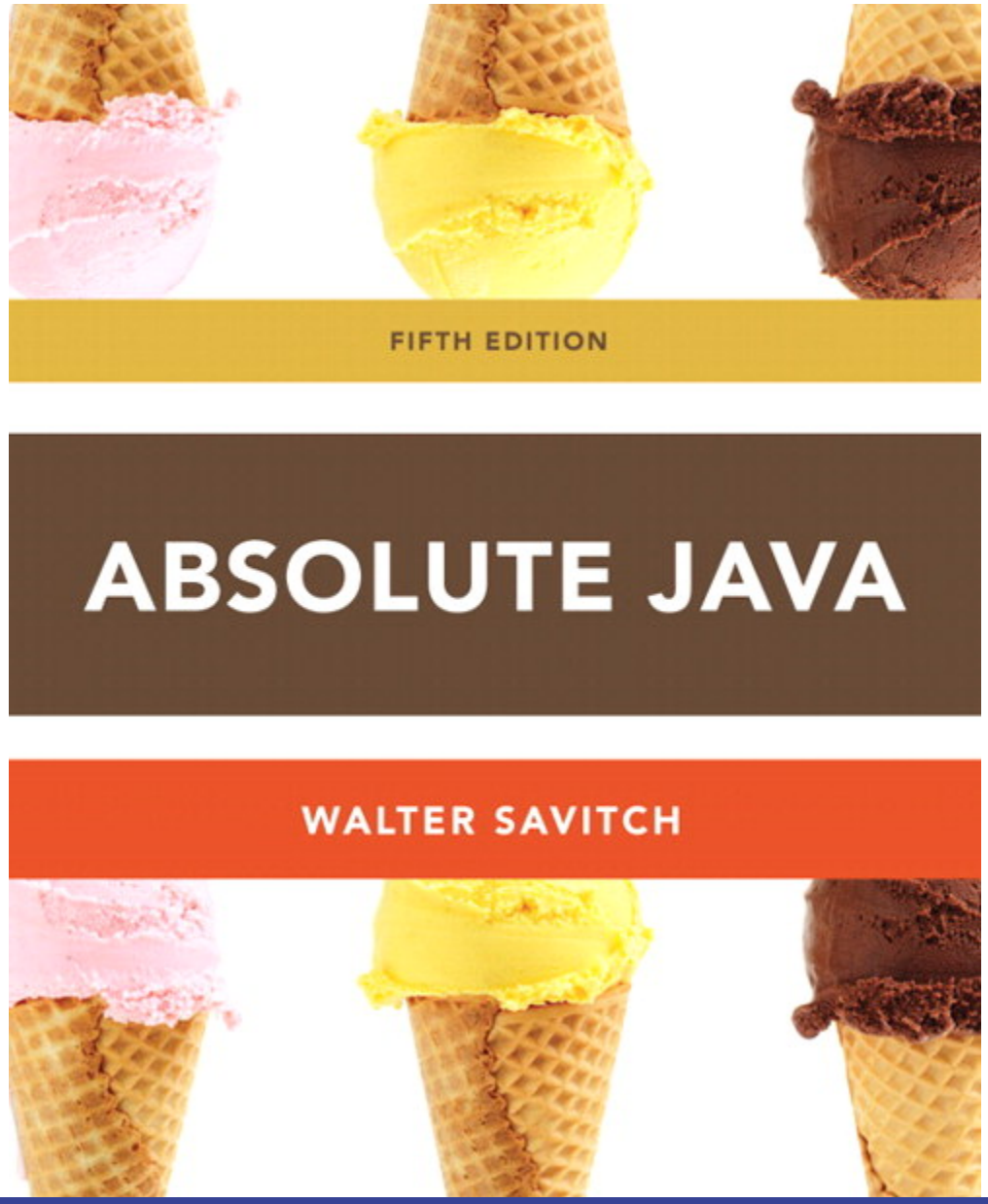


Chapter 19

Java Never Ends

Slides prepared by Rose Williams, *Binghamton University*

Kenrick Mock, *University of Alaska Anchorage*

Multithreading

- In Java, programs can have multiple threads
  - A *thread* is a separate computation process
- Threads are often thought of as computations that run in parallel
  - Although they usually do not really execute in parallel
  - Instead, the computer switches resources between threads so that each one does a little bit of computing in turn
- Modern operating systems allow more than one program to run at the same time
  - An operating system uses threads to do this

`Thread.sleep`

- **`Thread.sleep`** is a static method in the class **`Thread`** that pauses the thread that includes the invocation
  - It pauses for the number of milliseconds given as an argument
  - Note that it may be invoked in an ordinary program to insert a pause in the single thread of that program
- It may throw a checked exception, **`InterruptedException`**, which must be caught or declared
  - Both the **`Thread`** and **`InterruptedException`** classes are in the package **`java.lang`**

The `getGraphics` Method

- The method `getGraphics` is an accessor method that returns the associated `Graphics` object of its calling object
  - Every `JComponent` has an associated `Graphics` object

```
Component.getGraphics();
```

A Nonresponsive GUI

- The following program contains a simple GUI that draws circles one after the other when the "Start" button is clicked
  - There is a 1/10 of a second pause between drawing each circle
- If the close-window button is clicked, nothing happens until the program is finished drawing all its circles
- Note the use of the **`Thread.sleep`** (in the method **`doNothing`**) and **`getGraphics`** (in the method **`fill`**) methods

Nonresponsive GUI (Part 1 of 9)

Nonresponsive GUI

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

(continued)

Nonresponsive GUI (Part 2 of 9)

Nonresponsive GUI

```
/**
Packs a section of the frame window with circles, one at a time.
*/
public class FillDemo extends JFrame implements ActionListener
{
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;
    public static final int FILL_WIDTH = 300;
    public static final int FILL_HEIGHT = 100;
    public static final int CIRCLE_SIZE = 10;
    public static final int PAUSE = 100; //milliseconds


    private JPanel box;
```

(continued)

Nonresponsive GUI (Part 3 of 9)

Nonresponsive GUI

```
    public static void main(String[] args)
    {
        FillDemo gui = new FillDemo();
        gui.setVisible(true);
    }


    public FillDemo()
    {
        setSize(WIDTH, HEIGHT);
        setTitle("FillDemo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


       setLayout(new BorderLayout());
```

(continued)

Nonresponsive GUI (Part 4 of 9)

Nonresponsive GUI

```
        box = new JPanel();
        add(box, "Center");


        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout());
        JButton startButton = new JButton("Start");
        startButton.addActionListener(this);
        buttonPanel.add(startButton);
        add(buttonPanel, "South");
    }
```

(continued)

Nonresponsive GUI (Part 5 of 9)

Nonresponsive GUI

```
    public void actionPerformed(ActionEvent e)
    {
        fill();
    }

    public void fill()
    {
        Graphics g = box.getGraphics();

        for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
            for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
```

Nothing else can happen until `actionPerformed` *returns, which does not happen until* `fill` *returns.*

(continued)

Nonresponsive GUI (Part 6 of 9)

Nonresponsive GUI

```
            {
                g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
                doNothing(PAUSE);
            }
        }

        public void doNothing(int milliseconds)
        {
            try
            {
                Thread.sleep(milliseconds);
            }
            catch(InterruptedException e)
            {
                System.out.println("Unexpected interrupt");
                System.exit(0);
            }
        }
    }
```

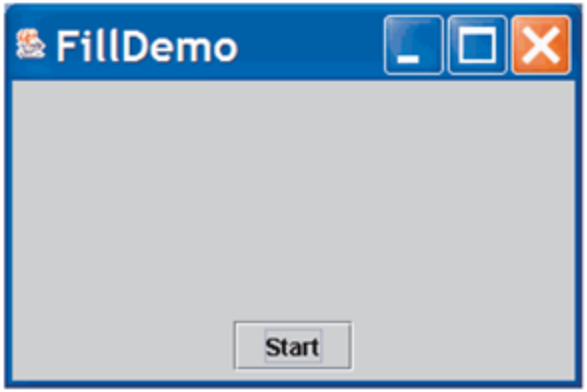
Everything stops for 100 milliseconds (1/10 of a second).

(continued)

Nonresponsive GUI (Part 7 of 9)

Nonresponsive GUI

RESULTING GUI (When started)

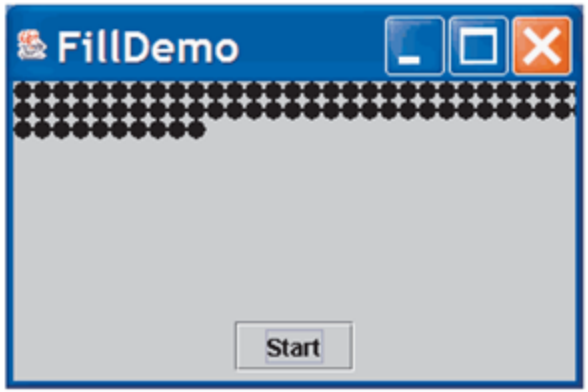


(continued)

Nonresponsive GUI (Part 8 of 9)

Nonresponsive GUI

RESULTING GUI (While drawing circles)



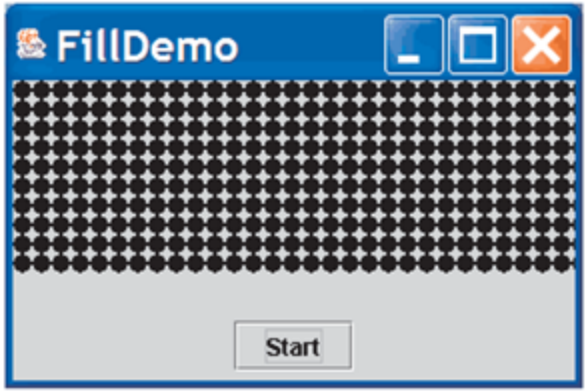
If you click the close-window button while the circles are being drawn, the window will not close until all the circles are drawn.

(continued)

Nonresponsive GUI (Part 9 of 9)

Nonresponsive GUI

RESULTING GUI (After all circles are drawn)



Fixing a Nonresponsive Program Using Threads

- This is why the close-window button does not respond immediately:
  - Because the method `fill` is invoked in the body of the method `actionPerformed`, the method `actionPerformed` does not end until after the method `fill` ends
  - Therefore, the method `actionPerformed` does not end until after the method `fill` ends
  - Until the method `actionPerformed` ends, the GUI cannot respond to anything else

Fixing a Nonresponsive Program Using Threads

- This is how to fix the problem:
  - Have the `actionPerformed` method create a new (independent) thread to draw the circles
  - Once created, the new thread will be an independent process that proceeds on its own
  - Now, the work of the `actionPerformed` method is ended, and the main thread (containing `actionPerformed`) is ready to respond to something else
  - If the close-window button is clicked while the new thread draws the circles, then the program will end

The Class `Thread`

- In Java, a thread is an object of the class **`Thread`**
- Usually, a derived class of **`Thread`** is used to program a thread
  - The methods **`run`** and **`start`** are inherited from **`Thread`**
  - The derived class overrides the method **`run`** to program the thread
  - The method **`start`** initiates the thread processing and invokes the **`run`** method

A Multithreaded Program that Fixes a Nonresponsive GUI

- The following program uses a main thread and a second thread to fix the nonresponsive GUI
  - It creates an inner class **`Packer`** that is a derived class of **`Thread`**
  - The method **`run`** is defined in the same way as the previous method **`fill`**
  - Instead of invoking **`fill`**, the **`actionPerformed`** method now creates an instance of **`Packer`**, a new independent thread named **`packerThread`**
  - The **`packerThread`** object then invokes its **`start`** method
  - The **`start`** method initiates processing and invokes **`run`**

Threaded Version of **`FillDemo`** (Part 1 of 6)

Threaded Version of `FillDemo`

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

(continued)

The GUI produced is identical to the GUI produced by Display 19.1 except that in this version the close window button works even while the circles are being drawn, so you can end the GUI early if you get bored.

Threaded Version of **`FillDemo`** (Part 2 of 6)

Threaded Version of `FillDemo`

```
public class ThreadedFillDemo extends JFrame implements ActionListener
{
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;
    public static final int FILL_WIDTH = 300;
    public static final int FILL_HEIGHT = 100;
    public static final int CIRCLE_SIZE = 10;
    public static final int PAUSE = 100; //milliseconds

    private JPanel box;

    public static void main(String[] args)
    {
        ThreadedFillDemo gui = new ThreadedFillDemo();
        gui.setVisible(true);
    }
```

(continued)

Threaded Version of **`FillDemo`** (Part 3 of 6)

Threaded Version of `FillDemo`

```
    public ThreadedFillDemo()
    {
        setSize(WIDTH, HEIGHT);
        setTitle("Threaded Fill Demo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new BorderLayout());

        box = new JPanel();
        add(box, "Center");

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout());
```

(continued)

Threaded Version of **`FillDemo`** (Part 4 of 6)

Threaded Version of `FillDemo`

```
        JButton startButton = new JButton("Start");
        startButton.addActionListener(this);
        buttonPanel.add(startButton);
        add(buttonPanel, "South");
    }

    public void actionPerformed(ActionEvent e)
    {
            Packer packerThread = new Packer();
            packerThread.start();
    }

    private class Packer extends Thread
```

You need a thread object, even if there are no instance variables in the class definition of `Packer`.

`start` *"starts" the thread and calls* `run`.

(continued)

Threaded Version of **`FillDemo`** (Part 5 of 6)

Threaded Version of `FillDemo`

```
    {
        public void run()
        {
            Graphics g = box.getGraphics();
            for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
              for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
              {
                  g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
                  doNothing(PAUSE);
              }
        }
```

run is inherited from Thread but needs to be overridden.

(continued)

Threaded Version of **`FillDemo`** (Part 6 of 6)

Threaded Version of `FillDemo`

```
        public void doNothing(int milliseconds)
        {
            try
            {
                Thread.sleep(milliseconds);
            }
            catch(InterruptedException e)
            {
                System.out.println("Unexpected interrupt");
                System.exit(0);
            }
        }
    } //End Packer inner class

}
```

The `Runnable` Interface

- Another way to create a thread is to have a class implement the `Runnable` interface
  - The `Runnable` interface has one method heading:

```
public void run();
```

- A class that implements `Runnable` must still be run from an instance of `Thread`
  - This is usually done by passing the `Runnable` object as an argument to the thread constructor

The **Runnable** Interface: Suggested Implementation Outline

```
public class ClassToRun extends SomeClass implements
   Runnable
{ . . .
  public void run()
  {
    // Fill this as if ClassToRun
    // were derived from Thread
  }
  . . .
  public void startThread()
  {
    Thread theThread = new Thread(this);
    theThread.run();
  }
  . . .
}
```

The Runnable Interface (Part 1 of 5)

The Runnable Interface

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ThreadedFillDemo2 extends JFrame
                               implements ActionListener, Runnable
{
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;
    public static final int FILL_WIDTH = 300;
    public static final int FILL_HEIGHT = 100;
    public static final int CIRCLE_SIZE = 10;
    public static final int PAUSE = 100; //milliseconds
```

(continued)

The Runnable Interface (Part 2 of 5)

The Runnable Interface

```
    private JPanel box;

    public static void main(String[] args)
    {
        ThreadedFillDemo2 gui = new ThreadedFillDemo2();
        gui.setVisible(true);
    }

    public ThreadedFillDemo2()
    {
        setSize(WIDTH, HEIGHT);
        setTitle("Threaded Fill Demo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new BorderLayout());
```

(continued)

The Runnable Interface (Part 3 of 5)

The Runnable Interface

```
        box = new JPanel();
        add(box, "Center");

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout());

        JButton startButton = new JButton("Start");
        startButton.addActionListener(this);
        buttonPanel.add(startButton);
        add(buttonPanel, "South");
    }
```

(continued)

The Runnable Interface (Part 4 of 5)

The Runnable Interface

```
    public void actionPerformed(ActionEvent e)
    {
        startThread();
    }

    public void run()
    {
        Graphics g = box.getGraphics();
        for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
          for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
          {
              g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
              doNothing(PAUSE);
          }
    }
```

(continued)

The Runnable Interface (Part 5 of 5)

The Runnable Interface

```
    public void startThread()
    {
        Thread theThread = new Thread(this);
        theThread.start();
    }

    public void doNothing(int milliseconds)
    {
        try
        {
            Thread.sleep(milliseconds);
        }
        catch(InterruptedException e)
        {
            System.out.println("Unexpected interrupt");
            System.exit(0);
        }
    }
}
```

Race Conditions

- When multiple threads change a shared variable it is sometimes possible that the variable will end up with the wrong (and often unpredictable) value.
- This is called a race condition because the final value depends on the sequence in which the threads access the shared value.
- We will use the Counter class to demonstrate a race condition.

Counter Class

Display 19.4 The **Counter** Class

```
public class Counter
{
  private int counter;
  public Counter()
  {
        counter = 0;
  }
  public int value()
  {
        return counter;
  }
  public void increment()
  {
        int local;
        local = counter;
        local++;
        counter = local;
  }
}
```

Race Condition Example

1. Create a single instance of the Counter class.
2. Create an array of many threads (30,000 in the example) where each thread references the single instance of the Counter class.
3. Each thread runs and invokes the increment() method.
4. Wait for each thread to finish and then output the value of the counter. If there were no race conditions then its value should be 30,000. If there were race conditions then the value will be less than 30,000.

Race Condition Test Class (1 of 3)

Display 19.5 The **RaceConditionTest** Class

```
public class RaceConditionTest extends Thread
{
  private Counter countObject;

  public RaceConditionTest(Counter ctr)
  {
       countObject = ctr;
  }
```

Stores a reference to a single **Counter** *object.*

Race Condition Test Class (2 of 3)

```
public void run()
{
  countObject.increment();
}

public static void main(String[] args)
{
  int i;
  Counter masterCounter = new Counter();
  RaceConditionTest[] threads = new RaceConditionTest[30000];

  System.out.println("The counter is " + masterCounter.value());
  for (i = 0; i < threads.length; i++)
  {
        threads[i] = new RaceConditionTest(masterCounter);
        threads[i].start();
  }
```

Invokes the code in Display 19.4 where the race condition occurs.

The single instance of the `Counter` *object.*

Array of 30,000 threads.

Give each thread a reference to the single `Counter` *object and start each thread.*

Race Condition Test Class (3 of 3)

```
        // Wait for the threads to finish
        for (i = 0; i < threads.length; i++)
        {
            try
            {
              threads[i].join();
            }
            catch (InterruptedException e)
            {
              System.out.println(e.getMessage());
            }
        }
        System.out.println("The counter is " + masterCounter.value());
    }
}
```

Waits for the thread to complete.

Sample Dialogue (output will vary)

```
The counter is 0
The counter is 29998
```

Thread Synchronization

- The solution is to make each thread wait so only one thread can run the code in increment() at a time.
- This section of code is called a **critical region** . Java allows you to add the keyword **synchronized** around a critical region to enforce that only one thread can run this code at a time.

Synchronized

- Two solutions:

```
public synchronized void increment()
{
  int local;
  local = counter;
  local++;
  counter = local;
}
```

```
public void increment()
{
  int local;
  synchronized (this)
  {
   local = counter;
   local++;
   counter = local;
  }
}
```