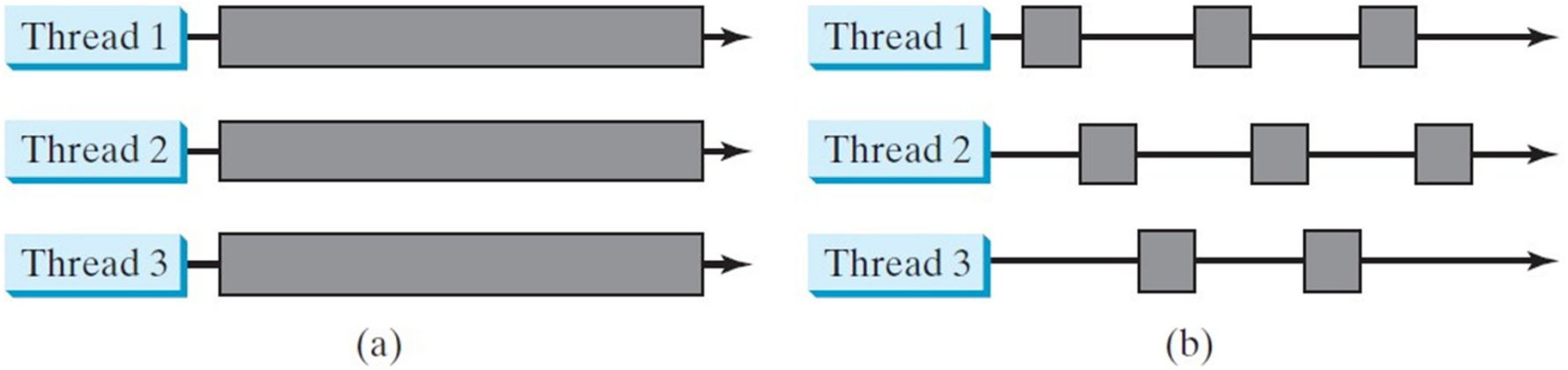
Introduction to Multithreading

Threads Concept

- 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.
- A *thread* provides the mechanism for running a task. With Java, you can launch multiple threads from a program concurrently.



Multiple threads on multiple CPUs

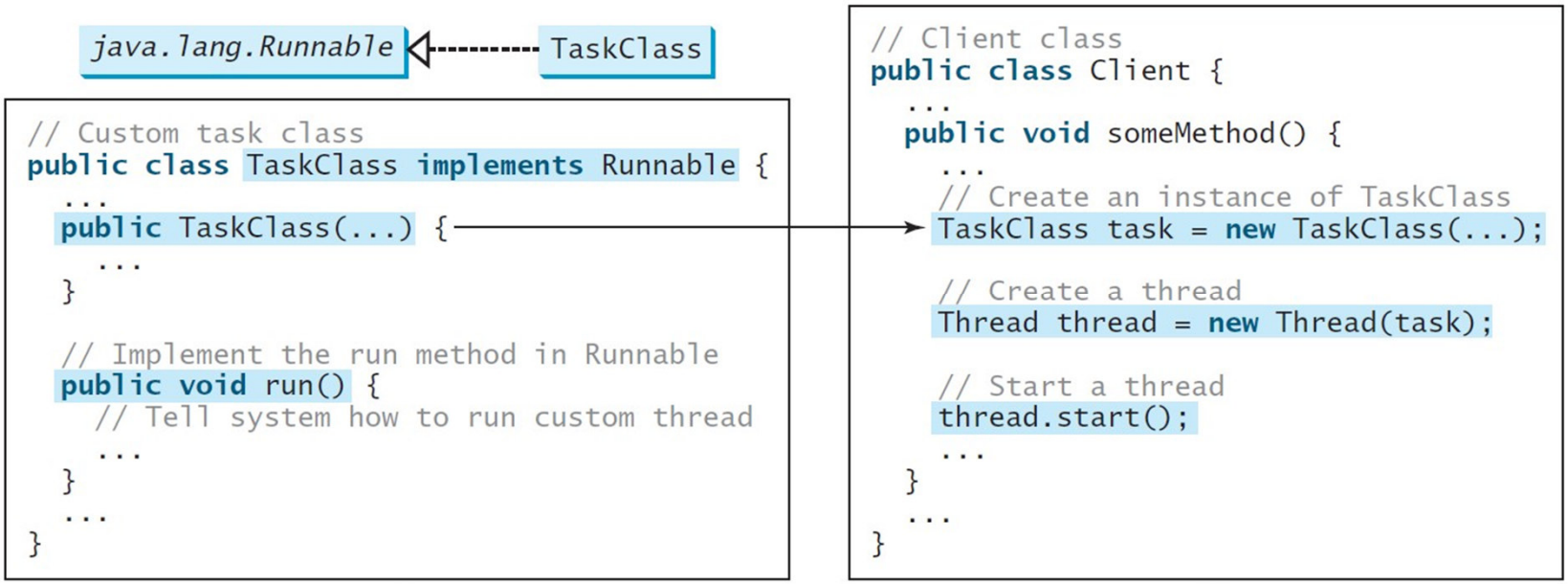
Multiple threads sharing a single CPU

Threads Concept

- 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 must be run from a thread

Creating Tasks and Threads

- Tasks are objects. To create tasks, you have to first define a class for tasks, which implements the `Runnable` interface.
- The `runnable` interface is rather simple. All it contains is the `run` method. You need to implement this method to tell the system how your thread is going to run.



Example: Using the Runnable Interface to Create and Launch Threads

- Gives a program that creates three tasks and three threads to run them.
  - The first prints the letter a 100 times.
  - The second task prints the letter b 100 times.
  - The third task prints the integers 1 through 100.
- When you run this program, the three threads will share the CPU and take turns printing letters and numbers on the console.

Example: Using the Runnable Interface to Create and Launch Threads

```
// The task for printing a character a specified number of times
public class PrintChar implements Runnable {

     private char charToPrint; // The time to print
     private int times; // The number of times to repeat

     /* Constructor
      * a task with a specified character and number of
      * times to print the character
      */
     public PrintChar(char c, int t) {
          this.charToPrint = c;
          this.times = t;
     }
     @Override
     /* Override the run() method to tell the system
      * what task to perform
      */
     public void run() {
          for (int i = 0; i < this.times; i++) {
               System.out.print(this.charToPrint);
     }
}
```

Example: Using the Runnable Interface to Create and Launch Threads

```
// The task class for printing numbers from 1 to n for a given n
public class PrintNum implements Runnable {

     private int lastNum;

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

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

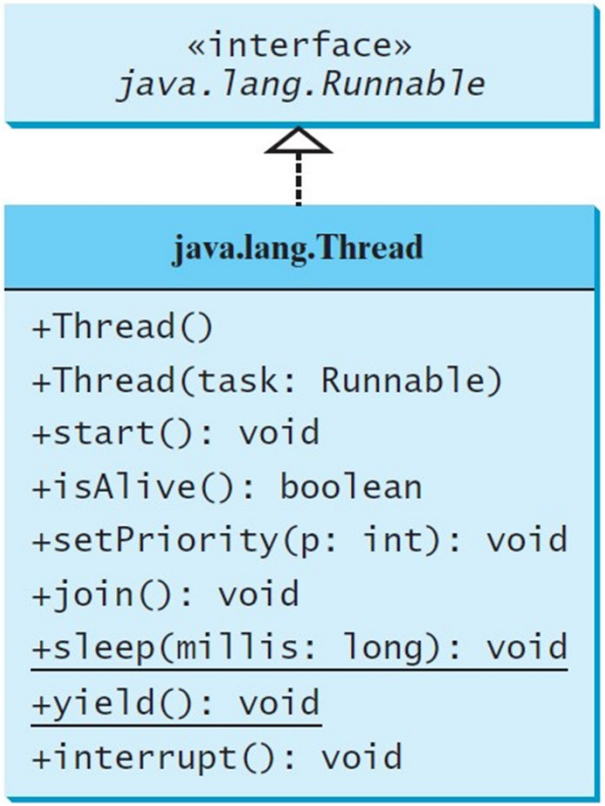
Example: Using the Runnable Interface to Create and Launch Threads

```
public class TaskThreadDemo {
    public static void main(String[] args) {

    // create tasks
    PrintChar printA = new PrintChar('a', 100);
    PrintChar printB = new PrintChar('b', 100);
    PrintNum 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();
    }
}
```

RUN

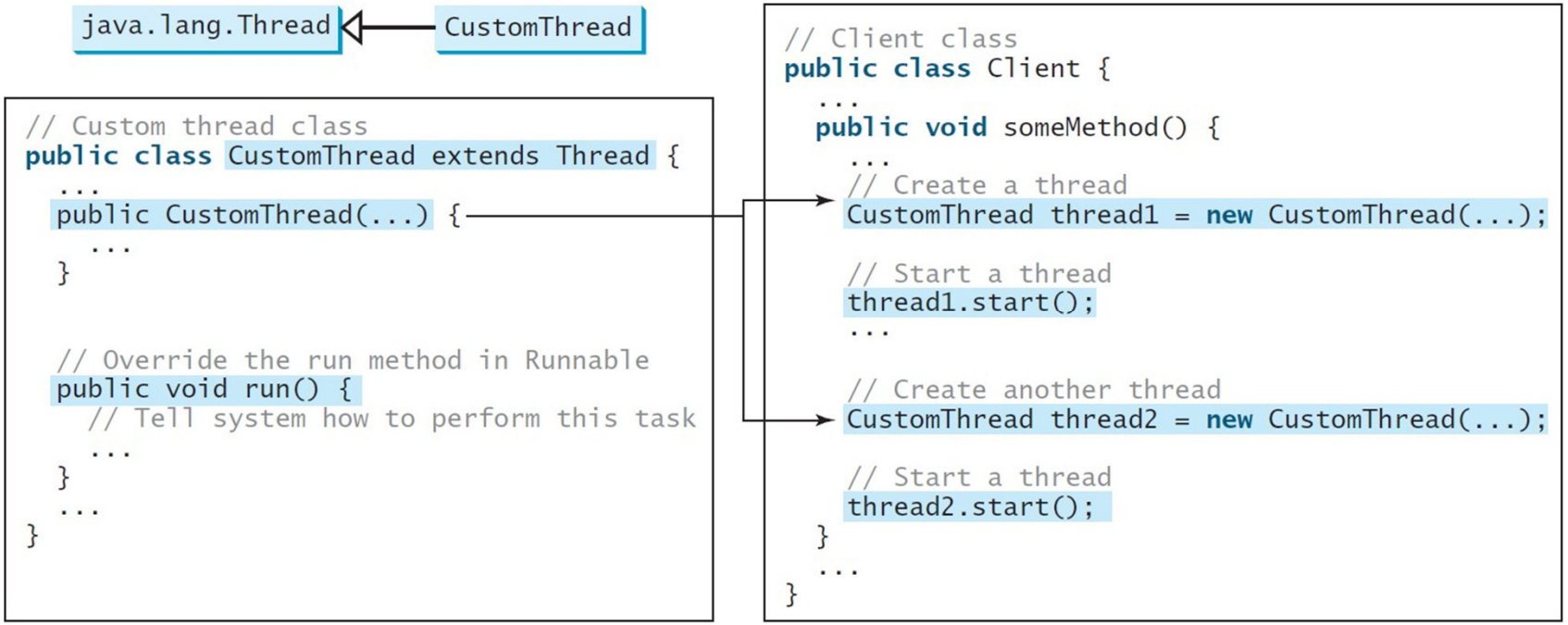
The Thread Class



Creates an empty thread.	
Creates a thread for a specified task.	
Starts the thread that causes the `run()` method to be invoked by the JVM.	
Tests whether the thread is currently running.	
Sets priority `p` (ranging from 1 to 10) for this thread.	
Waits for this thread to finish.	
Puts a thread to sleep for a specified time in milliseconds.	
Causes a thread to pause temporarily and allow other threads to execute.	
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, and then create an object from the class and invoke its `start()` method in a client program to start the 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 TaskThreadDemo.java as follows:

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

- Every time a number is printed, the print 100 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 TaskThreadDemo.java as follows:

```
public void run() {
    for (int i = 1; i <= this.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 1millisecond