**Multitaksing🡪 processing multiple task at a time.Two type of multitasking. 1)Process based, 2)Thread based.**

\*\*Process based multitasking is OS level concept. It cannot be done in programming level.

\*\*Thread is flow of execution. Lightweight thread consumes less memory and heavyweight thread consumes more memory.

Main aim of multithreading is to execute the program in less time.

\*\*If you don’t do any threading JVM calls **MainThread** and creates runtime stack for **MainThread**. Then **MainThread** will find the .class file and load the class into memory. Then main method execution starts and one frame is allocated in stack for main method.

**MainThread** name customized to **main**. The parent of this **main is main (this is a threadgroup).**

**\*\*Whenever a new thread is created JVM will create one new runtime stack for that thread.only run method will be placed in that runtime stack as child threads have only access to run method.**

**\*\*If any exception occurs in main thread and no catch block is there, then exception will be printed in the console for main thread, but still the child thread will be executed normally but the child threads will go in dead state.**

\*\*Main thread will get highest priority in multithreading.

**\*\*Best way to create thread ::🡪**

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**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | class ImplementsRunnable implements Runnable {     private int counter = 0;     public void run() {   counter++;   System.out.println("ImplementsRunnable : Counter : " + counter);   }   }     class ExtendsThread extends Thread {     private int counter = 0;     public void run() {   counter++;   System.out.println("ExtendsThread : Counter : " + counter);   }   }     public class ThreadVsRunnable {     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();   }   } |

**Output of the above program.**

ImplementsRunnable : Counter : 1  
ImplementsRunnable : Counter : 2  
ImplementsRunnable : Counter : 3  
ExtendsThread : Counter : 1  
ExtendsThread : Counter : 1  
ExtendsThread : Counter : 1

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](https://manikandanmv.wordpress.com/2010/12/18/coding-to-interfaces/).

**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.

Example :

**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 area is enough rather than to all the usage places. I will explain this elaborately in the upcoming post.

\*\*class ThreadedLocalMethods{  
  
void process(){  
int i=0;  
Object obj = new Object();  
}  
  
}  
In the above code snippet, my objects reference and object are defined at method level,   
a. will each thread create a new object on the heap for this method local object or  
b. it will only create new reference (Object obj) and share the same object (new Object()) on the heap amongst the threads?

Ans:🡪 t's A. i will always be created on the thread local stack.

Actually, since JDK 6 the JVM sometimes creates objects on the stack rather than the heap. This only happens in cases where the JVM can determine that an object is only ever accessible from one thread - in this case, allocating it on the stack is very quick, and when the method completes, the object is promptly deallocated. This can give very fast GC.

Ill take it a step further. Even if you call that method recursively, each call of the method will create a new reference to a new object. So its not that each thread will create new objects per method, but its each invocation of that method (even if by the same thread) will create new objects.

**Creating in Thread**

1. **Extending Thread class🡪**

package demo.ThreadLearning;

class Runner extends Thread{

public void run() {

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

System.out.println("Hello "+i+ "in "+Thread.currentThread());

try {

Thread.sleep(500);

} catch (InterruptedException e) {

}

}

}

}

public class AppExtendingThread {

public static void main(String[] args) {

Runner runner1=new Runner();

runner1.start();

Runner runner2 = new Runner();

runner2.start();

}

}

1. **Implementing Runnable Interface🡪**

package demo.ThreadLearning;

class Runner1 implements Runnable {

public void run() {

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

System.out.println("Hello " + i + "in " + Thread.currentThread());

try {

Thread.sleep(500);

} catch (InterruptedException e) {

}

}

}

}

public class AppImplementingRunnable {

public static void main(String[] args) {

Runner1 appImplementingRunnable = new Runner1();

Thread t1=new Thread(appImplementingRunnable);

Thread t2 = new Thread(appImplementingRunnable);

t1.start();

t2.start();

}

}

**Instead of declaring a class for your thread implementation you can also create thread inline like this 🡪**

package demo.ThreadLearning;

public class AppThreadInOneClass {

public static void main(String[] args) {

Thread t1=new Thread(new Runnable(){

public void run() {

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

System.out.println("Hello " + i + "in " + Thread.currentThread());

try {

Thread.sleep(500);

} catch (InterruptedException e) {

}

}

}

});

Thread t2 = new Thread(new Runnable() {

public void run() {

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

System.out.println("Hello " + i + "in " + Thread.currentThread());

try {

Thread.sleep(500);

} catch (InterruptedException e) {

}

}

}

});

t1.start();

t2.start();

}

}

**You can also create only one class which implements runnable and have your multithreading code inside run() method and in that class you can have main method which will create separate thread** and **call start on that objects🡪**

package demo.ThreadLearning;

public class AppThreadOnlyOneClass implements Runnable {

public void run() {

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

System.out.println("Hello " + i + "in " + Thread.currentThread());

try {

Thread.sleep(500);

} catch (InterruptedException e) {

}

}

}

public static void main(String[] args) {

AppThreadOnlyOneClass class1=new AppThreadOnlyOneClass();

Thread t1=new Thread(class1);

Thread t2 = new Thread(class1);

t1.start();

t2.start();

}

}

**It is important to remember when start() method called on thread object only run() method of the thread is called in different call stack. If you directly call the run() method of the thread object then no different thread will be created everything will be executed by main thread only🡪**

package demo.ThreadLearning;

public class AppThreadOnlyOneClass implements Runnable {

public void run() {

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

System.out.println("Hello " + i + "in " + Thread.currentThread());

try {

Thread.sleep(500);

} catch (InterruptedException e) {

}

}

}

public static void main(String[] args) {

AppThreadOnlyOneClass class1=new AppThreadOnlyOneClass();

Thread t1=new Thread(class1);

Thread t2 = new Thread(class1);

t1.run();

t2.run();

}

}

**Thread Synchronization**

1. **Volatile Keyword:🡪**

package demo.ThreadLearning;

import java.util.Scanner;

class Processor implements Runnable {

private boolean running=true;

public void run() {

while(running) {

System.out.println("Hello");

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public void shutDown() {

running=false;

}

}

public class ThreadSynchronization1 {

public static void main(String[] args) {

Processor processor1 = new Processor();

Thread t1=new Thread(processor1);

t1.start();

System.out.println("Press return key to stop");

Scanner scanner=new Scanner(System.in);

scanner.nextLine();

processor1.shutDown();

}

}

\*\*in this code the problem is that some system thread decided to keep catched version of **running** variable in his stack so the loop goes indefinitely.

We can solve this problem by declaring the running variable as volatile using **volatile** keyword.

This will ensure that the variable is not catched in the thread stack and at a time only one thread can change the value of that variable .

1. **Synchronized method :🡪**

If we call t1.join() from main thread then main thread will wait until t1 completes its execution.

package demo.ThreadLearning;

public class ThreadSynchronizedKeyword {

private int count;

public static void main(String[] args) {

ThreadSynchronizedKeyword threadSynchronizedKeyword = new ThreadSynchronizedKeyword();

threadSynchronizedKeyword.doWork();

}

public void doWork() {

Thread t1=new Thread(new Runnable(){

public void run() {

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

count++;

}

}

});

Thread t2 = new Thread(new Runnable() {

public void run() {

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

count++;

}

}

});

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

}

System.out.println("Value of count is : "+count);

}

}

Here count++🡺 count=count+1.

So in this code two thread operates on count at a time then it will create value mismatch.

So the issue is with count value increments. Writing count as volatile will not serve the purpose as it is not a problem with variable catching.

But the following code will work :🡪

package demo.ThreadLearning;

public class ThreadSynchronizedKeyword {

private int count;

public static void main(String[] args) {

ThreadSynchronizedKeyword threadSynchronizedKeyword = new ThreadSynchronizedKeyword();

threadSynchronizedKeyword.doWork();

}

public synchronized void increment() {

count++;

}

public void doWork() {

Thread t1=new Thread(new Runnable(){

public void run() {

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

increment();

}

}

});

Thread t2 = new Thread(new Runnable() {

public void run() {

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

increment();

}

}

});

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

}

System.out.println("Value of count is : "+count);

}

}

Because of the synchronized method increment() as at a time only one method can acquire a lock on this method.

But if we want to acquire lock on multiple things then it is better to use synchronized block.

1. **Synchronized keyword:🡪**

class A {

public synchronized void methodA() {

//method A

}

public synchronized void methodB() {

// method B

}

}

in this case two thread cannot simultaneously call two methods on same class A instance as when you make one method synchronized it creates lock on the whole object so other thread cannot do anything on the object until first thread finishes.

To overcome this we can write the code as🡪

class A {

private final Object lockA = new Object();

private final Object lockB = new Object();

public void methodA() {

synchronized(lockA) {

//method A

}

}

public void methodB() {

synchronized(lockB) {

//method B

}

}

}

In this code no two thread can access synchronized(lockA)/ synchronized(lockB) at the same time but one thread can access inside code of synchronized(lockA) while other is accessing synchronized(lockB) synchronized(lockB).

\*\*\*\* Java Thread acquires an **object level lock** when it enters into an **instance** synchronized java method and acquires a **class level lock** when it enters into **static** synchronized java method.

**THREAD POOL**

Instead of manually creating threads and assigning tasks to multiple thread we can create a thread pool of some threads and submit the tasks to them. They will complete the tasks one by one we don’t have to think about thread starting and assigning tasks. It comes under java.util.concurrent package.

package demo.ThreadLearning;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

class Processor1 implements Runnable {

private int id;

public Processor1(int id) {

this.id=id;

}

public void run() {

System.out.println("Starting " + this.id);

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

}

System.out.println("Completed " + this.id);

}

}

public class ThreadPoolDemo {

public ThreadPoolDemo() {

super();

}

public static void main(String[] args) {

ExecutorService executor=Executors.newFixedThreadPool(3);

for (int i=0 ;i<5 ;i++ ) {

executor.submit(new Processor1(i));

}

executor.shutdown();

System.out.println("All Task Submitted");

try {

executor.awaitTermination(1,TimeUnit.DAYS);

} catch (InterruptedException e) {

}

System.out.println("All Task Completed");

}

}

\*\* This code shows use of ThreadPool class.

* shutdown() will just tell the executor service that it can't accept new tasks, but the already submitted tasks continue to run
* shutdownNow() will do the same AND will **try to cancel** the already submitted tasks by interrupting the relevant threads. Note that if your tasks ignore the interruption, shutdownNowwill behave exactly the same way as shutdown.

\*\* According to the description of awaitTermination(): "Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first."  Await Termination will wait for the already submitted task to complete before shutdown/time interval of its parameter which comes first.

**How to create thread safe Singleton in Java - Java Singleton Example**

1)public enum Singleton {

INSTANCE**;**

Public void show()

{

System.out.println(“Singleton using Enum in java”);

}

}

Singleton.INSTANCE.show();

This,

public enum MySingleton {

INSTANCE;

}

has an implicit empty constructor. Let's be explicit instead,

public enum MySingleton {

INSTANCE;

private MySingleton() {

System.out.println("Here");

}

}

If you then added another class with a main() method like

public static void main(String[] args) {

System.out.println(MySingleton.INSTANCE);

}

You would see

Here

INSTANCE

enum fields are compile time constants, but they are instances of their enum type. And, they're constructed when the enum type is referenced for the first time.

2) You can also create thread safe Singleton in Java by creating Singleton instance during [class loading](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html). static fields are initialized during class loading and [Classloader](http://javarevisited.blogspot.com.au/2012/12/how-classloader-works-in-java.html) will guarantee that instance will not be visible until its fully created. Here is example of creating thread safe singleton in Java using static factory method. Only disadvantage of this implementing Singleton patter using static field is that this is *not a lazy initialization* and Singleton is initialized even before any clients call there getInstance() method.

Public class Singleton {

Private static final Singleton INSTANCE=new Singleton();

***/\*\*  
\* Singleton pattern example with static factory method  
\*/***  
  
**public** **class** Singleton{  
    *//initailzed during class loading*  
    **private** **static** **final** Singleton INSTANCE = **new** Singleton();  
    
    *//to prevent creating another instance of Singleton*  
    **private** Singleton(){}  
  
    **public** **static** Singleton getSingleton(){  
        **return** INSTANCE;  
    }  
}  
here we are not creating Singleton instance inside getInstance() method instead it will be created by ClassLoader. Also [private constructor](http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) makes impossible to create another instance , except one case. You can still access private constructor by reflection and calling setAccessible(true). By the way You can still prevent creating another instance of Singleton by this way by [throwing Exception](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) from constructor.

**1) Enum Singletons are easy to write**

This is by far biggest advantage, if you have been writing Singletons prior ot Java 5 than you know that even with double checked locking you can have more than one instances. though that issue is fixed with Java memory model improvement and gurantee provided by volatile variables from Java 5 onwards but it still tricky to write for many beginners. compared to double checked locking with synchronization Enum singletons are cake walk.

**2) Enum Singletons handled Serialization by themselves**

Another problem with conventional Singletons are that once you implement [serializable interface](http://javarevisited.blogspot.sg/2011/04/top-10-java-serialization-interview.html) they are no longer remain Singleton because readObject() method always return a new instance just like constructor in Java. you can avoid that by using readResolve() method and discarding newly created instance by replacing with Singeton as shwon in below example :

*//readResolve to prevent another instance of Singleton*  
    **private** Object readResolve(){  
        **return** INSTANCE;  
    }

This can become even more complex if your Singleton Class maintain state, as you need to make them [transient](http://javarevisited.blogspot.sg/2012/03/difference-between-transient-and.html), but witn **Enum Singleton**, Serialization is guarnateed by JVM.

**3) Creation of Enum instance is thread-safe**

As stated in point 1 since creatino of Enum instance is thread-safe by default you don't need to worry about double checked locking.

# \*\*)[shutdown and awaitTermination which first call have any difference?](http://stackoverflow.com/questions/18425026/shutdown-and-awaittermination-which-first-call-have-any-difference)

Ans:🡪

|  |  |
| --- | --- |
| down vote | Reading the documentation always helps:  **shutdownNow** :  Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. These tasks are drained (removed) from the task queue upon return from this method.  This method **does not wait for actively executing tasks to terminate.** Use awaitTermination to do that.  There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. This implementation cancels tasks via Thread.interrupt(), so any task that fails to respond to interrupts may never terminate  **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.  This method does not wait for previously submitted tasks to complete execution. Use awaitTermination to do that.  **awaitTermination:**  **Blocks until all tasks have completed execution after a shutdown request**, or the timeout occurs, or the current thread is interrupted, whichever happens first. |

You should call shutdown first. Otherwise, you might be waiting for a very long time, since awaitTermination doesn't actually shut down your executor.