Advance java – Multi Threading

Fresher Learning Program
January, 2012

People matter, results count.

Objectives of Java Threads

- Purpose:
  - Understanding of threads and multithreading programs in Java

- Product:
  - To understand what is thread, and how it is different from process
  - Two ways of creating thread in Java
  - How to write multithreading programs in Java
  - Concept of thread safety, and how to achieve it
  - Various methods of thread class and inter thread communications

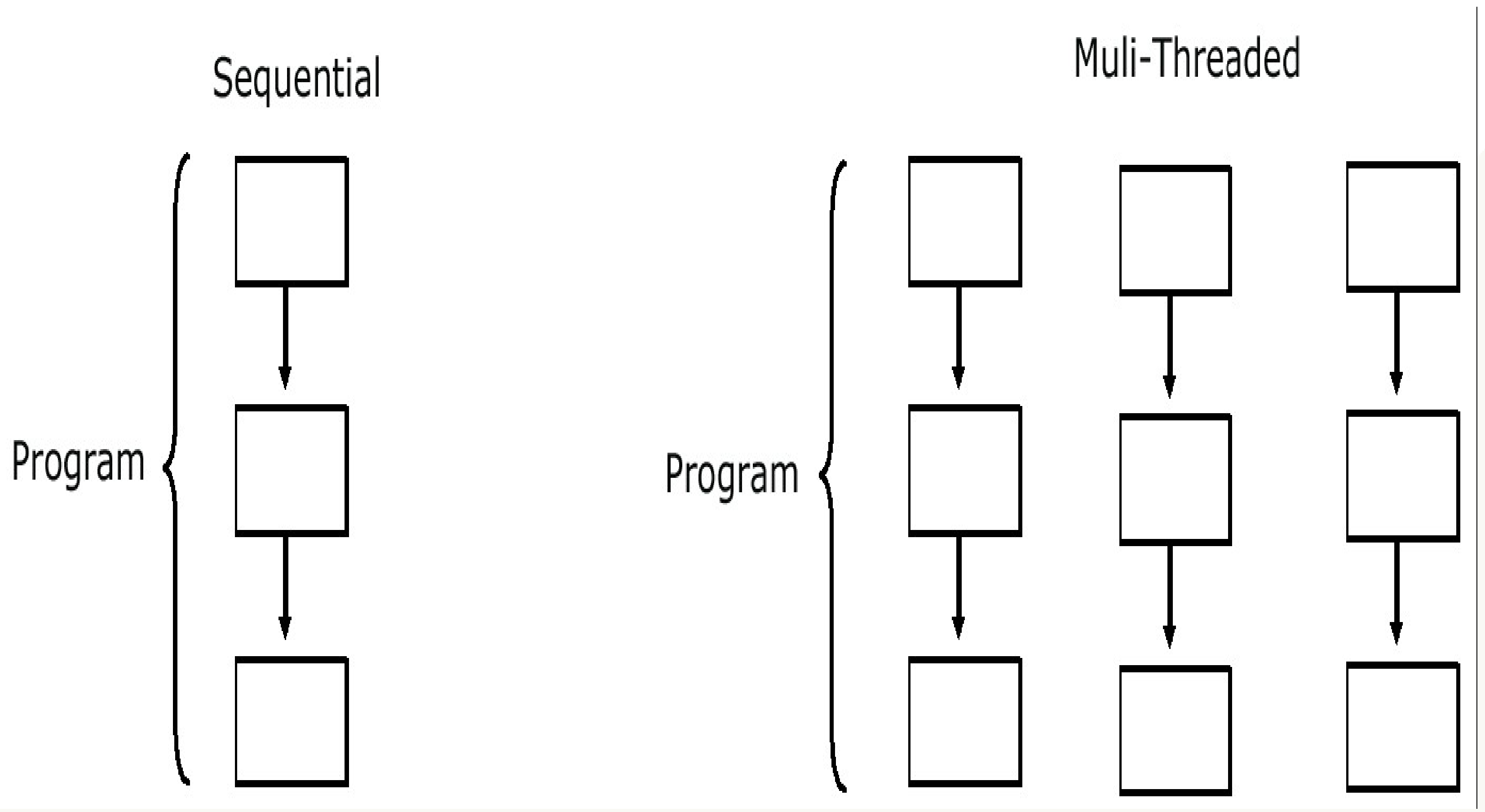
- Process:
  - Theory Sessions along with assignments
  - A recap at the end of the session in the form of Quiz

Table of Contents

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
    - Hot Java web browser

Threads



Multi-threading in Java Platform

- Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling
- But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads

Thread States

A thread can in one of several possible states:

1.Running Currently running In control of CPU

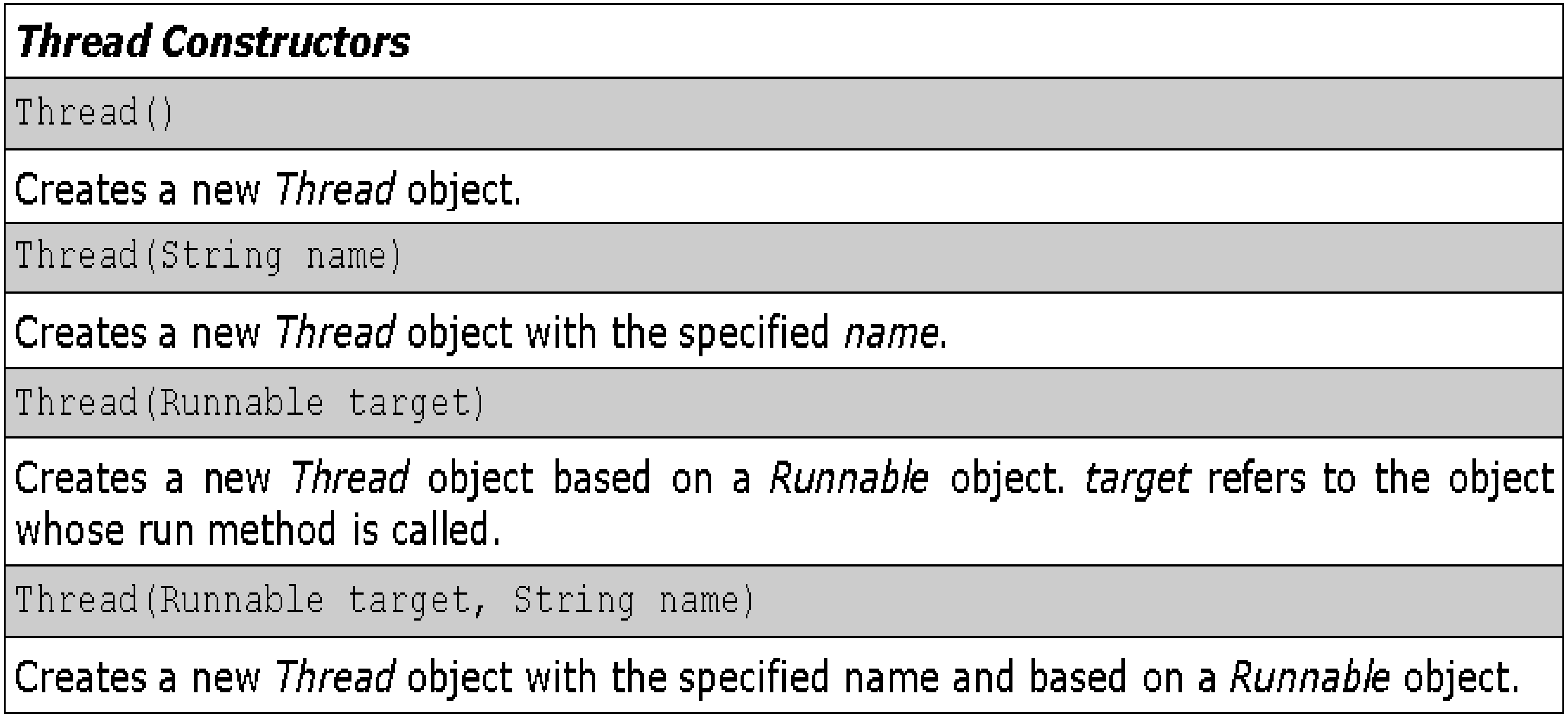
2.Ready to run Can run but not yet given the chance

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

The *Thread Class: Constructor*



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.

Threads…

Two Ways of Creating and Starting a Thread

1. Extending the *Thread class*
2. Implementing the *Runnable interface*

Extending Thread Class

- The subclass extends *Thread class*
- The subclass overrides the *run() method of Thread class*
- An object instance of the subclass can then be created
- Calling the *start() method of the object instance* starts the execution of the thread
- Java runtime starts the execution of the thread by calling *run() method of object instance*

Two Schemes of starting a thread from a subclass

Scheme #1: The *start() method is not in the* constructor of the subclass

– The start() method needs to be explicitly invoked after object instance of the subclass is created in order to start the thread

Scheme #2: The *start() method is in the constructor* of the subclass

– Creating an object instance of the subclass will start the thread

Scheme #1: start() method is Not in the constructor of subclass

```
class PrintNameThread extends Thread {

    PrintNameThread(String name) {
            super(name);
    }

    public void run() {
            String name = getName();
            for (int i = 0; i < 100; i++) {
                    System.out.print(name);
            }
    }

}
```

```
class ExtendThreadClassTest1 {

   public static void main(String args[]) {
              PrintNameThread pnt1 =new PrintNameThread("A");
              pnt1.start(); // Start the first thread
              PrintNameThread pnt2 = new PrintNameThread("B");
              pnt2.start(); // Start the second thread
   }

}
```

Scheme #2: start() method is in aconstructor of the subclass

```
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);
                }
        }

}
```

```
class ExtendThreadClassTest2 {

        public static void main(String args[]) {
                PrintNameThread pnt1 = new PrintNameThread("A");
                PrintNameThread pnt2 = new PrintNameThread("B");
        }

}
```

```
class ExtendThreadClassTest3 {

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

Implementing Runnable Interface

Runnable Interface

● The *Runnable interface should be implemented by* any class whose instances are intended to be executed as a thread

● The class must define run() method of no arguments

– The run() method is like main() for the new thread

● Provides the means for a class to be active while not sub classing *Thread*

– A class that implements *Runnable can run without* sub classing *Thread by instantiating a Thread instance* and passing itself in as the target

Two Ways of Starting a Thread For a class that Implements Runnable

- **Scheme #1:** Caller thread creates Thread object and starts it explicitly after an object instance of the class that implements Runnable interface is created

  – The start() method of the Thread object needs to be explicitly invoked after object instance is created

- **Scheme #2:** The Thread object is created and started within the constructor method of the class that implements Runnable interface

  – The caller thread just needs to create object instances of the Runnable class

Scheme #1:

PrintNameRunnable implements Runnable interface

```
class PrintNameRunnable implements Runnable {
   String name;
   PrintNameRunnable(String name) {
              this.name = name;
   }

   // Implementation of the run() defined in the Runnable interface.
   public void run() {
              for (int i = 0; i < 10; i++) {
                          System.out.print(name);
              }
   }
}
```

```
public class RunnableThreadTest1 {

   public static void main(String args[]) {
             PrintNameRunnable pnt1 = new PrintNameRunnable("A");
             Thread t1 = new Thread(pnt1);
             t1.start();
   }

}
```

Scheme #2:

```
// PrintNameRunnable implements Runnable interface

class PrintNameRunnable implements Runnable {
        Thread thread;
        PrintNameRunnable(String name) {
                thread = new Thread(this, name);
                thread.start();
        }
        // Implementation of the run() defined in the Runnable interface.
        public void run() {
                String name = thread.getName();
                for (int i = 0; i < 10; i++) {
                        System.out.print(name);
                }
        }
}
```

```
public class RunnableThreadTest2 {

        public static void main(String args[]) {
                /* Since the constructor of the PrintNameRunnable object
                * creates a Thread object and starts it,
                * there is no need to do it here.
                */
                new PrintNameRunnable("A");
                new PrintNameRunnable("B");
                new PrintNameRunnable("C");
        }
}
```

Extending Thread vs. Implementing Runnable Interface

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

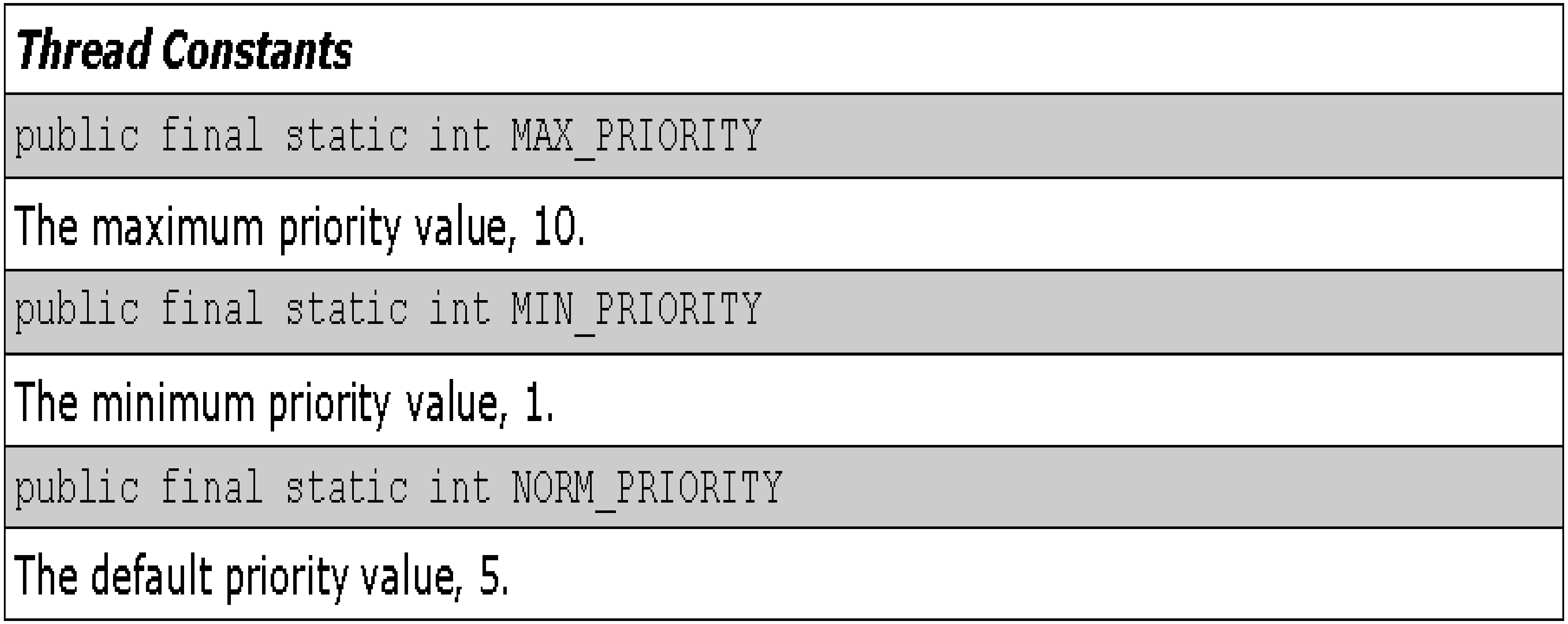
Thread Priorities

- Why priorities?
  - – Determine which thread receives CPU control and getsto 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

The *Thread Class: Constants*

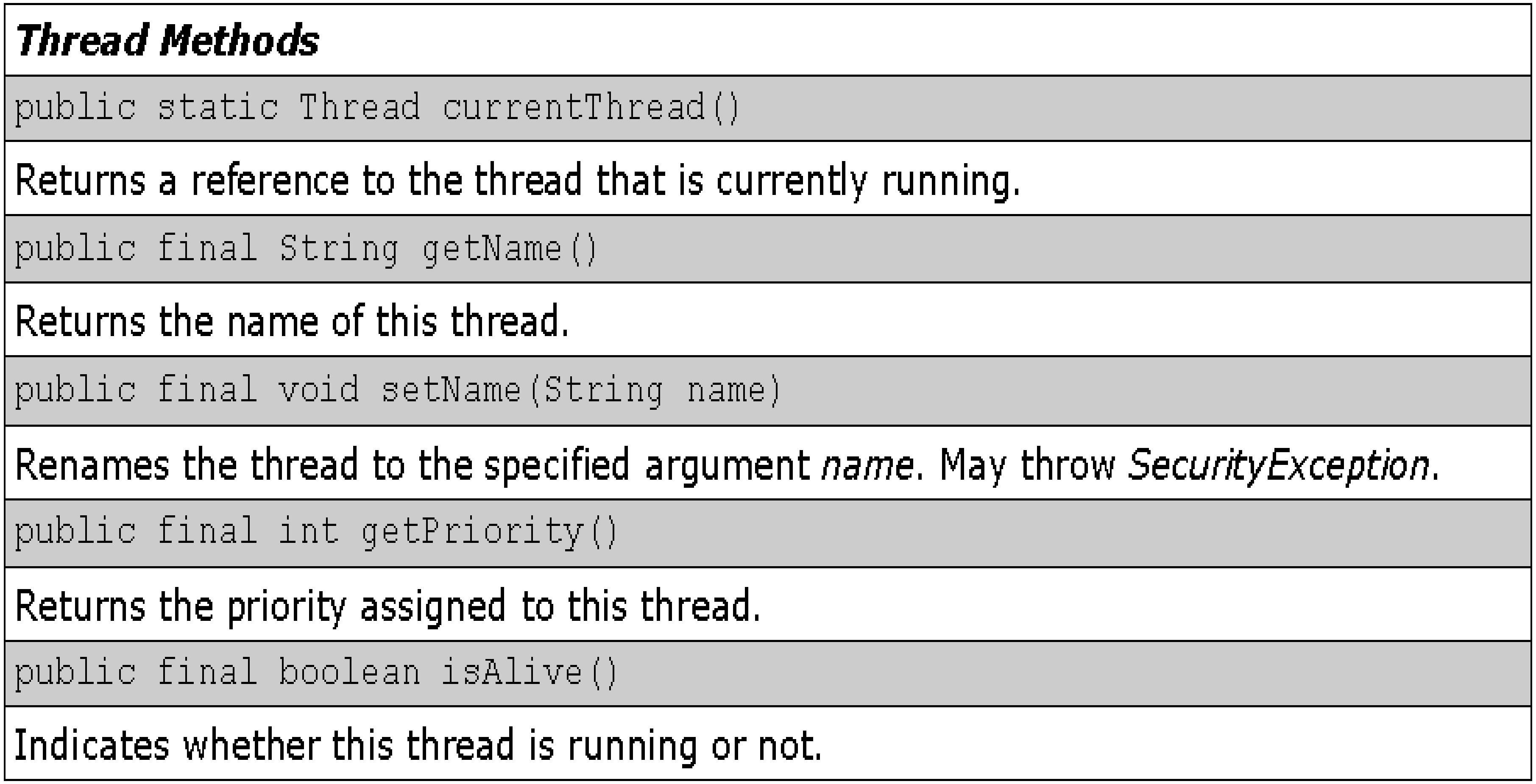
Contains fields for priority values



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.

Synchronization

Race condition & How to Solve it

Race conditions occur when multiple, asynchronously executing threads access the same object (called a shared resource) returning unexpected (wrong) results

Example:

– Threads often need to share a common resource ie a file, with one thread reading from the file while another thread writes to the file

- They can be avoided by synchronizing the threads which access the shared resource

Synchronization:Locking an Object

- A thread is synchronized by becoming an owner of the object's monitor
  - – Consider it as locking an object

- A thread becomes the owner of the object's monitor in one of three ways
  - – Option 1: Use *synchronized method*
  - – Option 2: Use *synchronized statement on a common* object

Use synchronized method

```
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...

```
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);
        }
  }
```

```
class TestThread {

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

Executing SynchronizedMethod

Sample output:

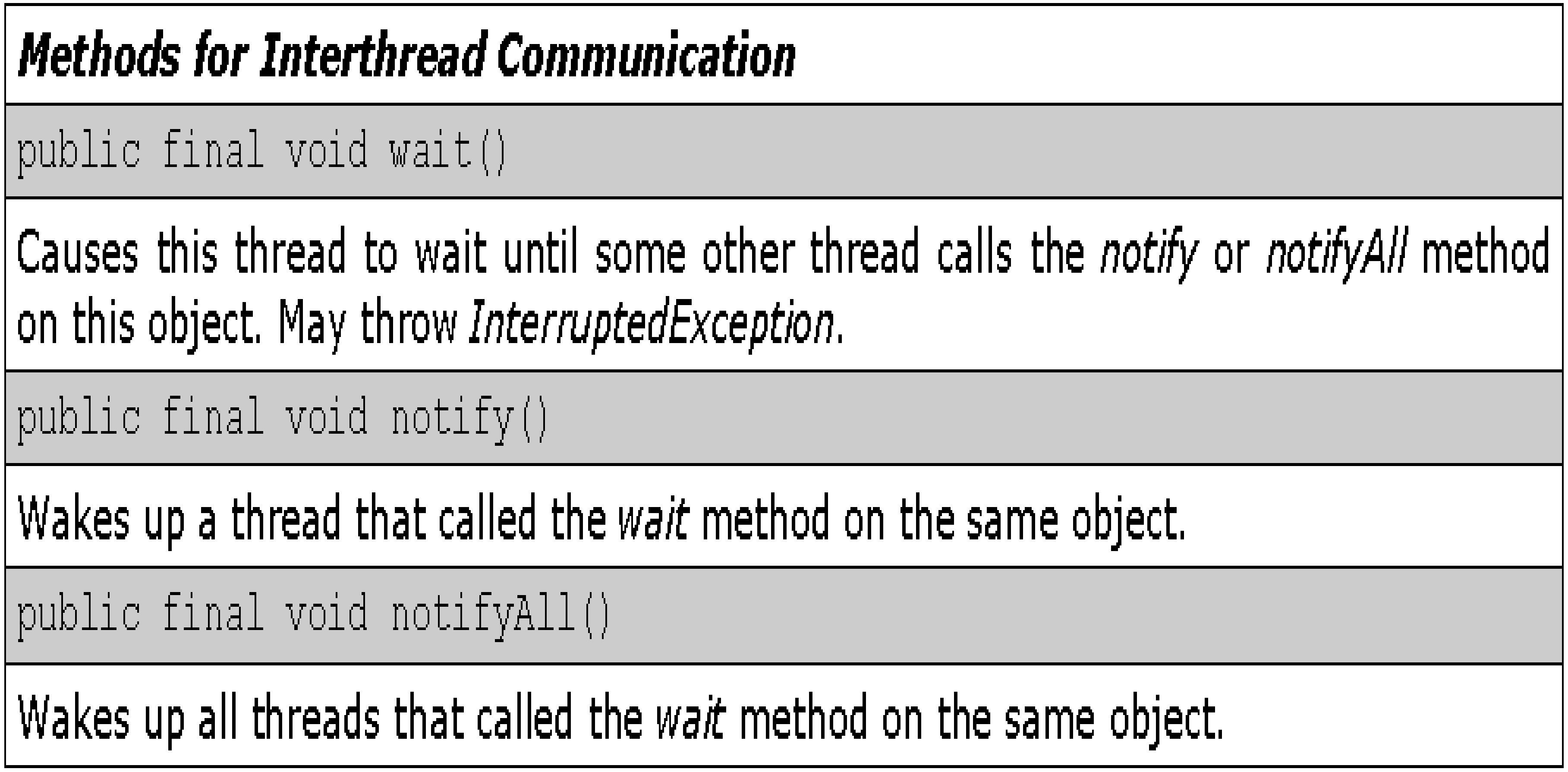
```
        Hello there.

        How are you?

        Thank you very much!
```

Inter-thread Synchronization

Methods related Inter thread communication



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.

wait() method of Object Class

- wait() method causes a thread to release the lock it is holding on an object; allowing another thread to run
- wait() method is defined in the Object class
- wait() can only be invoked from within synchronizedcode
- it should always be wrapped in a try block as it throws IOExceptions
- wait() can only invoked by the thread that own's thelock on the object

notify() method

- Wakes up a single thread that is waiting on this object's monitor
  - If any threads are waiting on this object, one of them is chosen to be awakened
  - The choice is arbitrary and occurs at the discretion of the implementation

- Can only be used within synchronized code

- The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object

Timer Class

- Provides a facility for threads to schedule tasks for future execution in a background thread

- Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

- Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially

- Timer tasks should complete quickly

  – If a timer task takes excessive time to complete, it "hogs“ the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch

Timer Class

up" and execute in rapid succession when (and if) the offending task finally completes.

- Abstract class with an abstract method called run()
- Concrete class must implement the run() abstract method

Example: Timer Class

```
public class TimerReminder {
        Timer timer;
        public TimerReminder(int seconds) {
                timer = new Timer();
                timer.schedule(new RemindTask(), seconds*1000);
        }
        class RemindTask extends TimerTask {
                public void run() {
                        System.out.format("Time's up!%n");
                        timer.cancel();     //Terminate the timer thread
                }
        }
        public static void main(String args[]) {
                System.out.format("About to schedule task.%n");
                new TimerReminder(5);
                System.out.format("Task scheduled.%n");
        }
}
```

Recap (Keywords)

sleep

wait

Runnable

thread priority

dead lock

lock

yeild

blocked thread

timer class

synchronized block

Thank You For Your Time

People matter, results count.