What is Thread?

Single flow of execution is called Thread.
Task performed by Thread is called Job.

It is part of any process which is considered as a sub-process or lightweight process. It is a class available in the **java.lang.package**. Its objects have their own call stack.

In Java every task is done as a process, even if we do not specify any thread, two threads simultaneously run in our project, Main thread and GC thread.

What is multithreading?

Flow of execution of more than one independent threads.

What are the advantages of threads?

Threads support concurrent operations. For example,

- Multiple requests by a client on a server can be handled as an individual client thread.
- All threads run under the same process so they share resources internally.

What is the difference between Thread and Process?

Thread	Process
They share common address space	They have their own address
They have direct access to the data segment of its process	They have their own copy of the data segment of the parent process.
Threads can directly communicate with other threads of its process	Processes must use interprocess communication to communicate with sibling processes
Threads can control other threads of the same process	Processes can control only its child processes.

Difference between Multitasking and Multithreading?

Multitasking -
Execution of multiple tasks simultaneously is called multitasking. It is divided into two categories:

Process based multitasking (Each task is separate individual process)
Thread based multitasking (Multithreading)
Multithreading -
Execution of multiple tasks simultaneously where each task is part of the same program and each task is independent task.

Does multithreading improve performance of an application?

Yes, if for any application we have some tasks which can be done by individual and independent threads it is better to create multiple threads to improve performance and scalability of that application.

What do you mean by scalability?

It means ability to improve performance by adding further resources.

Different states of thread life cycle?

New: (Object instantiated): When thread is instantiated. In this state thread is not considered to be alive.
RUNNABLE: When start method called with thread object. There are two internal states:

- **READY:** In this state thread is considered to be alive but not running.
- **RUNNABLE:** When thread scheduler allocates memory and puts in thread pool. It is just a dummy state to understand the thread flow.

BLOCKED: When thread is waiting for monitor lock. In this state thread is considered to be alive but not running.
WAITING: When thread is waiting for another thread. In this state thread is considered to be alive but not running.
TIMED_WAITING: When thread is waiting for another thread for up to specified waiting time. In this state thread is considered to be alive but not running.
TERMINATED: When thread is terminated.

How many threads does a Java program have at least?

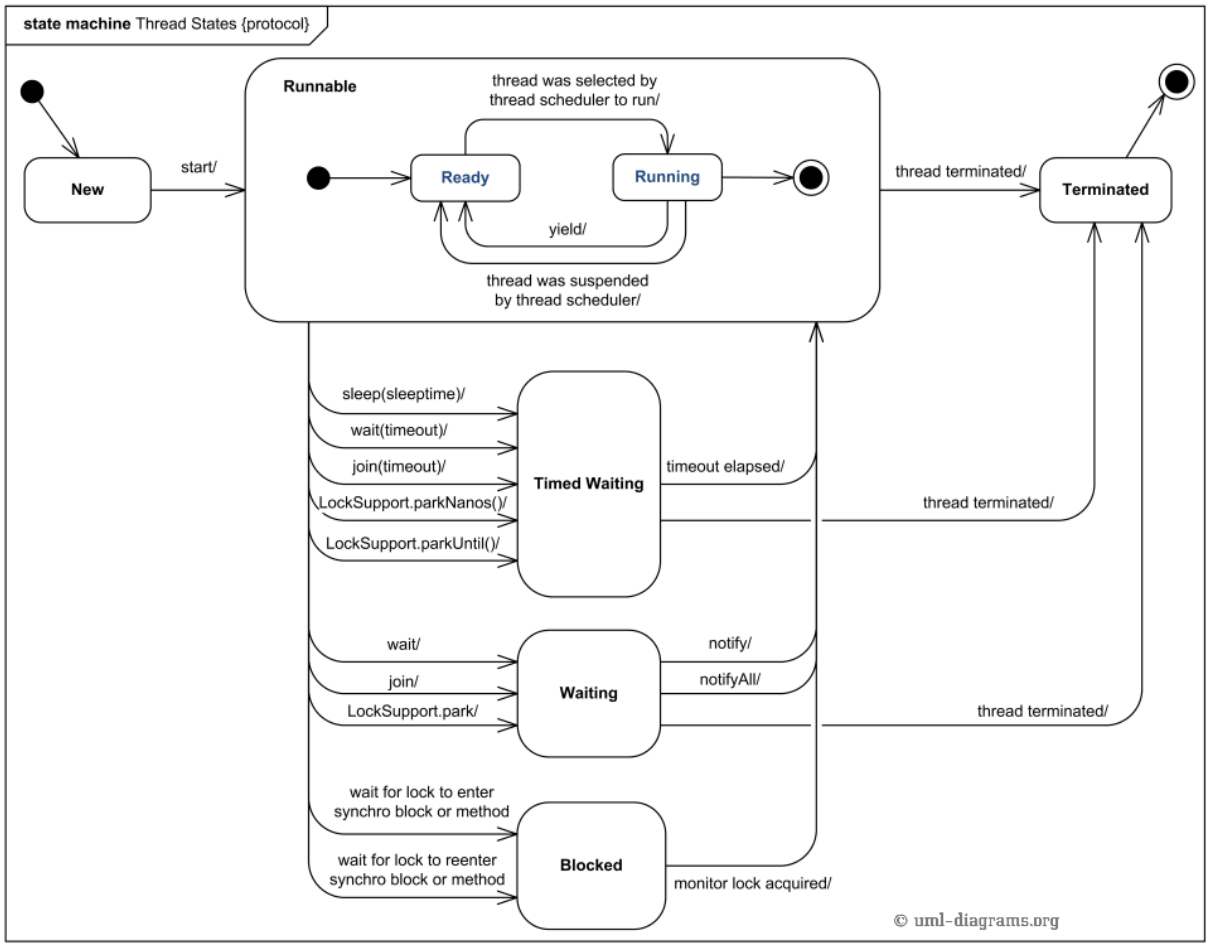
Two threads, each Java program is executed within the main thread, hence each Java application has at least one thread. Second thread is a GC thread which runs in the background for cleanup activity.

Which thread executes first when JVM starts up?

main thread (main method)

Flow diagram of thread?

http://www.uml-diagrams.org/examples/state-machine-example-java-6-thread-states.png



How many ways are there to define a thread?

Theoretically there are only two ways but practically three ways to achieve thread.

1. By instantiating java.lang.Thread and call start method which internally creates new thread and calls run method of that thread. This can be done in two ways:
    a. By extending Thread
    b. By implementing Runnable

2. By implementing Callable interface.

1. By extending Thread

Create class which extends Thread class and override run method.
Create another class, create an object of above thread class and call start method.

```
class MyThread extends Thread {
        public void run() {
                for (int i = 0; i <= 3; i++) {
                        System.out.println("In thread class");
                }
        }
}
class Main {
        public static void main(String args[]) {
                MyThread mt = new MyThread();
                mt.start();
                for (int i = 0; i <= 3; i++) {
                        System.out.println("In Main");
                }
        }
}
```

O/P:

```
In Main
In Main
In Main
In thread class
In thread class
In Main
In thread class
In thread class
```

Output may vary for each call because any thread can start first. Which thread to execute first is handled by the thread scheduler.

2. By implementing Runnable

Create class and implement Runnable.

```
class MyThread implements Runnable {
  public void run() {
        for (int i = 0; i <= 5; i++) {
                System.out.println("In thread class");
        }
  }
}

class Main {
  public static void main(String args[]) {
        MyThread run = new MyThread();
        Thread t = new Thread(run);
        t.start();
        for (int i = 0; i <= 5; i++) {
                System.out.println("In Main");
        }
```

```
  }
}
```

3. By implementing Callable

Create class which implements Callable interface. Create object of ExecutorService class to run Callable worker. This feature is introduced in Java 1.5

```
public class MyCallable implements Callable<String>{
	@Override
	public String call() throws Exception {
		for (int i = 0; i <= 5; i++) {
			System.out.println("In callable thread class");
		}
		return "Done";
	}
}

class Main {
	public static void main(String[] args) throws InterruptedException,
ExecutionException {
		System.out.println("In Main");
		ExecutorService executor = Executors.newFixedThreadPool(10);
		Callable worker = new MyCallable();
		executor.submit(worker);
		executor.shutdown();
		System.out.println("Out Of Main");
	}
}
```

To create Thread in Java, which one is better, extending Thread class or implementing Runnable interface?

Implementing Runnable is a better choice as it gives additional flexibility to extend another class. If there is any requirement where we do not want to extend any other class from that class then we can extend from Thread class.

Relation between Thread Class and Runnable Interface?

Thread class is having both is-a and has-a relationship with Runnable interface. Internally they have is-a relation as Thread class implements Runnable interface, has-a relationship created by programmer to create thread when implementing runnable interface.

What is the difference between start and run methods in Java Thread?

run method is responsible to run any thread.
start method creates a new thread and call run method to execute that thread.

Can we call the start method twice?

No, we can not call start method twice. Once thread is started we can not start it again, if we try to do so we will get an "IllegalThreadStateException" exception.

Instead of calling start method can we call run method directly?

Yes, syntactically we are allowed to call run method directly but it is considered as normal method call and runs within the same thread. It does not create new threads. To create a new thread we must call start method on thread object.

Ex:
Output with run method:-

```
public class Main {
    public static void main(String args[]) {
        System.out.println("In Main");
        MyThread mt = new MyThread();
        mt.run();
        try {
            Thread.sleep(2000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("Out Of Main");
    }
}

class MyThread extends Thread {
    public void run() {
        int count=0;
        while(true) {
            System.out.println("Count:" + count++);
            try {
                sleep(1000);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

O/P:

In Main
Count:0
Count:1
Count:2
Count:3
...

Output with start method:-

```
public class Main {
   public static void main(String args[]) {
         System.out.println("In Main");
         MyThread mt = new MyThread();
         mt.start();
         try {
                  Thread.sleep(2000);
         } catch (Exception ex) {
                  ex.printStackTrace();
         }
         System.out.println("Out Of Main");
   }
}

class MyThread extends Thread {
   public void run() {
         int count=0;
         while(true) {
                  System.out.println("Count:" + count++);
                  try {
                           sleep(1000);
                  } catch (Exception ex) {
                           ex.printStackTrace();
                  }
         }
   }
}
```

O/P:
In Main
Count:0
Count:1
Out Of Main
Count:2
Count:3
.....

What properties does each Java thread have?

Each Java thread has the following properties:

- An identifier of type long that is unique within the JVM.
- A name of type String
- A priority of type int
- A state of type java.lang.Thread.State
- A thread group the thread belongs to

What is a scheduler?

Scheduler is the one who decides which thread to call in which order. It follows some algorithms for different JVM to perform its operation.

How does the scheduler work?

A scheduler is the implementation of a scheduling algorithm that manages processes and threads. The main goal of a scheduler is to provide load balancing for available resources that guarantees that each process/thread gets an appropriate time frame to access the requested resource exclusively.

Scheduler decides which thread to call in which order. It follows a different algorithm for different JVM. Below are option which a thread scheduler can follow-

1. First come first serve:
   Thread calls start method first gets executed first.
2. Shortest Job:
   Threads who can finish their task quickly get a chance first.
3. Round robin (time slicing):
   scheduler executes thread only for predefined time. If thread is not able to complete its task in that predefined time then thread will enter into ready state.
4. Priority wise (preemptive scheduling):
   Scheduler executes highest priority threads first. Other threads get a chance to execute only when highest priority thread enters into waiting or terminated state, or any other highest priority thread comes into existence.

What is a shutdown hook?

A shutdown hook is a thread which gets executed when JVM shuts down. It is also called a shutdown thread. It is mostly used for cleanup activity and tracking purposes.

Ex:

```
public class Main {
  public static void main(String args[]) {
        System.out.println("In Main");
        Runtime.getRuntime().addShutdownHook(new ShutDownThread());
        MyThread mt = new MyThread();
        mt.start();
  }
```

```
}

class MyThread extends Thread {
  public void run() {
        int count=0;
        while(true) {
                System.out.println("Count:" + count++);
                try {
                        sleep(1000);
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
                if (count > 4) {
                        break;
                }
        }
  }
}

class ShutDownThread extends Thread {
  public void run() {
        System.out.println("In Shutdown Thread");
  }
}
```

O/P:
In Main
Count:0
Count:1
Count:2
Count:3
Count:4
In Shutdown Thread

What do we understand by the term concurrency?

Concurrency is the ability of a program to execute several computations simultaneously. This can be achieved by distributing the computations over the available CPU cores of a machine or even over different machines within the same network.

What is synchronization?

Synchronization is a way to prevent data manipulation of some resources simultaneously by multiple threads. It can be achieved with synchronized keyword.

What is the use of synchronized keyword?

It is used to prevent data concurrency within multiple threads where one thread accessing some data and another thread trying to update the same. It can be applied to methods(static + non static) or block of code. This keyword ensures that only one thread enters into this block at any single time by creating a lock which is also called as Intrinsic lock or Monitor lock.

How synchronization works?

When we specify any synchronized area with a "synchronized" keyword, before entering into this area every thread first checks whether lock is available. If lock is available, thread creates new lock to prevent other threads entering into the same block and perform its operation. If lock is not available, it waits for lock to be available.

Non static and block level synchronization creates lock at object level.
Static and static block level synchronization creates lock at class level.

If one thread accesses any static block, another thread can not access same or any other static block of that class.

Difference between synchronized method and block?

A thread enters a synchronized method only when it gets the lock of the method's object (for non static method) or class object (for static method).
A thread enters a synchronized block only when it gets a lock of an object or class referenced in synchronized block.

Synchronized block creates lock for smaller area of methods. We can have multiple synchronized blocks under the same method.

Can a constructor be synchronized?

No, a constructor cannot be synchronized.

Can the variables or classes be Synchronized?

No. Only methods or blocks can be synchronized.

Can lock be created at class level?

Yes, in the case of a static synchronized method, lock is created at class level.

How many locks does an object have?

Each object has only one lock.

Is it possible to check whether any Thread holds lock on any object?

Yes, with the help of the static method "holdsLock" of Thread class by specifying the object for which we want to check lock. It returns boolean true if the condition is satisfied.

Ex:
Thread.holdsLock(this)
Thread.holdsLock(emp) ...

What is the difference between wait() and sleep()?

- sleep() is a method of Thread class. wait() is a method of Object class.
- sleep() allows the thread to go to sleep state for x milliseconds. When a thread goes into sleep state it doesn't release the lock. wait() allows thread to release the lock and goes to suspended state. The thread is only active when a notify() or notifyAll() method is called on the same object or some condition stands true if thread is waiting for any condition to be true.

What does yield() method do?

yield() method tells the scheduler that it completed its major task and ready to free the process. It is not guaranteed that it will free the process as it is just a hint to the scheduler, it depends on the scheduler to free process and allow some other waiting threads to run.

It might also be possible that even after calling yield, the same thread runs continuously or after releasing process it becomes next thread to be executed.

What is the difference between yield() and sleep()?

yield() allows the current thread to release its lock and inform JVM to give chance to other threads to execute.
sleep() allows the thread to go to sleep state for x milliseconds. When a thread goes into a sleep state it doesn't release the lock.

By calling yield(), thread returns to ready state.
By calling sleep(), thread returns to the waiting state.

What is the difference between notify() and notifyAll()?

notify() wakes up the first thread that called wait() on the same object.
notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

In how many ways can a thread enter into a waiting state?

1. By invoking the sleep method of Thread class.
2. By invoking the wait method of Object class.
3. By I/O blocking
4. By unsuccessful attempt to acquire the lock.
5. By calling the deprecated suspend method.

How to determine which threads wakes up by calling notify()?

It is not possible to determine which thread wakes up as it completely depends on JVM implementation. If more than one thread is waiting, it will first pick up the thread with highest priority, but it is not guaranteed.

What is a visibility problem?

In a multi-threaded environment, if any thread updates the value of any variable then updated value remains with that thread only. Other threads cannot see updated value as they use their local cache to store variables. This is called visibility problem, as updates from one thread are not visible to other threads. To share data across threads, we can use volatile.

What is a volatile keyword?

In general each thread has its own copy of the variables, such that one thread is not concerned with the value of the same variables in the other threads, but sometimes this may not be the case. Consider a scenario in which the count variable is holding the number of times a method is called for a given class irrespective of which thread is calling, in this case we can declare the variable as volatile.

The copy of volatile variable is stored in the main memory, so every time a thread accesses the variables even for reading purpose the local copy is updated each time from the main memory. The volatile variable also has performance issues.

Few important facts about volatile:

- It can only be used with variables.
- Use of volatile keyword also prevents compilers or JVM from reordering code or moving them away from synchronization barriers.
- volatile keyword in Java guarantees that the value of volatile variable will always be read from main memory and not from Thread's local cache.

Why and when do we need to use volatile?

When:
Volatile is used when we want to share data across threads, where we want the value of some variable updated by some thread should immediately reflect other threads.

Why:
Each thread has their own stack, which is used to store local variables, method parameters and call stack. Variables stored in one Thread's stack are not visible to others. Objects created are stored inside heap and to improve performance thread cache values from heap into their stack. This creates a problem when some variable modified by one or more threads is not immediately reflected to other threads. To prevent this problem volatile is used which tells the thread to read value always from main memory.

Thread 1	Thread 2
Core 1	Core 2
Local Cache	Local Cache
Shared Cache	

With volatile, every change happened to any variable gets pushed to Shared Cache and from there it gets refreshed to all other local caches.

How data can be shared between two threads?

We can share data by:

- Thread safe collection objects.
- Non thread safe collection object using synchronization.
- Creating volatile variables.
- Object of Immutable class.
- Atomic variables

Difference between volatile and synchronize?

Synchronize	Volatile
Synchronization uses locking which guarantees visibility and atomicity.	Volatile guarantees visibility and atomicity without locking.
synchronize used with blocks or methods.	volatile used with variables.

What is mutual exclusion? and how to handle it in Java thread?

It is a phenomenon where more than one thread tries to access a critical section of program/application/memory at the same time. This can occur in case of multi-threading.

Synchronization can be used in this case with a synchronized keyword. Synchronization ensures that only one thread will access a synchronized area at a time. Methods or blocks can be synchronized.

Object class wait, notify and notifyAll can also be used for such cases, where by calling wait method thread can wait for some process to end and notify/notifyAll notify to start again.

Which is better, synchronize block or synchronize method?

Block synchronization in Java is preferred over method synchronization because by using block synchronization, you only need to lock the critical section of code instead of the whole method.

Which JVM parameter is used to control stack size of thread?

-Xss parameter is used to control stack size of Thread in Java.

How to force Thread to start?

It is not possible in java, we can request to start thread to thread scheduler and it handles the thread.

What is a daemon thread?

Daemon thread is considered to be service provider thread and run in the background, this is not used to run the application code generally. When all user threads(non-daemon threads) complete their execution the jvm exits the application whatever may be the state of the daemon threads. Jvm does not wait for the daemon threads to complete their execution if all user threads have completed their execution.

How to create daemon thread?

To create a Daemon thread set the daemon value of Thread using setDaemon(boolean value) method. By default all the threads created by the user are user threads. To check whether a thread is a Daemon thread or a user thread, isDaemon() method is used.

Example of the Daemon thread is the Garbage Collector run by jvm to reclaim the unused memory by the application. The Garbage collector code runs in a Daemon thread which terminates as soon as all the user threads are done with their execution.

What is gc()?

gc() is a daemon thread. gc() method is defined in System class which is used to request a JVM to perform garbage collection.

What is priority in threads? What is the default value?

We can specify priority for any thread between 1 (MIN_PRIORITY) to 10 (MAX_PRIORITY). This is an indicator for the thread scheduler to decide which thread to run first, but it is not guaranteed that the highest priority thread will always run first.

Default value is 5 (NORM_PRIORITY)

How do we set the priority of a thread?

The priority of a thread is set by using the method setPriority(int). To set the priority to the maximum value, we use the constant Thread.MAX_PRIORITY and to set it to the minimum value we use the constant Thread.MIN_PRIORITY because these values can differ between different JVM implementations.

If one thread creates another thread, what will be the priority of the second thread?

It will be the same as the thread which created it.

What is the priority for Garbage collector thread?

Since it is a Daemon thread, which are considered to be low priority threads. JVM internally treat it as a low priority thread, if we call getPriority() method on daemon thread, it returns 5.

How to prove that Daemon threads are low priority threads and work as supporting thread for any active thread?

```
public class MyDaemon {
	public static void main(String args[]) {
		System.out.println("In Main");
		MyThread mt = new MyThread();
		mt.setDaemon(true);
		mt.start();
		System.out.println("Priority Of Thread: " + mt.getPriority());
		System.out.println("Out Of Main");
	}
}
class MyThread extends Thread {
	public void run() {
		while (true) {
			System.out.println("In My Thread");
		}
	}
}
```

O/P:

```
In Main
Priority Of Thread: 5
Out Of Main
In My Thread
In My Thread
In My Thread
In My Thread
In My Thread
In My Thread
In My Thread
```

Line "In My Thread" might appear more or less times but it will terminate within seconds as soon as the main thread ends. Which proves that it runs only when at least one normal thread is running.

Is it possible to convert a normal user thread into a daemon thread after it has been started?

A user thread cannot be converted into a daemon thread once it has been started. Invoking the method thread.setDaemon(true) on an already running thread instance causes a IllegalThreadStateException.

Important methods of Thread class?

public static native Thread **currentThread**():
Returns a reference to the currently executing thread object.

public static native void **yield**():
A hint to the scheduler that all major tasks have been done and thread is ready to release resources.

public static native void **sleep**(long millis) throws InterruptedException
Causing currently executing thread to sleep.

public static void **sleep**(long millis, int nanos) throws InterruptedException:
It rounds off nanoseconds to milliseconds and calls sleep(long millis).
Ex:
sleep(0, 10) will call sleep(1)
sleep(1, 10) will call sleep(1)
sleep(1, 500010) will call sleep(2)

public synchronized void **start**():
Causes this thread to begin execution.

public void **run**():
Called internally by start method. If called with Runnable run object then method is called otherwise method does nothing.

private void **exit**():
Called internally by JVM to perform cleanup activity before exiting.

private void **interrupts**():
Interrupts calling thread.

public static boolean **interrupted**():
Check whether current thread interrupted.

public boolean **isInterrupted**():
Check whether the calling thread interrupted.

public final void **setPriority**(int newPriority):
Changes the priority of this thread

public final int **getPriority**():
Returns this thread's priority.

public final synchronized void **setName**(String name):
Changes the name of this thread.

public final String **getName**():
Returns this thread's name.

public final void **join**() throws InterruptedException:
Waits for this thread to die. It calls join(0) internally which means wait forever till this thread dies. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

public final synchronized void **join**(long millis) throws InterruptedException:
OR
public final synchronized void **join**(long millis, int nanos) throws InterruptedException:
Waits for this thread to die for specified duration.

public final void **setDaemon**(boolean on):
To mark thread as daemon (true) or user thread (false), default flag value is false.

public final boolean **isDaemon**():
To get whether thread is daemon(true) or user thread(false).

public String **toString**():
Return thread object in string representation.
Ex: Thread[Thread-0,5,main] (Thread-0 -> Thread name, 5 -> Priority, main -> Thread group name)

public State **getState**():
Return state of this thread.

Deprecated methods of Thread Class?

public final void **stop**():
Forces the thread to stop executing.

public void **destroy**():
Destroy the thread.

public final void **suspend**():
Suspends this thread.

public final void **resume**():
Resumes a suspended thread.

Why stop(), destroy(), suspend() and resume() methods deprecated?

stop(): It causes thread to release all the acquired monitors and throw a ThreadDeath error, which ultimately causes the thread to die. Since, the thread releases all the acquired monitors immediately, so it may leave few objects (whose monitors were acquired by the thread) in an inconsistent state.

destroy(): It was designed to destroy thread objects without cleanup. Any monitor held by thread would have remained locked.

suspend(): If thread holds the lock of a critical resource, by calling suspend, it does not release the lock. So no other thread can use resource until the same thread resumes.

resume(): This method used to resume suspended thread, since suspend method is deprecated this also deprecated.

What should be used instead of Thread.suspend and Thread.resume?

wait and notify

Which thread related methods are available in Object class?

- public final void **wait**() throws InterruptedException:
  Causes the current thread to wait until another thread invokes the notify or notifyAll, internally it calls wait(0).
- public final native void **wait**(long timeout) throws InterruptedException
  OR
  public final void **wait**(long timeout, int nanos) throws InterruptedException:
  Causes the current thread to wait until another thread invokes the notify or notifyAll or waiting time expires.
- public final native void **notify**():
  Wakes up a single thread that is waiting on this object's monitor.
- public final native void **notifyAll**():
  Wakes up all threads that are waiting on this object's monitor.

How can a Java application access the current thread?

The current thread can be accessed by calling the static method currentThread() of the JDK class java.lang.Thread:

```
public class MainThread {
        public static void main(String[] args) {
                long id = Thread.currentThread().getId();
                String name = Thread.currentThread().getName();
        }
}
```

Which is the best way to pass objects from one thread to another?

If we do not want to allow any modification then design an immutable object and pass.
If we want to allow modification then use thread safe collection objects.

What is the difference between HashMap and Hashtable particularly with regard to thread-safety?

All methods of Hashtable class are synchronized hence it is thread safe, HashMap is not thread safe.
Since all methods of Hashtable are synchronized, it is slower than HashMap and preferred for multithreaded applications only.

Which is faster? ConcurrentHashMap or Hashtable? and why?

ConcurrentHashMap is faster than Hashtable. Both are thread safe, it means both can be used for multi-threading. ConcurrentHashMap divides the map into different segments and locks that particular segment only whereas Hashtable locks complete the map.

When should we interrupt a thread?

We should interrupt a thread if we want to break out the sleep or wait state of a thread.

After starting a child thread, how do we wait in the parent thread for the termination of the child thread?

It can be achieved with a join method.

```
Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
                // Perform some activity in child thread
        }
});
thread.start();
thread.join();
```

How to make sure that join is happening with a running thread?

By using isAlive() method before join.

```
Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
                // Perform some activity in child thread
        }
});
thread.start();
if (thread.isAlive()) {
        thread.join();
}
```

Can thread class be serialized? If no then why, if yes then how?

No, because it does not implement a serializable interface.

Can user defined thread class be serialized? If no then why, if yes then how?

Syntactically Yes, but we should never serialize, because when we de-serialize an object it will be considered as new thread, as thread dependents on JVM machine, so state of thread from JVM1 does not preserve for JVM2 or even for the same JVM.

Thread Class:

```
public class MyThread extends Thread implements Serializable {
        public int count = 0;
        @Override
        public void run() {
                while (true) {
                        try {
                                sleep(1000);
                                System.out.println("Count: " + ++count);
                                if (count > 2) {
                                        break;
                                }
                        } catch (InterruptedException ex) {
                                ex.printStackTrace();
                        }
                }
        }
}
```

Thread Call Class:

```
public class ThreadCall {
        public static void main(String[] args) throws IOException, InterruptedException,
ClassNotFoundException {
                System.out.println("Save Object:------");
```

```
                MyThread mt = new MyThread();
                mt.start();
                Thread.sleep(3000);
                new ThreadCall().saveObject(mt);
                System.out.println("Read Object:------");
                MyThread mt1 = new ThreadCall().getObject();
                System.out.println(mt1.count);
                mt1.start();
        }

        public void saveObject(MyThread myT) throws IOException {
                ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("thread.txt"));
                out.writeObject(myT);
                System.out.println("Object Saved!!");
        }

        public MyThread getObject() throws IOException, ClassNotFoundException {
                ObjectInputStream in = new ObjectInputStream(new
FileInputStream("thread.txt"));
                MyThread mt = (MyThread) in.readObject();
                return mt;
        }
   }
```

O/P:

```
   Save Object:------
   Count: 1
   Count: 2
   Count: 3
   Object Saved!!
   Read Object:------
   3
   Count: 4
```

Here even though thread completed its execution before serialization but after deserialization it started again.

What is deadlock in threads?

This is a situation where one thread waits for the second thread to free some resource and the second thread waits for the first thread.

Ex:
Thread 1: locks resource A, waits for resource B
Thread 2: locks resource B, waits for resource A

How to prevent deadlock?

- By adding timeout for locks. So that thread waits for each other for less time and releases locks after specified timeout.
- If any thread does not succeed to get lock for any exclusive resource, it should release all other locks.
- Use a proper synchronize method.

How to detect deadlock?

By checking code, if nested synchronized blocks are used.
By looking at thread dump. (In linux "kill -3" command is used to check thread dump)
With the help of jconsole.

Write sample code for deadlock.

```
public class MyDeadLock extends Thread {
	@Override
	public void run() {
		try {
			if (this.getName().equals("First")) {
				this.m1();
			} else {
				this.m2();
			}
		} catch (InterruptedException ex) {
			ex.printStackTrace();
		}
	}
	public static void main(String[] args) throws InterruptedException {
		MyDeadLock mdl1 = new MyDeadLock();
		MyDeadLock mdl2 = new MyDeadLock();
		mdl1.setName("First");
		mdl2.setName("Second");
		mdl1.start();
		mdl2.start();
	}
	public void m1 () throws InterruptedException {
		synchronized (A.class) {
			Thread.sleep(1000);
			System.out.println("In M1 synchronized A");
			synchronized (B.class) {
				System.out.println("In M1 synchronized B");
			}
		}
	}
	public void m2 () throws InterruptedException {
```

```
                synchronized (B.class) {
                        Thread.sleep(1000);
                        System.out.println("In M2 synchronized B");
                        synchronized (A.class) {
                                System.out.println("In M2 synchronized A");
                        }
                }
        }
}
class A { }
class B { }
```

O/P:
In M1 synchronized A
In M2 synchronized B
OR
In M2 synchronized B
In M1 synchronized A

Here by calling m1 method thread 'First' gets lock for A class and waits to get lock for B class, and by calling m2 method thread 'Second' gets lock for B class and waits to get lock for A class.

Here Thread.sleep() is added to make sure we get deadlock every time. Without sleep also deadlock will occur but not all the time.

How to prevent such deadlock?

Here deadlock can be avoided if we maintain the same sequence for synchronized blocks.

Rewrite m2 method with below changes:

```
public void m2 () throws InterruptedException {
        synchronized (A.class) {
                Thread.sleep(1000);
                System.out.println("In M2 synchronized A");
                synchronized (B.class) {
                        System.out.println("In M2 synchronized B");
                }
        }
}
```

Here the sequence of synchronized blocks are the same as the m1 method. synchronized A class first and then B.

What is race condition in Java?

It occurs in multithreaded applications. As the name suggests, it occurs due to race between multiple threads, if a thread which is supposed to execute first lost the race and the second thread is executed.

Which methods are used to prevent IllegalMonitorStateException and race condition?

wait (), notify () or notifyAll() methods in synchronized block.

Why do we need to call wait/notify/notifyAll within synchronized block?

wait/notify/notifyAll all methods belong to the object class. To call these methods within thread, thread must own the lock which can be possible with synchronized block.

Thread can not invoke wait, notify and notifyAll unless it owns the lock.

What will happen if we call wait/notify/notifyAll outside the synchronized block?

We will get java.lang.IllegalMonitorStateException exception

Why wait, notify and notifyAll are not inside thread class?

Java provides lock at object level not at thread level, every object has lock, which is acquired by thread. Now if thread needs to wait for a certain lock it makes sense to call wait() on that object rather than on that thread.

If the wait method had been declared in Thread class, it wouldn't have cleared for which lock thread is waiting.

For the same lock reason, notify and notifyAll also declared in Object class.

Why do we always call the wait method inside the loop?

We always want to wait for any object in certain conditions. Without condition, thread might wait forever or start running with notify method even if notify called to wake up some other thread.

Difference between synchronized and concurrent collection in Java?

Both provide thread safe collection but concurrent collection provides more scalability. Concurrent collection introduced in Java 1.5 whereas synchronized keyword exists from the beginning.

If multiple threads access the same synchronized collection concurrently, we may face lock contention problems.

What is lock contention?

When two or more threads are competing to get the lock it is called lock contention. In this case the scheduler decides to give lock to which thread and puts another thread in waiting.

Lock contention occurs when a thread is trying to enter the synchronized block/method currently executed by another thread. This second thread is now forced to wait until the first thread has completed executing the synchronized block and releases the monitor.

How to reduce lock contention?

1. Reduce scope of lock.

Ex:

```
synchronized(emp) {
        String val = Employee.getVal();
        emp.setVal(val);
        emp.save();
}
```

Can be re-written as:

```
String val = Employee.getVal();
synchronized(emp) {
        emp.setVal(val);
        emp.save();
}
```

2. By lock splitting. Split locks if more than one independent process participated in the same lock.

Ex:

```
public synchronized boolean updateAndDelete(String action) {
        if (action.equals("Update")) {
                //Perform update operation
        } else {
                //Perform delete operation
        }
```

```
            return true;
        }
```

Can be re-written as:

```
        public boolean updateAndDelete(String action) {
            if (action.equals("Update")) {
                update();
            } else {
                delete();
            }
            return true;
        }
        public synchronized boolean update() {
            //Perform update operation
            return true;
        }
        public synchronized boolean delete() {
            //Perform delete operation
            return true;
        }
```

3. By lock stripping (By using concurrent collection objects): This technique uses multiple locks to guard the same data structure.

Ex: ConcurrentHashMap works in the same technique, it is a Map implementation which uses difference buckets to store its value based on key. It creates locks at bucket level instead of object level. This gives flexibility if one thread locks the first bucket, another thread can access second bucket.

What kind of lock technique is used by the SDK class for ReadWriteLock?

It uses the fact that more than one thread can read the data simultaneously, there is no need to specific lock. ReadWriteLock implemented by a pair of locks (Read lock and Write lock), where read lock can be obtained by more than one thread and write lock obtained by a single thread at a time. This implementation guarantees that once write lock is released all read operations see updated values.

What happens when an Exception occurs in a thread?

If not caught, thread will die, otherwise it will complete its process.

Program without proper catch:

```
    public class Main {
        public static void main(String args[]) throws InterruptedException {
            System.out.println("In Main");
            MyThread mt = new MyThread();
```

```
                mt.start();
        }
    }

    class MyThread extends Thread {
        @Override
        public void run() {
                int count = 0;
                while (true) {
                        System.out.println("Count:" + count++);
                        try {
                                sleep(500);
                                if (1 == 1) {
                                        throw new NullPointerException();
                                }
                        } catch (InterruptedException ex) {
                                ex.printStackTrace();
                        }
                        if (count > 3) {
                                break;
                        }
                }
        }
    }
```

O/P:

```
In Main
Count:0
Exception in thread "Thread-0" java.lang.NullPointerException
    at threads.MyThread.run(Main.java:22)
```

If we also catch NullPointerException then output will be:

```
In Main
Count:0
java.lang.NullPointerException
    at threads.MyThread.run(Main.java:22)
Count:1
Count:2
java.lang.NullPointerException
    at threads.MyThread.run(Main.java:22)
java.lang.NullPointerException
    at threads.MyThread.run(Main.java:22)
Count:3
java.lang.NullPointerException
    at threads.MyThread.run(Main.java:22)
```

What do you mean by Thread Safety?

It is a concept applicable for multi-threaded applications. A code is considered to be thread safe if it guarantees safe execution of shared resource/data by multiple threads.

How to achieve Thread Safety?

If shared resource/data can be avoided:

By creating local variables-

Since each and every thread has its own private copy of the local variable. It is thread safe.

By creating immutable object-

Since no thread can change its value it is thread safe.

If shared resource/data can not be avoided:

By mutual exclusion-

Provide access to shared data with synchronization. Improper use of synchronization may lead to deadlock, livelock and resource starvation.

By using atomic variables-

Data can be shared using atomic operation and can not be interrupted by other threads. Atomic operations form the basis of locking mechanisms for multi-threaded environments.

Meaning of atomicity?

It means complete or nothing. It's the same as a transaction in sql, which prevents partial data modification by any thread.

Can i=5 and i++ both be considered as atomic operations?

i=5 is atomic as JVM assigns value directly.
i++ is not atomic as JVM follows below steps to do so:

1. load value of i into register
2. add one to register
3. store register into i

It can happen in a multithreaded environment that before thread1 reaches the third step, thread2 executes first and second steps and gets an incorrect value.

Read and write reference variable is also atomic.
Most primitives read and write are atomic except Long and Double.

How to achieve atomicity?

1. By using atomic variables.
   ```
   AtomicInteger ai = new AtomicInteger();
   ```

2. Variables can be made atomic if used with volatile.
   public volatile long l;

Why do we need atomic variables when synchronization works for all cases?

There are few problems with synchronization:

- Each has to acquire lock on the object first.
- If a thread holding a lock is delayed, then no other thread requiring that lock may take progress.

Advantage of using Atomic variables?

Atomic variables provide support to create wait-free, lock-free programming. These are used for multi threading, where multiple threads can access the same variable without lock.

How to create atomic variables?

Since JDK5.0, java introduced many classes under the package "java.util.concurrent.atomic" which can be used to create atomic variables.

Ex:
Instead of creating synchronized methods to increase count by multiple thread, AtomicInteger class can be used.

Code with synchronization:

```
private int count;
public synchronized int incrementCounter() {
   return ++count;
}
```

Code with AtomicInteger:

```
private AtomicInteger count = new AtomicInteger();
public int incrementCounter(){
   return count.incrementAndGet();
}
```

There are many other classes: AtomicBoolean, AtomicIntegerArray, AtomicLong ...

Difference between static and volatile?

Static: These variables are at class level which are associated with specific class. In a multi-threaded environment, JVM caches some variables for faster execution which also

includes static variables for that thread. So it may be possible that changes made by Thread1 to any static variable immediately not reflect to Thread2.

Volatile: These variables are at instance level which are associated to instances of that class. JVM does not cache variables declared as volatile so every time it fetches data from main memory and latest changes reflect to another thread.

It is recommended to use atomic variables (ex: AtomicInteger) instead of volatile for multi-threaded environments. As it provides more accurate results when multiple threads try to update the same variable at same time.

Can we use static with volatile?

Yes, volatile can be applied to static, not static variables (Even transient). volatile can not be applied with final.

What is the advantage of a concurrent package?

Java 1.5 introduced the new package "java.util.concurrent" to make multi-threading easy and efficient.

It supports below important feature:

Executor framework (java.util.concurrent.Executor)
Callable interface (java.util.concurrent.Callable)
Future interface (java.util.concurrent.Future)
Concurrent collection (java.util.concurrent..)
Lock (java.util.concurrent.locks..)
Atomic variables (java.util.concurrent.atomic..)
CountDownLatch (java.util.concurrent.CountDownLatch)
CyclicBarrier (java.util.concurrent.CyclicBarrier)
Semaphore (java.util.concurrent.Semaphore)

What is Executor Framework?

It is a framework which defines a set of execution policies to provide standardization of how to invoke, schedule, execute and control asynchronous tasks.

It is used to run Runnable objects without creating new threads every time and mostly re-using the already created threads.

How many classes are there to work with Executor Framework?

There are three types of implementations provided by the Java:

ExecutorService
ThreadPoolExecutor
Executors (A Class containing factory methods)

Advantage of Executor Framework? and why do we need it?

Problem with traditional thread approach:

- **Time consuming**
  Every time a new thread is created to process requests.
- **Poor resource performance**
  Every time a new thread is created it utilizes memory and resources.
- **Not robust**
  There is no limit on the number of threads which can be created so there is a chance that JVM throws Out of memory exception.

Executor framework approach:

- **Good in time consumption and resource management**
  It creates a thread pool which is a pool of threads which are known as worker thread. Once execution of thread is over it goes into the pool again to wait for new requests instead of dying which happens in traditional approach.
- **Robust**
  Entire life cycle of threads managed by Executor framework with thread pool before JVM manages thread pool. We always specify the number of threads required in the thread pool which place the upper limit for thread creation.

Code example:

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(some worker);
executor.shutdown();
```

Disadvantage of Executor Framework?

Thread pool with limited thread once it crosses the limit it tries to do a task based on it has with that limited threads which might slow down speed a bit.

What is CountDownLatch?

It is used when we want to wait for more than one thread to complete its task. It is same as join but provides extra feature than join.

Practical scenario with example for CountDownLatch?

Requirement:
Manager divides modules between development teams (A and B) and he wants to assign it to the QA team for testing only when both the teams complete their task.

```
public class Manager {
	public static void main(String[] args) throws InterruptedException {
		CountDownLatch countDownLatch = new CountDownLatch(2);
		MyDevTeam teamDevA = new MyDevTeam(countDownLatch, "devA");
		MyDevTeam teamDevB = new MyDevTeam(countDownLatch, "devB");
		teamDevA.start();
		teamDevB.start();
		countDownLatch.await();
		MyQATeam qa = new MyQATeam();
		qa.start();
	}
}

class MyDevTeam extends Thread {
	CountDownLatch countDownLatch;
	public MyDevTeam (CountDownLatch countDownLatch, String name) {
		super(name);
		this.countDownLatch = countDownLatch;
	}
	@Override
	public void run() {
		System.out.println("Task assigned to development team " +
Thread.currentThread().getName());
		try {
			Thread.sleep(2000);
		} catch (InterruptedException ex) {
			ex.printStackTrace();
		}
		System.out.println("Task finished by development team
	Thread.currentThread().getName());
		this.countDownLatch.countDown();
	}
}

class MyQATeam extends Thread {
	@Override
	public void run() {
		System.out.println("Task assigned to QA team");
		try {
			Thread.sleep(2000);
		} catch (InterruptedException ex) {
			ex.printStackTrace();
		}
```

```
                System.out.println("Task finished by QA team");
        }
}
```

O/P:
Task assigned to development team devB
Task assigned to development team devA
Task finished by development team devB
Task finished by development team devA
Task assigned to QA team
Task finished by QA team

Can we apply join for the same scenario above?

Yes, here JOIN can also be used in this case by calling the join method in the main thread.

Ex:
teamDevA.start();
teamDevB.start();
teamDevA.join();
teamDevB.join();

If JOIN can be used why do we still need CountDownLatch?

CountDownLatch is needed to overcome below JOIN method limitation:

1. JOIN can not be used if a thread is created using ExecutorService.
2. JOIN can not be used as partial completion of thread. (Ex: Modify above example where Manager wants to handover code to QA team as soon as Development team completes their 80% task. It means that CountDownLatch allows us to modify implementation which can be used to wait for another thread for their partial execution.)

How does CountDownLatch work?

Suppose we want main thread to wait three worker threads

1. Create an object of CountDownLatch by passing 3 in the constructor and call await() method on that object after stating threads.
2. Pass created object in thread class by creating constructor in thread class.
3. Call countDown() method on the same object after thread completes its execution.

Here **await()** method waits for countdownlatch flag to become 0, and **countDown()**method decrements countdownlatch flag by 1.

What will happen if we directly call await() method before starting thread?

Ex: (Modification on above code)

```
CountDownLatch countDownLatch = new CountDownLatch(2);
countDownLatch.await();
MyDevTeam teamDevA = new MyDevTeam(countDownLatch, "devA");
MyDevTeam teamDevB = new MyDevTeam(countDownLatch, "devB");
teamDevA.start();
teamDevB.start();
MyQATeam qa = new MyQATeam();
qa.start();
```

await() method stops further execution and waits there only till it gets countdownlatch flag 0. In this case it will **wait forever** in the second line and will not start any thread.

Disadvantages of CountDownLatch?

One of the disadvantages of CountDownLatch is that it is not reusable once count reaches to zero it can not be reused.

What is CyclicBarrier?

It is used when we want more than one worker thread to wait for each other and perform some activity once all worker threads task is done.

Practical scenario with example for CyclicBarrier?

In the same previous scenario, suppose the manager is on leave, and both development teams decide to hand over the project only after both the teams complete their task.

```
public class Manager {
	public static AtomicInteger ai = new AtomicInteger(1);
	public static void main(String[] args) throws InterruptedException {
		System.out.println("Manager assigned tasks to development team");
		CyclicBarrier cyclicBarrier = new CyclicBarrier(2);
		MyDevTeam teamDevA = new MyDevTeam(cyclicBarrier, "devA");
		MyDevTeam teamDevB = new MyDevTeam(cyclicBarrier, "devB");
		teamDevA.start();
		teamDevB.start();
		System.out.println("Manager is on leave");
	}
}
```

```
class MyDevTeam extends Thread {
	CyclicBarrier cyclicBarrier;
	public MyDevTeam (CyclicBarrier cyclicBarrier, String name) {
		super(name);
		this.cyclicBarrier = cyclicBarrier;
	}
	@Override
	public void run() {
		System.out.println("Task assigned to development team " +
Thread.currentThread().getName());
		try {
			Thread.sleep(2000);
		} catch (InterruptedException ex) {
			ex.printStackTrace();
		}
		System.out.println("Task finished by development team " +
Thread.currentThread().getName());
		try {
			this.cyclicBarrier.await();
		} catch (Exception ex) {
			ex.printStackTrace();
		}
		if (Manager.ai.getAndDecrement() == 1) {
			MyQATeam qa = new MyQATeam();
			qa.start();
		}
	}
}

class MyQATeam extends Thread {
	@Override
	public void run() {
		System.out.println("Task assigned to QA team");
		try {
			Thread.sleep(2000);
		} catch (InterruptedException ex) {
			ex.printStackTrace();
		}
		System.out.println("Task finished by QA team");
	}
}
```

O/P:

Manager assigned tasks to development team
Manager is on leave
Task assigned to development team devA
Task assigned to development team devB
Task finished by development team devB
Task finished by development team devA
Task assigned to QA team

Task finished by QA team

How does CyclicBarrier work?

Suppose we want three worker threads to wait for each other to finish their work.

1. Create an object of CyclicBarrier by passing 3 in the constructor.
2. Pass created object in thread class by creating constructor in thread class.
3. Call await() method on the same object after thread completes its execution.

Here **await()** method decrements cyclicbarrier flag by 1 and waits for cyclicbarrier flag to become 0.

Difference between CountDownLatch and CyclicBarrier?

In CountDownLatch, the main thread waits for other threads to complete their execution. In CyclicBarrier, worker threads wait for each other to complete their execution.

You can not reuse the same CountDownLatch instance once count reaches to zero and latch is open, on the other hand CyclicBarrier can be reused by resetting Barrier, once barrier is broken.

What is Semaphore?

It is a set of permits. It is similar to lock with difference that it can have multiple locks hence it is considered as a pool of lock where more than one thread can acquire lock of object.

Important methods of Semaphore?

acquire() method used to acquire a lock.
release() method used to release acquired lock.

Practical scenario and example for Semaphore?

Suppose there is a conference room and two team members can access it at a time. If we consider the conference room as class then two members can have lock of this object. It is not possible to do it with a typical thread approach.

```
public class Team {
	public static void main(String[] args) throws InterruptedException {
		Semaphore systemPool = new Semaphore(2);
```

```
                MyTeamMember memA = new MyTeamMember(systemPool, "memA");
                MyTeamMember memB = new MyTeamMember(systemPool, "memB");
                MyTeamMember memC = new MyTeamMember(systemPool, "memC");
                MyTeamMember memD = new MyTeamMember(systemPool, "memD");
                memA.start();
                memB.start();
                memC.start();
                memD.start();
        }
}

class MyTeamMember extends Thread {
        Semaphore systemPool;
        public MyTeamMember (Semaphore systemPool, String name) {
                super(name);
                this.systemPool = systemPool;
        }
        @Override
        public void run() {
                System.out.println(Thread.currentThread().getName() + " waiting for
system");
                try {
                        systemPool.acquire();
                        Thread.sleep(2000);
                        System.out.println(Thread.currentThread().getName() + " got
                system");
                        systemPool.release();
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }
}
```

O/P:

```
memA waiting for system
memB waiting for system
memC waiting for system
memD waiting for system
memA got system
memB got system
memC got system
memD got system
```

Here the last two lines will appear after sometime.

What will happen if we pass 0 in the constructor while creating semaphore?

It will create a lock pool with 0, so thread can not get the lock and threads will wait to get lock forever.

What is the producer consumer problem?

Consider we have two independent thread Producer and Consumer. Task of the Producer thread is to produce resources and the Consumer thread is to consume.

Since both are running parallely and independently, there is a chance that Consumer thread try to consume resource even before Producer thread could produce it.

How to handle Producer Consumer problems?

It can be solved with inter thread communication methods. wait, notify and notifyAll methods can be used to achieve this.

The Consumer thread will call wait() method to wait for the Producer thread to produce resources.
Producer thread will call notify() or notifyAll() method once it produces a resource and call wait() method to wait for Consumer thread to consume that resource.

Sample code for Producer Consumer functionality with 3 products.

Create Producer thread.
Create Consumer thread.
Create ProducerConsumer which can communicate with both threads with wait() and notify() methods.

```
public class Main {
	public static void main(String args[]) throws InterruptedException {
		System.out.println("In Main");
		ProducerConsumer pc = new ProducerConsumer();
		Producer produce = new Producer(pc);
		produce.start();
		Consumer consume = new Consumer(pc);
		consume.start();
	}
}
class ProducerConsumer {
	private String content;
	private boolean isAvailable;
	public synchronized void Produce(String content) {
		while (this.isAvailable == true) {
			try {
				wait();
			} catch (InterruptedException ex) {
				ex.printStackTrace();
```

```
                }
            }
            this.content = content;
            this.isAvailable = true;
            notifyAll();
            System.out.println("In Produce with content: " + this.content);
        }
        public synchronized String Consume() {
            while (this.isAvailable == false) {
                try {
                    wait();
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
            Object obj = new Object();
            this.isAvailable = false;
            System.out.println("In Consume with content: " + this.content);
            notifyAll();
            return this.content;
        }
}
class Producer extends Thread {
        private ProducerConsumer pc;
        public Producer(ProducerConsumer pc) {
            this.pc = pc;
        }
        @Override
        public void run() {
            int count = 1;
            while(count <= 3 ) {
                pc.Produce("Product #" + count);
                count++;
            }
        }
}
class Consumer extends Thread {
        private ProducerConsumer pc;
        public Consumer(ProducerConsumer pc) {
            this.pc = pc;
        }
        @Override
        public void run() {
            int count = 1;
            while(count <= 3) {
                pc.Consume();
                count++;
            }
        }
}
```

O/P:
In Main
In Produce with content: Product #1
In Consume with content: Product #1
In Produce with content: Product #2
In Consume with content: Product #2
In Produce with content: Product #3
In Consume with content: Product #3

What is Callable interface?

This is another interface added in Java 1.5 which can also be used in place of Runnable interface in Java.

Difference between Callable and Runnable interface?

Callable interface contains call method, Runnable interface contains run method.
call method of Callable can return a value to indicate that thread execution is over, run method of Runnable interface is void.
Callable can throw exception, Runnable can not.

Advantage of Callable over Runnable interface?

There are two disadvantages with using the Runnable interface in JAVA:

- Cannot return value from method run
- Cannot throw Checked Exception

To overcome above problems, java.util.concurrent package has introduced the Callable interface. Instead of the run() method, Callable interface defines a single call() method that takes no parameter but can throw an exception.

Callable interface use Generic to define the return type of Object. (Ex: Callable<String> to return String value)

What is the Future interface and FutureTask class?

FutureTask class implements the interface Future. A Future represents the result of an asynchronous computation.

Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed.

How to create thread using Callable interface? and how to get return value with Callable interface?

Create class which implements Callable interface.
Create object of ExecutorService class by specifying length for thread pool and associate callable object by calling submit method.
Call shutdown method.
Create a reference to Future and use get() method to get value from call method. Future promises to return a value in future.

```
public class MyCallable implements Callable<String>{
	@Override
	public String call() throws Exception {
		for (int i = 0; i <= 5; i++) {
			System.out.println("In callable thread class");
		}
		return "Done";
	}
}
class Main {
	public static void main(String[] args) throws InterruptedException,
ExecutionException {
		System.out.println("In Main");
		ExecutorService executor = Executors.newFixedThreadPool(1);
		Callable worker = new MyCallable();
		Future future = executor.submit(worker);
		executor.shutdown();
		System.out.println(future.get());
		System.out.println("Out Of Main");
	}
}
```

O/P:

```
In Main
In callable thread class
In callable thread class
In callable thread class
In callable thread class
In callable thread class
In callable thread class
Done
Out Of Main
```

How to cancel Callable thread with Future interface?

Thread can be cancelled by calling cancel() method of Future interface.

```
public class MyCallable implements Callable<String>{
	@Override
	public String call() throws Exception {
		boolean flag = true;
		while (flag) {
			Thread.sleep(1);
			System.out.println("In call method, count is " + ++Main.count);
		}
		return "Done";
	}
}
class Main {
	public static int count = 0;
	public static void main(String[] args) throws InterruptedException,
ExecutionException {
		System.out.println("In Main");
		ExecutorService executor = Executors.newFixedThreadPool(1);
		Callable worker = new MyCallable();
		Future future = executor.submit(worker);
		executor.shutdown();
		Thread.sleep(10);
		if (Main.count > 5) {
			System.out.println("Cancel thread");
			future.cancel(true);
		}
		System.out.println("Main count: " + Main.count);
		System.out.println("Out Of Main");
	}
}
```

O/P:

```
In Main
In call method, count is 1
In call method, count is 2
In call method, count is 3
In call method, count is 4
In call method, count is 5
In call method, count is 6
In call method, count is 7
Cancel thread
In call method, count is 8
Main count: 8
Out Of Main
```

Output may differ for each run.

How does get() method work with Future reference?

Whenever the get() method is called with Future reference, control stays there only till the time call method does not return anything. It works the same as join with Thread class.

There is one overloaded method that exists for get(long timeout, TimeUnit unit), where we can wait for the required amount of time. If the call method does not return till time mentioned in the get method then java.util.concurrent.TimeoutException exception is thrown.

```
public class MyCallable implements Callable<String>{
	@Override
	public String call() throws Exception {
		for (int i = 0; i <= 4; i++) {
			System.out.println("In callable thread class");
			Thread.sleep(1000);
		}
	return "Done";
	}
}
class Main {
	public static int count = 0;
	public static void main(String[] args) throws InterruptedException,
ExecutionException, TimeoutException {
		System.out.println("In Main");
		ExecutorService executor = Executors.newFixedThreadPool(1);
		Callable worker = new MyCallable();
		Future future = executor.submit(worker);
		executor.shutdown();
		System.out.println(future.get(3, TimeUnit.SECONDS));
		System.out.println("Out Of Main");
	}
}
```

O/P:

```
In Main
In callable thread class
In callable thread class
In callable thread class
In callable thread class
Exception in thread "main" java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask.get(FutureTask.java:205)
    at Main.main(MyCallable.java:30)
In callable thread class
Java Result: 1
```

What additional methods exist in the Future interface?

boolean isDone(): Returns true if task is done else return false.
boolean isCancelled(): Returns true if task is cancelled else return false.

Can we create callable thread without executor service?

Practically yes by calling call() method inside run() method of normal thread, but we lose all functionality of callable interface hence it is of no use.

Ex:

```
public class Try {
	public static void main(String[] args) {
		new MyThread().start();
		System.out.println("End of main thread");
	}
}
class MyThread extends Thread {
	@Override
	public void run() {
		try {
			new MyCallable().call();
		} catch (Exception ex) {
			ex.printStackTrace();
		}
	}
}
class MyCallable implements Callable {
	@Override
	public Object call() throws Exception {
		while (true) {
			System.out.println("In Thread");
			Thread.sleep(100);
		}
	}
}
```

O/P:
End of main thread
In Thread
In Thread
In Thread
...
...

Limitations or disadvantages of Callable interface?

When we call the get() method of Future it blocks further execution till we get some result.
It can not work standalone like Runnable, it has to be used with ExecutorService.

What is a thread pool?

It is a container for threads which has readymade thread to perform our task. If we use ThreadPool, we do not need to create a thread on our own, thread pool takes care of it.

How to create a thread pool?

Thread pool is created using the Executor framework which is also known as ThreadPool framework.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

Above code will create a thread pool of 10 threads.

How to run thread inside thread pool?

Create a thread pool with Executors class.
Create a Runnable or Callable instance and run this instance by calling the submit method of ExecutorService object.

```
public class Main {
	public static void main(String[] args) {
		ExecutorService executor = Executors.newFixedThreadPool(5);
		executor.submit(new MyCallable());
		executor.submit(new MyRunnable());
		executor.shutdown();
		System.out.println("Out of main");
	}
}
class MyRunnable implements Runnable {
	@Override
	public void run() {
		for (int i = 1; i <= 3; i++) {
			System.out.println("In Runnable " + i);
			try {
				sleep(500);
			} catch (InterruptedException ex) {
				ex.printStackTrace();
			}
		}
	}
}
class MyCallable implements Callable<String> {
	@Override
```

```
        public String call() throws Exception {
                for (int i = 1; i <= 3; i++) {
                        System.out.println("In Callable " + i);
                        try {
                                sleep(500);
                        } catch (InterruptedException ex) {
                                ex.printStackTrace();
                        }
                }
                return "Done";
        }
}
```

O/P:
In Callable 1
Out of main
In Runnable 1
In Callable 2
In Runnable 2
In Callable 3
In Runnable 3

Why should you use thread pools in Java?

Creating thread is expensive in terms of time and resource. If we create thread at the time of request processing it will slow down our response time, also there is only a limited number of threads a process can create. To avoid both of these issues, a pool of thread is created when application starts-up, and threads are reused for request processing. This pool of thread is known as "thread pool" and threads are known as worker thread.

From JDK 1.5 release, Java API provides Executor framework, which allows us to create different types of thread pools e.g. single thread pool, which process one task at a time, fixed thread pool (a pool of fixed number of thread) or cached thread pool (an expandable thread pool suitable for applications with many short lived tasks).

What will happen if we create a thread pool of 1 and run two threads?

One thread will execute first completely and then the second thread will get a chance to execute.

Modify Main class of above example:

```
ExecutorService executor = Executors.newFixedThreadPool(1);
executor.submit(new MyCallable());
executor.submit(new MyRunnable());
executor.shutdown();
```

```
System.out.println("Out of main");
```

O/P:
In Callable 1
Out of main
In Callable 2
In Callable 3
In Runnable 1
In Runnable 2
In Runnable 3

What will happen if we call get() on a future object to get Runnable thread output?

We will not get any return value.

Modify Main class of above example:

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future future1 = executor.submit(new MyCallable());
Future future2 = executor.submit(new MyRunnable());
executor.shutdown();
System.out.println(future1.get());
System.out.println(future2.get());
System.out.println("Out of main");
```

O/P:
In Callable 1
In Callable 2
In Callable 3
Done
In Runnable 1
In Runnable 2
In Runnable 3
null
Out of main

How to continuously check whether thread has finished its execution with Future?

It can be achieved by calling isDone method on future reference.

Modify Main class of above example:

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future future1 = executor.submit(new MyCallable());
executor.shutdown();
```

```
while (!future1.isDone()) {
        System.out.println("Thread is running ..");
        sleep(500);
}
System.out.println("Thread execution finished");
System.out.println("Out of main");
```

O/P:
In Callable 1
Thread is running ..
Thread is running ..
In Callable 2
Thread is running ..
In Callable 3
Thread is running ..
Thread execution finished
Out of main

What is FutureTask?

It is a class which implements RunnableFuture. RunnableFuture extends Runnable and Future interface. This is used to get the result of thread after completion or to check whether execution is complete.

Future object returned by the submit() method is also an instance of FutureTask.

A FutureTask can be used to wrap a Callable or Runnable object. Because FutureTask implements Runnable, a FutureTask can be submitted to an Executor for execution.

Difference between Future and futureTask?

Future is an interface which has four sub-interface Response, RunnableFuture, RunnableScheduledFuture and ScheduledFuture.
FutureTask is an implementation class of Future, as it implements RunnableFuture.

Difference between shutdown and shutdownNow method of ExecutorService?

shutdown() method used to stop a thread, it waits for the task to finish if it is submitted, it waits even if the task has not yet started.
shutdownNow() method used to stop a thread, it waits for the task to finish if it is submitted and started. If a task is submitted but yet not started it will not wait for that task to finish.

Best practice to follow when writing multi-threaded programming?

- Give meaningful names to thread for proper understanding.
- Avoid locking and reduce scope of synchronization.
- Prefer synchronization over wait and notify method.
- Create threads with thread pools with executorservice.
- Prefer concurrent collection over synchronized collection.

How to create our own lock?

```
public class MyLock {
	private boolean isLocked = false;
	public synchronized void lock() {
		while (isLocked) {
			try {
				wait();
			} catch (Exception e) {
				e.printStackTrace();
			}
		}
		isLocked = true;
	}
	public synchronized void unlock() {
		isLocked = false;
		notify();
	}
}
class MyCounter {
	private MyLock lock = new MyLock();
	private int count = 1;
	public void m1() throws InterruptedException {
		System.out.println(Thread.currentThread().getName() + " try to get
	lock");
		lock.lock();
		System.out.println(Thread.currentThread().getName() + " get the lock");
		Thread.sleep(1000);
		// Perform some operation
		System.out.println(Thread.currentThread().getName() + " unlocking");
		lock.unlock();
		System.out.println(Thread.currentThread().getName() + " unlocked");
	}
}
class MyLockThread extends Thread {
	private MyCounter counter;
	public MyLockThread(String name, MyCounter count) {
		super(name);
		this.counter = count;
	}
	@Override
```

```
        public void run() {
                try {
                        counter.m1();
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }
}
class MyLockMain {
        public static void main(String[] args) {
                MyCounter count = new MyCounter();
                MyLockThread lockThread1 = new MyLockThread("Thread 1", count);
                MyLockThread lockThread2 = new MyLockThread("Thread 2", count);
                lockThread1.start();
                lockThread2.start();
        }
}
```

O/P:

```
Thread 2 try to get lock
Thread 1 try to get lock
Thread 2 get the lock
Thread 2 unlocking
Thread 2 unlocked
Thread 1 get the lock
Thread 1 unlocking
Thread 1 unlocked
```

Problem with above code?

Above code will not work for re-entrant code on the same thread.

What is re-entrant code?

When thread is already inside one method by acquiring lock and call another method which also requires lock is called re-entrant.

In the above example if we create another method m2 which also requires lock and call m2 method inside m1 after acquiring lock then thread will wait to get lock for method m2 which is locked for itself.

```
public void m1() throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + " try to get lock");
        lock.lock();
        System.out.println(Thread.currentThread().getName() + " get the lock");
        Thread.sleep(1000);
        this.m2();
```

```
        // Perform some operation
        System.out.println(Thread.currentThread().getName() + " unlocking");
        lock.unlock();
        System.out.println(Thread.currentThread().getName() + " unlocked");
}
public void m2() throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + " try to get lock");
        lock.lock();
        System.out.println(Thread.currentThread().getName() + " get the lock");
        Thread.sleep(1000);
        // Perform some operation
        System.out.println(Thread.currentThread().getName() + " unlocking");
        lock.unlock();
        System.out.println(Thread.currentThread().getName() + " unlocked");
}
```

O/P:
Thread 1 try to get lock
Thread 2 try to get lock
Thread 1 get the lock
Thread 1 try to get lock

Here thread will wait forever to get the lock.

How to fix the above code and create a lock for re-entrant?

Re-entrant lock can be achieved by adding below two criteria:
1. Check who acquired the lock along with lock availability.
2. Add a counter flag to maintain the number of locks acquired by a single thread.

```
public class MyLock {
        private boolean isLocked = false;
        private Thread lockedThread = null;
        private int lockedCount = 0;
        public synchronized void lock() {
                while (isLocked && lockedThread != Thread.currentThread()) {
                        try {
                                wait();
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                }
                isLocked = true;
                lockedThread = Thread.currentThread();
                lockedCount++;
        }
        public synchronized void unlock() {
                if (lockedThread == Thread.currentThread()) {
                        lockedCount--;
```

```
                        if (lockedCount == 0) {
                                isLocked = false;
                                notify();
                        }
                }
        }
}
class MyCounter {
        private MyLock lock = new MyLock();
        private int count = 1;
        public void m1() throws InterruptedException {
                System.out.println(Thread.currentThread().getName() + " try to get lock
        for m1");
                lock.lock();
                System.out.println(Thread.currentThread().getName() + " get the lock for
        m1");
                Thread.sleep(1000);
                this.m2();
                System.out.println(Thread.currentThread().getName() + " unlocking for
        m1");
                lock.unlock();
                System.out.println(Thread.currentThread().getName() + " unlocked for
        m1");
        }
        public void m2() throws InterruptedException {
                System.out.println(Thread.currentThread().getName() + " try to get lock
        for m2");
                lock.lock();
                System.out.println(Thread.currentThread().getName() + " get the lock for
        m2");
                Thread.sleep(1000);
                System.out.println(Thread.currentThread().getName() + " unlocking for
        m2");
                lock.unlock();
                System.out.println(Thread.currentThread().getName() + " unlocked for
        m2");
        }
}
class MyLockThread extends Thread {
        private MyCounter counter;
        public MyLockThread(String name, MyCounter count) {
                super(name);
                this.counter = count;
        }
        @Override
        public void run() {
                try {
                        counter.m1();
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
```

```
	}
}
class MyLockMain {
	public static void main(String[] args) {
		MyCounter count = new MyCounter();
		MyLockThread lockThread1 = new MyLockThread("Thread 1", count);
		MyLockThread lockThread2 = new MyLockThread("Thread 2", count);
		lockThread1.start();
		lockThread2.start();
	}
}
```

O/P:

```
Thread 2 try to get lock for m1
Thread 1 try to get lock for m1
Thread 2 get the lock for m1
Thread 2 try to get lock for m2
Thread 2 get the lock for m2
Thread 2 unlocking for m2
Thread 2 unlocked for m2
Thread 2 unlocking for m1
Thread 1 get the lock for m1
Thread 2 unlocked for m1
Thread 1 try to get lock for m2
Thread 1 get the lock for m2
Thread 1 unlocking for m2
Thread 1 unlocked for m2
Thread 1 unlocking for m1
Thread 1 unlocked for m1
```

Here "thread 2" got lock first to enter into the m1 method which incremented the lockedCount flag by 1. When the same thread entered into method m2 it incremented the lockedCount flag by 1 again to make it to 2. Once m2 method execution is over unlock method was called which decremented lockedCount flag by 1, till this time we can not call notify for waiting "thread 1" as "thread 2" still acquires one lock. Once m1 method execution is over we call notify as "thread 2" releases all its lock.

How to do reentrant code with synchronization?

synchronized code is by default re-entrant. it means if a java synchronized method calls another synchronized method which requires the same lock then current thread which is holding lock can enter into that method without acquiring lock.

Can we use a String as a lock for synchronized blocks?

Yes, but it is not recommended, because the string is an immutable object and is stored in a String pool. So by any chance if any other part of code or any third party library use

the same String as their lock then they both will be locked on the same object despite being completely unrelated which could result in unexpected behaviour and bad performance.

Instead of String object it's advised to use new Object() for Synchronization in Java on synchronized block.

Why do we need our own lock?

Limitation of synchronized block:

- synchronized block releases lock only when method or block ends.
- There is no way to make lock in one method and release in some other method.

How to use the Lock class provided by Java with a concurrent package?

Java supports some classes which implements the Lock interface to provide better performance and additional features. Below are the examples of ReentrantLock class.

```
class MyCounter {
	private final Lock lock = new ReentrantLock();
	private int count = 1;
	public void m1() throws InterruptedException {
		String threadName = Thread.currentThread().getName();
		System.out.println(threadName + " try to get lock for m1");
		lock.lock();
		System.out.println(threadName + " get the lock for m1");
		Thread.sleep(1000);
		this.m2();
		System.out.println(threadName + " unlocking for m1");
		lock.unlock();
		System.out.println(threadName + " unlocked for m1");
	}
	public void m2() throws InterruptedException {
		String threadName = Thread.currentThread().getName();
		System.out.println(threadName + " try to get lock for m2");
		lock.lock();
		System.out.println(threadName + " get the lock for m2");
		Thread.sleep(1000);
		System.out.println(threadName + " unlocking for m2");
		lock.unlock();
		System.out.println(threadName + " unlocked for m2");
	}
}
class MyLockThread extends Thread {
	private MyCounter counter;
	public MyLockThread(String name, MyCounter count) {
```

```
                super(name);
                this.counter = count;
        }
        @Override
        public void run() {
                try {
                        counter.m1();
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }
}
public class MyReentrantLock {
        public static void main(String[] args) {
                MyCounter count = new MyCounter();
                MyLockThread lockThread1 = new MyLockThread("Thread 1", count);
                MyLockThread lockThread2 = new MyLockThread("Thread 2", count);
                lockThread1.start();
                lockThread2.start();
        }
}
```

When to use concurrent collection and when to use normal collection?

If in application there are a lot of reads and less writes then use Concurrent collection otherwise use normal collection, because with concurrent collection before writing any data, clone for that object is created which degrade the performance.

What is Starvation?

Starvation means that thread not getting chance because of:

- Some other thread taking up all the time and other thread which has some job to do waiting for this thread.
- Scheduler always schedule some other thread to execute.

It usually occurs when one thread accessing synchronized block and running continuously and blocking other threads.

What is LiveLock? How is it different from deadlock?

This is exactly opposite of deadlock.

In deadlock two threads wait for each other to get the resource.
In livelock two threads get so busy responding to each other that they actually do not perform any operation and go in loop.

Like deadlock, livelock threads also do not make further progress. However, the threads are not blocked, they are simply too busy responding to each other to resume work.

This is comparable to two persons attempting to pass each other in a narrow corridor when both move in opposite direction and both say “you pass first” and wait for other person to pass.
