Threads - 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

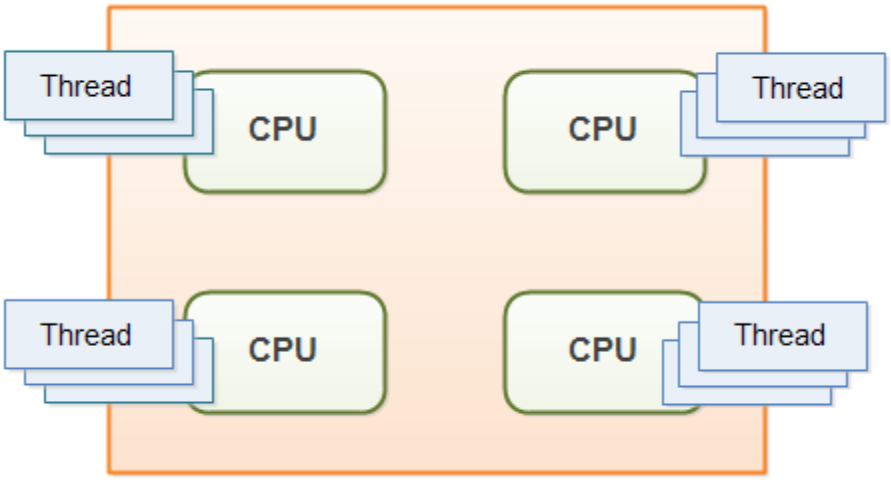
1) It doesn't block the user because threads are independent and you can perform multiple operations at same time.

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.

Any application can have multiple process (instances). Each of this process can be assigned either as a single thread or multiple threads.

- **Single Thread**: A single thread is basically a lightweight and the smallest unit of processing.
- **Multi-Thread**: While multithreading can be defined as the execution of two or more threads concurrently.



Better resource utilization

Imagine an application that reads and processes files from the local file system. Let's say that reading af file from disk takes 5 seconds and processing it takes 2 seconds. Processing two files then takes

```
5 seconds reading file A
2 seconds processing file A
5 seconds reading file B
2 seconds processing file B
```

```
----------------------
 14 seconds total
```

When reading the file from disk most of the CPU time is spent waiting for the disk to read the data. The CPU is pretty much idle during that time. It could be doing something else. By changing the order of the operations, the CPU could be better utilized. Look at this ordering:

```
  5 seconds reading file A
  5 seconds reading file B + 2 seconds processing file A
  2 seconds processing file B
----------------------
 12 seconds total
```

The CPU waits for the first file to be read. Then it starts the read of the second file. While the second file is being read, the CPU processes the first file. Remember, while waiting for the file to be read from disk, the CPU is mostly idle.

In general, the CPU can be doing other things while waiting for IO. It doesn't have to be disk IO. It can be network IO as well, or input from a user at the machine. Network and disk IO is often a lot slower than CPU's and memory IO.

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:

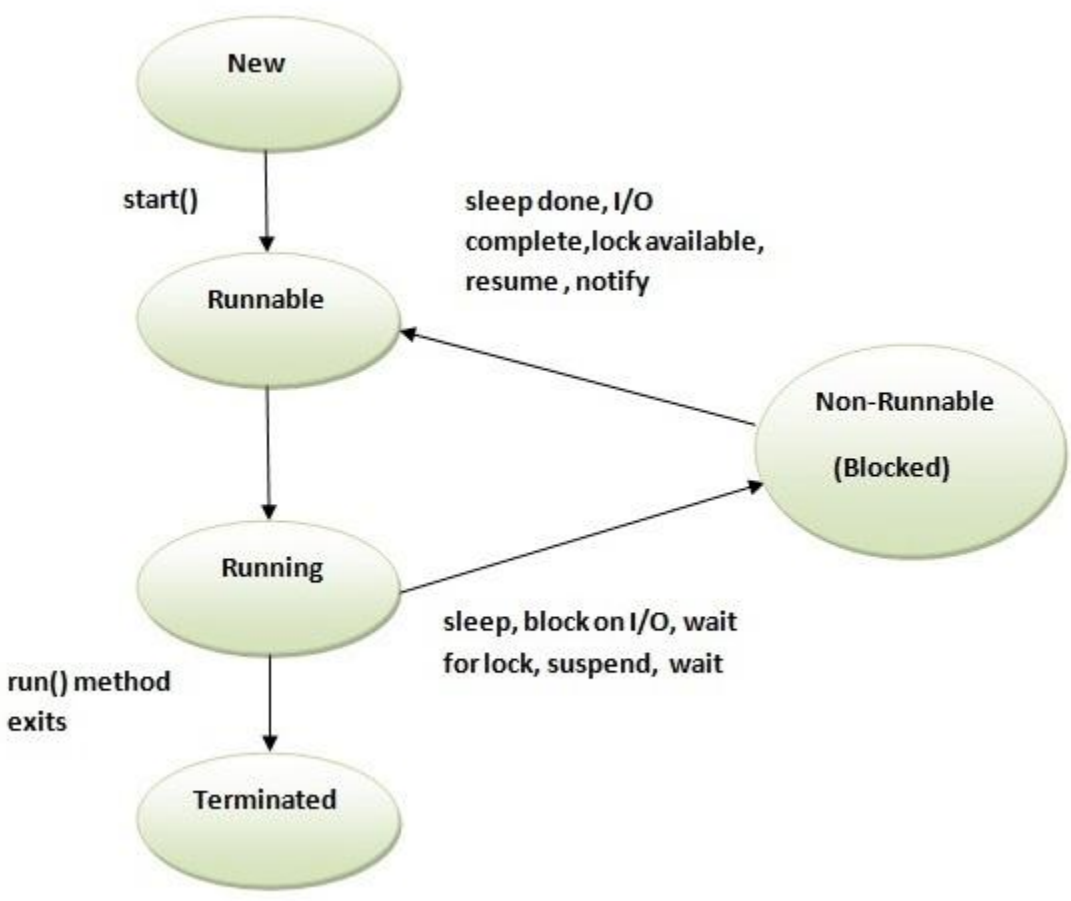
New

Runnable

Running

Non-Runnable (Blocked)

Terminated



How to create thread

There are two ways to create a thread:

By extending Thread class

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:

public void run(): is used to perform action for a thread.

public void start(): starts the execution of the thread.JVM calls the run() method on the thread.

public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public void join(): waits for a thread to die.

public void join(long miliseconds): waits for a thread to die for the specified miliseconds.

public int getPriority(): returns the priority of the thread.

public int setPriority(int priority): changes the priority of the thread.

public String getName(): returns the name of the thread.

public void setName(String name): changes the name of the thread.

public Thread currentThread(): returns the reference of currently executing thread.

public int getId(): returns the id of the thread.

public Thread.State getState(): returns the state of the thread.

public boolean isAlive(): tests if the thread is alive.

public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

public void suspend(): is used to suspend the thread(depricated).

public void resume(): is used to resume the suspended thread(depricated).

public void stop(): is used to stop the thread(depricated).

public boolean isDaemon(): tests if the thread is a daemon thread.

public void setDaemon(boolean b): marks the thread as daemon or user thread.

public void interrupt(): interrupts the thread.

public boolean isInterrupted(): tests if the thread has been interrupted.

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 explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

The most common difference is:

When you extend Thread class, you can’t extend any other class which you require. (As you know, Java does not allow inheriting more than one class). When you implement Runnable, you can save a space for your class to extend any other class in future or now.

However, the significant difference is.

When you extends Thread class, each of your thread creates unique object and associate with it. When you implements Runnable, it shares the same object to multiple threads.

If your class is *extending the Thread class* then it becomes a single thread which inherits the properties *Thread class*, so it'll be heavy.
(When *extending Thread class* each of the threads creates unique object and associate with it, but when *implementing Runnable*, it shares the same object to multiple Threads).

If your class is *Implementing the Runnable interface* then you only override the `run()` .So this instance creates a separate `Thread` and every individual Thread runs separately but not as a single heavy Thread in your program. Another thing, Since `Java` does not support *multiple inheritance*, if you *implement the Runnable* you'll avoid problems of multiple extending, so if you implement *Runnable interface* you can extend any class that you are required other than *Thread*class.

Thread is a block of code which can execute concurrently with other threads in the JVM. You can create and run a thread in either ways; Extending **Thread** class, Implementing **Runnable** interface.

Both approaches do the same job but there have been some differences. Almost everyone have this question in their minds: *which one is best to use?* We will see the answer at the end of this post.

The most common difference is

> When you **extends Thread** class, after that you can't extend any other class which you required. (As you know, Java does not allow inheriting more than one class).
>
> When you **implements Runnable**, you can save a space for your class to extend any other class in future or now.

However, the significant difference is.

> When you **extends Thread** class, each of your thread

creates unique object and associate with it.

When you **implements Runnable**, it shares the same object to multiple threads.

The following example helps you to understand more clearly.

ThreadVsRunnable.java

```
package com.t;

public class TestThreadExtendRunnable {

	public static void main(String args[]) throws
Exception {
		// Multiple threads share the same object.
		ImplementsRunnable rc = new
ImplementsRunnable();
		Thread t1 = new Thread(rc);
		t1.start();
		Thread.sleep(1000); // Waiting for 1 second
before starting next thread
		Thread t2 = new Thread(rc);
		t2.start();
		Thread.sleep(1000); // Waiting for 1 second
before starting next thread
		Thread t3 = new Thread(rc);
		t3.start();

		// Creating new instance for every thread
access.
		ExtendsThread tc1 = new ExtendsThread();
		tc1.start();
```

```
            Thread.sleep(1000); // Waiting for 1 second
before starting next thread
            ExtendsThread tc2 = new ExtendsThread();
            tc2.start();
            Thread.sleep(1000); // Waiting for 1 second
before starting next thread
            ExtendsThread tc3 = new ExtendsThread();
            tc3.start();
      }

}

class ImplementsRunnable implements Runnable {

      private int counter = 0;

      public void run() {
            counter++;
            System.out.println("ImplementsRunnable :
Counter : " + counter);
            foo();
      }

      void foo() {
            int ii = 23;
            System.out.println(ii);
            ii++;
            System.out.println(ii);

      }
}

class ExtendsThread extends Thread {
```

```
	private int counter = 0;

	public void run() {
		counter++;
		System.out.println("ExtendsThread : Counter :
" + counter);
		foo();
	}

	void foo() {
		int ii = 22;
		System.out.println(22);
		ii++;
		System.out.println(ii);
	}
}
```

Output of the above program.

```
ImplementsRunnable : Counter : 1
23
24
ImplementsRunnable : Counter : 2
23
24
ImplementsRunnable : Counter : 3
23
24
ExtendsThread : Counter : 1
22
```

```
23
ExtendsThread : Counter : 1
22
23
ExtendsThread : Counter : 1
22
23
```

In the Runnable interface approach, only one instance of a class is being created and it has been shared by different threads. So the value of **counter** is incremented for each and every thread access.

Whereas, Thread class approach, you must have to create separate instance for every thread access. Hence different memory is allocated for every class instances and each has separate **counter**, the value remains same, which means no increment will happen because none of the object reference is same.

When to use Runnable?
Use Runnable interface when you want to access the same resource from the group of threads. Avoid using Thread class here, because multiple objects creation consumes more memory and it becomes a big performance overhead.

Apart from this, object oriented designs have some guidelines for better coding.

> Coding to an interface rather than to implementation. This makes your software/application easier to extend. In other words, your code will work with all the interface's subclasses, even ones that have not been

created yet.

Interface inheritance (implements) is preferable – This makes your code is loosely coupling between classes/objects.(Note : Thread class internally implements the Runnable interface)

Example: coding to an interface.

```
Map subject = new HashMap();
```

Assigning HashMap object to interface *Map*, suppose in future if you want to change HashMap to Hashtable or LinkedHashMap you can simple change in the declaration part is enough rather than to all the usage places. This point has been elaborately explained here.

Which one is best to use?

Ans : Very simple, based on your application requirements you will use this appropriately. But I would suggest, try to use interface inheritance i.e., implements Runnable.

Thread Safety and Shared Resources

Code that is safe to call by multiple threads simultaneously is called *thread safe*. If a piece of code is thread safe, then it contains no **race conditions**. Race condition only occur when multiple threads update shared resources. Therefore it is important to know what resources Java threads share when executing.

Local Variables

Local variables are stored in each thread's own stack. That means that local variables are never shared between threads. That also means that all local primitive variables are thread safe. Here is an example of a thread safe local primitive variable:

```
public void someMethod(){

  long threadSafeInt = 0;

  threadSafeInt++;
}
```

Local Object References

Local references to objects are a bit different. The reference itself is not shared. The object referenced however, is not stored in each threads's local stack. All objects are stored in the shared heap.

If an object created locally never escapes the method it was created in, it is thread safe. In fact you can also pass it on to other methods and objects as long as none of these methods or objects make the passed object available to other threads.

Here is an example of a thread safe local object:

```
public void someMethod(){
```

```
  LocalObject localObject = new LocalObject();

  localObject.callMethod();
  method2(localObject);
}

public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

The LocalObject instance in this example is not returned from the method, nor is it passed to any other objects that are accessible from outside the someMethod() method. Each thread executing the someMethod() method will create its own LocalObject instance and assign it to the localObject reference. Therefore the use of the LocalObject here is thread safe.

In fact, the whole method someMethod() is thread safe. Even if the LocalObject instance is passed as parameter to other methods in the same class, or in other classes, the use of it is thread safe.

The only exception is of course, if one of the methods called with the LocalObject as parameter, stores the LocalObject instance in a way that allows access to it from other threads.

Object Member Variables

Object member variables (fields) are stored on the heap

along with the object. Therefore, if two threads call a method on the same object instance and this method updates object member variables, the method is not thread safe. Here is an example of a method that is not thread safe:

```
public class NotThreadSafe{
   StringBuilder builder = new StringBuilder();

   public add(String text){
      this.builder.append(text);
   }
}
```

If two threads call the add() method simultaneously **on the same NotThreadSafe instance** then it leads to race conditions. For instance:

```
NotThreadSafe sharedInstance = new NotThreadSafe();

new Thread(new MyRunnable(sharedInstance)).start();
new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{
  NotThreadSafe instance = null;

  public MyRunnable(NotThreadSafe instance){
    this.instance = instance;
  }

  public void run(){
```

```
    this.instance.add("some text");
  }
}
```

Notice how the two MyRunnable instances share the same NotThreadSafe instance. Therefore, when they call the add() method on the NotThreadSafe instance it leads to race condition.

However, if two threads call the add() method simultaneously **on different instances** then it does not lead to race condition. Here is the example from before, but slightly modified:

```
new Thread(new MyRunnable(new
NotThreadSafe())).start();
new Thread(new MyRunnable(new
NotThreadSafe())).start();
```

Now the two threads have each their own instance of NotThreadSafe so their calls to the add method doesn't interfere with each other. The code does not have race condition anymore. So, even if an object is not thread safe it can still be used in a way that doesn't lead to race condition.

The Thread Control Escape Rule

When trying to determine if your code's access of a certain resource is thread safe you can use the thread control escape rule:

```
If a resource is created, used and disposed within
the control of the same thread,
and never escapes the control of this thread,
the use of that resource is thread safe.
```

Resources can be any shared resource like an object, array, file, database connection, socket etc. In Java you do not always explicitly dispose objects, so "disposed" means losing or null'ing the reference to the object.

Even if the use of an object is thread safe, if that object points to a shared resource like a file or database, your application as a whole may not be thread safe. For instance, if thread 1 and thread 2 each create their own database connections, connection 1 and connection 2, the use of each connection itself is thread safe. But the use of the database the connections point to may not be thread safe. For example, if both threads execute code like this:

```
check if record X exists
if not, insert record X
```

If two threads execute this simultaneously, and the record X they are checking for happens to be the same record, there is a risk that both of the threads end up inserting it. This is how:

```
Thread 1 checks if record X exists. Result = no
Thread 2 checks if record X exists. Result = no
Thread 1 inserts record X
```

Thread 2 inserts record X

This could also happen with threads operating on files or other shared resources. Therefore it is important to distinguish between whether an object controlled by a thread **is** the resource, or if it merely **references**the resource (like a database connection does).

Java Memory Model

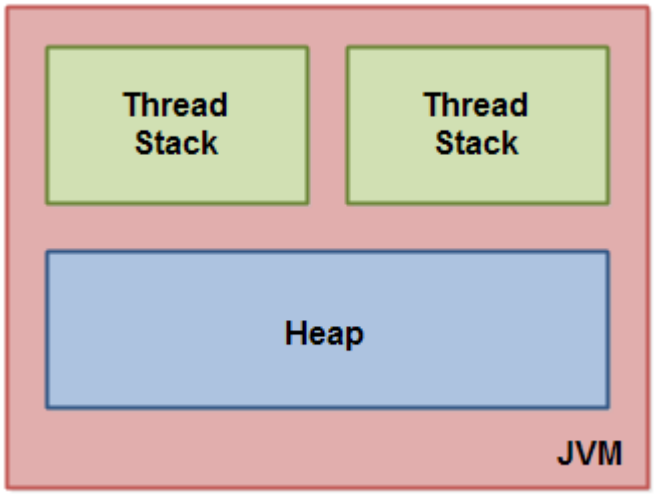
The Java memory model specifies how the Java virtual machine works with the computer's memory (RAM). The Java virtual machine is a model of a whole computer so this model naturally includes a memory model - AKA the Java memory model.

It is very important to understand the Java memory model if you want to design correctly behaving concurrent programs. The Java memory model specifies how and when different threads can see values written to shared variables by other threads, and how to synchronize access to shared variables when necessary.

The original Java memory model was insufficient, so the Java memory model was revised in Java 1.5. This version of the Java memory model is still in use in Java 8.

The Internal Java Memory Model

The Java memory model used internally in the JVM divides memory between thread stacks and the heap. This diagram illustrates the Java memory model from a logic perspective:



Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution. I will refer to this as the "call stack". As the thread executes its code, the call stack changes.

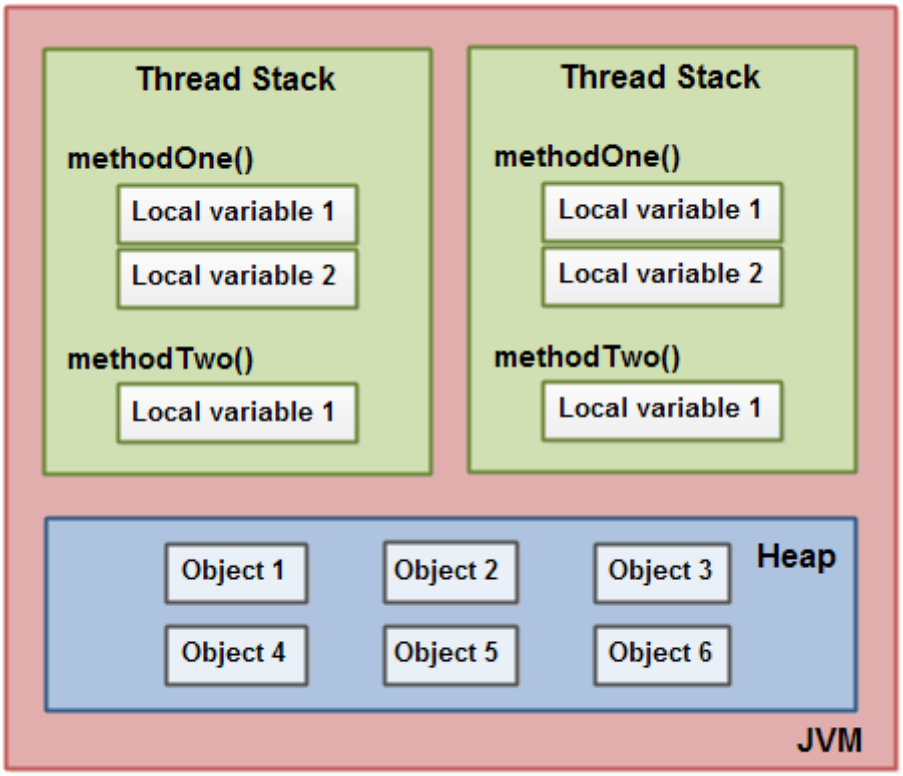
The thread stack also contains all local variables for each method being executed (all methods on the call stack). A thread can only access it's own thread stack. Local variables created by a thread are invisible to all other threads than the thread who created it. Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack. Thus, each thread has its own version of each local variable.

All local variables of primitive types
(boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a pritimive variable to another thread, but it cannot share

the primitive local variable itself.

The heap contains all objects created in your Java application, regardless of what thread created the object. This includes the object versions of the primitive types (e.g. Byte, Integer, Long etc.). It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.

Here is a diagram illustrating the call stack and local variables stored on the thread stacks, and objects stored on the heap:



A local variable may be of a primitive type, in which case it is totally kept on the thread stack.

A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself if stored on the heap.

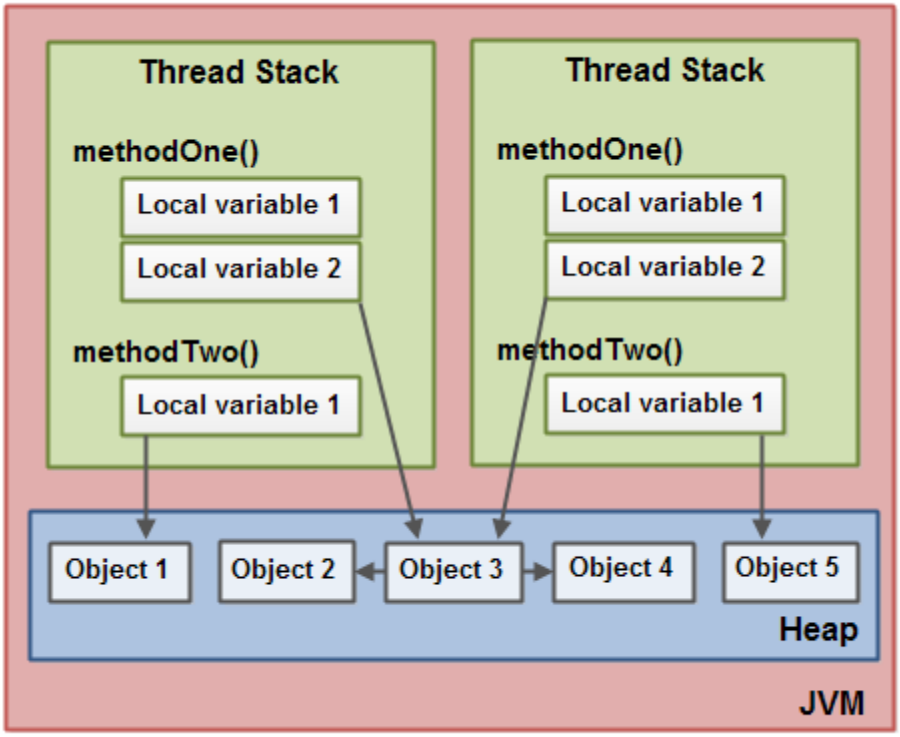
An object may contain methods and these methods may contain local variables. These local variables are also stored on the thread stack, even if the object the method belongs to is stored on the heap.

An object's member variables are stored on the heap along with the object itself. That is true both when the member variable is of a primitive type, and if it is a reference to an object.

Static class variables are also stored on the heap along with the class definition.

Objects on the heap can be accessed by all threads that have a reference to the object. When a thread has access to an object, it can also get access to that object's member variables. If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will have its own copy of the local variables.

Here is a diagram illustrating the points above:



Two threads have a set of local variables. One of the local variables (Local Variable 2) point to a shared object on the heap (Object 3). The two threads each have a different reference to the same object. Their references are local variables and are thus stored in each thread's thread stack (on each). The two different references point to the same object on the heap, though.

Notice how the shared object (Object 3) has a reference to Object 2 and Object 4 as member variables (illustrated by the arrows from Object 3 to Object 2 and Object 4). Via these member variable references in Object 3 the two threads can access Object 2 and Object 4.

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();
```

```
}
}
```

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

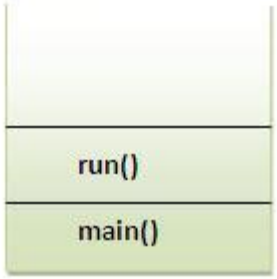
What if we call run() method directly instead start() method?

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...



Stack
(main thread)

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(InterruptedExcepti
      on 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.

The join() method is used to hold the execution of currently running thread until the specified thread is dead(finished execution). In this tutorial we will discuss the purpose and use of join() method with examples.

Why we use join() method?

In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads.
For example lets have a look at the following code:

Without using join()

Here we have three threads th1, th2 and th3. Even though we have started the threads in a sequential manner the thread scheduler does not start and end them in the specified order. Everytime you run this code,

you may get a different result each time. **So the question is: How can we make sure that the threads executes in a particular order. The Answer is: By using join() method appropriately.**

```
public class JoinExample2 {
   public static void main(String[] args) {
      Thread th1 = new Thread(new MyClass2(), "th1");
      Thread th2 = new Thread(new MyClass2(), "th2");
      Thread th3 = new Thread(new MyClass2(), "th3");

      th1.start();
      th2.start();
      th3.start();
   }
}

class MyClass2 implements Runnable{

    @Override
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("Thread started:
"+t.getName());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Thread ended: "+t.getName());
    }
}
```

Output:

```
Thread started: th1
Thread started: th3
Thread started: th2
Thread ended: th1
Thread ended: th3
Thread ended: th2
```

Lets have a look at the another code where we are using the join() method.

The same example with join()

Lets say our requirement is to execute them in the order of first, second and third. We can do so by using join() method appropriately.

```
public class JoinExample {
   public static void main(String[] args) {
      Thread th1 = new Thread(new MyClass(), "th1");
      Thread th2 = new Thread(new MyClass(), "th2");
      Thread th3 = new Thread(new MyClass(), "th3");

      // Start first thread immediately
      th1.start();

      /* Start second thread(th2) once first thread(th1)
       * is dead
       */
      try {
         th1.join();
      } catch (InterruptedException ie) {
```

```
            ie.printStackTrace();
          }
        th2.start();

        /* Start third thread(th3) once second thread(th2)
         * is dead
         */
        try {
            th2.join();
        } catch (InterruptedException ie) {
             ie.printStackTrace();
          }
        th3.start();

        // Displaying a message once third thread is dead
        try {
            th3.join();
        } catch (InterruptedException ie) {
              ie.printStackTrace();
          }
        System.out.println("All three threads have finished
execution");
    }
}

class MyClass implements Runnable{

    @Override
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("Thread started:
"+t.getName());
        try {
```

```
            Thread.sleep(4000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Thread ended: "+t.getName());

    }
}
```

Output:

```
Thread started: th1
Thread ended: th1
Thread started: th2
Thread ended: th2
Thread started: th3
Thread ended: th3
All three threads have finished execution
```

In this example we have used the join() method in such a way that our threads execute in the specified order.

We can prevent a thread from execution by using any of the 3 methods of Thread class:

1. yield()
2. join()
3. sleep()

1. 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. The yielded thread when it will

get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent.

2. join() If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.
3. sleep() Based on our requirement we can make a thread to be in sleeping state for a specified period of time (hope not much explanation required for our favorite method).

`sleep()` causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).
`yield()` basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should. Otherwise, the current thread will continue

---

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.ge
tName());
}
}
```

```
Output:Name of t1:Thread-0
      Name of t2:Thread-1
      id of t1:8
      running...
      After changling 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. But we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

public String getName(): is used to return the name of a thread.

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.ge
tName());
}
}
```

```
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 have 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("running 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
```

Thread with highest priority will get execution chance prior to other threads. Suppose there are 3 threads t1, t2 and t3 with priorities 4, 6 and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.

.

```
// Java program to demonstrat ethat a child thread
// gets same priority as parent
import java.lang.*;

class ThreadDemo extends Thread
{
   public void run()
   {
      System.out.println("Inside run method");
   }

   public static void main(String[]args)
   {
      // main thread priority is 6 now
      Thread.currentThread().setPriority(6);

      System.out.println("main thread priority : " +
               Thread.currentThread().getPriority());
```

```
        ThreadDemo t1 = new ThreadDemo();

        // t1 thread is child of main thread
        // so t1 thread will also have priority 6.
        System.out.println("t1 thread priority : "
                           + t1.getPriority());
    }
}
```

- Run on IDE
- Output:
- Main thread priority : 6
- t1 thread priority : 6

- If two threads have same priority then we can't expect which thread will execute first. It depends on thread scheduler's algorithm(Round-Robin, First Come First Serve, etc)
- If we are using thread priority for thread scheduling then we should always keep in mind that underlying platform should provide support for scheduling based on thread priority.

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 cur daemon thread or user t
2)	public boolean isDaemon()	is used to check th 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
```

Difference between Daemon threads and non-Daemon thread (user thread)

The main difference between Daemon thread and user threads is that the JVM does not wait for Daemon thread before exiting while it waits for user threads, it does not exit until unless all the user threads finish their execution.

Daemon thread in java can be useful to run some tasks in background. When we create a thread in java, by default it's a user thread and if it's running JVM will not terminate the program.

Daemon thread in java

When a thread is marked as daemon thread, JVM doesn't wait it to finish to terminate the program. As soon as all the user threads are finished, JVM terminates the program as well as all the associated daemon threads.

`Thread.setDaemon(true)` is used to create a daemon thread in java. This method should be invoked before the

thread is started otherwise it will throw `IllegalThreadStateException`.

We can check if a thread is daemon thread or not by calling `isDaemon()` method on it.

Another point is that when a thread is started, it inherits the daemon status of it's parent thread.

Daemon Thread in Java Example

Let's see a small example of daemon thread in java.

```
package com.journaldev.threads;


public class JavaDaemonThread {


    public static void main(String[] args) throws
InterruptedException {
        Thread dt = new Thread(new DaemonThread(),
"dt");
        dt.setDaemon(true);
        dt.start();
        //continue program
        Thread.sleep(30000);
        System.out.println("Finishing program");
```

```
    }

}

class DaemonThread implements Runnable{

    @Override
    public void run() {
        while(true){
            processSomething();
        }
    }

    private void processSomething() {
        try {
            System.out.println("Processing daemon
thread");
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }


}
```

When we execute above daemon thread program, JVM creates first user thread with `main()` method and then a daemon thread.

When main method is finished, the program terminates and daemon thread is also shut down by JVM.

Below image shows the output of the above program.

JavaDaemonThread.java

```
1  package com.journaldev.threads;
2
3  public class JavaDaemonThread {
4
5      public static void main(String[] args) throws InterruptedExcept
6          Thread dt = new Thread(new DaemonThread(), "dt");
7          dt.setDaemon(true);
8          dt.start();
9          // continue program
10         Thread.sleep(30000);
11         System.out.println("Finishing program");
12     }
13
14 }
15
16 class DaemonThread implements Runnable {
17
18     @Override
19     public void run() {
20         while (true) {
21             processSomething();
22         }
```

Problems @ Javadoc Declaration Console

<terminated> JavaDaemonThread [Java Application] /Library/Java/JavaVirtualMachines/jdk

```
Processing daemon thread
Processing daemon thread
Processing daemon thread
Processing daemon thread
Processing daemon thread
Processing daemon thread
Finishing program
```

If we don’t set the “dt” thread to be run as daemon thread, the program will never terminate even after main

thread is finished it's execution. Notice that DaemonThread is having a while true loop with thread sleep, so it will never terminate on it's own.

Try to comment the statement to set thread as daemon thread and run the program. You will notice that program runs indefinitely and you will have to manually quit it.

Daemon Thread Usage

Usually we create a daemon thread for functionalities that are not critical to system. For example logging thread or monitoring thread to capture the system resource details and their state. If you are not okay will a thread being terminated, don't create it as a daemon thread.

Also it's better to avoid daemon threads for IO operations because it can cause resource leak when program just terminates and resources are not closed properly.

// Daemon Thread Eclipse program

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().g
            etName()+" (Start) message = "+message);

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

            System.out.println(Thread.currentThread().g
            etName()+" (End)");//prints thread name
      }
      private void processmessage() {
            try {  Thread.sleep(2000);  } catch (Interru
            ptedException e) { e.printStackTrace(); }
      }
}
```

File: JavaThreadPoolExample.java

```
public class TestThreadPool {
      public static void main(String[] args) {
            ExecutorService executor = Executors.newFi
            xedThreadPool(5);//creating a pool of 5 thre
            ads
            for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i
            );
            executor.execute(worker);//calling execute
            method of ExecutorService
            }
            executor.shutdown();
            // Initiates an orderly shutdown in which
            previously submitted tasks are executed, but
            no // new tasks will be accepted. Invocation
            has no additional effect if already shut down.
```

```
            while (!executor.isTerminated()) {   }
// Returns true if all tasks have completed following
shut down. Note that isTerminated is never true
 // unless either shutdown or shutdownNow was called
first.

            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

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 co
            mpleted..");
      }
}

public class TestShutdown1{
public static void main(String[] args)throws Excepti
on {

Runtime r=Runtime.getRuntime();
r.addShutdownHook(new MyThread());

System.out.println("Now main sleeping... press ctrl+c t
o 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 Excepti
on {

Runtime r=Runtime.getRuntime();
  // Example of anonymous Thread
r.addShutdownHook(new Thread(){
public void run(){
      System.out.println("shut down hook task complete
      d..");
      }
}
);

System.out.println("Now main sleeping... press ctrl+c t
o 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 extending Thread

```
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
```

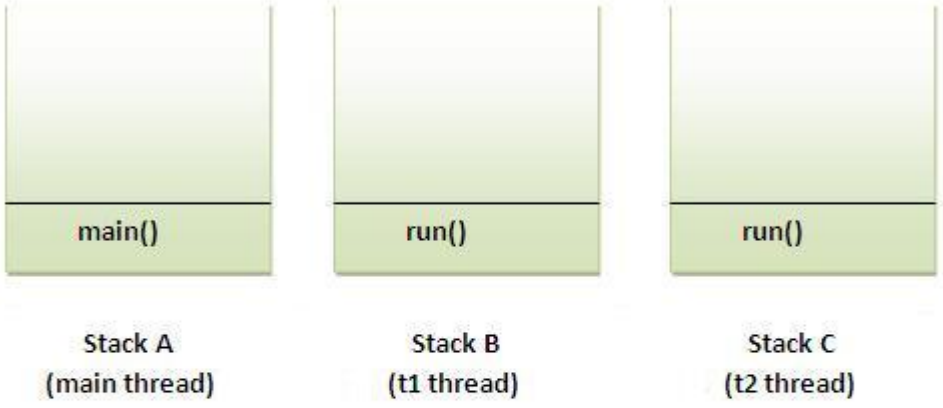
Program of performing single task by multiple threads using Runnable

```
class TestMultitasking2 implements Runnable{
public void run(){
```

```
System.out.println("task one");
}

public static void main(String args[]){
Thread t1 =new Thread(new TestMultitasking2());//p
assing annonymous object of TestMultitasking2 class
Thread t2 =new Thread(new TestMultitasking2());

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

}
}
```

Output:task one
task one

Note: Each thread runs 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 anonymous 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
```

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

To prevent thread interference.

To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

Process Synchronization

Thread Synchronization

Thread vs Process

1) A program in execution is often referred as process. A thread is a subset(part) of the process.

2) A process consists of multiple threads. A thread is a smallest part of the process that can execute concurrently with other parts(threads) of the process.

3) A process is sometime referred as task. A thread is often referred as lightweight process.

4) A process has its own address space. A thread uses the process's address space and share it with the other threads of that process.

5)

```
Per process items              | Per thread items
-------------------------------|----------------
Address space                  | Program counter
Global variables               | Registers
Open files                     | Stack
Child processes                | State
Pending alarms                 |
Signals and signal handlers    |
Accounting information         |
```

6) A thread can communicate with other thread (of the same process) directly by using methods like wait(), notify(), notifyAll(). A process can communicate with other process by using inter-process communication.

7) New threads are easily created. However the creation of new processes require duplication of the parent process.

8) Threads have control over the other threads of the same process. A process does not have control over the sibling process, it has control over its child processes only.

Here, we will discuss only thread synchronization.

Thread Synchronization

Race Conditions and Critical Sections

A *race condition* is a special condition that may occur inside a critical section. A *critical section* is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition. The term race condition stems from the metaphor that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

This may all sound a bit complicated, so I will elaborate more on race conditions and critical sections in the following sections.

Critical Sections

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.

In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the same resources, as long as the resources do not change.

Here is a critical section Java code example that may fail if executed by multiple threads simultaneously:

```
public class Counter {

   protected long count = 0;

   public void add(long value){
      this.count = this.count + value;
   }
}
```

Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class. There is no way to know when the operating system switches between the two threads. The code in the add() method is not executed as a single atomic instruction by the Java virtual machine. Rather it is executed as a set of smaller instructions, similar to this:

1. Read this.count from memory into register.
2. Add value to register.
3. Write register to memory.

Observe what happens with the following mixed execution of threads A and B:

```
    this.count = 0;

A:  Reads this.count into a register (0)
B:  Reads this.count into a register (0)
B:  Adds value 2 to register
```

```
   B:  Writes register value (2) back to memory.
this.count now equals 2
   A:  Adds value 3 to register
   A:  Writes register value (3) back to memory.
this.count now equals 3
```

The two threads wanted to add the values 2 and 3 to the counter. Thus the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, the result ends up being different.

In the execution sequence example listed above, both threads read the value 0 from memory. Then they add their i ndividual values, 2 and 3, to the value, and write the result back to memory. Instead of 5, the value left in this.count will be the value written by the last thread to write its value. In the above case it is thread A, but it could as well have been thread B.

Race Conditions in Critical Sections

The code in the add() method in the example earlier contains a critical section. When multiple threads execute this critical section, race conditions occur.

More formally, the situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section.

Preventing Race Conditions

To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Race conditions can be avoided by proper thread synchronization in critical sections. Thread synchronization can be achieved using a **synchronized block of Java code**. Thread synchronization can also be achieved using other synchronization constructs like **locks** or atomic variables like **java.util.concurrent.atomic.AtomicInteger**.

Critical Section Throughput

For smaller critical sections making the whole critical section a synchronized block may work. But, for larger critical sections it may be beneficial to break the critical section into smaller critical sections, to allow multiple threads to execute each a smaller critical section. This may decrease contention on the shared resource, and thus increase throughput of the total critical section.

Here is a very simplified Java code example to show what I mean:

```
public class TwoSums {

    private int sum1 = 0;
    private int sum2 = 0;
```

```
    public void add(int val1, int val2){
        synchronized(this){
            this.sum1 += val1;
            this.sum2 += val2;
        }
    }
}
```

Notice how the add() method adds values to two different sum member variables. To prevent race conditions the summing is executed inside a Java synchronized block. With this implementation only a single thread can ever execute the summing at the same time.

However, since the two sum variables are independent of each other, you could split their summing up into two separate synchronized blocks, like this:

```
public class TwoSums {

    private int sum1 = 0;
    private int sum2 = 0;

    private Integer sum1Lock = new Integer(1);
    private Integer sum2Lock = new Integer(2);

    public void add(int val1, int val2){
        synchronized(this.sum1Lock){
            this.sum1 += val1;
        }
        synchronized(this.sum2Lock){
```

```
            this.sum2 += val2;
        }
    }
}
```

Now two threads can execute the add() method at the same time. One thread inside the first synchronized block, and another thread inside the second synchronized block. The two synchronized blocks are synchronized on different objects, so two different threads can execute the two blocks independently. This way threads will have to wait less for each other to execute the add() method.

This example is very simple, of course. In a real life shared resource the breaking down of critical sections may be a whole lot more complicated, and require more analysis of execution order possibilities.

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

Mutual Exclusive

Synchronized method.

Synchronized block.

static synchronization.

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:

by synchronized method

by synchronized block

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 anonymous class

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

```
//Program of synchronized method by using annonymo
us 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

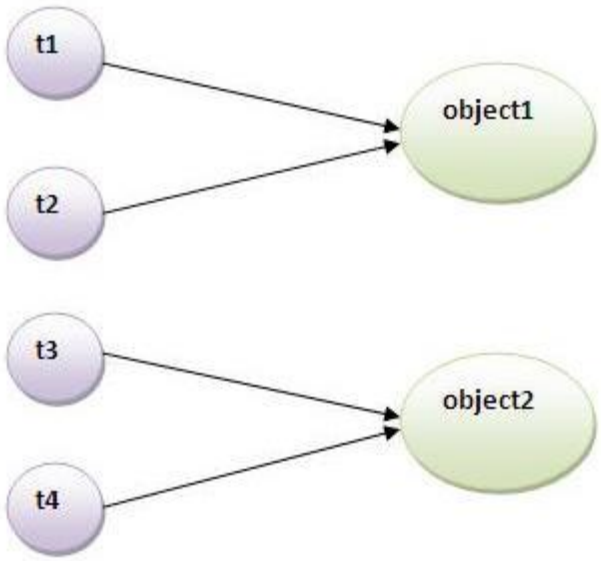
```
class Table{
```

```
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
        // ...
    }
}
```

synchronized in Java is an implementation of the Monitor concept. When you mark a block of code as synchronized you use an object as a parameter. When an executing thread comes to such a block of code, it has to first wait until there is no other executing thread in a synchronized block on that same object.

```
Object a = new Object();
Object b = new Object();
...
synchronized(a){
    doStuff();
}
```

```
...
synchronized(b){
    doSomeStuff();
}
...
synchronized(a){
    doOtherStuff();
}
```

In the above example, a thread running doOtherStuff() would block another thread from entering the block of code protecting doStuff(). However a thread could enter the block around doSomeStuff() without a problem as that is synchronized on Object b, not Object a.

When you use the synchronized modifier on an instance method (a non-static method), it is very similar to having a synchronized block with "this" as the argument. So in the following example, methodA() and methodB() will act the same way:

```
public synchronized void methodA() {
  doStuff();
}
...
public void methodB() {
    synchronized(this) {
        doStuff();
    }
}
```

Note that if you have a methodC() in that class which is not synchronized and does not have a synchronized block, nothing will stop a thread from entering that method and careless programming could let that thread access non-safe code in the object.

If you have a static method with the synchronized modifier, it is practically the same thing as having a synchronized block with `ClassName.class` as the argument (if you have an object of that class, `ClassName cn = new ClassName();`, you can access that object with `Class c = cn.getClass();`)

```
class ClassName {
  public void static synchronized staticMethodA() {
    doStaticStuff();
  }
  public static void staticMethodB() {
    synchronized(ClassName.class) {
      doStaticStuff();
    }
  }
  public void nonStaticMethodC() {
    synchronized(this.getClass()) {
      doStuff();
    }
  }
  public static void unSafeStaticMethodD() {
   doStaticStuff();
  }
}
```

So in the above example, staticMethodA() and staticMethodB() act the same way. An executing thread will also be blocked from accessing the code block in nonStaticMethodC() as it is synchronizing on the same object.

However, it is important to know that nothing will stop an executing thread from accessing unSafeStaticMethodD(). Even if we say that a static method "synchronizes on the

Class object", it does not mean that it synchronizes all accesses to methods in that class. It simply means that it uses the Class object to synchronize on. Non-safe access is still possible.

> In short if you synchronize on a static method you will synchronize on the class (object) and not on an instance (object). That means while execution of a static method the whole class is blocked. So other static synchronized methods are also blocked.
>
> This is not object level synchronization.

There is virtually no difference between synchronizing a block and synchronizing a method. Basically:

```
void synchronized m() {...}
```

is the same as

```
void m() { synchronized(this) {...} }
```

By comparison a static synchronized method is the same as:

```
static void m() { synchronized(MyClass.class) {...} }
```

Java Volatile Keyword

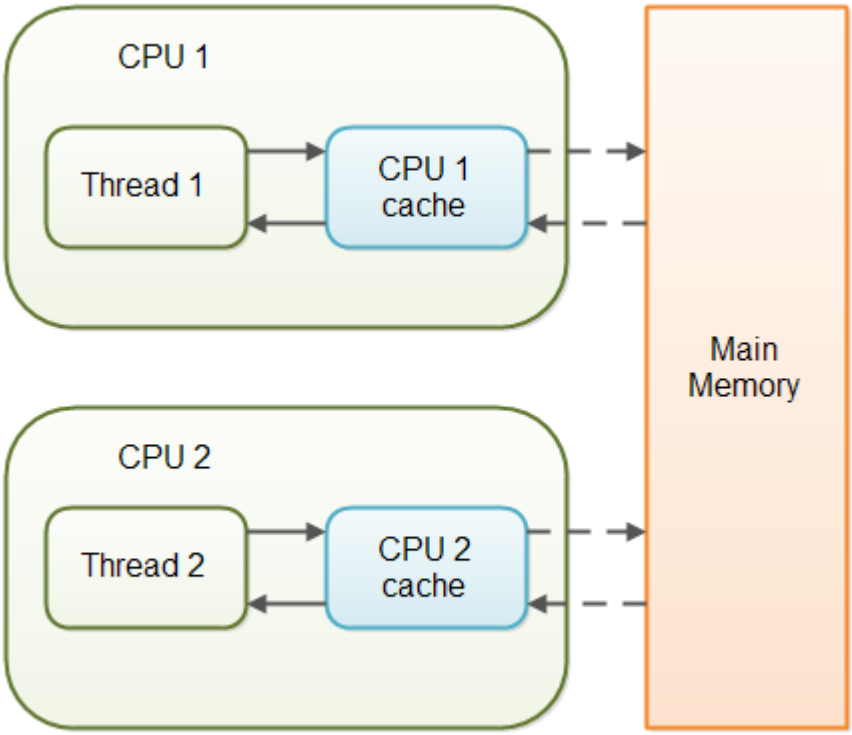
The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Actually, since Java 5 the volatile keyword guarantees more than just that volatile variables are written to and read from main memory. I will explain that in the following sections.

The Java volatile Visibility Guarantee

The Java volatile keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. This is illustrated here:



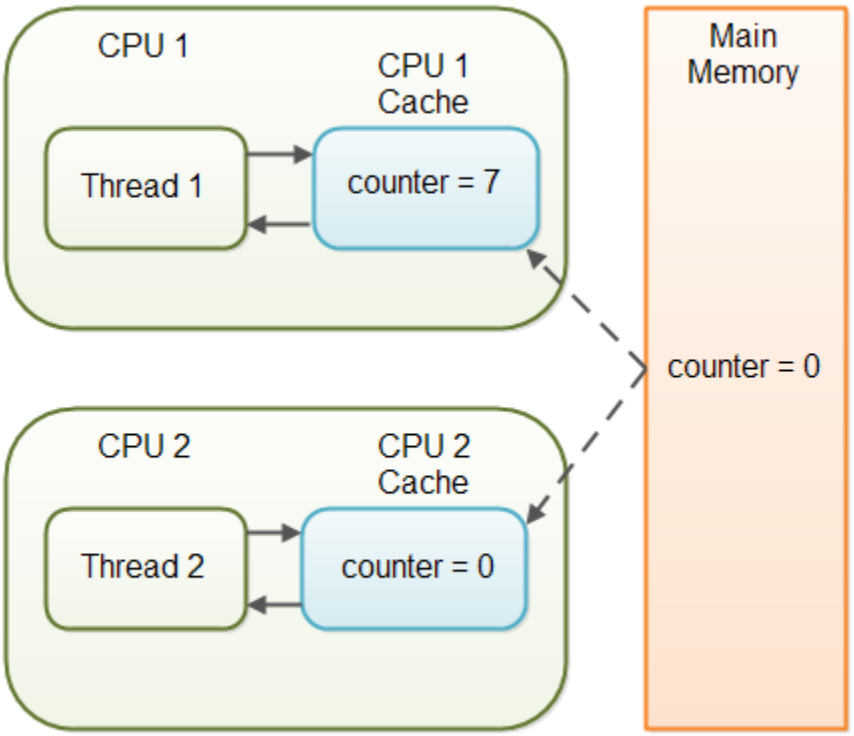
With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems which I will explain in the following sections.

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

```
public class SharedObject {

    public int counter = 0;

}
```

Imagine too, that only Thread 1 increments the counter variable, but both Thread 1 and Thread 2 may read the counter variable from time to time.

If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory. Here is how the volatile declaration of the counter variable looks:

```
public class SharedObject {

    public volatile int counter = 0;

}
```

Declaring a variable volatile thus *guarantees the visibility* for other threads of writes to that variable.

volatile is Not Always Enough

Even if the volatile keyword guarantees that all reads of a volatile variable are read directly from main memory, and all writes to a volatile variable are written directly to main memory, there are still situations where it is not enough to declare a variable volatile.

In the situation explained earlier where only Thread 1 writes to the shared counter variable, declaring the counter variable volatile is enough to make sure that Thread 2 always sees the latest written value.

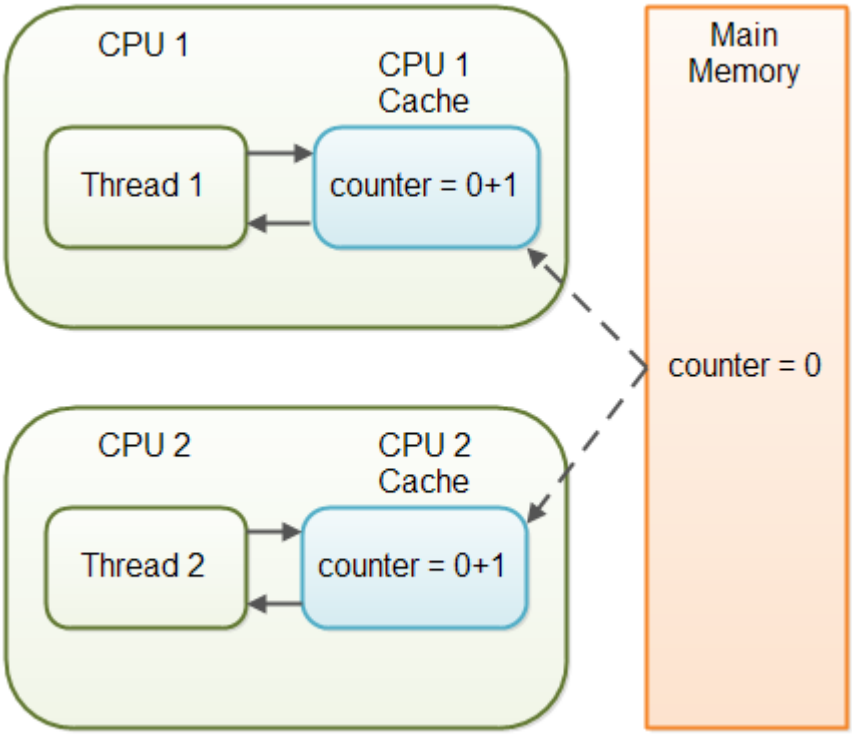
In fact, multiple threads could even be writing to a shared volatile variable, and still have the correct value stored in main memory, if the new value written to the variable does not depend on its previous value. In other words, if a thread writing a value to the

shared volatile variable does not first need to read its value to figure out its next value.

As soon as a thread needs to first read the value of a volatile variable, and based on that value generate a new value for the shared volatile variable, a volatile variable is no longer enough to guarantee correct visibility. The short time gap in between the reading of the volatile variable and the writing of its new value, creates an **race condition** where multiple threads might read the same value of the volatile variable, generate a new value for the variable, and when writing the value back to main memory - overwrite each other's values.

The situation where multiple threads are incrementing the same counter is exactly such a situation where a volatile variable is not enough. The following sections explain this case in more detail.

Imagine if Thread 1 reads a shared counter variable with the value 0 into its CPU cache, increment it to 1 and not write the changed value back into main memory. Thread 2 could then read the same counter variable from main memory where the value of the variable is still 0, into its own CPU cache. Thread 2 could then also increment the counter to 1, and also not write it back to main memory. This situation is illustrated in the diagram below:



Thread 1 and Thread 2 are now practically out of sync. The real value of the shared counter variable should have been 2, but each of the threads has the value 1 for the variable in their CPU caches, and in main memory the value is still 0. It is a mess! Even if the threads eventually write their value for the shared counter variable back to main memory, the value will be wrong.

When is volatile Enough?

As I have mentioned earlier, if two threads are both reading and writing to a shared variable, then using the volatile keyword for that is not enough. You need to use a **synchronized** in that case to guarantee that the reading and writing of the variable is atomic. Reading or

writing a volatile variable does not block threads reading or writing. For this to happen you must use the synchronized keyword around critical sections.

As an alternative to a synchronized block you could also use one of the many atomic data types found in the **java.util.concurrent package**. For instance, the **AtomicLong** or **AtomicReference** or one of the others.

The AtomicReference class provides an object reference variable which can be read and written atomically. By atomic is meant that multiple threads attempting to change the same AtomicReference (e.g. with a compare-and-swap operation) will not make the AtomicReference end up in an inconsistent state.

In case only one thread reads and writes the value of a volatile variable and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable. Without making the variable volatile, this would not be guaranteed.

The volatile keyword is guaranteed to work on 32 bit and 64 variables.

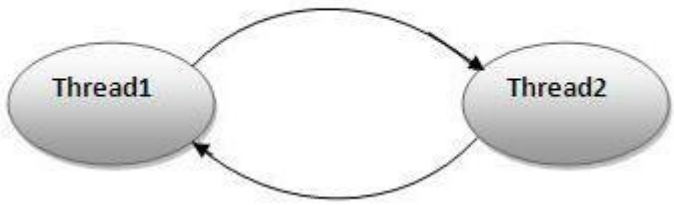
Performance Considerations of volatile

Reading and writing of volatile variables causes the variable to be read or written to main memory. Reading from and writing to main memory is more expensive than accessing the CPU cache. Accessing volatile variables also prevent instruction reordering which is a normal performance enhancement technique. Thus, you should

only use volatile variables when you really need to enforce visibility of variables.

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 = "abhishek";
    final String resource2 = "ketaki";
    // 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()

---

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()

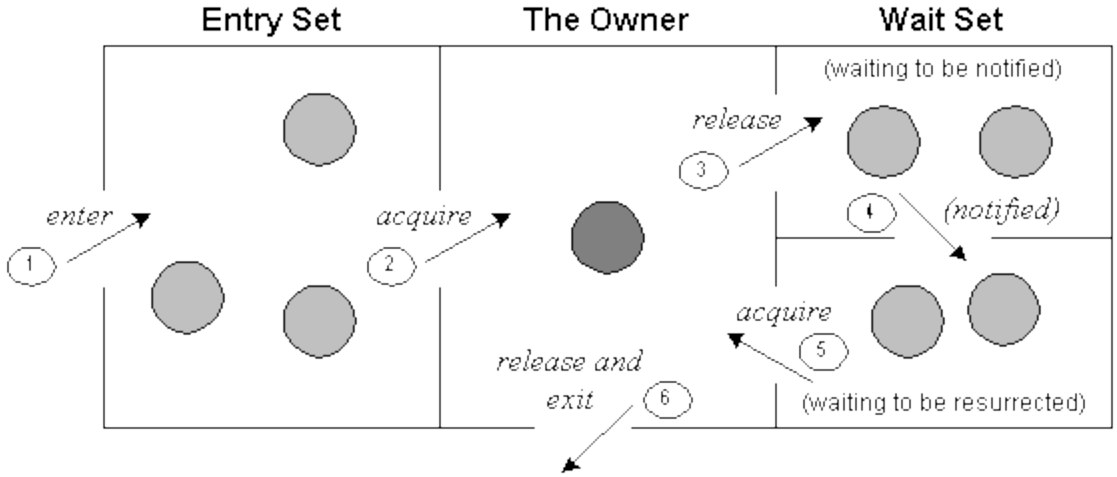
---

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:

Threads enter to acquire lock.

Lock is acquired by on thread.

Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.

If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).

Now thread is available to acquire lock.

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
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 TestInterruptingThre
ad1();
t1.start();
try{
Thread.sleep(2000);
t1.interrupt();
}catch(Exception e){System.out.println("Exception ha
ndled "+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);
// NO THROWS CLAUSE
}
System.out.println("thread is running...");
}

public static void main(String args[]){
TestInterruptingThread2 t1=new TestInterruptingThre
ad2();
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 TestInterruptingThre
ad3();
t1.start();

t1.interrupt();

}
}
```

Output:1
2
3
4

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 after that it sets the flag to false if it is true.

```
package com.t;

public class TestInterruptingThread4 extends Thread {

    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("Thread started:
"+t.getName());
        for (int i = 1; i <= 2; i++) {
            if (Thread.interrupted()) {
                System.out.println("code for
interrupted thread i: " + i + " " + t.getName());
            } else {
                System.out.println("code for normal
thread i: " + i+ " " + t.getName());
            }

        } // 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:

```
Thread started: Thread-0
Thread started: Thread-1
code for normal thread i: 1 Thread-1
code for normal thread i: 2 Thread-1
code for interrupted thread i: 1 Thread-0
code for normal thread i: 2 Thread-0
```

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 na

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	destroys this thread group and all its sub groups.

	destroy()	
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 conso

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(),"thre
e");
```

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]

Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads. For example, imagine a program in which one set of threads **is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file.** If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience. The following program, which creates two thread groups of two threads each, illustrates this usage:

```
// Demonstrate thread groups.
class NewThread extends Thread {
boolean suspendFlag;
NewThread(String threadname, ThreadGroup tgOb) {
super(tgOb, threadname);
```

```
System.out.println("New thread: " + this);
suspendFlag = false;
start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i—) {
System.out.println(getName() + ": " + i);
Thread.sleep(1000);
synchronized(this) {
while(suspendFlag) {
wait();
}
}
}
} catch (Exception e) {
System.out.println("Exception in " + getName());
}
System.out.println(getName() + " exiting.");
}
void mysuspend() {
suspendFlag = true;
}
synchronized void myresume() {
suspendFlag = false;
notify();
}
}
```

```
class ThreadGroupDemo {
public static void main(String args[]) {
ThreadGroup groupA = new ThreadGroup("Group A");
ThreadGroup groupB = new ThreadGroup("Group B");
NewThread ob1 = new NewThread("One", groupA);
NewThread ob2 = new NewThread("Two", groupA);
NewThread ob3 = new NewThread("Three", groupB);
NewThread ob4 = new NewThread("Four", groupB);
System.out.println("\\nHere is output from list():");
groupA.list();
groupB.list();
System.out.println();
System.out.println("Suspending Group A");
Thread tga[] = new Thread[groupA.activeCount()];
groupA.enumerate(tga); // get threads in group
for(int i = 0; i < tga.length; i++) {
((NewThread)tga[i]).mysuspend(); // suspend each
thread
}
try {
Thread.sleep(4000);
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Resuming Group A");
for(int i = 0; i < tga.length; i++) {
((NewThread)tga[i]).myresume(); // resume threads in
group}
// wait for threads to finish
```

```
try {
System.out.println("Waiting for threads to finish.");
ob1.join();
ob2.join();
ob3.join();
ob4.join();
} catch (Exception e) {
System.out.println("Exception in Main thread");
}
System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,Group A]
New thread: Thread[Two,5,Group A]
New thread: Thread[Three,5,Group B]
New thread: Thread[Four,5,Group B]
Here is output from list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
Thread[One,5,Group A]
Thread[Two,5,Group A]
java.lang.ThreadGroup[name=Group B,maxpri=10]
Thread[Three,5,Group B]
Thread[Four,5,Group B]
Suspending Group A
Three: 5
Four: 5
Three: 4
Four: 4
```

```
Three: 3
Four: 3
Three: 2
Four: 2
Resuming Group A
Waiting for threads to finish.
One: 5
Two: 5
Three: 1
Four: 1
One: 4
Two: 4
Three exiting.
Four exiting.
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
One exiting.
Two exiting.
Main thread exiting.
```

Inside the program, notice that thread group A is suspended for four seconds. As the output confirms, this causes threads One and Two to pause, but threads Three and Four continue running. After the four seconds, threads One and Two are resumed. Notice how thread group A is suspended and resumed. First, the threads in

group A are obtained by calling **enumerate( )** on group A. Then, each thread is suspended by iterating through the resulting array. To resume the threads in A, the list is again traversed and each thread is resumed. One last point: this example uses the recommended Java 2 approach to suspending and resuming threads. It does not rely upon the deprecated methods **suspend( )** and **resume( )**.

Java ThreadGroup.enumerate(Thread [] list)

Syntax

ThreadGroup.enumerate(Thread [] list) has the following syntax.

```
public int enumerate(Thread [] list)
```

Example

In the following code shows how to use ThreadGroup.enumerate(Thread [] list) method.

```
public class Main {
  public static void main(String[] args) {
    ThreadGroupDemo tg = new ThreadGroupDemo();

  }
}

class ThreadGroupDemo implements Runnable {
  public ThreadGroupDemo() {
```

```
        ThreadGroup pGroup = new
ThreadGroup("Parent ThreadGroup");

        ThreadGroup cGroup = new ThreadGroup(pGroup,
"Child ThreadGroup");

        Thread t1 = new Thread(pGroup, this);
        System.out.println("Starting " + t1.getName());
        t1.start();

        Thread t2 = new Thread(cGroup, this);
        System.out.println("Starting " + t2.getName());
        t2.start();

        Thread[] list = new
Thread[pGroup.activeCount()];
        int count = pGroup.enumerate(list);
        for (int i = 0; i < count; i++) {
          System.out.println("Thread " +
list[i].getName() + " found");
        }

   }

   // implements run()
   public void run() {


System.out.println(Thread.currentThread().getName() +
      " finished executing.");
   }
}
```

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
```

Java Monitors Are Reentrant

The Java runtime system allows a thread to re-acquire a monitor that it already holds because Java monitors are reentrant. Reentrant monitors are important because they eliminate the possibility of a single thread deadlocking itself on a monitor that it already holds.

Consider this class:

```
class Reentrant {
    public synchronized void a() {
```

```
        b();
        System.out.println("here I am, in a()");
    }
    public synchronized void b() {
        System.out.println("here I am, in b()");
    }
}
```

Reentrant contains two synchronized methods: a and b. The first synchronized method, a, calls the other synchronized method, b.

When control enters method a, the current thread acquires the monitor for the Reentrant object.
Now, a calls b and because b is also synchronized the thread attempts to acquire the same monitor again.
Because Java supports reentrant monitors, this works. The current thread can acquire the Reentrant object's monitor again and both a and b execute to conclusion as is evidenced by the output:

```
here I am, in b()
here I am, in a()
```

In systems that don't support reentrant monitors, this sequence of method calls would cause deadlock.

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 mach
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
7)	public long totalMemory()	returns amount of total memory in

Java Runtime exec() method

```
public class Runtime1{
public static void main(String args[])throws Excepti
on{
```

```
Runtime.getRuntime().exec("notepad");//will open a n
ew notepad
}
}
```

// SEE ECLIPSE PROGRAM

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 e.g. 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 Excepti
on{
Runtime.getRuntime().exec("shutdown -s -t 0");
}
}
```

How to shutdown windows system in Java

```
public class Runtime2{
public static void main(String args[])throws Excepti
on{
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 Excepti
on{
Runtime.getRuntime().exec("shutdown -r -t 0");
}
}
```

Java Runtime availableProcessors()

```
public class Runtime4{
public static void main(String args[])throws Excepti
on{
System.out.println(Runtime.getRuntime().availableProc
essors());
}
}
```

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 Excepti
on{
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, Fre
e Memory: "+r.freeMemory());
System.gc();
System.out.println("After gc(), Free Memory: "+r.free
Memory());
}
}
```

```
Total Memory: 100139008
Free Memory: 99474824
After creating 10000 instance, Free Memory: 99310552
After gc(), Free Memory: 100182832
```

// DISCUSS LAN WHITE PAPER

Java Timer TimerTask Example

Java **java.util.Timer** is a utility class that can be used to schedule a thread to be executed at certain time in future. **Java Timer** class can be used to schedule a task to be run one-time or to be run at regular intervals.

Java TimerTask

java.util.TimerTask is an **abstract class** that implements Runnable interface and we need to extend this class to create our own **TimerTask** that can be scheduled using *java Timer* class.

Java Timer class is thread safe and multiple threads can share a single **Timer** object without need for external synchronization. Timer class uses **java.util.TaskQueue** to add tasks at given regular interval and at any time there can be only one thread running the TimerTask, for example if you are creating a Timer to run every 10 seconds but single thread execution takes 20 seconds, then Timer object will keep adding tasks to the queue and as soon as one thread is finished, it will notify the queue and another thread will start executing.

Java Timer class uses Object wait and notify methods to schedule the tasks.

Here is a simple program for Java Timer and TimerTask example.

```
package com.journaldev.threads;


import java.util.Date;

import java.util.Timer;

import java.util.TimerTask;


public class MyTimerTask extends TimerTask {


    @Override

    public void run() {

        System.out.println("Timer task started at:"+new
Date());

        completeTask();

        System.out.println("Timer task finished at:"+new
Date());

    }


    private void completeTask() {

        try {

            //assuming it takes 20 secs to complete the
task
```

```
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]){
        TimerTask timerTask = new MyTimerTask();
        //running timer task as daemon thread
        Timer timer = new Timer(true);
        timer.scheduleAtFixedRate(timerTask, 0,
10*1000);
        System.out.println("TimerTask started");
        //cancel after sometime
        try {
            Thread.sleep(120000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        timer.cancel();
```

```
        System.out.println("TimerTask cancelled");
        try {
            Thread.sleep(30000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

Notice that one thread execution will take 20 seconds but Java Timer object is scheduled to run the task every 10 seconds. Here is the output of the program:

```
TimerTask started
Timer task started at:Wed Dec 26 19:16:39 PST 2012
Timer task finished at:Wed Dec 26 19:16:59 PST 2012
Timer task started at:Wed Dec 26 19:16:59 PST 2012
Timer task finished at:Wed Dec 26 19:17:19 PST 2012
Timer task started at:Wed Dec 26 19:17:19 PST 2012
Timer task finished at:Wed Dec 26 19:17:39 PST 2012
Timer task started at:Wed Dec 26 19:17:39 PST 2012
```

```
Timer task finished at:Wed Dec 26 19:17:59 PST 2012
Timer task started at:Wed Dec 26 19:17:59 PST 2012
Timer task finished at:Wed Dec 26 19:18:19 PST 2012
Timer task started at:Wed Dec 26 19:18:19 PST 2012
TimerTask cancelled
Timer task finished at:Wed Dec 26 19:18:39 PST 2012
```

The output confirms that if a task is already executing, Timer will wait for it to finish and once finished, it will start again the next task from the queue.

Java Timer object can be created to run the associated tasks as a daemon thread. Timer *cancel()* method is used to terminate the timer and discard any scheduled tasks, however it doesn't interfere with the currently executing task and let it finish. If the timer is run as daemon thread, whether we cancel it or not, it will terminate as soon as all the user threads are finished executing.

Timer class contains several **schedule**() methods to schedule a task to run once at given date or after some delay. There are several **scheduleAtFixedRate**() methods to run a task periodically with certain interval.

While scheduling tasks using Timer, you should make sure that time interval is more than normal thread execution, otherwise tasks queue size will keep growing and eventually task will be executing always. That's all for a quick roundup on Java Timer and Java TimerTask.

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.

Semaphore in Java

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then

access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Refer link

http://www.geeksforgeeks.org/semaphore-in-java/

See also ProcessBuilder later.

For Advanced Topics see other document " Advance Threads". Eg ThreadLocal or queues

Java ThreadLocal

The ThreadLocal class in Java enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to aThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables.

Creating a ThreadLocal

Here is a code example that shows how to create a ThreadLocal variable:

```
private ThreadLocal myThreadLocal = new ThreadLocal();
```

As you can see, you instantiate a new ThreadLocal object. This only needs to be done once per thread. Even if different threads execute the same code which accesses a ThreadLococal, each thread will see only its own ThreadLocal instance. Even if two

different threads set different values on the same ThreadLocal object, they cannot see each other's values.

Accessing a ThreadLocal

Once a ThreadLocal has been created you can set the value to be stored in it like this:

```
myThreadLocal.set("A thread local value");
```

You read the value stored in a ThreadLocal like this:

```
String threadLocalValue = (String) myThreadLocal.get();
```

The get() method returns an Object and the set() method takes an Object as parameter.

Generic ThreadLocal

You can create a generic ThreadLocal so that you do not have to typecast the value returned by get(). Here is a generic ThreadLocal example:

```
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();
```

Now you can only store strings in the ThreadLocal instance. Additionally, you do not need to typecast the value obtained from the ThreadLocal:

```
myThreadLocal.set("Hello ThreadLocal");
```

```
String threadLocalValue = myThreadLocal.get();
```

Initial ThreadLocal Value

Since values set on a ThreadLocal object only are visible to the thread who set the value, no thread can set an initial value on a ThreadLocal using set() which is visible to all threads.

Instead you can specify an initial value for a ThreadLocal object by subclassing ThreadLocal and overriding the initialValue() method. Here is how that looks:

```
private ThreadLocal myThreadLocal = new
ThreadLocal<String>() {
   @Override protected String initialValue() {
      return "This is the initial value";
   }
};
```

Now all threads will see the same initial value when calling get() before having called set() .

Full ThreadLocal Example

Here is a fully runnable Java ThreadLocal example:

```
public class ThreadLocalExample {
```

```
   public static class MyRunnable implements Runnable {

      private ThreadLocal<Integer> threadLocal =
            new ThreadLocal<Integer>();

      @Override
      public void run() {
         threadLocal.set( (int) (Math.random() * 100D) );

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

         System.out.println(threadLocal.get());
      }
   }


   public static void main(String[] args) {
      MyRunnable sharedRunnableInstance = new
MyRunnable();

      Thread thread1 = new
Thread(sharedRunnableInstance);
      Thread thread2 = new
Thread(sharedRunnableInstance);

      thread1.start();
      thread2.start();

      thread1.join(); //wait for thread 1 to terminate
```

```
        thread2.join(); //wait for thread 2 to terminate
    }

}
```