Thread execute sequence

```
Thread thread1 = createThread("T1", "run1", 3);
Thread thread2 = createThread("T2", "run2", 1);
Thread thread3 = createThread("T3", "run3", 2);
```

what case will show:

```
run1
T1sleep
T1wake
run3
T3sleep
T3wake
run2
T2sleep
T2wake
```

A:

```
thread1.start();
thread3.start();
thread2.start();

thread1.join();
thread3.join();
thread2.join();
```

B:

```
thread1.join();
thread3.join();
thread2.join();

thread1.start();
thread3.start();
thread2.start();
```

C:

yours

Difference between start and run

difference between `start`() and `run()` method in Thread is that `start` creates new thread while run doesn't create any thread and simply execute in current thread like a normal method call.

Producer Consumer Pattern

Benefit of Producer Consumer Pattern

1) Producer Consumer Pattern simple development. you can Code Producer and Consumer independently and Concurrently, they just need to know shared object.

2) Producer doesn't need to know about who is consumer or how many consumers are there. Same is true with Consumer.

3) Producer and Consumer can work with different speed. There is no risk of Consumer consuming half-baked item.
In fact by monitoring consumer speed one can introduce more consumer for better utilization.

4) Separating producer and Consumer functionality result in more clean, readable and manageable code.

Read more: http://javarevisited.blogspot.com/2012/02/producer-consumer-design-pattern-with.html#ixzz4dHNL8sPU

Thread-Safe Code

```
/*
 * Non Thread-Safe Class in Java
 */
public class Counter {

    private int count;

    /*
     * This method is not thread-safe because ++ is not an atomic operation
     */
    public int getCount(){
        return count++;
    }
}
```

Read more: http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html#ixzz4dHpKQDfk

A thread-safe version of Counter class in Java:

```
/*
 * Thread-Safe Example in Java
 */
public class Counter {

    private int count;
    AtomicInteger atomicCount = new AtomicInteger( 0 );


    /*
     * This method thread-safe now because of locking and synchornization
     */
    public synchronized int getCount(){
        return count++;
    }

    /*
     * This method is thread-safe because count is incremented atomically
     */
    public int getCountAtomically(){
        return atomicCount.incrementAndGet();
    }
}
```

Important points about Thread-Safety in Java

.) Immutable objects are by default thread-safe because there state can not be modified once created. Since String is immutable in Java, its inherently thread-safe.

2) Read only or final variables in Java are also thread-safe in Java.

3) Locking is one way of achieving thread-safety in Java.

4) Static variables if not synchronized properly becomes major cause of thread-safety issues.

5) Example of thread-safe class in Java: Vector, Hashtable, ConcurrentHashMap, String etc.

6) Atomic operations in Java are thread-safe e.g. reading a 32 bit int from memory because its an atomic operation it can't interleave with other thread.

7) local variables are also thread-safe because each thread has there own copy and using local variables is good way to writing thread-safe code in Java.

8) In order to avoid thread-safety issue minimize sharing of objects between multiple thread.

9) Volatile keyword in Java can also be used to instruct thread not to cache variables and read from main memory and can also instruct JVM not to reorder or optimize code from threading perspective.

Blocking methods

Read more: http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html#ixzz4dHxaEmxu

```
public class BlcokingCallTest {


    public static void main(String args[]) throws FileNotFoundException, IOException  {
      System.out.println("Calling blocking method in Java");
      int input = System.in.read();
      System.out.println("Blocking method is finished");
    }
}
```

Examples of blocking methods in Java:

1) **`InputStream.read()`** which blocks until input data is available, an exception is thrown or end of Stream is detected.
2) **`ServerSocket.accept()`** which listens for incoming socket connection in Java and blocks until a connection is made.
3) **`InvokeAndWait()`** wait until code is executed from Event Dispatcher thread.

Blocking methods

Best practices while calling blocking method in Java:

1) Always let separate worker thread handles time consuming operations e.g. reading and writing to file, database or socket.

2) Use **timeout** while calling blocking method. so if your blocking call doesn't return in specified time period, consider aborting it and returning back but again this depends upon scenario. if you are using Executor Framework for managing your worker threads, which is by the way recommended way than you can use `Future` object whose get() methods support timeout, but ensure that you properly terminate a blocking call.

3) Extension of first practices, don't call blocking methods on `keyPressed`() or `paint`() method which are supposed to return as quickly as possible.

4) Use call-back functions to process result of a blocking call.

5) If you are writing GUI application may be in Swing **never call blocking method in Event dispatcher thread** or in the event handler. for example if you are reading a file or opening a network connection when a button is clicked don't do that on `actionPerformed()` method, instead just create another worker thread to do that job and return from `actionPerformed()`. this will keep your GUI responsive, but again it depends upon design if the operation is something which requires user to wait than consider using `invokeAndWait()` for synchronous update.

Thread sleep()

1) Thread.sleep() method is used to **pause the execution, relinquish the CPU and return it to thread** scheduler.

2) Thread.The sleep() method is a static method and always *puts the current thread to sleep*.

3) Java has two variants of sleep method in Thread class one with one argument which takes milliseconds as the duration of sleep and another method with two arguments one is millisecond and other is the nanosecond.

4) Unlike wait() method in Java, sleep() method of Thread class **doesn't relinquish the lock** it has acquired.

5) sleep() method **throws Interrupted Exception** if another thread interrupts a sleeping thread in java.

6) With sleep() in Java it's not guaranteed that when sleeping thread woke up it will definitely get CPU, instead it will go to Runnable state and fight for CPU with other thread.

7) There is a **misconception** about sleep method in Java that calling t.sleep() will put Thread "t" into sleeping state, that's not true because Thread.sleep method is a static method it always put the current thread into Sleeping state and not thread "t".

The Volatile variable

Read more: http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html#ixzz4dMPEBp5Z

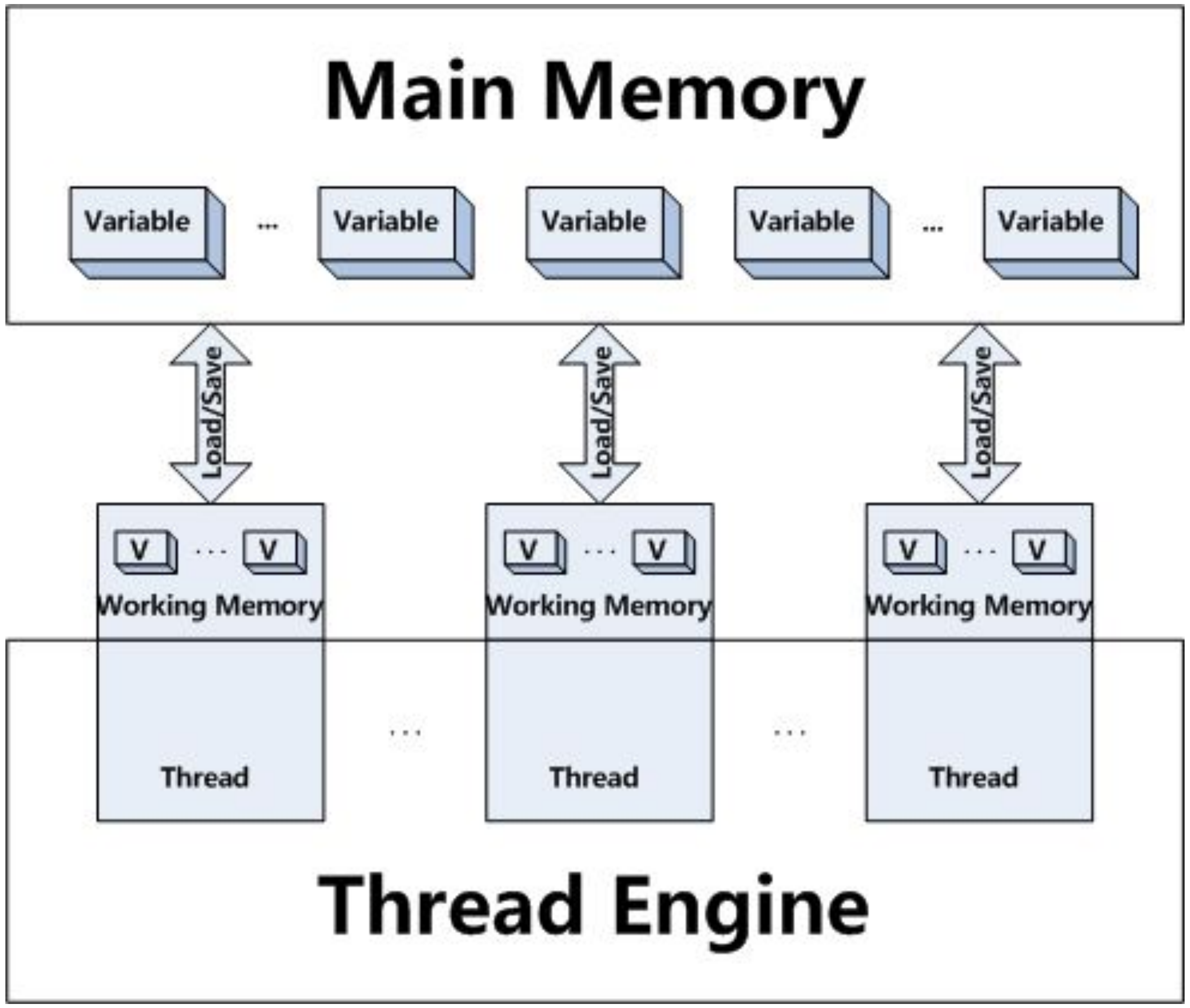
The volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory

```
/** * Java program to demonstrate where to use Volatile keyword in Java. * In this example Singleton Instance is
declared as volatile variable to ensure * every thread see updated value for _instance. * * @author Javin Paul */
public class Singleton{
    private static volatile Singleton _instance; //volatile variable
    public static Singleton getInstance() {
        if(_instance == null) {
            synchronized(Singleton.class) {
                if(_instance == null)
                    _instance = new Singleton();
            }
        }
        return _instance;
}
```

If you look at the code carefully you will be able to figure out:
1) We are only creating instance one time
2) We are creating instance lazily at the time of the first request comes.

The Volatile variable

If we do not make the `_instance` variable *volatile* than the Thread which is creating instance of Singleton is not able to communicate other thread, that instance has been created until it comes out of the Singleton block, so if Thread A is creating Singleton instance and just after creation lost the CPU, all other thread will not be able to see value of `_instance` as not null and they will believe its still null.



Java volatile keyword doesn't mean atomic, its common misconception that after declaring volatile ++ will be atomic, to make the operation atomic you still need to ensure exclusive access using synchronized method or block in Java.

Daemon threads

Read more: http://javarevisited.blogspot.com/2012/03/what-is-daemon-thread-in-java-and.html#ixzz4dMbkz8cK

1. Any thread created by main thread, which runs main method in Java is by default non daemon because Thread inherits its daemon nature from the Thread which creates it i.e. parent Thread and since main thread is a non daemon thread, any other thread created from it will remain non-daemon until explicitly made daemon by calling `setDaemon(true)`.

2. `Thread.setDaemon(true)` makes a Thread daemon but it can only be called before starting Thread in Java. It will throw `IllegalThreadStateException`if corresponding Thread is already started and running.

3. Daemon Threads are suitable for doing background jobs like housekeeping, Though I have yet to use it for any practical purpose in application code. let us know if you have used daemon thread in your java application for any practical purpose.

Difference between Daemon and Non Daemon thread

1) JVM doesn't wait for any daemon thread to finish before existing.

2) Daemon Thread are treated differently than User Thread when JVM terminates, finally blocks are not called, Stacks are not unwounded and JVM just exits.