Advanced Programming

Lecture 6

Thread Programming in Java

What Is a Thread?

All programmers are familiar with writing sequential programs.

You have written a program that displays "Hello World!", sorts a list of names, or computes Fibonacci numbers.

These are sequential programs:

- each has a beginning,
- an execution sequence, and
- an end.

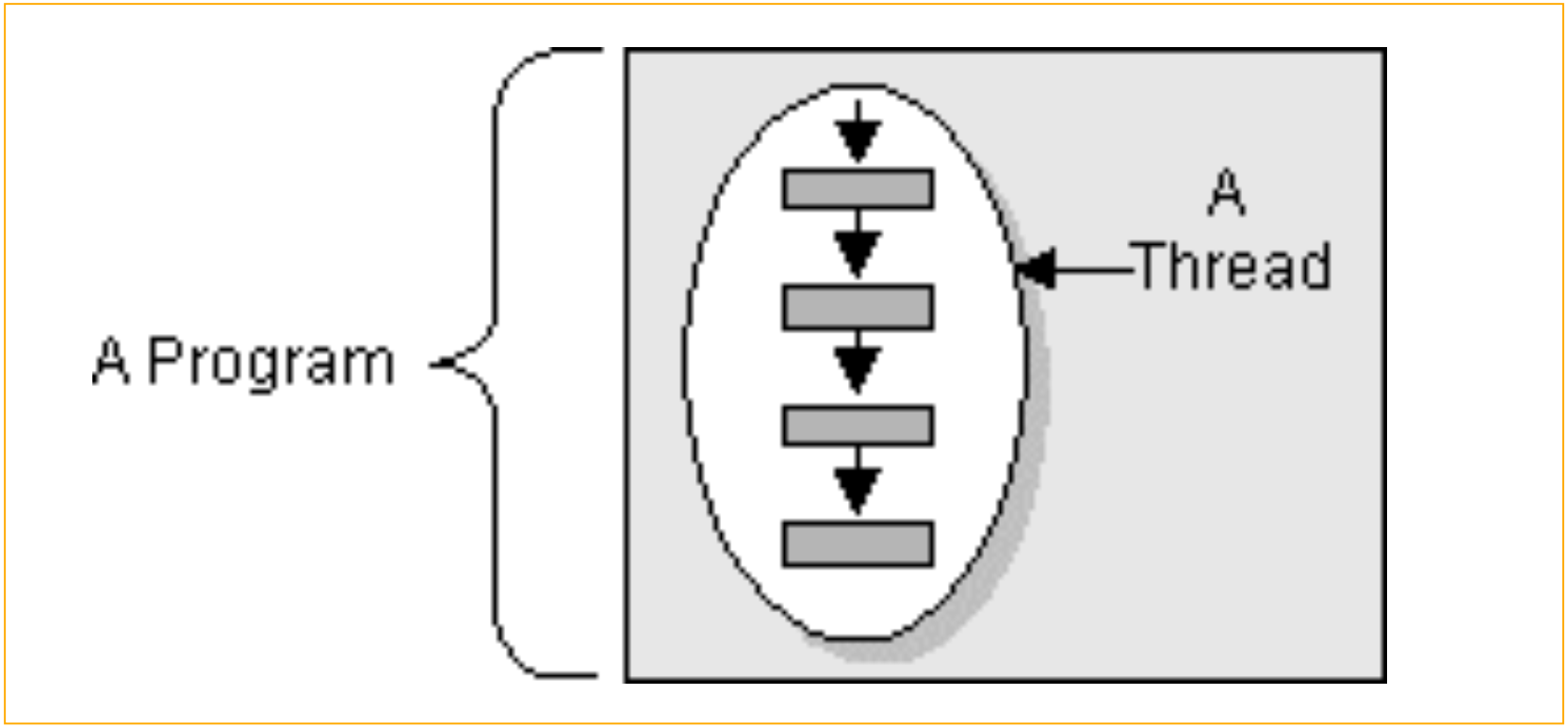
At any given time during the runtime of the program there is a single point of execution.

A thread is similar to the sequential programs described previously.

A **single thread** also has a beginning, a sequence, and an end and at any given time during the runtime of the thread, there is a single point of execution.

However, a thread itself is not a program; it cannot run on its own. Rather, a thread runs within a program.

The following figure shows this relationship.



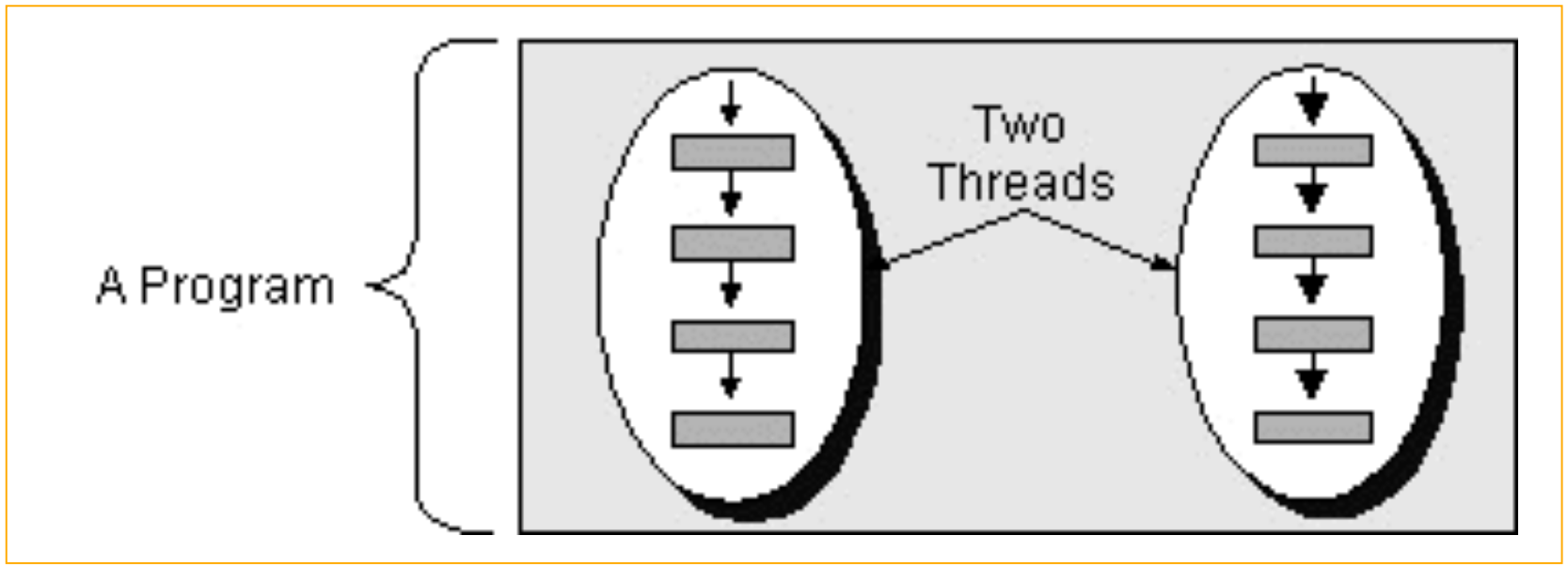
Definition: A thread is a single sequential flow of control within a program.

There is nothing new in the concept of a single thread.

The real interest and excitement surrounding threads is not about a single sequential thread.

It is about the use of multiple threads in a single program, running at the same time and performing different tasks.

This is illustrated by the following figure:



Multitasking is the operation of several programs running simultaneously on a single computer

- Each executable has its own memory space
- There is a moderate level of overhead associated with context switching between processes

Multithreading is the operation of several execution paths running simultaneously within a single application

- Threads can access the same application memory
- There is a lower level of overhead associated with context switching between threads

Every applet or application has at least one thread

When used appropriately, the creation of additional threads can provide efficiency and simplicity to otherwise complex tasks.

Lightweight Process = Thread

Some texts use the name lightweight process instead of thread.

A thread is similar to a real process in that a thread and a running program are both a single sequential flow of control.

However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

As a sequential flow of control, a thread must carve out some of its own resources within a running program. (It must have its own execution stack and program counter.)

The code running within the thread works only within that context.

Therefore, some other texts use execution context as a synonym for thread.

Customizing a Thread's run Method

The **run method** gives a thread something to do.

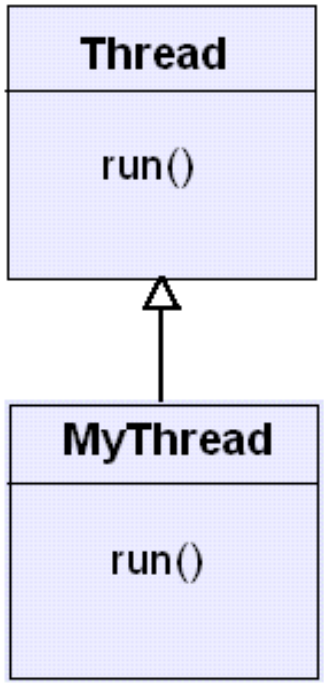
Its code implements the thread's running behavior.

It can do anything that can be encoded in Java statements:

- compute a list of prime's,
- sort some data,
- perform some animation.

The **Thread class** implements a generic thread that, by default, does nothing.

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.



The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

```
class MyThread extends Thread {
    public void run() {
        //......
    }
}
```

The **Thread class** implements a generic thread that, by default, does nothing.

That is, the implementation of its run method is empty. This is not particularly useful, so the Thread class defines an API that lets a **Runnable object** provide a more interesting run method for a thread.

There are two techniques for providing a run method for a thread:

Subclassing Thread and Overriding run

Implementing the Runnable Interface

We will now look at each of these…..

Subclassing Thread and Overriding run

The first way to customize what a thread does when it is running is to subclass **Thread** (itself a **Runnable** object) and override its empty **run method** so that it does something.

Let's look at the **SimpleThread** class, the first of twc classes in this example, which does just that:

```
public class SimpleThread extends Thread
{
    public SimpleThread (String str)
    {
        super(str);
    }
        public void run()
        {
            for (int i = 0; i < 10; i++)
            {
                System.out.println(i + " " + getName());

                try
                {
                    sleep((long)(Math.random() * 1000));
                }
                catch (InterruptedException e) {}
            }
            System.out.println("DONE! " + getName());
        }
}
```

The first method in the **SimpleThread** class is a constructor that takes a **String** as its only argument.

This constructor is implemented by calling a **superclass** constructor and is interesting to us only because it sets the Thread's name, which is used later in the program.

The next method in the **SimpleThread** class is the **run** method.

The first method in the **SimpleThread** class is a constructor that takes a **String** as its only argument.

This constructor is implemented by calling a **superclass** constructor and is interesting to us only because it sets the Thread's name, which is used later in the program.

The next method in the **SimpleThread** class is the **run** method.

The **run** method is the heart of any Thread and where the action of the Thread takes place.

The **run** method of the **SimpleThread** class contains a **for** loop that iterates ten times.

In each iteration, the method displays the iteration number and the name of the Thread, then sleeps for a random interval of up to 1 second.

After the loop has finished, the **run** method prints “DONE!” along with the name of the thread.

That's it for the **SimpleThread** class.

The **TwoThreadsTest** class provides a main method that creates two **SimpleThread** threads: one is named "Jamaica" and the other is named "Fiji".

```
public class TwoThreadsTest
{
    public static void main (String[] args)
    {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

The **main** method also starts each thread immediately following its construction by calling the **start** method.

Compile and run the program. You should see output similar to the following:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Jamaica
3 Fiji
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Jamaica
6 Fiji
7 Fiji
8 Fiji
7 Jamaica
8 Jamaica
9 Fiji
9 Jamaica
DONE! Jamaica
DONE! Fiji
Press any key to continue . . .
```

Notice how the output from each thread is intermingled with the output from the other.

This is because both **SimpleThread** threads are running concurrently.

Both run methods are running at the same time and each thread is displaying its output at the same time as the other.

Enlivening a Thread : using start() and isAlive()

- The start() method signals a Thread is ready for execution (however it is up to the scheduler to allocate CPU time for thread execution)
- Once a start() call is invoked the new thread is considered to come alive
- The thread remains alive until the run() method completes or the thread meets an untimely ‘death’
- The isAlive() method can be used to determine whether the Thread is started and still running

Calling stop on a Thread

- A thread will die when the stop() method is called
- The stop() method however has been deprecated since JDK 1.2 (i.e. the method can be used but you'll get a warning!!)
- The stop method was deprecated due the possibility of corrupting data in objects
- For instance the stop() method may not allow clean up to properly take place since the run operation does not go to completion
- Threads with stop() called also release locks on resources, they were most likely holding the lock for a good reason
- There are safe ways to stop threads, however this goes beyond an introduction to thread programming (one method involves the use of a stopRequested variable and the interupt method)

Finding the name of the current executing Thread

- Use the method Thread.currentThread()
- This will return a reference to the current thread in execution
- You can retrieve/change information about the Thread retrieved, e.g.
    - getName()
    - setName()

Putting a Thread to bed! (sleep())

- Sometimes it's not in the interest of the programmer to have a thread in execution all of the time
- This would cause the thread to use up resources that could be given to other threads
- For example a clock timer may only want to update the minutes display every sixty seconds, e.g.

```
try {
   Thread.sleep(60000);
}
catch (InterruptedException ex) {
   //ignore exception
}
```

Puts the thread to bed for 60 seconds

- The sleep method is static, i.e. one thread cannot call sleep on another, only the current thread will be put to sleep using the sleep method

Why catch InterruptedException?

```
try {
   Thread.sleep(60000);
}
catch (InterruptedException ex) {
   //ignore exception
}
```

- While a thread is sleeping it may be interrupted
- The interruption may be caused by another thread notifying the thread that a resource is now available
- When a thread is interrupted for whatever reason it throws an InterruptedException

Recall that the last thread example extended the Thread class and implemented the run() method, the following class uses some of the thread methods mentioned BUT does not extend the Thread class...IT IMPLEMENTS THE Runnable INTERFACE

Implementing the Runnable Interface

The **Clock applet** shown below displays the current time and updates its display every second.

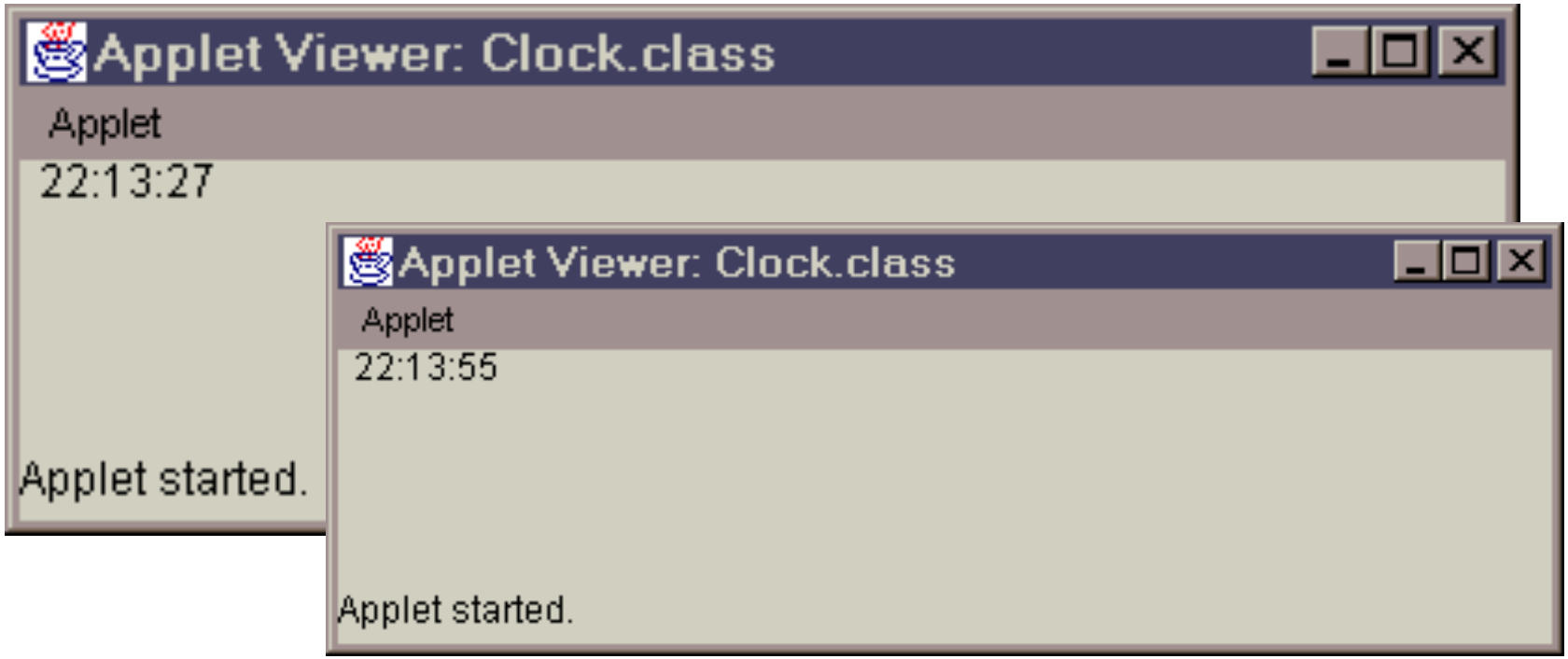
You can scroll this page and perform other tasks while the clock continues to update because the code that updates the clock's display runs within its own thread.

The **Clock applet** uses a different technique than **SimpleThread**, i.e. implements the Runnable interface

Instead of subclassing Thread, Clock implements the **Runnable** interface (and therefore implements the run method defined in it).

The Clock class calls start() on itself, it does this by **creating** a thread and provides itself as an argument to the Thread's constructor.

When created in this way, the Thread gets its **run** method from the object passed into the constructor



TextPad - [C:\ITB\SoftDev-3_JAVA\Misc\Threads\Clock.java]

File Edit Search View Tools Macros Configure Window Help

Clock.java
Command Results
SimpleThread.ja...
TwoThreadsTes...

```
1  import java.awt.Graphics;
2  import java.util.*;
3  import java.text.DateFormat;
4  import java.applet.Applet;
5
6  public class Clock extends Applet implements Runnable
7  {
8      private Thread clockThread = null;
9
10     public void start()
11     {
12         if (clockThread == null)
13         {
14             clockThread = new Thread(this, "Clock");
15             clockThread.start();
16         }
17     }
18
19     public void run()
20     {
21         Thread myThread = Thread.currentThread();
22         while (clockThread == myThread)
23         {
24             repaint();
25             try
26             {
27                 Thread.sleep(1000);
28             }
29             catch (InterruptedException e)
30             {
31              // the VM doesn't want us to sleep anymore,
32              // so get back to work
33             }
34         }
35     }
36
37     public void paint(Graphics g)
38     {
39         // get the time and convert it to a date
40         Calendar cal = Calendar.getInstance();
41         Date date = cal.getTime();
```

ANSI Characters

33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

35 6 Read Ovr Block Sync Rec Caps 23:14

The Clock applet's **run** method loops until the browser asks it to stop.

During each iteration of the loop, the clock **repaints** its display.

The paint method figures out what time it is, formats it in a localized way, and displays it.

TextPad - [C:\ITB\SoftDev-3_JAVA\Misc\Threads\Clock.java]

File Edit Search View Tools Macros Configure Window Help

Clock.java
Command Results
SimpleThread.ja...
TwoThreadsTes...

```
42
43          // format it and display it
44          DateFormat dateFormatter = DateFormat.getTimeInstance();
45          g.drawString(dateFormatter.format(date), 5, 10);
46      }
47
48  // overrides Applet's stop method, not Thread's
49      public void stop()
50      {
51          clockThread = null;
52      }
53  }
54
```

ANSI Characters

33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

54 | 1 | Read | Ovr | Block | Sync | Rec | Caps | 23:15

Using Threads to give other tasks a chance

Example

In the next program we use **two threads**.

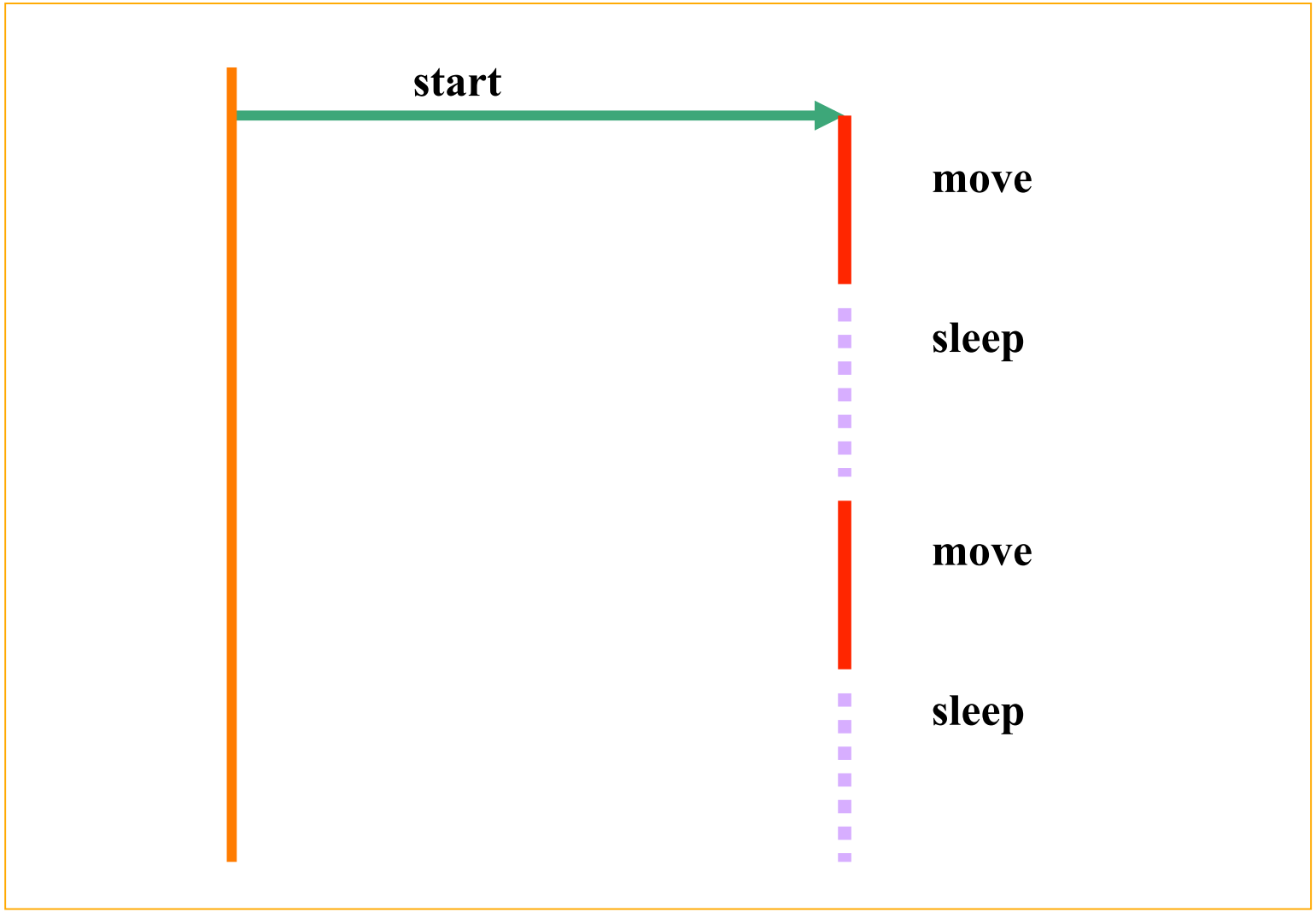
One for the **main thread** that takes care of the **user interface (UI)**, the **other** for the **bouncing ball**.

Because each thread gets a chance to run, the main thread has the opportunity to notice when you click on the “Close” button while the ball is bouncing…. It can then process the “close” action.

There is a simple procedure for running code in a separate thread, as we have seen earlier.

That is, place the code into the run method of a class derived from Thread.

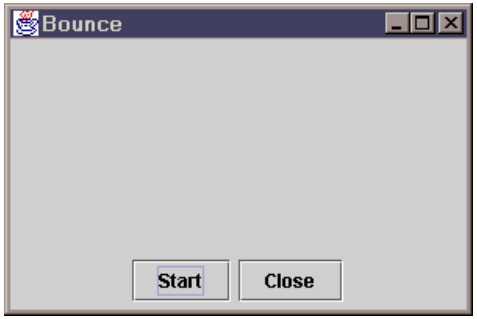
The UI and the ball threads.

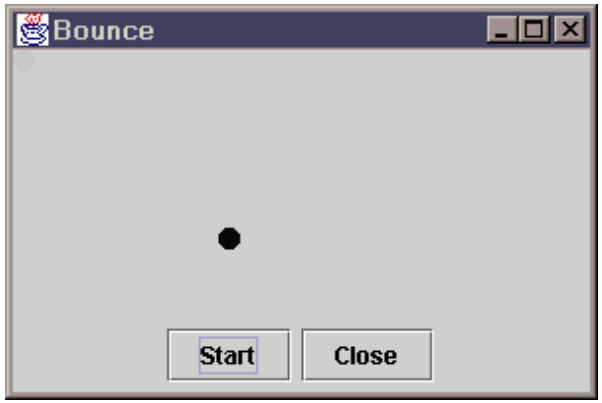


When you run this program, click on the “Start” button again, and again, while a ball is running.

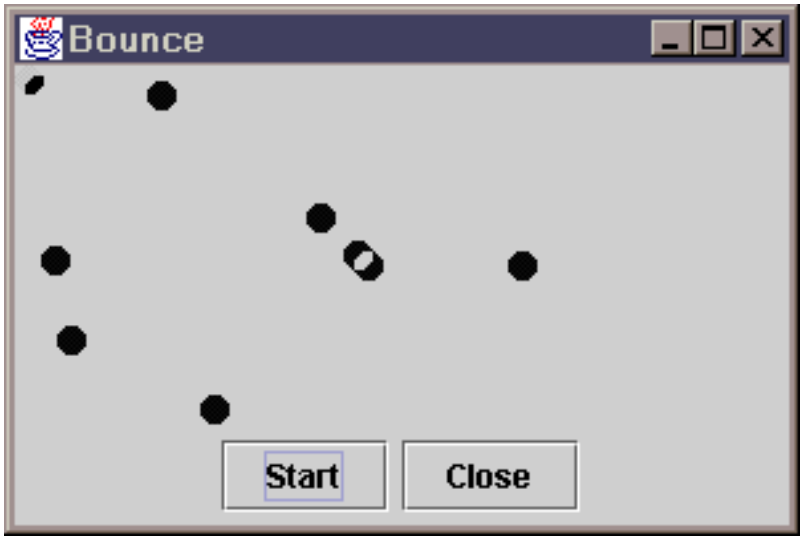
You will notice that many balls start bouncing.

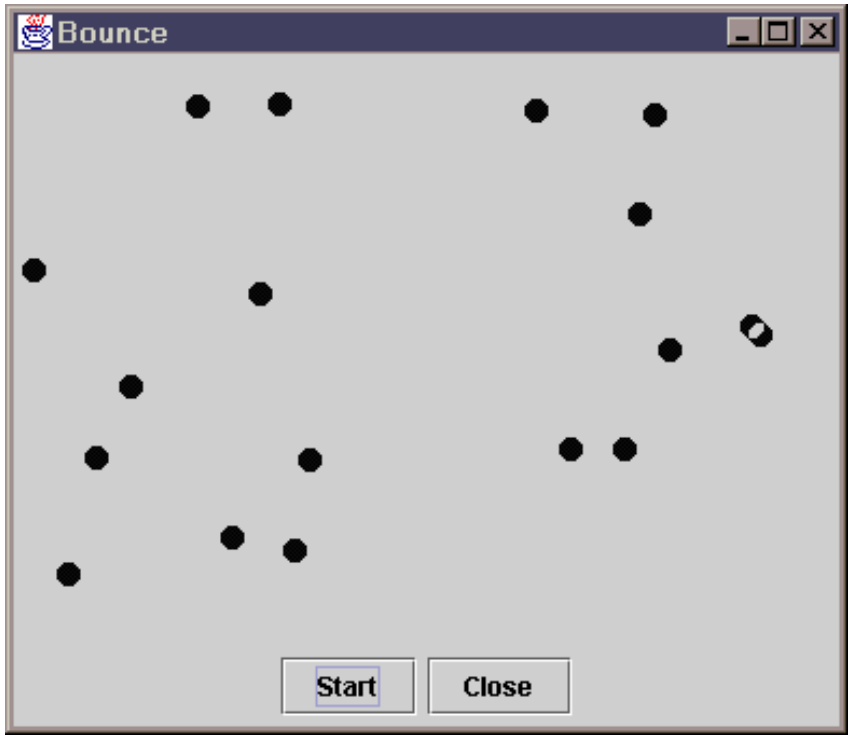
Each ball will move 1000 times then stop.





BounceThread.java





TextPad - [C:\ITB\Java-Information\Core-Java-Advanced-Features-vol2\v2ch1-tl

File Edit Search View Tools Macros Configure Window Help

Bounce.java
BounceThread.j...
Command Results

ANSI Characters

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class BounceThread
6  {  public static void main(String[] args)
7     {  JFrame frame = new BounceThreadFrame();
8        frame.show();
9     }
10 }
11
12 class BounceThreadFrame extends JFrame
13 {  public BounceThreadFrame()
14    {  setSize(300, 200);
15       setTitle("Bounce");
16
17       addWindowListener(new WindowAdapter()
18          {  public void windowClosing(WindowEvent e)
19             {  System.exit(0);
20             }
21          } );
22
23       Container contentPane = getContentPane();
24       canvas = new JPanel();
25       contentPane.add(canvas, "Center");
26       JPanel p = new JPanel();
27       addButton(p, "Start",
28          new ActionListener()
29          {  public void actionPerformed(ActionEvent evt)
30             {  Ball b = new Ball(canvas);
31                b.start();
32             }
33          });
34
35       addButton(p, "Close",
36          new ActionListener()
37          {  public void actionPerformed(ActionEvent evt)
38             {  canvas.setVisible(false);
39                System.exit(0);
40             }
41          });
```

33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

28 30 Read Ovr Block Sync Rec Caps 21:00

TextPad - [C:\ITB\Java-Information\Core-Java-Advanced-Features-vol2\v2ch1-tl

File Edit Search View Tools Macros Configure Window Help

Bounce.java
BounceThread.j...
Command Results

ANSI Characters

```
42          contentPane.add(p, "South");
43       }
44
45       public void addButton(Container c, String title,
46          ActionListener a)
47       {  JButton b = new JButton(title);
48          c.add(b);
49          b.addActionListener(a);
50       }
51
52       private JPanel canvas;
53    }
54
55    class Ball extends Thread
56    {  public Ball(JPanel b) { box = b; }
57
58       public void draw()
59       {  Graphics g = box.getGraphics();
60          g.fillOval(x, y, XSIZE, YSIZE);
61          g.dispose();
62       }
63
64       public void move()
65       {  if (!box.isVisible()) return;
66          Graphics g = box.getGraphics();
67          g.setXORMode(box.getBackground());
68          g.fillOval(x, y, XSIZE, YSIZE);
69          x += dx;
70          y += dy;
71          Dimension d = box.getSize();
72          if (x < 0)
73          { x = 0; dx = -dx; }
74          if (x + XSIZE >= d.width)
75          { x = d.width - XSIZE; dx = -dx; }
76          if (y < 0)
77          { y = 0; dy = -dy; }
78          if (y + YSIZE >= d.height)
79          { y = d.height - YSIZE; dy = -dy; }
80          g.fillOval(x, y, XSIZE, YSIZE);
81          g.dispose();
82       }
```

68 30 Read Ovr Block Sync Rec Caps 21:01

TextPad - [C:\ITB\Java-Information\Core-Java-Advanced-Features-vol2\v2ch1-tl

File Edit Search View Tools Macros Configure Window Help

Bounce.java
BounceThread.j...
Command Results

ANSI Characters

```
83
84      public void run()
85      {   try
86          {   draw();
87              for (int i = 1; i <= 1000; i++)
88              {   move();
89                  sleep(5);
90              }
91          }
92          catch(InterruptedException e) {}
93      }
94
95      private JPanel box;
96      private static final int XSIZE = 10;
97      private static final int YSIZE = 10;
98      private int x = 0;
99      private int y = 0;
100     private int dx = 2;
101     private int dy = 2;
102 }
103
104
105
```

33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

105 | 1 | Read | Ovr | Block | Sync | Rec | Caps | 21:02

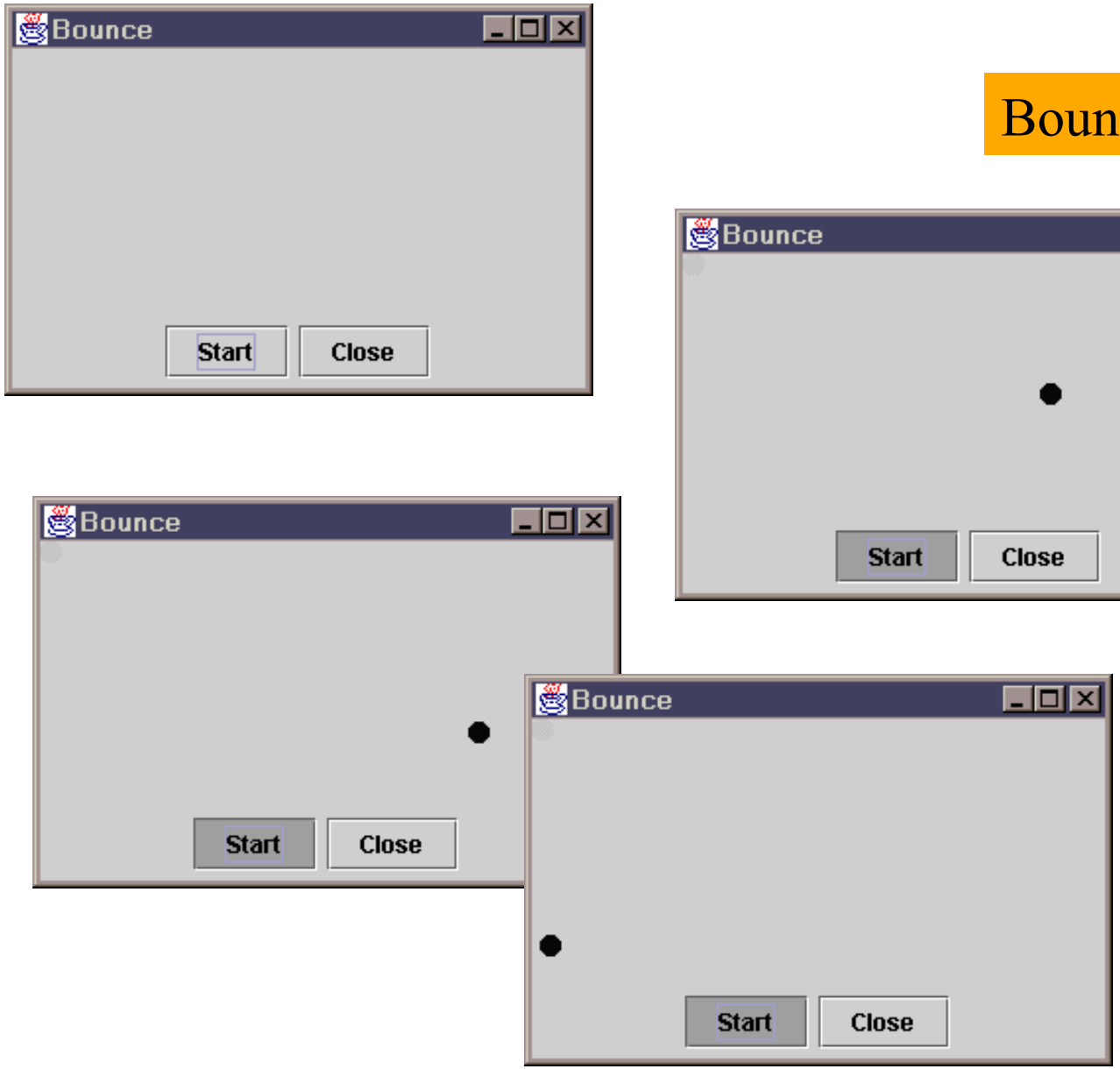
Now, run this next single thread program instead.

Try to click the start button over and over.

What happens?

Can you see the advantages that threads deliver.

Bounce.java


Bounce
Start
Close
Bounce
Start
Close
Bounce
Start
Close
Bounce
Start
Close

TextPad - [C:\ITB\Java-Information\Core-Java-Advanced-Features-vol2\v2ch1-tl

File Edit Search View Tools Macros Configure Window Help

Bounce.java*

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class Bounce
6  {  public static void main(String[] args)
7     {  JFrame frame = new BounceFrame();
8        frame.show();
9     }
10 }
11
12 class BounceFrame extends JFrame
13 {  public BounceFrame()
14    {  setSize(300, 200);
15       setTitle("Bounce");
16
17       addWindowListener(new WindowAdapter()
18          {  public void windowClosing(WindowEvent e)
19             {  System.exit(0);
20             }
21          } );
22
23       Container contentPane = getContentPane();
24       canvas = new JPanel();
25       contentPane.add(canvas, "Center");
26       JPanel p = new JPanel();
27       addButton(p, "Start",
28          new ActionListener()
29          {  public void actionPerformed(ActionEvent evt)
30             {  Ball b = new Ball(canvas);
31                b.bounce();
32             }
33          });
34
35       addButton(p, "Close",
36          new ActionListener()
37          {  public void actionPerformed(ActionEvent evt)
38             {  System.exit(0);
39             }
40          });
41       contentPane.add(p, "South");
```

ANSI Characters

33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

1 1 Read Ovr Block Sync Rec Caps 20:45

TextPad - [C:\ITB\Java-Information\Core-Java-Advanced-Features-vol2\v2ch1-tl

File Edit Search View Tools Macros Configure Window Help

Bounce.java*

```
42      }
43
44      public void addButton(Container c, String title,
45        ActionListener a)
46      {  JButton b = new JButton(title);
47         c.add(b);
48         b.addActionListener(a);
49      }
50
51      private JPanel canvas;
52   }
53
54   class Ball
55   {  public Ball(JPanel b) { box = b; }
56
57      public void draw()
58      {  Graphics g = box.getGraphics();
59         g.fillOval(x, y, XSIZE, YSIZE);
60         g.dispose();
61      }
62
63      public void move()
64      {  Graphics g = box.getGraphics();
65         g.setXORMode(box.getBackground());
66         g.fillOval(x, y, XSIZE, YSIZE);
67         x += dx;
68         y += dy;
69         Dimension d = box.getSize();
70         if (x < 0)
71         { x = 0; dx = -dx; }
72         if (x + XSIZE >= d.width)
73         { x = d.width - XSIZE; dx = -dx; }
74         if (y < 0)
75         { y = 0; dy = -dy; }
76         if (y + YSIZE >= d.height)
77         { y = d.height - YSIZE; dy = -dy; }
78         g.fillOval(x, y, XSIZE, YSIZE);
79         g.dispose();
80      }
81
82      public void bounce()
```

ANSI Characters

```
33   !
34   "
35   #
36   $
37   %
38   &
39   '
40   (
41   )
42   *
43   +
44   ,
45   -
46   .
47   /
48   0
```

42 | 1 | Read | Ovr | Block | Sync | Rec | Caps | 20:46

TextPad - [C:\ITB\Java-Information\Core-Java-Advanced-Features-vol2\v2ch1-tl

File Edit Search View Tools Macros Configure Window Help

Bounce.java*

```
82      public void bounce()
83      {   draw();
84          for (int i = 1; i <= 1000; i++)
85          {   move();
86              try { Thread.sleep(5); }
87              catch(InterruptedException e) {}
88          }
89      }
90
91      private JPanel box;
92      private static final int XSIZE = 10;
93      private static final int YSIZE = 10;
94      private int x = 0;
95      private int y = 0;
96      private int dx = 2;
97      private int dy = 2;
98  }
99
100
101
```

ANSI Characters

33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

82 1 Read Ovr Block Sync Rec Caps 20:47

Methods

Thread()

Constructs a new thread. You must **start** the thread to activate its **run** method.

void run()

You must override this and add the code that you want to have executed in the thread.

void start()

Starts this thread, causing the **run**() method to be called. This method will return immediately. The new thread runs concurrently.

static void sleep(long millis)

Puts the currently executing thread to sleep for the specified number of milliseconds.

Deciding to Use the Runnable Interface

You have now seen two ways to provide the run method for a Java thread:

1. Subclass the Thread class defined in the java.lang package and override the run method.

   Example: See the **SimpleThread** class described in Subclassing Thread and Overriding run.

2. Provide a class that implements the Runnable interface (also defined in the java.lang package) and therefore implements the run method.

   In this case, a Runnable object provides the run method to the thread.

   Example: See the Clock applet just shown.

There are good reasons for choosing either of these options over the other.

However, for most cases, including that of the Clock applet, the following rule of thumb will guide you to the best option.

Rule of Thumb: If your class must subclass some other class (the most common example being Applet), you should use Runnable as described in option #2.

To run in a Java-enabled browser, the Clock class has to be a subclass of the Applet class.

Also, the Clock applet needs a thread so that it can continuously update its display without taking over the process in which it is running.

Some browsers might create a new thread for each applet so as to prevent a misbehaved applet from taking over the main browser thread.

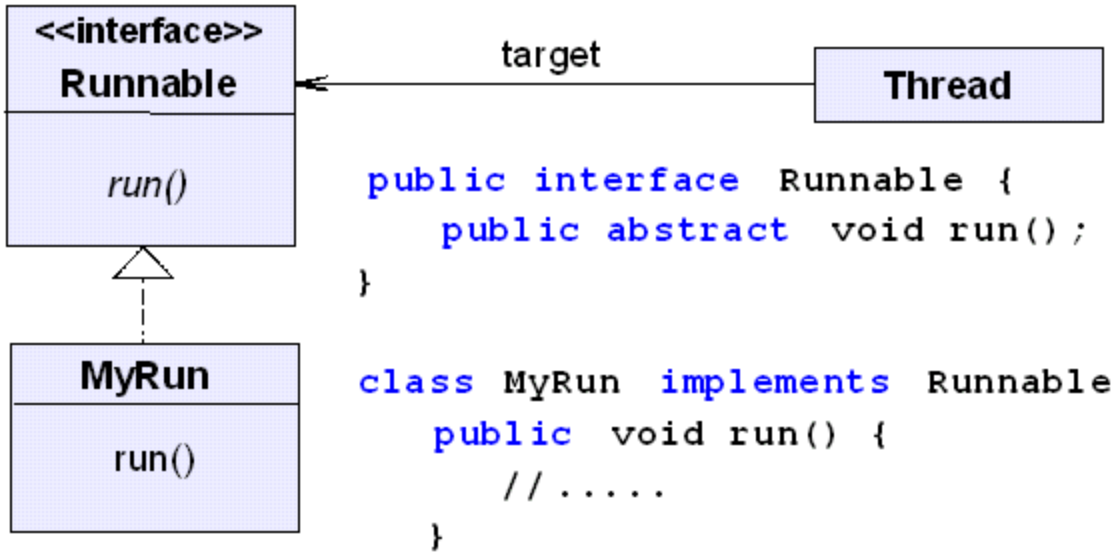
However, you should not count on this when writing your applets; your applets should create their own threads when doing computer-intensive work.

Since the Java language does not support multiple class inheritance, the Clock class cannot be a subclass of both Thread and Applet.

… Therefore the Clock class must use the Runnable interface tc provide its threaded behavior.

threads in Java

Since Java does not permit multiple inheritance, we often implement the **run()** method in a class not derived from Thread but from the interface Runnable.




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

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}
```

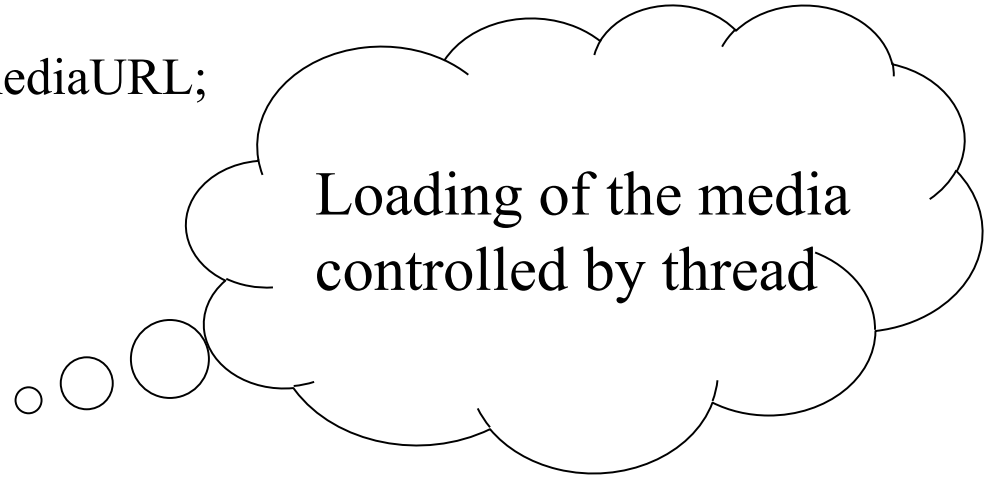
Threads and Multimedia

Threads are often used to allow multi-media items to load before they are played

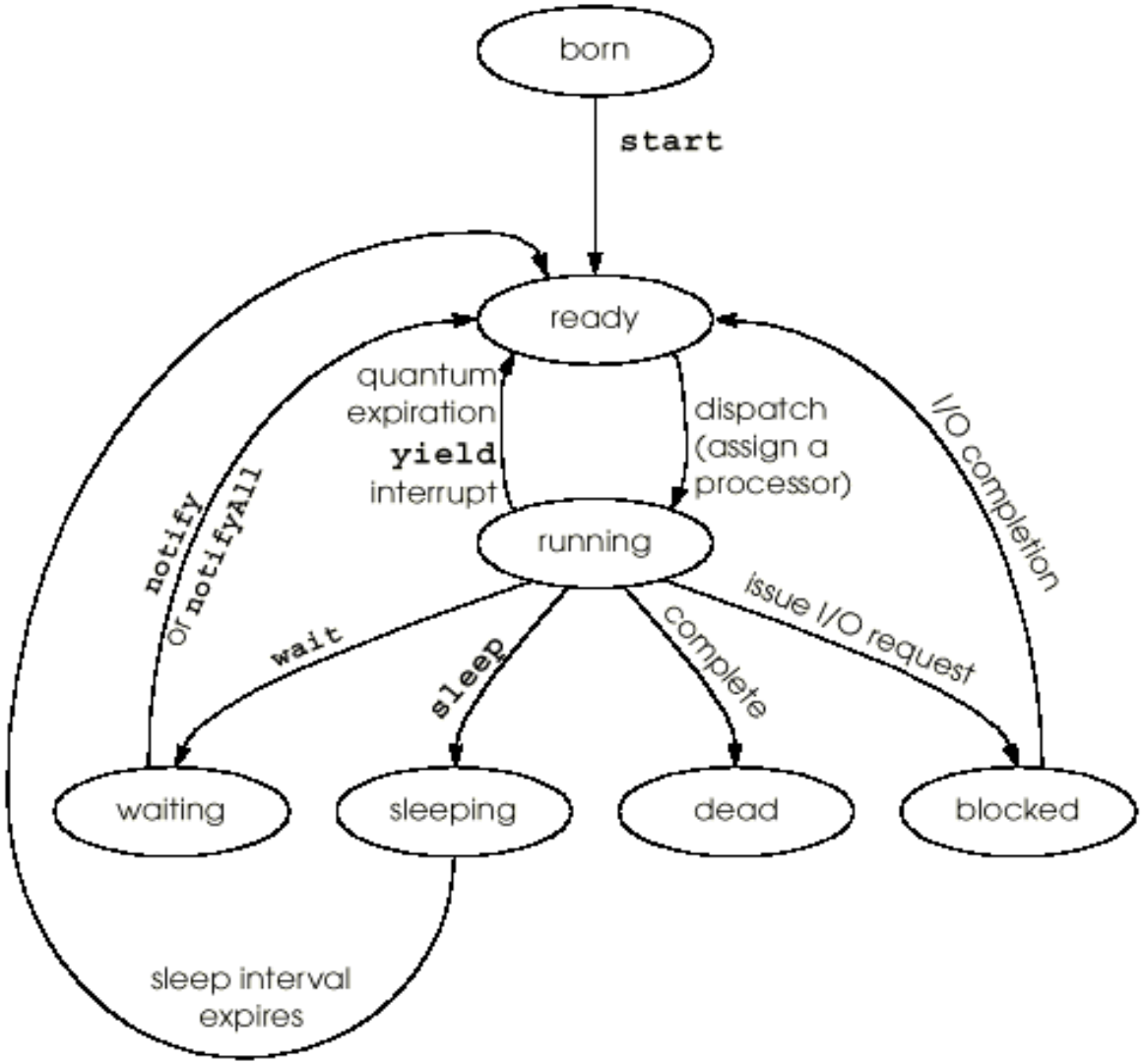
For example the JFrame application shown last week to play music using the AudioClip class could use threads to ensure that the sound file loads prior to playing the sound.

An updated example using threads is contained in this weeks lecture code examples (see MediaPreLoader class and the SoundApplication class)

```
public class MediaPreLoader extends Thread {
    URL completeURL;
    String relativeURL;
    AudioClip currentMedia;

    public MediaPreLoader(URL mediaURL) {

        completeURL = mediaURL;

        start();
    }

    public void run() {
        AudioClip audioClip = Applet.newAudioClip(completeURL);
        currentMedia = audioClip;
    }
}
```



The Life Cycle of a Thread



Life cycle of a thread.

Labwork Exercises

1. Get the examples in this lecture running.

2. Modify the TwoThreads example in this lecture to include and extra Thread and use the implements Runnable instead of extends Thread, e.g., create a third thread, e.g., “Hawaii” and run the test program to see the output.

3. Implement a simple JFrame with a JLabel that outputs a message in the JLabel, e.g., “Please wait” and then waits 10 seconds and overwrites the previous message, e.g., “Ready”.

4. Run the SoundApplication class contained in this lecture. Change it to run various different sound files and you will observe the loading of the media being carried out in a separate thread.