# Multithreading in Java

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

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But, we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

### Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent.** So, it doesn't affect other threads if exception occur in a single thread.

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

* **Process-based Multitasking(Multiprocessing)**
* **Thread-based Multitasking(Multithreading)**

### 1) Process-based Multitasking (Multiprocessing)

* Each process has its own address in memory i.e. each process allocates separate memory area.
* Process is heavyweight.
* Cost of communication between the processes is high.
* Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

### 2) Thread-based Multitasking (Multithreading)

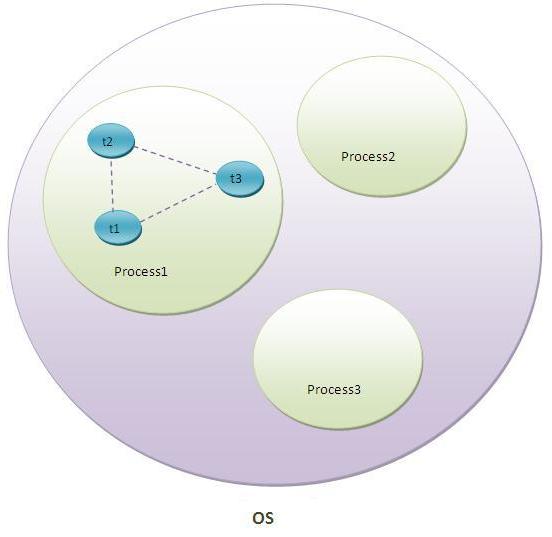
* Threads share the same address space.
* Thread is lightweight.
* Cost of communication between the thread is low.

#### Note: At least one process is required for each thread.

## 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.

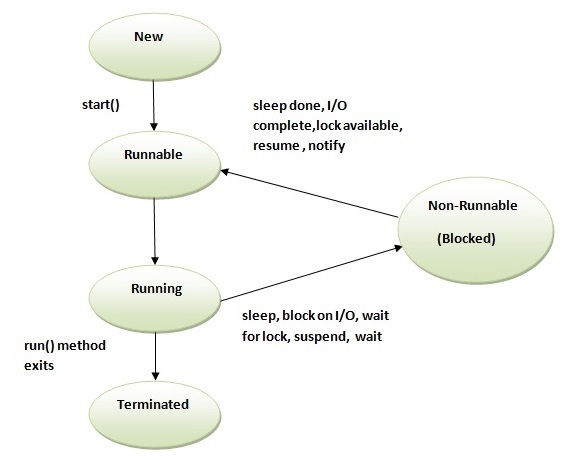
# Life cycle of a Thread (Thread States)

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**



|  |
| --- |
| 1) New The thread is in new state, if you create an instance of Thread class but before the invocation of start() method. |

### 2) 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.

### 3) Running

The thread is in running state if the thread scheduler has selected it.

### 4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### 5) 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.

### Thread class:

|  |
| --- |
| Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface. |

### Commonly used Constructors of Thread class:

|  |
| --- |
| * **Thread()** * **Thread(String name)** * **Thread(Runnable r)** * **Thread(Runnable r,String name)** |

### 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 void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. |

### Runnable interface:

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

### Starting a thread:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts (with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

### 1) Java Thread Example 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...

### 2) Java Thread Example by implementing 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...

|  |
| --- |
| If you are not extending the Thread class, your class object would not be treated as a thread object. So, you need to explicitly create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute. |

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

## Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

* *public static void sleep(long miliseconds)throws InterruptedException*
* *public static void sleep(long miliseconds, int nanos)throws InterruptedException*

## Example of sleep method in java

***class****TestSleepMethod1****extends****Thread{*

***public******void****run(){*

***for****(****int****i=1;i<5;i++){*

***try****{*

*Thread.sleep(500);*

*}****catch****(InterruptedException e){*

*System.out.println(e);*

*}*

*System.out.println(i);*

*}*

*}*

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

*TestSleepMethod1 t1=****new****TestSleepMethod1();*

*TestSleepMethod1 t2=****new****TestSleepMethod1();*

*t1.start();*

*t2.start();*

*}*

*}*

**Output:**

*1*

*1*

*2*

*2*

*3*

*3*

*4*

*4*

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

# 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.

**Let's understand it by the example given below:**

***public******class****TestThreadTwice1****extends****Thread{*

***public******void****run(){*

*System.out.println("running...");*

*}*

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

*TestThreadTwice1 t1=****new****TestThreadTwice1();*

*t1.start();*

*t1.start();*

*}*

*}*

**Output:**

running

Exception in thread "main" java.lang.IllegalThreadStateException

# 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. |

***class****TestCallRun1****extends****Thread{*

***public******void****run(){*

*System.out.println("running...");*

*}*

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

*TestCallRun1 t1=****new****TestCallRun1();*

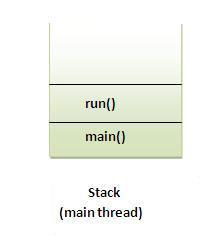
*t1.run();//fine, but does not start a separate call stack*

*}*

*}*

**Output**:

running...



***Problem if you direct call run() method***

***class****TestCallRun2****extends****Thread{*

***public******void****run(){*

***for****(****int****i=1;i<5;i++){*

***try****{*

*Thread.sleep(500);*

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

*}*

*System.out.println(i);*

*}*

*}*

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

*TestCallRun2 t1=****new****TestCallRun2();*

*TestCallRun2 t2=****new****TestCallRun2();*

*t1.run();*

*t2.run();*

*}*

*}*

**Output:**  
 1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object. |

# 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.

### Syntax:

|  |
| --- |
| *public void join()throws InterruptedException* |
| *public void join(long milliseconds)throws InterruptedException* |

***Example of join() method***

***class****TestJoinMethod1****extends****Thread{*

***public******void****run(){*

***for****(****int****i=1;i<=5;i++){*

***try****{*

*Thread.sleep(500);*

*}****catch****(Exception e){*

*System.out.println(e);*

*}*

*System.out.println(i);*

*}*

*}*

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

*TestJoinMethod1 t1=****new****TestJoinMethod1();*

*TestJoinMethod1 t2=****new****TestJoinMethod1();*

*TestJoinMethod1 t3=****new****TestJoinMethod1();*

*t1.start();*

***try****{*

*t1.join();*

*}****catch****(Exception e){*

*System.out.println(e);*

*}*

*t2.start();*

*t3.start();*

*}*

*}*

**Output:**

1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

|  |
| --- |
| As you can see in the above example,when t1 completes its task then t2 and t3 starts executing. |

***Example of join(long miliseconds) method***

***class****TestJoinMethod2****extends****Thread{*

***public******void****run(){*

***for****(****int****i=1;i<=5;i++){*

***try****{*

*Thread.sleep(500);*

*}****catch****(Exception e){*

*System.out.println(e);*

*}*

*System.out.println(i);*

*}*

*}*

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

*TestJoinMethod2 t1=****new****TestJoinMethod2();*

*TestJoinMethod2 t2=****new****TestJoinMethod2();*

*TestJoinMethod2 t3=****new****TestJoinMethod2();*

*t1.start();*

***try****{*

*t1.join(1500);*

*}****catch****(Exception e){*

*System.out.println(e);*

*}*

*t2.start();*

*t3.start();*

*}*

*}*

**Output:**  
 1

2

3

1

4

1

2

5

2

3

3

4

4

5

5

|  |
| --- |
| In the above example, when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing. |

### getName(),setName(String) and getId() method:

|  |
| --- |
| ***public String getName()*** |
| ***public void setName(String name)*** |
| ***public long getId()*** |

***class****TestJoinMethod3****extends****Thread{*

***public******void****run(){*

*System.out.println("running...");*

*}*

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

*TestJoinMethod3 t1=****new****TestJoinMethod3();*

*TestJoinMethod3 t2=****new****TestJoinMethod3();*

*System.out.println("Name of t1:"+t1.getName());*

*System.out.println("Name of t2:"+t2.getName());*

*System.out.println("id of t1:"+t1.getId());*

*t1.start();*

*t2.start();*

*t1.setName("Sonoo Jaiswal");*

*System.out.println("After changing name of t1:"+t1.getName());*

*}*

*}*

**Output:**  
 Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changing name of t1:Sonoo Jaiswal

running...

### The currentThread() method:

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

### Syntax:

|  |
| --- |
| public static Thread currentThread() |

***Example of currentThread() method***

***class****TestJoinMethod4****extends****Thread{*

***public******void****run(){*

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

*}*

*}*

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

*TestJoinMethod4 t1=****new****TestJoinMethod4();*

*TestJoinMethod4 t2=****new****TestJoinMethod4();*

*t1.start();*

*t2.start();*

*}*

*}*

**Output:**  
 Thread-0

Thread-1

# Naming Thread and Current Thread

## Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method.

The syntax of setName() and getName() methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

## Example of naming a thread

***class****TestMultiNaming1****extends****Thread{*

***public******void****run(){*

*System.out.println("running...");*

*}*

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

*TestMultiNaming1 t1=****new****TestMultiNaming1();*

*TestMultiNaming1 t2=****new****TestMultiNaming1();*

*System.out.println("Name of t1:"+t1.getName());*

*System.out.println("Name of t2:"+t2.getName());*

*t1.start();*

*t2.start();*

*t1.setName("Sonoo Jaiswal");*

*System.out.println("After changing name of t1:"+t1.getName());*

*}*

*}*

**Output**:

Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changeling name of t1:Sonoo Jaiswal

running...

## Current Thread

The currentThread() method returns a reference of currently executing thread.

***public static Thread currentThread()***

### Example of currentThread() method

***class****TestMultiNaming2****extends****Thread{*

***public******void****run(){*

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

*}*

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

*TestMultiNaming2 t1=****new****TestMultiNaming2();*

*TestMultiNaming2 t2=****new****TestMultiNaming2();*

*t1.start();*

*t2.start();*

*}*

*}*

**Output:**

Thread-0

Thread-1

# Priority of a Thread (Thread Priority):

|  |
| --- |
| Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as **preemptive scheduling**). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. |

## 3 constants defined in Thread class:

|  |
| --- |
| **public static int MIN\_PRIORITY**  **public static int NORM\_PRIORITY**  **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. |

### Example of priority of a Thread:

***class****TestMultiPriority1****extends****Thread{*

***public******void****run(){*

*System.out.println("running thread name is:"+Thread.currentThread().getName());*

*System.out.println("runing thread priority is:"+Thread.currentThread().getPriority());  }*

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

*TestMultiPriority1 m1=****new****TestMultiPriority1();*

*TestMultiPriority1 m2=****new****TestMultiPriority1();*

*m1.setPriority(Thread.MIN\_PRIORITY);*

*m2.setPriority(Thread.MAX\_PRIORITY);*

*m1.start();*

*m2.start();*

*}*

*}*

**Output:**

running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

# Daemon Thread in Java

**Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

## Points to remember for Daemon Thread in Java

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

### Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

### Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread as daemon thread or user thread. |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

### Simple example of Daemon thread in java

***File: MyThread.java***

***public******class****TestDaemonThread1****extends****Thread{*

***public******void****run(){*

***if****(Thread.currentThread().isDaemon()){*

*//checking for daemon thread*

*System.out.println("daemon thread work");*

*}****else****{*

*System.out.println("user thread work");*

*}*

*}*

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

*TestDaemonThread1 t1=****new****TestDaemonThread1();*

*//creating thread*

*TestDaemonThread1 t2=****new****TestDaemonThread1();*

*TestDaemonThread1 t3=****new****TestDaemonThread1();*

*t1.setDaemon(****true****);*

*//now t1 is daemon thread*

*t1.start();//starting threads*

*t2.start();*

*t3.start();*

*}*

*}*

#### Output

daemon thread work

user thread work

user thread work

#### Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

***File: MyThread.java***

***class****TestDaemonThread2****extends****Thread{*

***public******void****run(){*

*System.out.println("Name: "+Thread.currentThread().getName());*

*System.out.println("Daemon: "+Thread.currentThread().isDaemon());*

*}*

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

*TestDaemonThread2 t1=****new****TestDaemonThread2();*

*TestDaemonThread2 t2=****new****TestDaemonThread2();*

*t1.start();*

*t1.setDaemon(****true****);//will throw exception here*

*t2.start();*

*}*

*}*

**Output:**

exception in thread main: java.lang.IllegalThreadStateException

# 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: WorkerThread.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");*

*}*

*}*

**Output:**

pool-1-thread-1 (Start) message = 0

pool-1-thread-2 (Start) message = 1

pool-1-thread-3 (Start) message = 2

pool-1-thread-5 (Start) message = 4

pool-1-thread-4 (Start) message = 3

pool-1-thread-2 (End)

pool-1-thread-2 (Start) message = 5

pool-1-thread-1 (End)

pool-1-thread-1 (Start) message = 6

pool-1-thread-3 (End)

pool-1-thread-3 (Start) message = 7

pool-1-thread-4 (End)

pool-1-thread-4 (Start) message = 8

pool-1-thread-5 (End)

pool-1-thread-5 (Start) message = 9

pool-1-thread-2 (End)

pool-1-thread-1 (End)

pool-1-thread-4 (End)

pool-1-thread-3 (End)

pool-1-thread-5 (End)

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.

#### Note: Now suspend(), resume() and stop() methods are deprecated.

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.

***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]

# Java Shutdown Hook

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

### When does the JVM shut down?

The JVM shuts down when:

* **user presses ctrl+c on the command prompt**
* **System.exit(int) method is invoked**
* **user logoff**
* **user shutdown etc.**

#### The addShutdownHook(Thread hook) method

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine.

**Syntax:**

***public******void****addShutdownHook(Thread hook){}*

The object of Runtime class can be obtained by calling the static factory method getRuntime().

**For example:**

*Runtime r = Runtime.getRuntime();*

#### Factory method

The method that returns the instance of a class is known as **factory method**.

### Simple example of Shutdown Hook

***class****MyThread****extends****Thread{*

***public******void****run(){*

*System.out.println("shut down hook task completed..");*

*}*

*}*

***public******class****TestShutdown1{*

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

*Runtime r=Runtime.getRuntime();*

*r.addShutdownHook(****new****MyThread());*

*System.out.println("Now main sleeping... press ctrl+c to exit");*

***try****{*

*Thread.sleep(3000);*

*}****catch****(Exception e) {*

*}*

*}*

*}*

**Output:**

Now main sleeping... press ctrl+c to exit

shut down hook task completed..

#### Note: The shutdown sequence can be stopped by invoking the halt(int) method of Runtime class.

### Same example of Shutdown Hook by annonymous class:

***public******class****TestShutdown2{*

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

*Runtime r=Runtime.getRuntime();*

*r.addShutdownHook(****new****Thread(){*

***public******void****run(){*

*System.out.println("shut down hook task completed..");*

*}*

*}*

*);*

*System.out.println("Now main sleeping... press ctrl+c to exit");*

***try****{*

*Thread.sleep(3000);*

*}****catch****(Exception e) {}*

*}*

*}*

**Output**:

Now main sleeping... press ctrl+c to exit

shut down hook task completed..

# How to perform single task by multiple threads?

|  |
| --- |
| If you have to perform single task by many threads, have only one run() method. **For example:** |

***Program of performing single task by multiple threads***

***class****TestMultitasking1****extends****Thread{*

***public******void****run(){*

*System.out.println("task one");*

*}*

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

*TestMultitasking1 t1=****new****TestMultitasking1();*

*TestMultitasking1 t2=****new****TestMultitasking1();*

*TestMultitasking1 t3=****new****TestMultitasking1();*

*t1.start();*

*t2.start();*

*t3.start();*

*}*

*}*

**Output:**

task one

task one

task one

***Program of performing single task by multiple threads***

***class****TestMultitasking2****implements****Runnable{*

***public******void****run(){*

*System.out.println("task one");*

*}*

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

*Thread t1 =****new****Thread(****new****TestMultitasking2());*

*//passing annonymous object ofTestMultitasking2 class*

*Thread t2 =****new****Thread(****new****TestMultitasking2());*

*t1.start();*

*t2.start();*

*}*

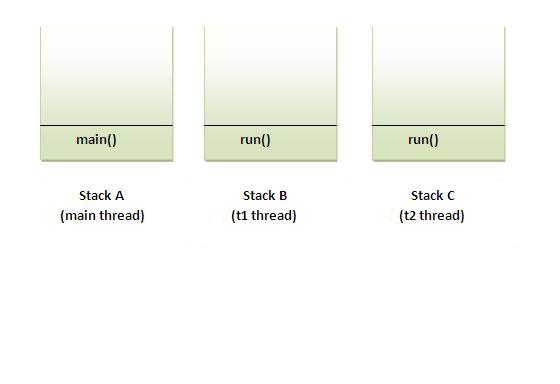
*}*

**Output:**

task one

task one

#### Note: Each thread run in a separate callstack.



**How to perform multiple tasks by multiple threads (multitasking in multithreading)?**

|  |
| --- |
| If you have to perform multiple tasks by multiple threads,have multiple run() methods.  **For example**: |

***Program of performing two tasks by two threads***

***class****Simple1****extends****Thread{*

***public******void****run(){*

*System.out.println("task one");*

*}*

*}*

***class****Simple2****extends****Thread{*

***public******void****run(){*

*System.out.println("task two");*

*}*

*}*

***class****TestMultitasking3{*

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

*Simple1 t1=****new****Simple1();*

*Simple2 t2=****new****Simple2();*

*t1.start();*

*t2.start();*

*}*

*}*

**Output:**

task one

task two

### Same example as above by annonymous class that extends Thread class:

***Program of performing two tasks by two threads***

***class****TestMultitasking4{*

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

*Thread t1=****new****Thread(){*

***public******void****run(){*

*System.out.println("task one");*

*}*

*};*

*Thread t2=****new****Thread(){*

***public******void****run(){*

*System.out.println("task two");*

*}*

*};*

*t1.start();*

*t2.start();*

*}*

*}*

**Output:**

task one

task two

### Same example as above by annonymous class that implements Runnable interface:

***Program of performing two tasks by two threads***

***class****TestMultitasking5{*

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

*Runnable r1=****new****Runnable(){*

***public******void****run(){*

*System.out.println("task one");*

*}*

*};*

*Runnable r2=****new****Runnable(){*

***public******void****run(){*

*System.out.println("task two");*

*}*

*};*

*Thread t1=****new****Thread(r1);*

*Thread t2=****new****Thread(r2);*

*t1.start();*

*t2.start();*

*}*

*}*

**Output**:

task one

task two

# Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### Advantage of Garbage Collection

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## How can an object be unreferenced?

**There are many ways:**

* By nulling the reference
* By assigning a reference to another
* By annonymous object etc.

### 1) By nulling a reference:

*Employee e=****new****Employee();*

*e=****null****;*

### 2) By assigning a reference to another:

*Employee e1=****new****Employee();*

*Employee e2=****new****Employee();*

*e1=e2;*

***//now the first object referred by e1 is available for garbage collection***

### 3) By annonymous object:

***new****Employee();*

## finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

***protected******void****finalize(){}*

#### Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

***public******static******void****gc(){}*

#### Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

### Simple Example of garbage collection in java

***public******class****TestGarbage1{*

***public******void****finalize(){*

*System.out.println("object is garbage collected");*

*}*

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

*TestGarbage1 s1=****new****TestGarbage1();*

*TestGarbage1 s2=****new****TestGarbage1();*

*s1=****null****;*

*s2=****null****;*

*System.gc();*

*}*

*}*

**Output:**

object is garbage collected

object is garbage collected

#### Note: Neither finalization nor garbage collection is guaranteed.

# Java Runtime class

**Java Runtime** class is used to interact with java runtime environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

## Important methods of Java Runtime class

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public static Runtime getRuntime() | returns the instance of Runtime class. |
| 2) | public void exit(int status) | terminates the current virtual machine. |
| 3) | public void addShutdownHook(Thread hook) | registers new hook thread. |
| 4) | public Process exec(String command)throws IOException | executes given command in a separate process. |
| 5) | public int availableProcessors() | returns no. of available processors. |
| 6) | public long freeMemory() | returns amount of free memory in JVM. |
| 7) | public long totalMemory() | returns amount of total memory in JVM. |

## Java Runtime exec() method

***public******class****Runtime1{*

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

*Runtime.getRuntime().exec("notepad");****//will open a new notepad***

*}*

*}*

## How to shutdown system in Java

You can use shutdown -s command to shutdown system. For windows OS, you need to provide full path of shutdown command

**Example:** c:\\Windows\\System32\\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system and -t switch to specify time delay.

***public******class****Runtime2{*

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

*Runtime.getRuntime().exec("shutdown -s -t 0");*

*}*

*}*

## How to shutdown windows system in Java

***public******class****Runtime2{*

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

*Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");*

*}*

*}*

## How to restart system in Java

***public******class****Runtime3{*

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

*Runtime.getRuntime().exec("shutdown -r -t 0");*

*}*

*}*

## Java Runtime availableProcessors()

***public******class****Runtime4{*

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

*System.out.println(Runtime.getRuntime().availableProcessors());*

*}*

*}*

## Java Runtime freeMemory() and totalMemory() method

In the given program, after creating 10000 instance, free memory will be less than the previous free memory. But after gc() call, you will get more free memory.

***public******class****MemoryTest{*

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

*Runtime r=Runtime.getRuntime();*

*System.out.println("Total Memory: "+r.totalMemory());*

*System.out.println("Free Memory: "+r.freeMemory());*

***for****(****int****i=0;i<10000;i++){*

***new****MemoryTest();*

*}*

*System.out.println("After creating 10000 instance, Free Memory: "+r.freeMemory());*

*System.gc();*

*System.out.println("After gc(), Free Memory: "+r.freeMemory());*

*}*

*}*

Total Memory: 100139008

Free Memory: 99474824

After creating 10000 instance, Free Memory: 99310552

After gc(), Free Memory: 100182832

# 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.

### Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

### 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)

### Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

### 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.

### Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent.

**Let's see the example:**

*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(){*

*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

10

200

15

300

20

400

25

500

### Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

**//Example of java synchronized method**

***class****Table{*

***synchronized******void****printTable(****int****n){//synchronized method*

***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(){*

*t.printTable(5);*

*}*

*}*

***class****MyThread2****extends****Thread{*

*Table t;*

*MyThread2(Table t){*

***this****.t=t;*

*}*

***public******void****run(){*

*t.printTable(100);*

*}*

*}*

***public******class****TestSynchronization2{*

***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

10

15

20

25

100

200

300

400

500

### Example of synchronized method by using annonymous class

In this program, we have created the two threads by annonymous class, so less coding is required.

**//Program of synchronized method by using annonymous class**

***class****Table{*

***synchronized******void****printTable(****int****n){//synchronized method*

***for****(****int****i=1;i<=5;i++){*

*System.out.println(n\*i);*

***try****{*

*Thread.sleep(400);*

*}****catch****(Exception e){*

*System.out.println(e);*

*}*

*}*

*}*

*}*

***public******class****TestSynchronization3{*

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

***final****Table obj =****new****Table();//only one object*

*Thread t1=****new****Thread(){*

***public******void****run(){*

*obj.printTable(5);*

*}*

*};*

*Thread t2=****new****Thread(){*

***public******void****run(){*

*obj.printTable(100);*

*}*

*};*

*t1.start();*

*t2.start();*

*}*

*}*

**Output:**

5

10

15

20

25

100

200

300

400

500

# Synchronized block in java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

### Points to remember for Synchronized block

* Synchronized block is used to lock an object for any shared resource.
* Scope of synchronized block is smaller than the method.

**Syntax to use synchronized block**

***synchronized****(object reference expression) {*

*//code block*

*}*

### Example of synchronized block

Let's see the simple example of synchronized block.

***Program of synchronized block***

***class****Table{*

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

***synchronized****(****this****){//synchronized block*

***for****(****int****i=1;i<=5;i++){*

*System.out.println(n\*i);*

***try****{*

*Thread.sleep(400);*

*}****catch****(Exception e){*

*System.out.println(e);*

*}*

*}*

*}*

*}//end of the method*

*}*

***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);*

*}*

*}*

***public******class****TestSynchronizedBlock1{*

***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

10

15

20

25

100

200

300

400

500

### Same Example of synchronized block by using annonymous class:

***//Program of synchronized block by using annonymous class***

***class****Table{*

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

***synchronized****(****this****){//synchronized block*

***for****(****int****i=1;i<=5;i++){*

*System.out.println(n\*i);*

***try****{*

*Thread.sleep(400);*

*}****catch****(Exception e){*

*System.out.println(e);*

*}*

*}*

*}*

*}//end of the method*

*}*

***public******class****TestSynchronizedBlock2{*

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

***final****Table obj =****new****Table();//only one object*

*Thread t1=****new****Thread(){*

***public******void****run(){*

*obj.printTable(5);*

*}*

*};*

*Thread t2=****new****Thread(){*

***public******void****run(){*

*obj.printTable(100);*

*}*

*};*

*t1.start();*

*t2.start();*

*}*

*}*

**Output:**

5

10

15

20

25

100

200

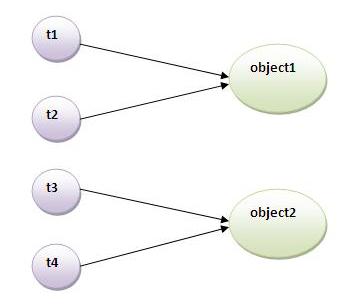
300

400

500

# Static synchronization

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



### Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

### Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

***class****Table{*

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

***for****(****int****i=1;i<=10;i++){*

*System.out.println(n\*i);*

***try****{*

*Thread.sleep(400);*

*}****catch****(Exception e){*

*}*

*}*

*}*

*}*

***class****MyThread1****extends****Thread{*

***public******void****run(){*

*Table.printTable(1);*

*}*

*}*

***class****MyThread2****extends****Thread{*

***public******void****run(){*

*Table.printTable(10);*

*}*

*}*

***class****MyThread3****extends****Thread{*

***public******void****run(){*

*Table.printTable(100);*

*}*

*}*

***class****MyThread4****extends****Thread{*

***public******void****run(){*

*Table.printTable(1000);*

*}*

*}*

***public******class****TestSynchronization4{*

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

*MyThread1 t1=****new****MyThread1();*

*MyThread2 t2=****new****MyThread2();*

*MyThread3 t3=****new****MyThread3();*

*MyThread4 t4=****new****MyThread4();*

*t1.start();*

*t2.start();*

*t3.start();*

*t4.start();*

*}*

*}*

**Output:**

1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

### Same example of static synchronization by annonymous class

In this example, we are using annonymous class to create the threads.

***class****Table{*

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

***for****(****int****i=1;i<=10;i++){*

*System.out.println(n\*i);*

***try****{*

*Thread.sleep(400);*

*}****catch****(Exception e){*

*}*

*}*

*}*

*}*

***public******class****TestSynchronization5 {*

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

*Thread t1=****new****Thread(){*

***public******void****run(){*

*Table.printTable(1);*

*}*

*};*

*Thread t2=****new****Thread(){*

***public******void****run(){*

*Table.printTable(10);*

*}*

*};*

*Thread t3=****new****Thread(){*

***public******void****run(){*

*Table.printTable(100);*

*}*

*};*

*Thread t4=****new****Thread(){*

***public******void****run(){*

*Table.printTable(1000);*

*}*

*};*

*t1.start();*

*t2.start();*

*t3.start();*

*t4.start();*

*}*

*}*

**Output:**

1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

### 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*

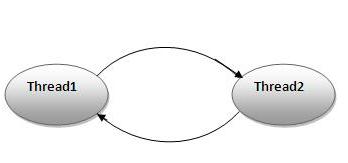
*// ...*

*}*

*}*

# 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();*

*}*

*}*

**Output:**

Thread 1: locked resource 1

Thread 2: locked resource 2

# 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.

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

|  |  |
| --- | --- |
| **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. |

### 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()*

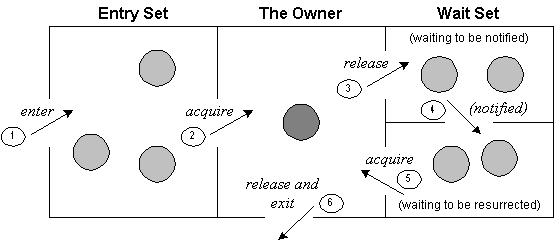
### 3) notifyAll() method

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

**Syntax:**

*public final void notifyAll()*

### Understanding the process of inter-thread communication



**The point to point explanation of the above diagram is as follows:**

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

### 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.

### 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 |
| 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

# 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);*

*}*

*}*

*}*

**Output:**

Exception in thread-0

java.lang.RuntimeException: Thread interrupted...

java.lang.InterruptedException: sleep interrupted

at A.run(A.java:7)

## Example of interrupting a thread that doesn't stop working

|  |
| --- |
| In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working. |

***class****TestInterruptingThread2****extends****Thread{*

***public******void****run(){*

***try****{*

*Thread.sleep(1000);*

*System.out.println("task");*

*}****catch****(InterruptedException e){*

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

*}*

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

*}*

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

*TestInterruptingThread2 t1=****new****TestInterruptingThread2();*

*t1.start();*

*t1.interrupt();*

*}*

*}*

**Output:**

Exception handled

java.lang.InterruptedException: sleep interrupted

thread is running...

## Example of interrupting thread that behaves normally

|  |
| --- |
| If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later. |

***class****TestInterruptingThread3****extends****Thread{*

***public******void****run(){*

***for****(****int****i=1;i<=5;i++)*

*System.out.println(i);*

*}*

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

*TestInterruptingThread3 t1=****new****TestInterruptingThread3();*

*t1.start();*

*t1.interrupt();*

*}*

*}*

**Output:**

1

2

3

4

5

## What about isInterrupted and interrupted method?

|  |
| --- |
| The isInterrupted() method returns the interrupted flag either true or false. The static interrupted() method returns the interrupted flag afterthat it sets the flag to false if it is true. |

***public******class****TestInterruptingThread4****extends****Thread{*

***public******void****run(){*

***for****(****int****i=1;i<=2;i++){*

***if****(Thread.interrupted()){*

*System.out.println("code for interrupted thread");*

*}****else****{*

*System.out.println("code for normal thread");*

*}*

*}//end of for loop*

*}*

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

*TestInterruptingThread4 t1=****new****TestInterruptingThread4();*

*TestInterruptingThread4 t2=****new****TestInterruptingThread4();*

*t1.start();*

*t1.interrupt();*

*t2.start();*

*}*

*}*

**Output:**

Code for interrupted thread

code for normal thread

code for normal thread

code for normal thread

# Reentrant Monitor in Java

According to Sun Microsystems, **Java monitors are reentrant** means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

#### Advantage of Reentrant Monitor

It eliminates the possibility of single thread deadlocking

Let's understand the java reentrant monitor by the example given below:

***class****Reentrant {*

***public******synchronized******void****m() {*

*n();*

*System.out.println("this is m() method");*

*}*

***public******synchronized******void****n() {*

*System.out.println("this is n() method");*

*}*

*}*

In this class, m and n are the synchronized methods. The m() method internally calls the n() method.

Now let's call the m() method on a thread. In the class given below, we are creating thread using annonymous class.

***public******class****ReentrantExample{*

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

***final****ReentrantExample re=****new****ReentrantExample();*

*Thread t1=****new****Thread(){*

***public******void****run(){*

*re.m();//calling method of Reentrant class*

*}*

*};*

*t1.start();*

*}*

*}*

**Output:**

this is n() method

this is m() method