CNT 4714: Enterprise Computing Spring 2010

Programming Multithreaded Applications in Java Part 1

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cnt4714/spr2010

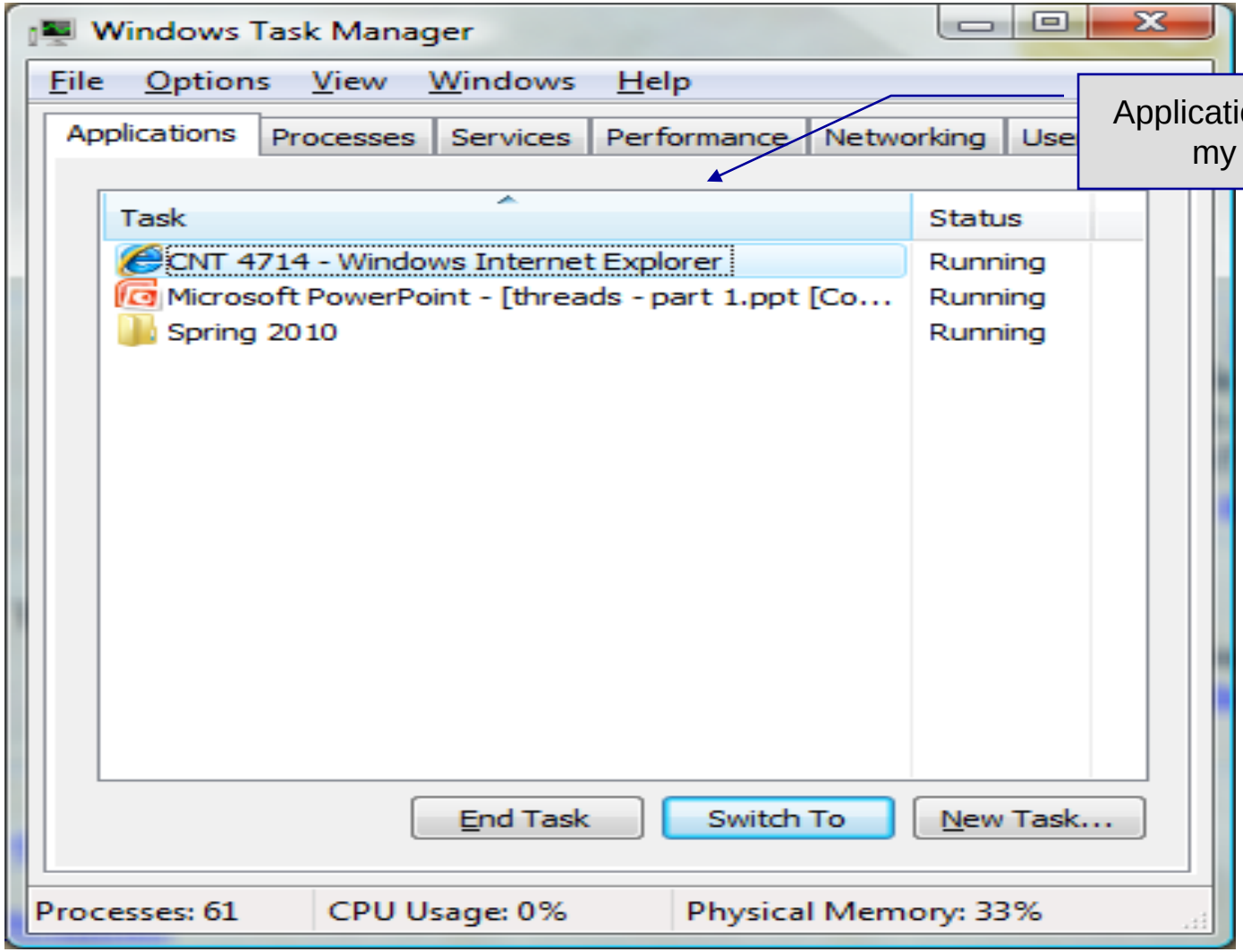
School of Electrical Engineering and Computer Science
University of Central Florida

Introduction to Threads in Java

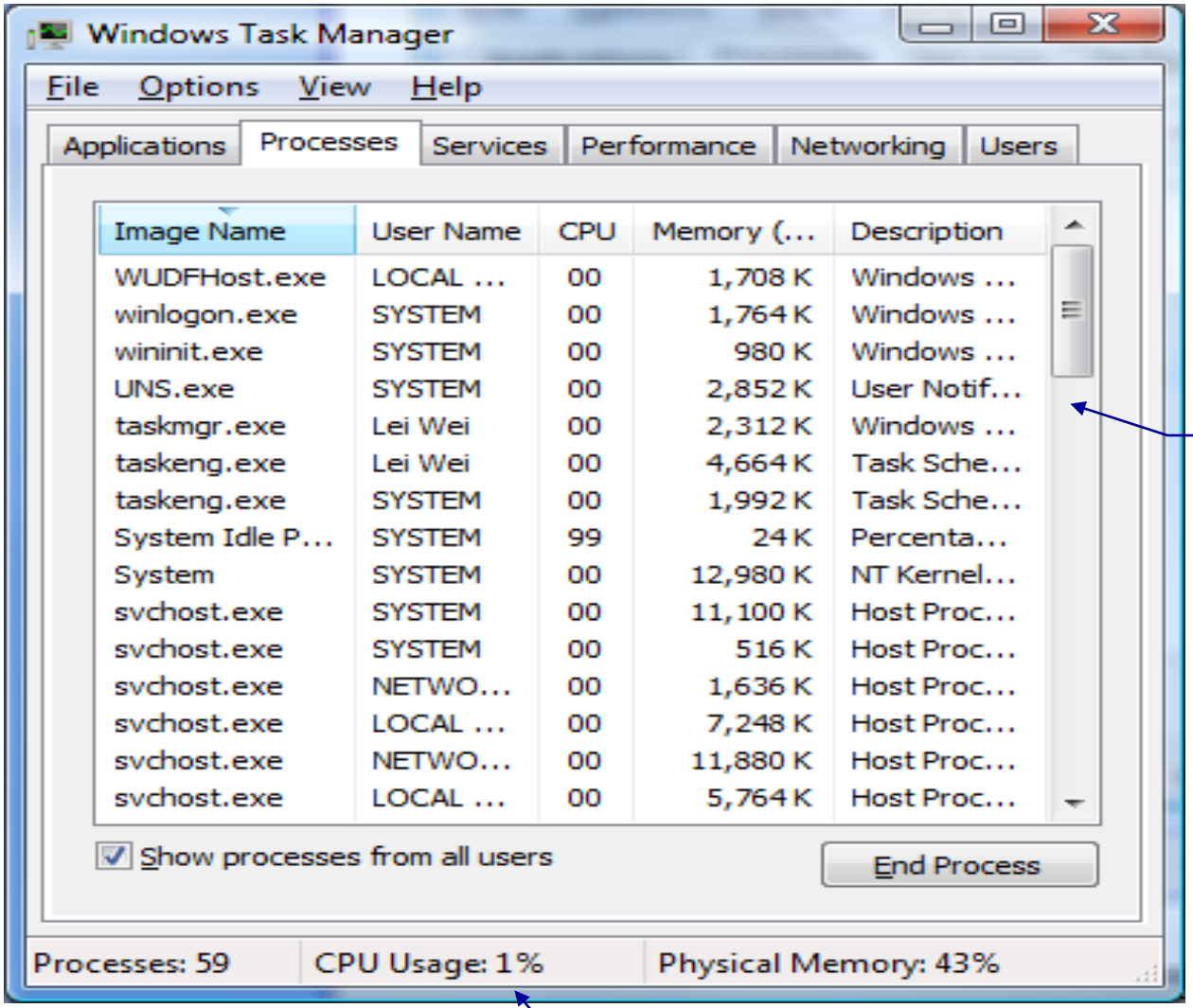
- In state-of-the art software, a program can be composed of multiple independent flows of control.
- A flow of control is more commonly referred to as a process or thread.
- In most of the Java programs that you've written (probably) there was a single flow of control. Most console-based programs begin with the first statement of the method `main()` and work forward to the last statement of the method `main()`. Flow of control is often temporarily passed to other methods through invocations, but the control returned to `main()` after their completion.
- Programs with a single control flow are known as sequential processes.

Introduction to Threads in Java (cont.)

- Java supports the creation of programs with concurrent flows of control. These independent flows of control are called threads.
- Threads run within a program and make use of that program's resources in their execution. For this reason threads are also called lightweight processes (LWP).
- The ability to run more than one process simultaneously is an important characteristic of modern OS such as Linux/Unix and Windows.
  - The following two pages show screen shots of a set of applications running on my office PC as well as the set of OS and applications processes required to run those applications.



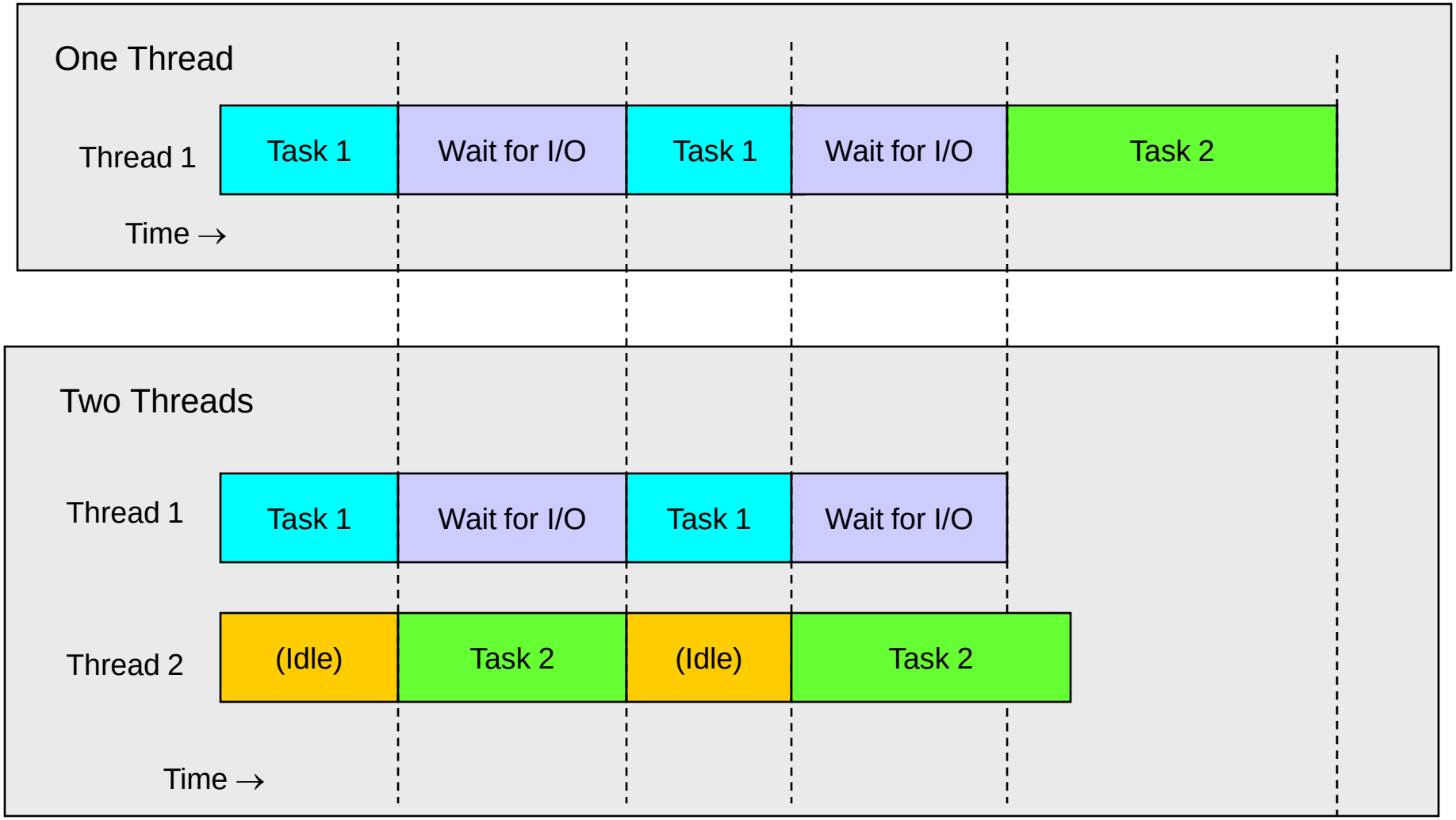
Applications running on my office PC



Some of the processes running the applications running on my office PC

CPU working hard!!!

Using Threads To Improve Performance



Improving Performance With Multithreading

- As the diagram on the previous page implies, applications that perform several tasks which are not dependent on one another will benefit the most from multithreading.
- For example, in the previous diagram, Task 2 can only be overlapped with Task 1 if Task 2 doesn't depend on the results of Task 1.
- However, some overlap between the two tasks may still be possible even if Task 2 depends on the results of Task 1. In this case the two tasks must communicate so that they can coordinate their operations.

Improving Performance With Multithreading
(cont.)

- Writing multithreaded programs can be tricky and complicated, particularly when synchronization between threads is required.
- Although the human mind can perform many functions concurrently, people find it difficult to jump between parallel trains of thought.
- To see why multithreading can be difficult to program and understand, try the experiment shown on the following page.

Multithreading Experiment

In this chapter, we introduce Swing components that enable developers to build functionally rich user interfaces.

Page 1

The Swing graphical interface components were introduced with the Java Foundation Classes (JFC) as a downloadable extension to the Java 1.1 Platform, then became a standard extension with the Java 2 Platform.

Page 2

Swing provides a more complete set of GUI components than the Abstract Windowing Toolkit (AWT), including advanced features such as a pluggable look and feel, lightweight component rendering and drag-and-drop capabilities.

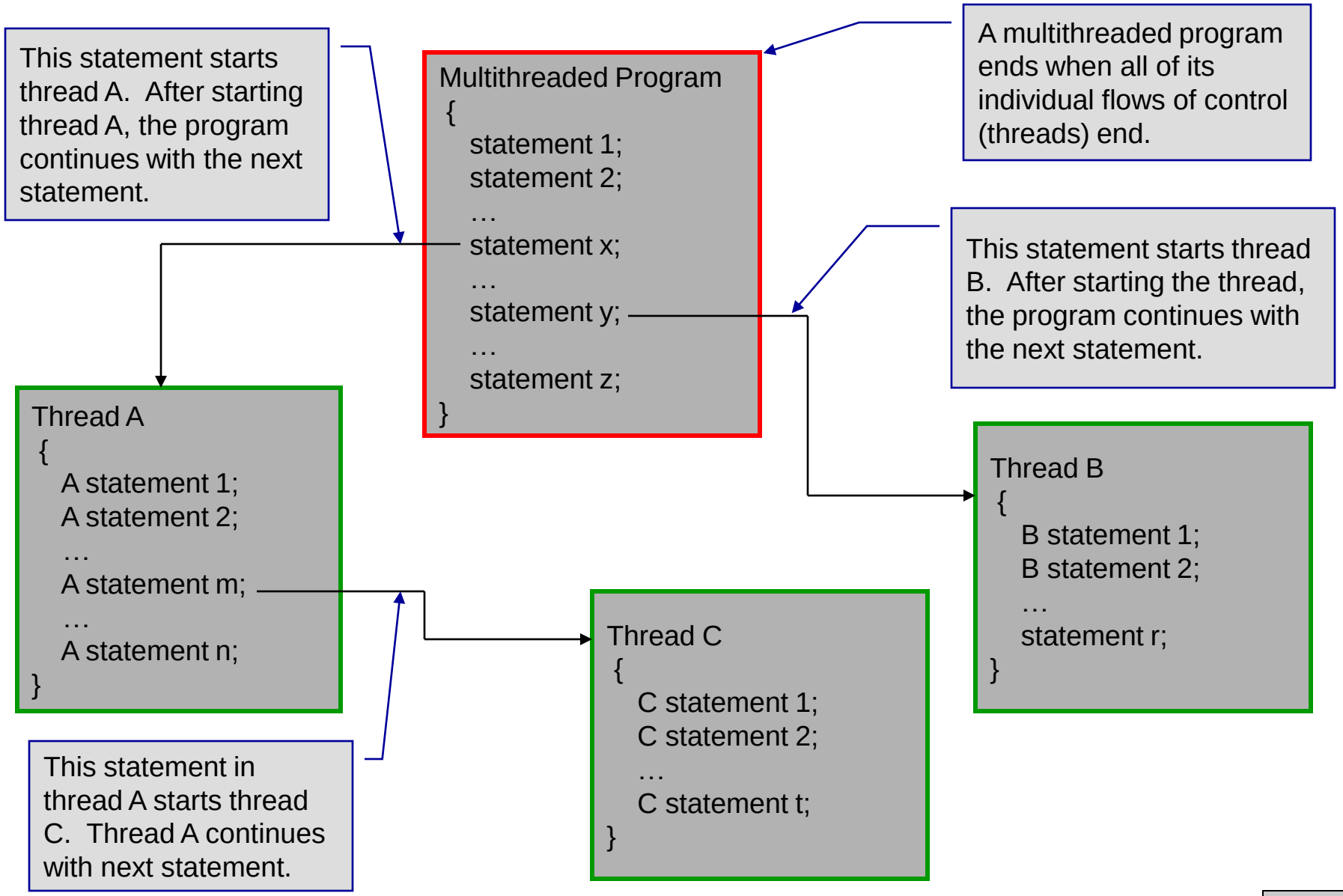
Page 3

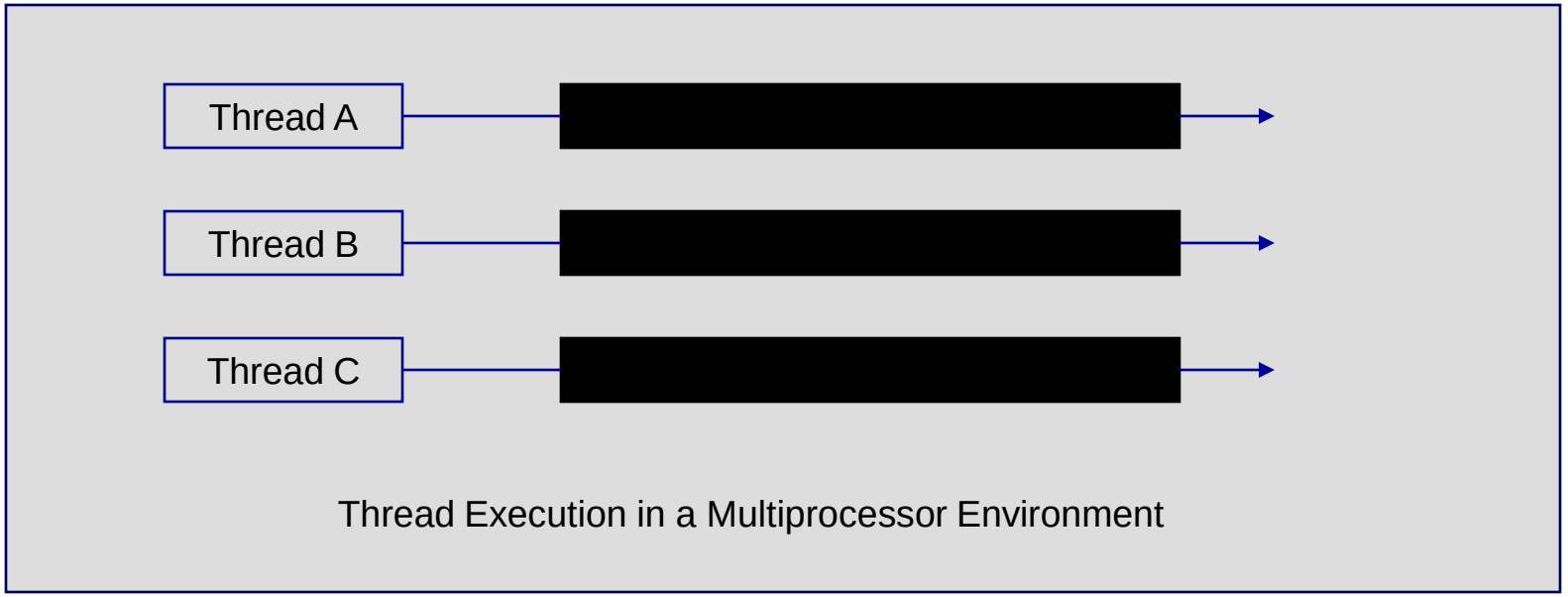
The experiment: Try reading the pages above concurrently by reading a few words from the first page, then a few words from the second page, then a few words from the third page, then loop back and read a few words from the first page, and so on. Does anything make sense? Can you construct a single sentence from what you have read? Can you remember on which page a particular word appeared? Can you even remember when you get back to the first page where you left off?

Typical Multithreaded Applications

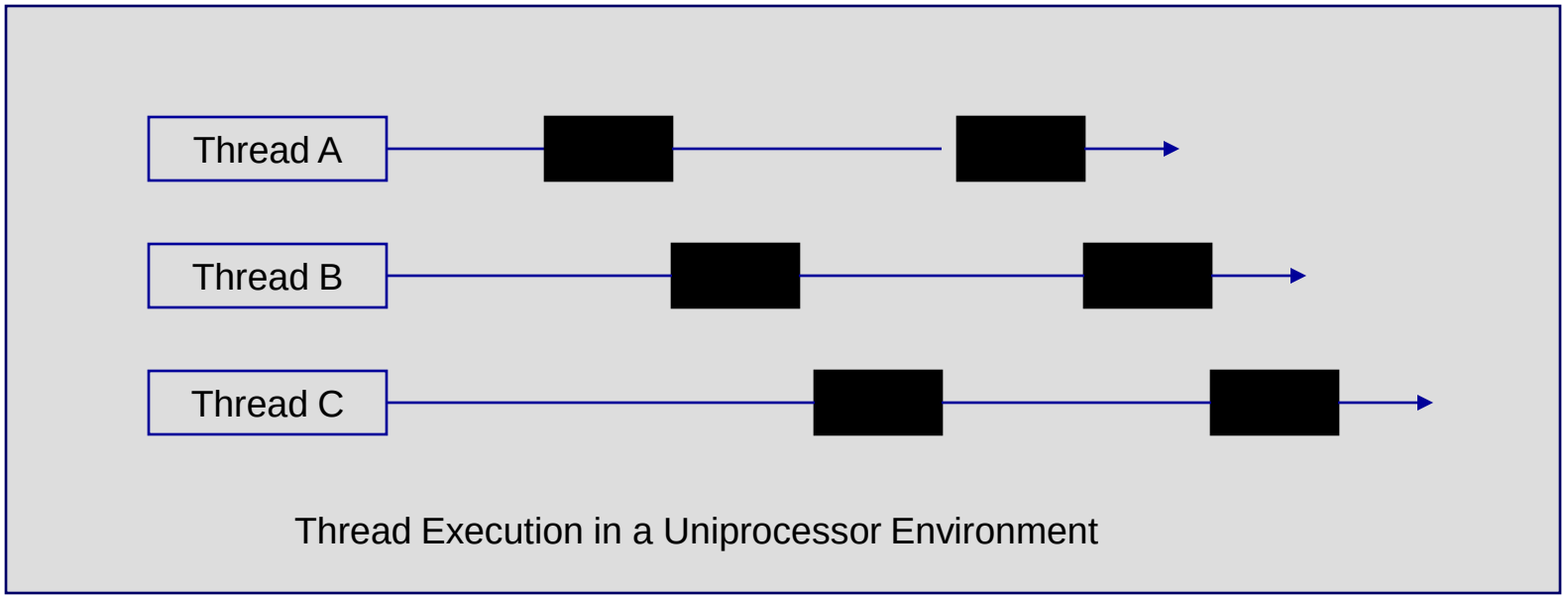
- Used to improve the performance of applications which require extensive I/O operations.
- Useful in improving the responsiveness of GUI-based applications.
- Used when two or more clients need to run server-based applications simultaneously.

Note: on a single CPU machine, threads don't actually execute simultaneously. Part of the JVM known as the thread scheduler time-slices threads which are runnable (we'll see more of this in a bit) giving the illusion of simultaneous execution.


This statement starts thread A. After starting thread A, the program continues with the next statement.
Multithreaded Program
{
statement 1;
statement 2;
…
statement x;
…
statement y;
…
statement z;
}
A multithreaded program ends when all of its individual flows of control (threads) end.
This statement starts thread B. After starting the thread, the program continues with the next statement.
Thread A
{
A statement 1;
A statement 2;
…
A statement m;
…
A statement n;
}
Thread B
{
B statement 1;
B statement 2;
…
statement r;
}
Thread C
{
C statement 1;
C statement 2;
…
C statement t;
}
This statement in thread A starts thread C. Thread A continues with next statement.

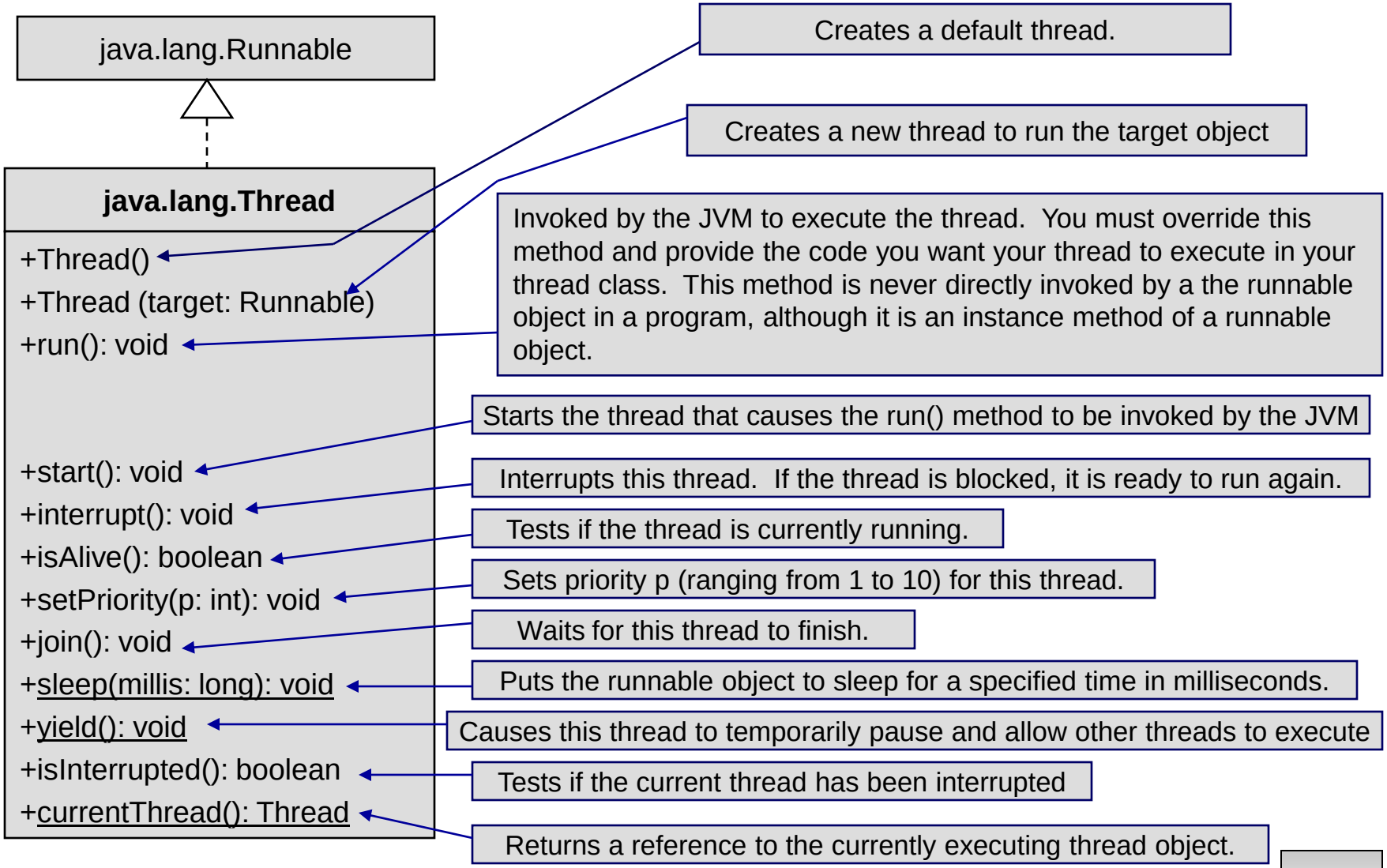


Thread Execution in a Multiprocessor Environment



Thread Execution in a Uniprocessor Environment

The Java Thread Class



Java Classes and Threads

- Java has several classes that support the creation and scheduling of threads.
- The two basic ways of creating threads in Java are:
  - 1) extending the `Thread` class
  - or 2) implementing the `Runnable` interface.

    (Both are found in package `java.lang`. `Thread` actually implements `Runnable`.)
- We'll also look at a slightly different technique for creating and scheduling threads later using the `java.util.Timer` and `java.util.TimerTask` classes.

Java Classes and Threads (cont.)

- The following two simple examples, illustrate the differences in creating threads using these two different techniques.
- The example is simple, three threads are created, one that prints the character 'A' twenty times, one that prints the character 'B' twenty times, and a third thread that prints the integer numbers from 1 to 20.
- The first program is an example of extending the thread class. The second program is an example of using the `Runnable` interface. This latter technique is the more common and preferred technique. While we will see more examples of this technique later, this simple example will illustrate the difference in the two techniques.

```
//Custom Thread Class
Public class CustomThread extends Thread
{ …
    public CustomThread(…)
    {
        …
    }
    //Override the run method in Thread
    //Tell system how to run custom thread
    public void run( )
    {
       …
    }
    …
} //end CustomThread Class
```

```
//Client Class to utilize CustomThread
Public class Client
{ …
    public void someMethod( )
    {
        …
        //create a thread
        CustomThread thread1 =
                new CustomThread(…);
        //start a thread
        thread1.start( );
        …

        //create another thread
        CustomThread thread2 =
                new CustomThread(…);
         //start another thread
         thread2.start( );
         …
    }
…
} //end Client Class
```

Template for defining a thread class by extending the Thread class. Threads thread1 and thread2 are runnable objects created from the CustomThread class. The start method informs the system that the thread is ready to run.

```
//Custom Thread Class
Public class CustomThread implements Runnable
{ …
    public CustomThread(…)
    {
        …
    }
    //Implement the run method in Runnable
    //Tell system how to run custom thread
    public void run( )
    {
       …
    }
    …
} //end CustomThread Class
```

```
//Client Class to utilize CustomThread
Public class Client
{ …
    public void someMethod( )
    {
        …
        //create an instance of CustomThread
        CustomThread custhread =
                new CustomThread(…);
        …
        //create a thread
        Thread thread =
                newThread(custhread);
        …
        //start a thread
        thread.start( );
        …
    }
…
} //end Client Class
```

Template for defining a thread class by implementing the Runnable interface. To start a new thread with the Runnable interface, you must first create an instance of the class that implements the Runnable interface (in this case custhread), then use the Thread class constructor to construct a thread.

```
//Class to generate threads by extending the Thread class
public class TestThread {
  // Main method
  public static void main(String[] args) {
    // Create threads
    PrintChar printA = new PrintChar('a', 20);
    PrintChar printB = new PrintChar('b', 20);
    PrintNum  print20 = new PrintNum(20);

    // Start threads
    print20.start();
    printA.start();
    printB.start();
  }
}

// The thread class for printing a specified character a specified number of times
class PrintChar extends Thread {
  private char charToPrint;  // The character to print
  private int times;  // The times to repeat

  // Construct a thread with specified character and number of times to print the character
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }
```

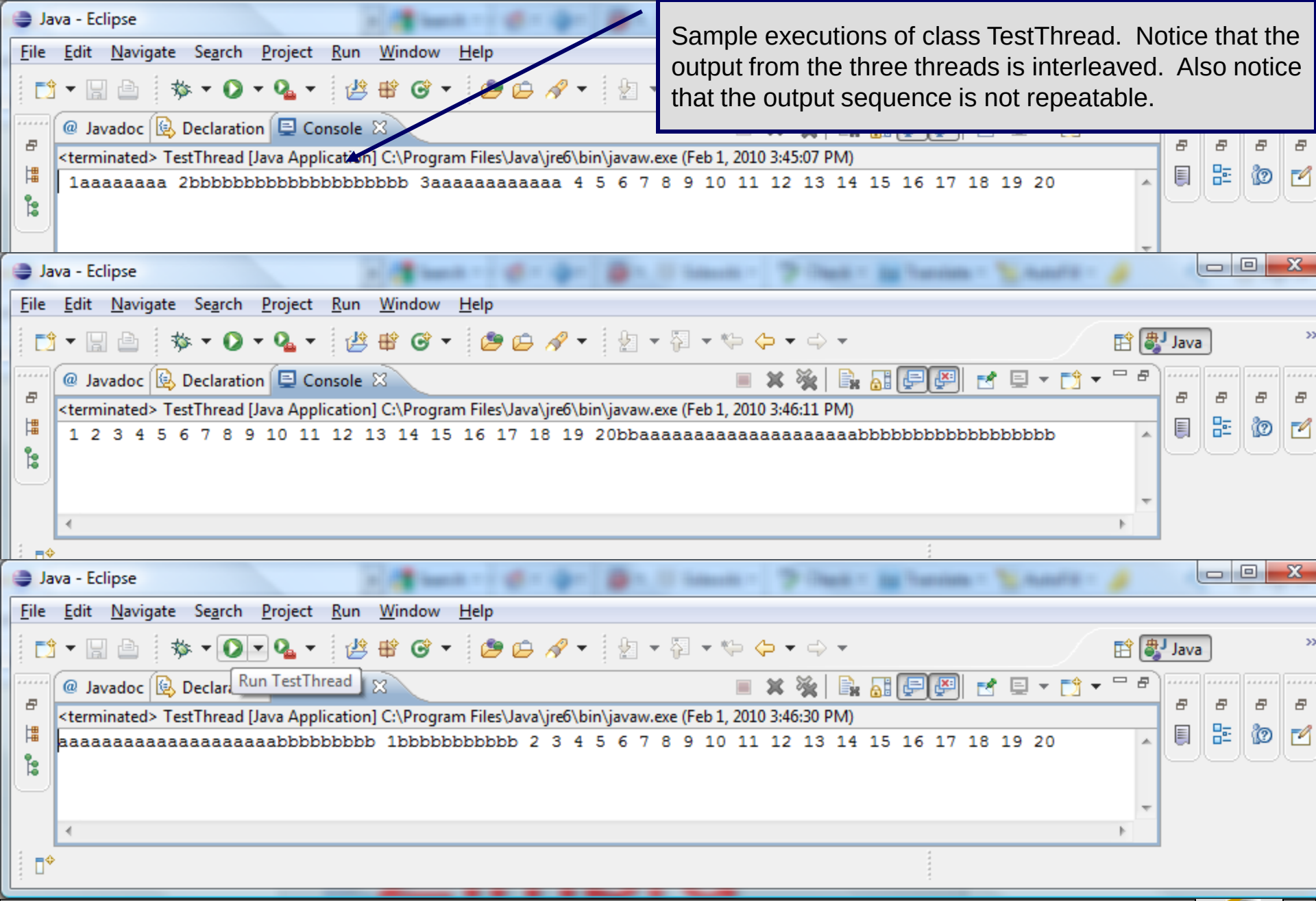
Start thread execution after a 0 msec delay (i.e., immediately)

Extension of the Thread class

```
// Override the run() method to tell the system what the thread will do
  public void run() {
    for (int i = 0; i < times; i++)
      System.out.print(charToPrint);
  }
}

// The thread class for printing number from 1 to n for a given n
class PrintNum extends Thread {
  private int lastNum;

  // Construct a thread for print 1, 2, ... i
  public PrintNum(int n) {
    lastNum = n;
  }

  // Tell the thread how to run
  public void run() {
    for (int i = 1; i <= lastNum; i++)
      System.out.print(" " + i);
  }
} //end class TestThread
```

Overriding the run method in the Thread class

Sample executions of class TestThread. Notice that the output from the three threads is interleaved. Also notice that the output sequence is not repeatable.



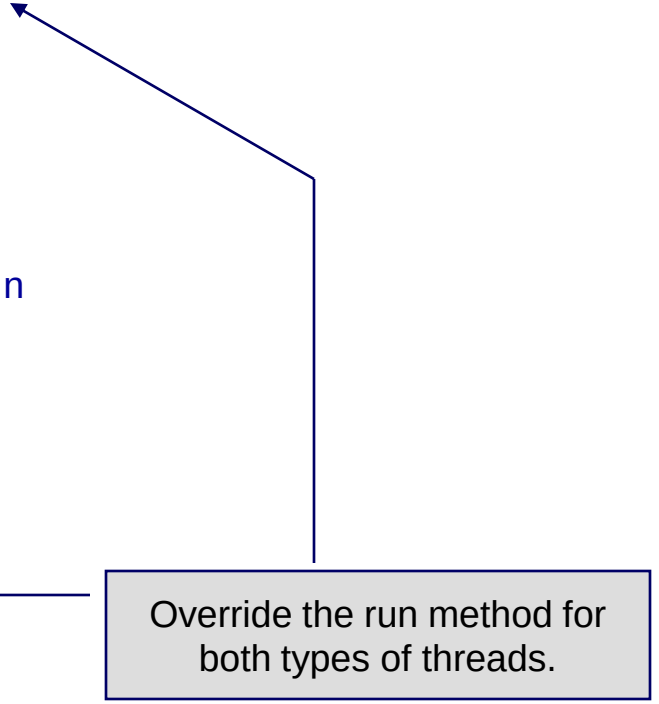
```
//Class to generate threads by implementing the Runnable interface
public class TestRunnable {
  // Create threads
  Thread printA = new Thread(new PrintChar('a', 20));
  Thread printB = new Thread(new PrintChar('b', 20));
  Thread print20 = new Thread(new PrintNum(20));

  public static void main(String[] args) {
    new TestRunnable();
  }

  public TestRunnable() {
    // Start threads
    print20.start();
    printA.start();
    printB.start();
  }

  // The thread class for printing a specified character in specified times
  class PrintChar implements Runnable {
    private char charToPrint;  // The character to print
    private int times;  // The times to repeat

    // Construct a thread with specified character and number of times to print the character
    public PrintChar(char c, int t) {
      charToPrint = c;
      times = t;
    }
```

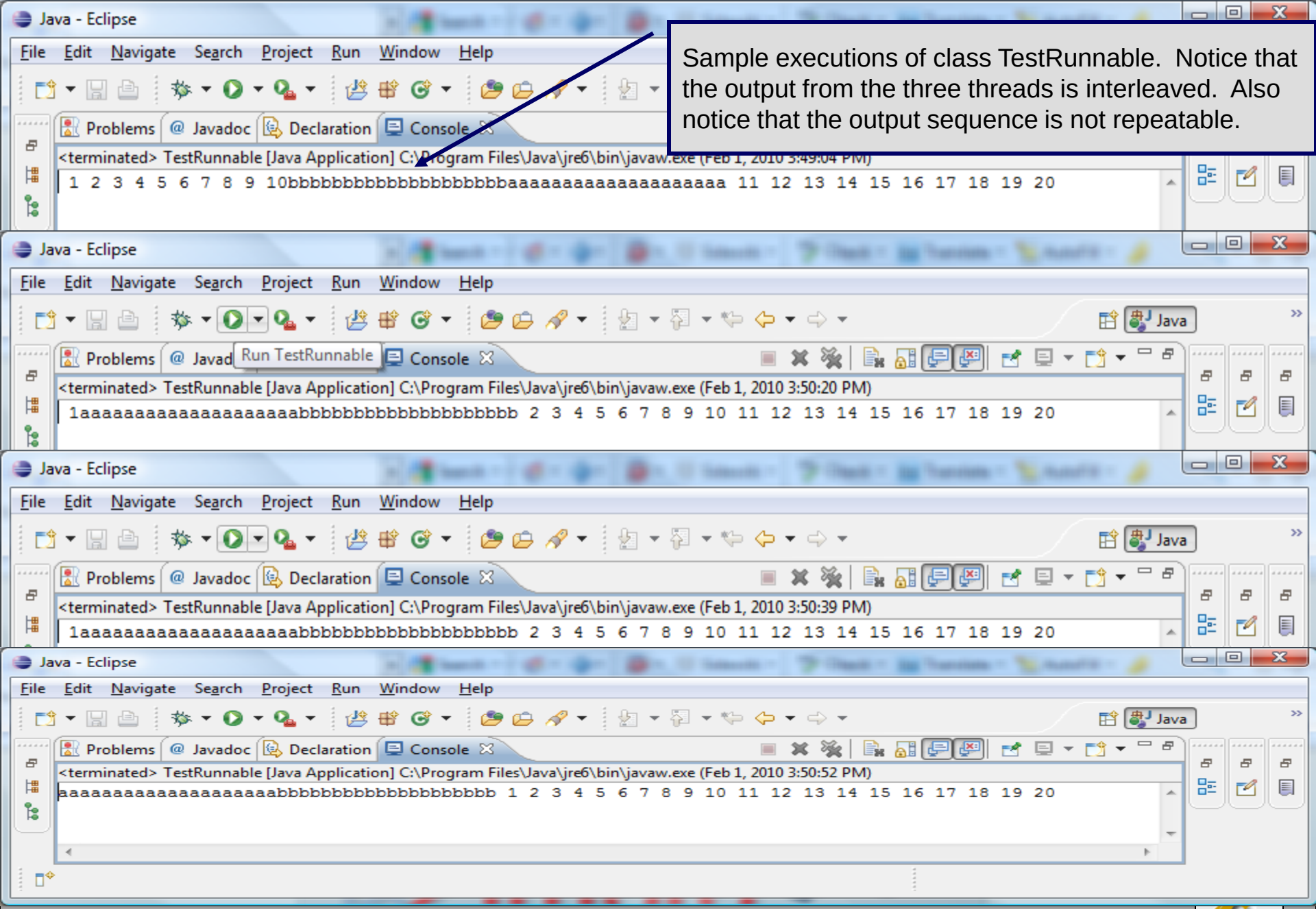
Main method simple creates a new Runnable object and terminates.

Runnable object starts thread execution.

Implements the Runnable interface.

```
// Override the run() method to tell the system what the thread will do
    public void run() {
      for (int i = 0; i < times; i++)
        System.out.print(charToPrint);
    }
  }

  // The thread class for printing number from 1 to n for a given n
  class PrintNum implements Runnable {
    private int lastNum;

    // Construct a thread for print 1, 2, ... i
    public PrintNum(int n) {
      lastNum = n;
    }

    // Tell the thread how to run
    public void run() {
      for (int i = 1; i <= lastNum; i++)
        System.out.print(" " + i);
    }
  }
} //end class TestRunnable
```





Sample executions of class TestRunnable. Notice that the output from the three threads is interleaved. Also notice that the output sequence is not repeatable.

Some Modifications to the Example

- To illustrate some of the methods in the `Thread` class, you might want to try a few modifications to the `TestRunnable` class in the previous example. Notice how the modifications change the order of the numbers and characters in the output.
- Use the `yield()` method to temporarily release time for other threads to execute. Modify the code in the `run` method in `PrintNum` class to the following:

```
public void run() {
      for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
         Thread.yield();
    }
```

  - Now every time a number is printed, the print20 thread yields, so each number will be followed by some characters.

Some Modifications to the Example (cont.)

- The `sleep(long millis)` method puts the thread to sleep for the specified time in milliseconds. Modify the code in the `run` method in `PrintNum` class to the following:

```
public void run() {
      for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        try {
                if (i >= 10) Thread.sleep(2);
        }
        catch (InterruptedException ex) { }
      }
}
```

  - Now every time a number greater than 10 is printed, the print20 thread is put to sleep for 2 milliseconds, so all the characters will complete printing before the last integer is printed.

Some Modifications to the Example (cont.)

- You can use the `join()` method to force one thread to wait for another thread to finish. Modify the code in the `run` method in `PrintNum` class to the following:

```
public void run() {
      for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        try {
                if (i == 10) printA.join();
        }
        catch (InterruptedException ex) { }
      }
}
```

  - Now the numbers greater than 10 are printed only after thread printA is finished.

Other Java Classes and Threads

- We noted earlier that Java has several different classes that support the creation and scheduling of threads. Classes `java.util.Timer` and `java.util.TimerTask` are generally the easiest to use. They allow a thread to be created and run either at a time relative to the current time or at some specific time.
- We'll look at these classes briefly and give a couple of examples.

Java Classes and Threads (cont.)

- Class `Timer` overloads the `schedule()` method three times for creating threads after either some specified delay or at some specific time.
  - `public void schedule(TimerTask task, long m);`
    - Runs `task.run()` after waiting `m` milliseconds.
  - `public void schedule(TimerTask task, long m, long n);`
    - Runs `task.run()` after waiting `m` milliseconds, then repeats it every `n` milliseconds.
  - `Public void schedule(TimerTask task, Date t);`
    - Runs `task.run()` at the time indicated by date `t`.
- By extending the abstract class `TimerTask` and specifying a definition for its abstract method `run()`, an application-specific thread can be created.

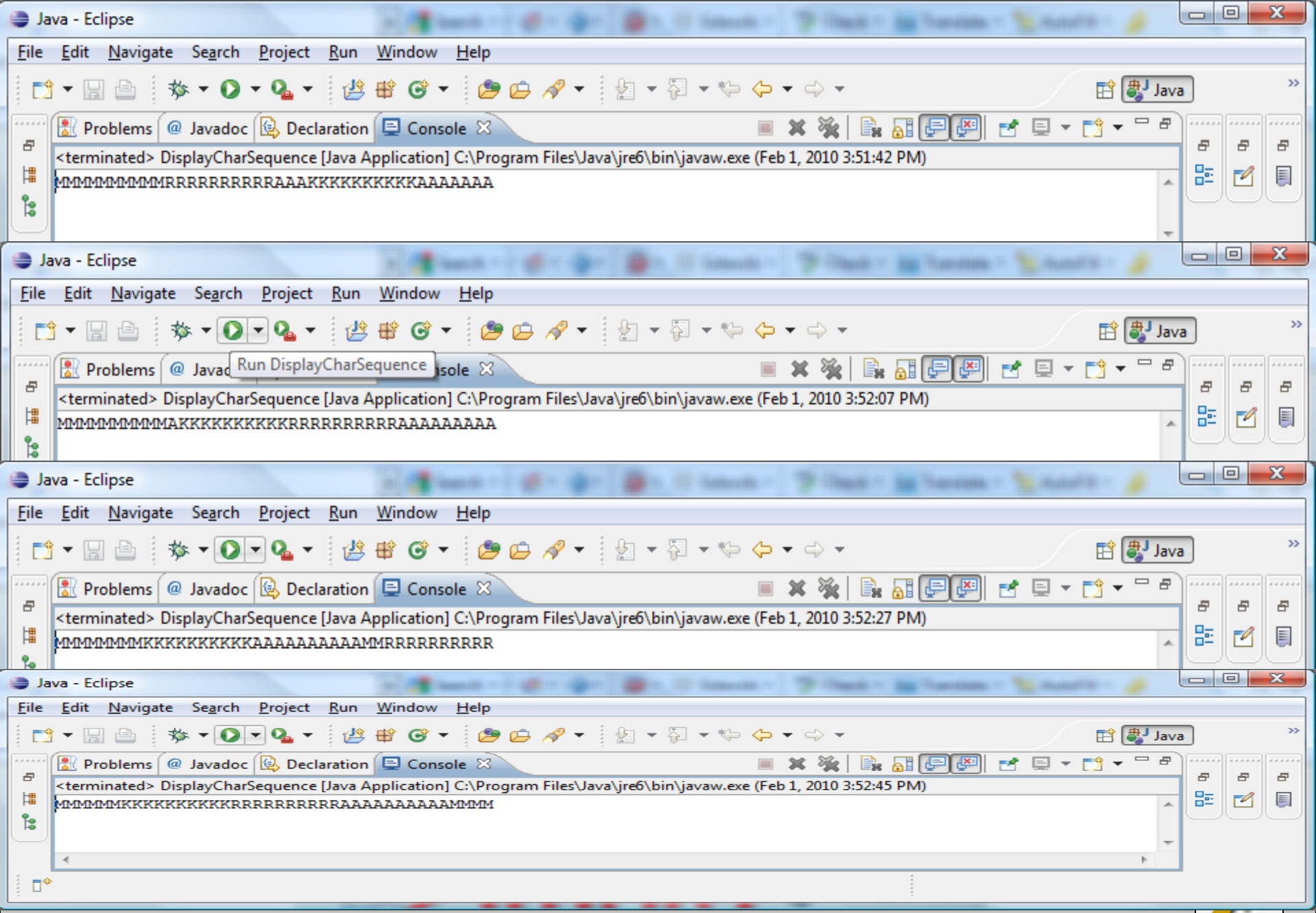
Example – Thread Execution After a Delay

- The code listing on the following page gives a very simple example of executing a thread after a delay (using the first schedule() method from the previous page).
- The thread in this example, simply prints a character 10 times and then ends.
- Look at the code and follow the flow, then execute it on your machine (code appears on the course webpage).

```
//displays characters in separate threads
import java.util.*;
public class DisplayCharSequence extends TimerTask {
   private char displayChar;
   Timer timer;

   //constructor for character displayer
   public DisplayCharSequence(char c){
      displayChar = c;
      timer = new Timer();
      timer.schedule(this, 0);
   }

   //display the occurrences of the character
   public void run() {
       for (int i = 0; i < 10; ++i) {
           System.out.print(displayChar);
       }
       timer.cancel();
   }

   //main
   public static void main (String[] args) {
      DisplayCharSequence s1 = new DisplayCharSequence('M');
      DisplayCharSequence s2 = new DisplayCharSequence('A');
      DisplayCharSequence s3 = new DisplayCharSequence('R');
      DisplayCharSequence s4 = new DisplayCharSequence('K');
   }
}
```

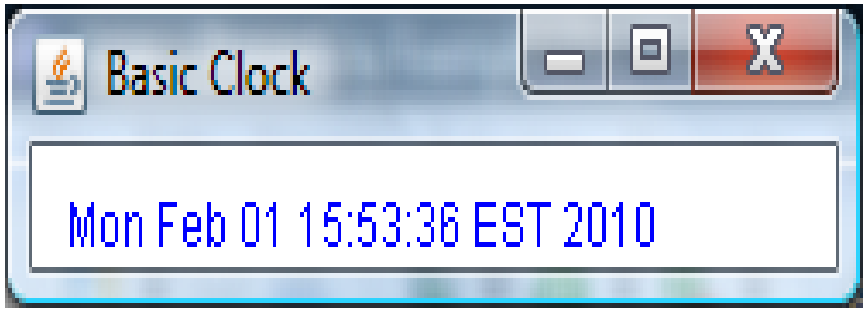
Start thread execution after a 0 msec delay (i.e., immediately)

A subclass implementation of TimerTask's abstract method run() has typically two parts – first part is application specific (what the thread is supposed to do) and the second part ends the thread.


Java - Eclipse
File Edit Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Console
<terminated> DisplayCharSequence [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 1, 2010 3:51:42 PM)
MMMMMMMMMRRRRRRRRRRAAAKKKKKKKKKAAAAAAA
Java - Eclipse
File Edit Navigate Search Project Run Window Help
Java
Problems
Run DisplayCharSequence
<terminated> DisplayCharSequence [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 1, 2010 3:52:07 PM)
MMMMMMMMMAKKKKKKKKKRRRRRRRRRRAAAAAAAAA
Java - Eclipse
File Edit Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Console
<terminated> DisplayCharSequence [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 1, 2010 3:52:27 PM)
MMMMMMMKKKKKKKKKKAAAAAAAAAAMMRRRRRRRRR
Java - Eclipse
File Edit Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Console
<terminated> DisplayCharSequence [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 1, 2010 3:52:45 PM)
MMMMMKKKKKKKKKRRRRRRRRRRAAAAAAAAAAMMMM

Example – Repeated Thread Execution

- This next example demonstrates how to schedule a thread to run multiple times. Basically, the thread updates a GUI-based clock every second.

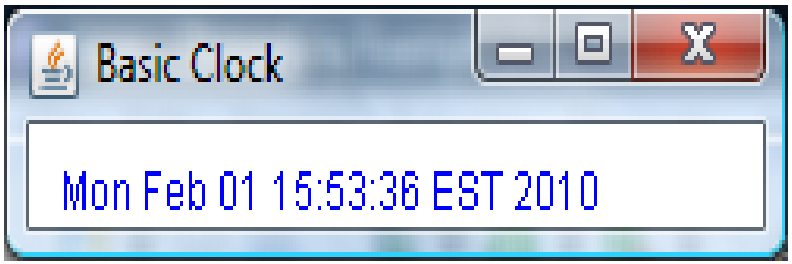


Sample GUI

```
//displays current time - threaded execution
import java.util.*;
import javax.swing.JFrame;
import java.text.*;
import java.awt.*;

public class BasicClock extends TimerTask {
   final static long MILLISECONDS_PER_SECOND = 1000;
   private JFrame window = new JFrame("Basic Clock");
   private Timer timer = new Timer();
   private String clockFace = "";

   //constructor for clock
   public BasicClock(){
      //set up GUI
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      window.setSize(200,60);
      Container c = window.getContentPane();
      c.setBackground(Color.WHITE);
      window.setVisible(true);
      //update GUI every second beginning immediately
      timer.schedule(this,0,1*MILLISECONDS_PER_SECOND);
    }
```

Two tasks: (1) configure the GUI and (2) schedule the thread to update the GUI-clock every second.

This form of the overloaded schedule() method is the second one shown on page 28 which uses a delay and a repetition factor.



Date() returns current time to the millisecond. toString() method returns a textual representation of the date in the form: *w c d h:m:s z y*
Where: w: 3 char-rep of day of week
c: 3 char-rep of month
d: 2 digit-rep of day of month
h: 2 digit-rep of hour
m: 2 digit-rep of minute within hr
s: 2 digit-rep of second within min
z: 3 char-rep of time zone
y: 4 char-rep of year

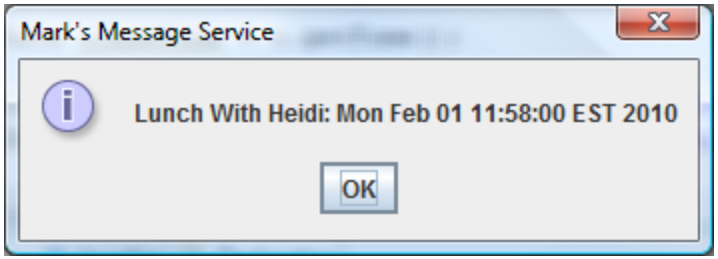
```
    //display updated clock
    public void run(){
        Date time = new Date();
        Graphics g = window.getContentPane().getGraphics();
        g.setColor(Color.WHITE);
        g.drawString(clockFace, 10, 20);
        clockFace = time.toString();
        g.setColor(Color.BLUE);
        g.drawString(clockFace,10, 20);
}

    //main
    public static void main (String[] args) {
       BasicClock clock = new BasicClock();
    }
}
```


!! CAUTION !!

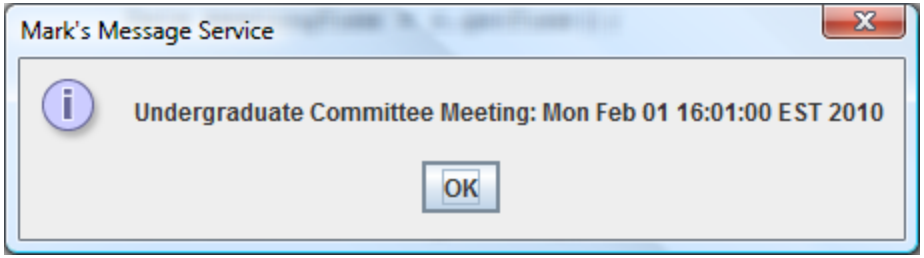
- Java provides two different standard classes named `Timer`. The class we've used in the past two examples is part of the `util` API. There is also a `Timer` class that is part of the `swing` API.
- In our previous example, we needed to make sure that we didn't inadvertently bring both `Timer` classes into our program which would have created an ambiguity about which `Timer` class was being used.
- Although you cannot import both `Timer` classes into a single Java source file, you can use both `Timer` classes in the same Java source file. An `import` statement exists to allow a syntactic shorthand when using Java resources; i.e., an `import` statement is not required to make use of Java resources. Using fully qualified class names will remove the ambiguity.
  - java.util.Time t1 = new java.util.Timer();
  - javax.swing.Timer t2 = new javax swing.Timer();

Example – Thread Execution At Specific Time

- This next example demonstrates how to schedule a thread to run at a specific time. This example will create a couple of threads to remind you of impending appointments. Basically, the thread pops-up a window to remind you of the appointment.



Sample DisplayAlert Window



This window pops up 4 minutes later

```
//Displays an alert at a specific time - threaded execution
   import javax.swing.JOptionPane;
   import java.awt.*;
   import java.util.*;

   public class DisplayAlert extends TimerTask {
        //instance variables
      private String message;
        private Timer timer;

        //constructor
        public DisplayAlert(String s, Date t){
           message = s + ": " + t;
                timer = new Timer();
                timer.schedule(this, t);
        }

        //execute thread
        public void run() {
           JOptionPane.showMessageDialog(null, message, "Mark's Message
   Service", JOPtionPane.INFORMATION_MESSAGE);  //application specific task
                timer.cancel();  //kill thread
        }
```

Third version of schedule() method as shown on page 28.

```
    public static void main(String[] args) {
       Calendar c = Calendar.getInstance();
            c.set(Calendar.HOUR_OF_DAY, 11);
            c.set(Calendar.MINUTE, 58);
            c.set(Calendar.SECOND, 0);

            Date lunchTime = c.getTime();

            c.set(Calendar.HOUR_OF_DAY, 16);
            c.set(Calendar.MINUTE, 01);
            c.set(Calendar.SECOND, 0);

            Date meetingTime = c.getTime();

            DisplayAlert alert1 = new DisplayAlert("Lunch With Heidi",
lunchTime);
            DisplayAlert alert2 = new DisplayAlert("Undergraduate
Committee Meeting", meetingTime);
     }
}
```

Create two messages to be displayed at different times.

Sleeping

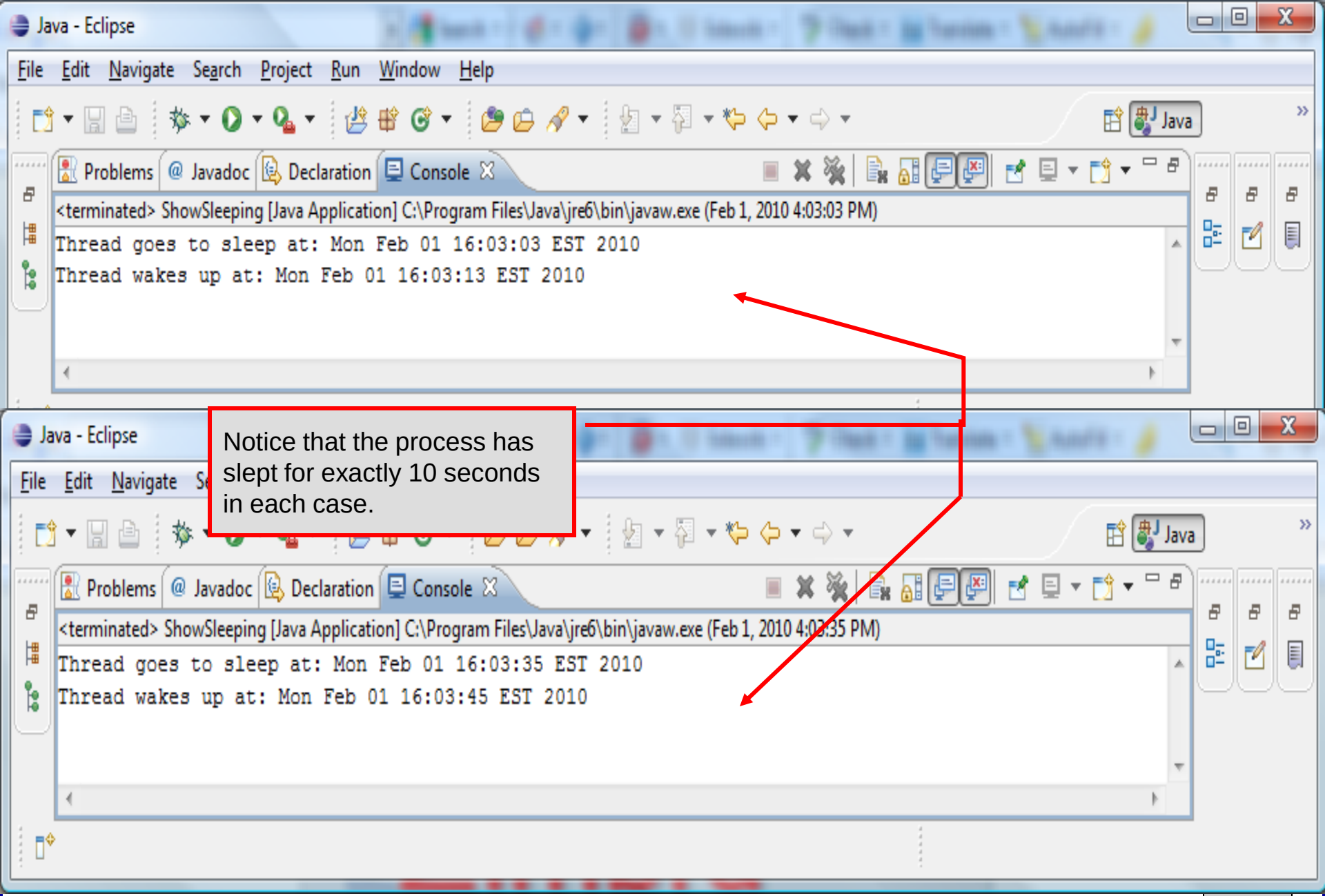
- In the three examples so far, all the threads performed some action. Threads are also used to pause a program for some period of time.
- Standard class `java.lang.Thread` has a class method `sleep()` for pausing the flow of control.

```
public static void sleep (long n) throws InterruptedException
```

- For example, the following code segment will twice get and display the current time, but the time acquisitions are separated by 10 seconds by putting the process to sleep.

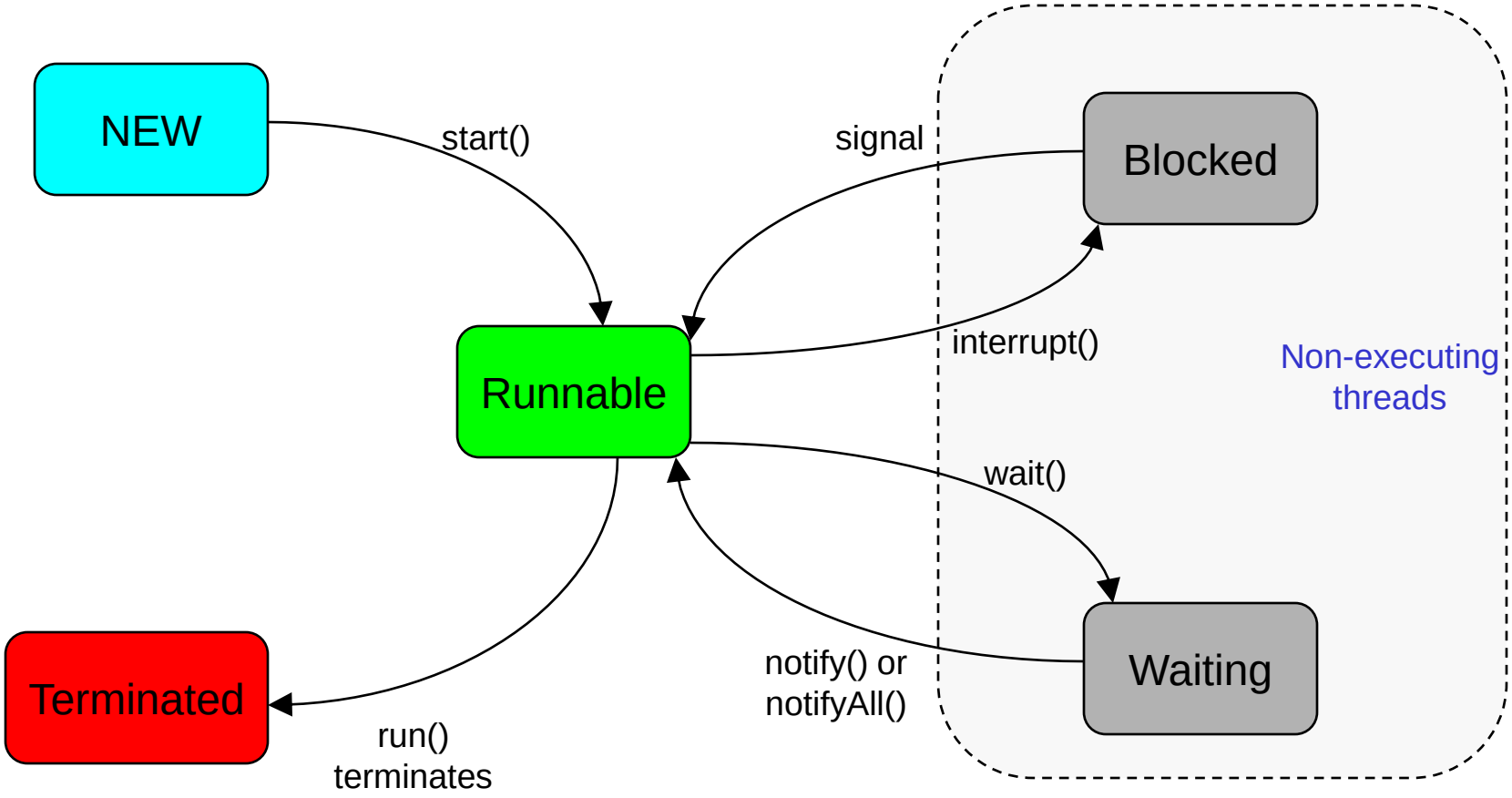
```
//Illustrates putting a process to sleep
   import java.util.*;

   public class ShowSleeping {

        public static void main(String[] args) {
           Date t1 = new Date();
                System.out.println("Thread goes to sleep at: " +
   t1);
                try {
                   Thread.sleep(10000);
                }
                catch (InterruptedException e) {
                }
                Date t2 = new Date();
                System.out.println("Thread wakes up at: " + t2);
        }
   }
```

Put the process to sleep for 10 seconds.

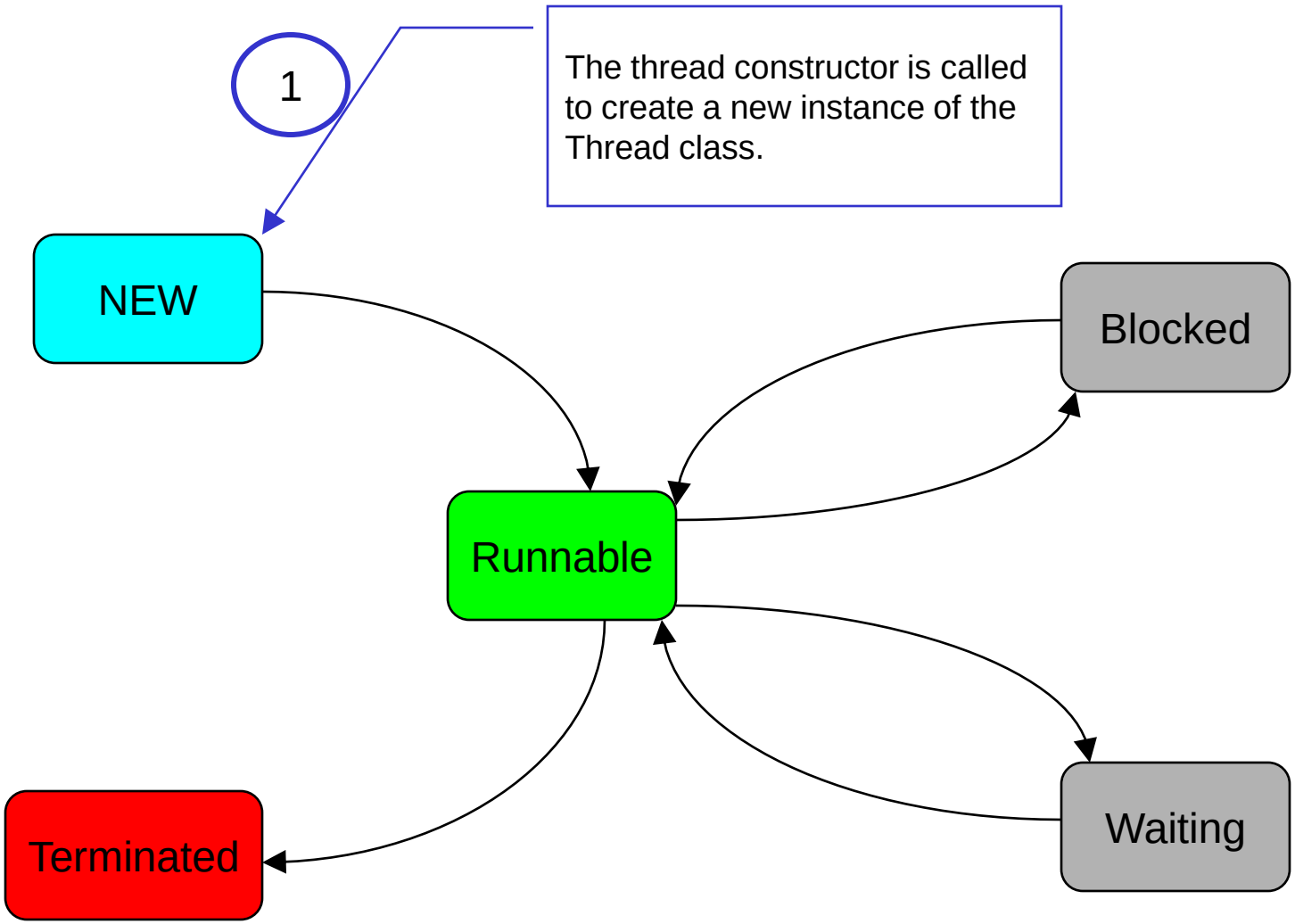

Java - Eclipse
File Edit Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> ShowSleeping [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 1, 2010 4:03:03 PM)
Thread goes to sleep at: Mon Feb 01 16:03:03 EST 2010
Thread wakes up at: Mon Feb 01 16:03:13 EST 2010
Java - Eclipse
File Edit Navigate
Problems Javadoc Declaration Console
<terminated> ShowSleeping [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 1, 2010 4:03:35 PM)
Thread goes to sleep at: Mon Feb 01 16:03:35 EST 2010
Thread wakes up at: Mon Feb 01 16:03:45 EST 2010
Notice that the process has slept for exactly 10 seconds in each case.

Life Cycle of a Thread

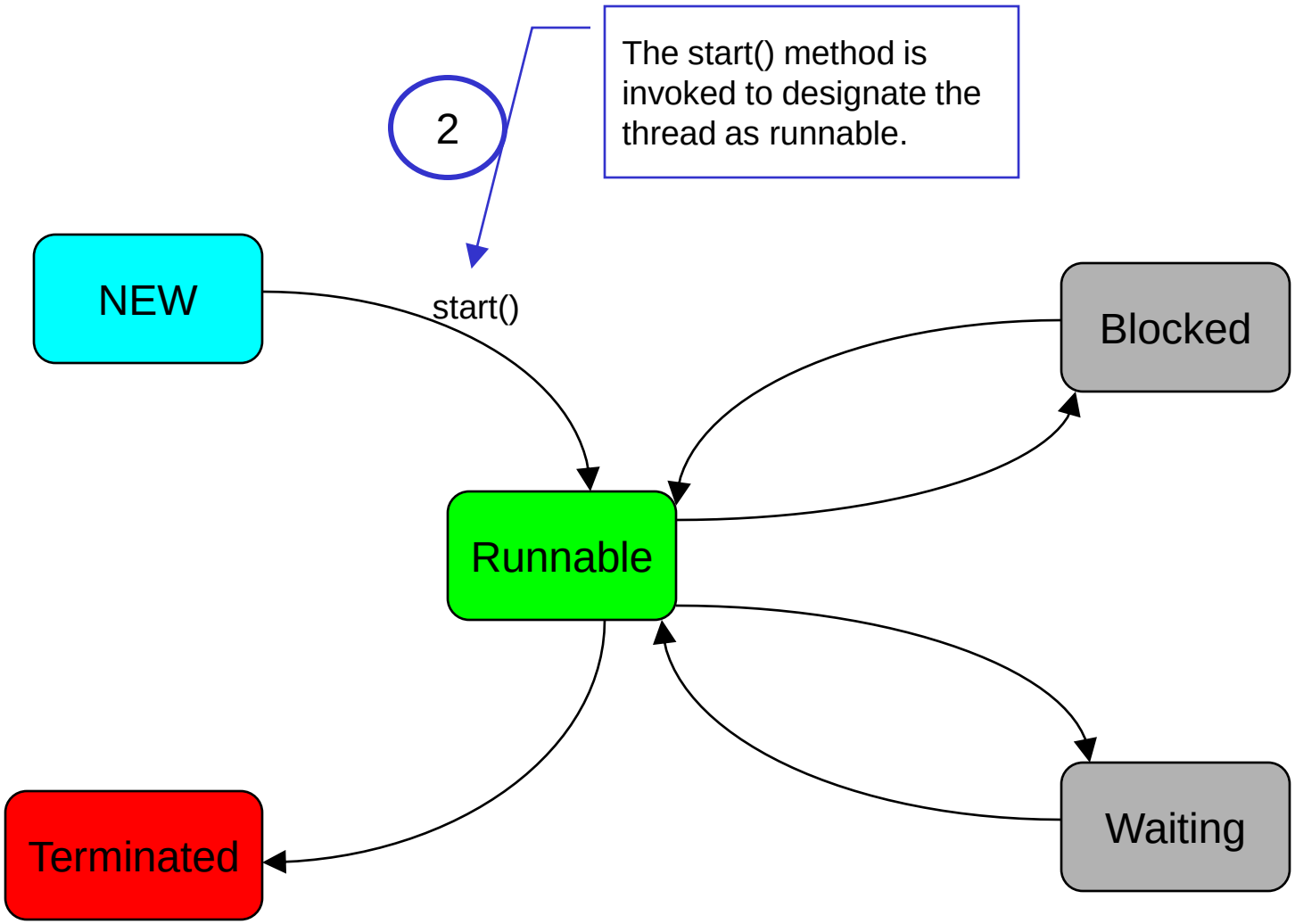
- At any given point in time, a thread is said to be in one of several thread states as illustrated in the diagram below.



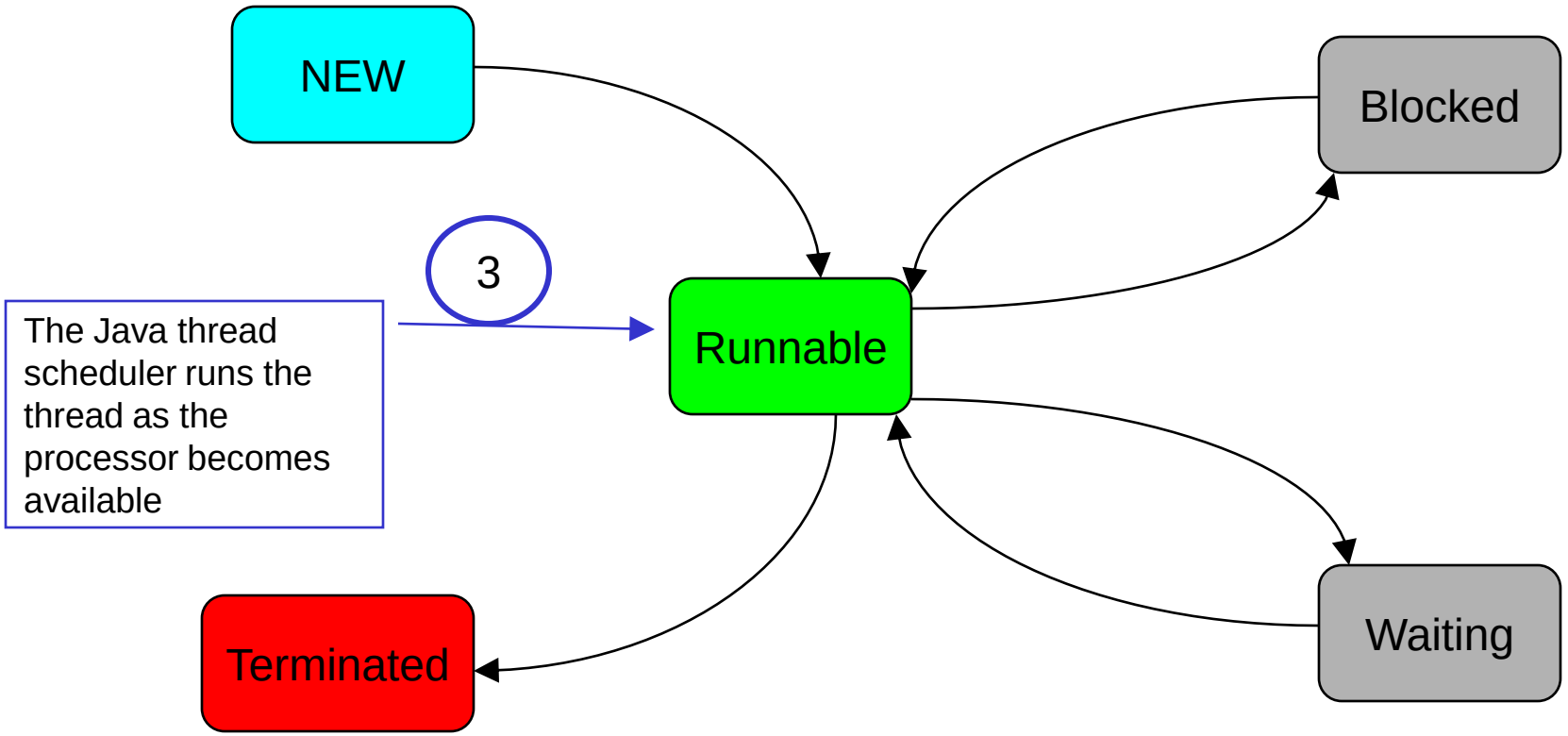
Life Cycle of a Thread (cont.)



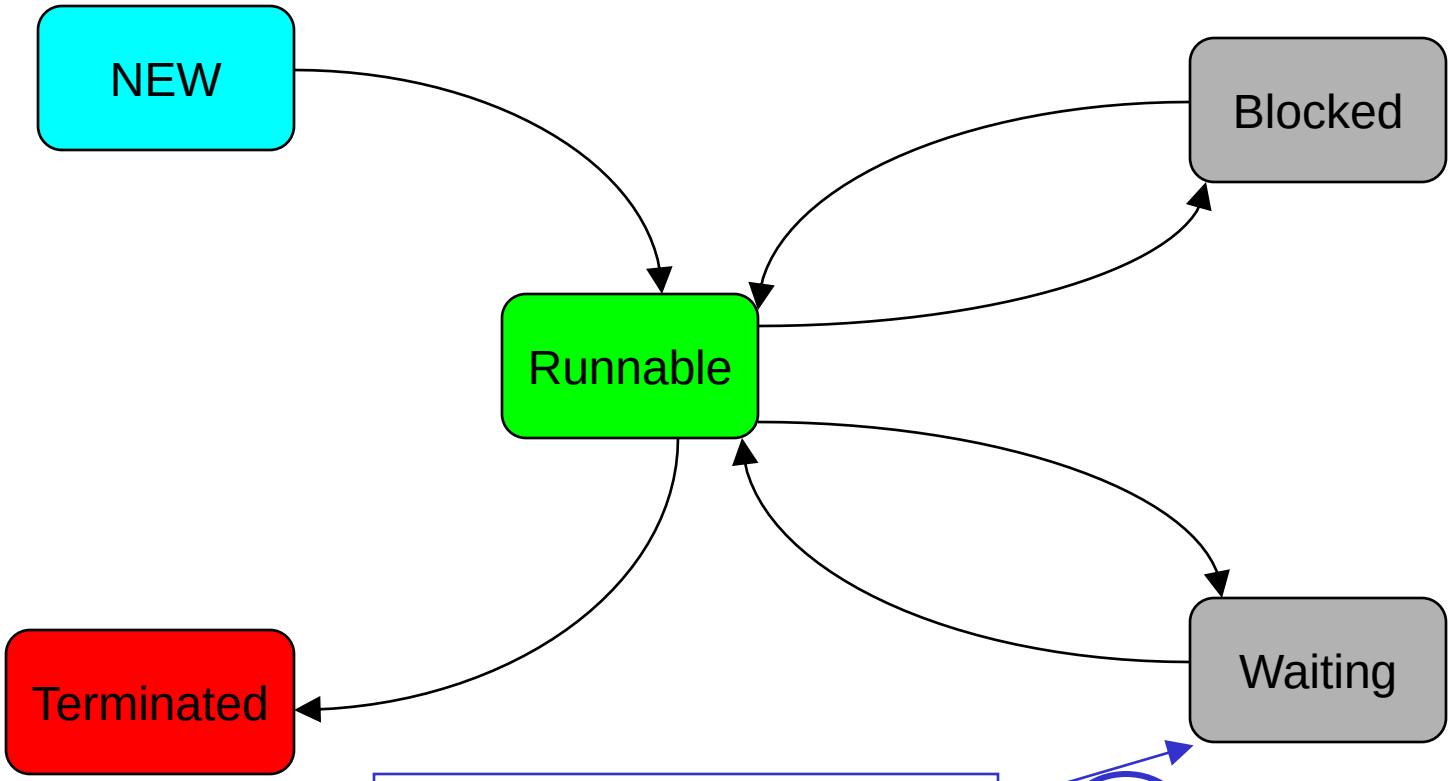
Life Cycle of a Thread (cont.)



Life Cycle of a Thread (cont.)

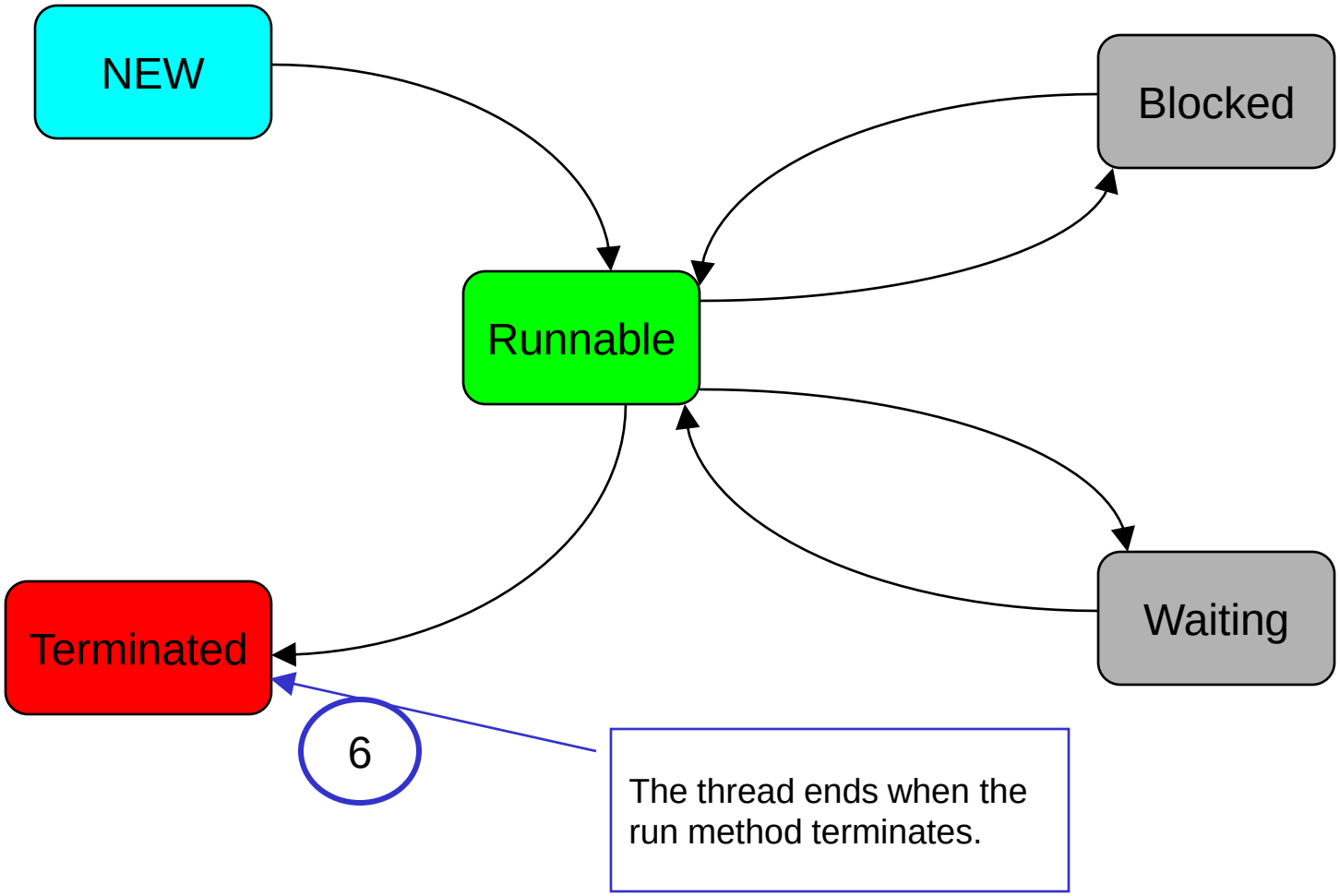


Life Cycle of a Thread (cont.)

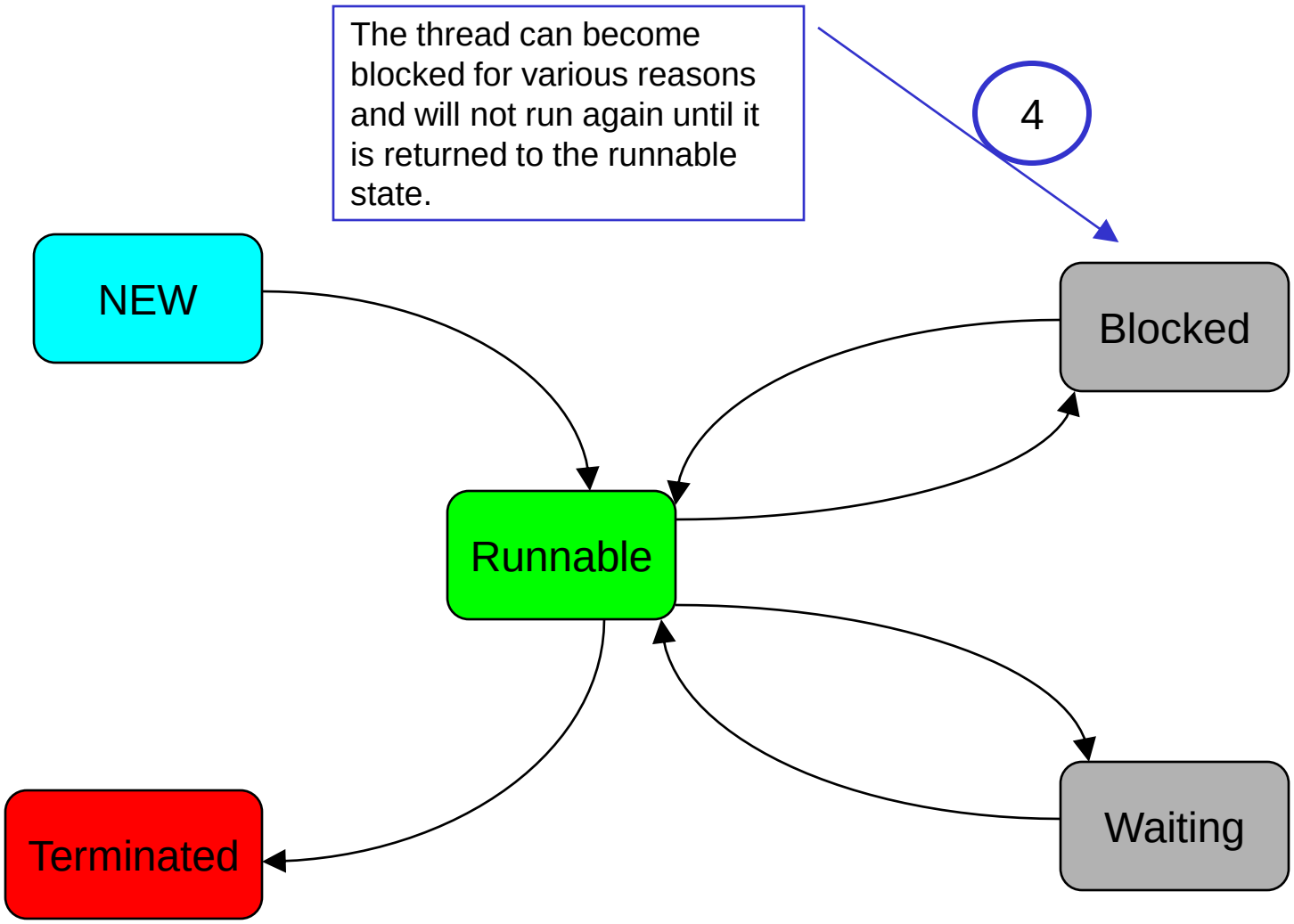


If the thread invokes the wait() method, it is put into the waiting state and will remain there until another thread invokes the notify() or notifyAll() method.

Life Cycle of a Thread (cont.)



Life Cycle of a Thread (cont.)

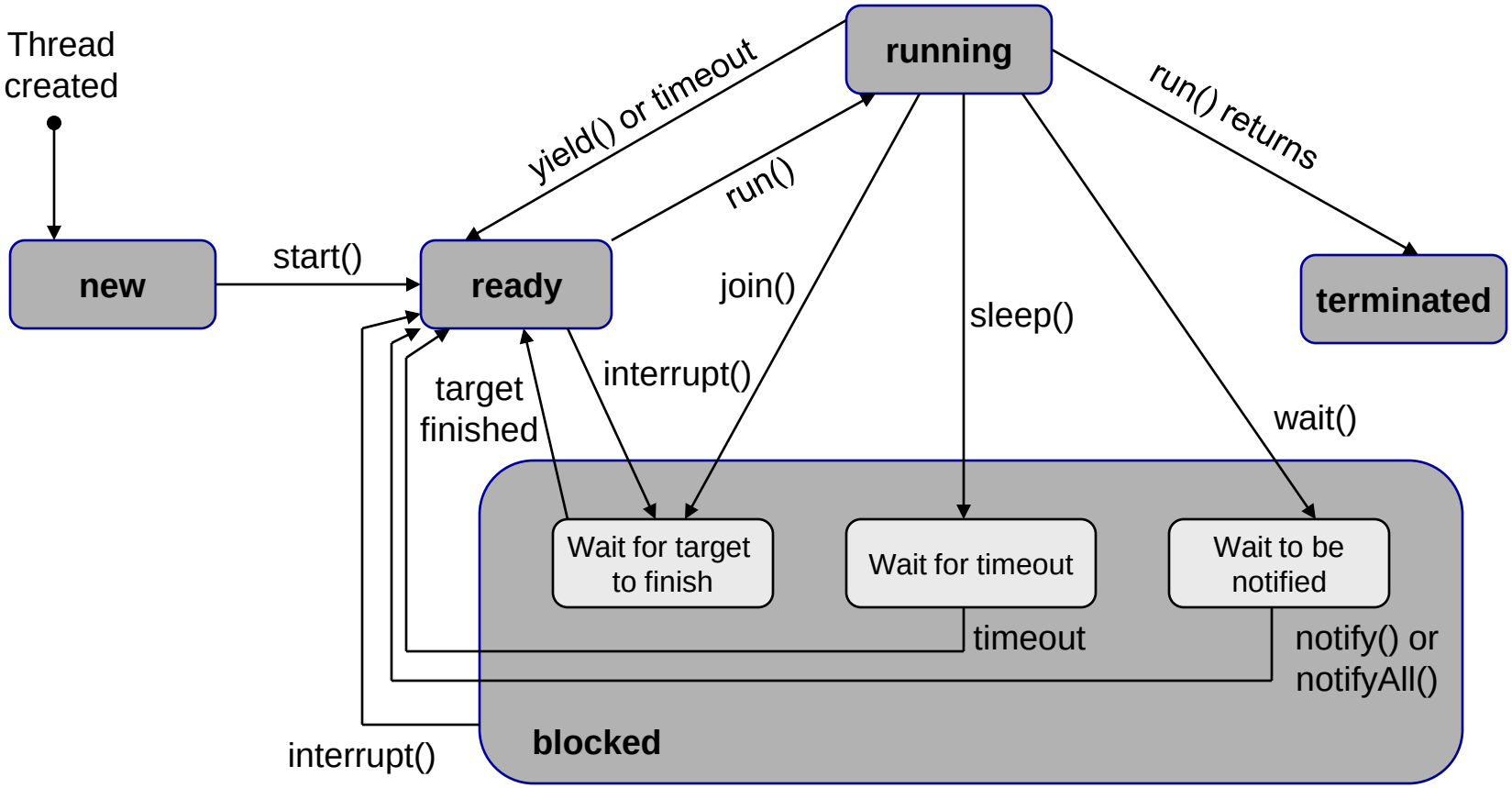


Summary of States In The Life Cycle of a Thread

State	Description
New	The thread has been created (its constructor has been invoked), but not yet started.
Runnable	The thread's start() method has been invoked and the thread is available to be run by the thread scheduler. A thread in the Runnable state may actually be running, or it may be waiting in the thread queue for an opportunity to run.
Blocked	The thread has been temporarily removed from the Runnable state so that it cannot be executed. This can happen if the thread's sleep() method is invoked, if the thread is waiting on I/O, or if the thread requests a lock on an object that is already locked. When the condition changes, the thread will be returned to the Runnable state.
Waiting	The thread has invoked its wait() method so that other threads can access an object. The thread will remain in the Waiting state until another thread invokes the notify() or notifyAll() method.
Terminated	The thread's run() method has ended.

Life Cycle of a Thread – A Slightly Different View

- At any given point in time, a thread is said to be in one of several thread states as illustrated in the diagram below.



Life Cycle of a Thread (cont.)

- A new thread begins its life cycles in the new state. It remains in this state until the program starts the thread, which places the thread in the ready state (also commonly referred to as the runnable state). A thread in this state is considered to be executing its task, although at any given moment it may not be actually executing.
- When a ready thread begins execution, it enters the running state. A running thread may return to the ready state if its CPU time slice expires or its `yield()` method is invoked.
- A thread can enter the blocked state (i.e., it becomes inactive) for several reasons. It may have invoked the `join()`, `sleep()`, or `wait()` method, or some other thread may have invoked these methods. It may be waiting for an I/O operation to complete.
- A blocked thread can be reactivated when the action which inactivated it is reversed. For example, if a thread has been put to sleep and the sleep time has expired, the thread is reactivated and enters the ready state.

Life Cycle of a Thread (cont.)

- A thread is terminated if it completes the execution of its `run()` method.
- The `isAlive()` method is used to query the state of a thread. This method returns true it a thread is in the ready, blocked, or running state; it returns false if a thread is new and has not started or if it is finished.
- The `interrupt()` method interrupts a thread in the following way: If a thread is currently in the ready or running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the ready state, and a java.lang.InterruptedException is thrown.
- Threads typically sleep when they momentarily do not have work to perform. Example, a word processor may contain a thread that periodically writes a copy of the current document to disk for recovery purposes.

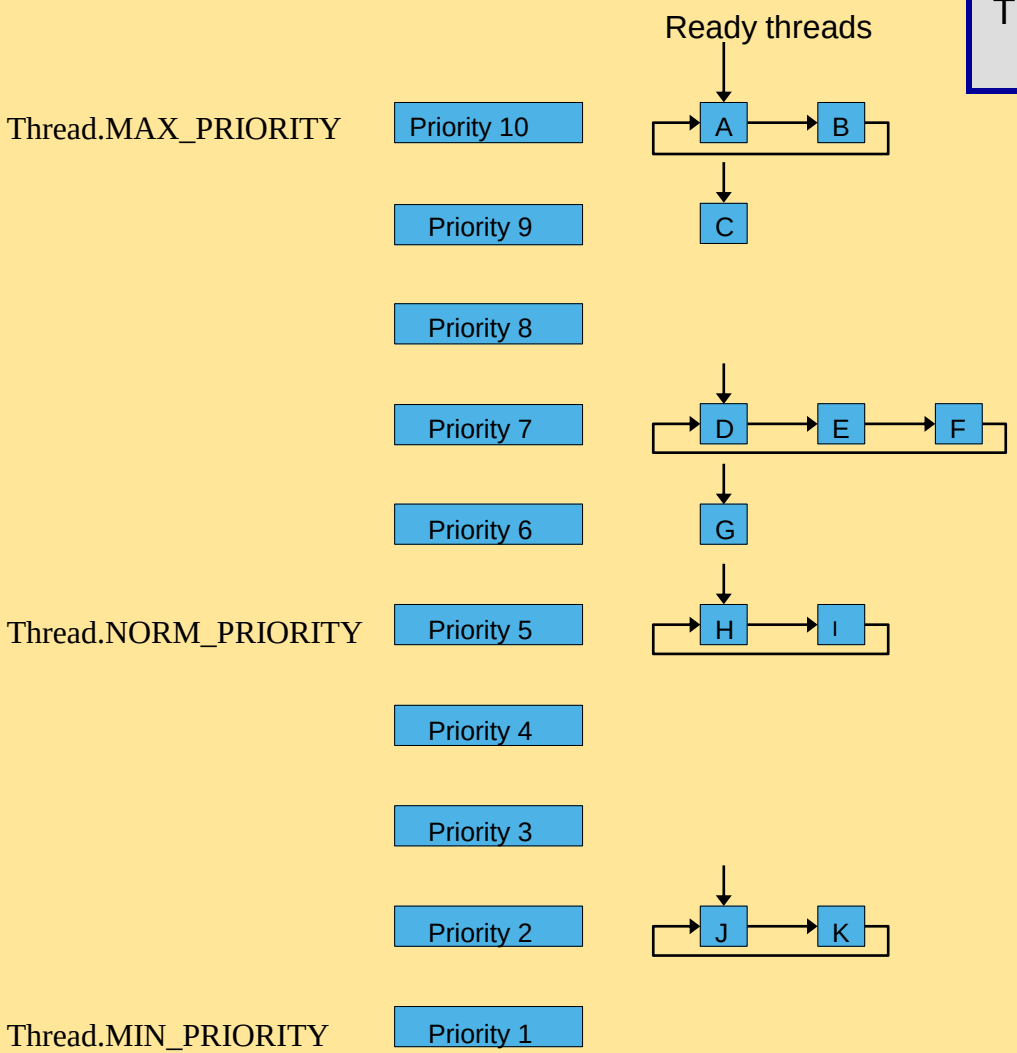
Life Cycle of a Thread (cont.)

- A runnable thread enters the terminated state when it completes its task or otherwise terminates (perhaps due to an error condition).
- At the OS level, Java's runnable state actually encompasses two separate states. The OS hides these two states from the JVM, which sees only the runnable state.
  - When a thread first transitions to the runnable state from the new state, the thread is in the ready state. A ready thread enters the running state (i.e., begins execution) when the OS assigns the thread to a processor (this is called dispatching the thread). In most OS, each thread is given a small amount of processor time – called a quantum or time slice – with which to perform its task. When the thread's quantum expires, the thread returns to the ready state and the OS assigns another thread to the processor. Transitions between these states are handled solely by the OS.

Thread Priorities

- Every Java thread has a priority that helps the OS determine the order in which threads are scheduled.
- Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10).
- Threads with a higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute.
- By default, every thread is given priority NORM_PRIORITY (a constant of 5). Each new thread inherits the priority of the thread that created it.

Thread Priority Scheduling



Creating and Executing Threads

- In J2SE 5.0, the preferred means of creating a multithreaded application is to implement the `Runnable` interface (package `java.lang`) (see earlier examples also) and use built-in methods and classes to create `Threads` that execute the `Runnables`.
- The `Runnable` interface declares a single method named `run`, `Runnables` are executed by an object of a class that implements the `Executor` interface (package `java.util.concurrent`). This interface declares a single method named `execute`.
- An `Executor` object typically creates and managed a group of threads called a thread pool. These threads execute the `Runnable` objects passed to the `execute` method.

Creating and Executing Threads (cont.)

- The `Executor` assigns each `Runnable` to one of the available threads in the thread pool. If there are no available threads in the thread pool, the `Executor` creates a new thread or waits for a thread to become available and assigns that thread the `Runnable` that was passed to method `execute`.

- Depending on the Executor type, there may be a limit to the number of threads that can be created. Interface ExecutorService (package `java.util.concurrent`) is a subinterface of Executor that declares a number of other methods for managing the life cycle of the Executor. An object that implements this ExecutorService interface can be created using static methods declared in class Executors (package `java.util.concurrent`). The next examples illustrates these.

Multithreading Example – Sleeping/Waking Threads

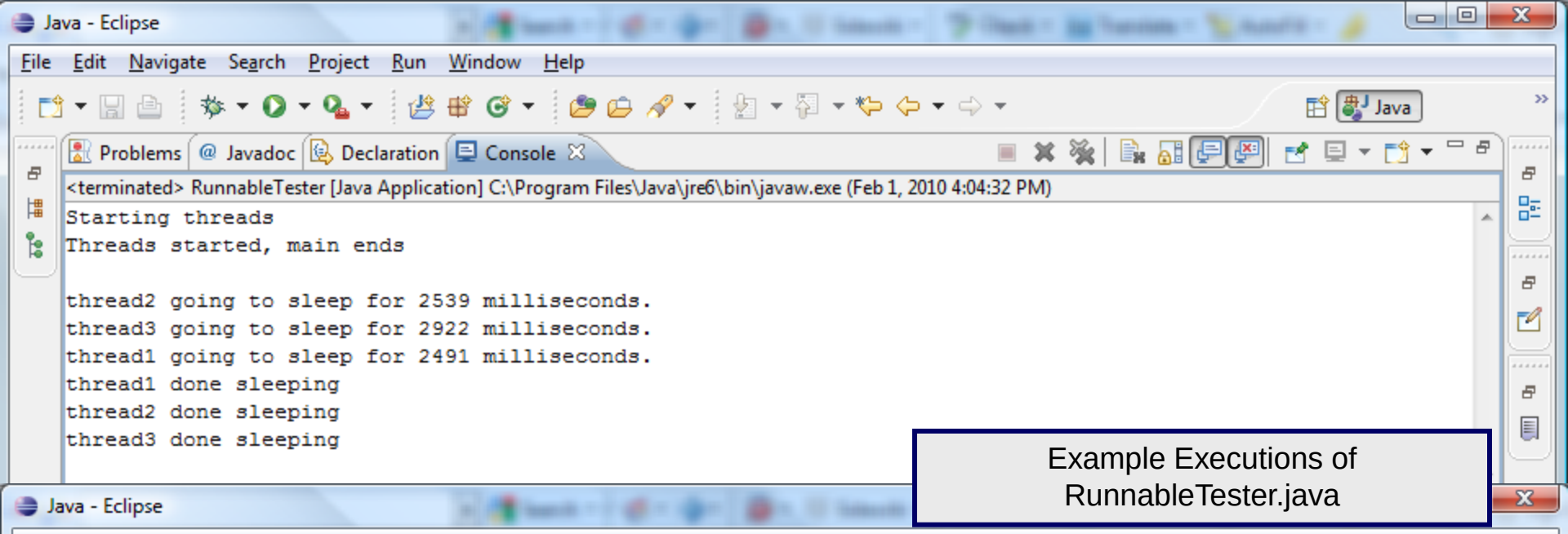
```
// PrintTask class sleeps for a random time from 0 to 5 seconds
import java.util.Random;

public class PrintTask implements Runnable
{
   private int sleepTime; // random sleep time for thread
   private String threadName; // name of thread
   private static Random generator = new Random();

   // assign name to thread
   public PrintTask( String name )
   {
      threadName = name; // set name of thread

      // pick random sleep time between 0 and 5 seconds
      sleepTime = generator.nextInt( 5000 );
   } // end PrintTask constructor
```

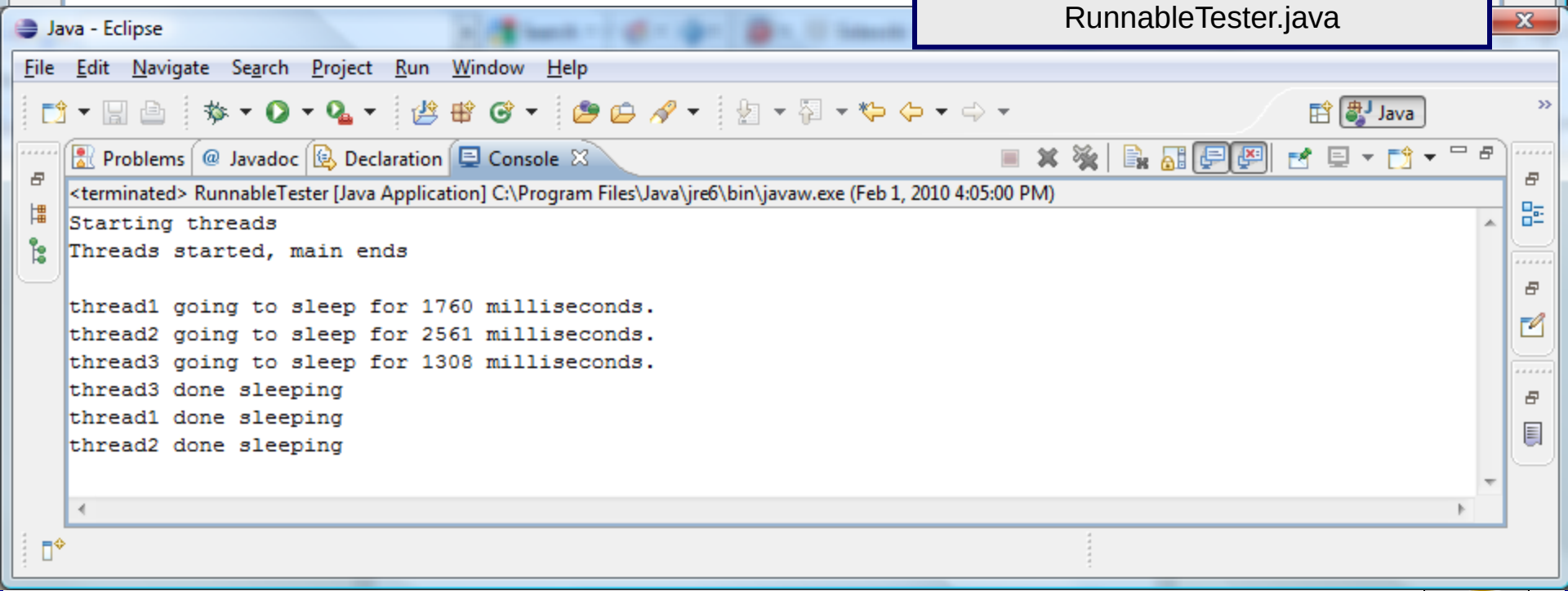

Multithreading Example – Sleeping/Waking Threads

```
// method run is the code to be executed by new thread
   public void run()
   {
      try // put thread to sleep for sleepTime amount of time
   {
   System.out.printf( "%s going to sleep for %d milliseconds.\n",
            threadName, sleepTime );

         Thread.sleep( sleepTime ); // put thread to sleep
   } // end try
    // if thread interrupted while sleeping, print stack trace
    catch ( InterruptedException exception )
    {
      exception.printStackTrace();
    } // end catch
      // print thread name
      System.out.printf( "%s done sleeping\n", threadName );
   } // end method run
} // end class PrintTask
```


Multithreading Example – Create Threads and Execute

```
// Multiple threads printing at different intervals.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
public class RunnableTester
{
   public static void main( String[] args )  {
      // create and name each runnable
      PrintTask task1 = new PrintTask( "thread1" );
      PrintTask task2 = new PrintTask( "thread2" );
      PrintTask task3 = new PrintTask( "thread3" );

      System.out.println( "Starting threads" );

      // create ExecutorService to manage threads
      ExecutorService threadExecutor = Executors.newCachedThreadPool();
      // start threads and place in runnable state
      threadExecutor.execute( task1 ); // start task1
      threadExecutor.execute( task2 ); // start task2
      threadExecutor.execute( task3 ); // start task3

      threadExecutor.shutdown(); // shutdown worker threads

      System.out.println( "Threads started, main ends\n" );
   } // end main
} // end class RunnableTester
```





Example Executions of RunnableTester.java