1. Thread has its own variables and methods; it lives and dies on the heap. But a thread of execution is an individual process that has its own call stack.

Below point need to be cleared.

ExecutorService : If we implement Runnable, we can start multiple thread created on runnable object with ExecutorService (because we can start Runnable object with new threads), but not in the case when we extend Thread (because thread can be started only once).

**Volatile keyword**

**Volatile variable is not Cached in CPU memory**

Volatile members are never cached in CPU by jvm, they are always read from main memory i.e. from stack where variable lives.

It is possible for multiple CPU’s to exist on machine, so it is possibility that thread might cache different values in different CPU’s for same variable, so it’s important that value is not cached in CPU and always read from main memory.

Let’s discuss a scenario where non-volatile variable is used :

int x=0; // Let’s say we have variable x with value 0, in main memory value of x is 0 and in cache value of x is also 0.

x++; //increment done by thread1, so now in main memory value of x is 1 but it might happen somehow that value is not updated in cache, so value of x in cache is still 0.

Now, lets say thread2 try to read that value, as value is cached, it will read read value from cache only, so value read of x will be 0 (but actual value of x is 1), which of course is a synchronization problem.

Let’s discuss a scenario where volatile variable is used :

int x=0; // Let’s say we have variable x with value 0, in main memory value of x is 0 and it’s not cached (as volatile variables are not cached).

x++; //increment done by thread1, so now in main memory value of x is 1 and it’s not cached.

Now, lets say thread2 try to read that value, as value is not cached, it will read read value from main memory, so value read of x will be 1 (and actual value of x is 1), we have solved synchronization problem.

**10 Key points about volatile keyword**

Java allows threads to access shared variables. As a rule, to ensure that shared variables are consistently updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that enforces mutual exclusion for those shared variables. We can ensure such behaviour by using volatile keyword.

If a field is declared volatile, in that case the Java memory model ensures that all threads see a consistent value for the variable.

Volatile can be used as a keyword against the variable, we cannot use volatile against method declaration.

volatile void method2() //it’s illegal

volatile int i; //legal

In certain cases, Volatile keyword can be used as an alternate to synchronization in java, as all threads always have access to latest updated value.

Using volatile is better than synchronization, as synchronization needs to block whole method (if used in method declaration) or block (if synchronization block is used), while volatile needs not to block any piece of code.

Not Cached in CPU- Volatile members are never cached in CPU by jvm, they are always read from main memory i.e. from stack where variable lives.

It is possible for multiple CPU’s to exist on machine, so it is possibility that thread might cache different values in different CPU’s for same variable, so it’s important that value is not cached in CPU and always read from main memory.

Volatile keyword must be used in multithreading environment, there is no use of using volatile keyword in non multi threading environment, it may cost us unnecessary performance issue as volatile keyword is not cached in memory by jvm.

A compile-time error will occur if a final variable is declared volatile.

volatile final int x = 0; //The field x can be either final or volatile, not both.

If variable which has been declared volatile, is a reference to object it may point to null as well,

volatile Integer i=null; //it’s allowed.

Performance issue - As volatile keyword is not cached in CPU, it is always read from main memory, so in terms of performance it’s always expensive to use volatile keyword.

Q. In Java thread is created in which memory (Heap or Stack)?

Q. Thread is executed in which memory?

Q. Thread Memory management?

**yield method()**

* Definition : yield() method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint.
* Thread state : when yield() method is called on thread it goes from running to runnable state, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at the discretion of the implementation.
* Exception : yield() method doesn’t throws any exception.
* Waiting time : yield() method stops thread for unpredictable time.
* Static method : yield() is a static method, hence calling Thread.yield() causes currently executing thread to yield.
* Native method : implementation of yield() method is provided by JVM.
* Let’s see definition of yield() method as given in java.lang.Thread -
* public static native void yield();
* synchronized block : thread need not to to acquire object lock before calling yield() method i.e. yield() method can be called from outside synchronized block.
* Belongs to which class : yield() method belongs to java.lang.Thread class.