CNT 4714: Enterprise Computing Summer 2012

Programming Multithreaded Applications in Java

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cnt4714/sum2012

Department of Electrical Engineering and Computer Science
Computer Science Division
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.
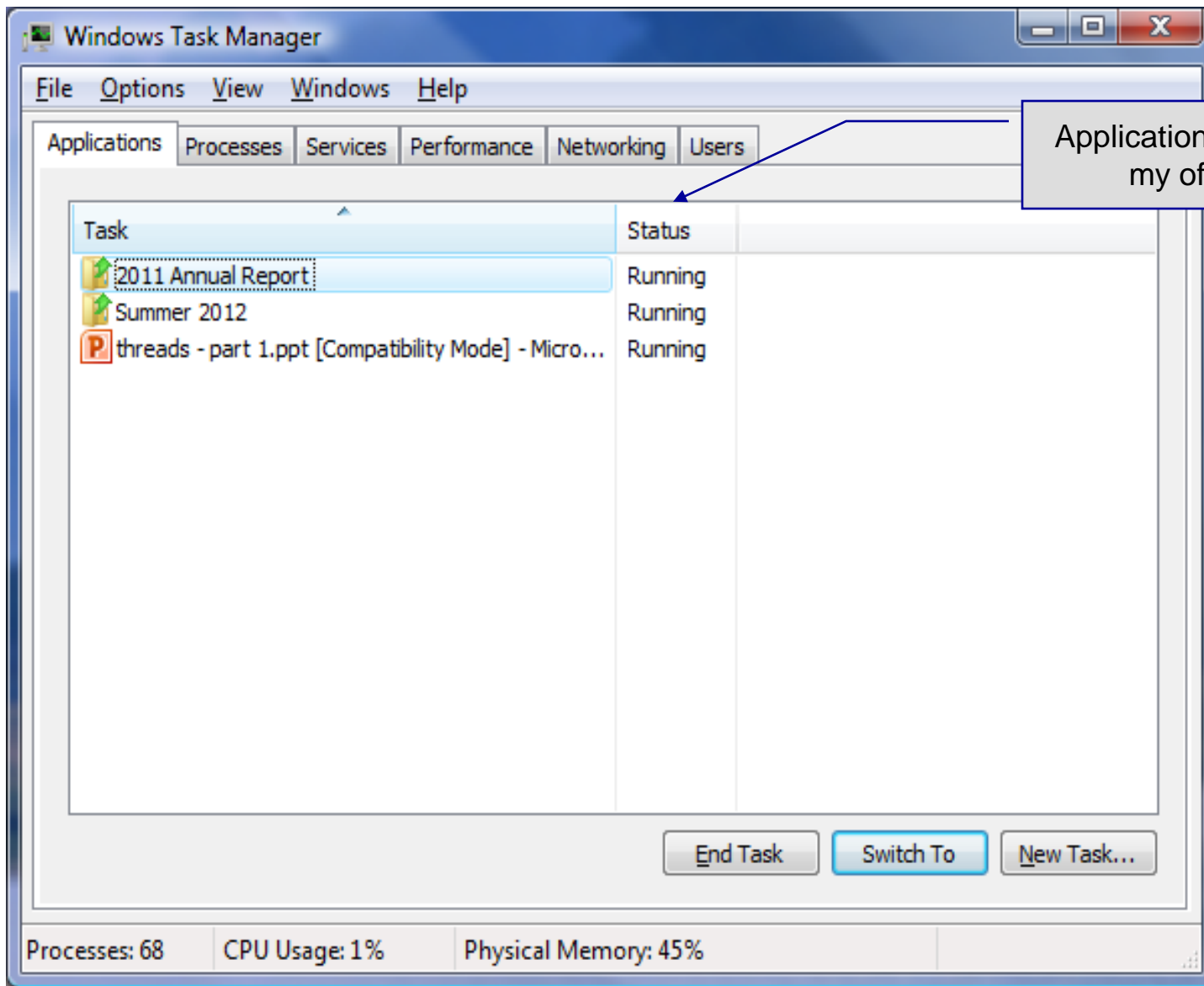
Windows Task Manager

File Options View Help

Applications | Processes | Services | Performance | Networking | Users

Image Name	User Name	CPU	Memory (...	Description
services.exe	SYSTEM	00	2,504 K	Services and Controll...
svchost.exe	SYSTEM	00	2,500 K	Host Process for Win...
svchost.exe	LOCAL SERVICE	00	2,424 K	Host Process for Win...
System	SYSTEM	00	2,280 K	NT Kernel & System
svchost.exe	SYSTEM	00	2,184 K	Host Process for Win...
lsm.exe	SYSTEM	00	2,140 K	Local Session Manag...
taskmgr.exe	Lei Wei	00	2,116 K	Windows Task Manager
winlogon.exe	SYSTEM	00	2,068 K	Windows Logon Appli...
svchost.exe	NETWORK SERV...	00	2,052 K	Host Process for Win...
hqtray.exe	Lei Wei	00	1,872 K	VMware Host Networ...
mDNSResponder.exe	SYSTEM	00	1,836 K	Bonjour Service
taskeng.exe	SYSTEM	00	1,740 K	Task Scheduler Engine
iPodService.exe	SYSTEM	00	1,688 K	iPodService Module (...
WUDFHost.exe	LOCAL SERVICE	00	1,676 K	Windows Driver Foun...
csrss.exe	SYSTEM	00	1,536 K	Client Server Runtim...

☑ Show processes from all users
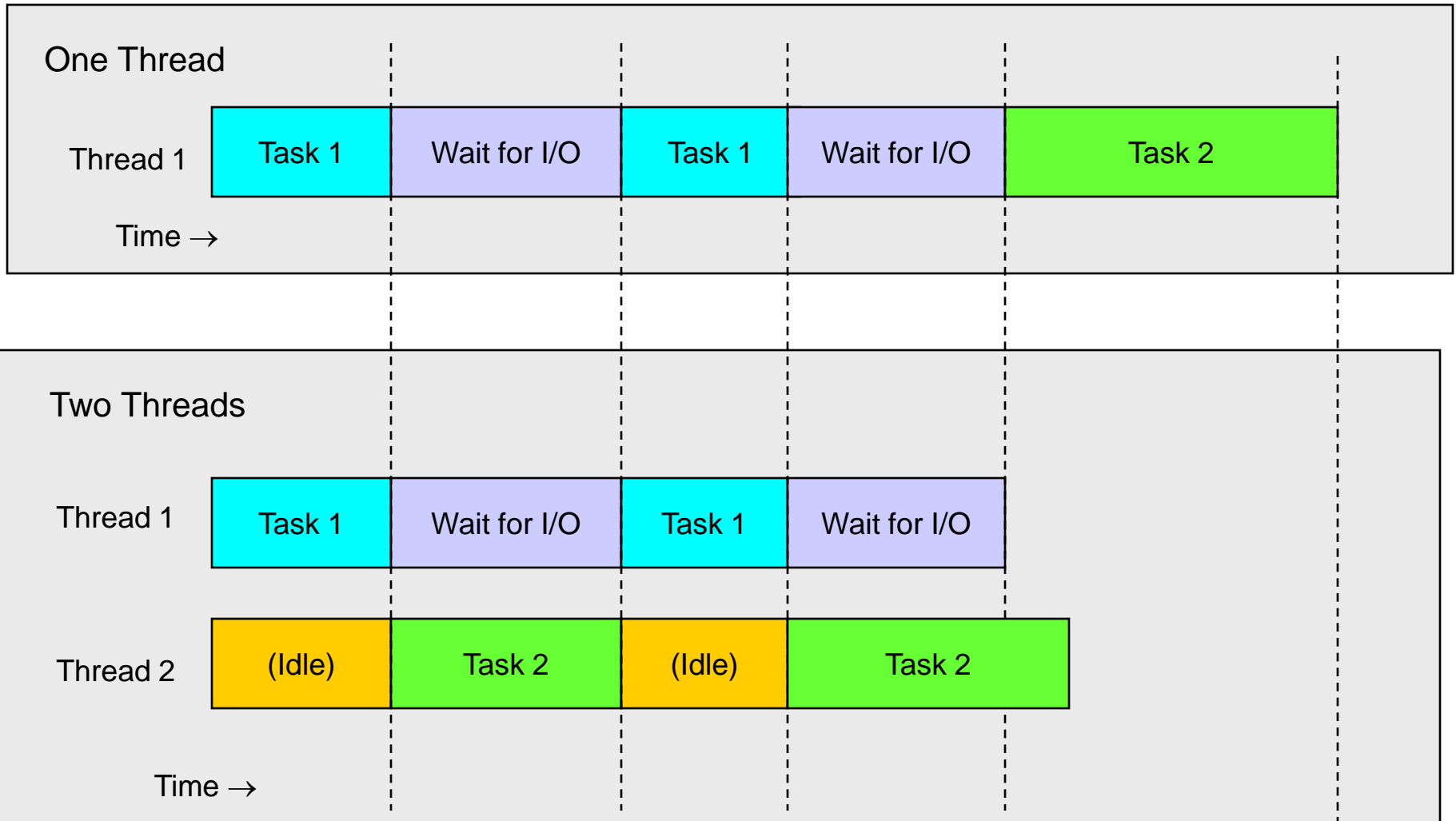
End Process

Processes: 66   CPU Usage: 1%   Physical Memory: 45%

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

CPU working hard!!!

Using Threads To Improve Performance

One Thread

Thread 1 | Task 1 | Wait for I/O | Task 1 | Wait for I/O | Task 2

Time →

Two Threads

Thread 1 | Task 1 | Wait for I/O | Task 1 | Wait for I/O

Thread 2 | (Idle) | Task 2 | (Idle) | Task 2

Time →

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.

A multithreaded program ends when all of its individual flows of control (threads) end.

Multithreaded Program
{
    statement 1;
    statement 2;
    …
    statement x;
    …
    statement y;
    …
    statement z;
}

This statement starts thread B. After starting the thread, the program continues with the next statement.

Thread A
{
    A statement 1;
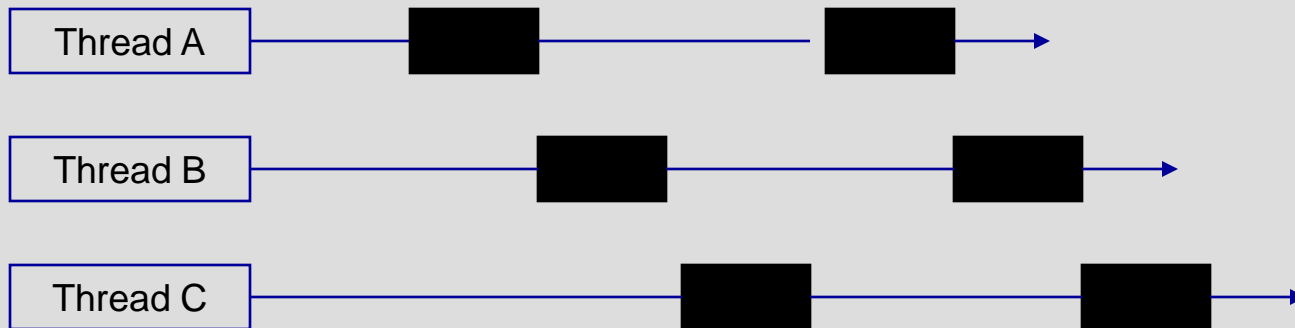   A statement 2;
    …
    A statement m;
    …
    A statement n;
}

This statement in thread A starts thread C. Thread A continues with next statement.

Thread C
{
    C statement 1;
    C statement 2;
    …
    C statement t;
}

Thread B
{
    B statement 1;
    B statement 2;
    …
    statement r;
}

Thread A
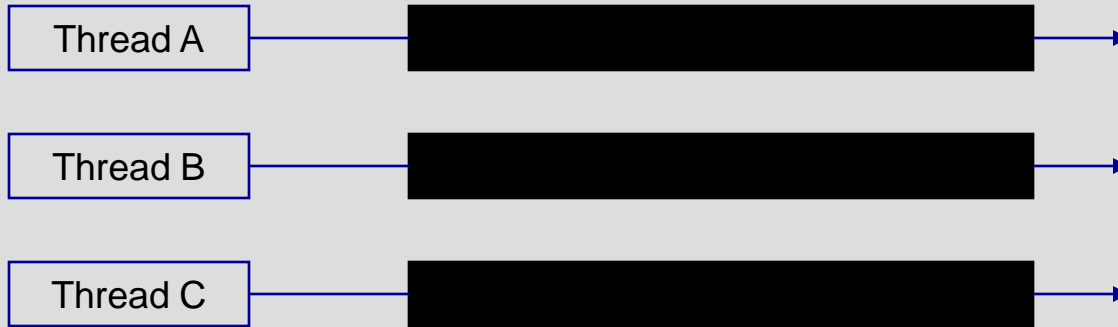
Thread B

Thread C

Thread Execution in a Multiprocessor Environment

Thread A

Thread B

Thread C

Thread Execution in a Uniprocessor Environment

The Java Thread Class

java.lang.Runnable

java.lang.Thread

+Thread()

+Thread (target: Runnable)

+run(): void

+start(): void

+interrupt(): void

+isAlive(): boolean

+setPriority(p: int): void

+join(): void

+sleep(millis: long): void

+yield(): void

+isInterrupted(): boolean

+currentThread(): Thread

Creates a default thread.

Creates a new thread to run the target object

Invoked by the JVM to execute the thread. You must override this method and provide the code you want your thread to execute in your thread class. This method is never directly invoked by a the runnable object in a program, although it is an instance method of a runnable object.

Starts the thread that causes the run() method to be invoked by the JVM

Interrupts this thread. If the thread is blocked, it is ready to run again.

Tests if the thread is currently running.

Sets priority p (ranging from 1 to 10) for this thread.

Waits for this thread to finish.

Puts the runnable object to sleep for a specified time in milliseconds.

Causes this thread to temporarily pause and allow other threads to execute

Tests if the current thread has been interrupted

Returns a reference to the currently executing thread object.

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.

```java
//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
```
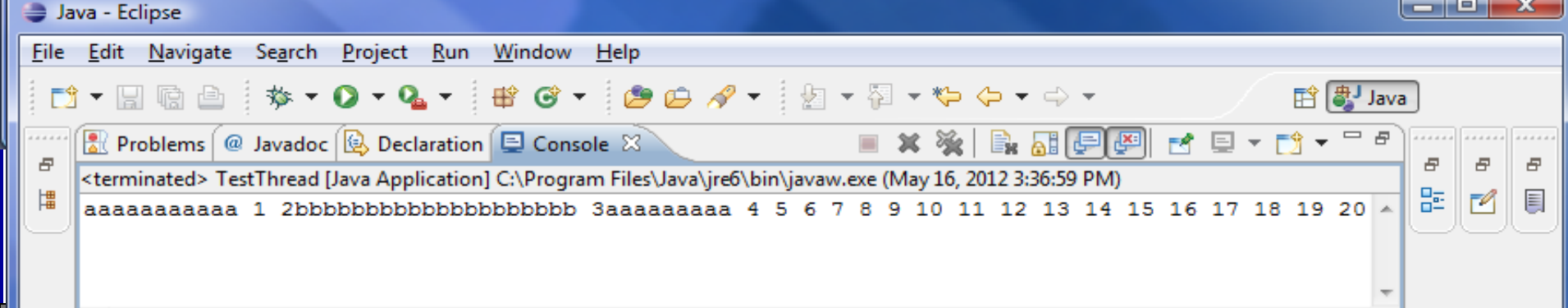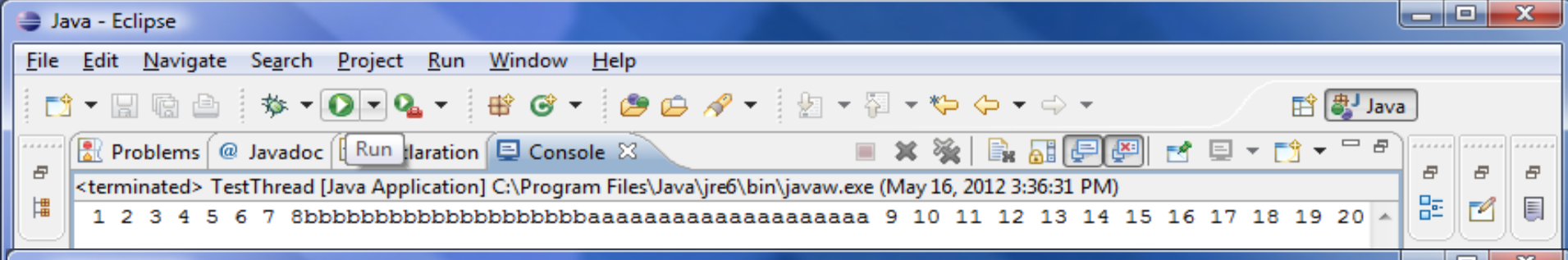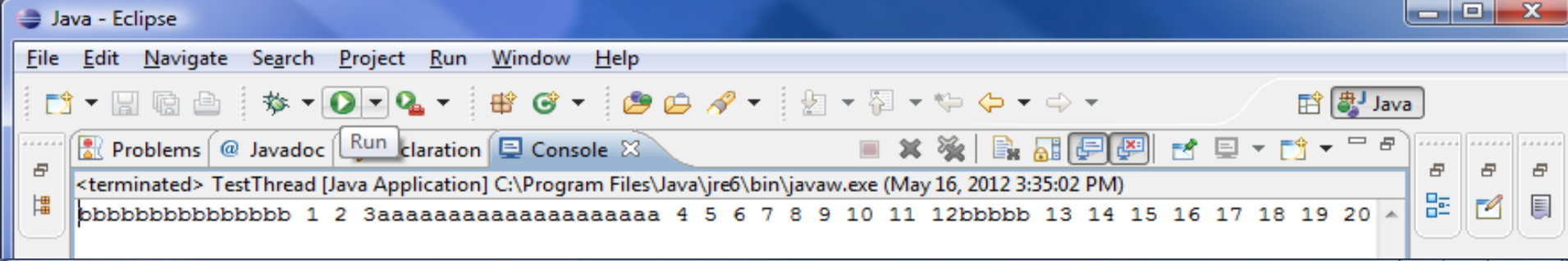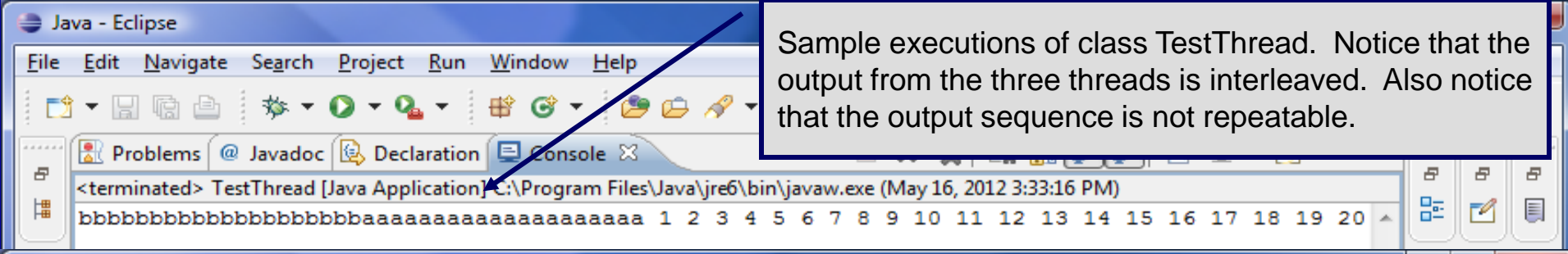
Overriding the run method
in the Thread class

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

```java
//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
```

Override the run method for both types of threads.

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

Java - Eclipse

File   Edit   Navigate   Search   Project   Run   Window   Help

Problems   @ Javadoc   Declaration   Console

<terminated> TestRunnable [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 16, 2012 3:38:00 PM)

1bbbbbbbbbbbbbbbbbbbaaaaaaaaaaa 2aaaaaaaaa 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Java - Eclipse

File   Edit   Navigate   Search   Project   Run   Window   Help

Problems   @ Javadoc   Declaration   Console

<terminated> TestRunnable [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 16, 2012 3:38:49 PM)

aaaaaaaaaaaabbbbbbbbbbbbbbbbbbbb 1aaaaaaaaa 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Java - Eclipse

File   Edit   Navigate   Search   Project   Run   Window   Help

Problems   @ Javadoc   Run   claration   Console

<terminated> TestRunnable [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 16, 2012 3:39:03 PM)

1 2 3 4 5aaaaaaaaaaaaaaaaaaaabbbbb 6bbbbbb 7bbbbbbbbb 8 9 10 11 12 13 14 15 16 17 18 19 20

Java - Eclipse

File   Edit   Navigate   Search   Project   Run   Window   Help

Problems   @ Javadoc   Declaration   Console

<terminated> TestRunnable [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 16, 2012 3:39:55 PM)

aaaaaaaaaaaabbbbbbbbbbbbbbbbbbbb 1a 2 3 4 5 6 7 8aaaaaaaaa 9 10 11 12 13 14 15 16 17 18 19 20

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).

```java
//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.
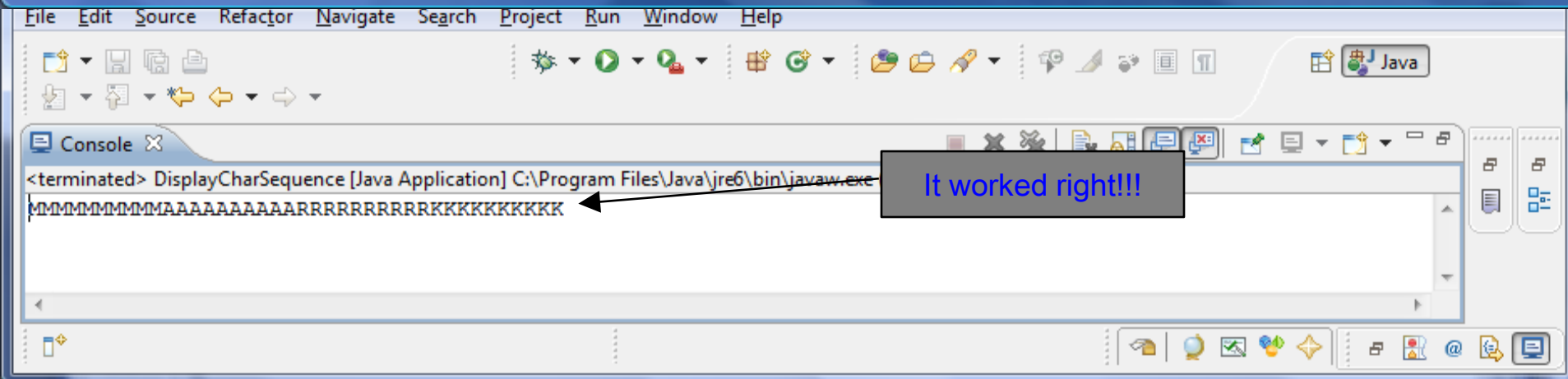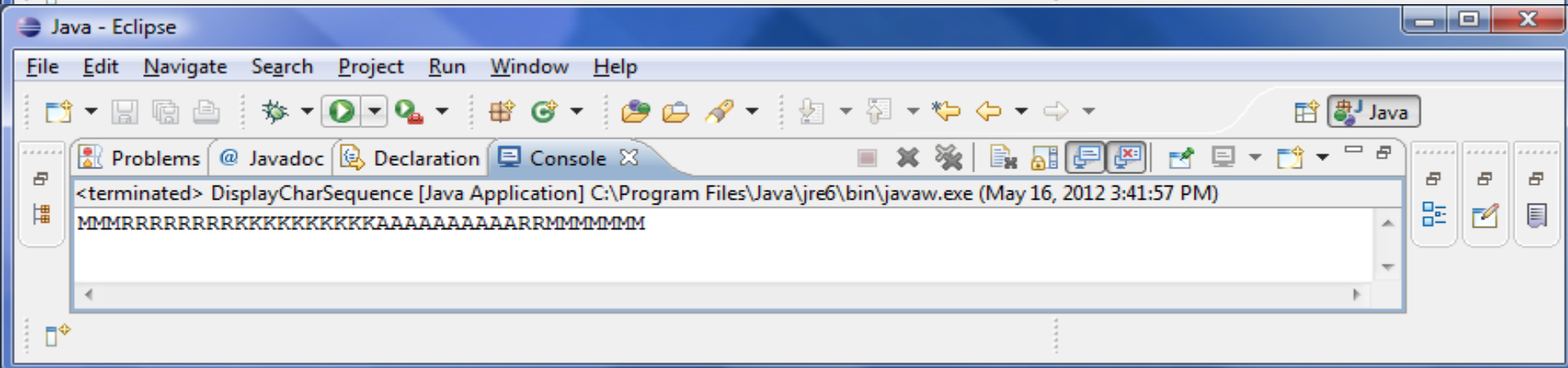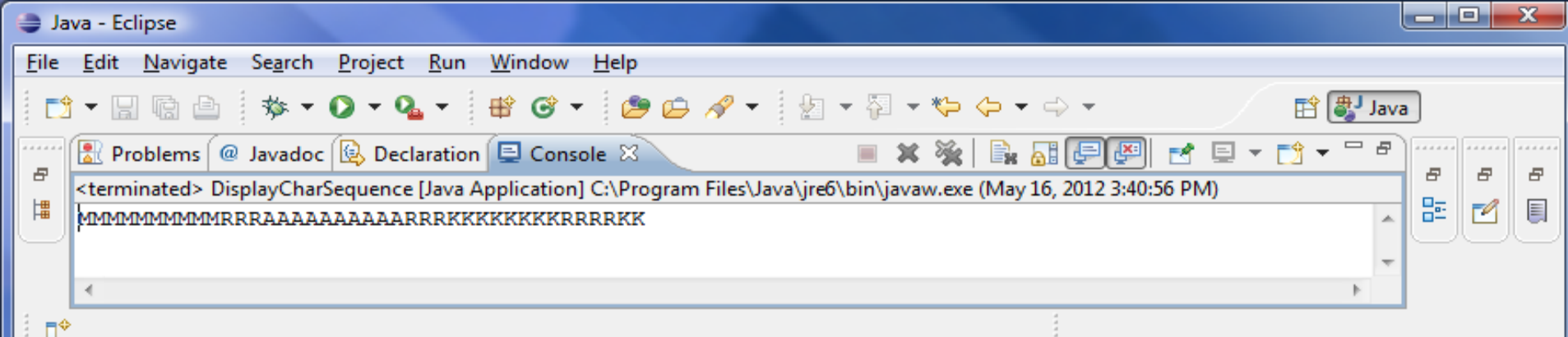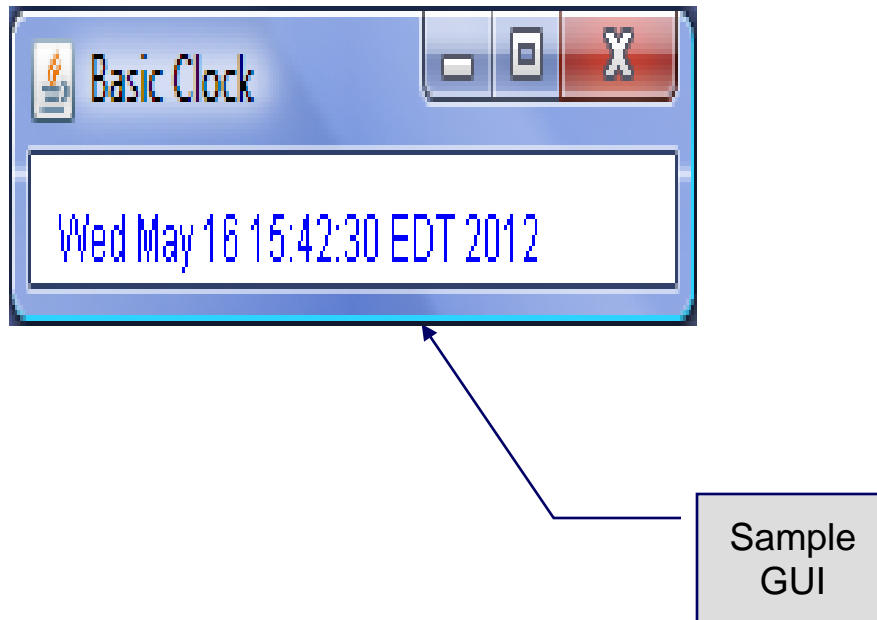
Example – Repeated Thread Execution

- 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

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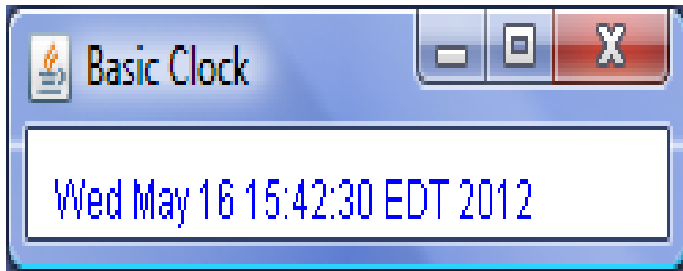
Two tasks: (1) configure the GUI and (2) schedule the thread to update the GUI-clock every second.

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

Basic Clock

Wed May 16 15:42:30 EDT 2012

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

```java
    //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();

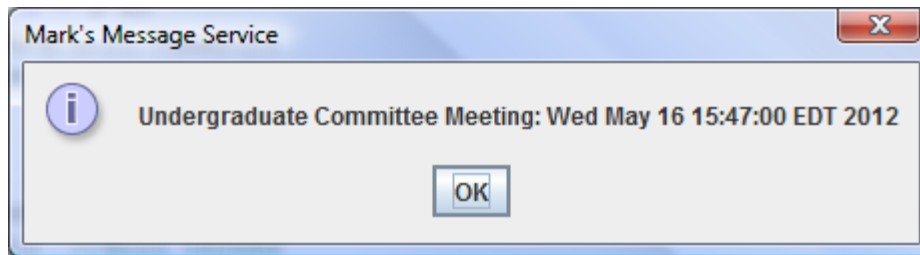   - javax.swing.Timer t2 = new javax swing.Timer();

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.
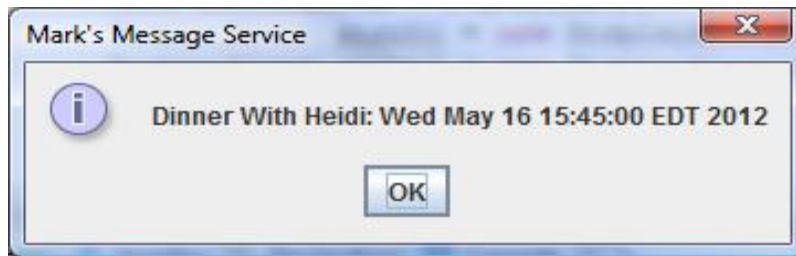
Sample DisplayAlert Window

This window pops up 2 minutes later

File   Edit   Source   Refactor   Navigate   Search   Project   Run   Window   Help

WelcomeServlet.java | TestThread.java | TestRunnable.java | BasicClock.java | DisplayAlert.java ✕ | » 4

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

public class DisplayAlert extends TimerTask {
    //instance variables
    private String message;
    private java.util.Timer timer;
    //constructor
    public DisplayAlert(String s, Date t){
        message = s + ": " + t;
        timer = new java.util.Timer();
        timer.schedule(this, t);
    }
    //execute thread
    public void run() {
        JOptionPane.showMessageDialog(null, message, "Mark's Message Service", JOptionPane.INFORMATION_ME
        timer.cancel();  //kill thread
    }
    public static void main(String[] args) {
        Calendar c = Calendar.getInstance();
        c.set(Calendar.HOUR_OF_DAY, 15);
        c.set(Calendar.MINUTE,45);
        c.set(Calendar.SECOND, 0);
        Date lunchTime = c.getTime();
        c.set(Calendar.HOUR_OF_DAY, 15);
        c.set(Calendar.MINUTE, 47);
        c.set(Calendar.SECOND, 0);
        Date meetingTime = c.getTime();
        DisplayAlert alert1 = new DisplayAlert("Dinner With Heidi", lunchTime);
        DisplayAlert alert2 = new DisplayAlert("Undergraduate Committee Meeting", meetingTime);
```

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

Writable   Smart Insert   28 : 34

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.

File   Edit   Source   Refactor   Navigate   Search   Project   Run   Window   Help

DisplayCharSequence.   BasicClock.java   DisplayAlert.java   ShowSleeping.java
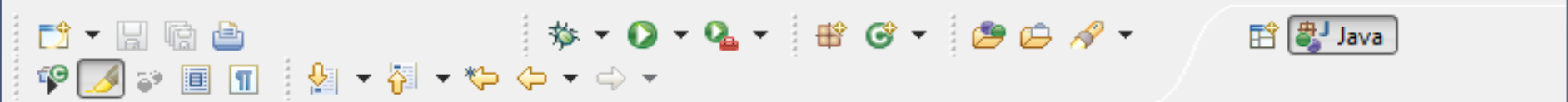
```java
//Illustrates putting a process to sleep
import java.util.*;

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.

Problems   @ Javadoc   Declaration

Writable   Smart Insert   1:1

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.)

1

The thread constructor is called to create a new instance of the Thread class.

NEW

Blocked

Runnable

Terminated

Waiting

Life Cycle of a Thread (cont.)

2

The start() method is invoked to designate the thread as runnable.

NEW

start()

Blocked

Runnable

Terminated

Waiting

Life Cycle of a Thread (cont.)

NEW

Blocked

③

The Java thread scheduler runs the thread as the processor becomes available

Runnable

Terminated

Waiting

Life Cycle of a Thread (cont.)

NEW

Blocked

Runnable

Terminated

Waiting

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.

5

Life Cycle of a Thread (cont.)

NEW

Blocked

Runnable

Waiting

Terminated

6

The thread ends when the run method terminates.
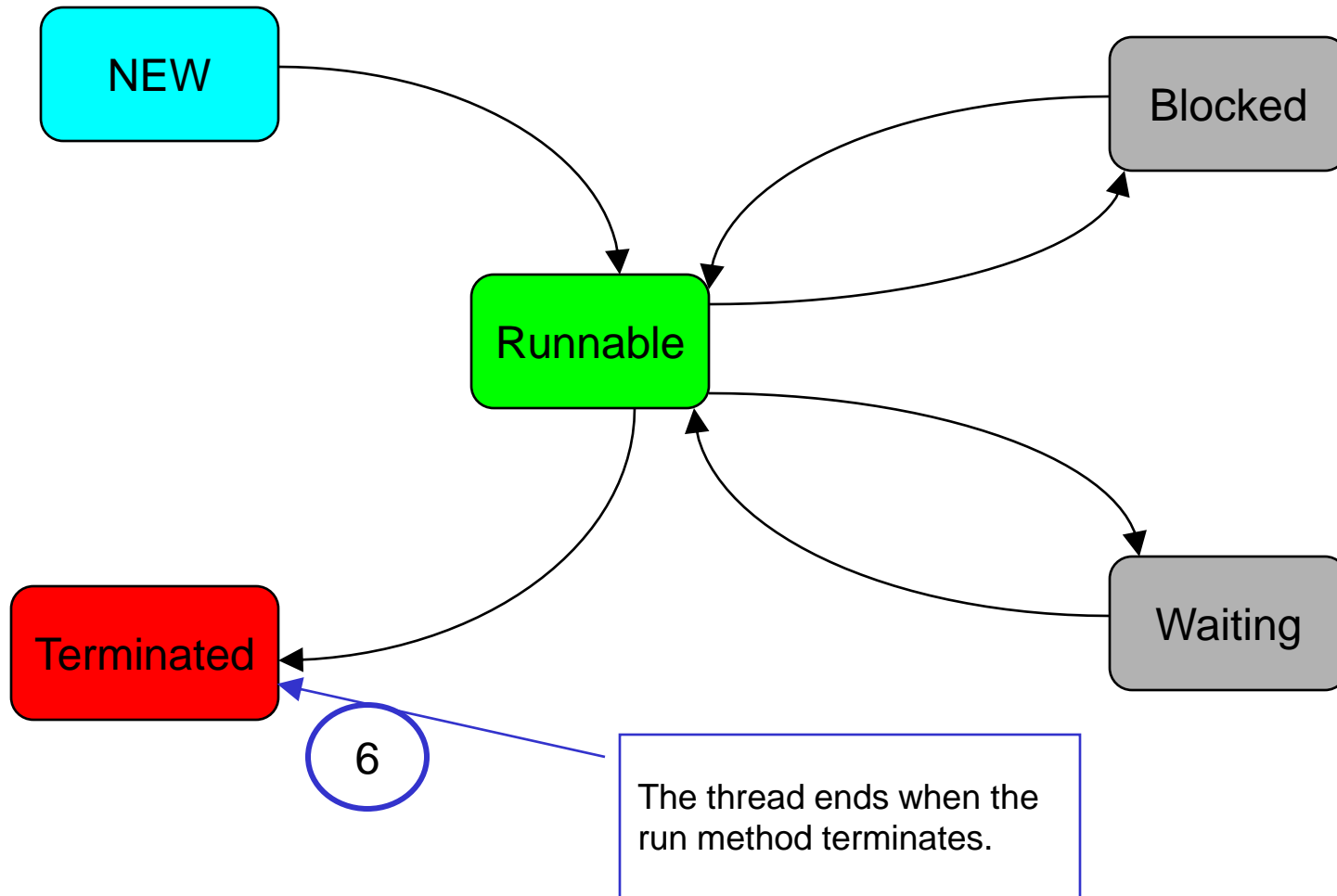
Life Cycle of a Thread (cont.)

The thread can become blocked for various reasons and will not run again until it is returned to the runnable state.

4

NEW

Blocked

Runnable

Waiting

Terminated
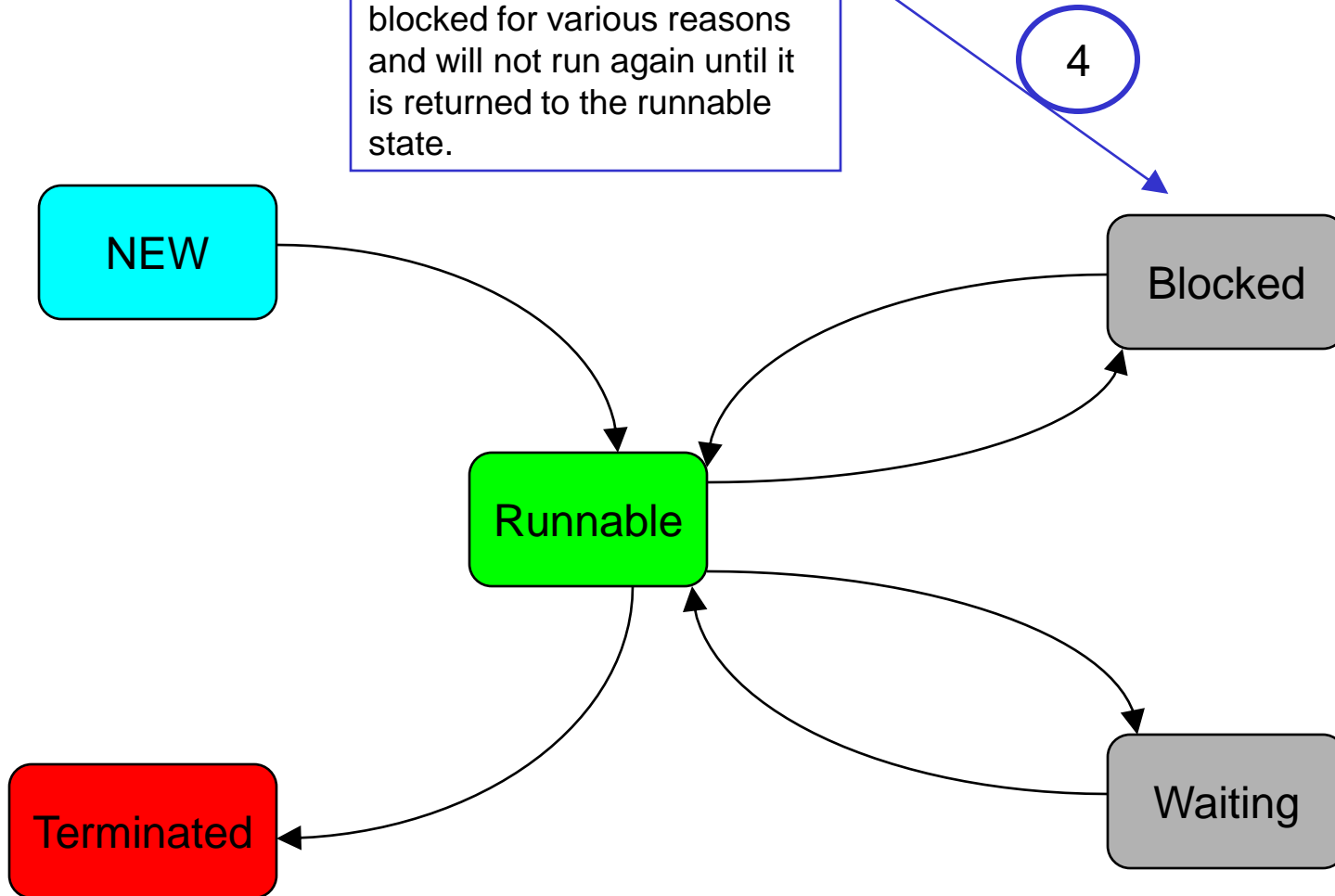
Summary of States In The Life Cycle of a Thread

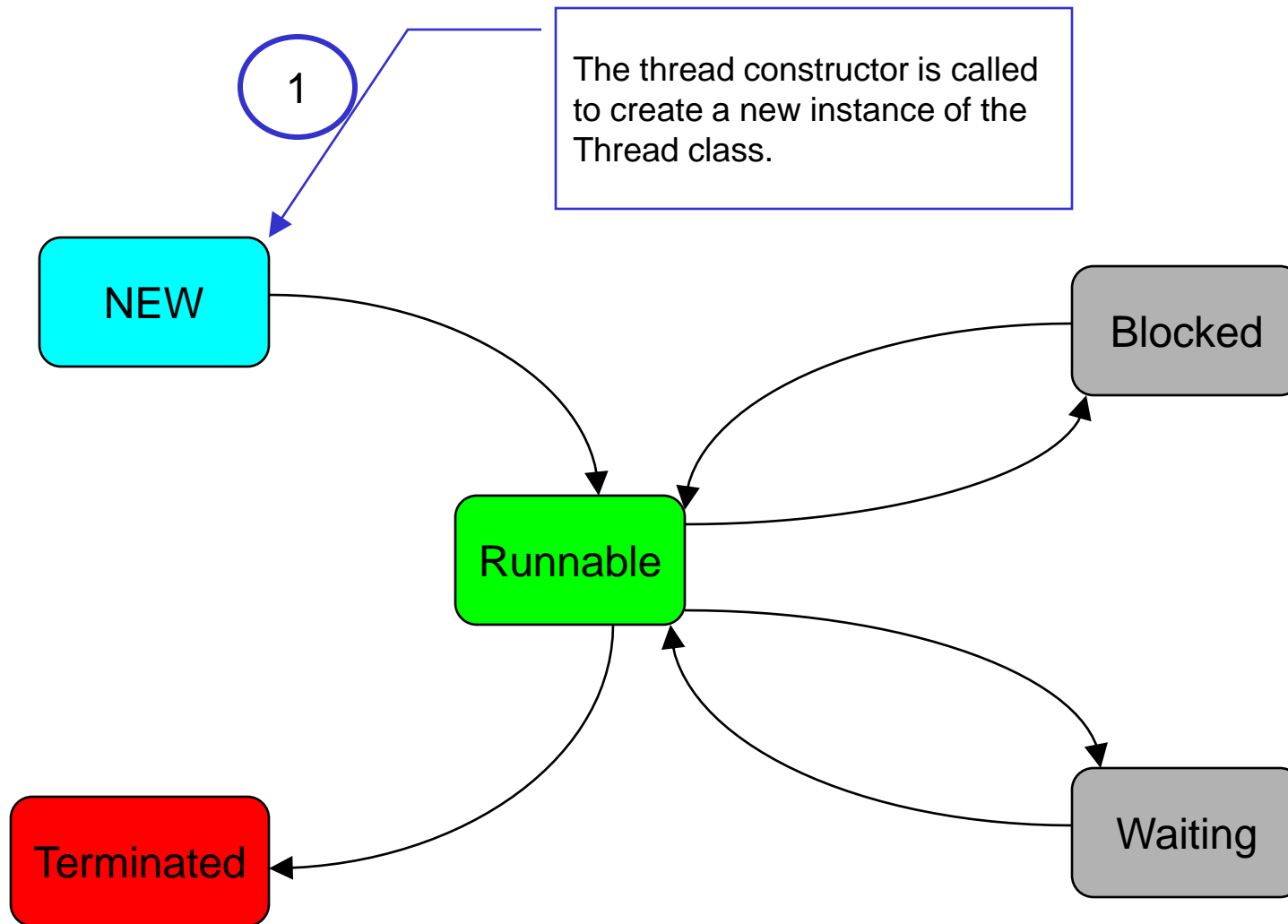| State	Description
New	The thread has been created (its constructor has been invoked), but not yet started.
Runnable	The thread's start() method has been invoked and the thread is available to be run by the thread scheduler. A thread in the Runnable state may actually be running, or it may be waiting 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
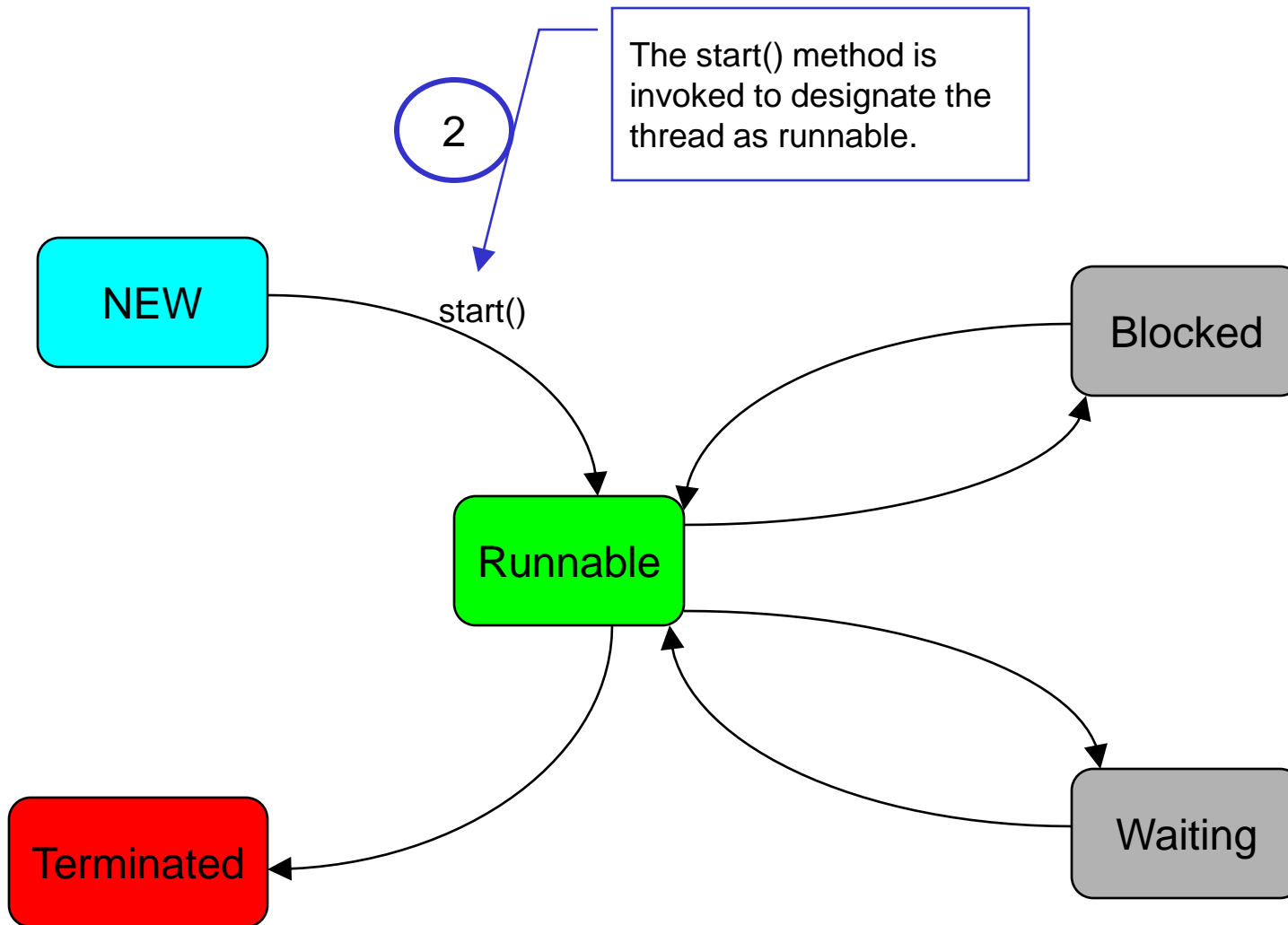
• At any given point in time, a thread is said to be in one of several thread states as illustrated in the diagram below.

Thread created

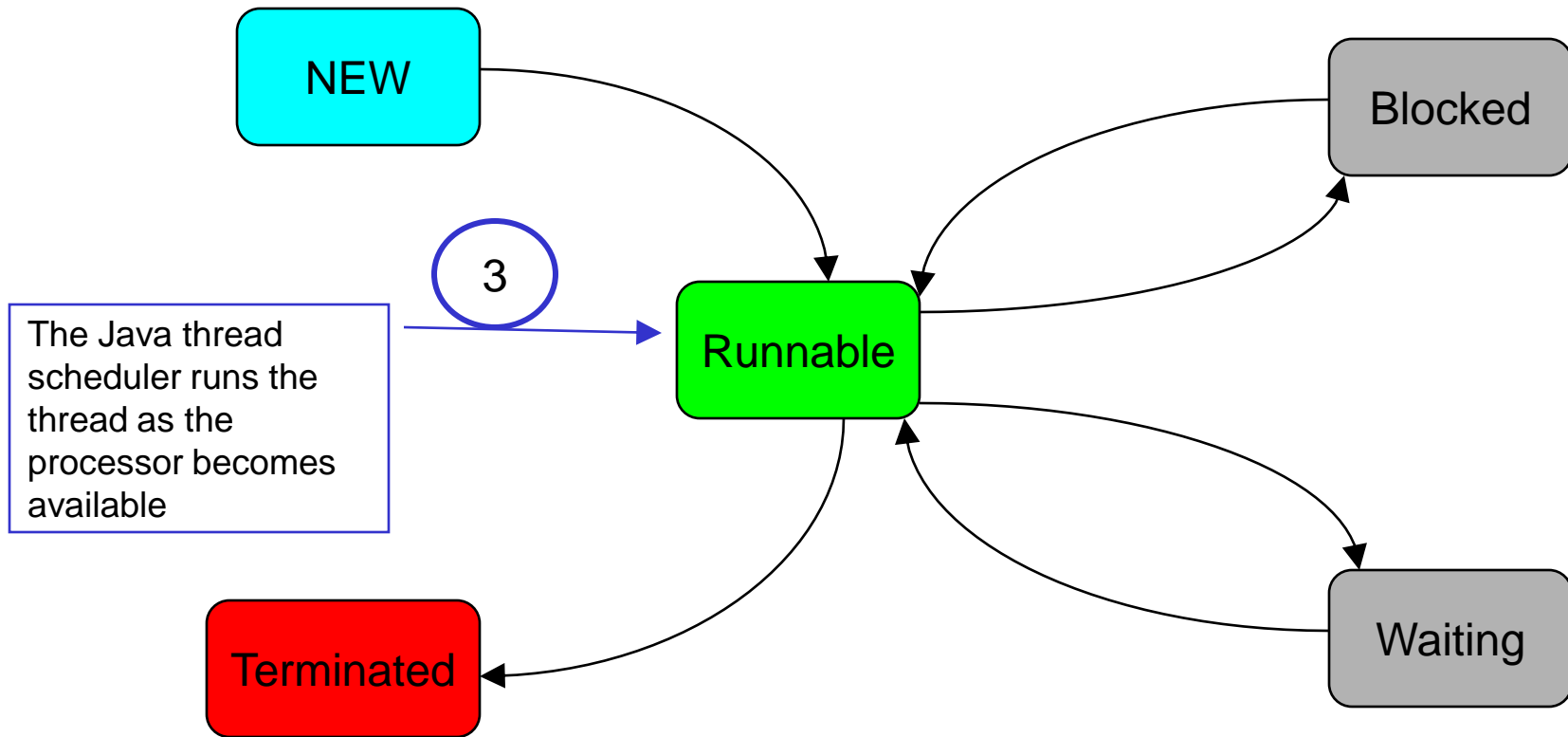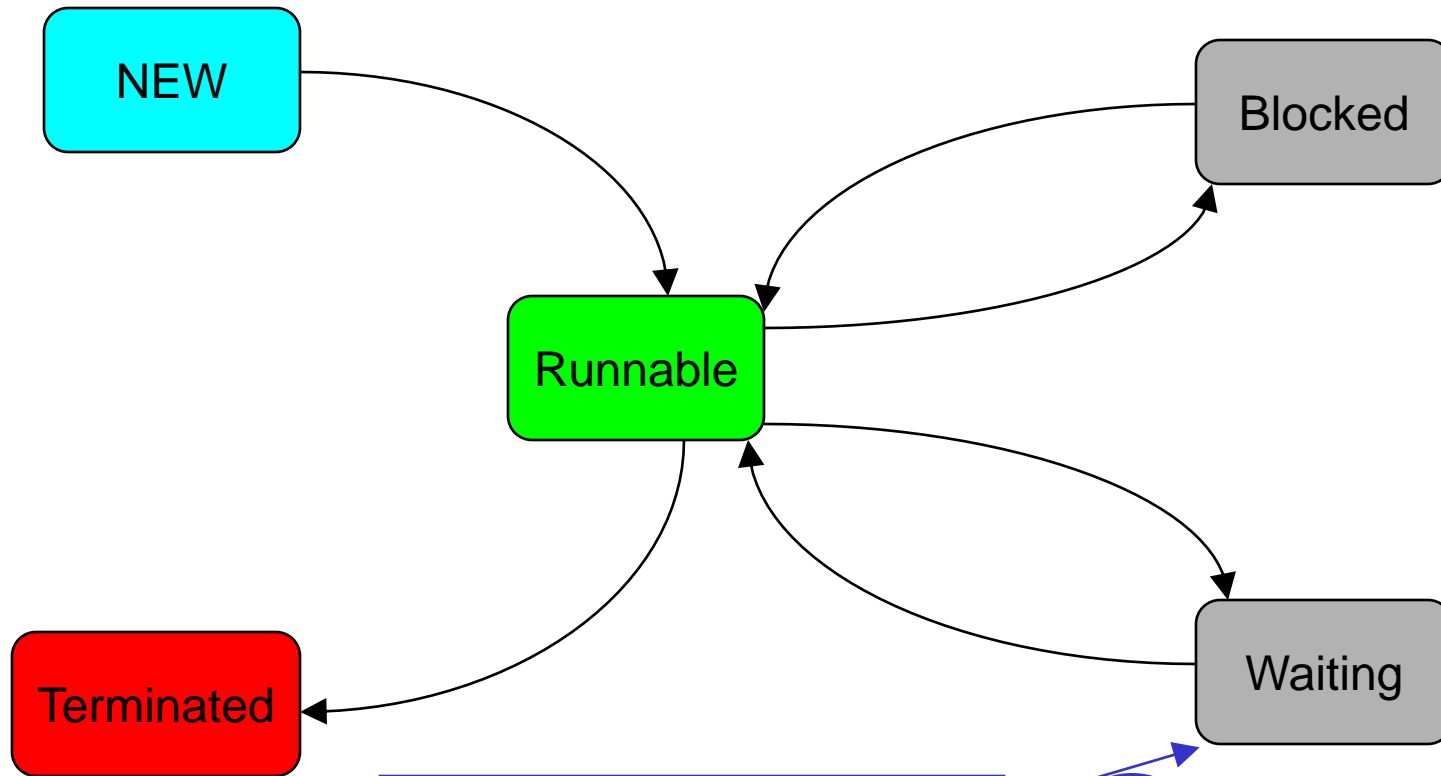new →start()→ ready

running

yield() or timeout
run()
run() returns
terminated

join()
interrupt()
sleep()
wait()

target finished

Wait for target to finish

Wait for timeout

Wait to be notified

timeout

notify() or notifyAll()

interrupt()

blocked
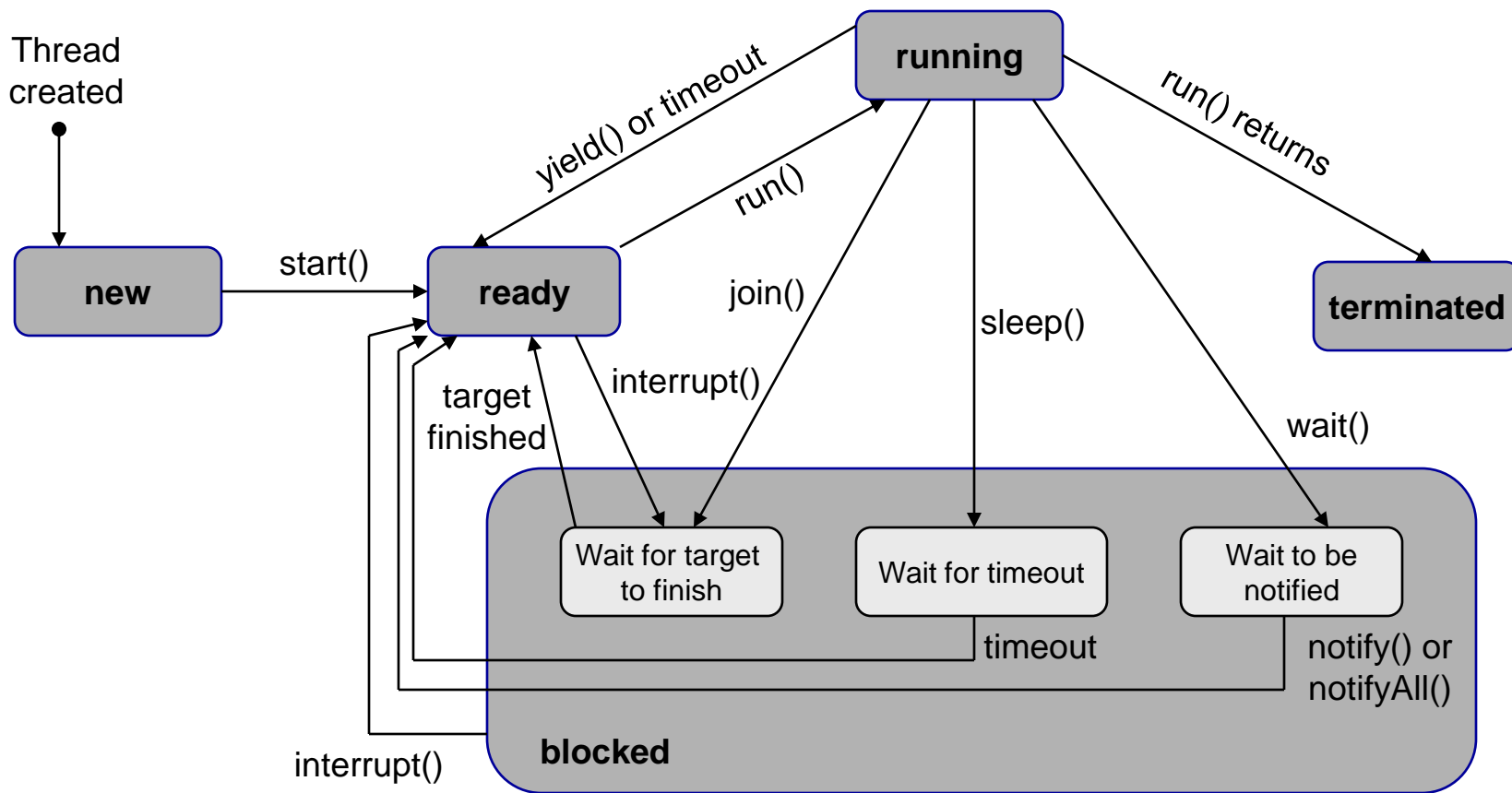
Life Cycle of a Thread (cont.)

- 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

Ready threads

Thread.MAX_PRIORITY

Priority 10	A → B
Priority 9	C
Priority 8	
Priority 7	D → E → F
Priority 6	G

Thread.NORM_PRIORITY

Priority 5	H → I
Priority 4	
Priority 3	
Priority 2	J → K

Thread.MIN_PRIORITY

| Priority 1 | |
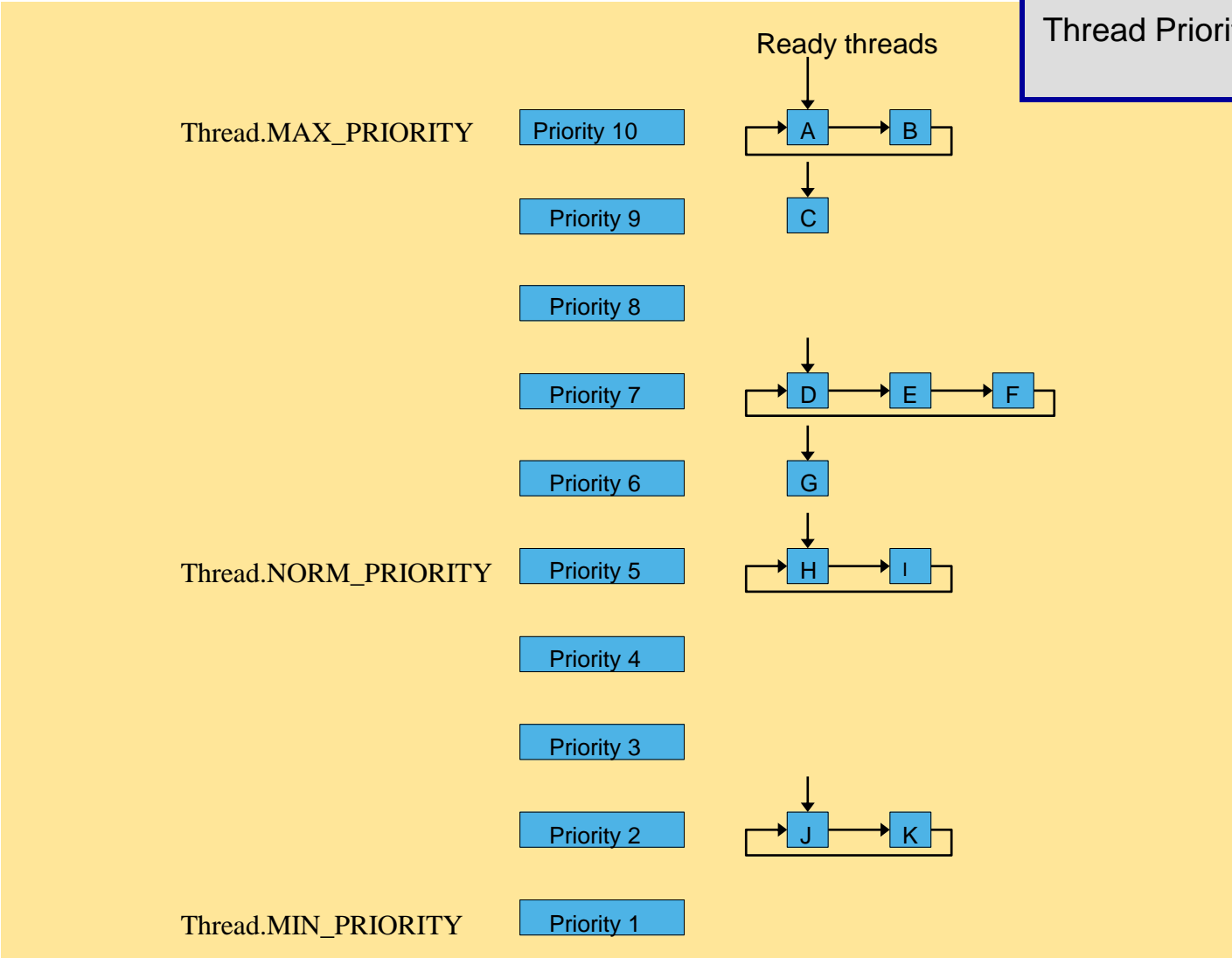
Creating and Executing Threads

- In J2SE 5.0 and above, 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 `Runnable` interface.

- 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 manages 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

```java
// 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

```java
// 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

More Sophisticated Threading

- All of the example applications up to this point involved threads which were unsynchronized. None of the threads actually needed to communicate with one another and they did not require access to a shared object.

- The threads we've seen so far fall into the category of unrelated threads. These are threads which do different tasks and do not interact with one another.

- A slightly more complex form of threading involves threads which are related but unsynchronized. In this case, multiple threads operate on different pieces of the same data structure. An example of this type of threading is illustrated on the next page with a threaded program to determine if a number is prime.

```
//class for threaded prime number testing
//no inheritance issues so using the simple form of thread creation
class testRange extends Thread {
   static long possPrime;
    long from, to; //test range for a thread
     //constructor
     //record the number to be tested and the range to be tried
     testRange(int argFrom, long argpossPrime) {
          possPrime = argpossPrime;
          if (argFrom ==0) from = 2; else from = argFrom;
          to=argFrom+99;
     }
     //implementation of run
     public void run() {
       for (long i=from; i <= to && i<possPrime; i++) {
           if (possPrime % i == 0) {
                //i divides possPrime exactly
                System.out.println("factor " + i + " found by thread " + getName() );
                break;  //exit for loop immediately
           }
           yield(); //suspend thread
       }
    }
}
```

```
//driver class to demonstrate threaded prime number tester
public class testPrime {
   public static void main (String s[]) {
              //number to be tested for primality is entered as a command line argument
              //examples: 5557 is prime, 6841 is prime, 6842 is not prime
              long possPrime = Long.parseLong(s[0]);
                    int centuries = (int) (possPrime/100) + 1;
                    for (int i=0; i<centuries;i++) {
                        new testRange(i*100, possPrime).start();
                    }
            }
}
```

- This is an example of related but unsynchronized threads. In this case the threads are related since they are each working on a piece of the same data, but approach it from a slightly different perspective. However, they are unsynchronized since they do not share information.

factor 256.0 found by thread Thread-2
factor 512.0 found by thread Thread-5
factor 128.0 found by thread Thread-1
factor 2.0 found by thread Thread-0
factor 1024.0 found by thread Thread-10
All threads complete...termination!

2048 and 6842 are not prime – their factors are shown by the thread which discovered the factor.

factor 311.0 found by thread Thread-3
factor 622.0 found by thread Thread-6
factor 2.0 found by thread Thread-0
factor 3421.0 found by thread Thread-34
All threads complete...termination!

All threads complete...termination!

5557 is prime so no thread will find a factor

Related and Synchronized Threads

- The most complicated type of threaded application involves threads which interact with each other. These are related synchronized threads (also referred to as cooperating threads).

- Without synchronization when multiple threads share an object and that object is modified by one or more of the threads, indeterminate results may occur. This is known as a data race or race condition.

- The following example illustrates a race condition. In this example, we simulate a steam boiler and the reading of its pressure. The program starts 10 unsynchronized threads which each read the pressure of the boiler and if it is found to be below the safe limit, the pressure in the boiler is increased by 15psi. If the pressure is found to already be above the safe limit, the pressure is not increased. Looking at the results you can clearly see the problem with this approach.

```java
// class to simulate a steam boiler to illustrate a race condition
//in unsynchronized threads
public class SteamBoiler1 {
    static int pressureGauge = 0;
     static final int safetyLimit = 50;


    public static void main(String [] args) {
        pressure1 []psi = new pressure1[10];
        for (int i = 0; i < 10; i++) {
            psi[i] = new pressure1();
            psi[i].start();
        }
        //we now have 10 threads in execution to monitor the pressure
        try {
          for (int i = 0; i < 10; i++)
              psi[i].join();  //wait for the thread to finish
              //psi[i].sleep(200);
        }
        catch (Exception e) { } //do nothing
        System.out.println();
        System.out.print("Gauge reads " + pressureGauge + ", the safe limit is " + safetyLimit);
        if (pressureGauge > safetyLimit)
            System.out.println("...B O O M ! ! !");
        else System.out.println("...System OK!");
    }
}
```

Class to Simulate a Steam Boiler – Pressure Gauge

```java
//thread class to raise the pressure in the Boiler
class pressure1 extends Thread {
    void RaisePressure() {
        if (SteamBoiler1.pressureGauge < SteamBoiler1.safetyLimit-15) {
            //wait briefly to simulate some calculations
            try {sleep(100); } catch (Exception e) { }
            SteamBoiler1.pressureGauge+= 15; //raise the pressure 15 psi
            System.out.println("Thread " + this.getName() + " finds pressure within limits - increases pr
        }
        else
            System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
    }
    public void run() {
        RaisePressure();  //this thread is to raise the pressure
    }
}
```

Thread Class to Read Steam Boiler Pressure Gauge and Increase the Pressure if Within Range
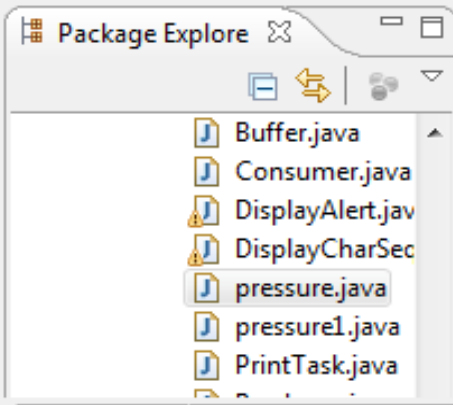
Java - Threading Examples/src/pressure1.java - Eclipse

File   Edit   Source   Refactor   Navigate   Search   Project   Run   Window   Help

This is what caused the race condition to occur.

Package Explorer ☒

- Buffer.java
- Consumer.java
- DisplayAlert.jav
- DisplayCharSec
- pressure.java
- pressure1.java
- PrintTask.java

J pressure1.java ☒   J SteamBoiler1.java   J SteamBoiler.java   J pressure.java

```java
//thread class to raise the pressure in the Boiler
class pressure1 extends Thread {
    void RaisePressure() {
        if (SteamBoiler1.pressureGauge < SteamBoiler1.safetyLimit-15
            //wait briefly to simulate some calculations
            try {sleep(100); } catch (Exception e) { }
            SteamBoiler1.pressureGauge+= 15; //raise the pressure 15
            System.out.println("Thread " + this.getName() + " finds p
        }
```

Console ☒

<terminated> SteamBoiler1 [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 16, 2012 4:08:47 PM)

```
Thread Thread-0 finds pressure within limits - increases pressure
Thread Thread-9 finds pressure within limits - increases pressure
Thread Thread-7 finds pressure within limits - increases pressure
Thread Thread-8 finds pressure within limits - increases pressure
Thread Thread-3 finds pressure within limits - increases pressure
Thread Thread-2 finds pressure within limits - increases pressure
Thread Thread-1 finds pressure within limits - increases pressure
Thread Thread-6 finds pressure within limits - increases pressure
Thread Thread-5 finds pressure within limits - increases pressure
Thread Thread-4 finds pressure within limits - increases pressure

Gauge reads 150, the safe limit is 50...B O O M ! ! !
```

Output From Execution Illustrating the Race Condition

Java - Threading Examples/src/pressure1.java - Eclipse

File   Edit   Source   Refactor   Navigate   Search   Project   Run   Window   Help

See. . ., I told you so!

pressure1.java   SteamBoiler1.java   SteamBoiler.java   pressure.java

```java
//thread class to raise the pressure in the boiler
class pressure1 extends Thread {
    void RaisePressure() {
        if (SteamBoiler1.pressureGauge < SteamBoiler1.safetyLimit-15) {
            //wait briefly to simulate some calculations
            //try {sleep(100); } catch (Exception e) { }
            SteamBoiler1.pressureGauge+= 15; //raise the pressure 15 psi
            System.out.println("Thread " + this.getName() + " finds pressure within limits
        }
```

Console

<terminated> SteamBoiler1 [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 16, 2012 4:09:50 PM)

```
Thread Thread-0 finds pressure within limits - increases pressure
Thread Thread-2 finds pressure within limits - increases pressure
ThreadThread-4 finds pressure too high - do nothing
Thread Thread-1 finds pressure within limits - increases pressure
ThreadThread-3 finds pressure too high - do nothing
ThreadThread-5 finds pressure too high - do nothing
ThreadThread-6 finds pressure too high - do nothing
ThreadThread-7 finds pressure too high - do nothing
ThreadThread-8 finds pressure too high - do nothing
ThreadThread-9 finds pressure too high - do nothing


Gauge reads 45, the safe limit is 50...System OK!
```

Output From Execution Illustrating No Race Condition

Interesting Note on Race Conditions

- You may remember the large North American power blackout that occurred on August 14, 2003. Roughly 50 million people lost electrical power in a region stretching from Michigan through Canada to New York City. It took three days to restore service to some areas.

- See http://en.wikipedia.org/wiki/Northeast_blackout_of_2003 (scroll down to computer failure)

- There were several factors that contributed to the blackout, but the official report highlights the failure of the alarm monitoring software which was written in C++ by GE Energy. The software failure wrongly led operators to believe that all was well, and precluded them from rebalancing the power load before the blackout cascaded out of control.

- Because the consequences of the software failure were so severe, the bug was analyzed exhaustively. The root cause was finally identified by artificially introducing delays in the code (just like we did in the previous example). There were two threads that wrote to a common data structure, and through a coding error, they could both update it simultaneously. It was a classic race condition, and eventually the program "lost the race", leaving the structure in an inconsistent state. That in turn caused the alarm event handler to spin in an infinite loop, instead of raising the alarm. The largest power failure in the history of the US and Canada was caused by a race condition bug in some threaded C++ code. Java is equally vulnerable to this kind of bug.

The Therac-25 Accidents

- Starting in 1976, the Therac-25 treatment system, built by Atomic Energy of Canada Limited (AECL) and COR MeV of France, was used to fight cancer by providing radiation to a specific part of the body in the hope of destroying tumors.

- See http://en.wikipedia.org/wiki/Therac-25 (See last line of the Problem description.)

- Six known Therac-25 accidents have been documented, all involved massive overdoses (100x normal dose) of radiation and three resulted in the death of the patient, serious long-term injury and disfigurement occurred in the other cases. Patients received an estimated 17,000 to 25,000 rads to very small body areas. By comparison, doses of 1000 rads can be fatal if delivered to the whole body.

- Analysis determined that the primary cause of the overdoses was faulty software. The software was written in assembly language and was developed and tested by the same person. The software included a scheduler and concurrency in its design. When the system was first built, operators complained that it took too long to enter the treatment plan into the computer. As a result, the software was modified to allow operators to quickly enter treatment data by simply pressing the Enter key when an input value did not require changing.

The Therac-25 Accidents (cont.)

- This modification created a synchronization error (a race condition developed) between the code that read the data entered by the operator and the code controlling the machine. As a result, the actions of the machine would lag behind the commands the operator entered. The machine appeared to administer the dose entered by the operator, but it fact had an improper setting that focused radiation at full power to a tiny spot on the body.

- The race condition was subsequently found to occur only when a certain non-typical keystroke sequence was entered (an "X" to select a 25MeV photon followed by "cursor-up" ,"E" to correctly set the 25MeV Electron mode, then "Enter"), since this sequence of keystrokes did not occur very often, the error went unnoticed for a long time.

- AECL was ultimately cited for improperly testing the software, which was only tested on site in hospitals after a machine was assembled in place.

- The designer had reused software from older Therac-6 and Therac-20 models that had hardware interlocks which masked the software defects. Some operators noted that certain situations caused the machines to display MALFUNCTION followed by a number between 1 and 64 on the display screen. However, the user manual did not explain nor even address error codes, so the operators pressed the "P" key (for proceed), to override the warning and proceed with the treatment.

Thread Synchronization

- To prevent a race condition, access to the shared object must be properly synchronized.

  - Lost update problem: one thread is in the process of updating the shared value and another thread also attempts to update the value.

  - Even worse is when only part of the object is updated by each thread in which case part of the object reflects information from one thread while another part of the same object reflects information from another thread.

- The problem can be solved by giving one thread at a time exclusive access to code that manipulates the shared object. During that time, other threads desiring to manipulate the object must be forced to wait.

Thread Synchronization (cont.)

- When the thread with exclusive access to the object finishes manipulating the object, one of the blocked threads will be allowed to proceed and access the shared object.

    - The next selected thread will be based on some protocol. The most common of these is simply FCFS (priority-queue based).

- In this fashion, each thread accessing the shared object excludes all other threads from accessing the object simultaneously. This is the process known as mutual exclusion.

- Mutual exclusion allows the programmer to perform thread synchronization, which coordinates access to shared objects by concurrent threads.

Synchronization Techniques

- There have been many different methods used to synchronize concurrent processes. Some of the more common ones are:

    - Test and Set Instructions. All general purpose processors now have this kind of instruction, and it is used to build higher-level synchronization constructs. Test and set does not block, that must be built on top of it.

    - p and v semaphores. Introduced by Dijkstra in the 1960's and was the main synchronization primitive for a long time. Its easy to build semaphores from test and set instructions. Semaphores are low-level and can be hard for programmers to read and debug. For your information the p is short for the Dutch words *proberen te verlangen* which means to "try to decrement" and the v stands for *verhogen* which means to increment.

Synchronization Techniques (cont.)

– Read/write Locks. These are also commonly referred to as mutexes (although some people still use the term `mutex` to refer to a semaphore.) A lock provides a simple "turnstile": only one thread at a time can be going through (executing in) a block protected by a lock. Again, it is easy to build a lock from semaphores.

– Monitors. A monitor is a higher-level synchronization construct built out of a lock plus a variable that keeps track of some related condition, such as "the number of unconsumed bytes in the buffer". It is easy to build monitors from read/write locks. A monitor defines several methods as a part of its protocol. Two of those predefined methods are `wait()` and `notify()`.

Types of Synchronization

- There are two basic types of synchronization between threads:

    1. Mutual exclusion is used to protect certain critical sections of code from being executed simultaneously by two or more threads. (Synchronization without cooperation.)

    2. Signal-wait is used when one thread need to wait until another thread has completed some action before continuing. (Synchronization with cooperation.)

- Java includes mechanisms for both types of synchronization.

- All synchronization in Java is built around locks. Every Java object has an associated lock. Using appropriate syntax, you can specify that the lock for an object be locked when a method is invoked. Any further attempts to call a method for the locked object by other threads cause those threads to be blocked until the lock is unlocked.

Thread Synchronization In Java

- Any object can contain an object that implements the `Lock` interface (package `java.util.concurrent.locks`).

- A thread calls the `Lock`'s `lock()` method to obtain the lock.

- Once a lock has been obtained by one thread the `Lock` object will not allow another thread to obtain the lock until the thread releases the lock (by invoking the `Lock`'s `unlock()` method).

- If there are several threads trying to invoke method `lock()` on the same `Lock` object, only one thread may obtain the lock, with all other threads being placed into the wait state.

An Aside on Reentrant Locks

- Class `ReentrantLock` (package java.util.concurrent.locks) is a basic implementation of the `Lock` interface.

  – The constructor for a `ReentrantLock` takes a boolean argument that specifies whether the lock has a fairness policy. If this is set to true, the ReentrantLock's fairness policy states that the longest-waiting thread will acquire the lock when it is available. If set to false, there is no guarantee as to which waiting thread will acquire the lock when it becomes available.

- Using a lock with a fairness policy helps avoid indefinite postponement (starvation) but can also dramatically reduce the overall efficiency of a program. Due to the large decrease in performance, fair locks should be used only in necessary circumstances.

Condition Variables

- If a thread that holds the lock on an object determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a condition variable.

- This removes the thread from contention for the processor by placing it in a wait queue for the condition variable and releases the lock on the object.

- Condition variables must be associated with a `Lock` and are created by invoking `Lock` method `newCondition`, which returns an object that implements the `Condition` interface.

- To wait on a condition variable, the thread can call the `Condition`'s `await()` method (see Life Cycle of a thread in previous set of notes).

Condition Variables (cont.)

- Invoking the `await()` method, immediately releases the associated `Lock` and places the thread in the wait state for that `Condition`. Other threads can then try to obtain the `Lock`.

- When a runnable thread completes a task and determines that the waiting thread can now continue, the runnable thread can call Condition method `signal()` to allow a thread in that Condition's wait queue to return to the runnable state. At this point, the thread that transitioned from the wait state to the runnable state can attempt to reacquire the Lock on the object. Of course there is no guarantee that it will be able to complete its task this time and the cycle may repeat.

Condition Variables (cont.)

- If multiple threads are in a `Condition`'s wait queue when a `signal()` is invoked, the default implementation of `Condition` signals the longest-waiting thread to move to the runnable state.

- If a thread calls `Condition` method `signalAll()`, then all of the threads waiting for that condition move to the runnable state and become eligible to reacquire the `Lock`.

- When a thread is finished with a shared object, it must invoke method `unlock()` to release the `Lock`.

Thread States With Synchronization

Queue of threads waiting for lock

Already locked by
another thread

notify() by
another thread

Queue of threads waiting for notify()

Thread attempting
access

Unlock by
another thread
– one in queue
moves to
running state

Lock obtained
by this thread

wait() by this thread

Running State

unlock by this thread
does not remove it from
the running state
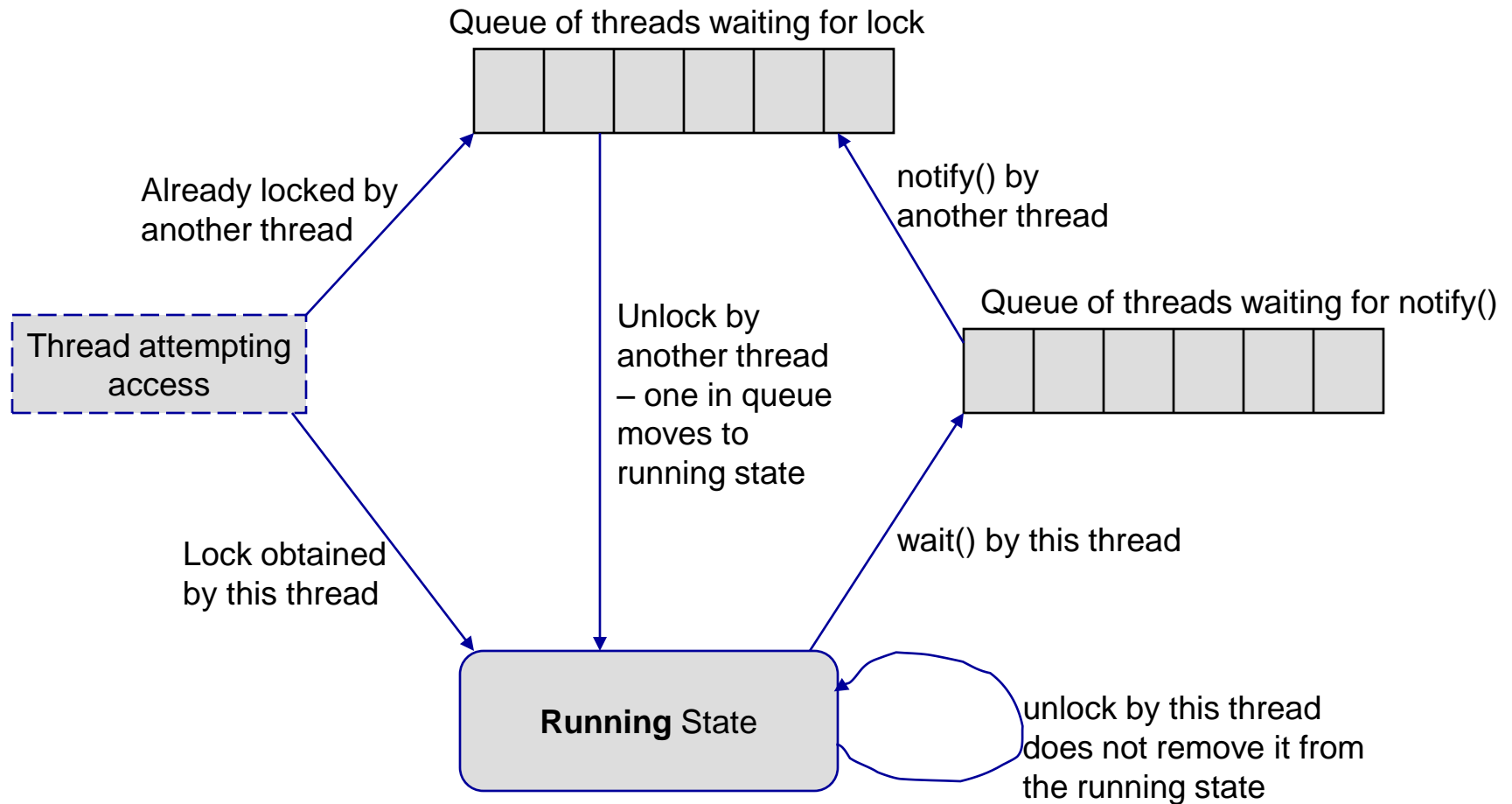
Deadlock

- Deadlock will occur when a waiting thread (call it thread 1) cannot proceed because it is waiting (either directly or indirectly) for another thread (call it thread 2) to proceed., while simultaneously thread 2 cannot proceed because it is waiting (either directly or indirectly) for thread 1 to proceed.

- When multiple threads manipulate a shared object using locks, ensure that if one thread invokes `await` to enter the wait state for a condition variable, a separate thread eventually will invoke method `signal` to transition the waiting thread on the condition variable back to the runnable state.

  - If multiple threads may be waiting on the condition variable, a separate thread can invoke method `signalAll` as a safeguard to ensure that all of the waiting threads have another opportunity to perform their tasks.

Producer/Consumer Problem
Threads Without Synchronization

- In a producer/consumer relationship, the producer portion of an application generates data and stores it in a shared object, and the consumer portion of an application reads data from the shared object.

  – Common examples are print spooling, copying data onto CDs, etc.

- In a multithreaded producer/consumer relationship, a producer thread generates data and places it in a shared object called a buffer. A consumer thread reads data from the buffer.

- What we want to consider first is how logic errors can arise if we do not synchronize access among multiple threads manipulating shared data.

Producer/Consumer w/o Synchronization

- The following example sets up a producer and consumer thread utilizing a shared buffer (code is on the webpage). The producer thread generates the integer numbers from 1 to 10, placing the values in the shared buffer. The consumer process reads the values in the buffer and prints the sum of all values consumed.

- Each value the producer thread writes into the buffer should be consumed exactly once by the consumer thread. However, the threads in this example are not synchronized.

  - This means that data can be lost if the producer writes new data into the buffer before the consumer has consumed the previous value.

  - Similarly, data can be incorrectly duplicated if the consumer thread consumes data again before the producer thread has produced the next value.

Producer/Consumer w/o Synchronization
(cont.)

- Since the producer thread will produce the values from 1 to 10, the correct sum that should be 55.

- The consumer process will arrive at this value only if each item produced by the producer thread is consumed exactly once by the consumer thread. No values are missed and none are consumed twice.

- I've set it up so that each thread writes to the screen what is being produced and what is being consumed.

- Note: the producer/consumer threads are put to sleep for a random interval between 0 and 3 seconds to emphasize the fact that in multithreaded applications, it is unpredictable when each thread will perform its task and for how long it will perform the task when it has a processor.

```java
// Producer's run method stores the values 1 to 10 in buffer.
import java.util.Random;
public class Producer implements Runnable{
    private static Random generator = new Random();
    private Buffer sharedLocation; // reference to shared object

    // constructor
    public Producer( Buffer shared )  {
        sharedLocation = shared;
    } // end Producer constructor

    // store values from 1 to 10 in sharedLocation
    public void run() {
        int sum = 0;
        for ( int count = 1; count <= 10; count++ ) {
            try  { // sleep 0 to 3 seconds, then place value in Buffer
                Thread.sleep( generator.nextInt( 3000 ) ); // sleep thread
                sharedLocation.set( count ); // set value in buffer
                sum += count; // increment sum of values
                System.out.printf( "\t%2d\n", sum );
            } // end try
            // if sleeping thread interrupted, print stack trace
            catch ( InterruptedException exception ) {
                exception.printStackTrace();
            } // end catch
        } // end for

        System.out.printf( "\n%s\n%s\n", "Producer done producing.",
            "Terminating Producer." );
    } // end method run
} // end class Producer
```

Producer Thread Class

Randomly sleep the thread for up to 3 seconds

```java
// Consumer's run method loops ten times reading a value from buffer.
import java.util.Random;
public class Consumer implements Runnable {
    private static Random generator = new Random();
    private Buffer sharedLocation; // reference to shared object
    // constructor
    public Consumer( Buffer shared ) {
        sharedLocation = shared;
    } // end Consumer constructor
    // read sharedLocation's value four times and sum the values
    public void run() {
        int sum = 0;
        for ( int count = 1; count <= 10; count++ ) {
            // sleep 0 to 3 seconds, read value from buffer and add to sum
            try {
                Thread.sleep( generator.nextInt( 3000 ) );
                sum += sharedLocation.get();
                System.out.printf( "\t\t\t%2d\n", sum );
            } // end try
            // if sleeping thread interrupted, print stack trace
            catch ( InterruptedException exception ) {
                exception.printStackTrace();
            } // end catch
        } // end for
        System.out.printf( "\n%s %d.\n%s\n",
            "Consumer read values totaling", sum, "Terminating Consumer." );
    } // end method run
} // end class Consumer
```

Consumer Thread Class

Randomly sleep the thread for up to 3 seconds

```
// Buffer interface specifies methods called by Producer and Consumer.
public interface Buffer {
  public void set( int value ); // place int value into Buffer (WRITE)
  public int get(); // return int value from Buffer (READ)
} // end interface Buffer
```

Buffer Interface

```
// UnsynchronizedBuffer represents a single shared integer.
public class UnsynchronizedBuffer implements Buffer {
  private int buffer = -1; // shared by producer and consumer threads
  // place value into buffer
  public void set( int value )   {
    System.out.printf( "Producer writes\t%2d", value );
    buffer = value;
  } // end method set

  // return value from buffer
  public int get()   {
    System.out.printf( "Consumer reads\t%2d", buffer );
    return buffer;
  } // end method get
} // end class UnsynchronizedBuffer
```

Unsynchronized Buffer
Class

```java
// Application shows two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest {
    public static void main( String[] args ){
        // create new thread pool with two threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );

        // create UnsynchronizedBuffer to store ints
        Buffer sharedLocation = new UnsynchronizedBuffer();
        System.out.println( "     \t\t     \tSum     \tSum");
        System.out.println( "Action\t\tValue\tProduced\tConsumed" );
        System.out.println( "------\t\t-----\t--------\t--------\n" );

        // try to start producer and consumer giving each of them access to SharedLocation
         try  {
            application.execute( new Producer( sharedLocation ) );
            application.execute( new Consumer( sharedLocation ) );
        } // end try
        catch ( Exception exception ) {
            exception.printStackTrace();
        } // end catch

        application.shutdown(); // terminate application when threads end
    } // end main
} // end class SharedBufferTest
```

Producer/Consumer
Driver Class

Unsynchronized Case

Producer Side   Consumer Side

sharedLocation.set(count) ────────→ sharedLocation (Buffer) ────────→ sum += sharedLocation.get()

set method returns   get method returns
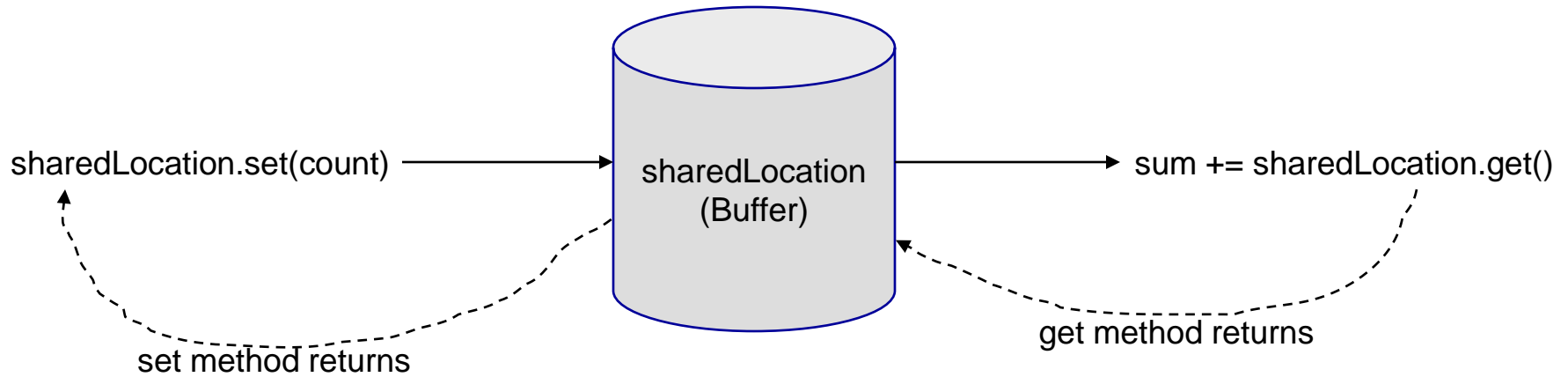
| running | Both the producer and consumer threads are always in the running state – never blocked. | running |

```
<terminated> SharedBufferTest [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe

                             Sum                Sum
Action            Value      Produced           Consumed
------            -----      --------           --------


Producer writes  1          1
Producer writes  2          3
Consumer reads   2                              2
Consumer reads   2                              4
Producer writes  3          6
Consumer reads   3                              7
Producer writes  4          10
Producer writes  5          15
Producer writes  6          21
Producer writes  7          28
Producer writes  8          36
Producer writes  9          45
Consumer reads   9                              16
Producer writes 10          55


Producer done producing.
Terminating Producer.    Producer produced values totaling: 55


Consumer reads   10                             26
Consumer reads   10                             36
Consumer reads   10                             46
Consumer reads   10                             56
Consumer reads   10                             66
Consumer reads   10                             76


Consumer read values totaling 76.   Terminating Consumer.
```

The unsynchronizd threads did not produce the same sum. The producer produced values that sum to 55, but the consumer consumed values that sum to 76! Notice that the consumer read the value 10 six times and failed to read the values of several values at all (e.g. 1, 4,5,6,7 and 8).
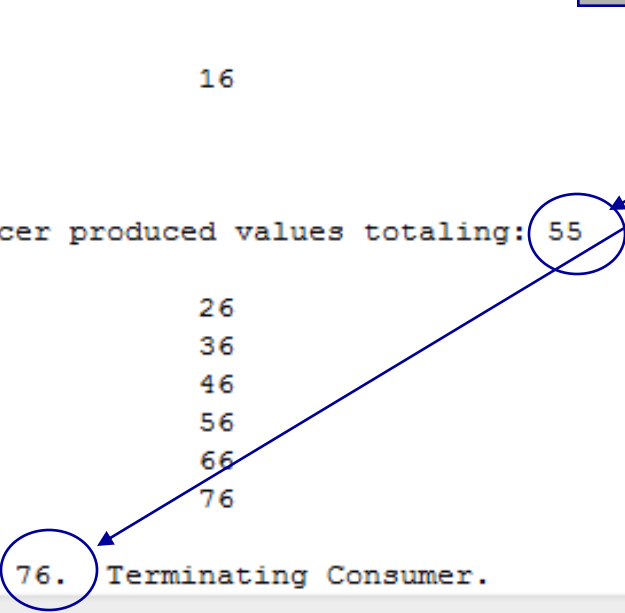
```
<terminated> SharedBufferTest [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Action          Value    Produced        Consumed
------          -----    --------        --------


Producer writes  1        1
Consumer reads   1                        1
Consumer reads   1                        2
Producer writes  2        3
Consumer reads   1                        4
Producer writes  3        6
Consumer reads   3                        7
Producer writes  4       10
Producer writes  5       15
Consumer reads   5                       12
Consumer reads   5                       17
Producer writes  6       21
Producer writes  7       28
Consumer reads   7                       24
Producer writes  8       36
Consumer reads   8                       32
Producer writes  9       45
Producer writes 10       55


Producer done producing.
Terminating Producer.    Producer produced values totaling: 55


Consumer reads  10                       42
Consumer reads  10                       52


Consumer read values totaling 52.   Terminating Consumer.
```

In this execution, the sum produced by the consumer is closer but still inaccurate because the consumer read the values of 1, 5 and 10 two times and failed to read the values of 2, 4, 6, and 9 at all.
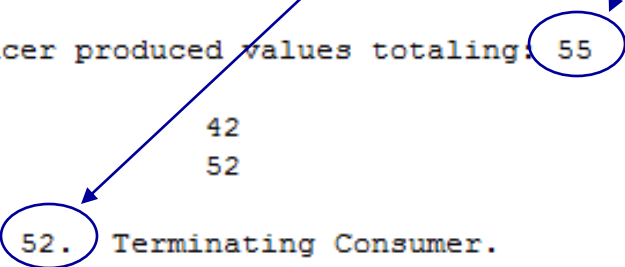
```java
// SynchronizedBuffer synchronizes access to a single shared integer.
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
public class SynchronizedBuffer implements Buffer
{
    // Lock to control synchronization with this buffer
    private Lock accessLock = new ReentrantLock();
    // condition variables to control reading and writing
    private Condition canWrite = accessLock.newCondition();
    private Condition canRead = accessLock.newCondition();

    private int buffer = -1; // shared by producer and consumer threads
    private boolean occupied = false; // whether buffer is occupied
    // place int value into buffer
    public void set( int value )
    {
        accessLock.lock(); // lock this object
        // output thread information and buffer information, then wait
        try
        {
            // while buffer is not empty, place thread in waiting state
            while ( occupied )
            {
                System.out.println( "Producer tries to write." );
                displayState( "Buffer full. Producer waits." );
                canWrite.await(); // wait until buffer is empty
            } // end while
```

Synchronized Buffer Class

No fairness policy needed since only a single producer thread and single consumer thread

Condition variables on the lock. Condition canWrite contains a queue for threads waiting to write while the buffer is full. If the buffer is full the Producer calls method await on this condition. When the Consumer reads data from a full buffer, it calls method signal on this Condition. Condition canRead contains a queue for threads waiting while the buffer is empty. If the buffer is empty the Consumer calls method await on this Condition. When the Producer writes to the empty buffer, it will call method signal on this Condition.

Acquire lock

```
    buffer = value; // set new buffer value
        // indicate producer cannot store another value
        // until consumer retrieves current buffer value
        occupied = true;
        displayState( "Producer writes " + buffer );
        // signal thread waiting to read from buffer
        canRead.signal();
      } // end try
      catch ( InterruptedException exception )   {
        exception.printStackTrace();
      } // end catch
      finally   {
        accessLock.unlock(); // unlock this object
      } // end finally
   } // end method set


    // return value from buffer
    public int get()   {
      int readValue = 0; // initialize value read from buffer
      accessLock.lock(); // lock this object
      // output thread information and buffer information, then wait
      try     {
        // while no data to read, place thread in waiting state
        while ( !occupied )   {
          System.out.println( "Consumer tries to read." );
          displayState( "Buffer empty. Consumer waits." );
          canRead.await(); // wait until buffer is full
        } // end while
```

Signal Consumer thread that a value has been produced and can be read.

Unlock object before exiting method

Acquire lock on the buffer

Consumer must wait until a value has been produced by the Producer. Await signal by Producer

```
    // indicate that producer can store another value
        // because consumer just retrieved buffer value
        occupied = false;
        readValue = buffer; // retrieve value from buffer
        displayState( "Consumer reads " + readValue );
        // signal thread waiting for buffer to be empty
        canWrite.signal();
    } // end try
    // if waiting thread interrupted, print stack trace
    catch ( InterruptedException exception )       {
        exception.printStackTrace();
    } // end catch
    finally    {
        accessLock.unlock(); // unlock this object
    } // end finally

    return readValue;
} // end method get


// display current operation and buffer state
public void displayState( String operation )
{
    System.out.printf( "%-40s%d\t\t\t\t%b\n", operation, buffer,
        occupied );
} // end method displayState
} // end class SynchronizedBuffer
```

Signal waiting Producer that the buffer is empty and it can write

Make sure lock is released

```java
// Application shows two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class SharedBufferTest2
{
  public static void main( String[] args )
  {
    // create new thread pool with two threads
    ExecutorService application = Executors.newFixedThreadPool( 2 );

    // create SynchronizedBuffer to store ints
    Buffer sharedLocation = new SynchronizedBuffer();
    System.out.println("Using Standard Locking");
    System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
       "Buffer Contents", "Occupied", "---------", "--------------\t\t--------" );

    try   {  // try to start producer and consumer
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );
    } // end try
    catch ( Exception exception )
    {
      exception.printStackTrace();
    } // end catch

    application.shutdown();
  } // end main
} // end class SharedBufferTest2
```

Driver Class For Illustrating Synchronization In Producer/Consumer Problem

Only change between SharedBufferTest for unsynchronized version

Synchronized Case

lock acquisition queue

no

accessLock.lock(); //acquire lock

no

accessLock.lock(); //acquire lock

released

released

while (occupied) //buffer not empty

canWrite condition queue

while (!occupied) //buffer empty

canWrite.await(); block on condition

wait

canRead.await(); block

wait

signal from consumer

signal from producer

canRead condition queue

buffer = value; //perform write

occupied = false; //indicate read

occupied = true; //indicate write

write

readValue= buffer;

read

canRead.signal(); //signal thread

canWrite.signal(); //signal thread

accessLock.unlock(); //release lock

sharedLocation
(Buffer)

accessLock.unlock(); //release lock

set method returns

get method returns

State Diagram – Synchronized Version

Running

buffer full (occupied)

canWrite condition queue

canWrite signaled

lock request fails

accessLock queue

lock released

lock request fails

buffer empty

canRead condition queue

Running

canRead signaled

Consumer Thread

Blocked

<terminated> SharedBufferTest2 [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 11:25:24 AM)
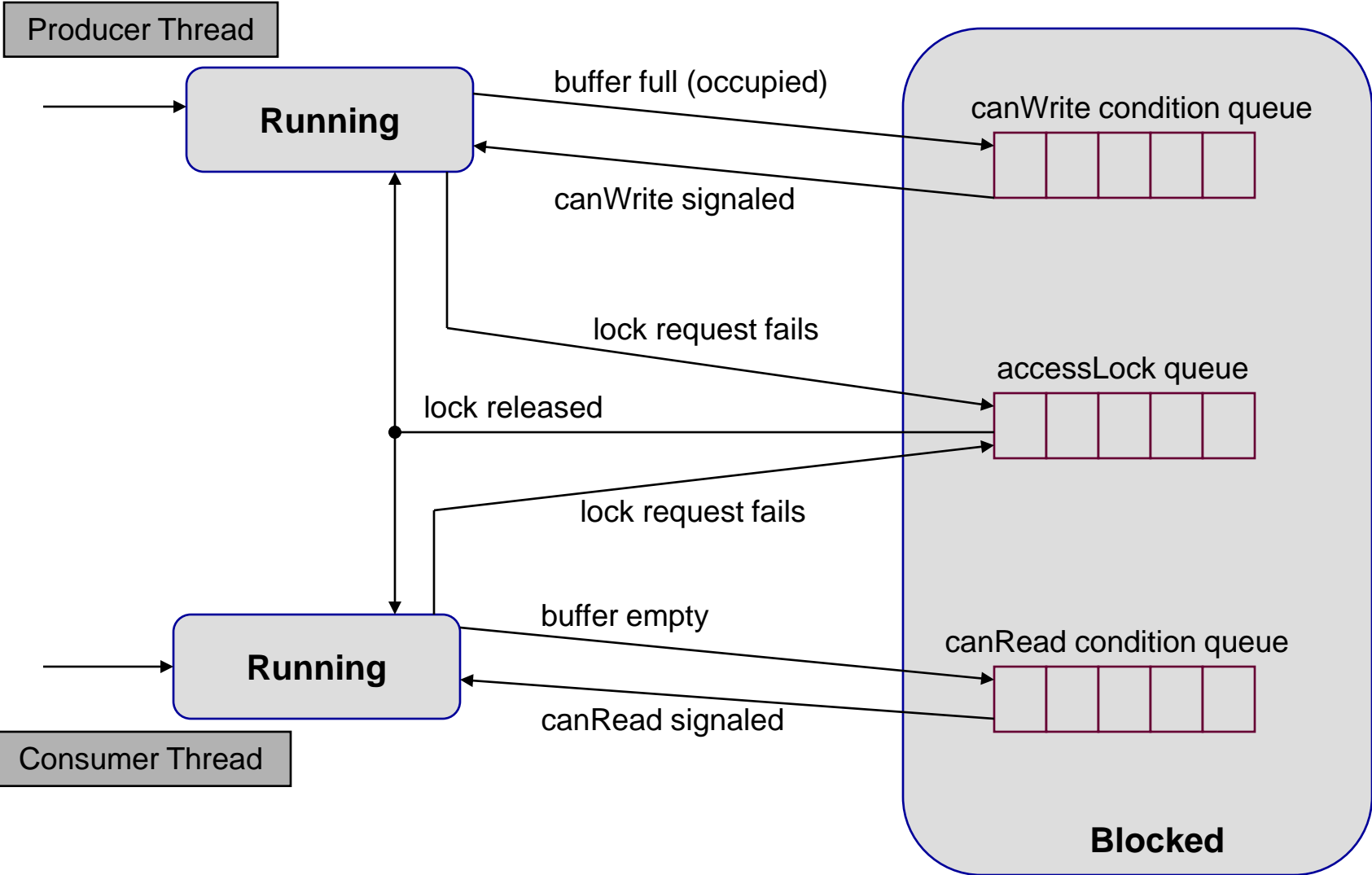
```
Using Standard Locking
Operation                              Buffer Contents          Occupied
---------                              ---------------          --------


Producer writes 1                      1                          true
Consumer reads 1                       1                          false
Consumer tries to read.
Buffer empty. Consumer waits.          1                          false
Producer writes 2                      2                          true
Consumer reads 2                       2                          false
Consumer tries to read.
Buffer empty. Consumer waits.          2                          false
Producer writes 3                      3                          true
Consumer reads 3                       3                          false
Producer writes 4                      4                          true
Consumer reads 4                       4                          false
Producer writes 5                      5                          true
Consumer reads 5                       5                          false
Producer writes 6                      6                          true
Producer tries to write.
Buffer full. Producer waits.           6                          true
Consumer reads 6                       6                          false
Producer writes 7                      7                          true
Producer tries to write.
Buffer full. Producer waits.           7                          true
Consumer reads 7                       7                          false
Producer writes 8                      8                          true
Producer tries to write.
Buffer full. Producer waits.           8                          true
Consumer reads 8                       8                          false
```

```
Producer writes 9                        9                              true
Consumer reads 9                         9                              false
Consumer tries to read.
Buffer empty. Consumer waits.            9                              false
Producer writes 10                       10                             true
Consumer reads 10                        10                             false

Producer done producing.
  Terminating Producer.

Consumer read values totaling 55.
  Terminating Consumer.
Producer produced values totaling: 55
```

Both the Producer and Consumer threads produced the same sum – synchronized threads

Monitors and Monitor Locks

- Another way to perform synchronization is to use Java's built-in monitors. Every object has a monitor. Strictly speaking, the monitor is not allocated unless it is used.

- A monitor allows one thread at a time to execute inside a synchronized statement on the object. This is accomplished by acquiring a lock on the object when the program enters the synchronized statement.

```
synchronized (object)
{
            statements
} //end synchronized statement
```

  - Where *object* is the object whose monitor lock will be acquired.

- If there are several synchronized statements attempting to execute on an object at the same time, only one of them may be active on the object at once – all the other threads attempting to enter a synchronized statement on the same object are placed into the blocked state.

Mutual Exclusion Over a Block of Statements

- When a synchronized statement finishes executing, the monitor lock on the object is released and the highest priority blocked thread attempting to enter a synchronized statement proceeds.

- Applying mutual exclusion to a block of statements rather than to an entire class or an entire method is handled in much the same manner, by attaching the keyword synchronized before a block of code.

- You must explicitly mention in parentheses the object whose lock must be acquired before the block can be entered.

- The next page illustrates the steam boiler pressure gauge problem using a synchronized statement block to control the threads access to the pressure gauge.

```java
//thread class to raise the pressure in the Boiler
class syncPressure extends Thread {
    static Object O = new Object();
    void RaisePressure() {
        synchronized(O) {
            if (syncSteamBoiler.pressureGauge < syncSteamBoiler.safetyLimit-15) {
                //wait briefly to simulate some calculations
                try {sleep(200); } catch (Exception e) { }
                syncSteamBoiler.pressureGauge+= 15; //raise the pressure 15 psi
                System.out.println("Thread " + this.getName() + " finds pressure within limits - increases
            }
            else
                System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
        }//end synchronized object
    }
    public void run() {
        RaisePressure();  //this thread is to raise the pressure
    }
}
```

Synchronized statement requires an Object to lock.

Synchronized block

```java
//thread class to raise the pressure in the Boiler
class syncPressure extends Thread {
    static Object O = new Object();
    void RaisePressure() {
        synchronized(O) {
        if (syncSteamBoiler.pressureGauge < syncSteamBoiler.safetyLimit-15) {
            //wait briefly to simulate some calculations
            try {sleep(200); } catch (Exception e) { }
            syncSteamBoiler.pressureGauge+= 15; //raise the pressure 15 psi
            System.out.println("Thread " + this.getName() + " finds pressure within limits - increases
         }
         else
            System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
        }//end synchronized object
```

Console ✕   ■ ✖ ✖ | ▤ ▤ ▣ ▣ | ▱ ▱ ▾ ▱ ▾

Show Console When Standard

<terminated> syncSteamBoiler [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 12:12:30 PM)

```
Thread Thread-0 finds pressure within limits - increases pressure
Thread Thread-9 finds pressure within limits - increases pressure
Thread Thread-8 finds pressure within limits - increases pressure
ThreadThread-7 finds pressure too high - do nothing
ThreadThread-6 finds pressure too high - do nothing
ThreadThread-5 finds pressure too high - do nothing
ThreadThread-4 finds pressure too high - do nothing
ThreadThread-3 finds pressure too high - do nothing
ThreadThread-2 finds pressure too high - do nothing
ThreadThread-1 finds pressure too high - do nothing

Gauge reads 45, the safe limit is 50...System OK!
```

Monitors and Monitor Locks (cont.)

- Java also allows synchronized methods. A synchronized method is equivalent to a synchronized statement enclosing the entire body of a method.

- If a thread obtains the monitor lock on an object and then discovers that it cannot continue with its task until some condition is satisfied, the thread can invoke Object method `wait`, releasing the monitor lock on the object. This will place the thread in the wait state.

- When a thread executing a synchronized statement completes or satisfies the condition on which another thread may be waiting, it can invoke Object method `notify` to allow a waiting thread to transition to the blocked state again.

Caution When Using Synchronization

- As with any multi-threaded application, care must be taken when using synchronization to achieve the desired effect and not introduce some serious defect in the application.

- Consider the variation of the pressure gauge example that we've been dealing with on the following page. Study the code carefully and try to determine if it will achieve the same effect as the previous version of the code.

- Is it correct? Why or why not?

Does this code correctly synchronize the pressure gauge reading threads?

```java
//thread class to raise the pressure in the Boiler
class syncPressure extends Thread {
    //static Object O = new Object();
    synchronized void RaisePressure() {
        if (syncSteamBoiler.pressureGauge < syncSteamBoiler.safetyLimit-15) {
            //wait briefly to simulate some calculations
            try {sleep(100); } catch (Exception e) { }
            syncSteamBoiler.pressureGauge+= 15; //raise the pressure 15 psi
            System.out.println("Thread " + this.getName() + " finds pressure within limits - increases
        }
        else
            System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
    }
    public void run() {
        RaisePressure();  //this thread is to raise the pressure
    }
}
```

No! The "this" object is one of the 10 different threads that are created. Each thread will successfully grab its own lock, and there will be no exclusion between the different threads.

Synchronization excludes threads working on the *same object*; it does not synchronize the *same method* on different objects!

```java
//thread class to raise the pressure in the Boiler
class syncPressure extends Thread {
    //static Object O = new Object();
    synchronized void RaisePressure() {
        if (syncSteamBoiler.pressureGauge < syncSteamBoiler.safetyLimit-15) {
            //wait briefly to simulate some calculations
            try {sleep(100); } catch (Exception e) { }
            syncSteamBoiler.pressureGauge+= 15; //raise the pressure 15 psi
            System.out.println("Thread " + this.getName() + " finds pressure within limits - increases
        }
        else
            System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
    }
    public void run() {
```

Console ✕

<terminated> syncSteamBoiler [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 11:43:47 AM)

```
Thread Thread-1 finds pressure within limits - increases pressure
Thread Thread-4 finds pressure within limits - increases pressure
Thread Thread-3 finds pressure within limits - increases pressure
Thread Thread-0 finds pressure within limits - increases pressure
Thread Thread-9 finds pressure within limits - increases pressure
Thread Thread-5 finds pressure within limits - increases pressure
Thread Thread-2 finds pressure within limits - increases pressure
Thread Thread-6 finds pressure within limits - increases pressure
Thread Thread-8 finds pressure within limits - increases pressure
Thread Thread-7 finds pressure within limits - increases pressure

Gauge reads 135, the safe limit is 50...B O O M ! ! !
```

NO!