

Chapter 3

Java Multithread Programming

Introduction: It is common for personal computers to perform many tasks at a time; for instance, compile a program, send a file to a printer and receive electronic mail messages over a network concurrently. How does computer do everything at once?

- ⌘ Multitasking
- ⌘ Multiprocessing

Multitasking (Time Sharing): Operating systems on single-processor computers create the illusion of concurrent execution by rapidly switching between activities(tasks).

- ⌘ Tasks managed by the operating system (scheduler)
- ⌘ Computer seems to work on tasks concurrently.
- ⌘ Can improve performance by reducing waiting.

Multiprocessing: Only computers that have multiple processors can truly execute multiple instructions concurrently. But, on such computers only a single instruction can execute at once.

- ⌘ Computer works on several tasks in parallel
- ⌘ Performance can be improved.

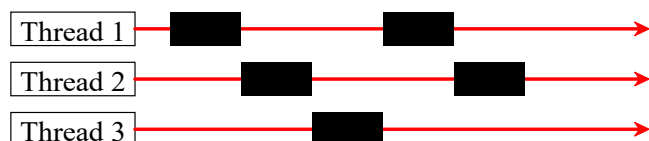
Multithreading: A multithreaded program contains two or more parts(tasks) that can run concurrently. A **task** is a program unit that is executed independently of other parts of the program. A thread provides the mechanism for a running a task.(i.e. each part of such a program is called a thread, and each thread defines a separate path of execution). With java, you can launch multiple threads from a program concurrently. These threads can be executed simultaneously in multiprocessor systems. In Single-processor systems, the multiple threads share CPU time, and the operating system is responsible for scheduling and allocating resources to them.

Threads Concept in Diagram:

Multiple threads on
multiple CPUs



Multiple threads sharing
a single CPU



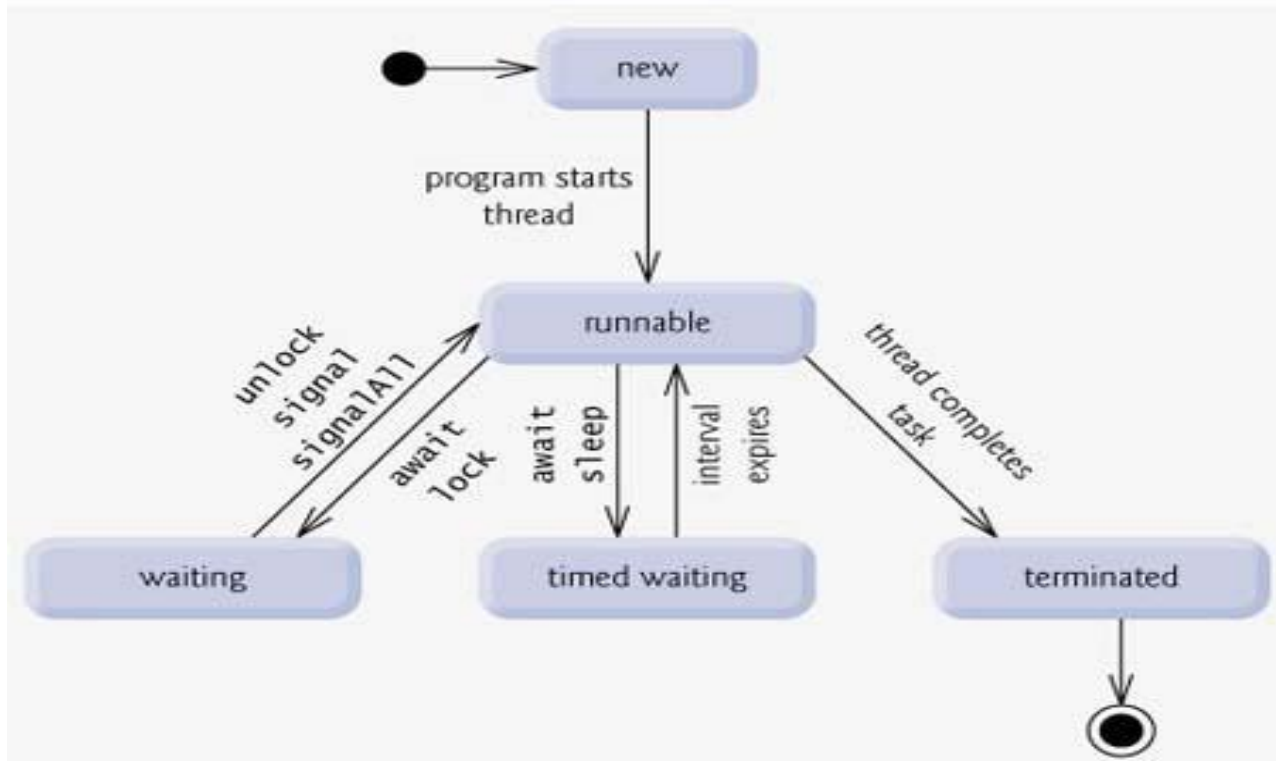
Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. For example, when downloading a large file (e.g., an image, an audio clip or a video clip) over the internet, the user may not want to wait until the entire clip downloads before starting the playback. To solve this problem, we can put multiple threads to work—one to download the clip, and another to play it. A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.

Java provides exceptionally good support for creating and running threads, and for lacking resources to prevent conflicts. 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.

Process vs. Thread: A program or process consists of the memory space allocated by the operating system that can contain one or more threads. It runs independently and isolated of other processes. It cannot directly access shared data in other processes. A thread is a lightweight process which cannot exist on its own; it must be a part of a process.

A process remains running until all the non-daemon threads are done executing. Unlike processes, threads share the same address space; that is, they can read and write the same variables and data structures. Like multiple processes that can run on one computer, multiple threads appear to be doing their work in parallel. Implemented on a multi-processor machine, they can work in parallel.

Thread states: Life Cycle of a Thread: A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Here are described the possible thread life cycles diagrammatically.



New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting: Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated: A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Creating a Thread: In java, there are two alternative ways to set up your own threads. These are implementing the Runnable interface or extending the Thread class itself.

Creating a threads by implementing Runnable Interface : The preferred means of creating multithreaded Java applications is by implementing the Runnable interface (of package java.lang). A Runnable object represents a “task” that can execute concurrently with other tasks.

To create tasks (runnable objects), you must declare a class for tasks. A task class must implement the Runnable interface.

To implement Runnable, a class need only implement the single method **run()**, which is declared like this: public void run(). It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can do. The **run()** method contains the code that defines the task that a **Runnable** object should perform. When a thread executing a **Runnable** is created and started, the thread calls the **Runnable** object's run() method, which executes in the new thread.

Here is a general structure to create a 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  
        ...  
    }  
    ...  
}
```

Once you have declared a TaskClass, you can create a task using its constructor.

```
TaskClass task = new TaskClass(...);
```

A task must be executed in a thread. The Thread class contains different constructors for creating threads and many useful methods for controlling threads. To create a thread for a task.

```
Thread thread = new Thread(task);
```

After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. You can then invoke the start() method to tell the JVM that the thread is ready to run, as follows:

```
thread.start();
```

The JVM will execute the task by invoking the task's run() method.

Some of the constructors of the Thread class:

- Thread() → This constructor has the same effect as Thread(null, null, gname), where gname is a newly generated name. Automatically generated names are of the form “Thread-”+n, where n is an integer.
- Thread(Runnable target) → This constructor has the same effect as Thread(null, target, gname), where gname is a newly generated name. Automatically generated names are of the form “Thread-”+n, where n is an integer. The parameter “target” is the object whose run method is invoked when this thread is started. If null, this classes run method does nothing.
- Thread(Runnable target, String name) → This constructor has the same effect as Thread(null, target, name). The parameter “target” is the object whose run method is invoked when this thread is started. If null, this classes run method does nothing whereas name refers the name of the new thread.
- Thread (String name) → This constructor has the same effect as Thread(null, null, name). The parameter name refers the name of the new thread.

Below is an example of instantiating task class and associating threads for the task instance.

```
// Client class
public class Client {
    ...
    public static void main(String[] args){
        ...
        // Create an instance of TaskClass
        TaskClass task = new TaskClass(...);
        // Create a thread
        Thread thread = new Thread(task);
        // Start a thread
        thread.start();
        ...
    }
    ...
}
```

Example 1: Write a program that create three tasks and three threads to run them:

- ∞ 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**.

// 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;

}

public void run() {

for (int i = 0; i < times; i++) {

System.out.print(charToPrint);

}

}

}

// The task class for printing number from 1 to n for a given n

class PrintNum implements Runnable {

private int lastNum;

public PrintNum(int n) { //Construct a task for printing 1, 2, ... I

lastNum = n;

}

public void run() { /** Tell the thread how to run */

for (int i = 1; i <= lastNum; i++) {

System.out.print(" " + i);

}

}

}

The output would be something like

Note that different outputs will be displayed on different runs

Below is a general structure to create a task instance (that extends Thread class)

Advanced [Java] Programming Chapter 3: Java Multithread Programming By: Eyuel H. Page 7 of 16

```

public TaskClass(...) {
    ...
}

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

```

Here is an example code on how to instantiate tasks and how to start executing the tasks

```

// Client class
public class Client {
    ...
    public static void main(String[] args){
        ...
        // Create a thread
        TaskClass thread = new TaskClass();
        // Start a thread
        thread.start();
        ...
    }
    ...
}

```

Example 1:

```

class MyThread extends Thread {
    int id;
    public MyThread( int ident) {
        id = ident;
    }
}

```



```

        public void run() {
            for(int i=0; i<4; ++i)
                System.out.println("Hello from thread " + id);
        }
    }

    public class MyThreadApplication {
        public static void main( String argv[] ) {
            MyThread myThread1, myThread2;
            myThread1 = new MyThread( 1);
            myThread2 = new MyThread( 2);
            myThread1.start();
            myThread2.start();
            System.out.println("Hello from main thread ");
        }
    }

```

Output

```

Hello from main thread
Hello from thread 1
Hello from thread 2
Hello from thread 1
Hello from thread 2
Hello from thread 2
Hello from thread 2
Hello from thread 1
Hello from thread 1

```

Thread Priorities: Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Thread scheduling is determining which thread runs next. Java priorities are in the range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10). By default, every thread is given priority `NORM_PRIORITY` (a constant of 5). Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Some Thread Methods:

- **public void start():** Starts the thread in a separate path of execution, then invokes the `run()` method on this Thread object.
- ∞ **public void run():** If this Thread object was instantiated using a separate Runnable target, the `run()` method is invoked on that Runnable object.
- ∞ **public final void setName(String name) :**Changes the name of the Thread object. There is also a `getName()` method for retrieving the name.

- ⌘ **public final void setPriority(int priority):** Sets the priority of this Thread object. The possible values are between 1 and 10.
- ⌘ **public void interrupt():** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
- ⌘ **public final boolean isAlive():** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.
- ⌘ **public static boolean holdsLock(Object x):** Returns true if the current thread holds the lock on the given Object.
- ⌘ **public static Thread currentThread():** Returns a reference to the currently running thread, which is the thread that invokes this method.
- ⌘ **public final void setDaemon(boolean on):** A parameter of true denotes this Thread as a daemon thread.
- ⌘ **public static void yield():** To temporarily release time for other threads(i.e. Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.)

For example, suppose you modify the code PrintNum of one of the examples we saw previously.

```
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.

- **public static void sleep(long millisec):** puts the thread to sleep for the specified time in milliseconds to allow the other threads to execute.

For example, suppose you modify the above code of PrintNum

```
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.

Note that the `sleep()` method may throw an `InterruptedException`, which is a checked exception. Such an exception may occur when a sleeping thread's `interrupt()` method is called. The `interrupt()` method is very rarely invoked on a thread, so an `InterruptedException` is unlikely to occur. But since Java forces you to catch checked exceptions, you have to put it in a try-catch block. If a sleep method is invoked in a loop, you should wrap the loop in a try-catch block. Otherwise the thread may continue to execute even though it is being interrupted.

- **public final void join(long millisec):** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.

For example, suppose you modify the above code of `PrintNum`

```

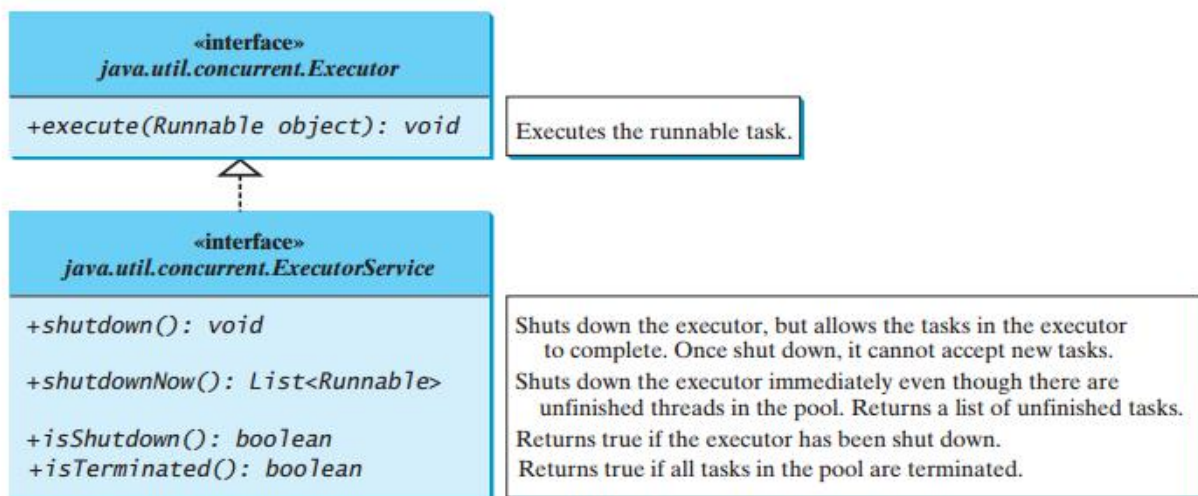
public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        try {
            if (i == 50) thread1.join();
        }
        catch (InterruptedException ex) {
        }
    }
}

```

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

Thread Pool: The approach of creating and running multithreading applications that we have seen so far is convenient for a single task execution. But it is not efficient for a large number of tasks, because we have to create a thread for each task. Starting a new thread for each task could limit throughput and cause poor performance. So, another approach called Thread Pool emerged to solve those challenges.

A thread pool is ideal to manage a number of tasks executing concurrently. Java provides 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**. Note that the **Executor** and **ExecutorService** interfaces of the JDK API are located in the package **java.util.concurrent**.



To create an **Executor** object, the static methods in the **Executors** class are used. The **Executors** class of the JDK API is found under the package **java.util.concurrent**. The following two static methods of **Executors** can be used to create thread pools.

- ⌘ The **newFixedThreadPool(int)**: It is a method that creates a fixed number of threads in a pool. If a thread completes executing a task, it can be reused to execute another task. If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it if all the threads in the pool are not idle and there are tasks waiting for execution
- ⌘ The **newCachedThreadPool()**: It is a method that 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

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.

Example:

```
1 import java.util.concurrent.*;
2
3 public class ExecutorDemo {
4     public static void main(String[] args) {
5         // Create a fixed thread pool with maximum three threads
6         ExecutorService executor = Executors.newFixedThreadPool(3);
7
8         // Submit runnable tasks to the executor
9         executor.execute(new PrintChar('a', 100));
10        executor.execute(new PrintChar('b', 100));
11        executor.execute(new PrintNum(100));
12
13        // Shut down the executor
14        executor.shutdown();
15    }
16 }
```

Exercise 1: Develop a Thread pool based java multithread application (with 3 threads) that runs the following four tasks concurrently. Task 1: prints numbers from 5 to 5000; Task 2: prints numbers from 500 to 50,000; Task 3: prints the English letter 'H' 100 times; Task 4: prints the English letter 'B' 2000 times.

Exercise 2: Develop a Thread pool based java multithread application (without specifying the number of threads to use for the application) that runs the following four tasks concurrently. Task 1: prints numbers from 5 to 5000; Task 2: prints numbers from 500 to 50,000; Task 3: prints the English letter 'H' 100 times; Task 4: prints the English letter 'B' 2000 times.

Thread Synchronization: When multiple threads share an object and that object is modified by one or more of the threads, indeterminate results may occur (as we will see in the examples) unless access to the shared object is managed properly. That is 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);

The problem is caused because of Task 1(thread[i]) and Task 2 (thread[j]) are accessing a common resource in a way that causes conflict. This is a common problem and it is 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.

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 problem can be solved by giving only one thread at a time exclusive access to code that manipulates the shared object. During that time, other threads desiring to manipulate the object are kept waiting. When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting can proceed. This process, called **thread synchronization**, coordinates access to shared data by multiple concurrent threads. By synchronizing threads in this manner, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called **mutual exclusion**.

To avoid resource conflicts and to avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as critical region. The critical region is a block of code for example a method or just sequence of statements. You can use the synchronized keyword to synchronize the block of code method so that only one thread can access the block at a time.

```
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.

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 a block of statement: The general form of a synchronized statement is as follows:

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

The expression `expr` must evaluate to an (any) object reference. If the object is already locked by another thread, the thread is blocked until the lock is released.

To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the critical region. You can use the keyword `synchronized` to synchronize the method so that only one thread can access the thread at a time. When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

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  
    }  
}
```

The Volatile Modifier: If a variable is declared as **volatile**, then it is guaranteed that any thread which reads the field will see the most recently written value. Access to the variable acts as though it is enclosed in a **synchronized** block, synchronized on itself. When multiple threads using the same variable, each thread will have its own copy of the local cache for that variable. So, when it's updating the value, it is actually updated in the local cache not in the main variable memory. The other thread which is using the same variable doesn't know anything about the values changed by another thread. To avoid this problem, if you declare a variable as **volatile**, then it will not be stored in the local cache. Whenever thread is updating the values, it is updated to the main memory. So, other threads can access the updated value.

Deadlocks: Synchronization can also be achieved using locks. That is to prevent multiple accesses, threads can acquire and release a lock before using resources. If locks are not used properly, they may lead to deadlock. Deadlocking is a classic multithreading problem in which all work is incomplete because different threads are waiting for locks that will never be released. Examples of deadlocks includes traffic deadlock and two hungry persons with one knife and a fork