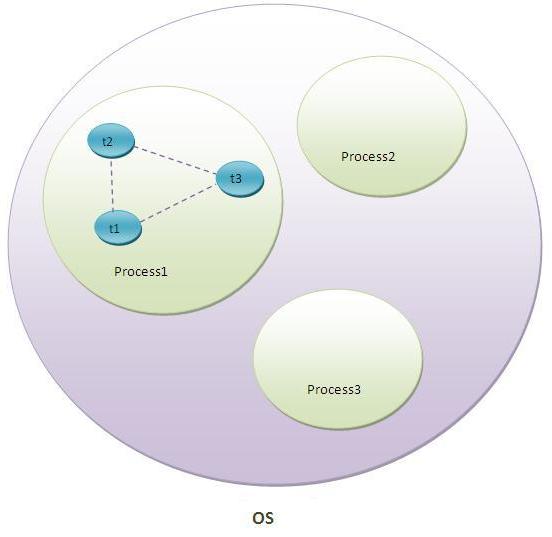
Multi Threading

**Multithreading in java** is a process of executing multiple threads simultaneously.

# What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time one thread is executed only.

|  |
| --- |
| Life cycle of a Thread (Thread States) A thread can be in one of the five states. According to sun, there is only 4 states in**thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.  But for better understanding the threads, we are explaining it in the 5 states.  The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:   1. New 2. Runnable 3. Running 4. Non-Runnable (Blocked) 5. Terminated   thread life cycle in java  **New**:The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

**Runnable**: 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.

**Non-Runnable (Blocked):** This is the state when the thread is still alive, but is currently not eligible to run.

**Terminated**: A thread is in terminated or dead state when its run() method exits.

# How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

# By extending Thread class:

**class** Multi **extends** Thread{

**public** **void** run(){

System.out.println("thread is running...");

}

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

Multi t1=**new** Multi();

t1.start();

 }

}

Output:thread is running...

# By implementing the Runnable interface:

**class** Multi3 **implements** Runnable{

**public** **void** run(){

System.out.println("thread is running...");

}

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

Multi3 m1=**new** Multi3();

Thread t1 =**new** Thread(m1);

t1.start();

 }

}

Output:thread is running...

# Commonly used methods of Thread class:

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public boolean isDaemon():**tests if the thread is a daemon thread. 16. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 17. **public void interrupt():**interrupts the thread. 18. **public boolean isInterrupted():**tests if the thread has been interrupted. 19. **public static boolean interrupted():**tests if the current thread has been interrupted. |

## Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

## What if we call run() method directly instead start() method?

|  |  |
| --- | --- |
| Each thread starts in a separate call stack.  Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack. The join() method:  |  | | --- | | The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task. | |

# Java Thread Pool

**Java Thread pool** represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

## Advantage of Java Thread Pool

Better performance It saves time because there is no need to create new thread.

## Real time usage

It is used in Servlet and JSP where container creates a thread pool to process the request.

## Example of Java Thread Pool

Let's see a simple example of java thread pool using ExecutorService and Executors.

*File: WorkerThrad.java*

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**class** WorkerThread **implements** Runnable {

**private** String message;

**public** WorkerThread(String s){

**this**.message=s;

    }

**public** **void** run() {

        System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);

        processmessage();//call processmessage method that sleeps the thread for 2 seconds

        System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name

    }

**private** **void** processmessage() {

**try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) { e.printStackTrace(); }

    }

}

*File: JavaThreadPoolExample.java*

**public** **class** TestThreadPool {

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

        ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads

**for** (**int** i = 0; i < 10; i++) {

            Runnable worker = **new** WorkerThread("" + i);

            executor.execute(worker);//calling execute method of ExecutorService

          }

        executor.shutdown();

**while** (!executor.isTerminated()) {   }

        System.out.println("Finished all threads");

    }

 }

# ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Java thread group is implemented by java.lang.ThreadGroup class.

## Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

|  |  |  |
| --- | --- | --- |
| **No.** | **Constructor** | **Description** |
| 1) | ThreadGroup(String name) | creates a thread group with given name. |
| 2) | ThreadGroup(ThreadGroup parent, String name) | creates a thread group with given parent group and name. |

## Important methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | int activeCount() | returns no. of threads running in current group. |
| 2) | int activeGroupCount() | returns a no. of active group in this thread group. |
| 3) | void destroy() | destroys this thread group and all its sub groups. |
| 4) | String getName() | returns the name of this group. |
| 5) | ThreadGroup getParent() | returns the parent of this group. |
| 6) | void interrupt() | interrupts all threads of this group. |
| 7) | void list() | prints information of this group to standard console. |

Let's see a code to group multiple threads.

ThreadGroup tg1 = **new** ThreadGroup("Group A");

Thread t1 = **new** Thread(tg1,**new** MyRunnable(),"one");

Thread t2 = **new** Thread(tg1,**new** MyRunnable(),"two");

Thread t3 = **new** Thread(tg1,**new** MyRunnable(),"three");

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

1. Thread.currentThread().getThreadGroup().interrupt();

## ThreadGroup Example

*File: ThreadGroupDemo.java*

**public** **class** ThreadGroupDemo **implements** Runnable{

**public** **void** run() {

          System.out.println(Thread.currentThread().getName());

    }

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

      ThreadGroupDemo runnable = **new** ThreadGroupDemo();

          ThreadGroup tg1 = **new** ThreadGroup("Parent ThreadGroup");

          Thread t1 = **new** Thread(tg1, runnable,"one");

          t1.start();

          Thread t2 = **new** Thread(tg1, runnable,"two");

          t2.start();

          Thread t3 = **new** Thread(tg1, runnable,"three");

          t3.start();

          System.out.println("Thread Group Name: "+tg1.getName());

         tg1.list();

    }

   }

Output:

one

two

three

Thread Group Name: Parent ThreadGroup

java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]

Thread[one,5,Parent ThreadGroup]

Thread[two,5,Parent ThreadGroup]

Thread[three,5,Parent ThreadGroup]

# Constructors of Thread Class

|  |
| --- |
| **Constructors** |
| **Constructor and Description** |
| [**Thread**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread())()  Allocates a new Thread object. |
| [**Thread**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.Runnable))([**Runnable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target)  Allocates a new Thread object. |
| [**Thread**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.Runnable,%20java.lang.String))([**Runnable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target, [**String**](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)  Allocates a new Thread object. |
| [**Thread**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.String))([**String**](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)  Allocates a new Thread object. |
| [**Thread**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.ThreadGroup,%20java.lang.Runnable))(**[ThreadGroup](http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html" \o "class in java.lang)** group, [**Runnable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target)  Allocates a new Thread object. |
| [**Thread**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.ThreadGroup,%20java.lang.Runnable,%20java.lang.String))(**[ThreadGroup](http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html" \o "class in java.lang)** group, [**Runnable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target, [**String**](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)  Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group. |
| [**Thread**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.ThreadGroup,%20java.lang.Runnable,%20java.lang.String,%20long))(**[ThreadGroup](http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html" \o "class in java.lang)** group, [**Runnable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target, [**String**](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name, long stackSize)  Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified *stack size*. |
| [**Thread**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.ThreadGroup,%20java.lang.String))(**[ThreadGroup](http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html" \o "class in java.lang)** group, [**String**](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)  Allocates a new Thread object. |

# Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

## Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

## Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. static synchronization.
2. Cooperation (Inter-thread communication in java)

## Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

## Syntax to use synchronized block

**synchronized** (object reference expression) {

  //code block

}

## Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.

## Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference .class name .class. A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

**static** **void** printTable(**int** n) {

**synchronized** (Table.**class**) {       // Synchronized block on class A

        // ...

    }

}

# Deadlock in java

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

## Example of Deadlock in java

**public** **class** TestDeadlockExample1 {

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

**final** String resource1 = "ratan jaiswal";

**final** String resource2 = "vimal jaiswal";

    // t1 tries to lock resource1 then resource2

    Thread t1 = **new** Thread() {

**public** **void** run() {

**synchronized** (resource1) {

           System.out.println("Thread 1: locked resource 1");

**try** { Thread.sleep(100);} **catch** (Exception e) {}

**synchronized** (resource2) {

            System.out.println("Thread 1: locked resource 2");

           }

         }

      }

    };

    // t2 tries to lock resource2 then resource1

    Thread t2 = **new** Thread() {

**public** **void** run() {

**synchronized** (resource2) {

          System.out.println("Thread 2: locked resource 2");

**try** { Thread.sleep(100);} **catch** (Exception e) {}

**synchronized** (resource1) {

            System.out.println("Thread 2: locked resource 1");

          }

        }

      }

    };

    t1.start();

    t2.start();

  }

}

## Deadlock Prevention

In some situations it is possible to prevent deadlocks. I'll describe three techniques in this text:

1. Lock Ordering
2. Lock Timeout
3. Deadlock Detection

### Lock Ordering

Deadlock occurs when multiple threads need the same locks but obtain them in different order.

If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur.

# Inter-thread communication in Java

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

* wait()
* notify()
* notifyAll()

## 1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final void wait()throws InterruptedException | waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException | waits for the specified amount of time. |

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

## 2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

public final void notify()

## 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

## Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

* It is because they are related to lock and object has a lock.
* Locks are made available on per Object basis, which is another reason wait and notify is declared in Object class rather then Thread class.

## Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| wait() method releases the lock | sleep() method doesn't release the lock. |
| is the method of Object class | is the method of Thread class |
| is the non-static method | is the static method |
| should be notified by notify() or notifyAll() methods | after the specified amount of time, sleep is completed. |

## Example of inter thread communication in java

Let's see the simple example of inter thread communication.

**class** Customer{

**int** amount=10000;

**synchronized** **void** withdraw(**int** amount){

System.out.println("going to withdraw...");

**if**(**this**.amount<amount){

System.out.println("Less balance; waiting for deposit...");

**try**{wait();}**catch**(Exception e){}

}

**this**.amount-=amount;

System.out.println("withdraw completed...");

}

**synchronized** **void** deposit(**int** amount){

System.out.println("going to deposit...");

**this**.amount+=amount;

System.out.println("deposit completed... ");

notify();

}

}

**class** Test{

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

**final** Customer c=**new** Customer();

**new** Thread(){

**public** **void** run(){c.withdraw(15000);}

}.start();

**new** Thread(){

**public** **void** run(){c.deposit(10000);}

}.start();

}}

Output: going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed

## Why wait (), notify () and notifyAll () must be called from synchronized block or method in Java

* If we don't call wait () or notify () method from synchronized context we will receive IllegalMonitorStateException in java.
* Race condition between wait () and notify () : Now let's think how does this potential race condition get resolved? This race condition is resolved by using synchronized keyword and locking provided by java. In order to call the wait (), notify () or notifyAll () methods in Java, we must have obtained the lock for the object on which we're calling the method. Since the wait () method in Java also releases the lock prior to waiting and reacquires the lock prior to returning from the wait () method, we must use this lock to ensure that checking the condition (buffer is full or not) and setting the condition (taking element from buffer) is atomic which can be achieved by using synchronized method or block in Java.

# Interrupting a Thread:

|  |
| --- |
| If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption. |

## The 3 methods provided by the Thread class for interrupting a thread

|  |
| --- |
| * **public void interrupt()** * **public static boolean interrupted()** * **public boolean isInterrupted()** |

## Example of interrupting a thread that stops working

|  |
| --- |
| In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception. |

**class** TestInterruptingThread1 **extends** Thread{

**public** **void** run(){

**try**{

Thread.sleep(1000);

System.out.println("task");

}**catch**(InterruptedException e){

**throw** **new** RuntimeException("Thread interrupted..."+e);

}

}

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

TestInterruptingThread1 t1=**new** TestInterruptingThread1();

t1.start();

**try**{

t1.interrupt();

}**catch**(Exception e){System.out.println("Exception handled "+e);}

}

}