Multithreading in Java

<https://www.geeksforgeeks.org/daemon-thread-java/>

<https://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html#axzz6D4dg0VM4>

<https://www.geeksforgeeks.org/inter-thread-communication-java/>

<https://www.javatpoint.com/java-multithreading-interview-questions>

What is a concurrent programming language?

A **concurrent programming language** is defined as one which uses the concept of simultaneously executing processes or threads of execution as a means of structuring a program. A **parallel language** is able to express programs that are executable on more than one processor.

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

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

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

However, we use multithreading than multiprocessing because threads use a shared 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 the 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 an exception occurs in a single thread.

## **Multitasking**

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

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

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

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

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

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

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

## **What is Thread in java**

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

#### **Note: At a time one thread is executed only.**

# Multithreading in Java

1. [Multithreading](https://www.javatpoint.com/multithreading-in-java)
2. [Multitasking](https://www.javatpoint.com/multithreading-in-java#multitasing)
3. [Process-based multitasking](https://www.javatpoint.com/multithreading-in-java#multiprocessing)
4. [Thread-based multitasking](https://www.javatpoint.com/multithreading-in-java#multithreading)
5. [What is Thread](https://www.javatpoint.com/multithreading-in-java#thread)

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

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

However, we use multithreading than multiprocessing because threads use a shared 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 the 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 an exception occurs in a single thread.

## **Multitasking**

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

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

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

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

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

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

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



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

#### **Note: At a time one thread is executed only.**

## **Java Thread class**

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

# Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

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

### **2) Runnable**

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### **3) Running**

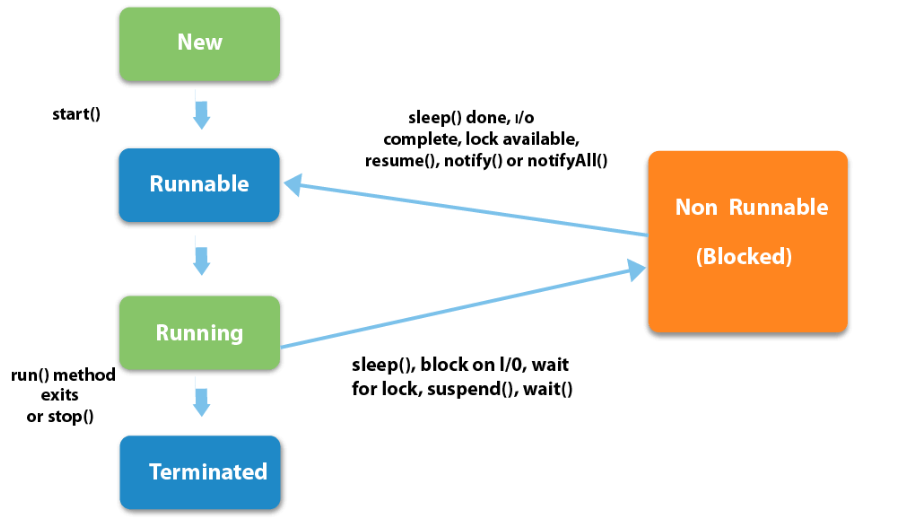
The thread is in running state if the thread scheduler has selected it.

### **4) Non-Runnable (Blocked)**

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

### **5) Terminated**

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



|  |  |  |
| --- | --- | --- |
| How to create thread There are two ways to create a thread:   1. By extending Thread class 2. By implementing Runnable interface.  **Thread class:**  |  | | --- | | Thread class provide constructors and methods to create and perform operations on a  thread.Thread class extends Object class and implements Runnable interface. |  **Commonly used Constructors of Thread class:**  |  | | --- | | * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) | |

### **Commonly used methods of Thread class:**

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

### **Some Important points to Remember**

1. When we extend Thread class, we cannot override **setName()** and **getName()** functions, because they are declared final in Thread class.
2. While using **sleep()**, always handle the exception it throws.

*static* void **sleep**(long *milliseconds*) throws **InterruptedException**

# 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.

### **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.

Daemon thread in Java

Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.

**Properties:**

* They can not prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution
* If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.
* It is an utmost low priority thread.

**Exceptions in Daemon thread**

* If you call the setDaemon() method after starting the thread, it would throw **IllegalThreadStateException**.

**Daemon vs User Threads**

1. **Priority:** When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
2. **Usage:** Daemon thread is to provide services to user thread for background supporting task.

**Q) How to make a thread (user thread) to Daemon thread?**  
A) By calling setDaemon() method we can make a user thread to daemon thread.  
Syntax: thread.setDaemon(true);

**Q) What is difference between user thread and Daemon thread?**  
A) By default a thread created in a program is always a user thread, however we can make it daemon by calling setDaemon(true) method, if needed. A daemon thread runs in a background and it doesn’t prevent JVM from being shutdown. Once all the user thread gets completed the JVM shutdowns without being bothered whether a daemon thread is running or not.

### **What do you understand by inter-thread communication?**

* The process of communication between synchronized threads is termed as inter-thread communication.
* Inter-thread communication is used to avoid thread polling in Java.
* The 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 can be obtained by wait(), notify(), and notifyAll() methods

**What is Polling and what are problems with it?**

The process of testing a condition repeatedly till it becomes true is known as polling.

Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain action is taken. This waste many CPU cycles and makes the implementation inefficient.  
For example, in a classic queuing problem where one thread is producing data and other is consuming it.

**How Java multi threading tackles this problem?**

To avoid polling, Java uses three methods, namely, **wait(), notify() and notifyAll().**  
All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

* **wait()-**It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
* **notify()-**It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.
* **notifyAll()-**It wakes up all the threads that called wait() on the same object.

what is thread safe concept ?

**Thread safety** is a [computer programming](https://en.wikipedia.org/wiki/Computer_programming) concept applicable to [multi-threaded](https://en.wikipedia.org/wiki/Thread_(computing)) code. Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. There are various strategies for making thread-safe data structures.

What is a thread safe class?

**thread**-**safety** or **thread**-**safe** code in Java refers to code which can safely be used or shared in concurrent or multi-**threading** environment and they will behave as expected. any code, **class** or object which can behave differently from its contract on concurrent environment is not **thread**-**safe.**

what is concurrent concept ?

A **concurrent** program is one defining actions that may be performed simultaneously. ... Particular attention is paid to programs that can be considered inherently **concurrent**, that is, programs that are constructed to control or model physical systems that involve parallel activity.

## What makes java application concurrent?

The very first class, you will need to make a java class concurrent, is [java.lang.Thread](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html" \t "_blank) class. This class is the basis of all concurrency concepts in java. Then you have [java.lang.Runnable](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html" \t "_blank) interface to abstract the thread behavior out of thread class.

Other classes you will need to build advance applications can be found at [java.util.concurrent](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html" \t "_blank) package added in Java 1.5

Thread Class

public class ****Thread****

extends [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

implements [Runnable](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html)

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

* The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
* All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

class PrimeThread extends Thread {

long minPrime;

PrimeThread(long minPrime) {

this.minPrime = minPrime;

}

public void run() {

// compute primes larger than minPrime

 . . .

}

}

The following code would then create a thread and start it running:

PrimeThread p = new PrimeThread(143);

p.start();

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

## Difference between Runnable vs Thread

There has been a good amount of debate on which is better way. Well, I also tried to find out and below is my learning.

1. Implementing Runnable is the preferred way to do it. Here, you’re not really specializing or modifying the thread’s behavior. You’re just giving the thread something to run. That means composition is the better way to go.
2. Java only supports single inheritance, so you can only extend one class.
3. Instantiating an interface gives a cleaner separation between your code and the implementation of threads.
4. Implementing Runnable makes your class more flexible. If you extend Thread then the action you’re doing is always going to be in a thread. However, if you implement Runnable it doesn’t have to be. You can run it in a thread, or pass it to some kind of executor service, or just pass it around as a task within a single threaded application.
5. If you are working on JDK 4 or lesser, then there is bug :

[http://bugs.java.com/bugdatabase/view\_bug.do;jsessionid=5869e03fee226ffffffffc40d4fa881a86e3:WuuT?bug\_id=4533087](http://bugs.java.com/bugdatabase/view_bug.do;jsessionid=5869e03fee226ffffffffc40d4fa881a86e3:WuuT?bug_id=4533087%20terget=)

It’s fixed in Java 1.5 but Sun doesn’t intend to fix it in 1.4.

The issue is that at construction time, a Thread is added to a list of references in an internal thread table. It won’t get removed from that list until its start() method has completed. As long as that reference is there, it won’t get garbage collected.

### Difference between sleep(), suspend() and wait()

**sleep()** is a static method that is used to send the calling thread into a non-runnable state for the given duration of time. The important part for this is recognizing the “calling thread”, which is actually the thread in which the sleep() method is invoked rather than the thread object which may (which is essentially a violation of Java standards) invoke this method. What this means is, while calling Thread.sleep(<duration in milliseconds>) is appropriate and will send the invoking thread to non-runnable state, but calling t.sleep(<duration in milliseconds>) is inappropriate as it would cause the thread in which this call appears to go to non-runnable state instead of the “t” thread whose object is invoking the method. While in “sleep”, the thread will keep all the monitor locks which it might be holding at the time of sleep invocation.

The t.**suspend()** method has been **deprecated**, as it is inherently deadlock-prone.It suspends the thread on which it is invoked. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results.

t.**wait()** Causes current thread to 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 difference between sleep() and wait() is that sleep() is called on a thread while wait() is called on an object (That's why it is part of Object class. The parent to all Java classes). The wait() method causes the current thread (call it T) to place itself in the wait set for this object and then to relinquish any and all synchronization claims on this object. Thread *T* becomes disabled for thread scheduling purposes. **Later, another thread can synchronize on the same lock object and call lock.notify(). This wakes up the original, waiting thread *T*.**