Topic 9: Processes and Threads

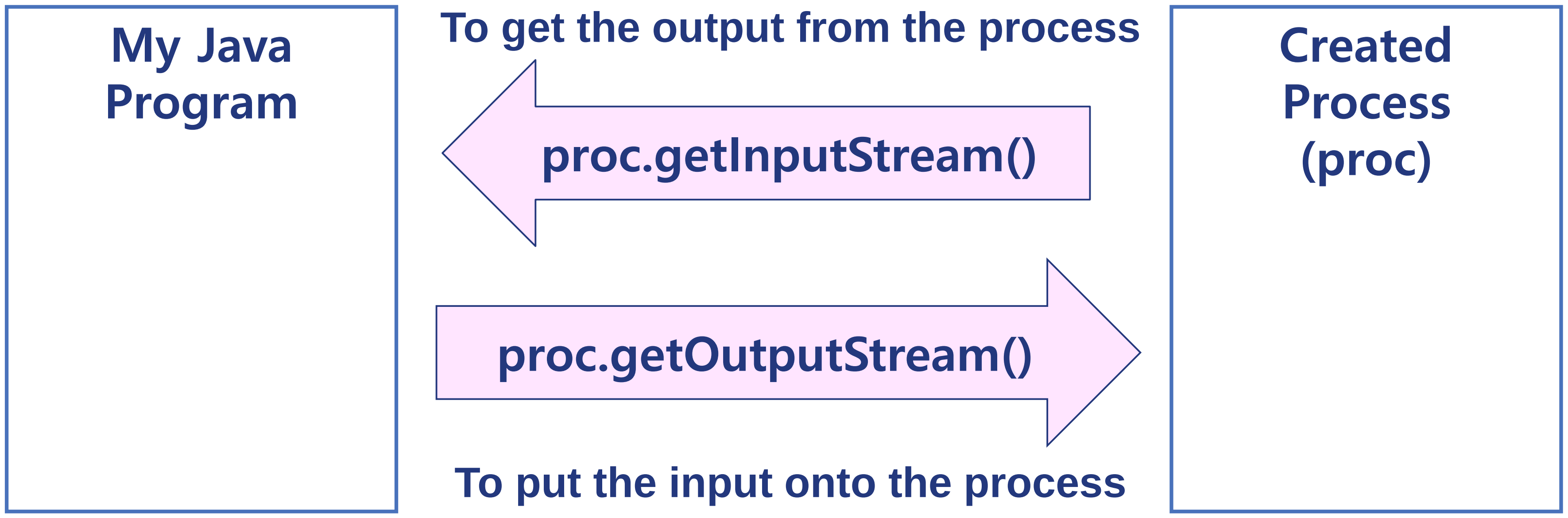
- ◆Processes
- ◆Threads
  - ◆Creating Threads
  - ◆Interrupting Threads
  - ◆Joining Threads
  - ◆Synchronization between Threads
  - ◆Thread-Safe Collections in Java

Core Java Volume 1 – Chapter 14 Concurrency

Creating and Executing Processes

```
public class Exec {
  public static void main(String[] args) {
     try {
        // method 1
        Process proc = Runtime.getRuntime().exec("cmd /c dir");
        // method 2
        Process proc = new ProcessBuilder("cmd", "/c").start();
     }
     catch(Exception e) { e.printStackTrace(); }
  }
}
```

Getting the Standard Input/Output from the Process



Getting the Output

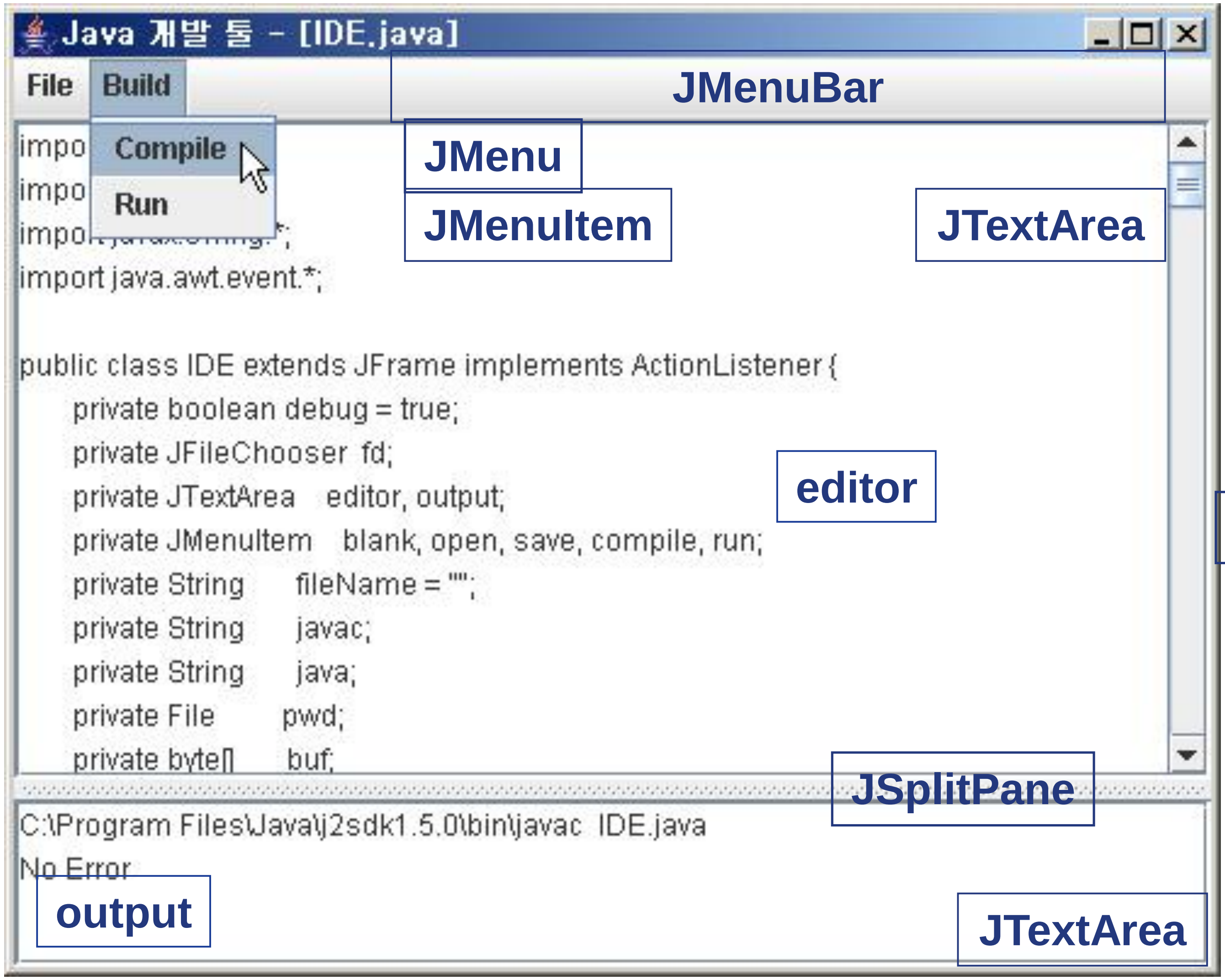
❖ To get the output from the process, use getInputStream()

```
import java.io.*;
public class Ls {
 public static void main(String args[]) {
   try {
     String param = "";
     for(int i = 0; i < args.length; i++) { param += " " + args[i]; }
     Process proc = Runtime.getRuntime().exec("cmd /c dir " + param);
     InputStream in = proc.getInputStream();
     byte buffer[] = new byte[1024];
     int n = -1;
     while ((n = in.read(buffer)) != -1) {
       System.out.print(new String(buffer, 0, n));
     }
   } catch(Exception e) { e.printStackTrace(); }
 }
}
```

❖ To put the input onto the process, use getOutputStream()

```
import java.io.*;
public class Less {
  public static void main(String args[]) throws Exception {
     Process proc = Runtime.getRuntime().exec("cmd /c more");
     InputStream in = proc.getInputStream();              // new process ➔ I
     OutputStream out = proc.getOutputStream();           // I ➔ new process
     byte buffer[] = new byte[1024];
     int n = -1;
     InputStream fin = null;
     if(args.length > 0) fin = new FileInputStream(args[0]);
     else fin = System.in;
     while((n = fin.read(buffer)) != -1) { out.write(buffer, 0, n); }
     fin.close();
     out.close();
     while((n = in.read(buffer)) != -1)
        System.out.print(new String(buffer, 0, n));
     in.close();
  }
}
```

Example: IDE.java



```
impo
impo
impo          *;
import java.awt.event.*;

public class IDE extends JFrame implements ActionListener {
     private boolean debug = true;
     private JFileChooser  fd;
     private JTextArea    editor, output;
     private JMenuItem    blank, open, save, compile, run;
     private String       fileName = "";
     private String       javac;
     private String       java;
     private File        pwd;
     private byte[]      buf;
```

```
C:\Program Files\Java\j2sdk1.5.0\bin\javac  IDE.java
No Error
```

12.1 프로세스

```
import java.awt.*;
import java.io.*;
import javax.swing.*;
import java.awt.event.*;
public class IDE extends JFrame implements ActionListener {
  private boolean debug = true;
  private JFileChooser            fd; // javax.swing.JFileChooser
  private JTextArea               editor, output;
  private JMenuItem               blank, open, save, compile, run;
  private String      fileName = "";
  private String      javac = "C:₩₩Program Files₩₩Java₩₩jdk1.6.0_16₩₩bin₩₩javac";
  private String      java = "C:₩₩Program Files₩₩Java₩₩jre6₩₩bin₩₩java";
  private File       pwd;
  private byte[]      buf;

  public IDE() {
    super("Java 개발 툴");
    setDefaultCloseOperation(EXIT_ON_CLOSE); setSize(500, 400);

    editor = new JTextArea(); editor.setTabSize(2); output = new JTextArea();
    JMenuBar bar = new JMenuBar();
    JMenu file = new JMenu("File"); blank = new JMenuItem("New");
    blank.addActionListener(this); file.add(blank);
    open = new JMenuItem("Open..."); open.addActionListener(this); file.add(open);
    save = new JMenuItem("Save..."); save.addActionListener(this); file.add(save);

    JMenu build = new JMenu("Build");
    compile = new JMenuItem("Compile"); compile.setEnabled(false);
    compile.addActionListener(this);
    build.add(compile);
    run = new JMenuItem("Run"); run.setEnabled(false); run.addActionListener(this);
    build.add(run);

    bar.add(file); bar.add(build); setJMenuBar(bar);

    JSplitPane jsp = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
    jsp.setTopComponent(new JScrollPane(editor));
    jsp.setBottomComponent(new JScrollPane(output));
    jsp.setDividerLocation(270);
    getContentPane().add(jsp, BorderLayout.CENTER);

    setVisible(true);
}
```

```
public void actionPerformed (ActionEvent e) {
   Object o = e.getSource();
   if(o == blank) { fileName = null; editor.setText("");
      compile.setEnabled(false); run.setEnabled(false);
   } else if ( o == open ) {
      if ( fd == null ) fd = new JFileChooser();
      int returnVal = fd.showOpenDialog(this);
      if(returnVal == JFileChooser.APPROVE_OPTION) {
        File file = fd.getSelectedFile();
        fileName = file.getName();
        pwd = file.getParentFile();
        String path = file.getPath();
        setTitle("Java 개발 툴 - [" + fileName + "]");
        editor.setText("");
        try {
           BufferedReader in = new BufferedReader(new FileReader(path));
           char buf[] = new char[1024];
           int n = 0;
           while((n = in.read(buf)) != -1) { editor.append(new String(buf, 0, n)); }
           in.close();
        } catch(Exception ex) {  if(debug) ex.printStackTrace(); }
      }
      compile.setEnabled(true); run.setEnabled(true);
   }
```

```
else if ( o == save ) {
    if(fd == null) fd = new JFileChooser();
    int returnVal = fd.showSaveDialog(this);
    if(returnVal == JFileChooser.APPROVE_OPTION) {
      File file = fd.getSelectedFile();
      fileName = file.getName();
      pwd = file.getParentFile();
      String path = file.getPath();
      setTitle("Java 개발 툴 - [" + fileName + "]");
      try {
         PrintWriter out = new PrintWriter(new FileWriter(path));
         String source = editor.getText();
         out.println(source);
         out.close();
      } catch(Exception ex) {
         if(debug) ex.printStackTrace();}
    }
    compile.setEnabled(true);
    run.setEnabled(true);
}
```

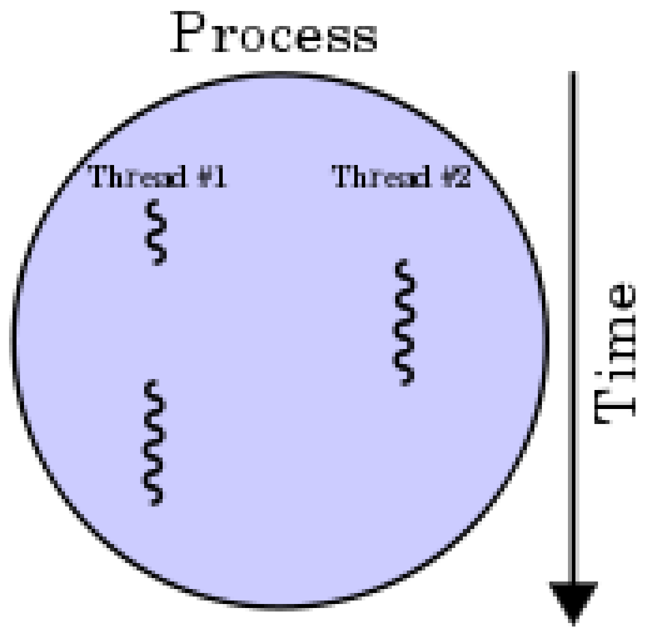
```
else if ( o == compile ) {
      String cmd = javac + "  " + fileName;
      output.setText(cmd + "₩n");
      try {
        Runtime rt = Runtime.getRuntime();
        Process ps = rt.exec(cmd, null, pwd);
        InputStream in = ps.getErrorStream(); // to read the error from the javac
        byte buf[] = new byte[1024];
        int n = 0;
        boolean hasError = false;
        while((n = in.read(buf)) != -1) {
           output.append(new String(buf, 0, n));
           hasError = true;
        }
        if ( !hasError ) output.append("No Error₩n");
      } catch( Exception ex ) {
        if( debug ) ex.printStackTrace();
      }
   }
```

```
 else if ( o == run ) {
        int index = fileName.lastIndexOf(".");
        String className = fileName.substring(0, index);
        String cmd = java + "  " + className;
        output.setText(cmd + "₩n");
        try {
          Runtime rt = Runtime.getRuntime();
          Process ps = rt.exec(cmd, null, pwd);
          InputStream in = ps.getInputStream();
          byte buf[] = new byte[1024];
          int n = 0;
          while((n = in.read(buf)) != -1) {
             output.append(new String(buf, 0, n));
          }
          in.close();
          in = ps.getErrorStream();
          while((n = in.read(buf)) != -1) {
             output.append(new String(buf, 0, n));
          }
          in.close();
        } catch(Exception ex) { if (debug) ex.printStackTrace(); }
    }
}
```

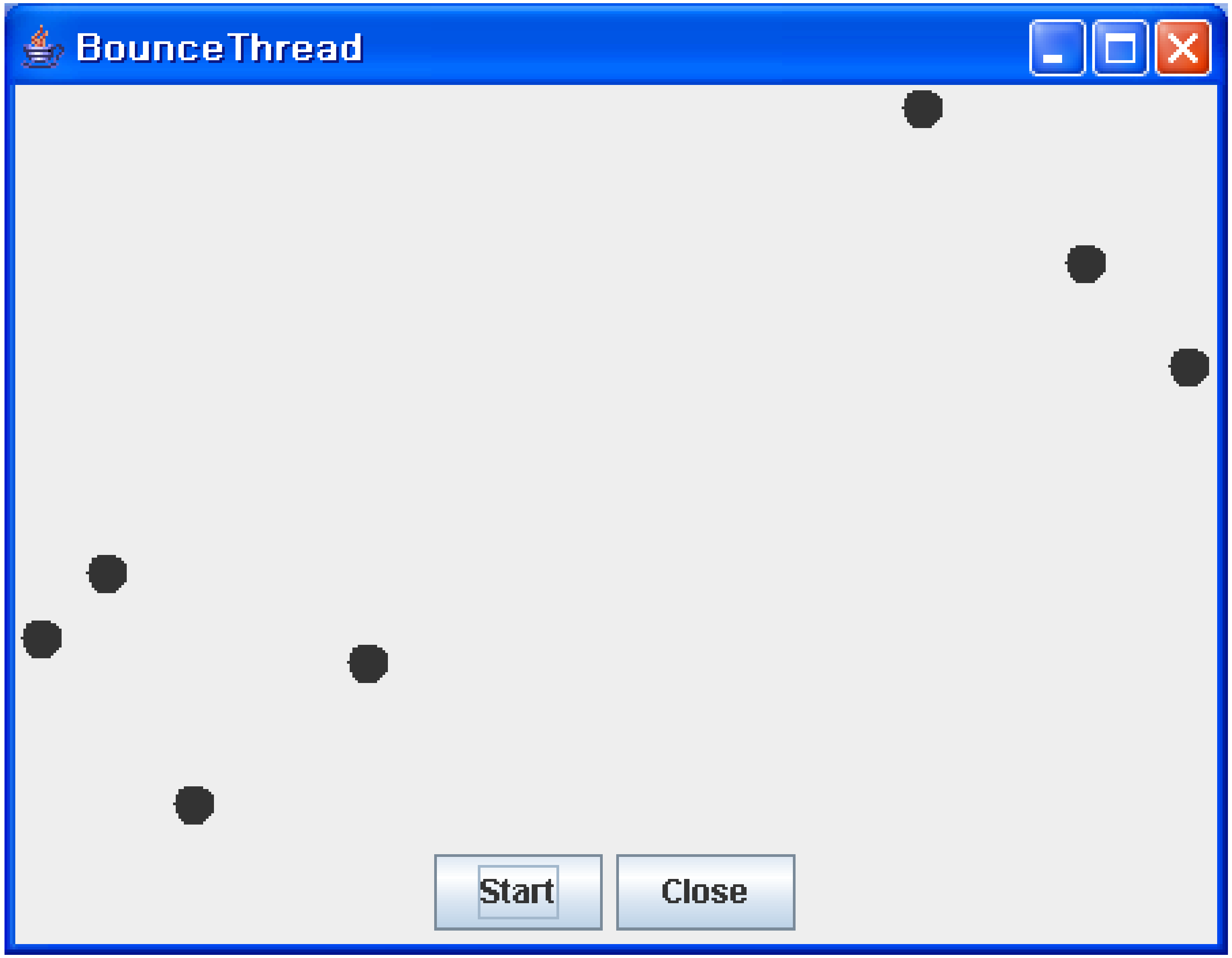
```
   public static void main(String args[]) {
      try {
         new IDE();
      } catch(Exception e) {
         System.out.println(e);
      }
   }
}
```

Thread

- Basically, threads is like processes.
- Threads or processes support concurrent programming.
- In Java, threads are mainly used to implement concurrent programs.
- Thread is a lightweight process.
- A process can consist of multiple threads



Animating Bouncing Balls



Without Threads

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

/**
   Shows an animated bouncing ball.
*/
public class Bounce
{
   public static void main(String[] args)
   {
      JFrame frame = new BounceFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

```
/**
   A ball that moves and bounces off the edges of a rectangle
*/
class Ball {
   /**
      Moves the ball to the next position, reversing direction if it hits one of the edges
   */
   public void move(Rectangle2D bounds) { // java.awt.geom.Rectangle2D
      x += dx; y += dy;
      if (x < bounds.getMinX()) {  x = bounds.getMinX(); dx = -dx; }
      if (x + XSIZE >= bounds.getMaxX()) { x = bounds.getMaxX() - XSIZE;  dx= -dx;  }
      if (y < bounds.getMinY()) { y = bounds.getMinY();  dy = -dy; }
      if (y + YSIZE >= bounds.getMaxY()) { y = bounds.getMaxY() - YSIZE; dy = -dy; }
   }
   /**
      Gets the shape of the ball at its current position.
   */
   public Ellipse2D getShape() { return new Ellipse2D.Double(x, y, XSIZE, YSIZE); }

   private static final int XSIZE = 15;
   private static final int YSIZE = 15;
   private double x = 0;
   private double y = 0;
   private double dx = 1;
   private double dy = 1;
}
```

```
/**
  The panel that draws the balls.
*/
class BallPanel extends JPanel {
  /**
    Add a ball to the panel.
    @param b the ball to add
  */
  public void add (Ball b) {
    balls.add(b);
  }
  // overriding Jcomponent.paintComponent
  public void paintComponent (Graphics g) { // public abstract class Graphics
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g; // public abstract class Graphics2D extends Graphics
    for (Ball b : balls)
    {
      g2.fill(b.getShape()); // Actual drawing occurs here
    }
  }

  private ArrayList<Ball> balls = new ArrayList<Ball>();
}
```

```
class BounceFrame extends JFrame {
   public BounceFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setTitle("Bounce");

      panel = new BallPanel(); add(panel, BorderLayout.CENTER);

      JPanel buttonPanel = new JPanel();
      addButton(buttonPanel, "Start", new ActionListener() {
            public void actionPerformed(ActionEvent event) { addBall(); }
         });

      addButton(buttonPanel, "Close", new ActionListener() {
            public void actionPerformed(ActionEvent event) { System.exit(0); }
         });

      add(buttonPanel, BorderLayout.SOUTH);
   }
   public void addButton(Container c, String title, ActionListener listener) {
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }
/**
      Adds a bouncing ball to the panel and makes it bounce 1,000 times.
   */
   public void addBall() {
      try {
         Ball ball = new Ball();
         panel.add(ball);
         for (int i = 1; i <= STEPS; i++) {
            ball.move(panel.getBounds());
            panel.paint(panel.getGraphics());
            Thread.sleep(DELAY);
         }
      } catch (InterruptedException e) { }
   }
   private BallPanel panel;
   public static final int DEFAULT_WIDTH = 450;
   public static final int DEFAULT_HEIGHT = 350;
   public static final int STEPS = 1000;
   public static final int DELAY = 3;
}
```

Before the completion of 1000 movements, another ball cannot be created !

Problems with the current program

- You cannot create a new ball before the current ball stops.
- Why ?
  - The reason is that the only one thread is moving the current ball.
  - After finishing the movement, creating a ball can start !
- What's a solution ?
  - To move each ball concurrently, separate thread for each ball is necessary !
  - Try the Bounce with thread

With Threads

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

public class BounceThread {
  public static void main(String[] args) {
    JFrame frame = new BounceFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

```
/**
   A runnable that animates a bouncing ball.
*/
class BallRunnable implements Runnable {
   public BallRunnable(Ball aBall, Component aComponent) {
      ball = aBall;  component = aComponent;
   }
   public void run() {
      try {
         for (int i = 1; i <= STEPS; i++) {
            ball.move(component.getBounds()); // update the location of the ball
            component.repaint(); // redraw the panel
            Thread.sleep(DELAY);
         }
      } catch (InterruptedException e) { }
   }
   private Ball ball;
   private Component component;
   public static final int STEPS = 1000;
   public static final int DELAY = 5;
}
```

```
/**
   A ball that moves and bounces off the edges of a rectangle
*/
class Ball {
   /**
      Moves the ball to the next position, reversing direction if it hits one of the edges
   */
   public void move(Rectangle2D bounds) { // java.awt.geom.Rectangle2D
      x += dx; y += dy;
      if (x < bounds.getMinX()) {  x = bounds.getMinX(); dx = -dx; }
      if (x + XSIZE >= bounds.getMaxX()) { x = bounds.getMaxX() - XSIZE;  dx= -dx;  }
      if (y < bounds.getMinY()) { y = bounds.getMinY();  dy = -dy; }
      if (y + YSIZE >= bounds.getMaxY()) { y = bounds.getMaxY() - YSIZE; dy = -dy; }
   }
   /**
      Gets the shape of the ball at its current position.
   */
   public Ellipse2D getShape() { return new Ellipse2D.Double(x, y, XSIZE, YSIZE); }

   private static final int XSIZE = 15;
   private static final int YSIZE = 15;
   private double x = 0;
   private double y = 0;
   private double dx = 1;
   private double dy = 1;
}
```

```
/**
   The panel that draws the balls.
*/
class BallPanel extends JPanel
{
   /**
      Add a ball to the panel.
      @param b the ball to add
   */
   public void add(Ball b) {
      balls.add(b);
   }

   public void paintComponent (Graphics g) {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      for (Ball b : balls) { g2.fill(b.getShape()); }
   }
   private ArrayList<Ball> balls = new ArrayList<Ball>();
}
```

```
class BounceFrame extends JFrame {
   public BounceFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setTitle("BounceThread");

      panel = new BallPanel();
      add(panel, BorderLayout.CENTER);
      JPanel buttonPanel = new JPanel();
      addButton(buttonPanel, "Start", new ActionListener() {
            public void actionPerformed(ActionEvent event) { addBall(); }
         });

      addButton(buttonPanel, "Close", new ActionListener() {
            public void actionPerformed(ActionEvent event) { System.exit(0); }
         });
      add(buttonPanel, BorderLayout.SOUTH);
   }
   public void addButton(Container c, String title, ActionListener listener) {
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }
/**
   Adds a bouncing ball to the canvas and starts a thread to make it bounce
*/
public void addBall() {
   Ball b = new Ball();
   panel.add(b);
   Runnable r = new BallRunnable(b, panel);
   Thread t = new Thread(r);
   t.start();
}


private BallPanel panel;
public static final int DEFAULT_WIDTH = 450;
public static final int DEFAULT_HEIGHT = 350;
public static final int STEPS = 1000;
public static final int DELAY = 3;
}
```

Whenever addBall() is called, that is, whenever "start" button is clicked, separate thread for each ball is created !

Because separate thread can move each ball, the main thread can process "start" button.

Two Methods for Creating Threads

- Method #1

```
class MyRunnable implements Runnable {
  public void run() {
    // task code
  }
}
…
Runnable r = new MyRunnable() ;
Thread t = new Thread(r) ;
t.start() ;
```

- Method #2

```
class MyThread extends Thread {
  public void run() {
    // task code
  }
}
…
MyThread t = new MyThread() ;
t.start() ;
```

Pausing Execution with Sleep

- Thread.sleep causes the current thread to suspend execution for a specified period.

```
public class SleepMessages {
    public static void main(String args[]) throws InterruptedException {
        String importantInfo[] = {
            "1st message", "2nd message", "3rd message", "4th message"
        } ;
        for (int i = 0; i < importantInfo.length; i++) {
            // Pause for 4 seconds; but not guaranteed !
            Thread.sleep(4000);
            // Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

Interrupts

- An *interrupt* is an indication to a thread that it should stop what it is doing and do something else.
  - Thread.**interrupt**()
- It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate

```
public class InterruptThread {

  private static class SimpleRunnable implements Runnable {
    public void run() {
      String threadName = Thread.currentThread().getName();
      int i = 0 ;
      while ( true ) { // the loop never stops !
        System.out.printf("%s: %d%n", threadName, i) ;
        i ++ ;
      }
    }
  }

  public static void main(String[] args) {
    Thread thread = new Thread(new SimpleRunnable()) ;
    thread.start();
    Scanner scanner = new Scanner(System.in) ;
    scanner.next() ;
    thread.interrupt() ; // The thread is now interrupted !
  }
}
```

```
Thread-0: 0
Thread-0: 1
Thread-0: 2
Thread-0: 3
Thread-0: 4
abc
Thread Terminated by Interrupt
```

Supporting Interrupts

- How does a thread support its own interruption? That is, how does the thread recognize that it has been interrupted !
- Method #1: Catch InterruptedException

```
while ( true ) {
  System.out.printf("%s: %d%n", threadName, i) ;
  i ++ ;
  try {
    // sleep method throw InterruptedException when interrupted
    Thread.sleep(1000) ;
  } catch (InterruptedException e) {
    System.out.println("Thread Terminated by Interrupt") ;
    break ;
  }
}
```

Supporting Interrupts

- Method #2
  - What if a thread goes a long time without invoking a method that throws InterruptedException?
  - Then it must periodically invoke **Thread.interrupted()**, which returns true if an interrupt has been received

```
while ( true ) {
   System.out.printf("%s: %d%n", threadName, i) ;
   i ++ ;
   if ( Thread.interrupted() )  {
      System.out.println("Thread Terminated by Interrupt") ;
      break ;
   }
}
```

```
public class InterruptThread {
  private static class SimpleRunnable implements Runnable {
    public void run() {
      String threadName = Thread.currentThread().getName();
      int i = 0 ;
      while ( true ) { // the loop can now stop !
        System.out.printf("%s: %d%n", threadName, i) ; i ++ ;
        /* // Method 1
        try { Thread.sleep(1000) ; }
        catch (InterruptedException e) {
          System.out.println("Thread Terminated by Interrupt") ;
          break ;
        }
        */
        if ( Thread.interrupted() ) { // Method 2
          System.out.println("Thread Terminated by Interrupt") ; break ;
        }
      }
    }
  }
  public static void main(String[] args) {
    Thread thread = new Thread(new SimpleRunnable()) ;
    thread.start();
    Scanner scanner = new Scanner(System.in) ; scanner.next() ;
    thread.interrupt() ; // The thread is now interrupted !
  }
}
```

Join

- The join method allows one thread to wait for the completion of another.
- If t is a Thread object whose thread is currently executing,
  - t.join();
  - causes the current thread to pause execution until t's thread terminates

```
public static void main(String[] args) throws InterruptedException {
    Thread thread = new Thread(new SimpleRunnable()) ;
    thread.start();
    //Wait maximum of 1 second for SimpleRunnable thread to finish.
    thread.join(1000);
    if (thread.isAlive()) { thread.join(2000); }
    ...
}
```

```
public class JoinInterrupt {
    //Display a message, preceded by the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    private static class SimpleRunnable implements Runnable {
        public void run() {
            String threadName = Thread.currentThread().getName();
            int i = 0 ;
            while ( true ) {
                System.out.printf("%s: %d%n", threadName, i) ;
                i ++ ;
                try { Thread.sleep(1000) ; }
                catch (InterruptedException e) {
                    threadMessage("Terminated by Interrupt") ; break ;
                }
            }
            threadMessage("End");
        }
    }
```

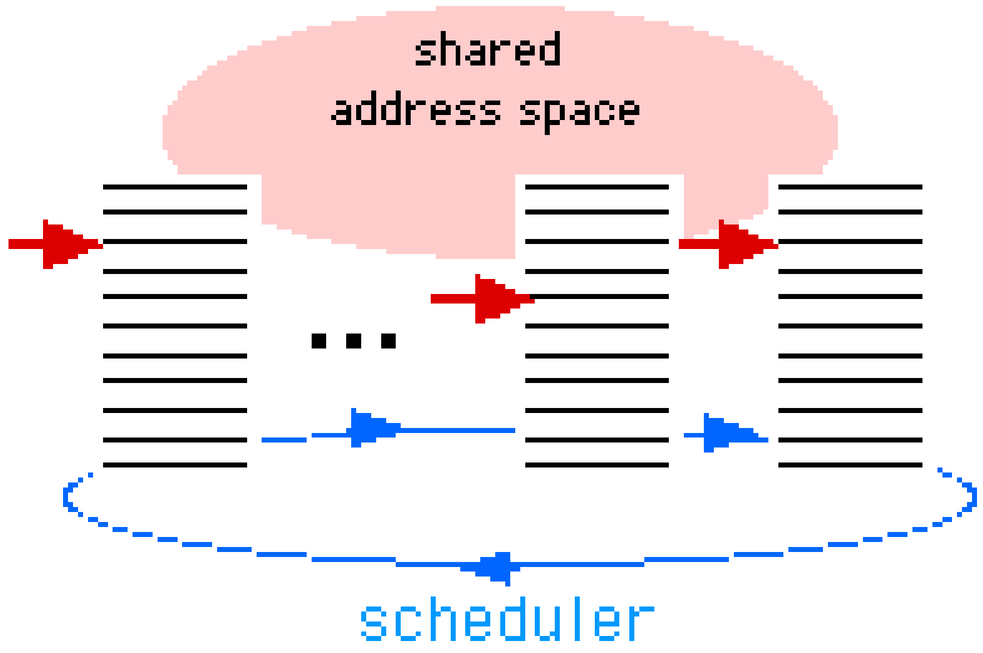
```java
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new SimpleRunnable()) ; thread.start();

        int waitingCount = 0 ;
        while (thread.isAlive()) {
            threadMessage("Still waiting...");
            thread.join(1000); //Wait maximum of 1 second for SimpleRunnable to finish.
            waitingCount ++ ;
            if ( waitingCount == 5 && thread.isAlive()) {
                threadMessage("Time is up!. It's time to interrupt " + thread.getName());
                thread.interrupt();
                thread.join(); // Shouldn't be long now -- wait indefinitely
            }
        }
        threadMessage("End!");
    }
}
```

```
main: Still waiting...
Thread-0: 0
Thread-0: 1
main: Still waiting...
Thread-0: 2
main: Still waiting...
Thread-0: 3
main: Still waiting...
Thread-0: 4
main: Still waiting...
Thread-0: 5
main: Time is up!. It's time to interrupt Thread-0
Thread-0: Terminated by Interrupt
Thread-0: End
main: End!
```

Thread

- All the threads in a process share the address space
- Therefore, some shared address spaces need to be protected from concurrent access; otherwise, they may be corrupted.



An example of race condition

```
public class UnsynchBankTest {
  public static void main(String[] args) {
    // A bank is created with NACCOUNTS accounts
    Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);

    for (int i = 0; i < NACCOUNTS; i++) {
      // A thread is created for each account
      TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
      Thread t = new Thread(r);
      t.start();
    }
  }

  public static final int NACCOUNTS = 100;
  public static final double INITIAL_BALANCE = 1000;
}
```

Several threads will work on the same bank because the reference to the Bank is delivered to the thread

```
class Bank {
    public Bank(int n, double initialBalance) {
       accounts = new double[n];
       for (int i = 0; i < accounts.length; i++) accounts[i] = initialBalance;
   }
    public void transfer (int from, int to, double amount) {
       // unsafe when called from multiple threads operates on the same account
       if (accounts[from] < amount) return;
       System.out.print(Thread.currentThread());
       accounts[from] -= amount;
       System.out.printf(" %10.2f from %d to %d", amount, from, to);
       accounts[to] += amount;
       System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
   }
    public double getTotalBalance() {
       double sum = 0;
       for (double a : accounts) sum += a;
       return sum;
   }
   public int size() { return accounts.length; }
   private final double[] accounts; // A bank has n accounts
}
```

shared data(accounts[]) can be corrupted by multiple threads

The total balance should always be 100 * 1,000 = 100,000

```
class TransferRunnable implements Runnable {
   public TransferRunnable(Bank b, int from, double max) {
      bank = b; // All the threads share the bank
      fromAccount = from;
      maxAmount = max;
   }
   public void run() {
      try {
         while ( true ) {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
         }
      } catch (InterruptedException e) {}
   }
   private Bank bank;
   private int fromAccount;
   private double maxAmount;
   private int DELAY = 10;
}
```

Several threads will work on the same accounts at the same time

```
Thread[Thread-0,5,main]     573.27 from 0 to 18Thread[Thread-
1,5,main]Thread[Thread-2,5,main]Thread[Thread-3,5,main]Thread[Thread-
4,5,main]Thread[Thread-5,5,main]Thread[Thread-6,5,main]Thread[Thread-
7,5,main]Thread[Thread-8,5,main]Thread[Thread-9,5,main]     869.03 from 1 to 28
470.70 from 2 to 30     330.73 from 3 to 41     969.38 from 4 to 92     573.76 from 5
to 23     452.03 from 6 to 10     952.24 from 7 to 0     755.73 from 8 to 84 Total
Balance:   94922.15
 Total Balance:   95392.86
 Total Balance:   95723.59
 Total Balance:   96692.96
 Total Balance:   97266.73
 Total Balance:   97718.75
 Total Balance:   98671.00
 Total Balance:   99426.73
     308.69 from 9 to 17 Total Balance:   99426.73
 Total Balance:  100000.00
Thread[Thread-10,5,main]     677.39 from 10 to 59Thread[Thread-2,5,main]     172.38
from 2 to 98 Total Balance:   99322.61
Thread[Thread-6,5,main]      53.02 from 6 to 66 Total Balance:   99322.61
Thread[Thread-3,5,main]     240.86 from 3 to 47 Total Balance:   99322.61
Thread[Thread-2,5,main]     221.04 from 2 to 62 Total Balance:   99322.61
Thread[Thread-0,5,main]     497.56 from 0 to 77 Total Balance:   99322.61
 Total Balance:  100000.00
```

Ideal Expected Situation

Thread for account 100

```
public void transfer (
  int from(=100),
  int to(=300),
  double amount(=500)) {
     accounts[100] -= 500;
     t1 = accounts[300] ;
     t1 += 500;
     accounts[300] = t1 ;
}
```

100	200	300	t1	t2	Sum
1000	1000	1000			3000
500					
			1000		2500
			1500		
		1500			3000
500	1000	1500			3000
	0				2000
				1500	
				2500	
		2500			3000

Thread for account 200

```
public void transfer (
  int from(=200),
  int to(=300),
  double amount(=1000)) {
     accounts[200] -= 1000;
     t2 = accounts[300] ;
     t2 += 1000;
     accounts[300] = t2 ;
}
```

Real Problematic Situation

Thread for account 100

```
public void transfer (
  int from(=100),
  int to(=300),
  double amount(=500)) {
     accounts[100] -= 500;
     t1 = accounts[300] ;


     t1 += 500;
     accounts[300] = t1 ;
}
```

100	200	300	t1	t2	Sum
1000	1000	1000			3000
500					
			1000		2500
	0				
				1000	
			1500		
		1500			
				2000	
		2000			
					2500

Thread for account 200

```
public void transfer (
  int from(=200),
  int to(=300),
  double amount(=1000)) {


     accounts[200] -= 1000;
     t2 = accounts[300] ;


     t2 += 1000;
     accounts[300] = t2 ;
}
```

The period between reading and writing on account should not be interrupted by other threads

Synchronization using Lock Objects

```
import java.util.concurrent.locks.*;
public class SynchBankTest {
  public static void main(String[] args) {
    Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
    for (int i = 0; i < NACCOUNTS; i++)
    {
      TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
      Thread t = new Thread(r);
      t.start();
    }
  }

  public static final int NACCOUNTS = 100;
  public static final double INITIAL_BALANCE = 1000;
}
```

```
class Bank {
  public Bank(int n, double initialBalance) {
    accounts = new double[n];
    for (int i = 0; i < accounts.length; i++) accounts[i] = initialBalance;
    bankLock = new ReentrantLock(); // use true for fairness
    // sufficientFunds = bankLock.newCondition();
  }
 public void transfer(int from, int to, double amount) throws InterruptedException {
    bankLock.lock();
    try {


      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f from %d to %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());

    }
    finally { bankLock.unlock(); }
  }
```

Critical section

Good ! Reentrant lock

```
public double getTotalBalance() {
    bankLock.lock();
    try {
      double sum = 0;
      for (double a : accounts) sum += a;
      return sum;
    }
    finally { bankLock.unlock(); }
  }
  public int size()  { return accounts.length; }

  private final double[] accounts;
  private Lock bankLock;
  // private Condition sufficientFunds;
}
```

Critical section

```
class TransferRunnable implements Runnable {
    public TransferRunnable(Bank b, int from, double max) {
      bank = b;
      fromAccount = from;
      maxAmount = max;
   }
   public void run() {
      try {
         while (true) {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
         }
      }
      catch (InterruptedException e) {}
   }
   private Bank bank;
   private int fromAccount;
   private double maxAmount;
   private int repetitions;
   private int DELAY = 10;
}
```

```
Thread[Thread-0,5,main]     749.07 from 0 to 49 Total Balance:  100000.00
Thread[Thread-1,5,main]     758.75 from 1 to 55 Total Balance:  100000.00
Thread[Thread-2,5,main]     498.47 from 2 to 66 Total Balance:  100000.00
Thread[Thread-3,5,main]     288.41 from 3 to 23 Total Balance:  100000.00
Thread[Thread-4,5,main]      91.94 from 4 to 57 Total Balance:  100000.00
Thread[Thread-5,5,main]     143.72 from 5 to 41 Total Balance:  100000.00
Thread[Thread-6,5,main]     507.47 from 6 to 83 Total Balance:  100000.00
Thread[Thread-7,5,main]     443.58 from 7 to 99 Total Balance:  100000.00
Thread[Thread-8,5,main]      20.96 from 8 to 79 Total Balance:  100000.00
Thread[Thread-3,5,main]     585.57 from 3 to 28 Total Balance:  100000.00
Thread[Thread-5,5,main]     782.21 from 5 to 39 Total Balance:  100000.00
Thread[Thread-0,5,main]     189.73 from 0 to 45 Total Balance:  100000.00
Thread[Thread-1,5,main]     205.57 from 1 to 52 Total Balance:  100000.00
Thread[Thread-4,5,main]     765.40 from 4 to 24 Total Balance:  100000.00
Thread[Thread-8,5,main]      30.21 from 8 to 99 Total Balance:  100000.00
Thread[Thread-9,5,main]     300.35 from 9 to 59 Total Balance:  100000.00
Thread[Thread-2,5,main]     201.73 from 2 to 80 Total Balance:  100000.00
Thread[Thread-9,5,main]     297.33 from 9 to 60 Total Balance:  100000.00
Thread[Thread-10,5,main]    653.55 from 10 to 22 Total Balance:  100000.00
Thread[Thread-11,5,main]    874.86 from 11 to 79 Total Balance:  100000.00
Thread[Thread-4,5,main]     108.56 from 4 to 96 Total Balance:  100000.00
Thread[Thread-8,5,main]     933.63 from 8 to 66 Total Balance:  100000.00
```

Why Need Condition Object?

- Now, what do we do when there is not enough money in the account?
- We wait until some other thread has added funds.
- But this thread has just gained exclusive access to the bankLock, so no other thread has a chance to make a deposit

```
public void transfer(int from, int to, int amount) {
 bankLock.lock();
 try {
   while (accounts[from] < amount) {
     // wait
     . . .
   }
   // transfer funds
   . . .
 }
 finally {
   bankLock.unlock();
 }
}
```

Condition Objects

- await()
  - The current thread is now deactivated and gives up the lock
  - it stays deactivated until another thread has called the signalAll method on the same condition
- signalAll()
  - When another thread has transferred money, it should call signalAll()

```
class Bank {
   public Bank(int n, double initialBalance) {
      bankLock = new ReentrantLock();
      sufficientFunds = bankLock.newCondition();
   }
  public void transfer(int from, int to, double amount) throws InterruptedException {
      bankLock.lock();
      try {
         while (accounts[from] < amount) sufficientFunds.await();
            // The current thread is now deactivated and gives up the lock.
            // This lets in another thread that can, we hope,
            // increase the account balance
         sufficientFunds.signalAll(); // Wakes up all waiting threads
      }
      finally { bankLock.unlock(); }
   }
```

Condition Objects

```
class Bank {
  public Bank(int n, double initialBalance) {
    accounts = new double[n];
    for (int i = 0; i < accounts.length; i++) accounts[i] = initialBalance;
    bankLock = new ReentrantLock(); // use true for fairness
    sufficientFunds = bankLock.newCondition();
  }
 public void transfer(int from, int to, double amount) throws InterruptedException {
    bankLock.lock();
    try {
      while (accounts[from] < amount) sufficientFunds.await();
        // causes the current thread to wait until it is signalled or interrupted
      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f from %d to %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
      sufficientFunds.signalAll(); // Wakes up all waiting threads
    }
    finally { bankLock.unlock(); }
  }
```

BoundedBuffer

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x)
     throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length) notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally { lock.unlock(); }
   }
```

When it is full, the thread will block until a space becomes available

```
 public Object take() throws
InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length)
         takeptr = 0;
       --count;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
 }
```

Synchronization using synchronized method

```
public synchronized void method() {
    method body
}
```

```
public void method() {
    implicitLock.lock() ;
    try {
      method body ;
    }
    finally { implicitLock.unlock() ; }
}
```

```
public class SynchBankTest2 {
  public static void main(String[] args) {
    Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
    for (int i = 0; i < NACCOUNTS; i++) {
      TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
      Thread t = new Thread(r);
      t.start();
    }
  }

  public static final int NACCOUNTS = 100;
  public static final double INITIAL_BALANCE = 1000;
}
```

```
class Bank {
   public Bank(int n, double initialBalance) {
     accounts = new double[n];
     for (int i = 0; i < accounts.length; i++) accounts[i] = initialBalance;
  }
   public synchronized void transfer(int from, int to, double amount)
     throws InterruptedException {
     while (accounts[from] < amount)
       wait(); // equivalent to implicitCondition.await()
     System.out.print(Thread.currentThread());
     accounts[from] -= amount;
     System.out.printf(" %10.2f from %d to %d", amount, from, to);
     accounts[to] += amount;
     System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
     notifyAll(); // equivalent to implicitCondition.signalAll()
  }
   public synchronized double getTotalBalance() {
     double sum = 0;
     for (double a : accounts) sum += a;
     return sum;
  }
   public int size() { return accounts.length; }
   private final double[] accounts;
}
```

```
class TransferRunnable implements Runnable {
    public TransferRunnable(Bank b, int from, double max) {
      bank = b;
      fromAccount = from;
      maxAmount = max;
   }
   public void run() {
      try {
         while (true) {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
         }
      }
      catch (InterruptedException e) {}
   }

   private Bank bank;
   private int fromAccount;
   private double maxAmount;
   private int repetitions;
   private int DELAY = 10;
}
```

Concurrent Implementation of Collection Classes

- Package **java.util.concurrent** offers concurrent versions of data structures

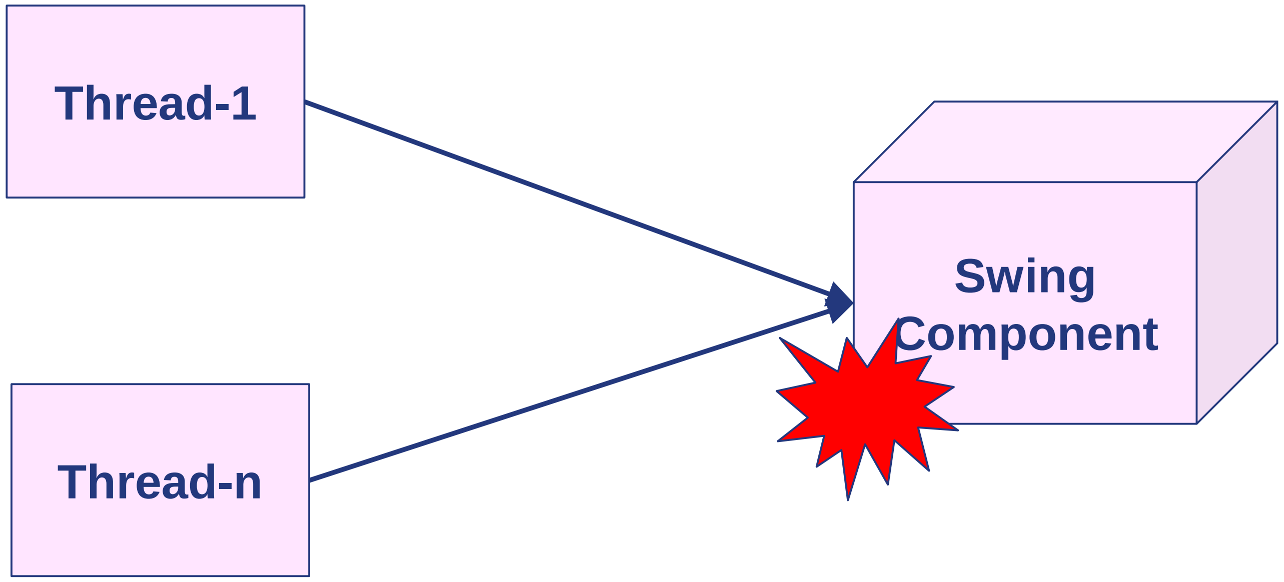
Interface	Implementation
Queue	**ConcurrentLinkedQueue** - An unbounded thread-safe queue based on linked nodes
BlockingQueue	**LinkedBlockingQueue** — an optionally bounded FIFO blocking queue backed by linked nodes **ArrayBlockingQueue** — a bounded FIFO blocking queue backed by an array **PriorityBlockingQueue** — an unbounded blocking priority queue backed by a heap **DelayQueue** — a time-based scheduling queue backed by a heap **SynchronousQueue** — a simple rendezvous mechanism that uses the BlockingQueue interface
BlockingDeque	**LinkedBlockingDeque** - An optionally-bounded blocking deque based on linked nodes
ConcurrentMap	**ConcurrentHashMap** - a highly concurrent, high-performance implementation backed up by a hash table

Interface: Blocking Queue

	Throws exception	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

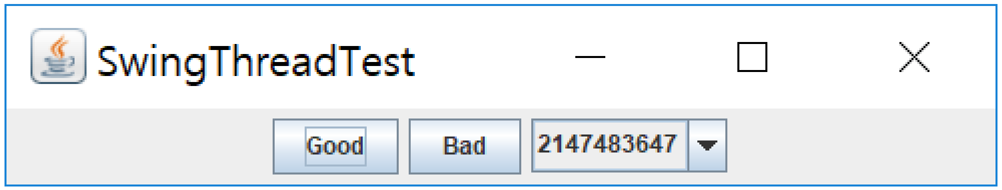
Threads and Swing

- Swing is not thread safe.
- If you try to manipulate user interface elements from multiple threads, your user interface can become corrupted.



Threads and Swing

```
public class SwingThreadTest {
 public static void main(String[] args) {
   EventQueue.invokeLater(() -> {
     JFrame frame = new SwingThreadFrame();
     frame.setTitle("SwingThreadTest");
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.setVisible(true);
   });
 }
}
```



```
class SwingThreadFrame extends JFrame {
 public SwingThreadFrame() {
   final JComboBox<Integer> combo = new JComboBox<>();
   combo.insertItemAt(Integer.MAX_VALUE, 0);
   combo.setPrototypeDisplayValue(combo.getItemAt(0));
   combo.setSelectedIndex(0);
   combo.setSize(200,1500);

   JPanel panel = new JPanel();

   JButton goodButton = new JButton("Good");
   goodButton.addActionListener(event ->
     new Thread(new GoodWorkerRunnable(combo)).start());
   panel.add(goodButton);
   JButton badButton = new JButton("Bad");
   badButton.addActionListener(event ->
     new Thread(new BadWorkerRunnable(combo)).start());
   panel.add(badButton);

   panel.add(combo);
   add(panel);
   pack();
 }
}
```

```
class BadWorkerRunnable implements Runnable {
 private JComboBox<Integer> combo;
 private Random generator;

 public BadWorkerRunnable(JComboBox<Integer> aCombo) {
   combo = aCombo;
   generator = new Random();
 }
 public void run() {
   try {
     while (true) {
       int i = Math.abs(generator.nextInt());
       if ( i % 2 == 0 ) {
         combo.insertItemAt(i, 0);
       }
       else if ( combo.getItemCount() > 0 )
         combo.removeItemAt(i % combo.getItemCount());
       Thread.sleep(10);
     }
   }
   catch (InterruptedException e) { }
 }
}
```

Violate single-thread rule of Swing
Do not touch Swing components in any thread other than the event dispatch thread.

```
class GoodWorkerRunnable implements Runnable {
 private JComboBox<Integer> combo;
 private Random generator;
 public GoodWorkerRunnable(JComboBox<Integer> aCombo) {
   combo = aCombo; generator = new Random();
 }
 public void run() {
   try {
     while (true) {
       EventQueue.invokeLater( new Runnable() {
         public void run() {
            int i = Math.abs(generator.nextInt());
            if ( i % 2 == 1 ) {
              combo.insertItemAt(i, 0);
            }
            else if ( combo.getItemCount() > 0 )
              combo.removeItemAt(i % combo.getItemCount());
         }
       }
       );
       Thread.sleep(10);
     }
   }
   catch (InterruptedException e) { }
 }
}
```

Use EventQueue.invokeLater() to use event dispatch thread

References

- Java Tutorials on Concurrency
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html