Multiple Threads

Threads

A thread is a single sequential flow of control within a program.
Every program has at least one thread.

Multithreaded Program

- A program with more than one thread.
- Each thread may execute on its own processor in a multiprocessor system.
- All threads may share a single processor.
- OS decides when each thread gets processor (called *scheduling*).
- Java specifies some properties that the scheduler must follow, but not all.

Two Ways to Create a Thread

1. Instantiate a subclass of the class Thread (in *java.lang*).

Override the method

```
public void run()
```

to specify the code that the thread will execute;
we *never* call *run* directly.

Define Sub as a subclass of Thread:

```
class Sub extends Thread
{
    // definition of run() here
}
```

Instantiate Sub to create a Thread object:

```
Sub thread = new Sub();
```

Begin the execution of thread using the instance method *start*:

```
thread.start();
```

One step (provides no reference to the thread):

```
new Sub().start();
```

Note: If we do call *run* directly, it will execute on the current thread (belonging to the method *main*), not a new thread.

Example

```
public class SimpleThread extends Thread
{
    private int threadNum;

    SimpleThread(int k)
    {
        threadNum = k;
        System.out.println("Creating " + threadNum);
    }

    public void run()
    {
        for (int count=5; count>=1; count--)
            System.out.println("Thread " + threadNum
                                          + ": Value = " + count);
    }

    public static void main(String [] a)
    {
        for (int k=1; k<=5; k++)
        {
            SimpleThread st = new SimpleThread(k);
            st.start();
        }

        // We now have 6 threads (5 created plus main)

        System.out.println("All threads started");
    }
}
```

Output

CodeWarrior	Blue (AIX)	HP JDK 1.1	HP JDK 1.2
Creating 1	Creating 1	Creating 1	Creating 1
Creating 2	Thread 1: Value = 5	Creating 2	Creating 2
Creating 3	Thread 1: Value = 4	Creating 3	Thread 1: Value = 5
Thread 1: Value = 5	Thread 1: Value = 3	Creating 4	Thread 1: Value = 4
Thread 1: Value = 4	Thread 1: Value = 2	Creating 5	Thread 1: Value = 3
Thread 1: Value = 3	Thread 1: Value = 1	All Threads Started	Thread 1: Value = 2
Thread 1: Value = 2	Creating 2	Thread 1: Value = 5	Thread 1: Value = 1
Thread 1: Value = 1	Creating 3	Thread 1: Value = 4	Creating 3
Creating 4	Thread 2: Value = 5	Thread 1: Value = 3	Creating 4
Thread 2: Value = 5	Thread 2: Value = 4	Thread 1: Value = 2	Thread 2: Value = 5
Thread 2: Value = 4	Thread 2: Value = 3	Thread 1: Value = 1	Thread 2: Value = 4
Thread 2: Value = 3	Thread 2: Value = 2	Thread 2: Value = 5	Thread 2: Value = 3
Thread 2: Value = 2	Thread 2: Value = 1	Thread 2: Value = 4	Thread 2: Value = 2
Thread 2: Value = 1	Creating 4	Thread 2: Value = 3	Thread 2: Value = 1
Creating 5	Creating 5	Thread 2: Value = 2	Thread 3: Value = 5
Thread 3: Value = 5	Thread 3: Value = 5	Thread 2: Value = 1	Thread 3: Value = 4
Thread 3: Value = 4	Thread 3: Value = 4	Thread 3: Value = 5	Thread 3: Value = 3
Thread 3: Value = 3	Thread 3: Value = 3	Thread 3: Value = 4	Thread 3: Value = 2
Thread 3: Value = 2	Thread 3: Value = 2	Thread 3: Value = 3	Thread 3: Value = 1
Thread 3: Value = 1	Thread 3: Value = 1	Thread 3: Value = 2	Creating 5
All Threads Started	Thread 4: Value = 5	Thread 3: Value = 1	Thread 4: Value = 5
Thread 4: Value = 5	Thread 4: Value = 4	Thread 4: Value = 5	Thread 4: Value = 4
Thread 4: Value = 4	Thread 4: Value = 3	Thread 4: Value = 4	Thread 4: Value = 3
Thread 4: Value = 3	Thread 4: Value = 2	Thread 4: Value = 3	Thread 4: Value = 2
Thread 4: Value = 2	Thread 4: Value = 1	Thread 4: Value = 2	Thread 4: Value = 1
Thread 4: Value = 1	All Threads Started	Thread 4: Value = 1	All Threads Started
Thread 5: Value = 5	Thread 5: Value = 5	Thread 5: Value = 5	Thread 5: Value = 5
Thread 5: Value = 4	Thread 5: Value = 4	Thread 5: Value = 4	Thread 5: Value = 4
Thread 5: Value = 3	Thread 5: Value = 3	Thread 5: Value = 3	Thread 5: Value = 3
Thread 5: Value = 2	Thread 5: Value = 2	Thread 5: Value = 2	Thread 5: Value = 2
Thread 5: Value = 1	Thread 5: Value = 1	Thread 5: Value = 1	Thread 5: Value = 1

2. Create a class that implements the interface Runnable.

```
public interface java.lang.Runnable
{
    public void run();
}
```

The class may extend any other class but must override the method

```
public void run()
```

Suppose the class is called Runner:

```
class Runner extends OtherClass implements Runnable
```

Now instantiate a Thread object using the Thread constructor that takes a Runnable object:

```
Thread th = new Thread(new Runner());
```

Then start the thread:

```
th.start();
```

Combining the steps:

```
new Thread(new Runner()).start();
```

Can also name the Runnable object:

```
Runnable rn = new Runner();
Thread t2 = new Thread(rn);
t2.start();
```

Example

```
public class Simple implements Runnable
{
    private int threadNum;

    Simple(int k)
    {
        threadNum = k;
        Thread st = new Thread(this);
        System.out.println("Creating " + threadNum);
        st.start();
    }

    public void run()
    {
        for (int count=5; count>0; count--)
            System.out.println("Thread " + threadNum +
                                                ": Value = " + count);
    }

    public static void main(String [] args)
    {
        for (int k=1; k<=5; k++) new Simple(k);
        System.out.println("All Threads Started");
    }
}
```

May create a thread outside of its class:

```
    Thread st = new Thread(new RunnableClass());
```

Producer-Consumer Problem

One or more threads produce values.

One or more threads consume values.

Values are maintained as private data in an object with store (put) and load (get) operations.

This object can only hold one value at a time.

```
class MyData
{
    private int datum;

    void put(int num)
    {
        datum = num;
    }

    int get()
    {
        return datum;
    }
}
```

Notice the possibility of interference between the threads:

- A second value is produced before the first is consumed.
- A value is consumed twice before the producer produces a new value.

Version 1: Interference

```
class Producer implements Runnable
{
   private MyData store;
   private int num;

   Producer(int k, MyData st)
   {
      store = st;
      num = k;
   }

   public void run()
   {
      for (int n=1; n<=10; n++)
      {
         try             // simulate time needed to produce next value
         {
            Thread.sleep((int)(Math.random()*500));
         }
         catch (InterruptedException e)
         {
            return;
         }
         store.put(n);
         System.out.println("Producer" + num + " : " + n);
      }
   }
}
```

```
class Consumer implements Runnable
{
   private MyData store;
   private int num;

   Consumer(int k, MyData st)
   {
      store = st;
      num = k;
   }

   public void run()
   {
      while (true)
      {
         int k = store.get();
         System.out.println ("     Consumer" + num + ": " + k);

         try             // simulate time needed to consume value
         {
            Thread.sleep((int)(Math.random()*500));
         }
         catch (InterruptedException e)
         {  return;  }
      }
   }
}

public class Main1
{
   public static void main(String [] args)
   {
      MyData store = new MyData();
      new Thread(new Producer(1, store)).start();
      new Thread(new Consumer(1, store)).start();
   }
}
```

Output

```
                    Consumer1: 0
Producer1: 1
Producer1: 2
                    Consumer1: 2
                    Consumer1: 2
Producer1: 3
                    Consumer1: 3
Producer1: 4
Producer1: 5
                    Consumer1: 5
                    Consumer1: 5
Producer1: 6
                    Consumer1: 6
Producer1: 7
Producer1: 8
                    Consumer1: 8
Producer1: 9
                    Consumer1: 9
Producer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                    Consumer1: 10
                      :
```

Version 2: Busy-waiting

Use flags (boolean variables) to convey the state of the data object.

ready	Next value is ready to be read.
taken	Previous value has been read.

get only when *ready* is true.

put only when *taken* is true.

Use busy-waiting loops to wait until conditions are right.

```
In put:    while (!taken)
               // do nothing
               ;

In get:    while (!ready)
               // do nothing
               ;
```

A thread may hog processor while in a busy-waiting loop, doing nothing.

Unselfish Threads

When a thread is involved in a long computation, say in a loop, it is best for the thread to give other threads a chance to execute.

- It can put itself to sleep, becoming inactive for a fixed amount of time.

  ```
  static void sleep(long millis) throws InterruptedException
  ```

  The exception is raised when another thread interrupts the sleeping thread.

- It can give up control, moving to the end of the queue of runnable threads.

  static void yield()

  If any other threads are currently waiting, one of them is given control to execute.

Basic Principle: If a thread is interacting correctly with other threads (no interference), inserting *sleep* or *yield* callswill not destroy correctness.

Contrapositive: If a program with multiple threads works incorrectly with *sleep* and *yield* calls, it is incorrect without them.

```
class MyData
{
    private int datum;
    private boolean ready;
    private boolean taken;

    MyData()
    {
        ready = false;     // Next value ready to be read
        taken = true;      // Previous value has been read
    }

    void put(int num)
    {
        while (!taken)
            Thread.yield();             // Note yield
        datum = num;
        taken = false;
        ready = true;
        Main2.delay(200);               // will not affect correctness
    }
    int get()
    {
        while (!ready)
            Thread.yield();             // Note yield

        int num = datum;
        Main2.delay(200);               // will not affect correctness
        ready = false;
        taken = true;
        return num;
    }
}
```

Producers and Consumers are the same, except we create two Consumers.

```
public class Main2
{
    public static void main(String [] args)
    {
        MyData store = new MyData();
        new Thread(new Producer(1, store)).start();
        new Thread(new Consumer(1, store)).start();
        new Thread(new Consumer(2, store)).start();
    }

    static void delay(int n)
    {                                   // use to force interleaving
        try
        {   Thread.sleep(n);   }
        catch (InterruptedException e) { }
    }
}
```

Output

```
Producer1: 1
                        Consumer1: 1
Producer1: 2
Producer1: 3
                        Consumer2: 1
                        Consumer2: 3
Producer1: 4
                        Consumer1: 4
Producer1: 5
                        Consumer2: 5
Producer1: 6
                        Consumer1: 6
Producer1: 7
                        Consumer2: 7
Producer1: 8
                        Consumer2: 8
Producer1: 9
                        Consumer2: 9
Producer1: 10
                        Consumer1: 10
```

Race Conditions

With more than one consumer, a race condition may occur.

This means that the result depends on the order of interleaving of the threads.

Producer Thread	**Consumer1 Thread**	**Consumer2 Thread**

	while (!ready)
	   Thread.yield();	

		while (!ready)
		   Thread.yield();

datum = num;
taken = **false**;
ready = **true**;		

	num = datum;	
		num = datum;

	ready = **false**;
	taken = **true**;
	return num;	

		ready = **false**;
		taken = **true**;
		return num;

Same value is consumed by two different threads.

Another Example

```
Producer                  Consumer1                 Consumer2
Thread                    Thread                    Thread

                          while (!ready)
                             Thread.yield();
                                                    while (!ready)
                                                       Thread.yield();
datum = num;
taken = false;
ready = true;
                          num = datum;
                                                    num = datum;
while (!taken)
   Thread.yield();
                          ready = false;
                          taken = true;
datum = num;
taken = false;
ready = true;
                          return num;
while (!taken)
   Thread.yield();
                                                    ready = false;
                                                    taken = true;
                                                    return num;
datum = num;
taken = false;
ready = true;
```

A value is missed by the consumers.

Monitors

A monitor acts as a wall around an object so that all of the "synchronized" code from the class is inside the wall.

Only one thread at a time may execute (synchronized) code inside a monitor for an object.

Monitor code is marked by the keyword **synchronized**.

- A modifier on a method.
- A modifier with an object parameter on a block.

A class may have several synchronized code segments, but they all lie inside the wall of the same monitor.

The code inside of a monitor is called a *critical section*.

With a monitor for the critical section, while one thread is manipulating the datum, no other thread can access or change the datum.

Version 3: Monitor

Use **synchronized** to create a monitor for MyData.

```
class MyData
{
    private int datum;
    private boolean ready = false;
    private boolean taken = true;

    synchronized void put(int num)
    {
        while (!taken)  Thread.yield();
        datum = num;
        taken = false;
        ready = true;
    }

    synchronized int get()
    {
        while (!ready)  Thread.yield();
        ready = false;
        taken = true;
        return datum;     // No need to save datum since put()
    }                     // cannot change it while a thread
}                         // is executing get().
```

Output

None

What is the Problem?

Suppose that a thread is in one of these methods,
stuck on the **while** (!flag).

No other thread can get inside the monitor to change the flag, and executing the *yield* method will not release the lock on the monitor.

The threads are *deadlocked* or in a *deadly embrace*.

Solution

Release the monitor when in a busy-waiting loop.

The polling loops that test ready and taken, the busy-waiting loops, are a problem.

- Processor time is wasted in busy-waiting loops.
- Without timeslicing one thread can get stuck in a busy-waiting loop forever.
- We want a thread to relinquish control and the monitor lock when it is not able to continue with "productive" work.

Methods in class Object

final void wait() **throws** InterruptedException

Causes thread to suspend and wait to be notified before continuing; in addition, the thread releases the monitor lock while it is waiting.

final void notify()

Allows one of the currently waiting threads to re-acquire the monitor.

```
    final void notifyAll()
```

Allows all of the currently waiting threads to be able to re-acquire the monitor.
Only one will get into the monitor at a time.

These methods may be called *only* from within *synchronized* code.

Version 4: wait and notify

```
class MyData
{
    private int datum;
    private boolean ready= false;      // Use only one flag
                                       // taken = !ready
    synchronized void put(int num)
    {
        while (ready)
            try { wait(); }
            catch (InterruptedException e)
            { return; }
        datum = num;
        ready = true;
        notifyAll();
    }

    synchronized int get()
    {
        while (!ready)
            try { wait(); }
            catch (InterruptedException e)
            { return Integer.MIN_Value; }
        int num = datum;
        ready = false;
        notifyAll();
        return num;    // could be an interleave before return
    }
}
```

Two Producers—Two Consumers (only up to 7)

```
Producer1: 1
                                          Consumer1: 1
                                                                 Consumer2: -1
                    Producer2: -1
Producer1: 2
                                                                 Consumer2: 2
                    Producer2: -2
                                          Consumer1: -2
Producer1: 3
                                                                 Consumer2: 3
                    Producer2: -3
                                          Consumer1: -3
                                                                 Consumer2: -4
                    Producer2: -4
Producer1: 4
                                          Consumer1: 4
                    Producer2: -5
Producer1: 5
                                                                 Consumer2: -5
                    Producer2: -6
                                                                 Consumer2: 5
                                          Consumer1: -6
                    Producer2: -7
                                                                 Consumer2: -7
Producer1: 6

                                                                 Consumer2: 6
Producer1: 7
                                                                 Consumer2: 7
// Need cntl-c to stop program
```

How to Stop the Consumers

The Producer threads terminate when their *run* methods exit after finishing their **for** loops.

The Consumer threads, however, are stuck in an endless **while** loop.

First Step: Change the loop in Consumer

```
class Consumer extends Thread
{
        private MyData store;
        private int num;
        boolean done = false;

        Consumer(int k, MyData st)
        {
                store = st;
                num = k;
        }

        void setDone(boolean b)
        {   done = b;   }

        public void run()
        {
                while (!done)
                {
                        int k = store.get();
                        System.out.println ("   Consumer" + num + ": " + k);

                        try   // simulate time needed to consume value
                        {
                                Thread.sleep((int)(Math.random()*500));
                        }
                        catch (InterruptedException e)
                        { return; }
                }
        }
}
```

Second Step: Have main wait for Producers to stop

The *main* method that we have now terminates right after starting the threads it has created.

We want the *main* method to keep running and wait for the Producers to complete their tasks.

Another instance method:

```
    public final void join() throws InterruptedException
```

If the code in a thread calls *thrd.join()* where *thrd* is another thread, the first thread waits (blocks) until the second thread, *thrd*, terminates.

Change the main method as follows:

```
    public static void main(String [] args)
    {
        MyData store = new MyData();

        Thread pt1 = new Thread(new Producer(1, store, 1));
        Thread pt2 = new Thread(new Producer(2, store, -1));
        pt1.start();
        pt2.start();

        Consumer c1 = new Consumer(1, store);
        Consumer c2 = new Consumer(2, store);
        Thread ct1 = new Thread(c1);
        Thread ct2 = new Thread(c2);
        ct1.start();
        ct2.start();

        try
        {
            pt1.join();   pt2.join();
        }
        catch (InterruptedException e) { }
        System.out.println("Both Producers are done.");
        c1.setDone(true);
        c2.setDone(true);
    }
```

After both Producers have terminated, the *main* method signals to the two Consumers that they may shut down now.

These alterations in Consumer and *main* work some of the time. A Consumer termintes only if it executes its **while** loop test.

But if one of the Consumers is in a wait state because it relinquished the monitor lock, at the end of the *main* method there are no threads available to signal (notify) the Consumer that if may continue.

Third Step: Have main interrupt the Consumers

Finally, add the following code at the very end of the *main* method.

```
        if (ct1.isAlive())
            ct1.interrupt();
        if (ct2.isAlive())
            ct2.interrupt();
```

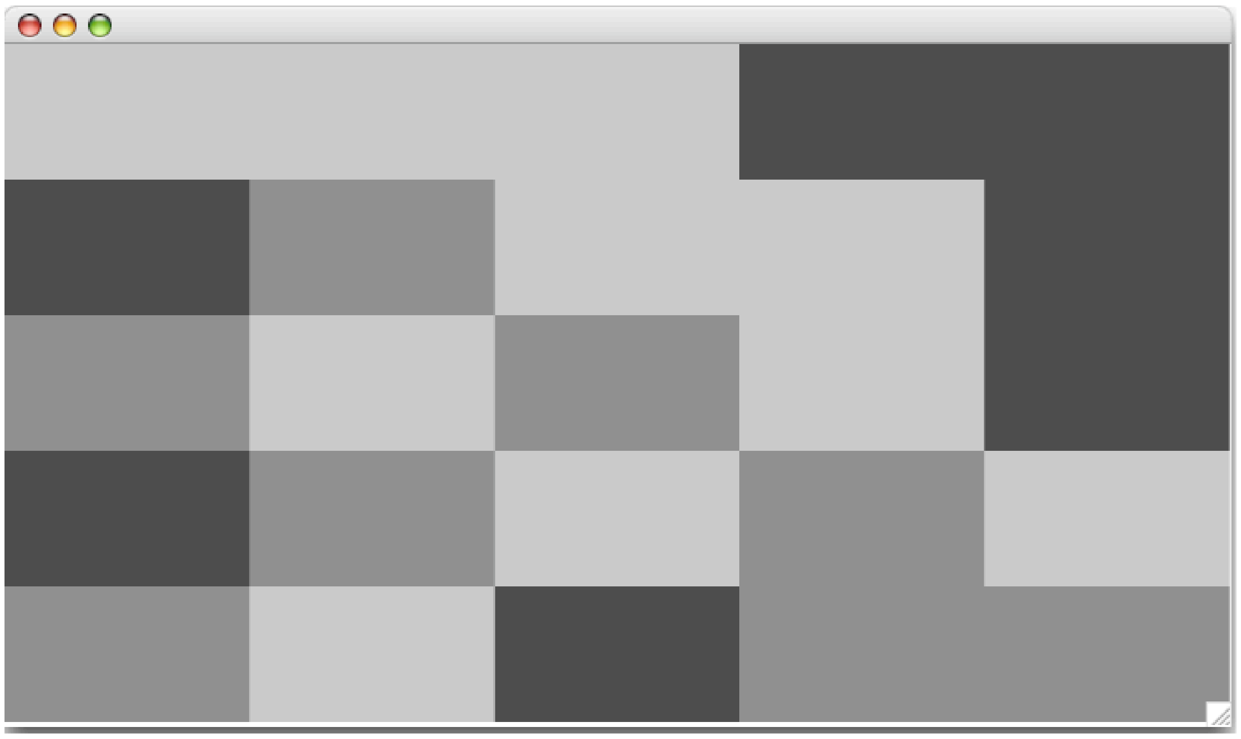
Now the entire program shuts down on it own.

Animation Example

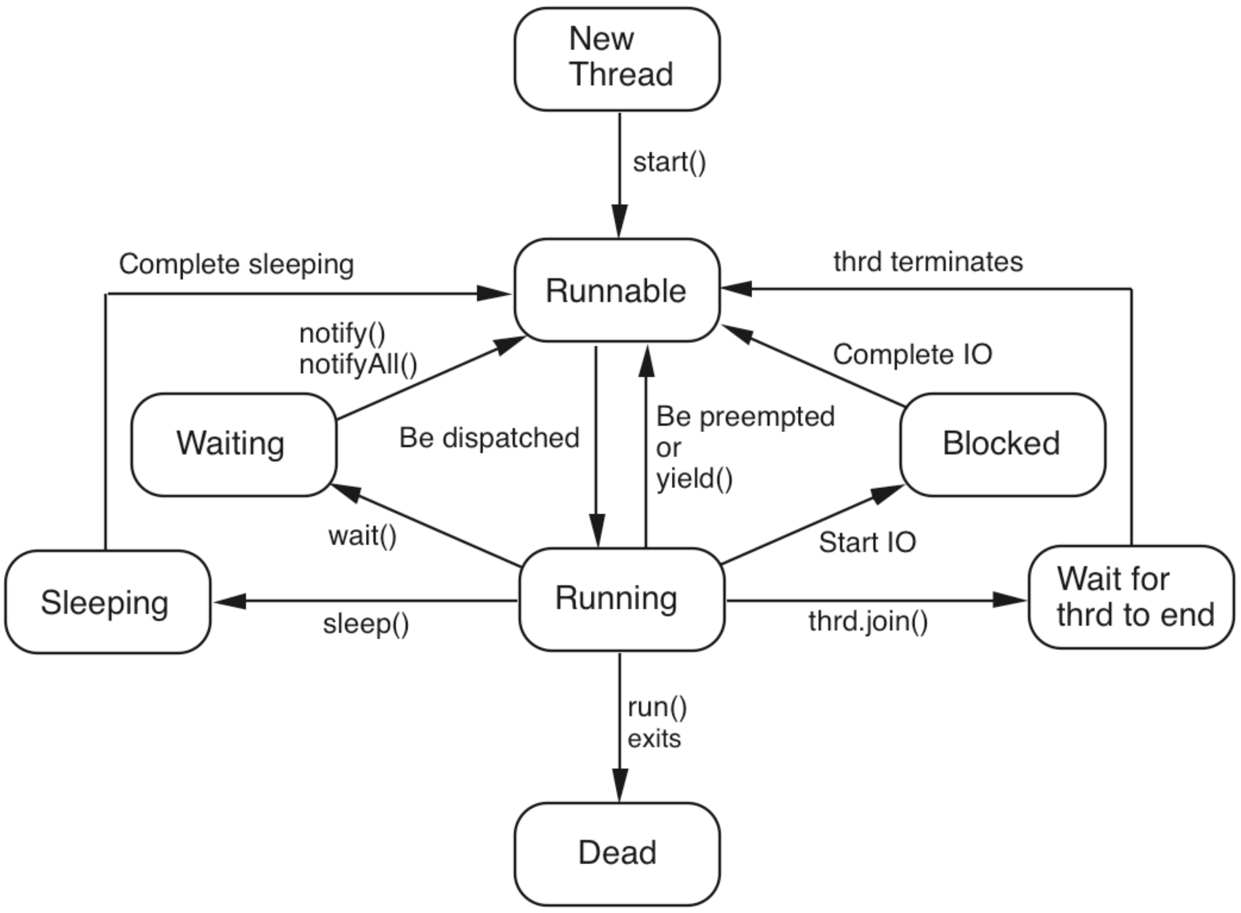
An application with 25 threads each of which is a panel whose color is chosen arbitrarily on the screen.

Create a 5 by 5 grid and instantiate 25 JPanels using the class CBox, each with a thread that runs independently, choosing one of set of colors for its rectangle that is drawn at its position on the grid.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BoxesApp extends JFrame
{
    static final int GRID = 5;

    BoxesApp()
    {
        Container con = getContentPane();
        con.setBackground(Color.black);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        con.setLayout(new GridLayout(GRID, GRID));
        for (int k = 1; k <= GRID*GRID; k++)
        {
            CBox cb = new CBox();
            con.add(cb);
            new Thread(cb).start();
        }
    }

    public static void main(String [] args)
    {
        BoxesApp ba = new BoxesApp();
        ba.setSize(700,500);
        ba.setVisible(true);
    }
}

class CBox extends JPanel implements Runnable
{
    static final Color [] c olors =
                { Color.gray, Color.lightGray, Color.darkGray, };
    Color  nextColor()
    {
        return colors[(int)(Math.random()*colors.length)];
    }

    public void paintComponent(Graphics  g)
    {                                              // redundant method
        super.paintComponent(g);
    }

    public void run()
    {
        while (true)
        {
            setBackground(nextColor());
            repaint();
            try
            {    Thread.sleep((int)(1000*Math.random()));  }
            catch (InterruptedException e) {  return;  }
        }
    }
}
```



States and Transitions of a Thread



Summary

A thread can be in one of four states:

1. New: Created but not started.
2. Runnable: Created and started and able to execute; whether it is actually running depends on the scheduler.
3. Blocked: Something prevents the thread from being runnable, so the scheduler ignores it.
4. Dead: Thread has returned from its *run* method

A thread can be blocked for one of five reasons:

1. Asleep because of a call to *sleep*.
2. Suspended because of a call to *join*.
3. Suspended because of a call to *wait*.
4. Waiting for IO to complete.
5. Trying to call a synchronized method on an object whose lock is unavailable.

A Threaded Domino Server

The Domino Server that we wrote earlier can handle only one client at a time.

Using threads, the server can be altered to deal with multiple clients at the same time easily.

The interaction with each client is handled in a thread defined by the class Handler.

```
import java.io.*;
import java.net.*;

public class DominoServer
{
    public static void main(String [] args)
    {
        try
        {   ServerSocket ss = new ServerSocket(9876);
            System.out.println("Domino Server running on " +
                    InetAddress.getLocalHost().getHostName());
            System.out.println("Use control-c to stop server.");
            while (true)
            {
                Socket inSock = ss.accept();
                Handler h = new Handler(inSock);
                h.start();
            }
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}

class Handler extends Thread
{
    private Socket sock;

    Handler(Socket s)
    {  sock = s;  }

    public void run()
    {
        try
        {   String client = sock.getInetAddress().getHostName();
            System.out.println("Connection from client: " + client);

            InputStream in = sock.getInputStream();
            OutputStream out = sock.getOutputStream();
            ObjectOutputStream outStrm =
                                new ObjectOutputStream(out);
            ObjectInputStream inStrm =
                                new ObjectInputStream(in);
            outStrm.writeUTF("Enter number of dominoes desired");
            outStrm.flush();
            int cnt = inStrm.readInt();
            for (int k=1; k<=cnt; k++)
            {
                Domino d = new Domino(true);
                outStrm.writeObject(d);
            }
            System.out.println(cnt + " dominoes sent to " + client);
            sock.close();
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}

// Do not forget the Domino class

class Domino implements Serializable
{
   :
}
```

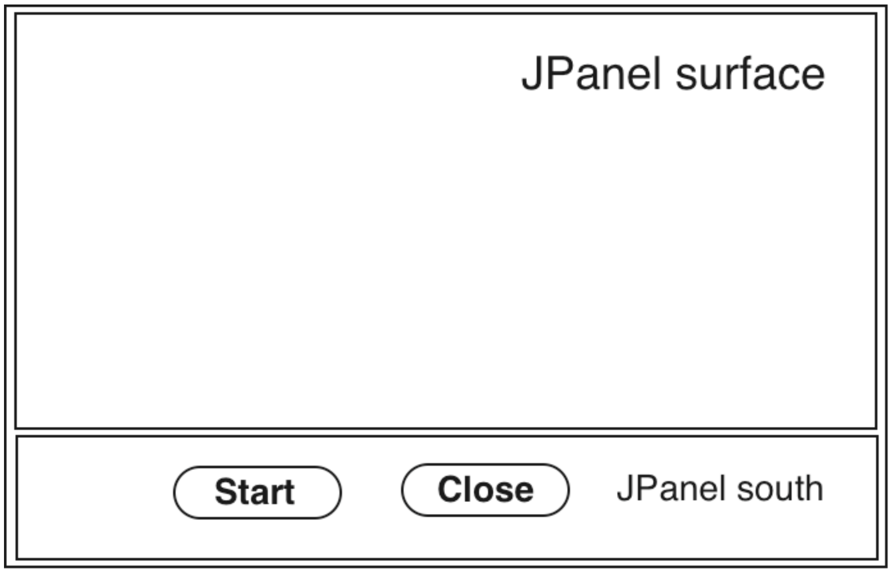
Sample Execution: Server Side

```
% java DominoServer
Domino Server running on desert.divms.uiowa.edu
Use control-c to terminate server.
Connection from client: breeze.cs.uiowa.edu
Connection from client: gorge.divms.uiowa.edu
Connection from client: gust.cs.uiowa.edu
Connection from client: lisbon.divms.uiowa.edu
4 dominoes sent to gorge.divms.uiowa.edu
7 dominoes sent to breeze.cs.uiowa.edu
11 dominoes sent to lisbon.divms.uiowa.edu
18 dominoes sent to gust.cs.uiowa.edu
Connection from client: gust.cs.uiowa.edu
29 dominoes sent to gust.cs.uiowa.edu
```

Bouncing Ball

class Bounce

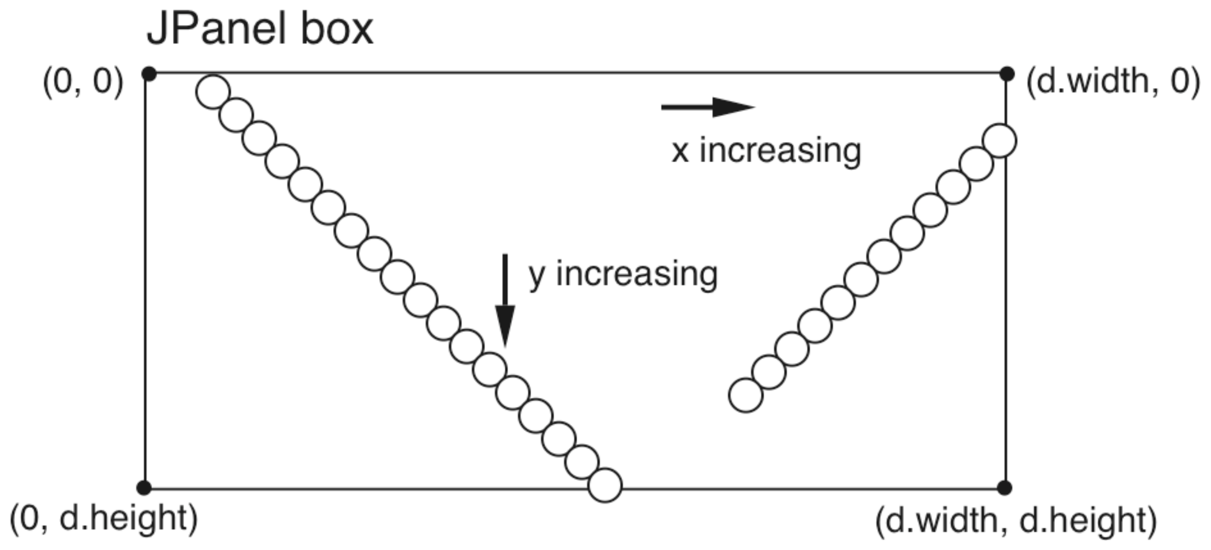
- Subclass of JFrame that instantiates itself in main.
- Initialization is in its constructor.
- Implement an ActionListener to respond to button presses.
- A JPanel that is passed to the ball class.
- A JPanel for two buttons at the bottom.



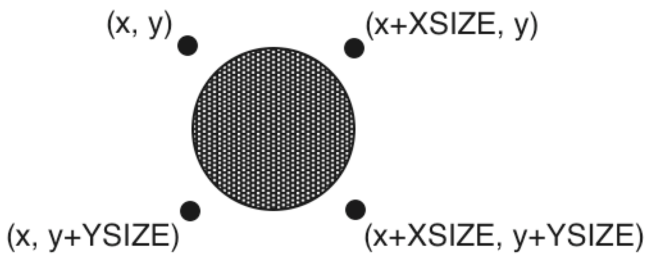
class Ball

- Constructor takes a panel, called *box* to bounce in, and a color.
- A method called *bounce* controls the ball using the methods:
  - *draw* to draw the ball in its initial position
  - *move* to advance the ball two pixels in each direction
    - Erases current ball, and then redraws ball in new spot
    - Does this 1000 times.

- Dimension d = box.getSize();
- Ball is XSIZE=20 by YSIZE=20 pixels.



Changing Direction of Ball



```
if (x <= 0)  { x = 0;  dx = -dx;  }

if (x+XSIZE >= d.width)
{  x = d.width-XSIZE;   dx = -dx;  }

if (y <= 0)  { y = 0;  dy = -dy;  }

if (y+YSIZE >= d.height)
{  y = d.height-YSIZE;   dy = -dy;  }
```

Code

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Bounce extends JFrame
{
    private JPanel surface;

    public static void main(String [] a)
    {
        JFrame jf = new Bounce();
        jf.setSize(600, 500);
        jf.setVisible(true);
    }

    Bounce()
    {
        setTitle("Bounce");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container cp = getContentPane();
        cp.setBackground(new Color(255, 204, 153));
        surface = new JPanel();
        surface.setBackground(new Color(255, 204, 153));
        cp.add(surface, "Center");
        JPanel south = new JPanel();
        south.setBackground(new Color(153, 204, 153));
        JButton start = new JButton("Start");
        start.addActionListener(new ButtonHandler());
        south.add(start);
```

```
          JButton close = new JButton("Close");
          close.addActionListener(new ButtonHandler());
          south.add(close);
          cp.add(south, "South");
     }

     class ButtonHandler implements ActionListener
     {
          public void actionPerformed(ActionEvent e)
          {
               String arg = e.getActionCommand();
               if (arg.equals("Start"))
               {
                    Ball b = new Ball(surface, Color.blue);
                    b.bounce();
               }
               else if (arg.equals("Close"))
               {
                    System.exit(0);
               }
          }
     }

}         // end of class Bounce
```

```
class Ball
{
    Ball(JPanel c, Color clr)
    {
        box = c;
        color = clr;
    }

    void bounce()
    {
        draw();                         // draw ball for first time
        for (int k=1; k<=1000; k++)
        {
            try
            {    Thread.sleep(20);
            }
            catch (InterruptedException e)
            { return; }
            move();
        }
    }


    void draw()
    {
        Graphics g = box.getGraphics();
        g.setColor(color);
        g.fillOval(x, y, XSIZE, YSIZE);
    }

    void move()
    {
        Graphics g = box.getGraphics();
        g.setColor(box.getBackground());
        g.fillOval(x, y, XSIZE, YSIZE);    // draw over old ball
```

```
            x = x + dx;

            y = y + dy;

            Dimension d = box.getSize();

            if (x <= 0)
            { x = 0;  dx = -dx;  }

            if (x+XSIZE >= d.width)
            {  x = d.width-XSIZE;   dx = -dx;  }

            if (y <= 0)
            { y = 0;  dy = -dy;  }

            if (y+YSIZE >= d.height)
            {  y = d.height-YSIZE;   dy = -dy;  }

            g.setColor(color);

            g.fillOval(x, y, XSIZE, YSIZE);
        }

        private JPanel box;

        private Color color;

        private static final int XSIZE=20, YSIZE=20;

        private int x=50, y=0, dx=2, dy=2;
}
```

Problem

While ball is moving, no other method can be executing.

- Button presses are ignored until ball is done moving.
- Window close is ignored until ball is done moving.
- Ball is executing its code using the event handling thread.

The execution of the program involves two threads:

1. A thread that executed the three lines of the main method and then terminates.
2. A thread that listens for the occurrence of events and calls the appropriate methods in the Listener objects. The method *bounce* executes on this thread. Drawing is done on this thread.

Solution

Make Ball a thread.

- Then other processes may execute while Ball thread is sleeping or not scheduled to execute.
- Also, may have multiple balls.
- Every time "Start" is pressed, a new ball thread is started.

In class Bounce

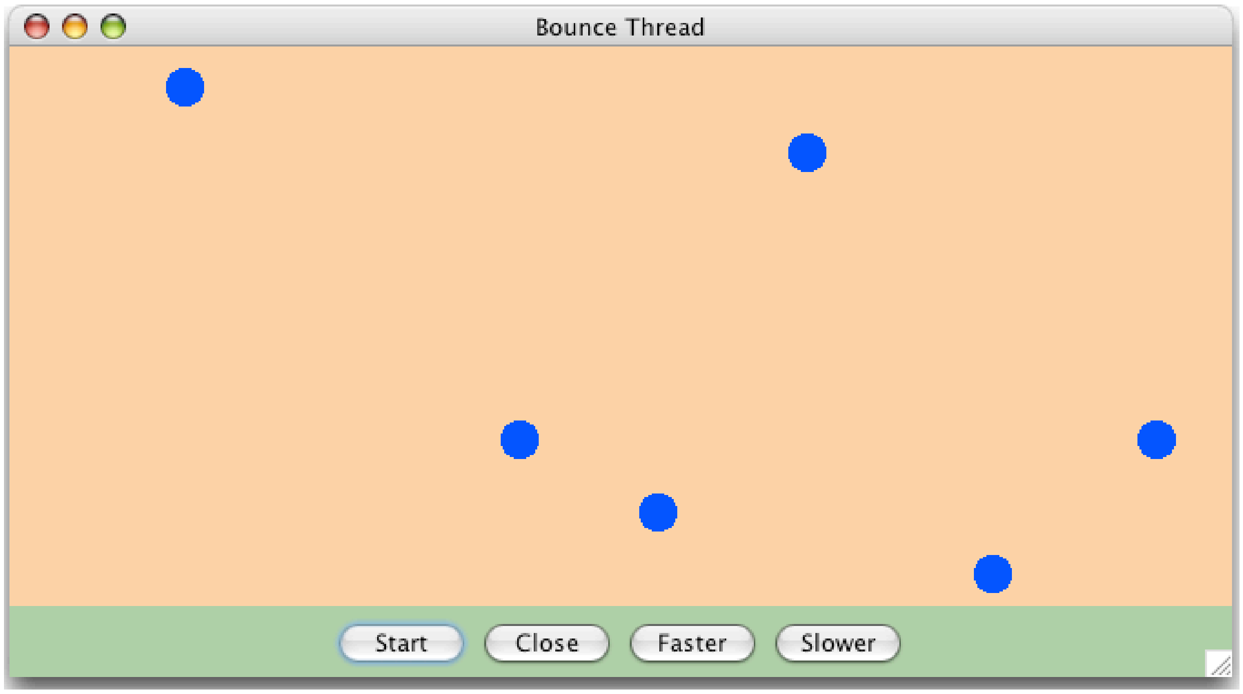
```
if (arg.equals("Start")
{
    Ball b = new Ball(surface, Color.blue):
    b.start();            // Note: calls start, not bounce
}
```

In class Ball

```
class Ball extends Thread
{
    // Instead of method bounce()

    public void run()
    {
        draw();            // draw ball for first time
        while (true)        //  run until Close button is pressed
        {
            try {  Thread.sleep(20);  }
            catch  (InterruptedException  e)
            {  return;  }
            move();
        }
    }
```

Other methods are identical.

Variation: Add the buttons to speed up and slow down the moving balls.


Bounce Thread
Start
Close
Faster
Slower