

Multithreaded programming

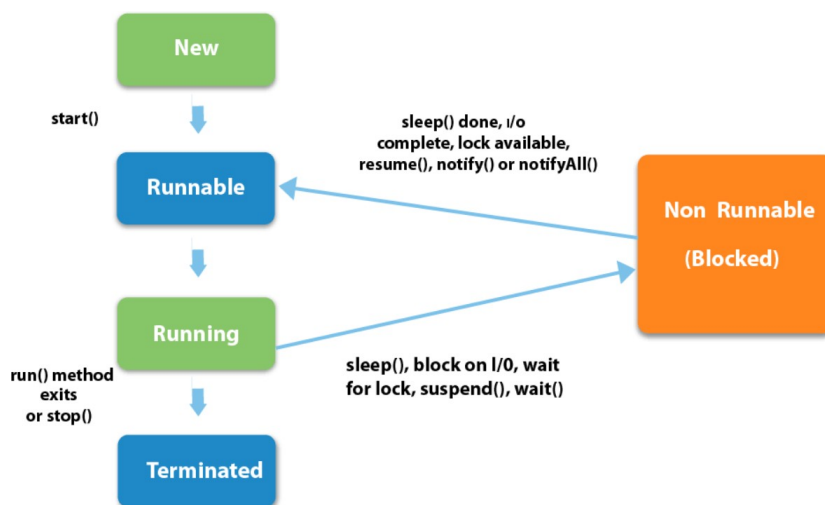
Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread. In a thread-based multitasking environment, single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. Multithreading enables the user to write efficient programs that make maximum use of the processing power available in the system.

Life cycle of a thread

A thread can be in one of the five states.

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



The thread will be in **new** state if an instance of thread is created but the `start()` method is

yet to be invoked. After the invocation of the `start()` method the thread moves to the **runnable** state. At this state the thread is ready to run but is not selected by the thread scheduler. The thread moves to the **running** state after it is selected by the thread scheduler to be run. The thread moves to the **blocked** state when the thread is still alive but not eligible to run due to reasons like block on I/O or sleep. When the thread exits from the `run()` method it is said to be in the **terminated** state.

Java's multithreading system is built upon the `Thread` class, its methods, and its companion interface, `Runnable`. The `Thread` class defines several methods that help manage threads.

getName() - Obtain a thread's name.

getPriority() - Obtain a thread's priority.

isAlive() - Determine if a thread is still running.

join() - Wait for a thread to terminate.

run() - Entry point for the thread.

sleep() - Suspend a thread for a period of time.

start() - Start a thread by calling its run method.

The Main Thread

When a Java program starts up, one thread begins running immediately. That is the *main thread* of the program. It is the thread from which other "child" threads will be spawned. It must be the last thread to finish execution because it performs various shutdown actions. Although the main thread is created automatically when the program is started, it can be controlled through a **Thread** object. A reference to it must be obtained by calling the method **currentThread()**, which is a **public static** member of **Thread**.

```
public class CurrentThreadDemo {  
  
    public static void main(String args[]) {  
  
        Thread t = Thread.currentThread();  
        System.out.println("Current thread: " + t);  
  
        t.setName("My Thread");  
        System.out.println("After name change: " + t);  
  
        try {
```

```

        for(int n = 5; n > 0; n--) {
            System.out.println(n);
            Thread.sleep(1000);
        }

        catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}

```

In this program, a reference to the current thread is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**. The program displays information about the thread. The program then calls **setName()** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds. The **sleep()** method in **Thread** might throw an **InterruptedException** and hence has to be handled appropriately.

Output:

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5
4
3
2
1

A Thread object when passed to the print method displays : the name of the thread, its priority, and the name of its group.

Creating a Thread

A thread can be created by :

- Implementing the Runnable interface
- Extending the Thread class

Runnable interface

Step 1 : Create a class that implements Runnable interface

Step 2 : Override the public void run method to include the required logic for a thread.

(In the main())

Step 3 : Create an object of the class that implemented Runnable.

Step 4 : Create an object of Thread class and pass the object created in Step 3 in the Thread constructor.

Step 5: Call the start() method using the Thread object.

```
public class NewThread_Runnable implements Runnable{

    public void run() {

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }

        }

        catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }

        System.out.println("Exiting child thread.");
    }

    public static void main(String args[]){

        NewThread_Runnable rn = new NewThread_Runnable();
        Thread t1 = new Thread(rn);
        t1.start();

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }

        }

        catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }

        System.out.println("Main thread exiting.");

    }

}
```

Extending Thread

Step 1 : Create a class that extends Thread class.

Step 2 : Override the public void run method to include the required logic for a thread.

(In the main())

Step 3 : Create an object of the class that extended Thread.

Step 4 : Call the start() method using the object created in Step 3.

```
public class NewThread_Thread extends Thread {

    public void run()
    {
```

```

        try {
            for(int i = 5; i > 0; i--)
            {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }

        }

        catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }

    public static void main(String args[])
    {

        NewThread_Thread t1 = new NewThread_Thread();
        t1.start();

        try {
            for(int i = 5; i > 0; i--)
            {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }

        }

        catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }

        System.out.println("Main thread exiting.");
    }
}

```

Creating Multiple Threads:

The above examples use only two threads: the main thread and one child thread. However, a program can spawn as many threads as needed. An example is shown below.

```

public class One extends Thread {

    public void run()
    {
        try
        {
            int i=0;
            while (i<5)
            {
                Thread.sleep(500);
                System.out.println("Good morning");
                i++;
            }

        }

    }
}

```

```

        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
    }
}

```

```

    public void run()
    {
        try
        {
            int i=0;
            while (i<5)
            {
                Thread.sleep(500);
                System.out.println("Hello");
                i++;
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
    }
}

public class Three extends Thread {

    public void run()
    {
        try
        {
            int i=0;
            while (i<5)
            {
                Thread.sleep(500);
                System.out.println("Welcome");
                i++;
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
    }
}

}

}
}

```

```

public class MyThread {

    public static void main(String[] args) {

        One t1=new One();
        Two t2=new Two();
        Three t3=new Three();
    }
}

```

```

        t1.start();
        t2.start();
        t3.start();

    try{        Thread.sleep(5000);

        }

    catch(InterruptedException e)
    {
        System.out.println("Interrupted");
    }

}

```

Using **isAlive()** and **join()**

In order to determine if a thread has finished execution, the **isAlive()** method of the Thread class is used. The **isAlive()** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise. The **join()** method waits until the thread on which it is called terminates. **join()** throws an **InterruptedException** and has to be appropriately handled.

```

public class IsAliveJoin extends Thread {

    public void run()
    {
        try {

            for(int i = 3; i > 0; i--)
            {

                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }

        }

        catch (InterruptedException e) {

            System.out.println("Child interrupted.");

        }

        System.out.println("Exiting child thread."+Thread.currentThread().getName());
    }
}

```

```

    }

    public static void main(String args[])
    {

        IsAliveJoin t1 = new IsAliveJoin();
        t1.start();

        IsAliveJoin t2 = new IsAliveJoin();
        t2.start();

        IsAliveJoin t3 = new IsAliveJoin();
        t3.start();

        System.out.println("Thread 1 is alive "+t1.isAlive());
        System.out.println("Thread 2 is alive "+t2.isAlive());
        System.out.println("Thread 3 is alive "+t3.isAlive());

        try {
            System.out.println("Waiting for threads to finish.");
            t1.join();
            t2.join();
            t3.join();
        }
        catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Thread 1 is alive "+t1.isAlive());
        System.out.println("Thread 2 is alive "+t2.isAlive());
        System.out.println("Thread 3 is alive "+t3.isAlive());

        System.out.println("Main thread exiting.");

    }
}

```

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. Each thread has a priority. Priorities are represented by a number between 1 and 10.

Methods that handle Thread priorities:

public final int getPriority(): The `java.lang.Thread.getPriority()` method returns the priority of the given thread.

public final void setPriority(int newPriority): The `java.lang.Thread.setPriority()` method updates or assigns the priority of the thread to `newPriority`. The method throws `IllegalArgumentException` if the value `newPriority` goes out of the range, which is 1 (minimum) to 10 (maximum).

There are 3 constants defined in Thread class:

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

Default priority of a thread is 5 (`NORM_PRIORITY`). The value of `MIN_PRIORITY` is 1 and the value of `MAX_PRIORITY` is 10.

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**. Java provides unique, language-level support for it.

The key to synchronization is the concept of the monitor. A monitor is an object that is used as a lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

What happens when there is no synchronization?

In the example below, there is no synchronization, so output is inconsistent.

```
public class Table {  
    void printTable(int n){  
        for(int i=1;i<=5;i++){  
            System.out.println(n*i);  
            try{  
                Thread.sleep(400);  
            }catch(InterruptedException e){  
                System.out.println(e);}  
        }  
    }  
}
```

```
}
```

```
class MyThread1 extends Thread{  
    Table t;  
    MyThread1(Table t){  
        this.t=t;  
    }  
    public void run(){  
        t.printTable(5);  
    }  
}
```

```
}
```

```
class MyThread2 extends Thread{  
    Table t;  
    MyThread2(Table t){  
        this.t=t;  
    }  
    public void run(){  
        t.printTable(100);  
    }  
}
```

```
}
```

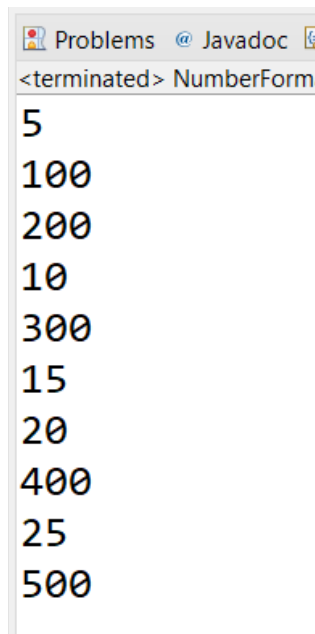
```
class TestSynchronization1 {  
    public static void main(String args[]){  
        Table obj = new Table();//only one object
```

```

MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
    }
}

```

Output:



```

5
100
200
10
300
15
20
400
25
500

```

Since there no synchronization, the output is inconsistent. That is, the multiples of 2 and 100 are printed out randomly. If there is a need to print the multiples of 2 entirely first then followed by the multiples of 10, then the printTable method has to be written as a synchronized method. In such a case the thread when it is running the synchronized method acquires a lock for the same and does not allow other threads to access the method till its run is complete.

```

public class Table {
    synchronized void printTable(int n){
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
        }
        try{
            Thread.sleep(400);
        }catch(InterruptedException e){
            System.out.println(e);}
    }
}

```

```
}
```

```
class MyThread1 extends Thread{
```

```
    Table t;
```

```
    MyThread1(Table t){
```

```
        this.t=t;
```

```
    }
```

```
    public void run(){
```

```
        t.printTable(5);
```

```
    }
```

```
}
```

```
class MyThread2 extends Thread{
```

```
    Table t;
```

```
    MyThread2(Table t){
```

```
        this.t=t;
```

```
    }
```

```
    public void run(){
```

```
        t.printTable(100);
```

```
    }
```

```
}
```

```
class TestSynchronization1 {
```

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

```
        Table obj = new Table();//only one object
```

```
        MyThread1 t1=new MyThread1(obj);
```

```
        MyThread2 t2=new MyThread2(obj);
```

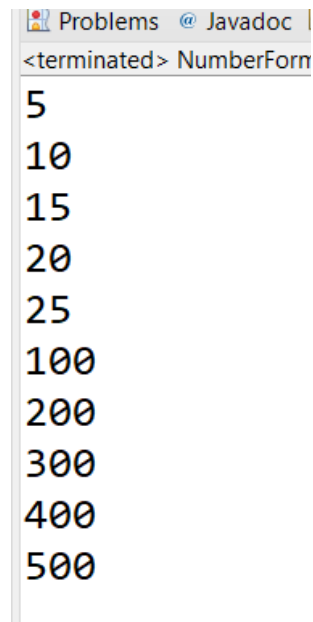
```
        t1.start();
```

```
        t2.start();
```

```
    }
```

```
}
```

Output:



```
Problems @ Javadoc
<terminated> NumberForm
5
10
15
20
25
100
200
300
400
500
```

As seen above, the output is synchronized because of the use of the synchronized keyword on the method printTable. The thread which accesses the object locks it till it finishes its execution. Another thread can access the object only when the lock is released.

Synchronized Block

While creating synchronized methods within classes is an easy and effective means of achieving synchronization, it will not work in all cases. One such case is when; user wants to synchronize access to objects of a class that was not designed for multithreaded access. (i.e. the class whose object is intended to be synchronized does not have synchronized method.) Further, this class might have been created by a third party, and one would not have access to the source code to add synchronized to the appropriate method/s within the class.

In this situation, the solution would be to put calls to such method/s inside a synchronized block.

This is the general form of the synchronized block is:

```
synchronized(object) {
// statements to be synchronized
}
```

object is a reference to the object being synchronized.

Example:

```
public class Table {  
    void printTable(int n){//method not synchronized  
        for(int i=1;i<=5;i++){  
            System.out.println(n*i);  
            try{  
                Thread.sleep(400);  
            } catch(Exception e){System.out.println(e);}  
        }  
    }  
}
```

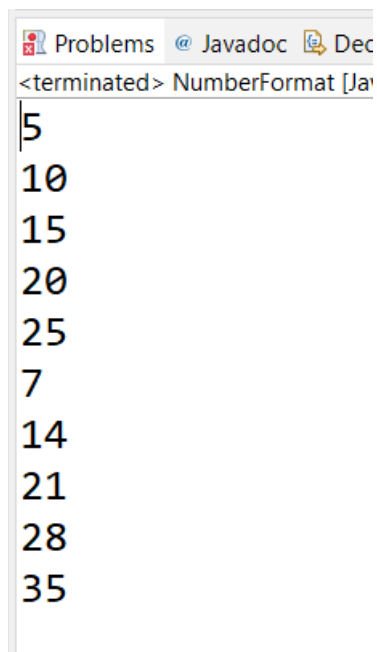
```
class MyThread1 extends Thread{  
    Table t;  
    MyThread1(Table t){  
        this.t=t;  
    }  
    public void run(){  
        synchronized(t) {  
            t.printTable(5);  
        }  
    }  
}
```

```
class MyThread2 extends Thread{  
    Table t;  
    MyThread2(Table t){  
        this.t=t;  
    }  
    public void run(){  
        synchronized(t) {  
            t.printTable(7);  
        }  
    }  
}
```

```
public class TestSynchronization1 {  
    public static void main(String args[]){
```

```
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

Output:



```
<terminated> NumberFormat [Ja
5
10
15
20
25
7
14
21
28
35
```

In the above example, the call to the printTable method is put in a synchronized block. And hence a synchronized output is obtained. That is the table of 5 is printed fully and is then followed by table of 7. The thread that takes control of an object locks it and releases it only when execution is completed.