**1.How to create a immutable object in Java? Count all benefits?**

**An immutable class is one whose state can not be changed once created.** Here, state of object essentially means the values stored in instance variable in class whether they are primitive types or reference types.

To make a class immutable, below steps needs to be followed:

1. **Don’t provide “setter” methods or methods that modify fields or objects referred to by fields**. Setter methods are meant to change the state of object and this is what we want to prevent here.
2. **Make all fields final and private**. Fields declared private will not be accessible outside the class and making them final will ensure the even accidentally you can not change them.
3. **Don’t allow subclasses to override methods**. The simplest way to do this is to declare the class as final. Final classes in java can not be overridden.
4. **Always remember that your instance variables will be either mutable or immutable**. Identify them and return new objects with copied content for all mutable objects (object references). Immutable variables (primitive types) can be returned safely without extra effort.

Also, you should memorize following benefits of immutable class. You might need them during interview. Immutable classes –

* are simple to construct, test, and use
* are automatically thread-safe and have no synchronization issues
* do not need a copy constructor
* do not need an implementation of clone
* allow hashCode to use lazy initialization, and to cache its return value
* do not need to be copied defensively when used as a field
* make good Map keys and Set elements (these objects must not change state while in the collection)
* have their class invariant established once upon construction, and it never needs to be checked again
* always have “**failure atomicity**” (a term used by Joshua Bloch) : if an immutable object throws an exception, it’s never left in an undesirable or indeterminate state.

**2.Is Java Pass by Reference or Pass by Value?**

The Java Spec says that ***everything in Java is pass-by-value***. There is no such thing as “*pass-by-reference*” in Java. These terms are associated with method calling and passing variables as method parameters. Well, primitive types are always pass by value without any confusion. But, the concept should be understood in context of method parameter of complex types.

In java, when we pass a reference of complex types as any method parameters, always the memory address is copied to new reference variable bit by bit. See in below picture:

In above example, address bits of first instance are copied to another reference variable, thus resulting both references to point a single memory location where actual object is stored. Remember, making another reference to null will not make first reference also null. But, changing state from either reference variable have impact seen in other reference also.

Read in detail here: [**Java Pass by Value or Reference?**](https://howtodoinjava.com/java/related-concepts/java-is-pass-by-value-lets-see-how/)

**3.What is the use of the finally block? Is finally block in Java guaranteed to be called? When finally block is NOT called?**

The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

**If the JVM exits while the try or catch code is being executed, then the finally block may not execute.** Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

**4.Why there are two Date classes; one in java.util package and another in java.sql?**

A java.util.Date represents date and time of day, a java.sql.Date only represents a date. The complement of java.sql.Date is java.sql.Time, which only represents a time of day.  
The java.sql.Date is a subclass (an extension) of java.util.Date. So, what changed in java.sql.Date:

– toString() generates a different string representation: **yyyy-mm-dd**  
– a static valueOf(String) methods to create a date from a string with above representation  
– the getters and setter for hours, minutes and seconds are deprecated

The java.sql.Date class is used with JDBC and it was intended to not have a time part, that is, hours, minutes, seconds, and milliseconds should be zero… but this is not enforced by the class.

**5.Explain marker interfaces?**

The marker interface pattern is a design pattern in computer science, used with languages that **provide run-time type information about objects**.**.** In java, it is used as interfaces with no method specified.

A good example of use of marker interface in java is [Serializable](https://howtodoinjava.com/java/serialization/a-mini-guide-for-implementing-serializable-interface-in-java/" \o "A mini guide for implementing serializable interface in java) interface. A class implements this interface to indicate that its non-transient data members can be written to a byte steam or file system.

A *major problem* with marker interfaces is that an interface defines a contract for implementing classes, and that contract is inherited by all subclasses. This means that **you cannot “un-implement” a marker**. In the example given, if you create a subclass that you do not want to serialize (perhaps because it depends on transient state), you must resort to explicitly throwing NotSerializableException.

**6.Why main() in java is declared as public static void?**

***Why public*?** main method is public so that it can be accessible everywhere and to every object which may desire to use it for launching the application. Here, i am not saying that JDK/JRE had similar reasons because java.exe or javaw.exe (for windows) use Java Native Interface (JNI) calls to invoke method, so they can have invoked it either way irrespective of any access modifier.

***Why static*?** Lets suppose we do not have main method as static. Now, to invoke any method you need an instance of it. Right? Java can have overloaded constructors, we all know. Now, which one should be used and from where the parameters for overloaded constructors will come.

***Why void*?** Then there is no use of returning any value to JVM, who actually invokes this method. The only thing application would like to communicate to invoking process is: normal or abnormal termination. This is already possible using System.exit(int). A non-zero value means abnormal termination otherwise everything was fine.

**7.What is the difference between creating String as new() and literal?**

When we create String with new() it’s created in heap and also added into string pool, while String created using literal are created in String pool only which exists in Perm area of heap.

**8.How does substring () inside String works?**

String in java are like any other programming language, a sequence of characters. This is more like a utility class to work on that char sequence. This char sequence is maintained in following variable:

/\*\* The value is used for character storage. \*/  
**private final char value[];**

To access this array in different scenarios, following variables are used:

/\*\* The offset is the first index of the storage that is used. \*/  
**private final int offset;**

/\*\* The count is the number of characters in the String. \*/  
**private final int count;**

Whenever we create a substring from any existing string instance, substring() method only set’s the new values of offset and count variables. The internal char array is unchanged. This is a possible source of memory leak if substring() method is used without care. [Read more here](https://howtodoinjava.com/java/string/interview-stuff-about-string-class-in-java/)

**9.Difference between interfaces and abstract classes?**

This is very common question if you are appearing interview for junior level programmer. Well, most noticeable differences are as below:

* Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
* Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behavior.
* Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private or abstract etc.
* Java interface should be implemented using keyword “**implements**“; A Java abstract class should be extended using keyword “**extends**“.
* A Java class can implement multiple interfaces but it can extend only one abstract class.
* Interface is  cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists. Since Java 8, you can define [**default methods in interfaces**](https://howtodoinjava.com/java8/default-methods-in-java-8/).
* Abstract class are slightly faster than interface because interface involves a search before calling any overridden method in Java. This is not a significant difference in most of cases but if you are writing a time critical application than you may not want to leave any stone unturned.

**10.When do you override hashCode() and equals()?**

hashCode() and equals() methods have been defined in Object class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

hashCode() method is used to get a unique integer for given object. This integer is used for determining the bucket location, when this object needs to be stored in some HashTable like data structure. By default, Object’s hashCode() method returns and integer representation of memory address where object is stored.  
equals() method, as name suggest, is used to simply verify the equality of two objects. Default implementation simply check the object references of two objects to verify their equality.

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode() method, which states that equal objects must have equal hash codes.

* equals() must define an equality relation (it must be **reflexive, symmetric and transitive**). In addition, it must be consistent (if the objects are not modified, then it must keep returning the same value). Furthermore, o.equals(null) must always return false.
* hashCode() must also be consistent (if the object is not modified in terms of equals(), it must keep returning the same value).

The relation between the two methods is:

Whenever a.equals(b) then a.hashCode() must be same as b.hashCode().