```java
public class A {
    //private constructor
    private A() { }

    //only instance of this class
    private static final A instance = new A();

    //public static method to return the only instance
    public static A getInstance() {
        return instance;
    }

    public void print() {
        System.out.println("Singleton class print method");
    }
}
```

*Test.java:*

```java
public class Test {
    public static void main(String[] args) {
        A obj = A.getInstance();
        obj.print();
    }
}
```

Output:

# Singleton class print method

We can also access the instance by using the classname like A.instance because of static keyword but we will have to change its access modifier to 'public', so that it is visible outside the singleton class.

Let's suppose you are creating a large object by using a lot of resources, there may be a chance that object creation may throw an exception but the above way of creating a singleton class does not provide any option for exception handling as you can write try-catch only inside a block of code. There is a solution to tackle this particular problem where you can create the class instance inside a static block.

**Static Block Eager Initialization:**

```java
public class A {

    private A() { }

    private static A instance;

    static {
        try{
            instance = new A();
        } catch (Exception e) {
            throw new RuntimeException("Exception while creating singleton object");
        }
    }

    public static A getInstance() {
        return instance;
    }
}
```

Both the above approaches create the object even before it is used (initialized at the time of class loading because of static variable and static block), but there are other approaches where we can

create a Singleton class instance in a lazy initialization way i.e. only when someone asks for it. These approaches are discussed below.

**Lazy Initialization:** using this way of creating Singleton class, the object will not get created unless someone asks for it. Here we will create the class instance inside the global access method.

```
public class A {

    private A() { }

    private static A instance;

    public static A getInstance() {
        if(instance == null) {
            instance = new A();
        }
        return instance;
    }
}
```

This approach is suitable for only single-threaded application, suppose there are 2 threads and both have checked that the instance is null and now they are inside the "if(...)" condition, it will destroy our singleton pattern and both threads will have different instances. So, we must overcome this problem so that our singleton pattern doesn't break in case of multi-threaded environment.

**Thread Safe Singleton implementation:** here the easiest way to prevent multiple threads from creating more than one instance is to make the global access method 'synchronized', this way threads will acquire a lock first before entering the getInstance() method.

```
public class A {

    private A() { }

    private static A instance;

    public synchronized static A getInstance() {
        if(instance == null) {
            instance = new A();
        }
        return instance;
    }
}
```

Synchronizing the entire method comes with performance degradation, also acquiring the lock and releasing the lock on every call to getInstance() method seems un-necessary, because only first few calls to getInstance() method needs to be synchronized, what I mean to say by this statement is: let's suppose there are 10 threads that are trying to call getInstance() method, now you need to apply synchronization to only these 10 threads at this time and the thread which first acquires the lock will create the object. After that, every thread will get the same object because of null check in

if condition, so we can optimize the above code by using **double-checked locking principle**, where we will use a synchronized block inside the if condition, like shown below:

```
public class A {

    private A() { }

    private static A instance;

    public static A getInstance() {
        if(instance == null) {
            synchronized (A.class) {
                if(instance == null) {
                    instance = new A();
                }
            }
        }
        return instance;
    }
}
```

**The reason of second if condition inside the synchronized block**: suppose there are 2 threads and both called the getInstance() method at the same time, now they will both be inside the first if condition as instance is null at this time, and the first thread which acquires the lock will create the object and as soon as it exits the synchronized block, other thread which was waiting, will acquire the lock and it will also create another object thus breaking the singleton pattern. This is why it is called "double-checked locking".

Now, some people have faced issues with the above approach in java 1.4 and earlier versions, which was solved in later versions **by using 'volatile' keyword** with the above approach like below:

```
public class A {

    private A() { }

    private static volatile A instance;

    public static A getInstance() {
        A localRef = instance;
        if(localRef == null) {
            synchronized (A.class) {
                localRef = instance;
                if(localRef == null) {
                    instance = localRef = new A();
                }
            }
        }
        return localRef;
    }
}
```

localRef variable is there for the cases where instance is already initialized (discussed above), the volatile field is only accessed once because we have written *return localRef* not *return instance*.

There is another approach where an inner static class is used to create the Singleton class instance and it is returned from the global access method. This approach is called ***Bill Pugh Singleton Implementation***.

*Bill Pugh Singleton Implementation Program:*

```
public class A {

    private A() { }

    private static class InnerA {
        private static final A instance = new A();
    }

    public static A getInstance() {
        return InnerA.instance;
    }
}
```

The inner class does not get loaded at the time of class A loading, only when someone calls getInstance() method, it gets loaded and creates the Singleton instance.

### *Question 66: What are the different ways in which a Singleton Design pattern can break and how to prevent that from happening?*

Answer: There are 3 ways which can break Singleton property of a class, they are discussed below with examples:

**Reflection**: Reflection API in java is used to change the runtime behavior of a class. Hibernate, Spring's Dependency injection also uses Reflection. So, even though in the above singleton implementations, we have defined the constructor as private, but using Reflection, even private constructor can be accessed, so Reflection can be used to break the singleton property of a class.

*A.java:*

```
package com.singleton.demo;

public class A {

    private A() { }

    public static final A instance = new A();

}
```

Notice, I am using public access modifier with the only instance of singleton class so that it can be accessed outside this class using just the class name.

Now, let's see how Reflection can break Singleton:

*Test.java:*

```java
package com.singleton.demo;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class Test {
    public static void main(String[] args) {

        A instance1 = A.instance;
        A instance2 = null;
        Constructor[] constructors = A.class.getDeclaredConstructors();

        for(Constructor constructor : constructors) {
            constructor.setAccessible(true);
            try {
                instance2 = (A) constructor.newInstance();
                break;
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (IllegalArgumentException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Instance1 hashcode : " + instance1.hashCode());
        System.out.println("Instance2 hashcode : " + instance2.hashCode());
    }
}
```

Output:

```
Instance1 hashcode : 366712642
Instance2 hashcode : 1829164700
```

As we can see from the output, the 2 instances have different hashcode, thus destroying Singleton.

Now to prevent Singleton from Reflection, one simple solution is to throw an exception from the private constructor, so when Reflection tries to invoke private constructor, there will be an error.

```java
package com.singleton.demo;

public class A {

    private A() {
        if(A.instance != null) {
            throw new InstantiationError("This object creation is not allowed");
        }
    }
    public static final A instance = new A();

}
```

Now, if you run *Test.java*, you will get below output:

```
java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
        at java.lang.reflect.Constructor.newInstance(Unknown Source)
        at com.singleton.demo.Test.main(Test.java:15)
Caused by: java.lang.InstantiationError: This object creation is not allowed
        at com.singleton.demo.A.<init>(A.java:7)
        ... 5 more
Instance1 hashcode : 2018699554Exception in thread "main" java.lang.NullPointerException

        at com.singleton.demo.Test.main(Test.java:29)
```

The other solution to prevent Singleton from Reflection is **using Enums**, as its constructor cannot be accessed via Reflection, JVM internally handles the creation and invocation of enum constructor

*SingletonEnum.java:*

```java
package com.singleton.demo;

public enum SingletonEnum {

    INSTANCE;

    public void print() {
        System.out.println("Inside print method");
    }
}
```

*TestSingletonEnum.java:*

```java
package com.singleton.demo;

public class TestSingletonEnum {

    public static void main(String[] args) {
        SingletonEnum.INSTANCE.print();
    }

}
```

Output:

```
Inside print method
```

Another way which can break Singleton property of a class is:

**Serialization and Deserialization**: Our Singleton class may implement Serializable interface so that the object's state can be saved and at a later point in time, it can be accessed back using Deserialization, now the problem here is, when we deserialize the object, a new instance of the class will be created, thus breaking the singleton pattern.

See, the example below:

*A.java:*

```java
package com.singleton.demo;

import java.io.Serializable;

public class A implements Serializable{

    private static final long serialVersionUID = -2020L;

    private A() { }
    public static final A instance = new A();

}
```

*Test.java:*

```
public class Test {
    public static void main(String[] args) {

        A instance1 = A.instance;
        A instance2 = null;
        String file = "C:\\temp\\object.ser";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(instance1);

            fos.close();
            oos.close();

            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            instance2 = (A) ois.readObject();

            fis.close();
            ois.close();
        } catch (Exception e) {
            System.out.println("Exception occurred");
        }

        System.out.println("Instance1 hashcode : " + instance1.hashCode());
        System.out.println("Instance2 hashcode : " + instance2.hashCode());
    }
}
```

Output:

```
Instance1 hashcode : 865113938
Instance2 hashcode : 2003749087
```

To prevent our Singleton class from Serialization, there is a method called ***readResolve()*** which is called when ObjectInputStream has read an object from the stream and is preparing to return it to the caller, so we can return the only instance of this class from this method, and this way the only instance of singleton will be assigned to instance2, see below:

*A.java:*

```
public class A implements Serializable {

    private static final long serialVersionUID = -2020L;

    private A() { }
    public static final A instance = new A();

    protected Object readResolve() {
        System.out.println("readResolve() method is called");
        return instance;
    }
}
```

*Test.java:*

```
public class Test {
    public static void main(String[] args) {

        A instance1 = A.instance;
        A instance2 = null;
        String file = "C:\\temp\\object1.ser";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(instance1);

            fos.close();
            oos.close();
```

```
            System.out.println("Starting De-serialization");
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            instance2 = (A) ois.readObject();
            System.out.println("De-serialization completes");
            fis.close();
            ois.close();
        } catch (Exception e) {
            System.out.println("Exception occurred");
        }

        System.out.println("Instance1 hashcode : " + instance1.hashCode());
        System.out.println("Instance2 hashcode : " + instance2.hashCode());
    }
}
```

Output:

```
Starting De-serialization
readResolve() method is called
De-serialization completes
Instance1 hashcode : 865113938
Instance2 hashcode : 865113938
```

You can also throw an exception from the readResolve() method, but returning the only instance approach is better as your program execution will not stop.

One last way which can break Singleton property of a class is:

**Cloning**: As we know, Cloning is used to create duplicate objects (copy of the original object). If we create a clone of the instance of our Singleton class then a new instance will be created thus breaking our Singleton pattern.

See the program below:

*TestSingleton.java:*

```
package com.singleton.demo;

class Parent implements Cloneable {
    int x = 20;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Singleton extends Parent {
    public static final Singleton instance = new Singleton();

    private Singleton() { }
}

public class TestSingleton {
    public static void main(String[] args) throws CloneNotSupportedException {

        Singleton instance1 = Singleton.instance;
        Singleton instance2 = (Singleton) instance1.clone();

        System.out.println("Instance1 hashcode : " + instance1.hashCode());
        System.out.println("Instance2 hashcode : " + instance2.hashCode());

    }
}
```

Output:

```
Instance1 hashcode : 366712642
Instance2 hashcode : 1829164700
```

As you can see, both instances have different hashcodes indicating our Singleton pattern is broken, so to prevent this we can override clone method in our Singleton class and either return the same instance or throw CloneNotSupportedException from it.

See the program changes below:

```
class Singleton extends Parent {
    public static final Singleton instance = new Singleton();

    private Singleton() { }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return new CloneNotSupportedException();
    }
}
```

Output:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.CloneNotSupportedException cannot be cast to com.singleton.demo.Singleton
        at com.singleton.demo.TestSingleton.main(TestSingleton.java:27)
```

clone() returning the same instance, see the program changes below:

```
class Singleton extends Parent {
    public static final Singleton instance = new Singleton();

    private Singleton() { }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return instance;
    }
}
```

Output:

```
Instance1 hashcode : 366712642
Instance2 hashcode : 366712642
```

***Question 67: What are the different design patterns you have used in your projects?***

Answer: You will be asked this question almost in all interviews nowadays. So, be prepared with some design patterns that are used in mostly all projects, like:

- Factory Design Pattern
- Abstract Factory Design Pattern
- Strategy Design Pattern
- Builder Design Pattern
- Singleton Design Pattern
- Observer Design Pattern

And any other if you have used in your projects.

### *Question 68: Explain Volatile keyword in java*

Answer: Volatile is a keyword in java, that can be applied only to variables. You cannot apply volatile keyword to classes and methods. Applying volatile to a shared variable that is accessed in a multi-threaded environment ensures that threads will read this variable from the main memory instead of their own local cache.

Consider below code:

```
public class Test {

    static int x = 10;

}
```

Now, let's suppose 2 threads are working on this class and both threads are running on different processors having their own local copy of variable x. if any thread modifies its value, the change will not be reflected back in the original variable x in the main memory leading to data inconsistency because the other thread is not aware of the modification.

So, to prevent data inconsistency, we can make variable x as volatile:

```
public class Test {

    static volatile int x = 10;

}
```

Now, all the threads will read and write the variable x from/to the main memory. Using volatile, also prevents compiler from doing any reordering or any optimization to the code.

### *Question 69: What is Garbage Collection in Java, how it works and what are the different types of Garbage Collectors?*

Answer: Garbage collection in java is the process of looking at heap memory, identifying which objects are in use and which are not and deleting the unused objects. An unused object or unreferenced object, is no longer referenced by any part of your program.

Garbage collector is a daemon thread that keeps running in the background, freeing up heap memory by destroying the unreachable objects.

There was an analysis done on several applications which showed that most objects are short lived, so this behavior was used to improve the performance of JVM. In this method, the heap space is divided into smaller parts or generations. These are, ***Young Generation***, ***Old*** or ***Tenured Generation*** and ***Permanent Generation***.

The **Young Generation** is where all new objects are allocated and aged. The young generation is further divided into 3 parts: Eden Space, Survivor space S0 and Survivor space S1. When the young generation fills up, this causes a **minor garbage collection**. Some surviving objects are aged and

eventually move to the old generation. All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**. Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So, for Responsive applications, major garbage collections should be minimized. Also note that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection. And Perm Gen was available till Java 7, it is removed from Java 8 onwards and JVM uses native memory for the representation of class metadata which is called MetaSpace.

There is a flag MaxMetaspaceSize, to limit the amount of memory used for class metadata. If we do not specify the value for this, the Metaspace re-sizes at runtime as per the demand of the running application.

***How Garbage collection works:***

When new objects are first created, they are stored in the eden space of Young Generation and at that time both Survivor spaces are empty. When the eden space is filled, then a minor garbage collection is triggered. All the unused or un-referenced objects are cleared from the eden space and the used objects are moved to first Survivor space S0.

At the next minor garbage collection, same process happens, un-referenced objects are cleared from the eden space but this time, the surviving objects are moved to Survivor space S1. In addition, the objects that were in S0 will also be matured and they also get moved to S1. Once all surviving objects are moved to S1, both eden and S0 are cleared.

At the next minor GC, the same process repeats. When surviving objects reached a certain threshold, they get promoted from Young generation to Old generation. These minor GC will continue to occur and objects will continue to be promoted to the Old generation.

Eventually, a major GC will be performed on the Old generation which cleans up and compacts the space.

***Types of Garbage collector in Java:***

***Serial GC:***

Serial GC is designed for smaller applications that have small heap sizes of up to a few hundred MBs. It only uses single virtual CPU for its garbage collection and the collection is done serially. It takes around couple of second for Full garbage collections.

It can be turned on by using **-XX:+UseSerialGC**

java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar C:\temp\test.jar

***Parallel/Throughput GC:***

Parallel garbage collector uses multiple threads to perform the garbage collection. By default, on a host with N CPUs, this collector uses N garbage collector threads for collection. The number of collector threads can be controlled with the command line option: **-XX:ParallelGCThreads=<N>**

It is called Throughput collector as it uses multiple CPUs to speed up the application throughput. A drawback of this collector is that it pauses the application threads while performing minor or full GC, so it is best suited for applications where long pauses are acceptable. It is the default collector in JDK 8.

It can be turned on by using below 2 options:

**-XX:+UseParallelGC**

With this command line option, you get a multi-thread young generation collector with a single-threaded old generation collector. The option also does single-threaded compaction of old generation.

java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseParallelGC -jar C:\temp\test.jar

**-XX:+UseParallelOldGC**

With this option, the GC is both a multithreaded young generation collector and multithreaded old generation collector. It is also a multithreaded compacting collector.

Compacting describes the act of moving objects in a way that there are no holes between objects. After a garbage collection sweep, there may be holes left between live objects. Compacting moves objects so that there are no remaining holes. This compacting of memory makes it faster to allocate new chunks of memory to the heap.

java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseParallelOldGC -jar C:\temp\test.jar

***Concurrent Mark Sweep (CMS) Collector:***

The CMS collector, also called as the concurrent low pause collector, collects the tenured generation. It attempts to minimize the pauses due to garbage collection, by doing most of the garbage collection work concurrently with the application threads.

It can be turned on by passing -XX:+UseConcMarkSweepGC in the command line option.

If you want to set number of threads with this collector, pass -XX:ParallelCMSThreads=<N>

java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseConcMarkSweepGC -XX:ParallelCMSThreads=2 -jar C:\temp\test.jar

***G1 Garbage Collector:***

The Garbage First or G1 collector is a parallel, concurrent and incrementally compacting low-pause garbage collector

G1 collector partitions the heap into a set of equal-sized heap regions. When G1 performs garbage collection then a concurrent global marking phase is performed to determine the liveliness of objects throughout the heap. After this mark phase is complete, G1 knows which regions are mostly empty. It collects unreachable objects from these regions first, which usually yields a large amount of free space, also called Sweeping. So G1 collects these regions (containing garbage) first, and hence the name Garbage-First.

It can be turned on by passing -XX:+UseG1GC in the command line options

java –Xmx25g –Xms5g -XX:+UseG1GC -jar C:\temp\test.jar

Java 8 has introduced one JVM parameter for reducing the unnecessary use of memory by creating too many instances of the same String. This optimizes the heap memory by removing duplicate

String values to a global single char[] array. We can use the -XX:+UseStringDeduplication JVM argument to enable this optimization.

G1 is the default garbage collector in JDK 9.

### *Question 70: Explain Generics in Java*

Answer: Java Generics provides a way to reuse the same code with different inputs.

Advantages:

-

Generics provide compile-time type safety that allows programmers to catch invalid types at compile time.

Before Generics:

```
List list = new ArrayList();
list.add("Mike");
list.add(10);
```

After Generics:

```
List<String> list = new ArrayList<>();
list.add("Mike");
list.add(10);
```

Errors (1 item)

The method add(int, String) in the type List<String> is not applicable for the arguments (int)

-

When using Generics, there is no need of type-casting.

Before Generics:

```
List list = new ArrayList();
list.add("Mike");

String name = list.get(0);
```

Errors (1 item)

Type mismatch: cannot convert from Object to String

After Generics:

```
List<String> list = new ArrayList<>();
list.add("Mike");

String name = list.get(0);
```

-

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized and are type safe and easier to read.

```java
package com.generic.demo;

class Test<T> {
    private T obj;

    public Test(T obj) {
        this.obj = obj;
    }

    public T getObject() {
        return this.obj;
    }
}
public class GenericDemo {
    public static void main(String[] args) {
        Test<String> t1 = new Test<>("Mark");
        System.out.println(t1.getObject());

        Test<Integer> t2 = new Test<>(10);
        System.out.println(t2.getObject());
    }
}
```

Output:

```
Mark
10
```

**Multi-threading:** you will be asked many questions on multi-threading, so, read as much as you can and whatever you can. Here, I am including some of the important questions that are mostly asked in every interview.

### *Question 71: What is Multi-threading?*

Answer: Multi-threading is a process of executing two or more threads concurrently, utilizing available CPU resources. A single thread is a lightweight sub-process and the smallest unit of processing. Threads are independent, if any exception occurs in one thread, it does not affect other threads.

When we execute a Java program without making any separate thread, then also our program runs on a thread called 'main thread'.

There are 2 types of threads in an application, ***user*** thread and ***daemon*** thread. When the application is first started, main thread is the first user thread created. We can create multiple user threads and daemon threads.

One thing to remember here is that, JVM does not have any control on a thread's execution. The thread execution is controlled by Thread scheduler which is part of Operating System. A thread can be assigned a priority using setPriority(int) method, where 1 is the minimum and 10 is the maximum priority, however thread priority is not guaranteed as it is platform dependent.

Multi-threading is used in a time-consuming task, one common example is File Upload.

### *Question 72: How to create a thread in Java?*

Answer: There are 2 ways to create a thread:

- By extending Thread class
- By implementing Runnable interface

**By extending Thread class:**

*ThreadTest.java:*

```
package com.multithreading.demo;

class Task extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()
                + " is running");
    }
}
public class ThreadTest {
    public static void main(String[] args) {
        for(int i=1; i<=5; i++) {
            Task task = new Task();
            task.setName("Thread " + i);
            task.start();
        }
    }
}
```

Output:

```
Thread 1 is running
Thread 3 is running
Thread 2 is running
Thread 4 is running
Thread 5 is running
```

Here a Task class extends Thread class and overrides the run() method which contains the business logic, then we make an object of this Task and call the start() method, which starts the thread execution. *start() method internally calls run() method.*

**By implementing Runnable interface:**

*ThreadTest.java:*

```
package com.multithreading.demo;

class Task implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()
                + " is running");
    }
}
```

```
public class ThreadTest {
    public static void main(String[] args) {
        for(int i=1; i<=5; i++) {
            Thread task = new Thread(new Task());
            task.setName("Thread " + i);
            task.start();
        }
    }
}
```

Output:

```
Thread 2 is running
Thread 1 is running
Thread 4 is running
Thread 3 is running
Thread 5 is running
```

If you see the outputs of this and previous program, you will see that they are different, because any thread can get a chance to execute its run() method, when the CPU resources are available.

### *Question 73: Which way of creating threads is better: Thread class or Runnable interface*

Answer: Implementing Runnable is always the preferred choice, see the reasons below:

- As you know, Java does not allow multiple inheritance through classes (because of Diamond problem discussed in Question 9), so if you are creating threads by extending Thread class then you will not be able to extend any other class.
- When we are working with multi-threading, we are not looking to overwrite any existing functionality of Thread class, we just want to execute the code with multiple threads, so in that way also, Runnable is a good choice.
- One more reason to choose Runnable is that, most people don't work with just Raw Threads, they use the Executor framework that is provided from Java 5, that separates the task from its execution and we can execute Runnables using ***execute(Runnable Task)*** method of ***Executor interface***.

### *Question 74: What will happen if I directly call the run() method and not the start() method to execute a thread*

Answer: if run() method is called directly, then a new thread will not be created instead the code will run on the current thread which is main thread. Calling run() method directly will make it behave as any other normal method call. Only a call to start() method creates separate thread.

### *Question 75: Once a thread has been started can it be started again*

Answer: No. A thread can be started only once in its lifetime. If you try to start a thread which has already been started, an *IllegalThreadStateException* is thrown, which is a runtime exception. A thread in runnable state or a dead thread cannot be restarted.

### *Question 76: Why wait, notify and notifyAll methods are defined in the Object class instead of Thread class*

Answer: This is another very famous multi-threading interview question. The methods wait, notify and notifyAll are present in the Object class, that means they are available to all class objects, as Object class is the parent of all classes.

*wait() method* – it tells the current thread to release the lock and go to sleep until some other thread enters the same monitor and calls notify()

notify() method – wakes up the single thread that is waiting on the same object's monitor

notifyAll() method – wakes up all the threads that called wait() on the same object

if these methods were in Thread class, then thread T1 must know that another thread T2 is waiting for this particular resource, so T2 can be notified by something like T2.notify()

But in java, the object itself is shared among all the threads, so one thread acquires the lock on this object's monitor, runs the code and while releasing the lock, it calls the notify or notifyAll method on the object itself, so that any other thread which was waiting on the same object's monitor will be notified that now the shared resource is available. That is why these methods are defined in the Object class.

Threads have no specific knowledge of each other. They can run asynchronously and are independent. They do not need to know about the status of other threads. They just need to call notify method on an object, so whichever thread is waiting on that resource will be notified.

Let's consider this with a real-life example:

Suppose there is a petrol pump and it has a single washroom, the key of which is kept at the service desk. This washroom is a shared resource for all. To use this shared resource, the user must acquire the key to the washroom lock. So, the user goes to service desk, acquires the key, opens the door, locks it from the inside and use the facility.

Meanwhile if another user arrives at the petrol pump and wants to use the washroom, he goes to the washroom and found that it is locked. He goes to the service desk and the key is not there because it is with the current user. When the current user finishes, he unlocks the door and returns the key to the service desk. He does not bother about the waiting users. The service desk gives the

key to waiting user. If there are more than one user waiting to use the facility, then they must form a queue.

Now, apply this analogy to Java, one user is one thread and the washroom is the shared resource which the threads wish to execute. The key will be ***synchronized keyword*** provided by Java, through which thread acquires a lock for the code it wants to execute and making other threads wait until the current thread finishes. Java will not be as fair as the service station, because any of the waiting threads may get a chance to acquire the lock, regardless of the order in which the threads came. The only guarantee is that all the waiting threads will get to use the shared resource sooner or later.

In this example, the lock can be acquired on the key object or the service desk and none of them is a thread. These are the objects that decide whether the washroom is locked or not.

### *Question 77: Why wait(), notify(), notifyAll() methods must be called from synchronized block*

Answer: these methods are used for inter-thread communication. So, a wait() method only makes sense when there is a notify() method also.

If these methods are not called from a synchronized block then

- IllegalMonitorStateException will be thrown
- Race condition can occur

Let's first look at the IllegalMonitorStateException:

*WaitDemo.java:*

```
package com.multithreading.demo;

public class WaitDemo {
    public static void main(String[] args) {
        WaitDemo wd = new WaitDemo();

        try {
            wd.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Exception in thread "main" java.lang.IllegalMonitorStateException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Unknown Source)
        at com.multithreading.demo.WaitDemo.main(WaitDemo.java:8)
```

Now, let's understand how a race condition can occur:

***Producer-Consumer problem***: The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time. The problem is to make sure that the producer

won't try to add data into the buffer, if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

*Pseudo code:*

```
class Test{
    List<String> buffer;

    public void produce(String data) {
        buffer.add(data);
        notify();
    }

    public String consume() {
        while(buffer.isEmpty())
            wait();
        return buffer.remove();
    }
}
```

**How the race condition problem can occur:**

Suppose, a thread has called the consume() method and it finds that the buffer is Empty

Now, just before the wait() method is called, another thread calls produce() and adds an item to the buffer and calls notify()

And The first thread calls the wait() method. In this case, notify() call by the second thread will be missed

if by any chance, produce() method is not called then the consumer thread will be stuck in waiting state indefinitely, even though there is data available in the buffer.

The solution to above problem is using synchronized method/block to make sure that notify() is never called between the condition isEmpty() and wait()

### *Question 78: difference between wait() and sleep() method*

Answer: The differences are:

- wait() method can only be called from a synchronized context while sleep() method can be called without synchronized context
- wait() method releases the lock on the object while waiting but sleep() method does not release the lock it holds while waiting, it means if the thread is currently in a synchronized block/method then no other thread can enter this block/method
- wait() method is used for inter-thread communication while sleep() method is used to introduce a pause on execution
- waiting thread can be waked by calling notify() or notifyAll(), while sleeping thread will wake up when the specified sleep time is over or the sleeping thread gets interrupted
- wait() method is non-static, it gets called on an object on which synchronization block is locked while sleep() is a static method, we call this

method like Thread.sleep(), that means it always affects the currently executing thread

- wait() is normally called when a condition is fulfilled like if the buffer size of queue is full then producer thread will wait, whereas sleep() method can be called without a condition

### *Question 79: join() method*

Answer: join() method causes the current thread to pause execution until the thread which has called join() method is dead.

join() method can be used to execute the threads sequentially or in some specific order.

Let's see an example below:

There are 3 threads and I want to execute them in the order 1, 3, 2:

*JoinMethodDemo.java:*

```
package com.multithreading.demo;

class JoinTask implements Runnable {
    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName());
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class JoinMethodDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new JoinTask());
        t1.setName("First Thread");
        Thread t2 = new Thread(new JoinTask());
        t2.setName("Second Thread");
        Thread t3 = new Thread(new JoinTask());
        t3.setName("Third Thread");

        t1.start();
        System.out.println("Current Thread : "
                    + Thread.currentThread().getName());
        t1.join();

        t3.start();
        System.out.println("Current Thread : "
                    + Thread.currentThread().getName());
        t3.join();

        t2.start();
        System.out.println("Current Thread : "
                    + Thread.currentThread().getName());
        t2.join();

        System.out.println("Exiting from Current Thread : "
                + Thread.currentThread().getName());
    }
}
```

Output: