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

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 subrocess.
* Cost of communicatin between the thread is low.
* Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

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

======================================================

* 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.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

# 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.

Naming Thread

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

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

# 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.

Volatile Keyword vs Synchronized

Important points

* You can use the volatile keyword with variables. Using volatile keyword with classes and methods is illegal.
* It guarantees that value of the volatile variable will always be read from the main memory, not from the local thread cache.
* If you declared variable as volatile, Read and Writes are atomic
* It reduces the risk of memory consistency error.
* Any write to volatile variable in Java establishes a happen before the relationship with successive reads of that same variable.
* The volatile variables are always visible to other threads.
* The volatile variable that is an object reference may be null.
* When a variable is not shared between multiple threads, you do not need to use the volatile keyword with that variable.

It is a alternate way to achieve synchronization. Volatile is better than synchronized.

We use synchronisation when multiple threads are accessing the data . Each thread work on its local memory . So when we are using a variable its value should be same for all threads but different thread read the variable at different time and modifies it so it causes inconsistency in data . So we use Synchronisation but the problem with the synchronisation is that it can be applied with block or method only. So other code will get blocked. So solution is volatile keyword.

Varibale always will be read from main memory instead of thread local copy.

Variable a

Main memory

Local copy

Local copy

T22

T1

Collable

There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable. However, one feature lacking in  Runnable is that we cannot make a thread return result when it terminates, i.e. when run() completes. For supporting this feature, the Callable interface is present in Java.

For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion. Note that a thread can’t be created with a Callable, it can only be created with a Runnable.

* Another difference is that the call() method can throw an exception whereas run() cannot.
* Method signature that has to overridden for implementing Callable.
* public Object call() throws Exception;
* **Future-interface**
* When the call() method completes, answer must be stored in an object known to the main thread, so that the main thread can know about the result that the thread returned. How will the program store and obtain this result later? For this, a Future object can be used. Think of a Future as an object that holds the result – it may not hold it right now, but it will do so in the future (once the Callable returns). Thus, a Future is basically one way the main thread can keep track of the progress and result from other threads. To implement this interface, 5 methods have to be overridden, but as the example below uses a concrete implementation from the library, only the important methods are listed here

Atomic

* the primary use of AtomicInteger is when we are in **multi-threaded context** and we need to perform **atomic operations** on an int value without using [synchronized](https://howtodoinjava.com/java/multi-threading/java-synchronized/) keyword.
* Using the AtomicInteger is equally faster and more readable than performing the same using synchronization.