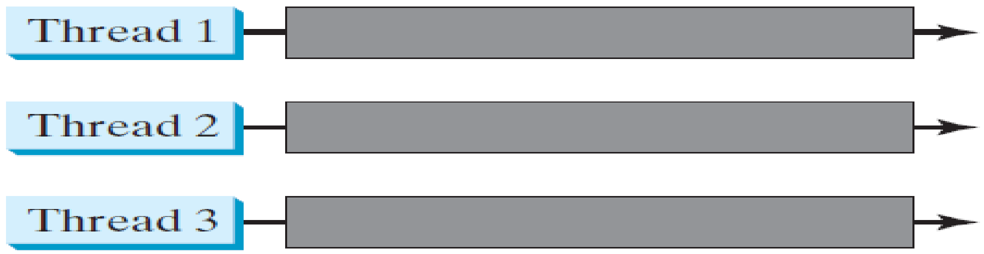
Chapter 30 - Multithreading

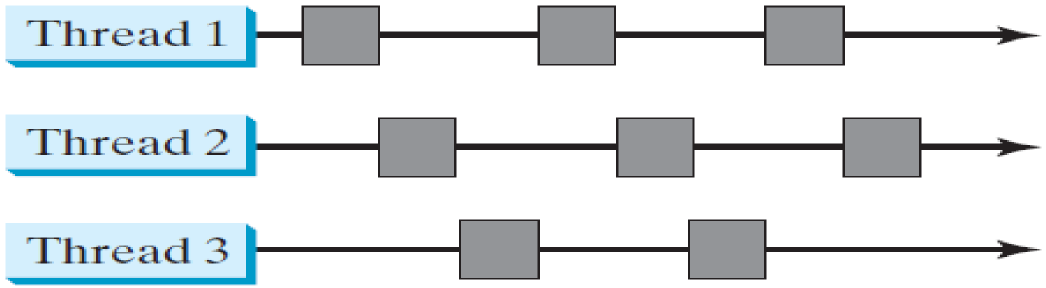
Multithreading enables multiple tasks in a program to be executed concurrently. A program consists of many tasks that can run concurrently A thread is the flow of execution, from beginning to end, of a task.

In single-processor systems, the multiple threads share CPU time, known as *time sharing*, and the operating system is responsible for scheduling and allocating resources to them. This arrangement is practical because most of the time the CPU is idle. It does nothing, for example, while waiting for the user to enter data.

Multiple threads on multiple CPUs



Multiple threads sharing a single CPU



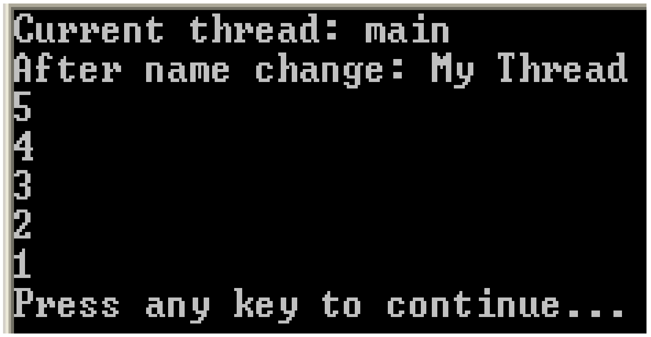
Main Thread

```
// Ex. 1: Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();
System.out.println("Current thread: " + t.getName());
// change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t.getName());
try {
      for(int n = 5; n > 0; n--) {
              System.out.println(n);
              t.sleep(1000);      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }  }}
```

دالة تعيد مرجعا للمسلك الذي استدعت فيه ولابد أن يكون المسلك موجودا

للحصول على اسم المسلك

لتغيير اسم المسلك

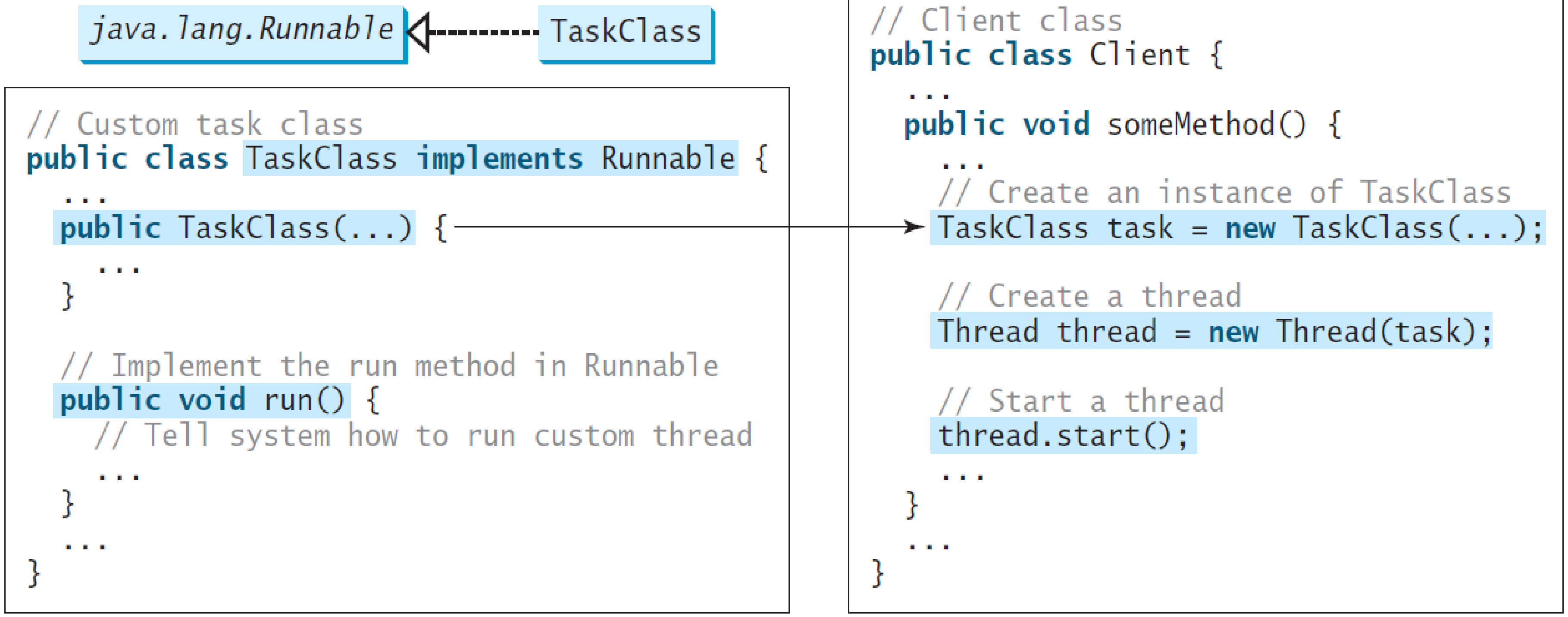


```
Current thread: main
After name change: My Thread
5
4
3
2
1
Press any key to continue...
```

Creating Tasks and Threads

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.

A task class must implement the **Runnable** *interface. A task must be run from a thread.*



```
// 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
    ...
  }
  ...
}
```

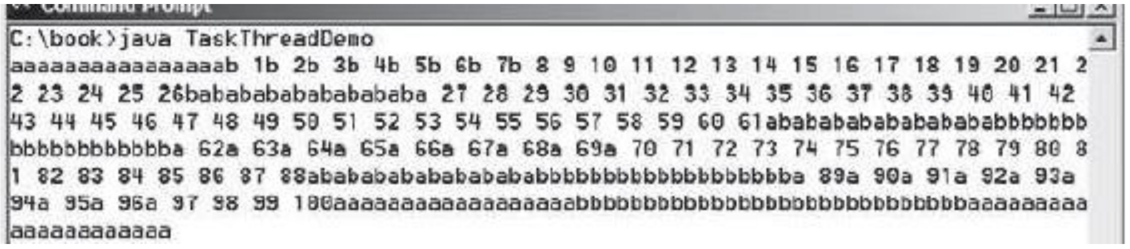
(a)

```
// 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();
    ...
  }
  ...
}
```

(b)

LISTING 30.1

- Objective: Create and run three threads:
  - The first thread prints the letter *a* 100 times.
  - The second thread prints the letter *b* 100 times.
  - The third thread prints the integers 1 through 100.



```
C:\book>java TaskThreadDemo
aaaaaaaaaaaaaaaab 1b 2b 3b 4b 5b 6b 7b 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
2 23 24 25 26babababababababa 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61abababababababababbbbbbb
bbbbbbbbbbbba 62a 63a 64a 65a 66a 67a 68a 69a 70 71 72 73 74 75 76 77 78 79 80 8
1 82 83 84 85 86 87 88abababababababababbbbbbbbbbbbbbbbbba 89a 90a 91a 92a 93a
94a 95a 96a 97 98 99 100aaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaa
aaaaaaaaaaaa
```

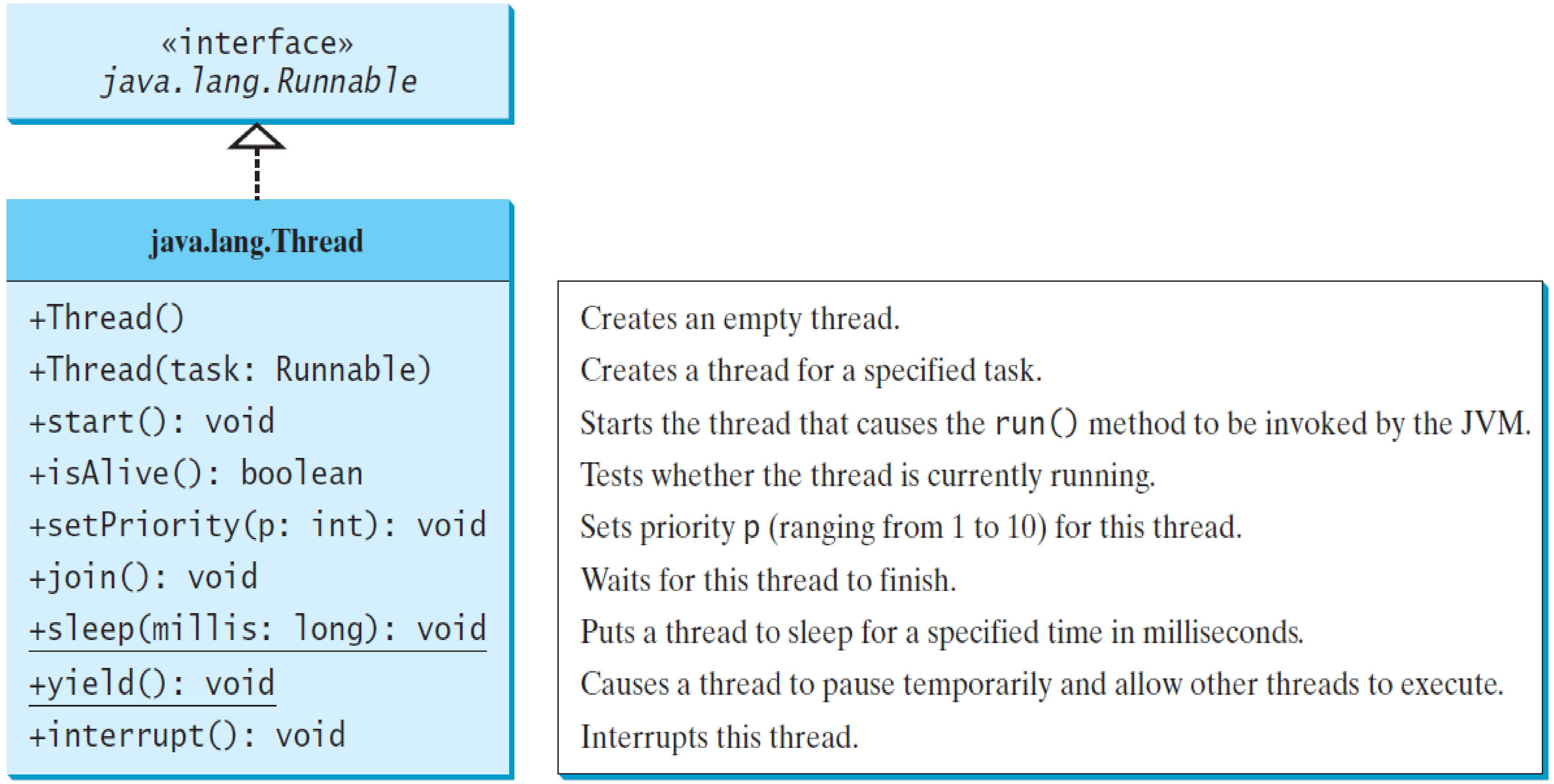
```java
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // 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();
  }
}
```

LISTING 30.1

```java

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

  /** Construct a task with a 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 task to perform
   */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}

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

  /** Construct a task for printing 1, 2, ..., n */
  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

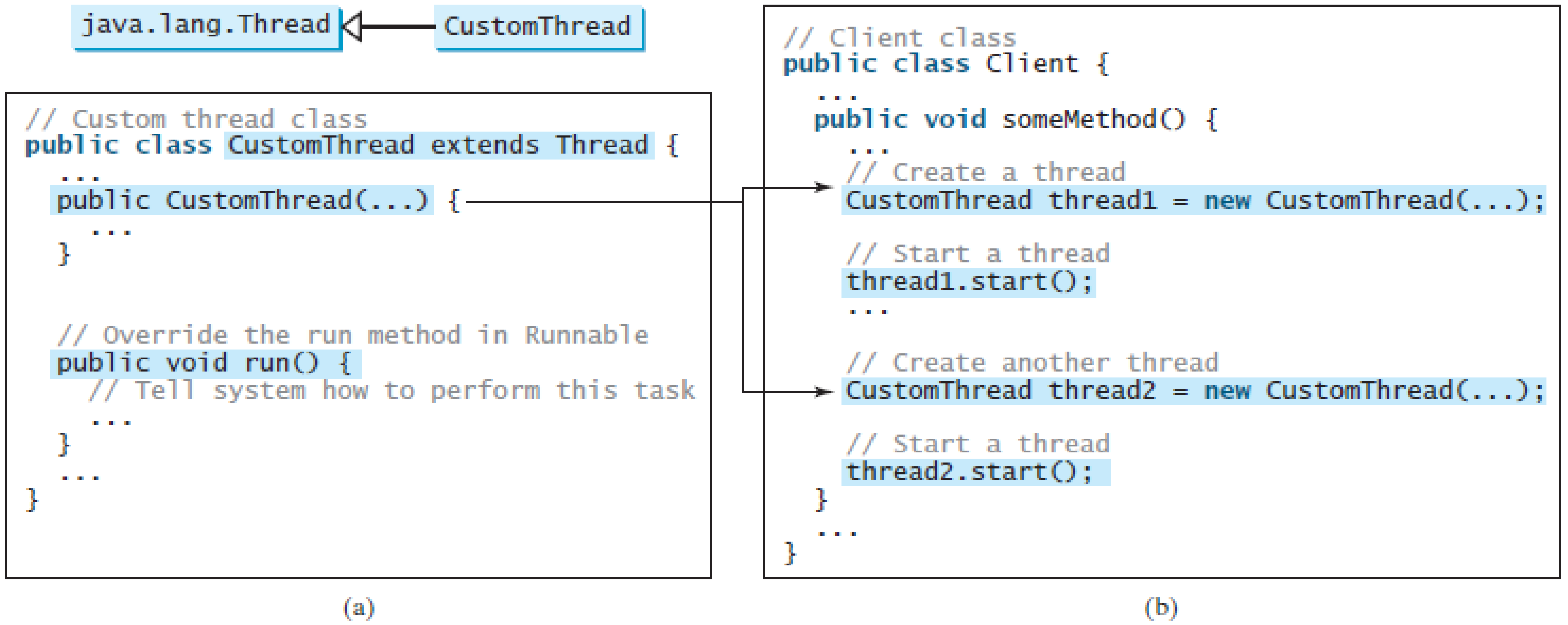
The **Thread** class contains the constructors for creating threads for tasks and the methods for controlling threads.



java.lang.Thread	
+Thread()	Creates an empty 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 a thread to sleep for a specified time in milliseconds.
+yield(): void	Causes a thread to pause temporarily and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

The Thread Class

Since the **Thread** class implements **Runnable**, you could define a class that extends **Thread** and implements the **run** method, as shown in Figure 30.5a, and then create an object from the class and invoke its **start** method in a client program to start the thread



```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }

  // Override the run method in Runnable
  public void run() {
    // Tell system how to perform this task
    ...
  }
  ...
}
```

```
// Client class
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 a thread
    thread2.start();
  }
  ...
}
```

FIGURE 30.5 Define a thread class by extending the Thread class.

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) {
    }
  }
}
```

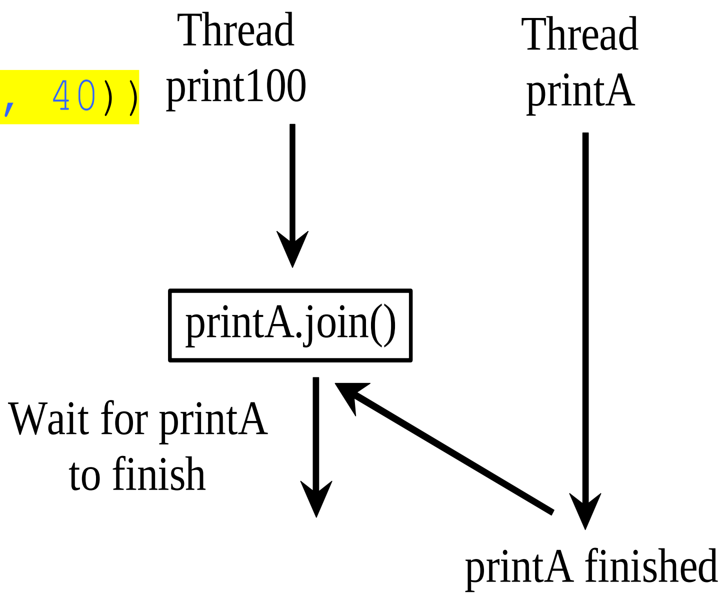
Every time a number (>= 50) is printed, the print100 thread is put to sleep for 1 millisecond.

The **sleep** method may throw an **InterruptedException**, which is a checked exception.

The join() 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 thread4 = new Thread(new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



☞A new **thread4** is created and it prints character *c* 40 times. The numbers from **50** to **100** are printed after thread **thread4** is finished.

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

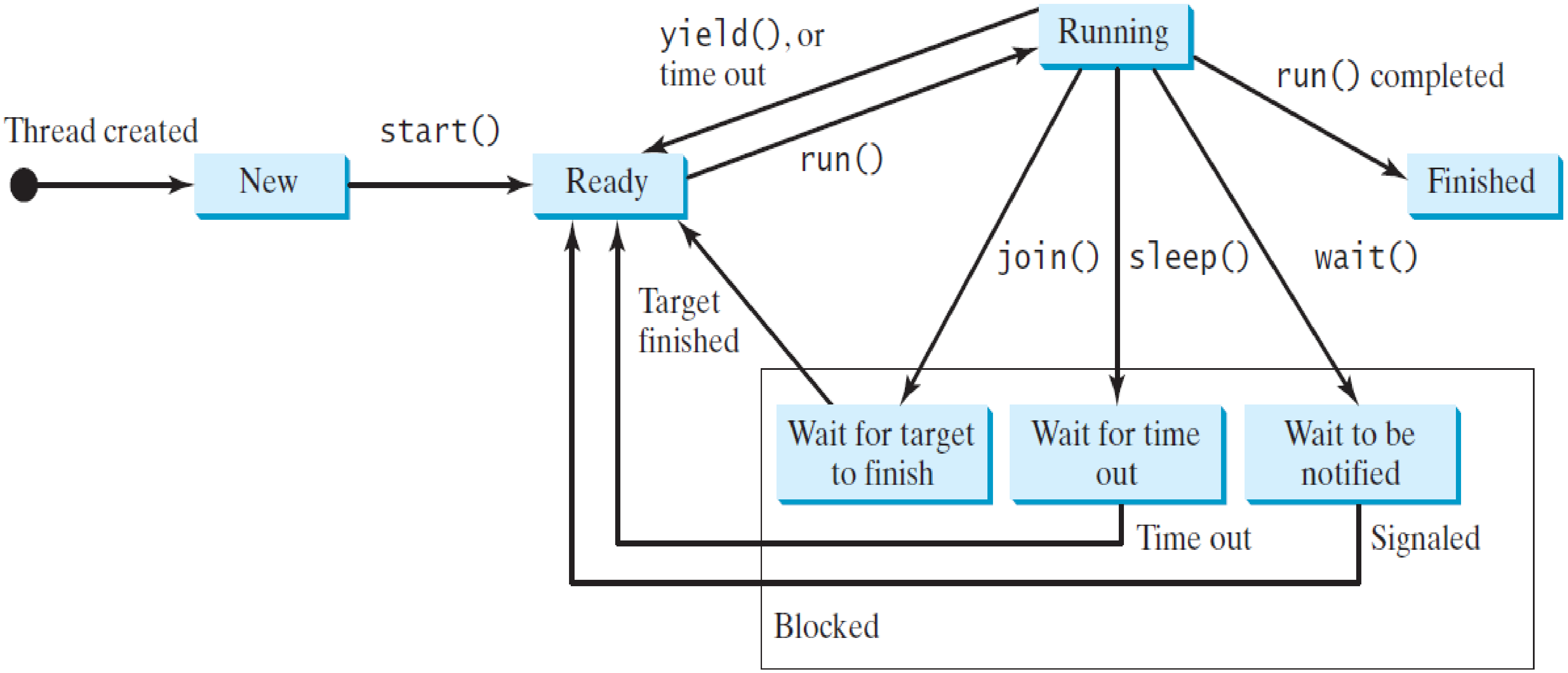
The **isAlive**() method is used to find out the state of a thread. It returns true if a thread is in the Ready, Blocked, or Running state; it returns false if a thread is new and has not started or if it is finished.

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 **isInterrupt**() method tests whether the thread is interrupted.

Thread States

☞A thread can be in one of five states: New, Ready, Running, Blocked, or Finished. After a thread is started by calling its **start()** method, it enters the ***Ready*** *state*. A ready thread is runnable but may not be running yet. The operating system has to allocate CPU time to it.



Thread Priority

- Java assigns every thread a priority. You can increase or decrease the priority of any thread by using **setPriority** `(int priority)` method, and you can get the thread's priority by using **getPriority()** method.
- Priorities are numbers ranging from **1** to **10**. Each thread is assigned a default priority of **`Thread.NORM_PRIORITY`** that is 5.
- Some constants for priorities include `Thread.MIN_PRIORITY` that is 1 `Thread.MAX_PRIORITY` that is 10.
- JVM always selects the currently runnable thread with the highest priority. A lower priority thread can run only when no higher-priority threads are running. If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue. This is called *round-robin scheduling*.
- Suppose you insert the following code in line 16 in Listing 30.1:
  - thread3.setPriority(Thread.MAX_PRIORITY);
- The thread for the **print100** task will be finished first.

Thread Pools

☞ Starting a new thread for each task is convenient for a single task execution, but it is not efficient for a large number of tasks because you have to create a thread for each task. This could limit throughput and cause poor performance.

☞ A **thread pool** is ideal to manage the number of tasks executing concurrently. JDK 1.5 uses **ExecutorService** interface for managing and controlling tasks.

☞ The **newFixedThreadPool(int)** method creates a fixed number of threads in a pool. If a thread completes executing a task, it can be reused to execute another task.

☞ **ExecutorService executor = Executors.newFixedThreadPool(3);**

☞ The **newCachedThreadPool()** method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution. A thread in a cached pool will be terminated if it has not been used for 60 seconds. A cached pool is efficient for many short tasks.

☞ **ExecutorService executor = Executors.newCachedThreadPool();**

Creating Executors

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

java.util.concurrent.Executors	
+newFixedThreadPool(numberOfThreads: int): 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.
+newCachedThreadPool(): ExecutorService	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

LISTING 30.3 ExecutorDemo.java

```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();
  }
}
```

Creating Executors

☞ Suppose that you replace line 6 with:

ExecutorService executor = Executors.newFixedThreadPool(1);

What will happen? The three runnable tasks will be executed sequentially because there is only one thread in the pool.

☞ Suppose you replace line 6 with

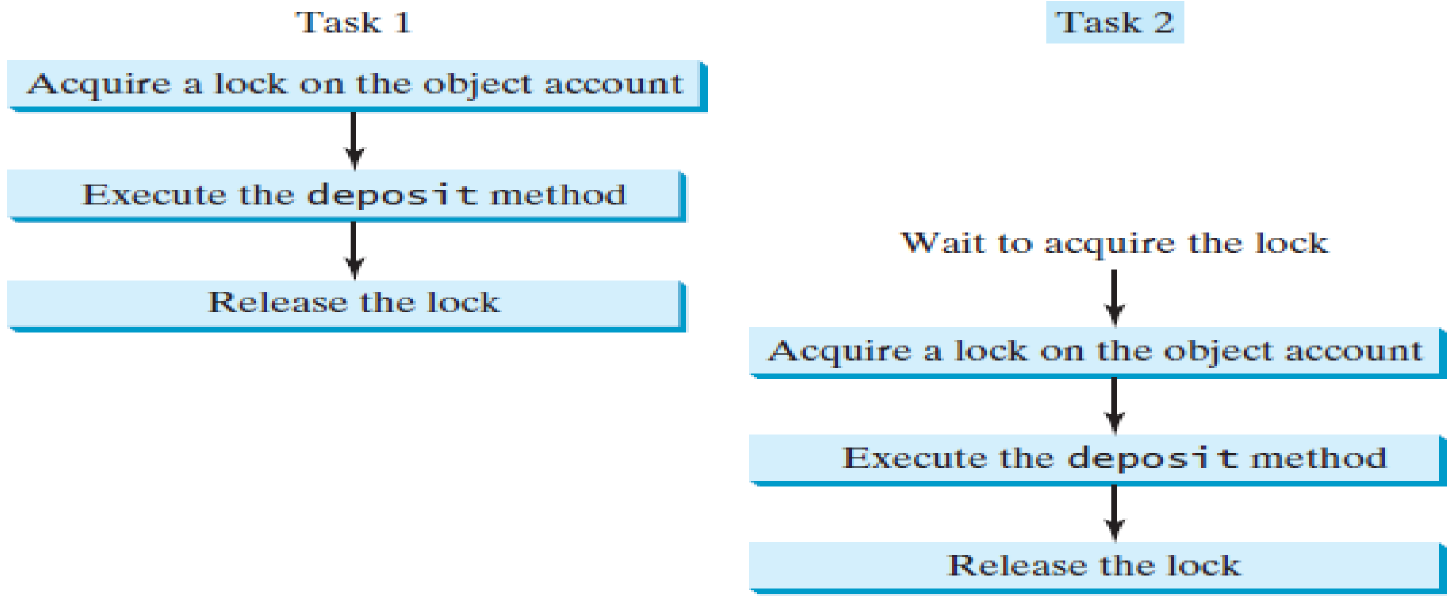
ExecutorService executor = Executors.newCachedThreadPool();

What will happen? New threads will be created for each waiting task, so all the tasks will be executed concurrently.

☞ The **shutdown()** method in line 14 tells the executor to shut down. No new tasks can be accepted, but any existing tasks will continue to finish.

Thread Synchronization

- Thread synchronization is to coordinate the execution of the dependent threads. A shared resource may become corrupted if it is accessed simultaneously by multiple threads.
- You can use the **synchronized** keyword to synchronize the method so that only one thread can access the method at a time. For example:
- public synchronized void deposit(double amount)
- 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.


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

Example: Use Synchronized method

```
class SumArray {
 int sum;
 synchronized int sumArray(int nums[]) {
   sum = 0;                                   // reset sum
  for(int i=0; i<nums.length; i++) {
     sum += nums[i];
     System.out.println("Running total for " +
          Thread.currentThread().getName() + " is " + sum);
  try {      Thread.sleep(10);      }          // allow task-switch
      catch(InterruptedException exc)
               { System.out.println("interrupted"); }
        }                                      // End For
   return sum;
  } }
class MyThread extends Thread{
  SumArray sa = new SumArray();
  int a[];
  int answer;
```

Example Cont.

```
MyThread (String name, int nums[]) {      // Construct a new thread.
    super (name);
    a = nums;
    start();                                          // start the thread
}
 public void run() {                                   // Begin execution of new thread
 int sum;
 System.out.println( getName() + " starting.");
 answer = sa.sumArray (a);
 System.out.println("Sum for " + getName() +" is " + answer);
 System.out.println( getName() + " terminating.");  }}
class Sync {
  public static void main(String args[]) {
    int a[] = {1, 2, 3, 4, 5};
    int b[] = {10, 20, 30, 40, 50};
    MyThread t1 = new MyThread ("Child #1", a);
    MyThread t2 = new MyThread ("Child #2", b);
  }  }
```

Example Output

Synchronized

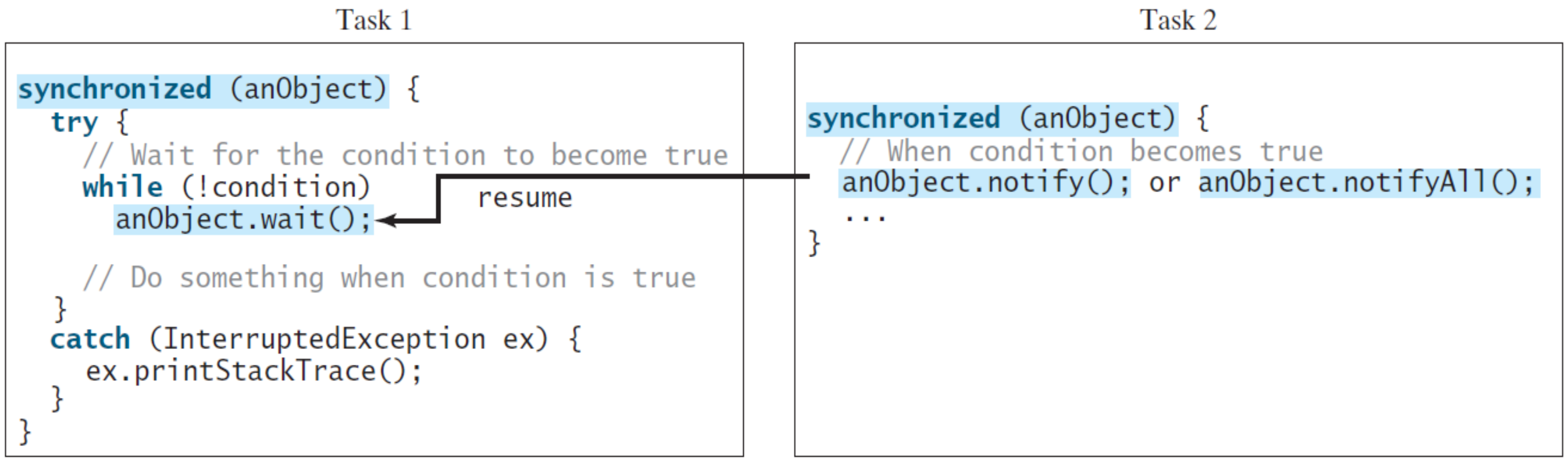
```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Running total for Child #2 is 10
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 30
Running total for Child #2 is 60
Running total for Child #2 is 100
Running total for Child #2 is 150
Sum for Child #2 is 150
Child #2 terminating.
```

Without Synchronized

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #2 is 10
Running total for Child #1 is 12
Running total for Child #2 is 32
Running total for Child #1 is 35
Running total for Child #2 is 65
Running total for Child #1 is 69
Running total for Child #2 is 109
Running total for Child #1 is 114
Running total for Child #2 is 164
Sum for Child #1 is 164
Child #1 terminating.
Sum for Child #2 is 164
Child #2 terminating.
```

wait(), notify(), and notifyAll()

- Use the **wait**(), **notify**(), and **notifyAll**() methods to facilitate communication among threads.
- The **wait**(), **notify**(), and **notifyAll**() methods must be called in a synchronized method or a synchronized block on the calling object of these methods.
- The **wait**() method lets the thread wait until some condition occurs. When it occurs, you can use the **notify**() or **notifyAll**() methods to notify the waiting threads to resume normal execution.
- The **notifyAll**() method wakes up all waiting threads, while **notify**() picks up only one thread from a waiting queue.



```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

    // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or anObject.notifyAll();
  ...
}
```