Threads

Prepared by TalentSprint WISE Team

Threads

Threads

A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

A thread represents a seperate path of execution of a group of statements. In a Java program, if we write a group of statements, then these statements are executed by JVM one by one. This execution is called a `'Thread'`, because JVM uses a thread to execute these statements. This means that in every Java program, there is always a thread running internally. This thread is used by JVM to execute the program statements.

```
1 public class PrintmainThread {
2     public static void main(String args[]) {
3         System.out.println (“ Let us find the current thread’’);
4         Thread t = Thread.currentThread();
5         System.out.println(t);
6         System.out.println (“ It’s name: ’’ + t.getName());
7     }
8 }
```

Here, 't' is a `Thread` class object, the JVM creates a starting thread of execution, which executes the `main()` method.

We get the output as `Thread[main,5,main]`, where main is the thread name and 5 is the priority. Every thread will have a priority number associated with it.

The next `main` indicates the thread group name to which this thread belongs. A thread group represents a group of threads as a single unit. The 'main' thread belongs to 'main' thread group.

Uses of Threads

- In client-server based systems, the server program creates threads that allows it to respond to multiple users at the same time.
- GUI programs have a separate thread to gather user's interface events from the host OS.
- Do other things while waiting for slow I/O operations.
- Animations

Thread States

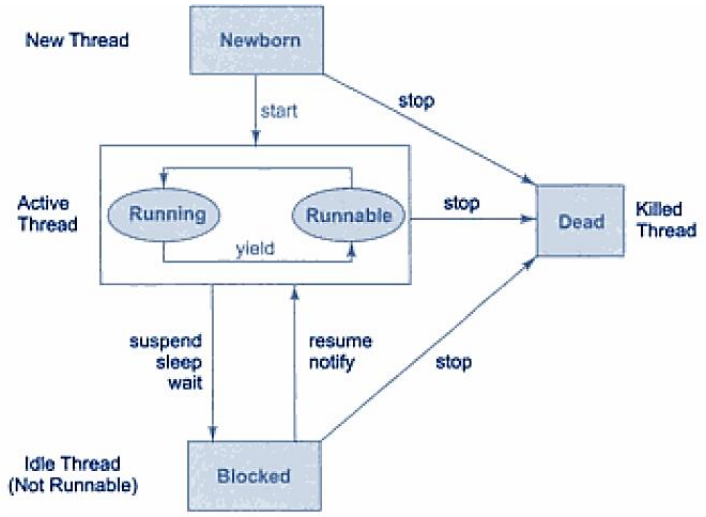


Figure 1: Life cycle of a Thread

New A thread that is just instantiated is in new state. When a `start()` method is invoked, the thread moves to the ready state from which it is automatically moved to runnable state by the thread scheduler.

Ready The thread is in runnable state after invocation of `start()` method, but the thread scheduler has not selected it to be the running thread.

Running The thread is in running state if the thread scheduler has selected it i.e, whenever the thread is under execution it is known as running state.

Suspended This is the state when the thread is still alive, but is currently not to executing.

Dead A runnable thread enters the dead or terminated state when it completes its task or terminates.

Thread Creation

Java defines two ways in which this can be accomplished:

1. **Extending the *Thread* Class**

   The first way to create a thread is to create a new class that extends `Thread`, and then to create an instance of that class. The extending class must override the `run()` method, which is the entry point for the new thread. It must also call `start()` to begin execution of the new thread.

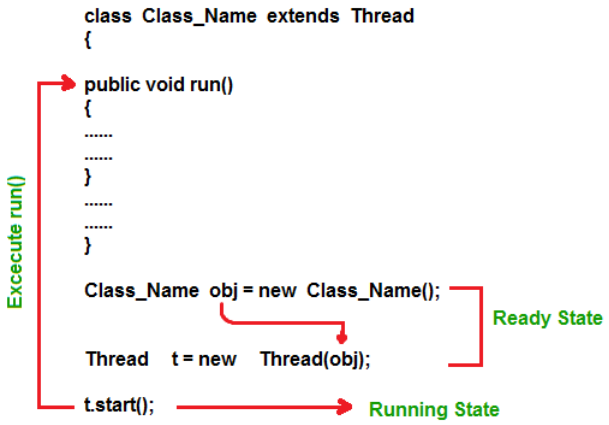


Figure 2: creating thread by extending `Thread` class

   Here, when extending the `Thread` class, the derived class cannot extend any other base classes because Java only allows single inheritance.

   This means that this class cannot inherit from any other class. Another way to program threads is by implementing `Runnable` interface.

2. **Implementing the Runnable Interface** This is useful approach to create a thread because we can implement The steps for creating a thread by using this mechanism are:

Step 1 Create a class that implements the interface `Runnable` and override `run()` method.

```
class MyThread implements Runnable {
    public void run() {
        // thread body of execution
    }
}
```

Step 2 Creating Object

```
MyThread myObject = new MyThread();
```

Step 3 Creating `Thread` object

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

Step 4 Start execution of a thread

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

An example program illustrating creation and invocation of a thread object is given below:

```
1  class MyThreadRunnable implements Runnable {
2      public void run() {
3        System.out.println(“thread is running...”);
4      }
5      public static void main(String args[]) {
6       MyThreadRunnable myObject = new MyThreadRunnable();
7       Thread thread = new Thread(myObject);
8       thread.start();
9      }
10 }
```

In the above program, we created an object to MyThreadRunnable class. This object contains only run() but we need start() to initiate the thread thus, we attached this object to Thread object(thread).

Thread Class Methods

sleep() method causes the current thread to sleep for a specified amount of time in milliseconds.

```
public static void sleep(long millis) throws
    InterruptedException
```

For example, the code below puts the thread in sleep state for 3 minutes.

```
try {
    Thread.sleep(3 * 60 * 1000); // thread sleeps for 3 minutes
} catch(InterruptedException ex) {}
```

yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution.

```
public static void yield()
```

isAlive() method returns **true** if the thread upon which it is called has been started but not moved to the dead state.

```
public final boolean isAlive()
```

join() When a thread calls `join()` on another thread instance, the caller thread will wait till the called thread finishes execution.

```
final void join() throws InterruptedException
```

`join()` can also be specified with some timeout, in which case the thread waits at most milliseconds for this thread to die. If timeout specified as '0' means the thread will wait forever.

```
final void join(long millis) throws InterruptedException
final void join(long millis, int nanos) throws
    InterruptedException
```

A thread waiting because of `join()` can also be interrupted. Hence the method throws a `InterruptedException`.

Naming Threads

Every thread is given a name. If you don't specify a name, a default name will be created. For example the main thread is named "main". The default name of a user defined thread is "Thread-0" for the first thread created, "Thread-1" for the second and so on. The Thread class provides methods to change and get the name of a thread.

public void setName(String name) is used to change the name of a thread.

publie String getName() returns the name of a thread.

Thread Priority

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

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.

```
1  class NameAndPriority extends Thread {
2      public void run() {
3        System.out.println (“ running thread name is: ’’+
4                        Thread.currentThread().getName());
5        System.out.println (“ running thread priority  is :’’+
6                        Thread.currentThread().getPriority());
7      }
8      public static void main(String args[])  {
9        TestPriority  priorityOne = new TestPriority();
10       TestPriority  priorityTwo = new TestPriority();
11       priorityOne. setPriority (Thread.MAX_PRIORITY);
12       priorityTwo. setPriority (Thread.MIN_PRIORITY);
13       priorityOne.setName(“User1’’); // naming a thread
14       priorityTwo.setName(“User2’’); // naming a thread
15       priorityOne. start ();
16       priorityTwo.start ();
17     }
18 }
```

Daemon Threads

So far the threads that have been created are called foreground threads. A program continues to execute as long as it has at least one foreground (non-daemon) thread that is alive. The daemon threads are also called service threads. They are used for background processes that will continue only as long as the active threads of the program are alive.

Daemon threads cease to execute when there are no non-daemon threads alive because when VM detects that the only remaining threads are daemon threads, it exits.

Note: garbage collector thread is a daemon thread that runs with a low priority.

Example

```
1  public class DollDaemon extends Thread {
2      public void run() {
3          while(true) {
4            try {
5                System.out.println (“ Have a nice day’’ ) ;
6                Thread.sleep(1000);
7            } catch(InterruptedException e) {
8                break;
9            }
10         }
11     }
12     public static void main(String args[]) {
13         DollDaemon doll = new DollDaemon();
14         doll . setDaemon(true);
15         doll . start ( ) ;
16         char c[] = {‘H’, ‘A’, ‘I’};
17         int i = 0;
18         try {
19             while (i < 3) {
20                 System.out.print(c[i++] + “ ’’ ) ;
21                 Thread.sleep(1000);
22             }
23         } catch(InterruptedException e) {}
24         System.out.println (“ End of main method’’);
25     }
26 }
```

Synchronization

Whenever multiple threads are trying to use same resource then their may be a chance of getting wrong output, to overcome this problem thread synchronization can be used.

Allowing only one thread at a time to utilize the same resource out of multiple threads is known as thread synchronization or thread safe.

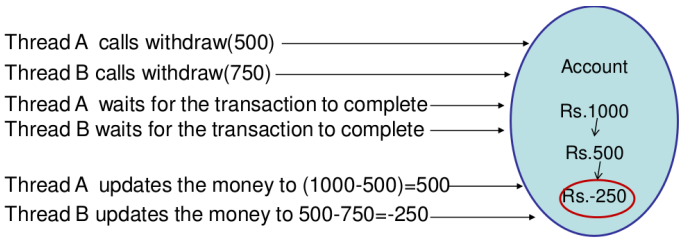
Java implements this concept by using `synchronized` keyword. `synchronized` can be used in 2 ways.

1. A method can be marked `synchronized`

2. A block of code can be marked `synchronized`

Why threads synchronization?

- Let us say we have an Account class which has withdraw and deposit methods.
- For each transaction (deposit or withdraw) a thread is created.
- Let us visualize what happens when two people simultaneously withdraw from the same account object .
- Let us assume that there is Rs. 1000 in the account.

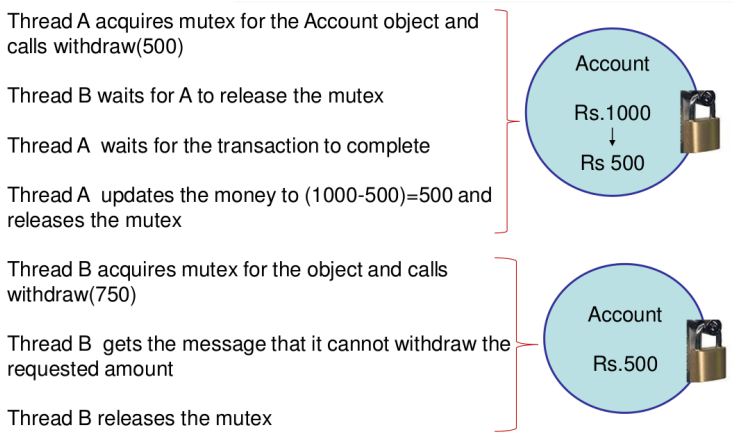


Example - Code without synchronization

```
1  class Account {
2      private int money;
3      Account(int amt) {
4          money = amt;
5      }
6      void withdraw(int amt) {
7          if (amt < money) {
8              try {
9                  Thread.sleep(1000);
10                 money = money − amt;
11             } catch(Exception e){}
12         System.out.println (“ Received Rs. ’’ + amt +
13         “ by ’’ + Thread.currentThread().getName());
14         }
15         else
16         {
17             System.out.println (“ Sorry ’’ +
18             Thread.currentThread().getName() +
19             “, Requested amt Rs.” + amt +
20             “ is not available . ’’);
21         }
```

```
22            System.out.println (“ Balance Rs. ’’ + money);
23        }
24    }
25    public class AccountUpdate implements Runnable {
26      Account a;
27      int amt;
28      public static void main(String str[]){
29            Account account = new Account(1000);
30            new AccountUpdate(account, “Ramesh’’, 500);
31            new AccountUpdate(account, “Suresh’’, 750);
32      }
33      public AccountUpdate(Account a, String name, int amt)
34      {
35            this.a = a;
36            this.amt = amt;
37            new Thread(this, name).start();
38      }
39      public void run() {
40            a.withdraw(amt);
41      }
42    }
```

Solution to the Account problem



Example - Code using synchronization

If an object is visible to more than one thread, all reads or writes to that object's non final attributes should be done through `synchronized` methods.

Approach 1: Add `synchronized` keyword to withdraw and other critical methods of the `Account` object.

```
synchronized void withdraw(int amt)
```

Approach 2: Use `synchronized` statements by explicitly locking the object before calling critical methods of the `Account` object.

```
public void run() {
    synchronized(a) {
        a.withdraw(amt);
    }
}
```

Inter-Thread Communication

Inter-thread communication is required when execution of one thread depends on another thread's task. In such case, the second thread intimates or notifies the first thread when it has finished some task that the first thread is waiting for. The best suited situation to understand this is a producer-consumer problem.

A producer thread produces something which consumer thread consumes. A producer and consumer thread can run independently. Producer makes sure that it has produced enough for consumer to consume. If producer has not produced then consumer will have to wait till producer finishes.

```
1  class Customer {
2     int amount = 0;
3     int flag = 0;
4     public synchronized int withdraw(int amount) {
5
6     System.out.println(Thread.currentThread().getName() +
7     " is going to withdraw");
8       if (flag == 0) {
9           try {
10              System.out.println ("Waiting...'');
11              wait();
12          } catch(Exception e) {}
13      }
14       this.amount -= amount;
15       System.out.println ("Withdraw Complete");
16       return amount;
17    }
18    public synchronized void deposit(int amount) {
```

```
19          System.out.println(Thread.currentThread().getName() +
20          " is going to deposit");
21          this.amount += amount;
22          notifyAll();
23          System.out.println("Deposit Complete");
24          flag = 1;
25      }
26  }
27  class ProducerConsumerThreads {
28      public static void main(String args[]) {
29          final Customer customer = new Customer();
30          Thread one = new Thread() {
31              public void run() {
32                  customer.withdraw(5000);
33                  System.out.println("Balance Rs. " +
34                                      customer.amount);
35              }
36          };
37          Thread two = new Thread() {
38              public void run() {
39                  customer.deposit(10000);
40                  System.out.println("Deposited amount Rs. "+
41                                      customer.amount);
42              }
43          };
44          one.start();
45          two.start();
46      }
47  }
```