Multithreading
Chapter 30

Introduction

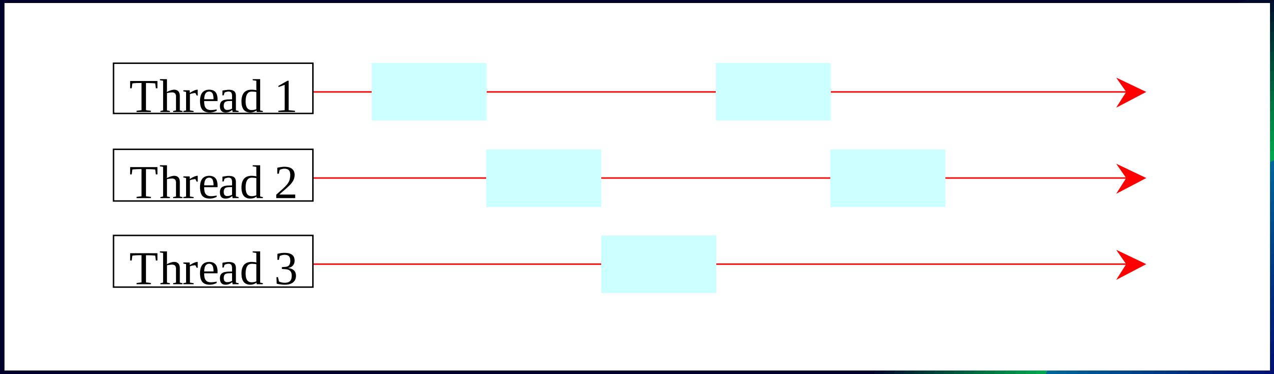
Multithreading: The ability for multiple tasks in a program to be executed concurrently.

Thread: A program may consist of many tasks that can run concurrently. A thread is the flow of execution, from beginning to end, of a task.

Threads Concept

In single-processor systems, as shown in the figure below, the multiple threads share CPU time, known as *time sharing*, and the operating system is responsible for scheduling and allocating resources to them.

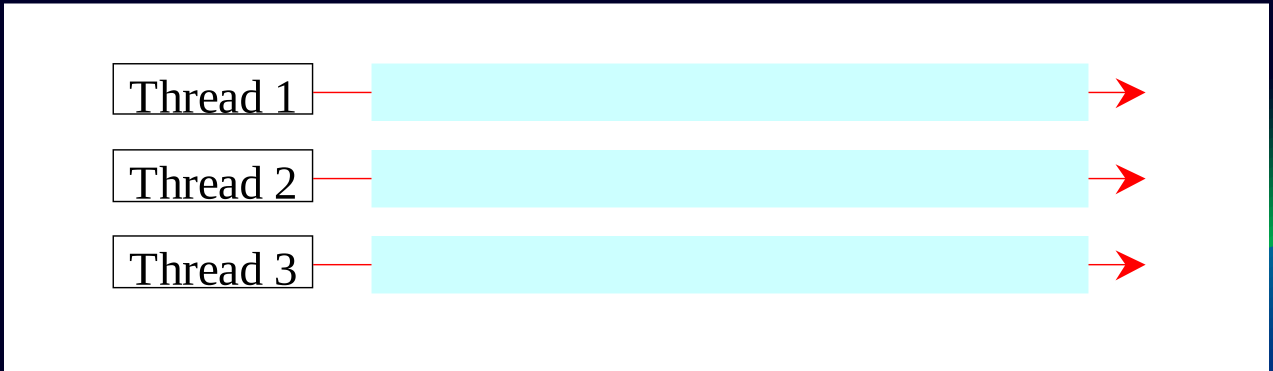
Multiple threads sharing a single CPU



Threads Concept

On the other hand, threads can be executed simultaneously in multiprocessor systems.

Multiple threads on multiple CPUs



Threads Concept

- When your program executes as an application, the Java interpreter starts a thread for the **main** method.
- You can create additional threads to run concurrent tasks in the program.
- In Java, each task is an instance of the **Runnable** interface, also called a *runnable object*.
- A *thread* is essentially an object that facilitates the execution of a task.

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no argu

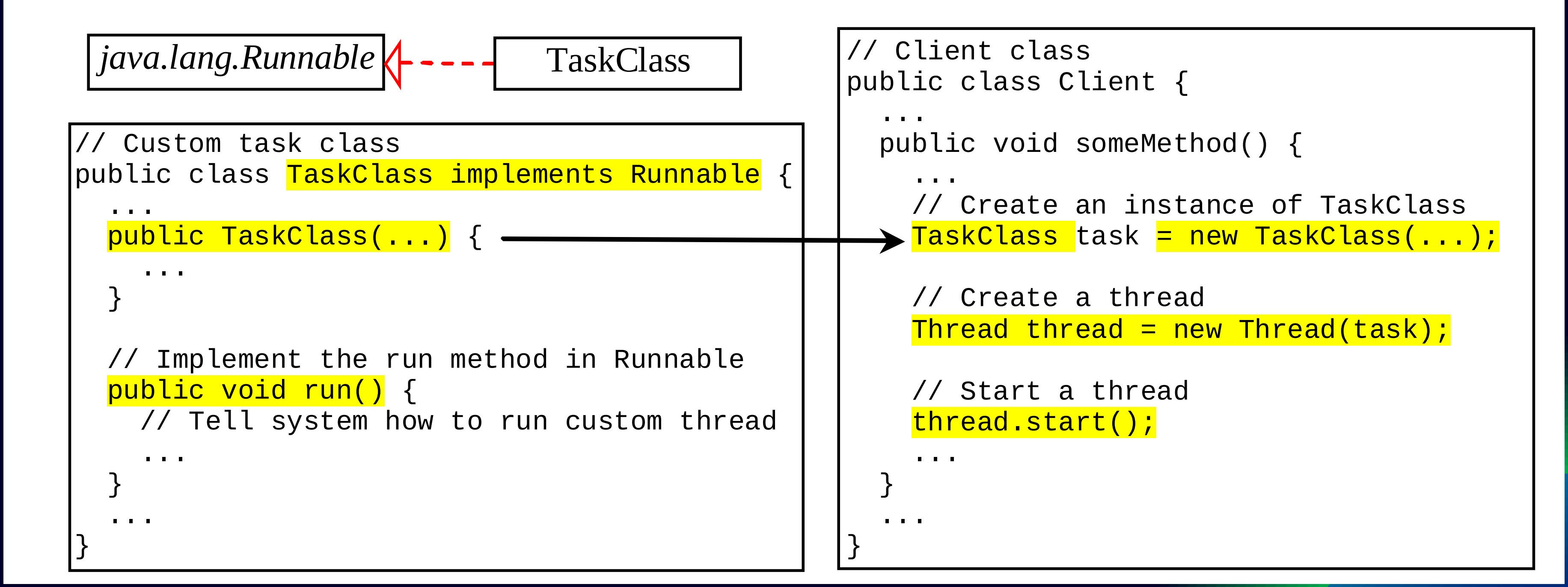
The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no a

Creating Tasks and Threads

- *In order to create a thread, you must create a task class and a thread object.*
- *A task class must implement the* **Runnable** *interface.*
- *A task must be run from a thread.*

- *The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.*
- *The class must define a method of no arguments called run.*
- *When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.*
- *The general contract of the method run is that it may take any action whatsoever.*

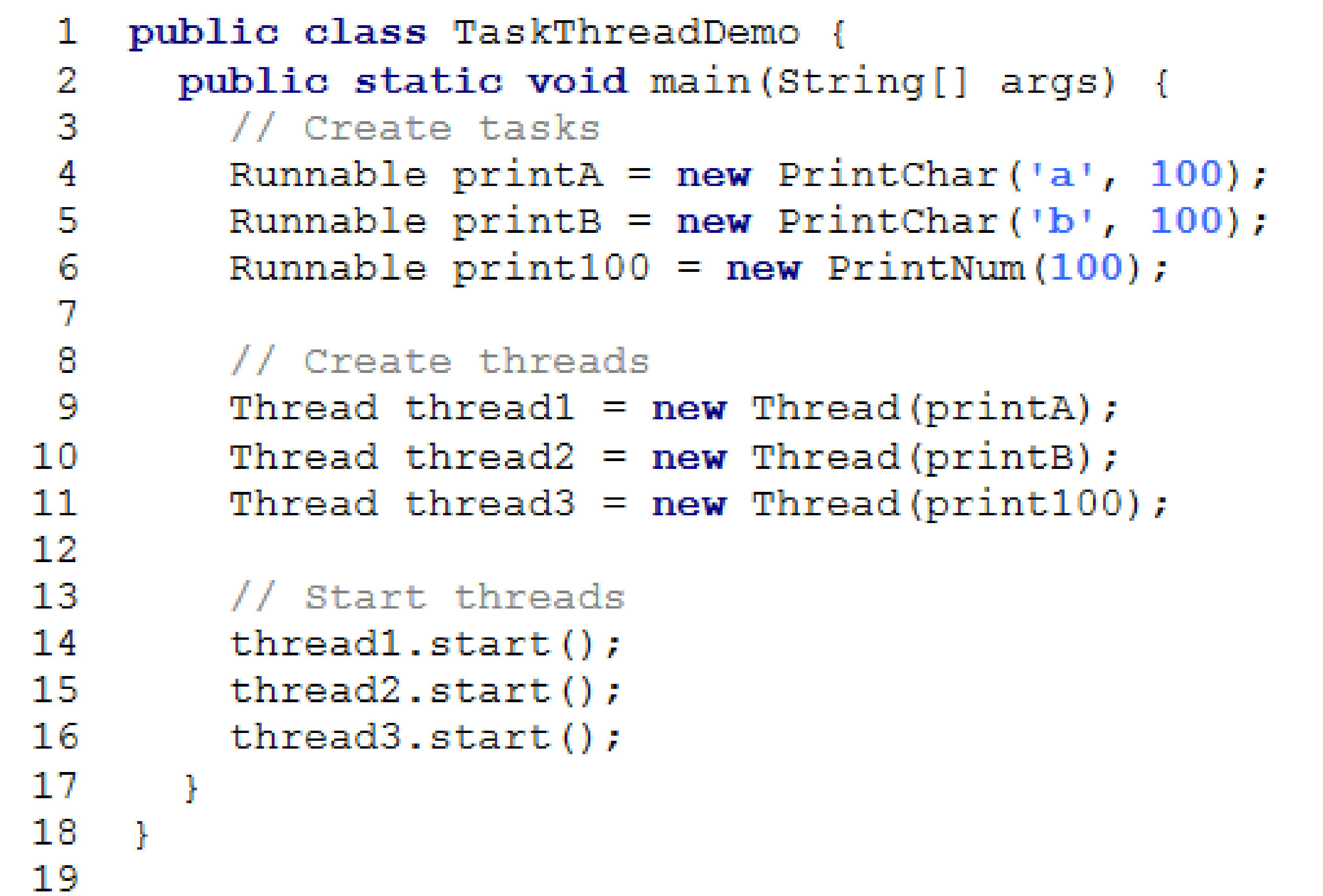
Creating Tasks and Threads



```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

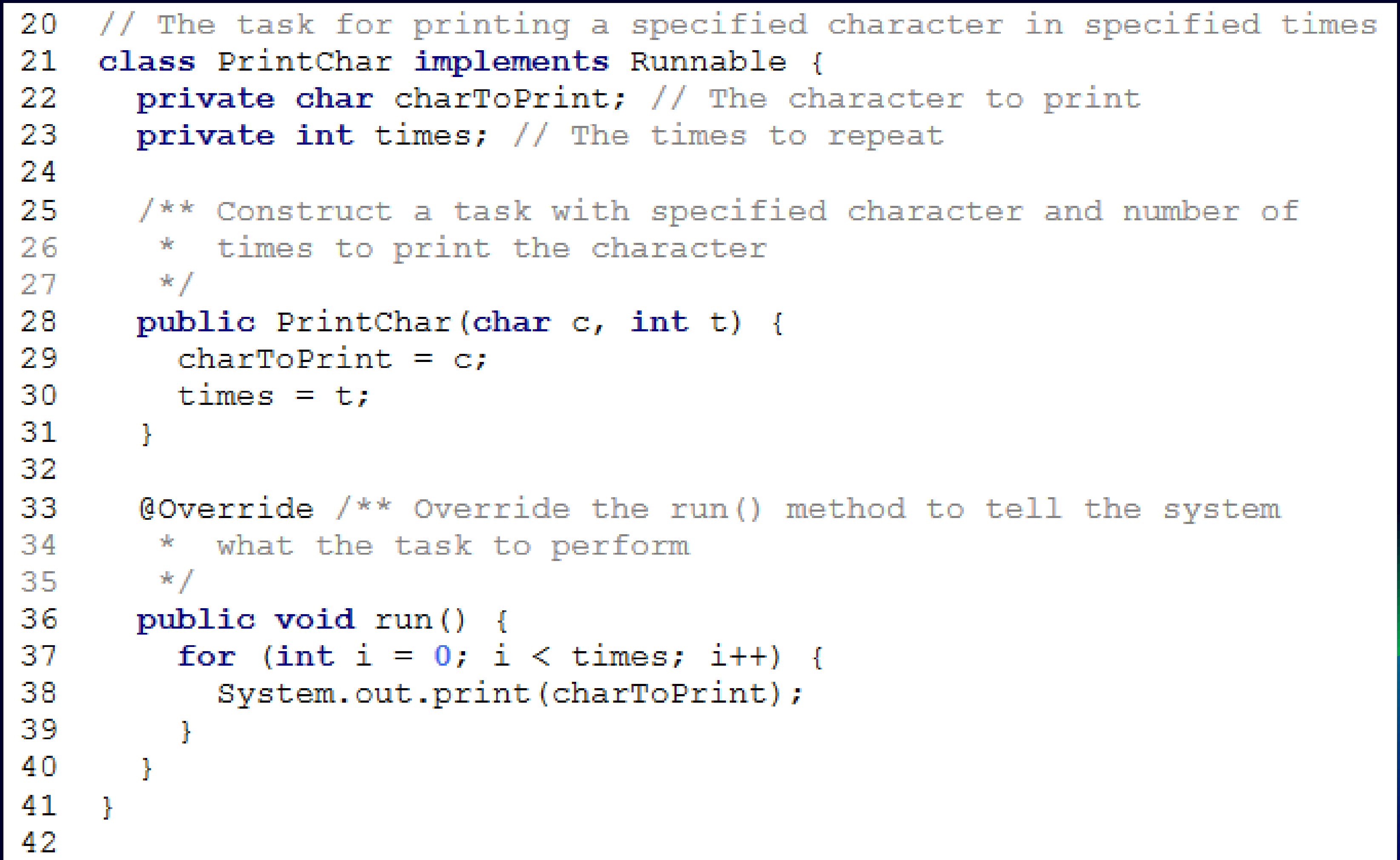
```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Example: Using the Runnable Interface to Create and Launch Threads (1 of 3)



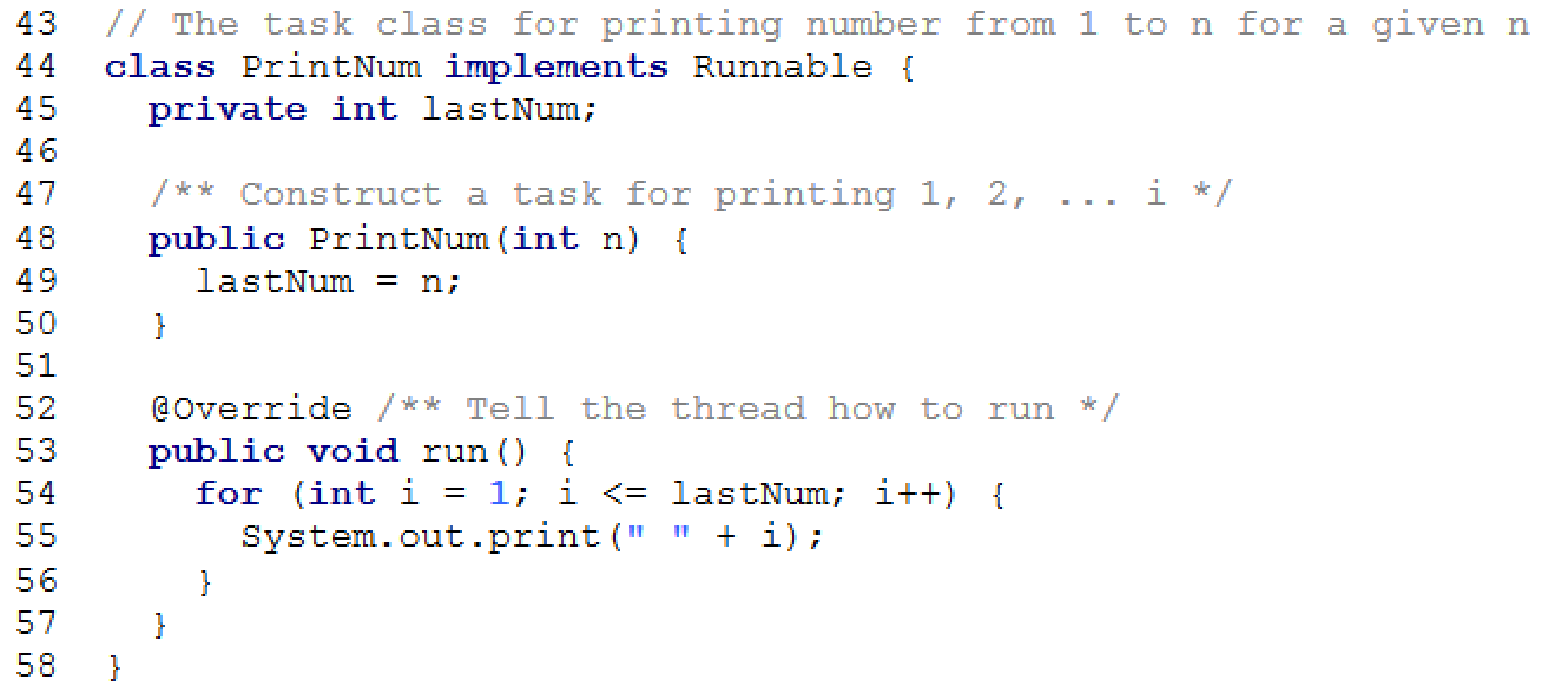
```java
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

Example: Using the `Runnable` Interface to Create and Launch Threads (2 of 3)



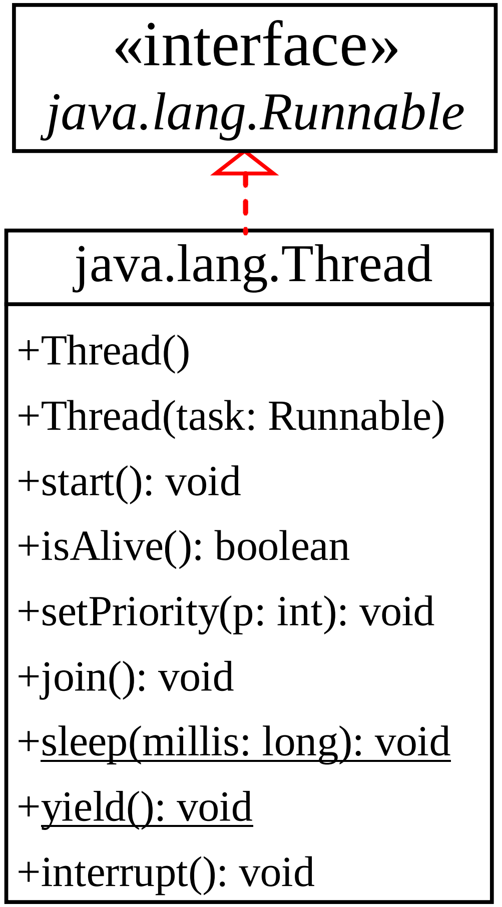
```java
// The task for printing a specified character in specified times
class PrintChar implements Runnable {
  private char charToPrint; // The character to print
  private int times; // The times to repeat

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

  @Override /** Override the run() method to tell the system
   *  what the task to perform
   */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}

```

Example: Using the `Runnable` Interface to Create and Launch Threads (3 of 3)



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

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

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

The Thread Class



«interface»
java.lang.Runnable

java.lang.Thread

+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

The Static yield() Method

You can use the yield() method to temporarily release time for other threads. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:

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

Every time a number is printed, the print100 thread is yielded. So, the numbers are printed after the characters.

The Static sleep(milliseconds) Method

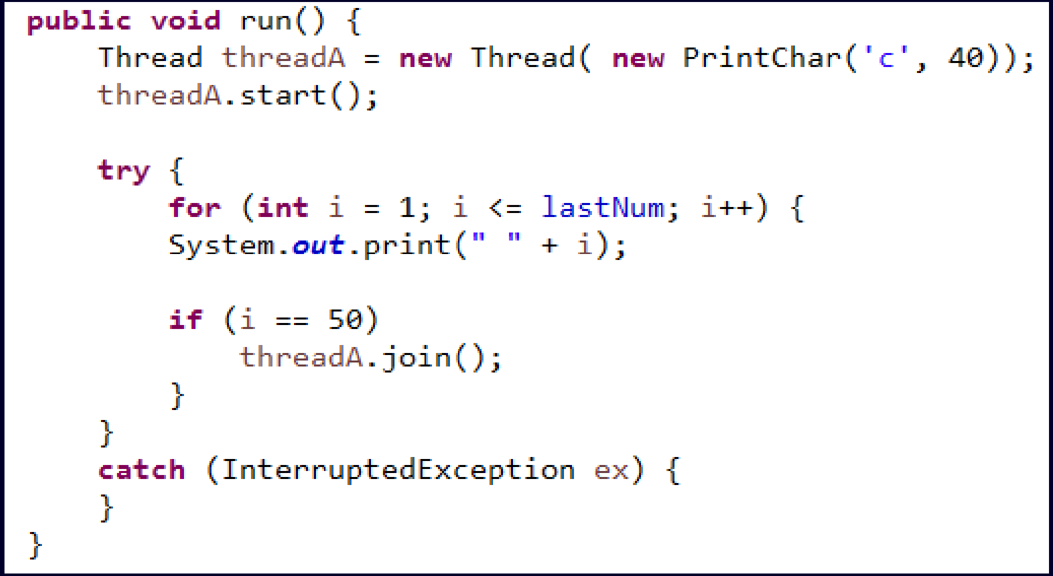
The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

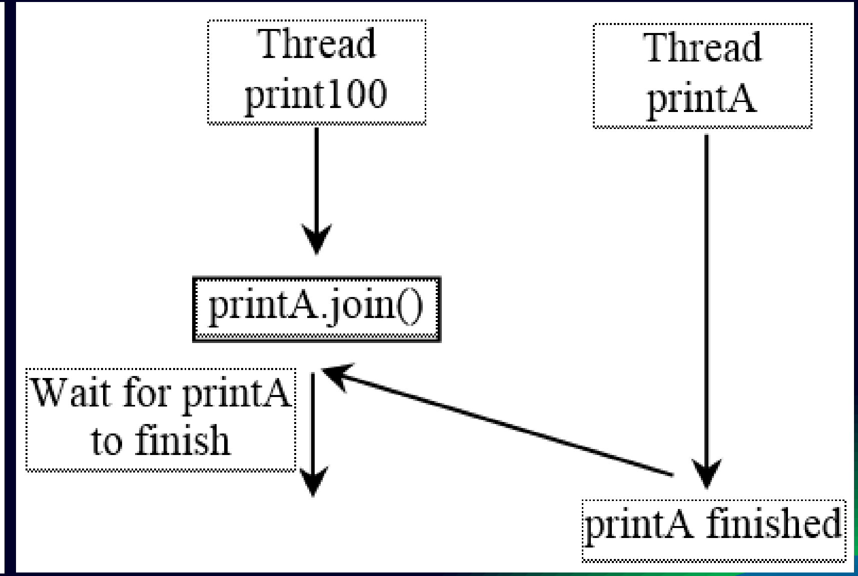
Every time a number (>= 50) is printed, the print100 thread is put to sleep for 1 millisecond.

The
() Method

You can use the join() method to force one thread to wait for another thread to finish. For example, suppose you modify the code in Lines 53-57 in TaskThreadDemo.java as follows:



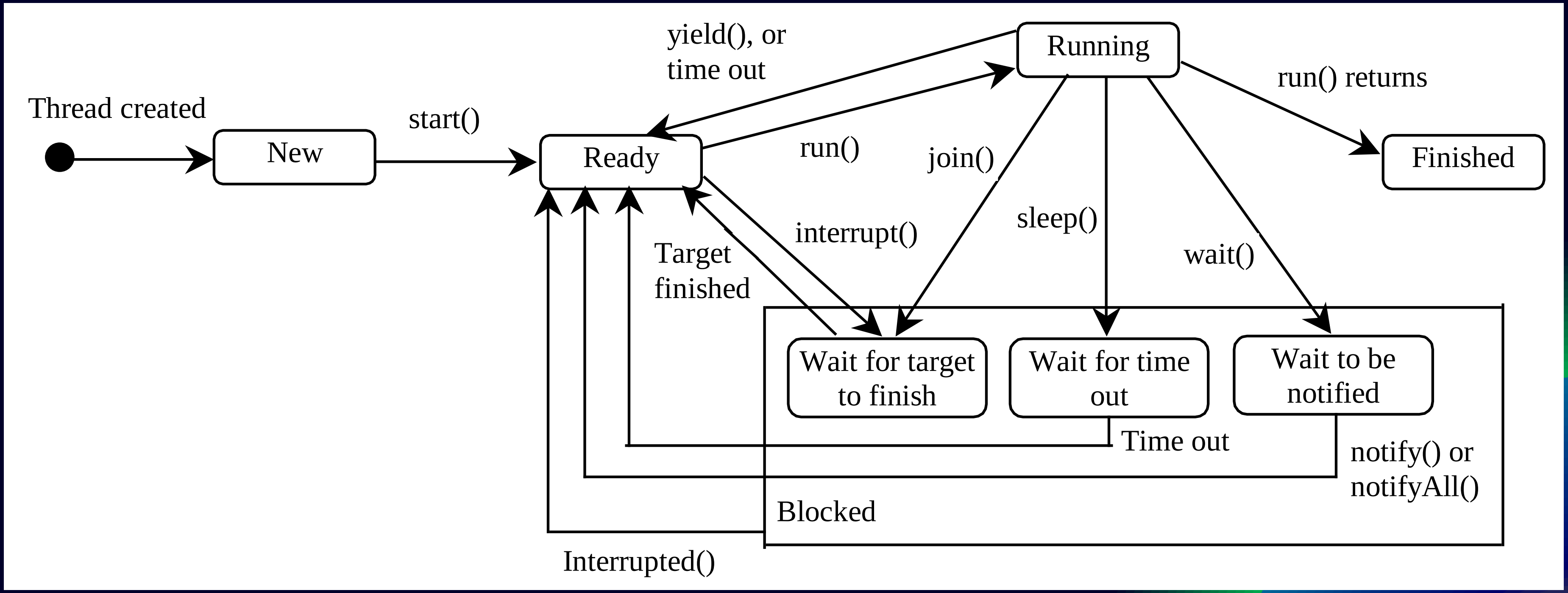
```
public void run() {
    Thread threadA = new Thread( new PrintChar('c', 40));
    threadA.start();

    try {
        for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);

        if (i == 50)
            threadA.join();
        }
    }
    catch (InterruptedException ex) {
    }
}
```



The numbers after 50 are printed after thread printA is finished.

Thread States

A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.



isAlive(), interrupt(), and isInterrupted()

The isAlive() method is used to find out the state of a thread. It returns true if 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 an java.io.InterruptedException is thrown.

The isInterrupted() method tests whether the thread is interrupted.

The deprecated stop(), suspend(), and resume() Methods

NOTE: The Thread class also contains the stop(), suspend(), and resume() methods. As of Java 2, these methods are *deprecated* (or *outdated*) because they are known to be inherently unsafe. You should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.

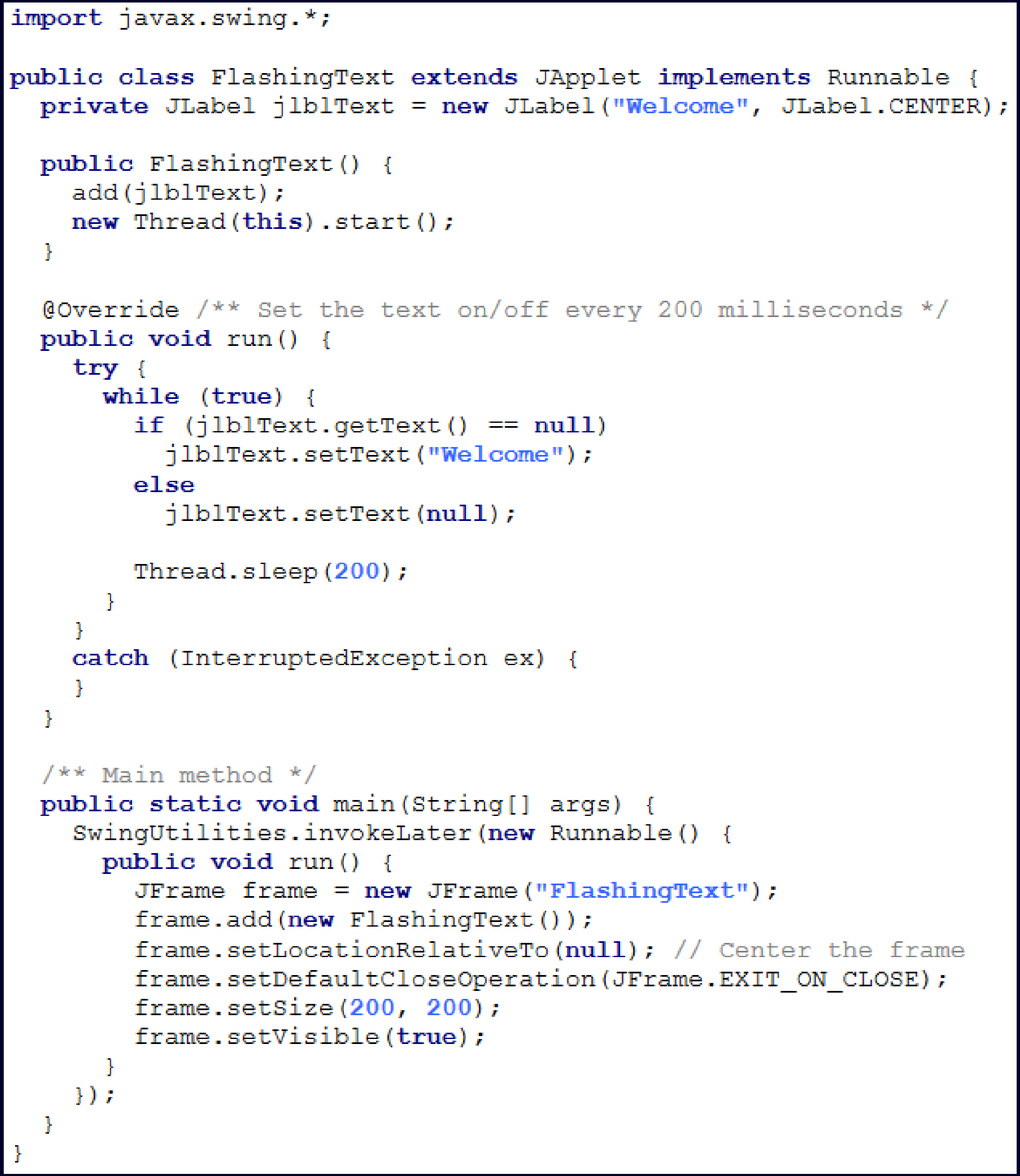
Thread Priority

- Each thread is assigned a default priority of `Thread.NORM_PRIORITY`. You can reset the priority using `setPriority(int priority)`.

- Some constants for priorities include

```
Thread.MIN_PRIORITY
Thread.MAX_PRIORITY
Thread.NORM_PRIORITY
```

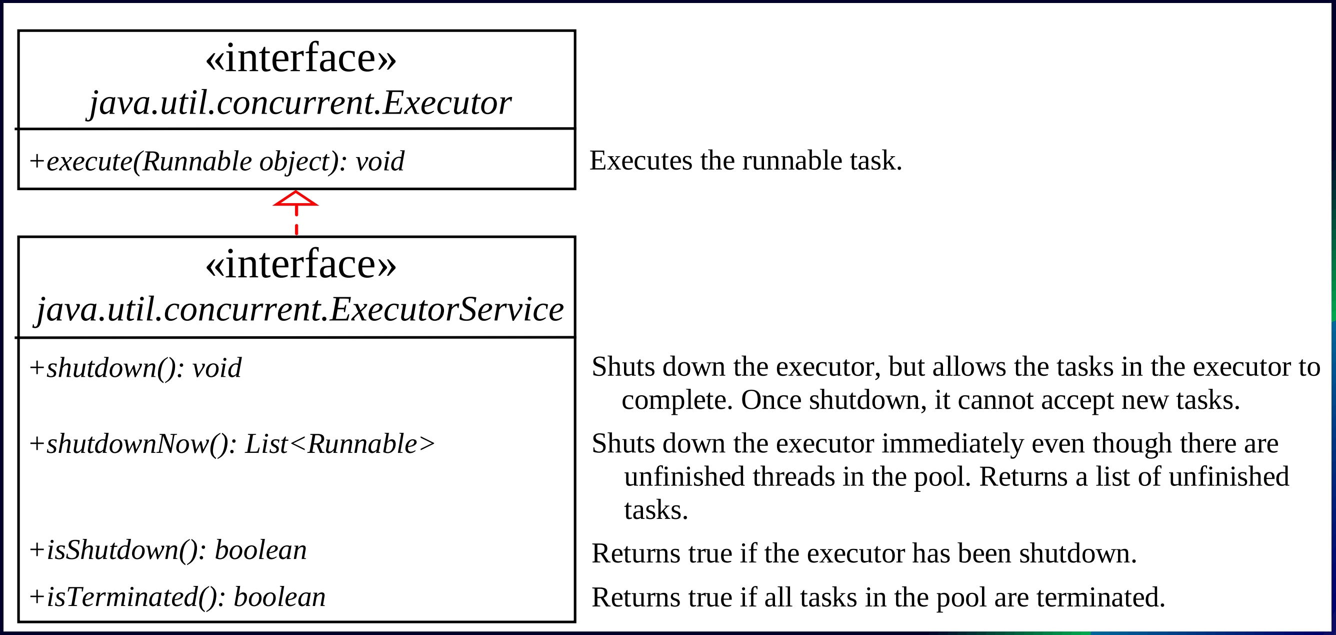
Example: Flashing Text



```
import javax.swing.*;

public class FlashingText extends JApplet implements Runnable {
  private JLabel jlblText = new JLabel("Welcome", JLabel.CENTER);

  public FlashingText() {
    add(jlblText);
    new Thread(this).start();
  }

  @Override /** Set the text on/off every 200 milliseconds */
  public void run() {
    try {
      while (true) {
        if (jlblText.getText() == null)
          jlblText.setText("Welcome");
        else
          jlblText.setText(null);

        Thread.sleep(200);
      }
    }
    catch (InterruptedException ex) {
    }
  }

  /** Main method */
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        JFrame frame = new JFrame("FlashingText");
        frame.add(new FlashingText());
        frame.setLocationRelativeTo(null); // Center the frame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 200);
        frame.setVisible(true);
      }
    });
  }
}
```

Thread Pools

Starting a new thread for each task could limit throughput and cause poor performance. A thread pool is ideal to manage the number of tasks executing concurrently. Starting with JDK 1.5 you can use the Executor interface for executing tasks in a thread pool and the ExecutorService interface for managing and controlling tasks. ExecutorService is a subinterface of Executor.



«interface»
java.util.concurrent.Executor

+execute(Runnable object): void

Executes the runnable task.

«interface»
java.util.concurrent.ExecutorService

+shutdown(): void

Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.

+shutdownNow(): List<Runnable>

Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

+isShutdown(): boolean

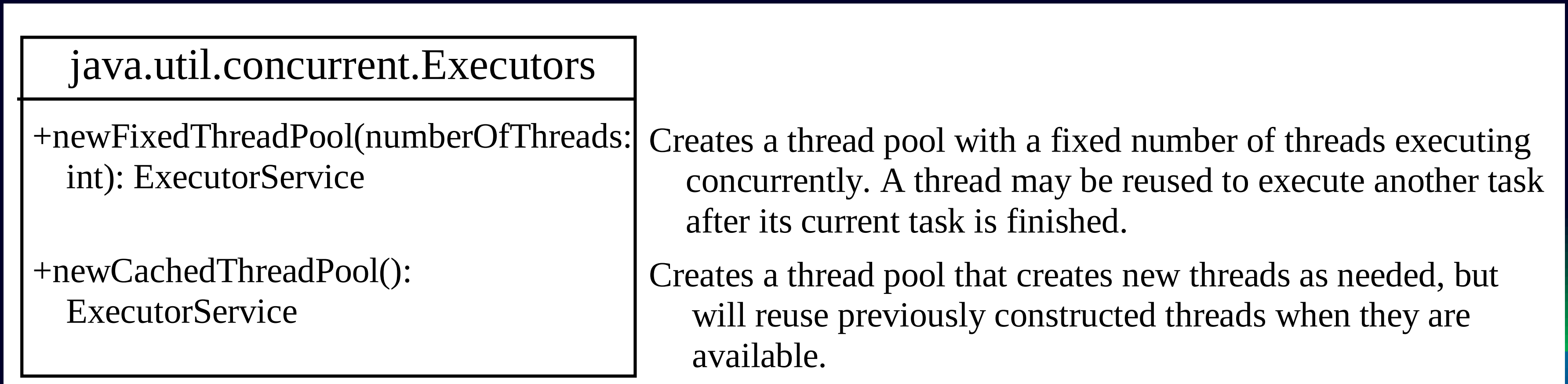
Returns true if the executor has been shutdown.

+isTerminated(): boolean

Returns true if all tasks in the pool are terminated.

Creating Executors

To create an Executor object, use the static methods in the Executors class.



java.util.concurrent.Executors

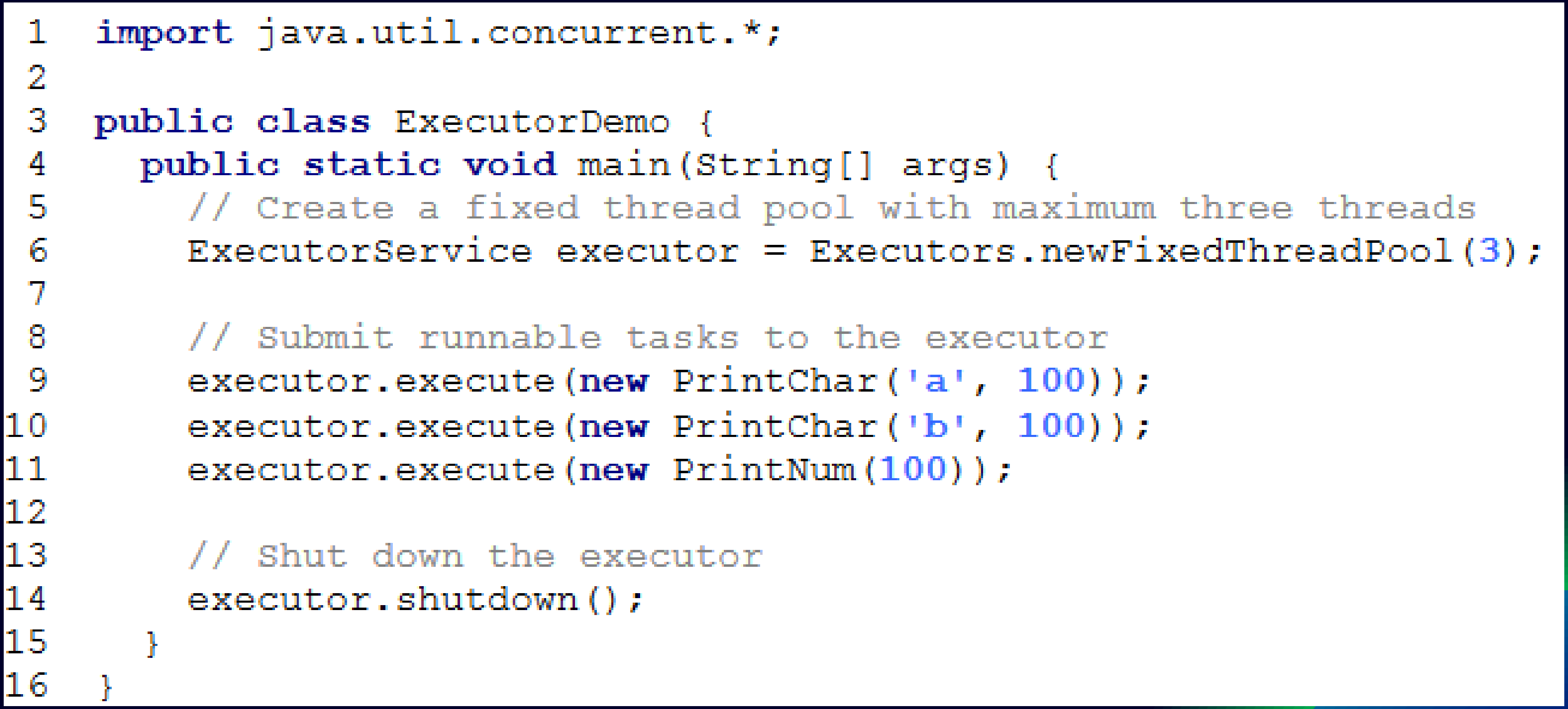
+newFixedThreadPool(numberOfThreads: int): ExecutorService

+newCachedThreadPool(): ExecutorService

Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

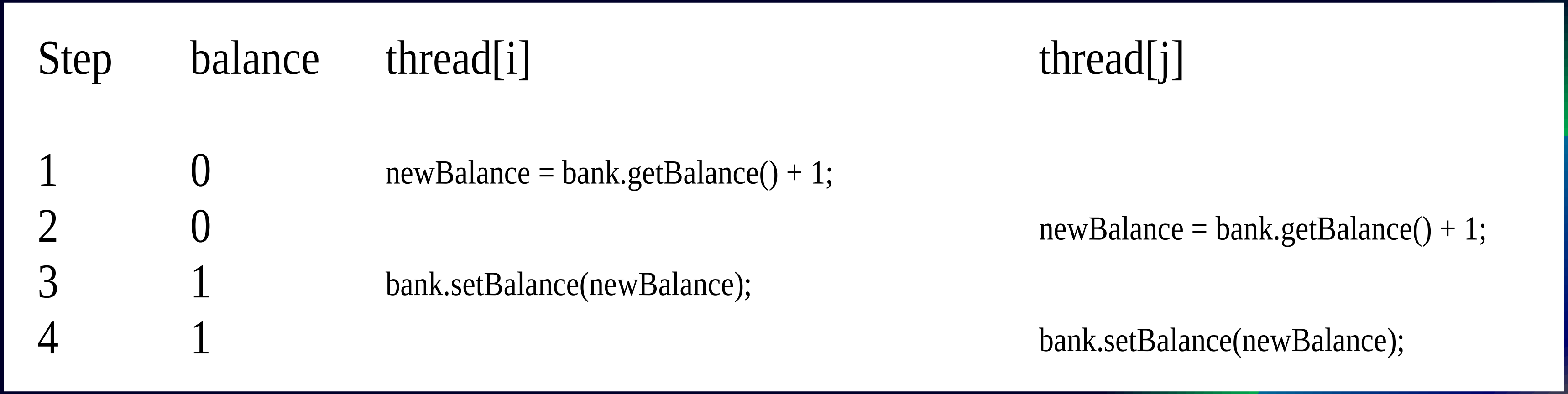
Creating Executors



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

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Shut down the executor
    executor.shutdown();
  }
}
```

Thread Synchronization

A shared resource may be corrupted if it is accessed simultaneously by multiple threads. For example, two unsynchronized threads accessing the same bank account may cause conflict.



Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

Race Condition

What caused the error in the example? Here is a possible scenario:

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;
`			

- The effect of this scenario is that Task 1 did nothing, because in Step 4 Task 2 overrides Task 1's result. Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes conflict. This is a common problem known as a *race condition* in multithreaded programs.

- A class is said to be *thread-safe* if an object of the class does not cause a race condition in the presence of multiple threads.

The `synchronized` keyword

- To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as critical region (critical section).

- You can use the synchronized keyword to synchronize the method so that only one thread can access the method at a time.

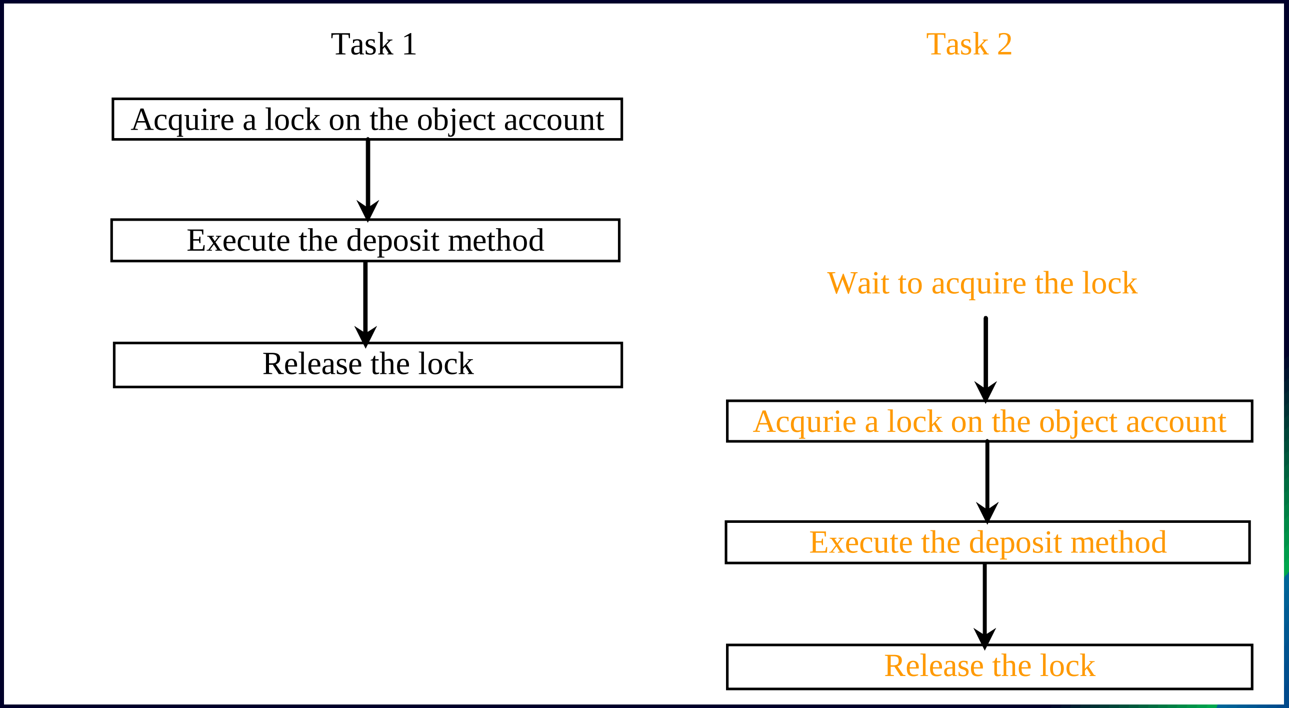
```
public synchronized void withdraw(double amount)
```

Synchronizing Instance Methods and Static Methods

- A synchronized method acquires a lock before it executes.
- In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class.
- If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released.
- Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

Synchronizing Instance Methods and Static Methods

With the deposit method synchronized, the preceding scenario cannot happen. If Task 2 starts to enter the method, and Task 1 is already in the method, Task 2 is blocked until Task 1 finishes the method.


Task 1
Acquire a lock on the object account
Execute the deposit method
Release the lock
Task 2
Wait to acquire the lock
Acqurie a lock on the object account
Execute the deposit method
Release the lock

Synchronizing Statements

Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class.
A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method. This block is referred to as a *synchronized block*. The general form of a synchronized statement is as follows:

```
synchronized (expr) {
  statements;
}
```

The expression expr must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released. When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronizing Statements vs. Methods

Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

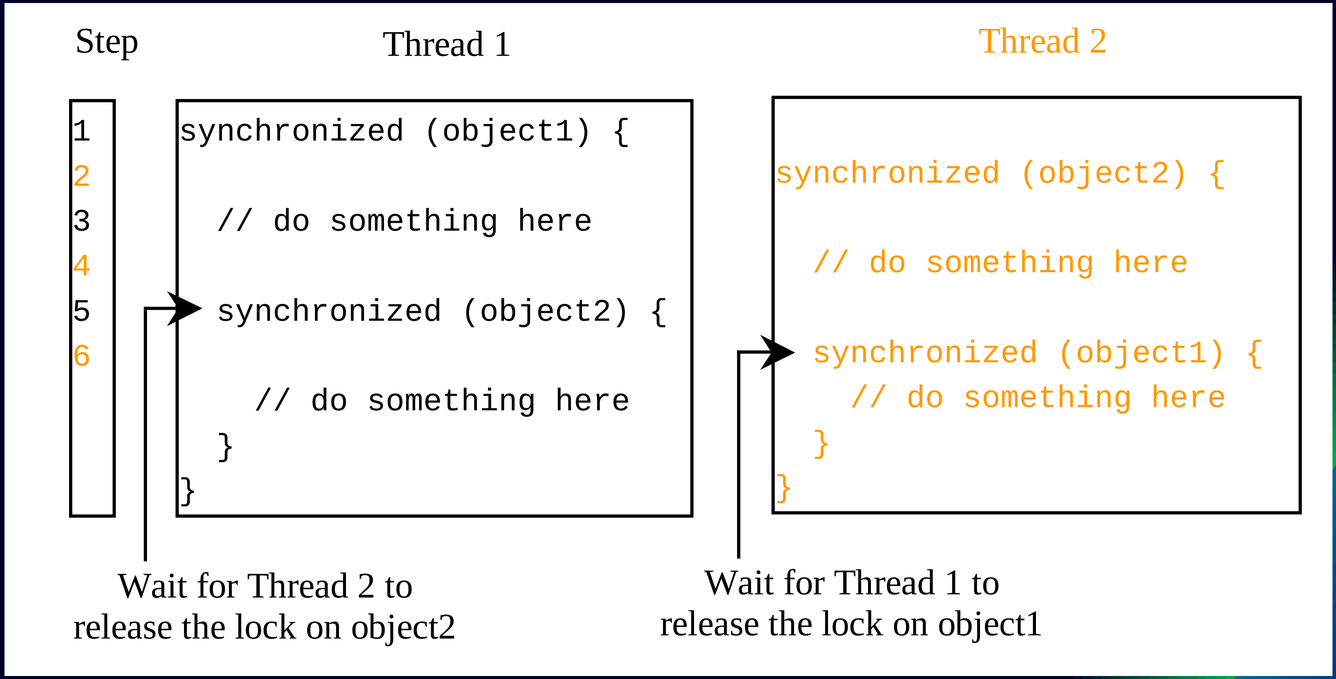
```
public synchronized void xMethod() {
  // method body
}
```

This method is equivalent to

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

Deadlock

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects, as shown below. Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2. Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1. The two threads wait for each other to release the in order to get the lock, and neither can continue to run.



Thread 1:

```
synchronized (object1) {

  // do something here

  synchronized (object2) {

    // do something here
  }
}
```

Thread 2:

```

synchronized (object2) {

  // do something here

  synchronized (object1) {
    // do something here
  }
}
```

Wait for Thread 2 to release the lock on object2

Wait for Thread 1 to release the lock on object1

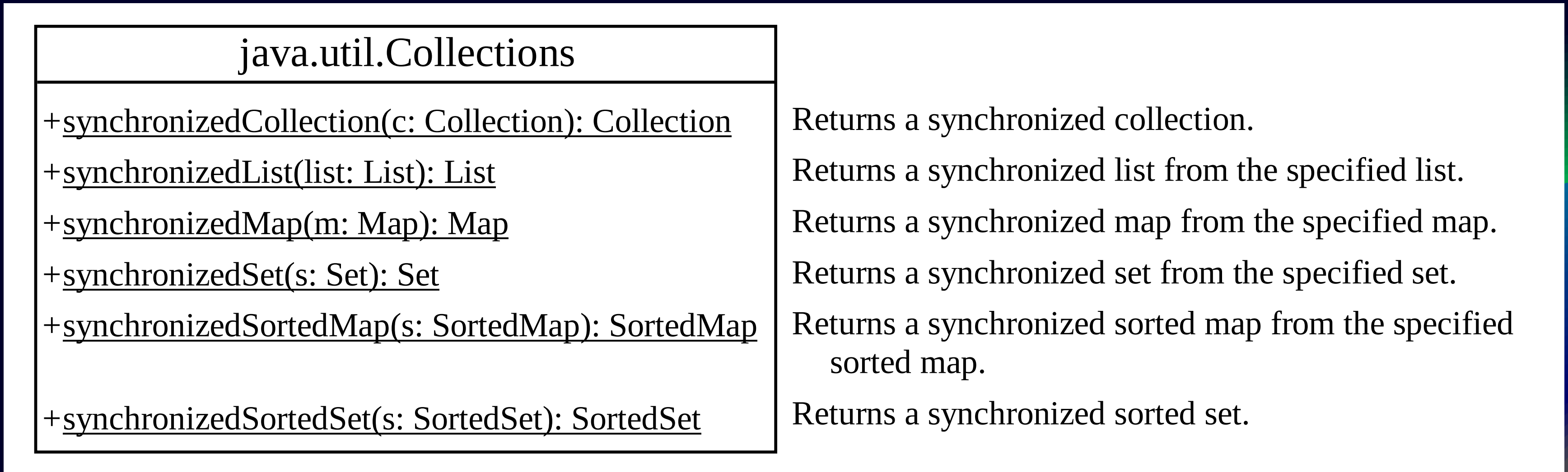
Preventing Deadlock

- Deadlock can be easily avoided by using a simple technique known as resource ordering.
- With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order.
- For the example in the previous figure, suppose the objects are ordered as object1 and object2.
- Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2. Once Thread 1 acquired a lock on object1, Thread 2 has to wait for a lock on object1.
- So Thread 1 will be able to acquire a lock on object2 and no deadlock would occur.

Synchronized Collections

The classes in the Java Collections Framework are not thread-safe, i.e., the contents may be corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or using synchronized collections.

The Collections class provides six static methods for wrapping a collection into a synchronized version. The collections created using these methods are called *synchronization wrappers*.



java.util.Collections

+synchronizedCollection(c: Collection): Collection
+synchronizedList(list: List): List
+synchronizedMap(m: Map): Map
+synchronizedSet(s: Set): Set
+synchronizedSortedMap(s: SortedMap): SortedMap
+synchronizedSortedSet(s: SortedSet): SortedSet

Returns a synchronized collection.
Returns a synchronized list from the specified list.
Returns a synchronized map from the specified map.
Returns a synchronized set from the specified set.
Returns a synchronized sorted map from the specified sorted map.
Returns a synchronized sorted set.

Vector, Stack, and Hashtable

Invoking synchronizedCollection(Collection c) returns a new Collection object, in which all the methods that access and update the original collection c are synchronized. These methods are implemented using the synchronized keyword. For example, the add method is implemented like this:

```
public boolean add(E o) {
    synchronized (this) {
      return c.add(o);
    }
}
```

The synchronized collections can be safely accessed and modified by multiple threads concurrently.

The methods in java.util.Vector, java.util.Stack, and Hashtable are already synchronized. These are old classes introduced in JDK 1.0. Starting with JDK 1.5, you should use java.util.ArrayList to replace Vector, java.util.LinkedList to replace Stack, and java.util.Map to replace Hashtable. If synchronization is needed, use a synchronization wrapper.

Programming Challenge

Write a program that launches 1,000 threads. Each thread adds **1** to a variable **sum** that initially is **0**. Define an **int** variable to hold **sum**.

Run the program with and without synchronization to see its effect.