threads

- A thread terminates when its run method terminates.
- The `run` method can check whether its thread has been interrupted by calling the `interrupted` method.

Review: Race Conditions

- A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

Review: Synchronizing Object Access

- By calling the `lock` method, a thread acquires a `Lock` object. Then no other thread can acquire the lock until the first thread releases the lock.

Review: Avoiding Deadlocks

- A deadlock occurs if no thread can proceed because each thread is waiting for another to do some work first.
- Calling `await` on a condition object makes the current thread wait and allows another thread to acquire the lock object.
- A waiting thread is blocked until another thread calls `signalAll` or `signal` on the condition object for which the thread is waiting.