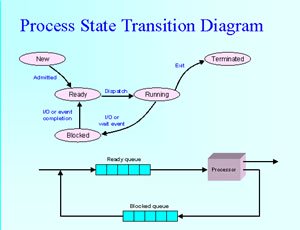
# Java General Questions

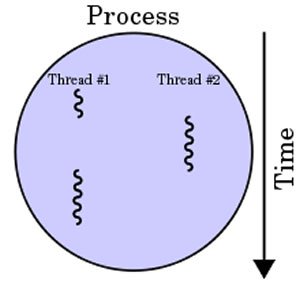
* What is difference between thread and process?

##### **Key difference: Thread and Process are two closely related terms in multi-threading. The main difference between the two terms is that the threads are a part of a process, i.e. a process may contain one or more threads, but a thread cannot contain a process.**

In programming, there are two basic units of execution: processes and threads. They both execute a series of instructions. Both are initiated by a program or the operating system. This article helps to differentiate between the two units.

A process is an instance of a program that is being executed. It contains the program code and its current activity. Depending on the operating system, a process may be made up of multiple threads of execution that execute instructions concurrently. A program is a collection of instructions; a process is the actual execution of those instructions.

A process has a self-contained execution environment. It has a complete set of private basic run-time resources; in particular, each process has its own memory space. Processes are often considered similar to other programs or applications. However, the running of a single application may in fact be a set of cooperating processes. To facilitate communication between the processes, most operating systems use Inter Process Communication (IPC) resources, such as pipes and sockets. The IPC resources can also be used for communication between processes on different systems. Most applications in a virtual machine run as a single process. However, it can create additional processes using a process builder object.



In computers, a thread can execute even the smallest sequence of programmed instructions that can be managed independently by an operating system. The applications of threads and processes differ from one operating system to another. However, the threads are made of and exist within a process; every process has at least one. Multiple threads can also exist in a process and share resources, which helps in efficient communication between threads.

On a single processor, multitasking takes place as the processor switches between different threads; it is known as multithreading. The switching happens so frequently that the threads or tasks are perceived to be running at the same time. Threads can truly be concurrent on a multiprocessor or multi-core system, with every processor or core executing the separate threads simultaneously.

In summary, threads may be considered lightweight processes, as they contain simple sets of instructions and can run within a larger process. Computers can run multiple threads and processes at the same time.

Comparison between Process and Thread:

|  |  |  |
| --- | --- | --- |
|  | **Process** | **Thread** |
| Definition | An executing instance of a program is called a process. | A thread is a subset of the process. |
| Process | It has its own copy of the data segment of the parent process. | It has direct access to the data segment of its process. |
| Communication | Processes must use inter-process communication to communicate with sibling processes. | Threads can directly communicate with other threads of its process. |
| Overheads | Processes have considerable overhead. | Threads have almost no overhead. |
| Creation | New processes require duplication of the parent process. | New threads are easily created. |
| Control | Processes can only exercise control over child processes. | Threads can exercise considerable control over threads of the same process. |
| Changes | Any change in the parent process does not affect child processes. | Any change in the main thread may affect the behavior of the other threads of the process. |
| Memory | Run in separate memory spaces. | Run in shared memory spaces. |
| File descriptors | Most file descriptors are not shared. | It shares file descriptors. |
| File system | There is no sharing of file system context. | It shares file system context. |
| Signal | It does not share signal handling. | It shares signal handling. |
| Controlled by | Process is controlled by the operating system. | Threads are controlled by programmer in a program. |
| Dependence | Processes are independent. | Threads are dependent |

# **Java String Interview Questions**

* What is String in Java? String is a data type?

String is a Class in java and defined in java.lang package. It’s not a primitive data type like int and long. String class represents character Strings. String is used in almost all the Java applications and there are some interesting facts we should know about String. String in immutable and final in Java and JVM uses String Pool to store all the String objects.  
Some other interesting things about String is the way we can instantiate a String object using double quotes and overloading of “+” operator for concatenation.

* What are different ways to create String Object?

We can create String object using new operator like any normal java class or we can use double quotes to create a String object. There are several constructors available in String class to get String from char array, byte array, StringBuffer and StringBuilder.

String str = new String("abc");

String str1 = "abc";

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with same value. If found, it just returns the reference to that String object else it creates a new String object with given value and stores it in the String pool.  
When we use new operator, JVM creates the String object but don’t store it into the String Pool. We can use intern() method to store the String object into String pool or return the reference if there is already a String with equal value present in the pool.

* Write a method to check if input String is Palindrome?

A String is said to be Palindrome if it’s value is same when reversed. For example “aba” is a Palindrome String.  
String class doesn’t provide any method to reverse the String but StringBuffer and StringBuilder class has reverse method that we can use to check if String is palindrome or not.

private static boolean isPalindrome(String str) {

if (str == null)

return false;

StringBuilder strBuilder = new StringBuilder(str);

strBuilder.reverse();

return strBuilder.toString().equals(str);

}

Sometimes interviewer asks not to use any other class to check this, in that case we can compare characters in the String from both ends to find out if it’s palindrome or not.

private static boolean isPalindromeString(String str) {

if (str == null)

return false;

int length = str.length();

System.out.println(length / 2);

for (int i = 0; i < length / 2; i++) {

if (str.charAt(i) != str.charAt(length - i - 1))

return false;

}

return true;

}

* Write a method that will remove given character from the String?

We can use replaceAll method to replace all the occurance of a String with another String. The important point to note is that it accepts String as argument, so we will use Character class to create String and use it to replace all the characters with empty String.

private static String removeChar(String str, char c) {

if (str == null)

return null;

return str.replaceAll(Character.toString(c), "");

}

* How to compare two Strings in java program?

Java String implements Comparable interface and it has two variants of compareTo() methods.

**compareTo(String anotherString)** method compares the String object with the String argument passed lexicographically. If String object precedes the argument passed, it returns negative integer and if String object follows the argument String passed, it returns positive integer. It returns zero when both the String have same value, in this case equals(String str) method will also return true.

**compareToIgnoreCase(String str)**: This method is similar to the first one, except that it ignores the case. It uses String CASE\_INSENSITIVE\_ORDER Comparator for case insensitive comparison. If the value is zero then equalsIgnoreCase(String str) will also return true.

Check this post for String compareTo example.

* How to convert String to char and vice versa?

This is a tricky question because String is a sequence of characters, so we can't convert it to a single character. We can use use charAt method to get the character at given index or we can use toCharArray()method to convert String to character array.

String class has three methods related to char. Let’s look at them before we look at a java program to convert string to char array.

char[] toCharArray(): This method converts string to character array. The char array size is same as the length of the string.

char charAt(int index): This method returns character at specific index of string. This method throws StringIndexOutOfBoundsException if the index argument value is negative or greater than the length of the string.

getChars(int srcBegin, int srcEnd, char dst[], int dstBegin): This is a very useful method when you want to convert part of string to character array. First two parameters define the start and end index of the string; the last character to be copied is at index srcEnd-1. The characters are copied into the char array starting at index dstBegin and ending at dstBegin + (srcEnd-srcBegin) – 1.

public class StringToCharJava {

public static void main(String[] args) {

String str = "journaldev";

//string to char array

char[] chars = str.toCharArray();

System.out.println(chars.length);

//char at specific index

char c = str.charAt(2);

System.out.println(c);

//Copy string characters to char array

char[] chars1 = new char[7];

str.getChars(0, 7, chars1, 0);

System.out.println(chars1);

}

}

* Does String is thread-safe in Java?

Strings are immutable, so we can't change it's value in program. Hence it's thread-safe and can be safely used in multi-threaded environment.

* Why String is popular HashMap key in Java?

Since String is immutable, its hashcode is cached at the time of creation and it doesn’t need to be calculated again. This makes it a great candidate for key in a Map and it’s processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.

* What does String intern() method do?

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

This method always return a String that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

* What is String Pool?

As the name suggests, String Pool is a pool of Strings stored in Java heap memory. We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes.

* Why String is immutable or final in Java

There are several benefits of String because it's immutable and final.

1. String Pool is possible because String is immutable in java.
2. It increases security because any hacker can't change its value and it's used for storing sensitive information such as database username, password etc.
3. Since String is immutable, it's safe to use in multi-threading and we don't need any synchronization.
4. Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader.
5. Since String is immutable, its hashcode is cached at the time of creation and it doesn’t need to be calculated again. This makes it a great candidate for key in a Map and it’s processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.

* Difference between String, StringBuffer and StringBuilder?

String is immutable and final in java, so whenever we do String manipulation, it creates a new String. String manipulations are resource consuming, so java provides two utility classes for String manipulations - StringBuffer and StringBuilder.

StringBuffer and StringBuilder are mutable classes. StringBuffer operations are thread-safe and synchronized where StringBuilder operations are not thread-safe. So when multiple threads are working on same String, we should use StringBuffer but in single threaded environment we should use StringBuilder.

StringBuilder performance is fast than StringBuffer because of no overhead of synchronization.

* + Java Program to find all Permutations of a String

To get all the permutations, we will first take out the first char from String and permute the remaining chars.

If String = “ABC”

First char = A and remaining chars permutations are BC and CB.

Now we can insert first char in the available positions in the permutations.

BC -> ABC, BAC, BCA

CB -> ACB, CAB, CBA

So we can write a recursive function call to return the permutations and then another function call to insert the first characters to get the complete list of permutations.

import java.util.HashSet;

import java.util.Set;

/\*\*

\* Java Program to find all permutations of a String

\* @author pankaj

\*

\*/

public class StringHelper {

public static Set<String> permutationFinder(String str) {

Set<String> perm = new HashSet<String>();

//Handling error scenarios

if (str == null) {

return null;

} else if (str.length() == 0) {

perm.add("");

return perm;

}

char initial = str.charAt(0); // first character

String rem = str.substring(1); // Full string without first character

Set<String> words = permutationFinder(rem);

for (String strNew : words) {

for (int i = 0;i<=strNew.length();i++){

perm.add(charInsert(strNew, initial, i));

}

}

return perm;

}

public static String charInsert(String str, char c, int j) {

String begin = str.substring(0, j);

String end = str.substring(j);

return begin + c + end;

}

public static void main(String[] args) {

String s = "AAC";

String s1 = "ABC";

String s2 = "ABCD";

System.out.println("\nPermutations for " + s + " are: \n" + permutationFinder(s));

System.out.println("\nPermutations for " + s1 + " are: \n" + permutationFinder(s1));

System.out.println("\nPermutations for " + s2 + " are: \n" + permutationFinder(s2));

}

}

* + Can we use String in switch case?

This is a tricky question used to check your knowledge of current Java developments. Java 7 extended the capability of switch case to use Strings also, earlier java versions doesn't support this.

If you are implementing conditional flow for Strings, you can use if-else conditions and you can use switch case if you are using Java 7 or higher versions.

* + What is the output of below program?

package com.journaldev.strings;

public class Test {

public void foo(String s) {

System.out.println("String");

}

public void foo(StringBuffer sb){

System.out.println("StringBuffer");

}

public static void main(String[] args) {

new Test().foo(null);

}

}

The above program will not compile with error as "The method foo(String) is ambiguous for the type Test".

* + What is the output of below code snippet?

String s1 = new String("abc");

String s2 = new String("abc");

System.out.println(s1 == s2);

It will print false because we are using new operator to create String, so it will be created in the heap memory and both s1, s2 will have different reference. If we create them using double quotes, then they will be part of string pool and it will print true.

* + What will be output of below code snippet?

String s1 = "abc";

StringBuffer s2 = new StringBuffer(s1);

System.out.println(s1.equals(s2));

It will print false because s2 is not of type String. If you will look at the equals method implementation in the String class, you will find a check using instanceof operator to check if the type of passed object is String? If not, then return false.

* + How many String objects got created in below code snippet?

String s1 = new String("Hello");

String s2 = new String("Hello");

Answer is 3.

First - line 1, "Hello" object in the string pool.

Second - line 1, new String with value "Hello" in the heap memory.

Third - line 2, new String with value "Hello" in the heap memory. Here "Hello" string from string pool is reused.

# **Java Multithreading Concurrency**

* What is the difference between Process and Thread?

A process is a self contained execution environment and it can be seen as a program or application whereas Thread is a single task of execution within the process. Java runtime environment runs as a single process which contains different classes and programs as processes. Thread can be called lightweight process. Thread requires less resources to create and exists in the process, thread shares the process resources.

* What are the benefits of multi-threaded programming?

In Multi-Threaded programming, multiple threads are executing concurrently that improves the performance because CPU is not idle incase some thread is waiting to get some resources. Multiple threads share the heap memory, so it’s good to create multiple threads to execute some task rather than creating multiple processes. For example, Servlets are better in performance than CGI because Servlet support multi-threading but CGI doesn’t.

* What is difference between user Thread and daemon Thread?

When we create a Thread in java program, it’s known as user thread. A daemon thread runs in background and doesn’t prevent JVM from terminating. When there are no user threads running, JVM shutdown the program and quits. A child thread created from daemon thread is also a daemon thread.

* How can we create a Thread in Java?

There are two ways to create Thread in Java – first by implementing Runnable interface and then creating a Thread object from it and second is to extend the Thread Class.

Java provides two ways to create a thread programmatically.

1. Implementing the java.lang.Runnable interface.
2. Extending the java.lang.Thread class.

**Java Thread Example – implementing Runnable interface**

To make a class runnable, we can implement java.lang.Runnable interface and provide implementation in public void run() method. To use this class as Thread, we need to create a Thread object by passing object of this runnable class and then call start() method to execute the run() method in a separate thread.

Here is a java thread example by implementing Runnable interface.

package com.journaldev.threads;

public class HeavyWorkRunnable implements Runnable {

@Override

public void run() {

System.out.println("Doing heavy processing - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Doing heavy processing - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

**Java Thread Example – extending Thread class**

We can extend java.lang.Thread class to create our own java thread class and override run() method. Then we can create it’s object and call start() method to execute our custom java thread class run method.

Here is a simple java thread example showing how to extend Thread class.

package com.journaldev.threads;

public class MyThread extends Thread {

public MyThread(String name) {

super(name);

}

@Override

public void run() {

System.out.println("MyThread - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("MyThread - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

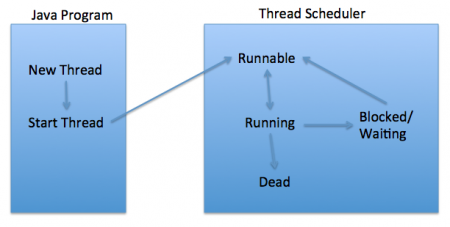
Thread.sleep(5000);

}

}

* What are different states in lifecycle of Thread?

Below diagram shows different states of thread life cycle in java. We can create a thread in java and start it but how the thread states change from Runnable to Running to Blocked depends on the OS implementation of thread scheduler and java doesn’t have full control on that.

[](http://cdn.journaldev.com/wp-content/uploads/2012/12/Thread-Lifecycle-States.png)

**New**

When we create a new Thread object using *new* operator, thread state is New Thread. At this point, thread is not alive and it’s a state internal to Java programming.

Runnable

When we call start() function on Thread object, it’s state is changed to Runnable. The control is given to Thread scheduler to finish it’s execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.

**Running**

When thread is executing, it’s state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it’s state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

**Blocked/Waiting**

A thread can be waiting for other thread to finish using [thread join](http://www.journaldev.com/1024/java-thread-join-example) or it can be waiting for some resources to available. For example [producer consumer problem](http://www.journaldev.com/1034/java-blockingqueue-example) or [waiter notifier implementation](http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example) or IO resources, then it’s state is changed to Waiting. Once the thread wait state is over, it’s state is changed to Runnable and it’s moved back to runnable thread pool.

**Dead**

Once the thread finished executing, it’s state is changed to Dead and it’s considered to be not alive.

Above are the different **states of thread**. It’s good to know them and how thread changes it’s state. That’s all for thread life cycle in java.

* Can we call run() method of a Thread class?

Yes, we can call run() method of a Thread class but then it will behave like a normal method. To actually execute it in a Thread, we need to start it using **Thread.start()** method.

* How can we pause the execution of a Thread for specific time?

We can use Thread class sleep() method to pause the execution of Thread for certain time. Note that this will not stop the processing of thread for specific time, once the thread awake from sleep, it’s state gets changed to runnable and based on thread scheduling, it gets executed.

* What do you understand about Thread Priority?

Every thread has a priority, usually higher priority thread gets precedence in execution but it depends on Thread Scheduler implementation that is OS dependent. We can specify the priority of thread but it doesn’t guarantee that higher priority thread will get executed before lower priority thread. Thread priority is an int whose value varies from 1 to 10 where 1 is the lowest priority thread and 10 is the highest priority thread.

* What is Thread Scheduler and Time Slicing?

Thread Scheduler is the Operating System service that allocates the CPU time to the available runnable threads. Once we create and start a thread, it’s execution depends on the implementation of Thread Scheduler. Time Slicing is the process to divide the available CPU time to the available runnable threads. Allocation of CPU time to threads can be based on thread priority or the thread waiting for longer time will get more priority in getting CPU time. Thread scheduling can’t be controlled by java, so it’s always better to control it from application itself.

* What is context-switching in multi-threading?

Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time. Context Switching is the essential feature for multitasking operating system and support for multi-threaded environment.

* How can we make sure main() is the last thread to finish in Java Program?

We can use Thread join() method to make sure all the threads created by the program is dead before finishing the main function.

public final void join(): This java thread join method puts the current thread on wait until the thread on which it’s called is dead. If the thread is interrupted, it throws InterruptedException.

public final synchronized void join(long millis): This java thread join method is used to wait for the thread on which it’s called to be dead or wait for specified milliseconds. Since thread execution depends on OS implementation, it doesn’t guarantee that the current thread will wait only for given time.

public final synchronized void join(long millis, int nanos): This java thread join method is used to wait for thread to die for given milliseconds plus nanoseconds.

Here is a simple example showing usage of Thread join methods. The goal of the program is to make sure main is the last thread to finish and third thread starts only when first one is dead.

package com.journaldev.threads;

public class ThreadJoinExample {

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable(), "t1");

Thread t2 = new Thread(new MyRunnable(), "t2");

Thread t3 = new Thread(new MyRunnable(), "t3");

t1.start();

//start second thread after waiting for 2 seconds or if it's dead

try {

t1.join(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

t2.start();

//start third thread only when first thread is dead

try {

t1.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

t3.start();

//let all threads finish execution before finishing main thread

try {

t1.join();

t2.join();

t3.join();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("All threads are dead, exiting main thread");

}

}

class MyRunnable implements Runnable{

@Override

public void run() {

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

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

e.printStackTrace();

}

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

}

}

* How does thread communicate with each other?

When threads share resources, communication between Threads is important to coordinate their efforts. Object class wait(), notify() and notifyAll() methods allows threads to communicate about the lock status of a resource.

**wait**

Object wait methods has three variance, one which waits indefinitely for any other thread to call notify or notifyAll method on the object to wake up the current thread. Other two variances puts the current thread in wait for specific amount of time before they wake up.

**notify**

notify method wakes up only one thread waiting on the object and that thread starts execution. So if there are multiple threads waiting for an object, this method will wake up only one of them. The choice of the thread to wake depends on the OS implementation of thread management.

**notifyAll**

notifyAll method wakes up all the threads waiting on the object, although which one will process first depends on the OS implementation.

These methods can be used to implement producer consumer problem where consumer threads are waiting for the objects in Queue and producer threads put object in queue and notify the waiting threads.

Let’s see an example where multiple threads work on the same object and we use wait, notify and notifyAll methods.

Message

A java bean class on which threads will work and call wait and notify methods.

package com.journaldev.concurrency;

public class Message {

private String msg;

public Message(String str){

this.msg=str;

}

public String getMsg() {

return msg;

}

public void setMsg(String str) {

this.msg=str;

}

}

Waiter

A class that will wait for other threads to invoke notify methods to complete it’s processing. Notice that Waiter thread is owning monitor on Message object using synchronized block.

package com.journaldev.concurrency;

public class Waiter implements Runnable{

private Message msg;

public Waiter(Message m){

this.msg=m;

}

@Override

public void run() {

String name = Thread.currentThread().getName();

synchronized (msg) {

try{

System.out.println(name+" waiting to get notified at time:"+System.currentTimeMillis());

msg.wait();

}catch(InterruptedException e){

e.printStackTrace();

}

System.out.println(name+" waiter thread got notified at time:"+System.currentTimeMillis());

//process the message now

System.out.println(name+" processed: "+msg.getMsg());

}

}

}

**Notifier**

A class that will process on Message object and then invoke notify method to wake up threads waiting for Message object. Notice that synchronized block is used to own the monitor of Message object.

package com.journaldev.concurrency;

public class Notifier implements Runnable {

private Message msg;

public Notifier(Message msg) {

this.msg = msg;

}

@Override

public void run() {

String name = Thread.currentThread().getName();

System.out.println(name+" started");

try {

Thread.sleep(1000);

synchronized (msg) {

msg.setMsg(name+" Notifier work done");

msg.notify();

// msg.notifyAll();

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**WaitNotifyTest**

Test class that will create multiple threads of Waiter and Notifier and start them.

package com.journaldev.concurrency;

public class WaitNotifyTest {

public static void main(String[] args) {

Message msg = new Message("process it");

Waiter waiter = new Waiter(msg);

new Thread(waiter,"waiter").start();

Waiter waiter1 = new Waiter(msg);

new Thread(waiter1, "waiter1").start();

Notifier notifier = new Notifier(msg);

new Thread(notifier, "notifier").start();

System.out.println("All the threads are started");

}

}

When we will invoke the above program, we will see below output but program will not complete because there are two threads waiting on Message object and notify() method has wake up only one of them, the other thread is still waiting to get notified.

waiter waiting to get notified at time:1356318734009

waiter1 waiting to get notified at time:1356318734010

All the threads are started

notifier started

waiter waiter thread got notified at time:1356318735011

waiter processed: notifier Notifier work done

If we comment the notify() call and uncomment the notifyAll() call in Notifier class, below will be the output produced.

waiter waiting to get notified at time:1356318917118

waiter1 waiting to get notified at time:1356318917118

All the threads are started

notifier started

waiter1 waiter thread got notified at time:1356318918120

waiter1 processed: notifier Notifier work done

waiter waiter thread got notified at time:1356318918120

waiter processed: notifier Notifier work done

Since notifyAll() method wake up both the Waiter threads and program completes and terminates after execution. That’s all for wait, notify and notifyAll in java.

* Why thread communication methods wait(), notify() and notifyAll() are in Object class?

In Java every Object has a monitor and wait, notify methods are used to wait for the Object monitor or to notify other threads that Object monitor is free now. There is no monitor on threads in java and synchronization can be used with any Object, that’s why it’s part of Object class so that every class in java has these essential methods for inter thread communication.

* Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?

When a Thread calls wait() on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call notify() on this Object. Similarly when a thread calls notify() on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object. Since all these methods require Thread to have the Object monitor, that can be achieved only by synchronization, they need to be called from synchronized method or block.

* Why Thread sleep() and yield() methods are static?

Thread sleep() and yield() methods work on the currently executing thread. So there is no point in invoking these methods on some other threads that are in wait state. That’s why these methods are made static so that when this method is called statically, it works on the current executing thread and avoid confusion to the programmers who might think that they can invoke these methods on some non-running threads.

* What is volatile keyword in Java

When we use volatile keyword with a variable, all the threads read it’s value directly from the memory and don’t cache it. This makes sure that the value read is the same as in the memory.

* How to create daemon thread in Java?

Thread class setDaemon(true) can be used to create daemon thread in java. We need to call this method before calling start() method else it will throw IllegalThreadStateException.

* What is Thread Pool? How can we create Thread Pool in Java?

A thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed.

A thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue.

java.util.concurrent.Executors provide implementation of java.util.concurrent.Executor interface to create the thread pool in java.

**Java thread pool** manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed. We can use ThreadPoolExecutor to create thread pool in java.

[](http://cdn.journaldev.com/wp-content/uploads/2012/12/threadpoolexecutor-example-executorservice-java-thread-pool.jpg)

Java thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue. **java.util.concurrent.Executors** provide implementation of **java.util.concurrent.Executor**interface to create the thread pool in java. Let’s write a simple program to explain it’s working.

First we need to have a Runnable class, named WorkerThread.java

package com.journaldev.threadpool;

public class WorkerThread implements Runnable {

private String command;

public WorkerThread(String s){

this.command=s;

}

@Override

public void run() {

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

processCommand();

System.out.println(Thread.currentThread().getName()+" End.");

}

private void processCommand() {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public String toString(){

return this.command;

}

}

## ExecutorService Example

Here is the test program class SimpleThreadPool.java, where we are creating fixed thread pool from **Executors framework**.

package com.journaldev.threadpool;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class SimpleThreadPool {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(5);

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

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

executor.execute(worker);

}

executor.shutdown();

while (!executor.isTerminated()) {

}

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

}

}

In above program, we are creating fixed size thread pool of 5 worker threads. Then we are submitting 10 jobs to this pool, since the pool size is 5, it will start working on 5 jobs and other jobs will be in wait state, as soon as one of the job is finished, another job from the wait queue will be picked up by worker thread and get's executed.

Here is the output of the above program.

pool-1-thread-2 Start. Command = 1

pool-1-thread-4 Start. Command = 3

pool-1-thread-1 Start. Command = 0

pool-1-thread-3 Start. Command = 2

pool-1-thread-5 Start. Command = 4

pool-1-thread-4 End.

pool-1-thread-5 End.

pool-1-thread-1 End.

pool-1-thread-3 End.

pool-1-thread-3 Start. Command = 8

pool-1-thread-2 End.

pool-1-thread-2 Start. Command = 9

pool-1-thread-1 Start. Command = 7

pool-1-thread-5 Start. Command = 6

pool-1-thread-4 Start. Command = 5

pool-1-thread-2 End.

pool-1-thread-4 End.

pool-1-thread-3 End.

pool-1-thread-5 End.

pool-1-thread-1 End.

Finished all threads

The output confirms that there are five threads in the pool named from "pool-1-thread-1" to "pool-1-thread-5" and they are responsible to execute the submitted tasks to the pool.

## ThreadPoolExecutor Example

**Executors** class provide simple implementation of **ExecutorService** using **ThreadPoolExecutor** but ThreadPoolExecutor provides much more feature than that. We can specify the number of threads that will be alive when we create ThreadPoolExecutor instance and we can limit the size of thread pool and create our own **RejectedExecutionHandler** implementation to handle the jobs that can't fit in the worker queue.

Here is our custom implementation of RejectedExecutionHandler interface.

package com.journaldev.threadpool;

import java.util.concurrent.RejectedExecutionHandler;

import java.util.concurrent.ThreadPoolExecutor;

public class RejectedExecutionHandlerImpl implements RejectedExecutionHandler {

@Override

public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {

System.out.println(r.toString() + " is rejected");

}

}

ThreadPoolExecutor provides several methods using which we can find out the current state of executor, pool size, active thread count and task count. So I have a monitor thread that will print the executor information at certain time interval.

package com.journaldev.threadpool;

import java.util.concurrent.ThreadPoolExecutor;

public class MyMonitorThread implements Runnable

{

private ThreadPoolExecutor executor;

private int seconds;

private boolean run=true;

public MyMonitorThread(ThreadPoolExecutor executor, int delay)

{

this.executor = executor;

this.seconds=delay;

}

public void shutdown(){

this.run=false;

}

@Override

public void run()

{

while(run){

System.out.println(

String.format("[monitor] [%d/%d] Active: %d, Completed: %d, Task: %d, isShutdown: %s, isTerminated: %s",

this.executor.getPoolSize(),

this.executor.getCorePoolSize(),

this.executor.getActiveCount(),

this.executor.getCompletedTaskCount(),

this.executor.getTaskCount(),

this.executor.isShutdown(),

this.executor.isTerminated()));

try {

Thread.sleep(seconds\*1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

Here is the thread pool implementation example using **ThreadPoolExecutor**.

package com.journaldev.threadpool;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.Executors;

import java.util.concurrent.ThreadFactory;

import java.util.concurrent.ThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

public class WorkerPool {

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

//RejectedExecutionHandler implementation

RejectedExecutionHandlerImpl rejectionHandler = new RejectedExecutionHandlerImpl();

//Get the ThreadFactory implementation to use

ThreadFactory threadFactory = Executors.defaultThreadFactory();

//creating the ThreadPoolExecutor

ThreadPoolExecutor executorPool = new ThreadPoolExecutor(2, 4, 10, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(2), threadFactory, rejectionHandler);

//start the monitoring thread

MyMonitorThread monitor = new MyMonitorThread(executorPool, 3);

Thread monitorThread = new Thread(monitor);

monitorThread.start();

//submit work to the thread pool

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

executorPool.execute(new WorkerThread("cmd"+i));

}

Thread.sleep(30000);

//shut down the pool

executorPool.shutdown();

//shut down the monitor thread

Thread.sleep(5000);

monitor.shutdown();

}

}

Notice that while initializing the ThreadPoolExecutor, we are keeping initial pool size as 2, maximum pool size to 4 and work queue size as 2. So if there are 4 running tasks and more tasks are submitted, the work queue will hold only 2 of them and rest of them will be handled by RejectedExecutionHandlerImpl.

Here is the output of above program that confirms above statement.

pool-1-thread-1 Start. Command = cmd0

pool-1-thread-4 Start. Command = cmd5

cmd6 is rejected

pool-1-thread-3 Start. Command = cmd4

pool-1-thread-2 Start. Command = cmd1

cmd7 is rejected

cmd8 is rejected

cmd9 is rejected

[monitor] [0/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false

[monitor] [4/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false

pool-1-thread-4 End.

pool-1-thread-1 End.

pool-1-thread-2 End.

pool-1-thread-3 End.

pool-1-thread-1 Start. Command = cmd3

pool-1-thread-4 Start. Command = cmd2

[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false

[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false

pool-1-thread-1 End.

pool-1-thread-4 End.

[monitor] [4/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true

[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true

Notice the change in active, completed and total completed task count of the executor. We can invoke **shutdown()** method to finish execution of all the submitted tasks and terminate the thread pool.

If you want to schedule a task to run with delay or periodically then you can use **ScheduledThreadPoolExecutor** class