Chapter 30 Multithreading and Parallel Programming

Objectives

- To get an overview of multithreading (§30.2).
- To develop task classes by implementing the **Runnable** interface (§30.3).
- To create threads to run tasks using the **Thread** class (§30.3).
- To control threads using the methods in the **Thread** class (§30.4).
- To control animations using threads and use **Platform.runLater** to run the code in application thread (§30.5).
- To execute tasks in a thread pool (§30.6).
- To use synchronized methods or blocks to synchronize threads to avoid race conditions (§30.7).
- To synchronize threads using locks (§30.8).
- To facilitate thread communications using conditions on locks (§§30.9–30.10).
- To restrict the number of accesses to a shared resource using semaphores (§30.12).
- To use the resource-ordering technique to avoid deadlocks (§30.13).
- To describe the life cycle of a thread (§30.14).

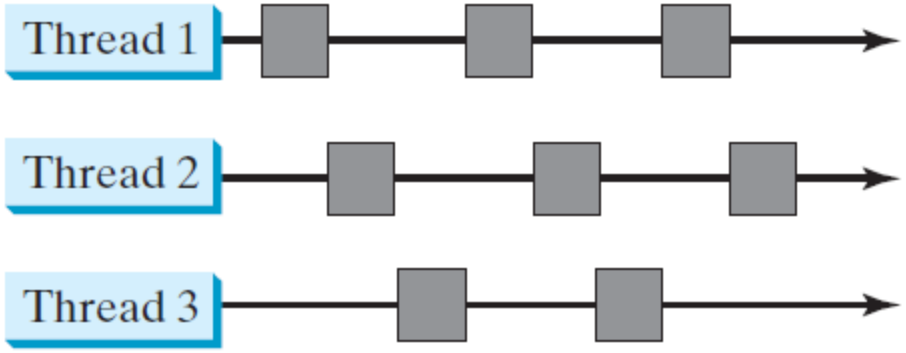
Thread Concepts

- 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.
- These threads can be executed simultaneously in multiprocessor systems

Threads Concept (continue)

Multiple threads on multiple CPUs

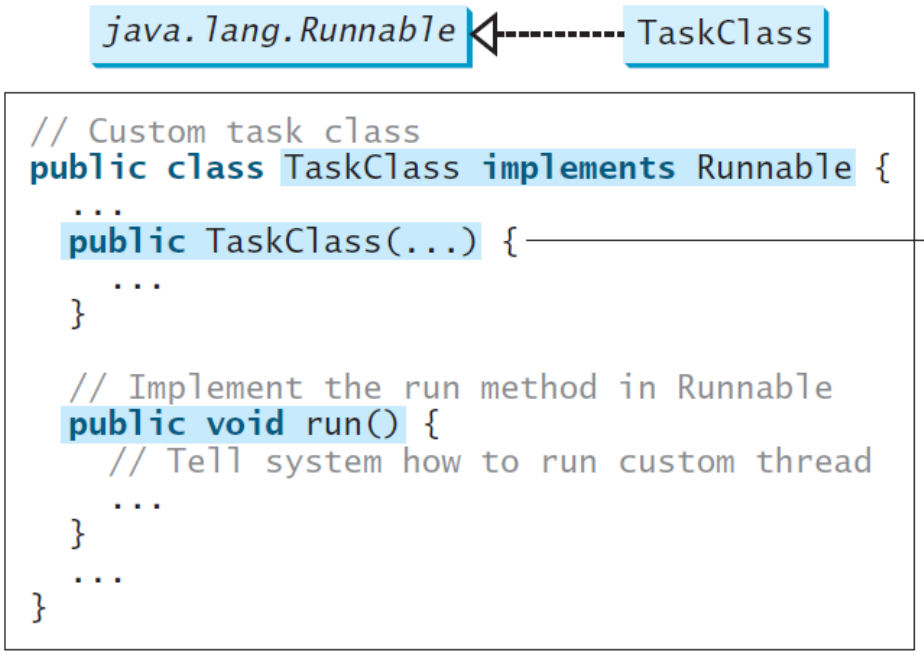
Multiple threads sharing a single CPU



Threads Concept (continue)

- 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
- Multithreading can make your program more responsive and interactive, as well as enhance performance
- 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

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
    ...
  }
  ...
}
```

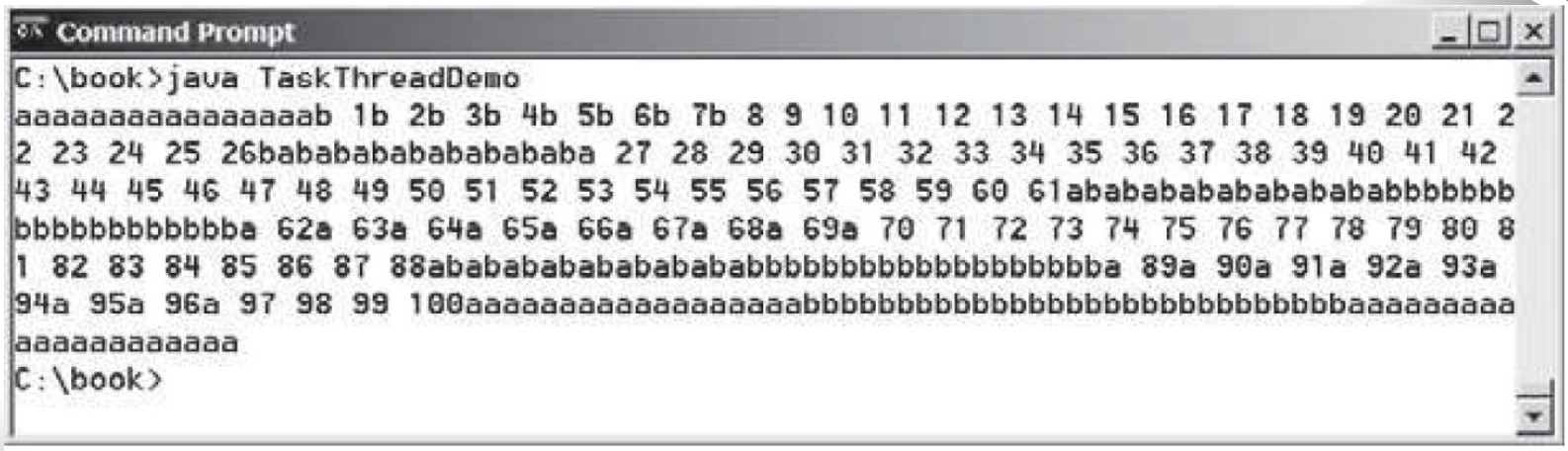
(a)

```
// 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)

Example: Using the `Runnable` Interface to Create and Launch Threads

- 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.



create tasks

create threads

start threads

task class

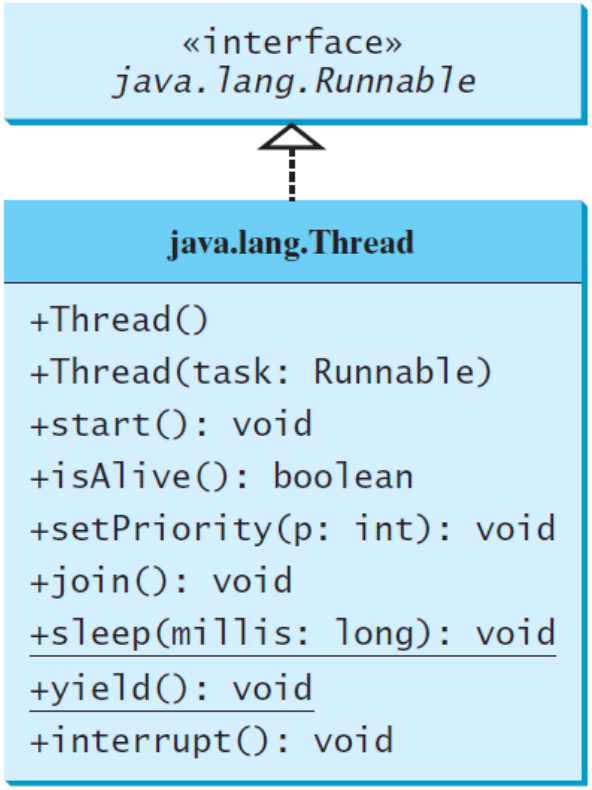
run

task class

run

```
1  public class TaskThreadDemo {
2    public static void main(String[] args) {
3      // Create tasks
4      Runnable printA = new PrintChar('a', 100);
5      Runnable printB = new PrintChar('b', 100);
6      Runnable print100 = new PrintNum(100);
7
8      // Create threads
9      Thread thread1 = new Thread(printA);
10     Thread thread2 = new Thread(printB);
11     Thread thread3 = new Thread(print100);
12
13     // Start threads
14     thread1.start();
15     thread2.start();
16     thread3.start();
17   }
18 }
19
20 // The task for printing a character a specified number of times
21 class PrintChar implements Runnable {
22   private char charToPrint; // The character to print
23   private int times; // The number of times to repeat
24
25   /** Construct a task with a specified character and number of
26    *  times to print the character
27    */
28   public PrintChar(char c, int t) {
29     charToPrint = c;
30     times = t;
31   }
32
33   @Override /** Override the run() method to tell the system
34    *  what task to perform
35    */
36   public void run() {
37     for (int i = 0; i < times; i++) {
38       System.out.print(charToPrint);
39     }
40   }
41 }
42
43 // The task class for printing numbers from 1 to n for a given n
44 class PrintNum implements Runnable {
45   private int lastNum;
46
47   /** Construct a task for printing 1, 2, ..., n */
48   public PrintNum(int n) {
49     lastNum = n;
50   }
51
52   @Override /** Tell the thread how to run */
53   public void run() {
54     for (int i = 1; i <= lastNum; i++) {
55       System.out.print(" " + i);
56     }
57   }
58 }
```

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.	

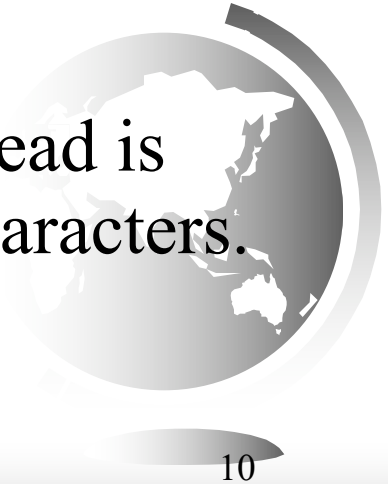
The *Thread* class contains the constructors for creating threads for tasks and the methods for controlling threads.

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:

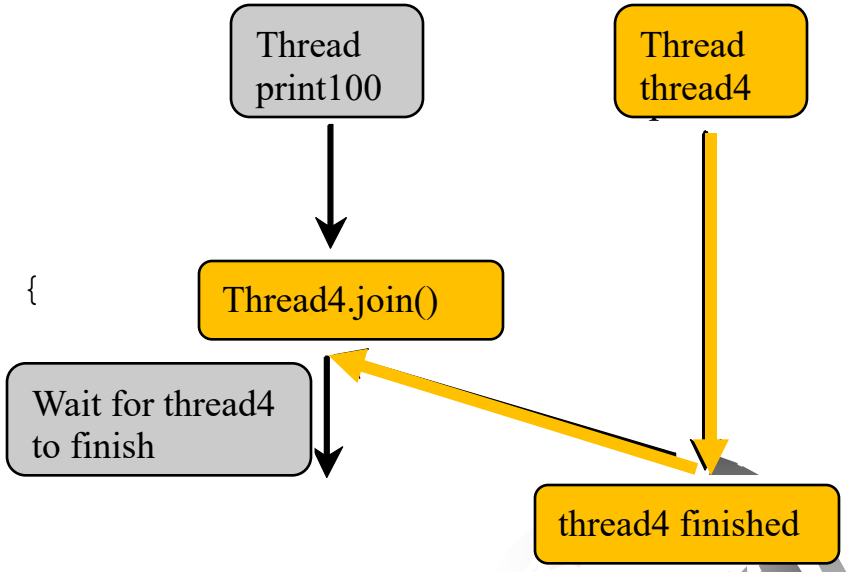
```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1000);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

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

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) {
  }
```



The numbers after 50 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.

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

- Priorities are numbers ranging from 1 to 10. The Thread class has the int constants `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`, representing 1, 5, and 10, respectively.
- 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`