9 Threads

Topics

- Thread Definition
- Thread Basics
  - Thread States
  - Priorities
- The *Thread* Class
  - Constructor
  - Constants
  - Methods

Topics

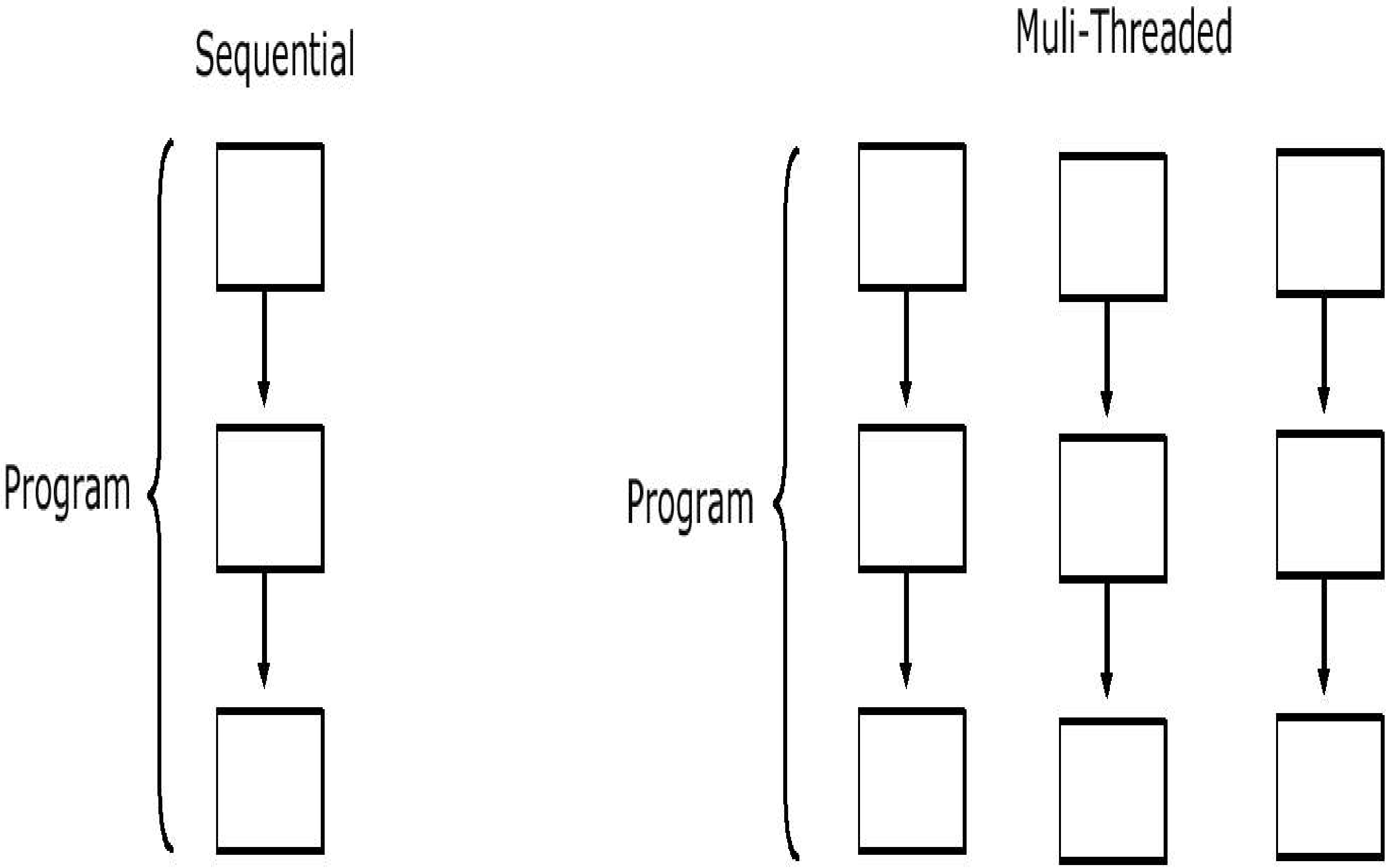
- Creating Threads
  - Extending the *Thread* Class
  - Implementing the Runnable Interface
  - Extending vs. Implementing
- Synchronization
  - Locking an Object
- Interthread Communication
- Concurrency Utilities

Threads

- Why threads?
  - Need to handle concurrent processes

- Definition
  - Single sequential flow of control within a program
  - For simplicity, think of threads as processes executed by a program
  - Example:
    - Operating System
    - HotJava web browser

Threads



Thread States

- A thread can in one of several possible states:
  1. Running
     - Currently running
     - In control of CPU
  2. Ready to run
     - Can run but not yet given the chance
  3. Resumed
     - Ready to run after being suspended or block
  4. Suspended
     - Voluntarily allowed other threads to run
  5. Blocked
     - Waiting for some resource or event to occur

Thread Priorities

- Why priorities?
  - Determine which thread receives CPU control and gets to be executed first

- Definition:
  - Integer value ranging from 1 to 10
  - Higher the thread priority → larger chance of being executed first
  - Example:
    - Two threads are ready to run
    - First thread: priority of 5, already running
    - Second thread = priority of 10, comes in while first thread is running

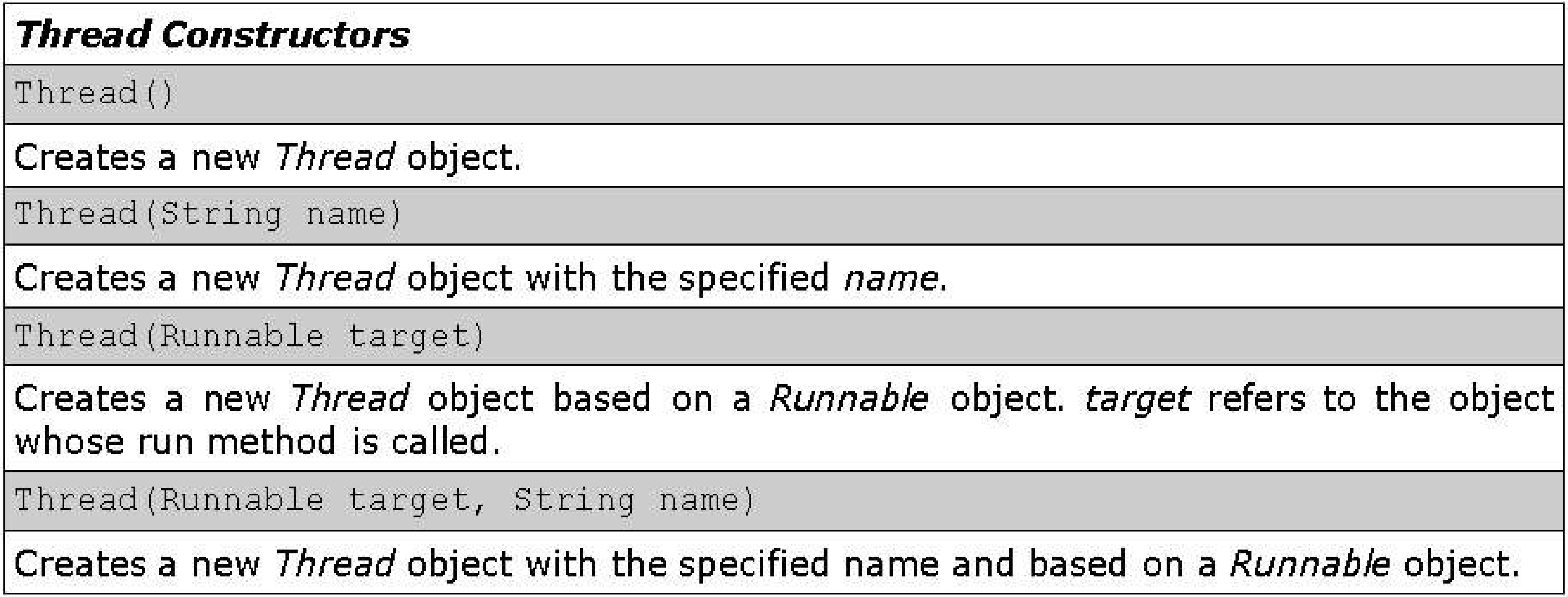
Thread Priorities

- Context switch
  - Occurs when a thread snatches the control of CPU from another
  - When does it occur?
    - Running thread voluntarily relinquishes CPU control
    - Running thread is preempted by a higher priority thread

- More than one highest priority thread that is ready to run
  - Deciding which receives CPU control depends on the operating system
  - Windows 95/98/NT: Uses time-sliced round-robin
  - Solaris: Executing thread should voluntarily relinquish CPU control

The *Thread* Class: Constructor

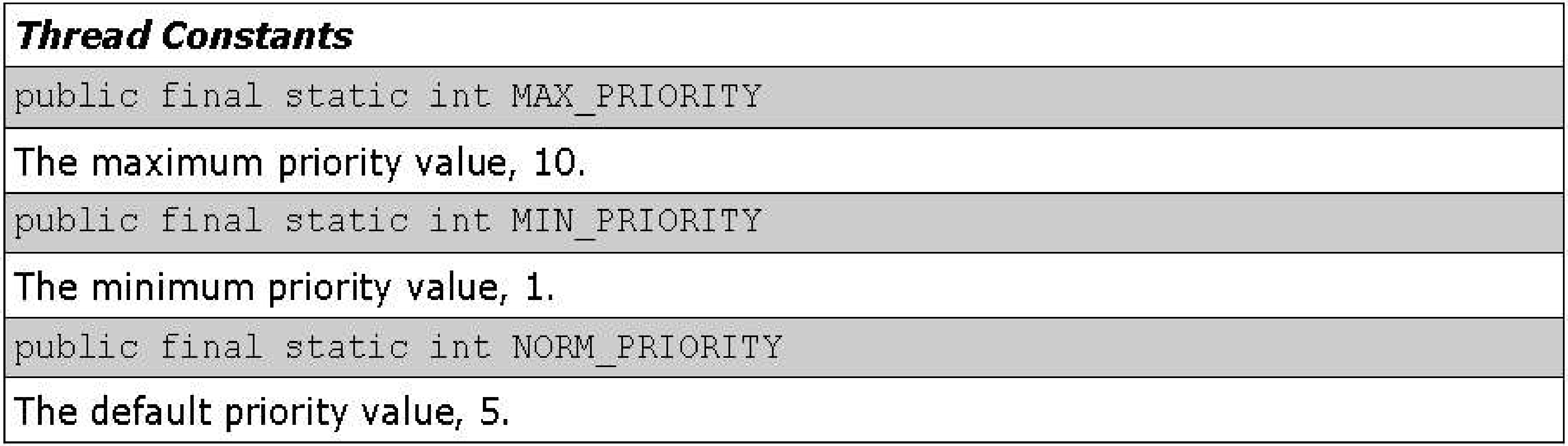
- Has eight constructors



Thread Constructors
`Thread()`
Creates a new *Thread* object.
`Thread(String name)`
Creates a new *Thread* object with the specified *name*.
`Thread(Runnable target)`
Creates a new *Thread* object based on a *Runnable* object. *target* refers to the object whose run method is called.
`Thread(Runnable target, String name)`
Creates a new *Thread* object with the specified name and based on a *Runnable* object.

The *Thread* Class: Constants

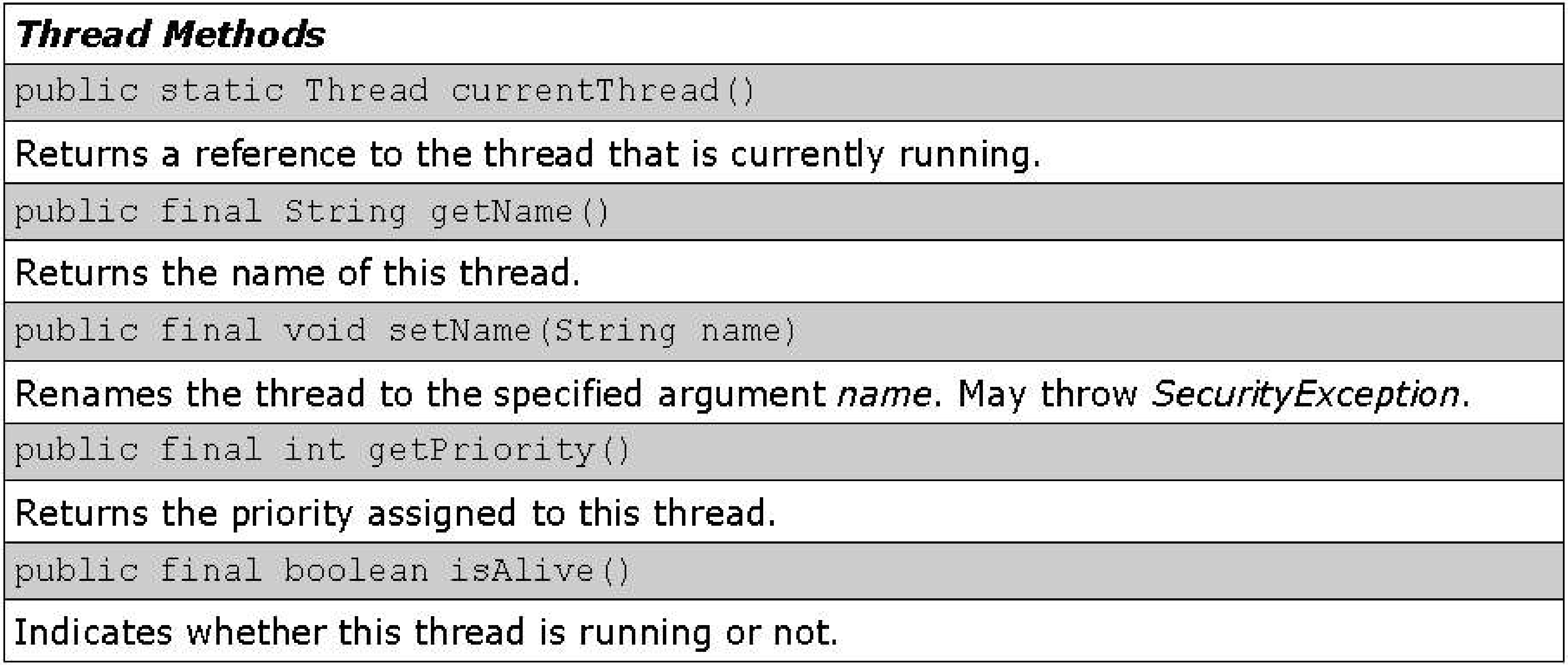
- Contains fields for priority values



Thread Constants
`public final static int MAX_PRIORITY`
The maximum priority value, 10.
`public final static int MIN_PRIORITY`
The minimum priority value, 1.
`public final static int NORM_PRIORITY`
The default priority value, 5.

The *Thread* Class: Methods

- Some *Thread* methods



Thread Methods
`public static Thread currentThread()`
Returns a reference to the thread that is currently running.
`public final String getName()`
Returns the name of this thread.
`public final void setName(String name)`
Renames the thread to the specified argument *name*. May throw *SecurityException*.
`public final int getPriority()`
Returns the priority assigned to this thread.
`public final boolean isAlive()`
Indicates whether this thread is running or not.

A Thread Example

```
import javax.swing.*;
import java.awt.*;
class CountDownGUI extends JFrame {
   JLabel label;
   CountDownGUI(String title) {
      super(title);
      label = new JLabel("Start count!");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      getContentPane().add(new Panel(),orderLayout.WEST);
      getContentPane().add(label);
      setSize(300,300);
      setVisible(true);
   }
//continued...
```

A Thread Example

```
   void startCount() {
      try {
         for (int i = 10; i > 0; i--) {
            Thread.sleep(1000);
            label.setText(i + "");
         }
         Thread.sleep(1000);
         label.setText("Count down complete.");
         Thread.sleep(1000);
      } catch (InterruptedException ie) {
      }
      label.setText(Thread.currentThread().toString());
   }
//continued...
```

A Thread Example

```
   public static void main(String args[]) {
      CountDownGUI cdg =
                  new CountDownGUI("Count down GUI");
      cdg.startCount();
   }
}
```

Creating Threads

- Two ways of creating threads:
  - Extending the *Thread* class
  - Implementing the *Runnable* interface

Extending the *Thread* Class

```java
class PrintNameThread extends Thread {
   PrintNameThread(String name) {
      super(name);
      start(); //runs the thread once instantiated
   }
   public void run() {
      String name = getName();
      for (int i = 0; i < 100; i++) {
         System.out.print(name);
      }
   }
}
//continued
```

Extending the *Thread* Class

```
class TestThread {
   public static void main(String args[]) {
      PrintNameThread pnt1 =
                      new PrintNameThread("A");
      PrintNameThread pnt2 =
                      new PrintNameThread("B");
      PrintNameThread pnt3 =
                      new PrintNameThread("C");
      PrintNameThread pnt4 =
                      new PrintNameThread("D");
   }
}
```

Extending the *Thread* Class

- Can modify *main* method as follows:

```
class TestThread {
   public static void main(String args[]) {
      new PrintNameThread("A");
      new PrintNameThread("B");
      new PrintNameThread("C");
      new PrintNameThread("D");
   }
}
```

Extending the *Thread* Class

- Sample output:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABC
DABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDAB
CDABCDABCDABCDABCDABCDABCDABCDABCDABCDBCDBCDBCDBCDBCD
BCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBC
DBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDB
CDBCDBCDBCDBCDBCDBCDBCDBCDBCD
```

Implementing the *Runnable* Interface

- Only need to implement the *run* method
  - Think of *run* as the *main* method of created threads

- Example:

```
class TestThread {
   public static void main(String args[]) {
      new PrintNameThread("A");
      new PrintNameThread("B");
      new PrintNameThread("C");
      new PrintNameThread("D");
   }
}
//continued...
```

Implementing the *Runnable* Interface

```
class PrintNameThread implements Runnable {
    Thread thread;
    PrintNameThread(String name) {
        thread = new Thread(this, name);
        thread.start();
    }
    public void run() {
        String name = thread.getName();
        for (int i = 0; i < 100; i++) {
            System.out.print(name);
        }
    }
}
```

Extending vs. Implementing

- Choosing between these two is a matter of taste

- Implementing the *Runnable* interface
  - May take more work since we still
    - Declare a *Thread* object
    - Call the *Thread* methods on this object
  - Your class can still extend other class

- Extending the *Thread* class
  - Easier to implement
  - Your class can no longer extend any other class

Example: The *join* Method

- Causes the currently running thread to wait until the thread that calls this method finishes execution

- Example:

```
class PrintNameThread implements Runnable {
   Thread thread;
   PrintNameThread(String name) {
      thread = new Thread(this, name);
      thread.start();
   }
//continued
```

Example: The *join* Method

```
    public void run() {
        String name = thread.getName();
        for (int i = 0; i < 100; i++) {
            System.out.print(name);
        }
    }
}
class TestThread {
    public static void main(String args[]) {
        PrintNameThread pnt1 = new PrintNameThread("A");
        PrintNameThread pnt2 = new PrintNameThread("B");
        PrintNameThread pnt3 = new PrintNameThread("C");
        PrintNameThread pnt4 = new PrintNameThread("D");
//continued...
```

Example: The *join* Method

```
      System.out.println("Running threads...");
      try {
         pnt1.thread.join();
         pnt2.thread.join();
         pnt3.thread.join();
         pnt4.thread.join();
      } catch (InterruptedException ie) {
      }
      System.out.println("Threads killed.");
   }
}
//try removing the entire catch block
```

Synchronization

- Why synchronize threads?
  - Concurrently running threads may require outside resources or methods
  - Need to communicate with other concurrently running threads to know their status and activities
  - Example: Producer-Consumer problem

An Unsynchronized Example

```
class TwoStrings {
   static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}
//continued...
```

An Unsynchronized Example

```
class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   PrintStringsThread(String str1, String str2) {
      this.str1 = str1;
      this.str2 = str2;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      TwoStrings.print(str1, str2);
   }
}
//continued...
```

An Unsynchronized Example

```
class TestThread {
   public static void main(String args[]) {
      new PrintStringsThread("Hello ", "there.");
      new PrintStringsThread("How are ", "you?");
      new PrintStringsThread("Thank you ",
                                   "very much!");
   }
}
```

An Unsynchronized Example

- Sample output:

```
Hello How are Thank you there.
you?
very much!
```

Synchronization: Locking an Object

- Locking of an object:
  - Assures that only one thread gets to access a particular method
  - Java allows you to lock objects with the use of monitors
    - Object enters the implicit monitor when the object's synchronized method is invoked
    - Once an object is in the monitor, the monitor makes sure that no other thread accesses the same object

Synchronization: Locking an Object

- Synchronizing a method:
  - Use the *synchronized* keyword
    - Prefixed to the header of the method definition
    - Can synchronize the object of which the method is a member of

```
synchronized (<object>) {
   //statements to be synchronized
}
```

First Synchronized Example

```
class TwoStrings {
   synchronized static void print(String str1,
                                  String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}
//continued...
```

First Synchronized Example

```
class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   PrintStringsThread(String str1, String str2) {
      this.str1 = str1;
      this.str2 = str2;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      TwoStrings.print(str1, str2);
   }
}
//continued...
```

First Synchronized Example

```
class TestThread {
   public static void main(String args[]) {
      new PrintStringsThread("Hello ", "there.");
      new PrintStringsThread("How are ", "you?");
      new PrintStringsThread("Thank you ",
                                   "very much!");
   }
}
```

First Synchronized Example

- Sample output:

```
Hello there.
How are you?
Thank you very much!
```

Second Synchronized Example

```
class TwoStrings {
   static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}
//continued...
```

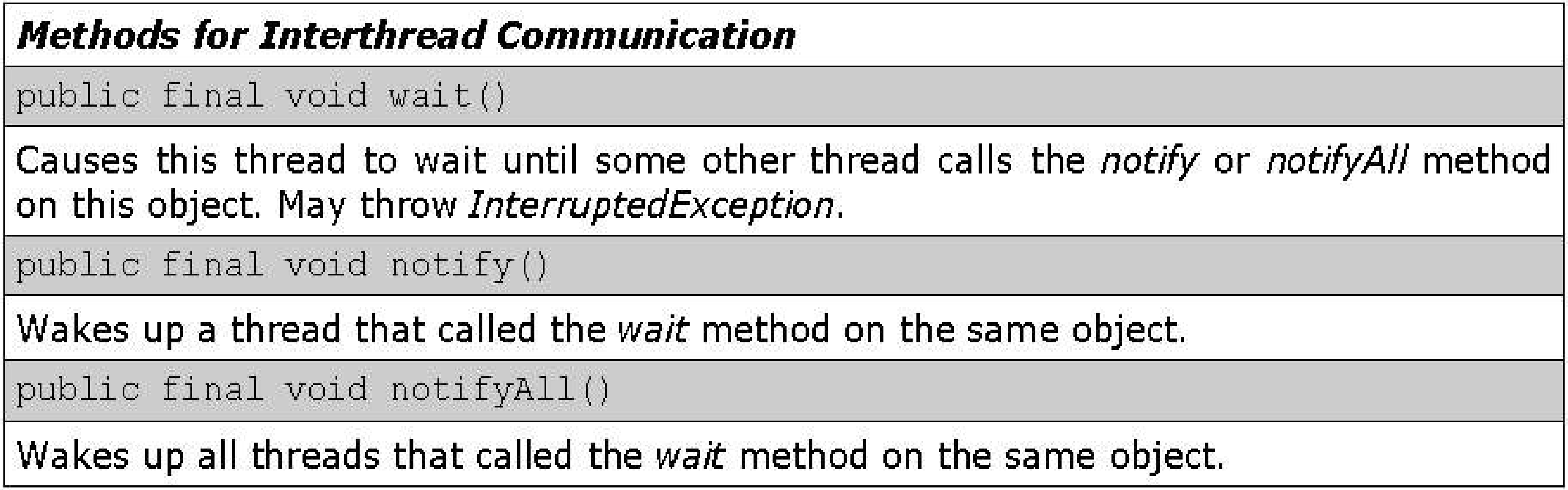
Second Synchronized Example

```
class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   TwoStrings ts;
   PrintStringsThread(String str1, String str2,
                      TwoStrings ts) {
      this.str1 = str1;
      this.str2 = str2;
      this.ts = ts;
      thread = new Thread(this);
      thread.start();
   }
//continued...
```

Second Synchronized Example

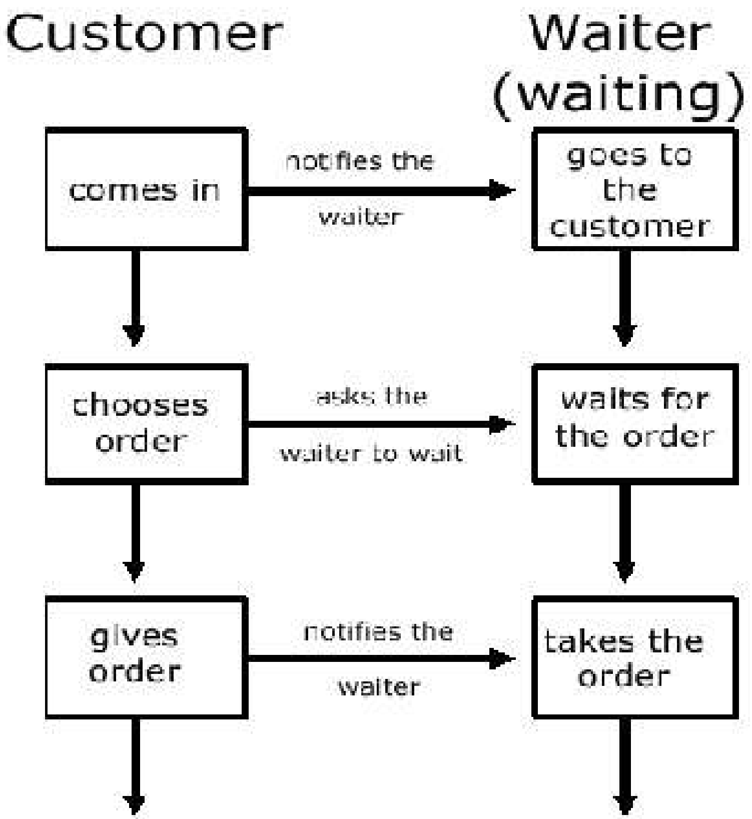
```
   public void run() {
      synchronized (ts) {
         ts.print(str1, str2);
      }
   }
}
class TestThread {
   public static void main(String args[]) {
      TwoStrings ts = new TwoStrings();
      new PrintStringsThread("Hello ", "there.", ts);
      new PrintStringsThread("How are ", "you?", ts);
      new PrintStringsThread("Thank you ",
                                   "very much!", ts);
}}
```

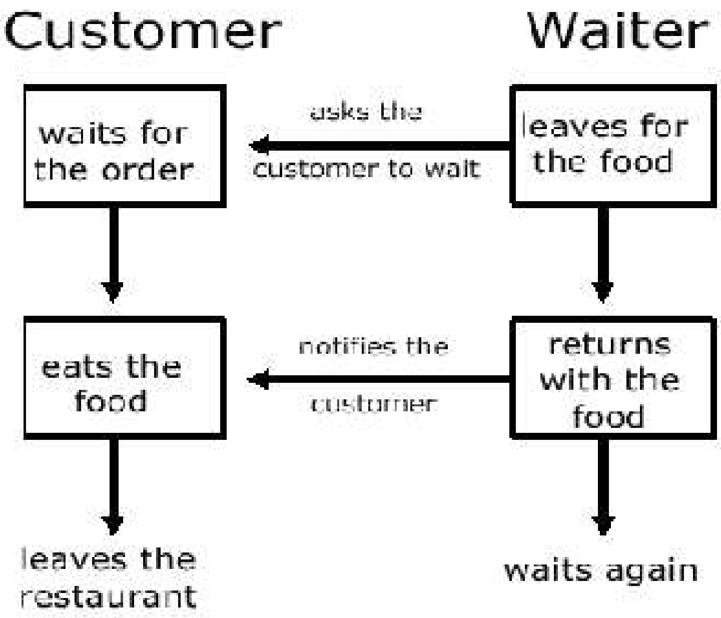
Interthread Communication: Methods



Methods for Interthread Communication
`public final void wait()`
Causes this thread to wait until some other thread calls the *notify* or *notifyAll* method on this object. May throw *InterruptedException*.
`public final void notify()`
Wakes up a thread that called the *wait* method on the same object.
`public final void notifyAll()`
Wakes up all threads that called the *wait* method on the same object.

Interthread Communication





Producer-Consumer Example

```
class SharedData {
    int data;
    synchronized void set(int value) {
        System.out.println("Generate " + value);
        data = value;
    }
    synchronized int get() {
        System.out.println("Get " + data);
        return data;
    }
}
//continued...
```

Producer-Consumer Example

```
class Producer implements Runnable {
    SharedData sd;
    Producer(SharedData sd) {
        this.sd = sd;
        new Thread(this, "Producer").start();
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            sd.set((int)(Math.random()*100));
        }
    }
}
//continued...
```

Producer-Consumer Example

```
class Consumer implements Runnable {
    SharedData sd;
    Consumer(SharedData sd) {
        this.sd = sd;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        for (int i = 0; i < 10 ; i++) {
            sd.get();
        }
    }
}
//continued...
```

Producer-Consumer Example

```
class TestProducerConsumer {
   public static void main(String args[])
                           throws Exception {
      SharedData sd = new SharedData();
      new Producer(sd);
      new Consumer(sd);
   }
}
```

Producer-Consumer Example

- Sample output:

```
Generate 8
Generate 45
Generate 52
Generate 65
Get 65
Generate 23
Get 23
Generate 49
Get 49
Generate 35
Get 35
Generate 39
Get 39
Generate 85
Get 85
Get 85
Get 85
Generate 35
Get 35
Get 35
```

Fixed Producer-Consumer Example

```
class SharedData {
   int data;
   boolean valueSet = false;
   synchronized void set(int value) {
      if (valueSet) {   //has just produced a value
         try {
            wait();
         } catch (InterruptedException ie) {
         }
      }
//continued...
```

Fixed Producer-Consumer Example

```
        System.out.println("Generate " + value);
        data = value;
        valueSet = true;
        notify();
    }
//continued...
```

Fixed Producer-Consumer Example

```
    synchronized int get() {
        if (!valueSet) {
            //producer hasn't set a value yet
            try {
                wait();
            } catch (InterruptedException ie) {
            }
        }
//continued...
```

Fixed Producer-Consumer Example

```
      System.out.println("Get " + data);
      valueSet = false;
      notify();
      return data;
   }
}

/* The remaining part of the code doesn't change. */
```

Producer-Consumer Example

- Sample output:

```
Generate 76
Get 76
Generate 25
Get 25
Generate 34
Get 34
Generate 84
Get 84
Generate 48
Get 48
Generate 29
Get 29
Generate 26
Get 26
Generate 86
Get 86
Generate 65
Get 65
Generate 38
Get 38
Generate 46
Get 46
```

Concurrency Utilities

- Introduced in Java 2 SE 5.0
- Found in the *java.util.concurrent* package
- Interfaces
  - *Executor*
  - *Callable*

The *Executor* Interface

- (BEFORE) Executing *Runnable* tasks:

```
new Thread(<aRunnableObject>).start();
```

- (AFTER) Executing *Runnable* tasks:

```
<anExecutorObject>.execute(<aRunnableObject>);
```

The *Executor* Interface

- Problems with the older technique:
  - Thread creation is expensive.
    - Need stack and heap space
    - May cause out of memory errors
    - Remedy:
      - Use thread pooling
        - Well-designed implementation is not simple
  - Difficulty in cancellation and shutting down of threads

The *Executor* Interface

- Solution to problems with the older technique:
  - Use the *Executor* interface
    - Decouples task submission from the mechanics of how each task will be run

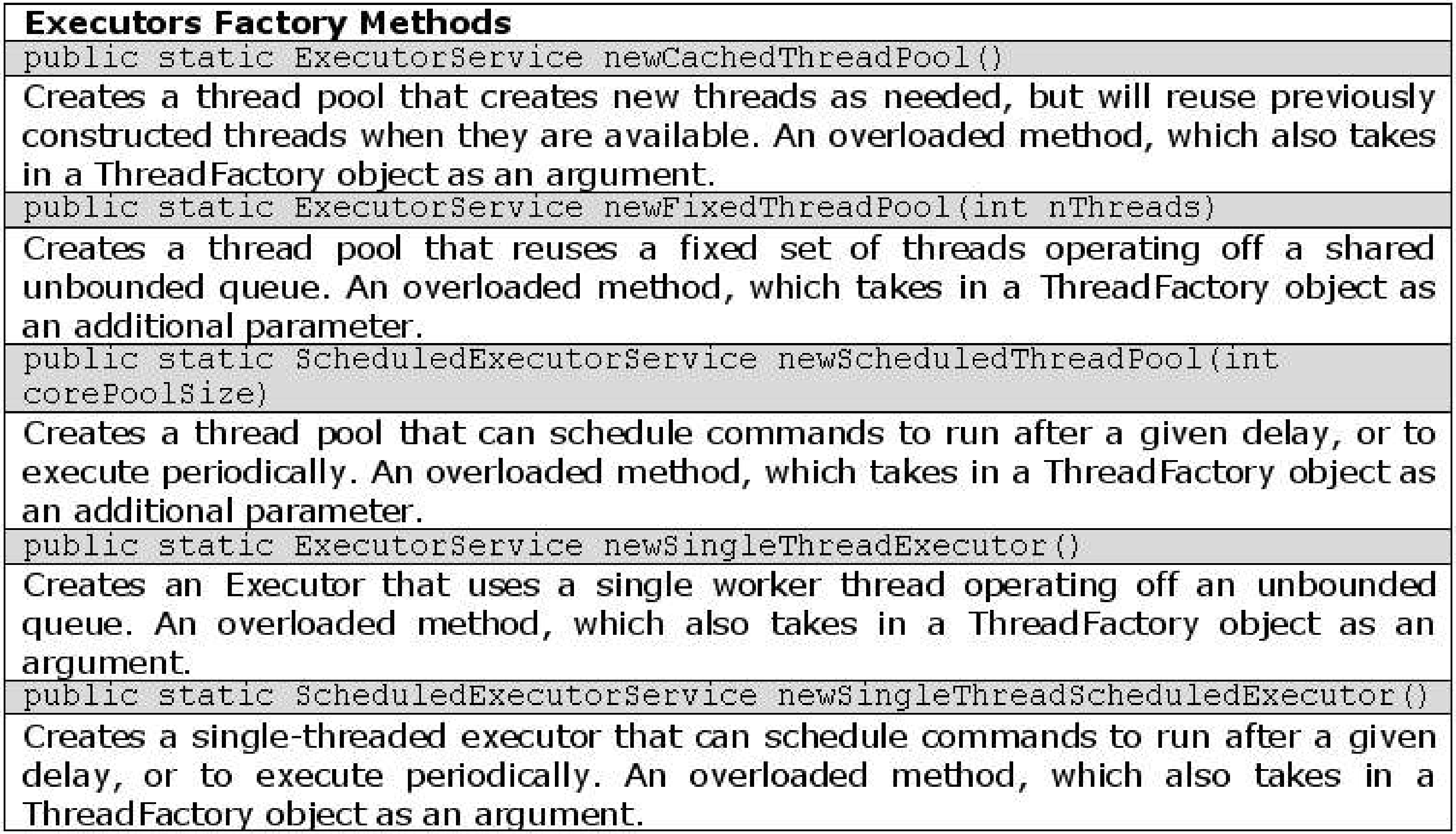
- Using the *Executor* interface:

```
Executor <executorName> = <anExecutorObject>;
<executorName>.execute(new <RunnableTask1>());
<executorName>.execute(new <RunnableTask2>());
...
```

The *Executor* Interface

- Creating an object of *Executor* type:
  - Cannot be instantiated
  - Can create a class implementing this interface
  - Can use factory method provided in the *Executors* class
    - Executors also provides factory methods for simple thread pool management

The *Executors* Factory Methods



Executors Factory Methods
`public static ExecutorService newCachedThreadPool()`
Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. An overloaded method, which also takes in a ThreadFactory object as an argument.
`public static ExecutorService newFixedThreadPool(int nThreads)`
Creates a thread pool that reuses a fixed set of threads operating off a shared unbounded queue. An overloaded method, which takes in a ThreadFactory object as an additional parameter.
`public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`
Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically. An overloaded method, which takes in a ThreadFactory object as an additional parameter.
`public static ExecutorService newSingleThreadExecutor()`
Creates an Executor that uses a single worker thread operating off an unbounded queue. An overloaded method, which also takes in a ThreadFactory object as an argument.
`public static ScheduledExecutorService newSingleThreadScheduledExecutor()`
Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically. An overloaded method, which also takes in a ThreadFactory object as an argument.

The *Executor* Interface

- Controls execution and completion of *Runnable* tasks

- Killing threads:

```
executor.shutdown();
```

The *Callable* Interface

- Recall:
  - Two way of creating threads:
    - Extend *Thread* class
    - Implement *Runnable* interface
  - Should override the *run* method

```
public void run() //no throws clause
```

- *Callable* interface
  - *Runnable* interface without the drawbacks

The *Callable* Interface

- (BEFORE) Getting result from a *Runnable* task:

```
public MyRunnable implements Runnable {
  private int result = 0;
  public void run() {
    ...
    result = someValue;
  }
  /* The result attribute is protected from changes
     from other codes accessing this class */
  public int getResult() {
    return result;
  }
}
```

The *Callable* Interface

- (AFTER) Getting result from a *Runnable* task:

```
import java.util.concurrent.*;

public class MyCallable implements Callable {
  public Integer call() throws java.io.IOException {
    ...
    return someValue;
  }
}
```

The *Callable* Interface

- The *call* method

  ```
  V call throws Exception
  ```

  where

  V is a generic type.

- Other concurrency features still
  - J2SE 5.0 API documentation

Summary

- Threads
  - Thread Definition
  - Thread States
    - Running
    - Ready to run
    - Resumed
    - Suspended
    - Blocked
  - Priorities
    - Range from 1 to 10
    - Context switch

Summary

- The *Thread* Class
  - Constructor
  - Constants
    - MAX_PRIORITY
    - MIN_PRIORITY
    - NORM_PRIORITY
  - Methods
    - currentThread()
    - getName()
    - setName(String name)
    - getPriority()
    - isAlive()
    - join([long millis, [int nanos]])
    - sleep(long millis)
    - run()
    - start()

Summary

- Creating Threads
  - Extending the *Thread* Class
  - Implementing the *Runnable* Interface
  - Extending vs. Implementing
- Synchronization
  - Locking an Object
  - The *synchronized* keyword
    - Method header
    - Object

Summary

- Interthread Communication
  - Methods
    - wait
    - notify
    - notifyAll
- Concurrency Utilities
  - The *Executor* Interface
    - The *Executors* factory methods
  - The *Callable* Interface