**CORE JAVA**

**Explain OOP Concepts.**

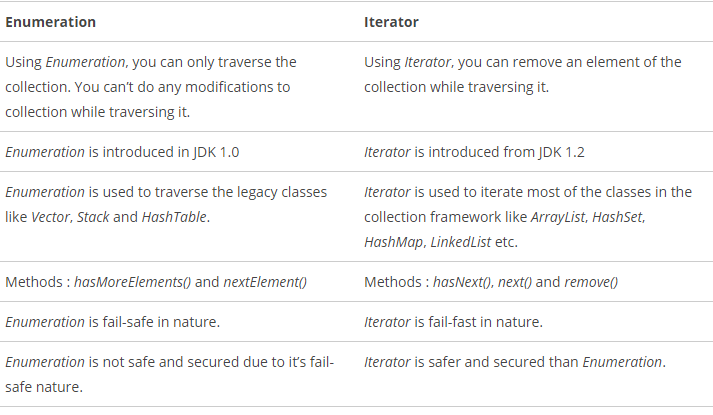
Object-Oriented Programming is a methodology of designing a program using classes, objects, inheritance, polymorphism, [abstraction](https://en.wikipedia.org/wiki/Abstraction_(software_engineering)), and [encapsulation](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)).

**Differences between abstract classes and interfaces?**[**GitHub**](https://arjun-sna.github.io/java/2017/02/02/abstractvsinterface/)

An abstract class, is a class that contains both concrete and abstract methods (methods without implementations). An abstract method must be implemented by the abstract class sub-classes. Abstract classes cannot be instantiated and need to be extended to be used.

An interface is like a blueprint/contract of a class (or it may be thought of as a class with methods, but without their implementation). It contains empty methods that represent, what all of its subclasses should have in common. The subclasses provide the implementation for each of these methods. Interfaces are implemented.

**What is the difference between iterator and enumeration in java?**

*Iterator* is a fail-fast in nature. i.e. it throws *ConcurrentModificationException* if a collection is modified while iterating other than it’s own *remove ()* method. Where as *Enumeration* is fail-safe in nature. It doesn’t throw any exceptions if a collection is modified while iterating.

**Do you agree we use composition over inheritance?**

[Composition vs Inheritance](https://www.journaldev.com/12086/composition-vs-inheritance)

**Difference between method overloading and overriding.**



Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class as overridden static methods are chosen by the reference class and not by the class of the object.

Static binding is being used for overloaded methods and dynamic binding is being used for overridden/overriding methods.

Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.

Private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.

**What are the access modifiers you know? What does each one do?**

There are four access modifiers in Java language (from strictest to the most lenient):

**private** *variables*, *methods*, *constructors* or *inner classes* are only visible to its' containing class and its' methods. This modifier is most commonly used, for example, to allow variable access only through getters and setters or to hide underlying implementation of classes that should not be used by user and therefore maintain encapsulation. Singleton constructor is also marked private to avoid unwanted instantiation from outside.

**protected** can be used on *variables*, *methods* and *constructors* therefore allowing access only to subclasses and classes that are inside the same package as protected members' class.

**Default** (no keyword is used) this modifier can be applied to *classes*, *variables*, *constructors* and *methods* and allows access from classes and methods inside the same package.

**public** modifier is widely-used on *classes*, *variables*, *constructors* and *methods* to grant access from any class and method anywhere. It should not be used everywhere as it implies that data marked with public is not sensitive and can not be used to harm the program.

**Can an Interface implement another Interface?**

Yes, an interface can implement another interface (and more than one), but it needs to use extends, rather than implements keyword. And while you can not remove methods from parent interface, you can add new ones freely to your subinterface.

**Polymorphism**

Polymorphism in Java has two types: Compile time polymorphism (static binding) and Runtime polymorphism (dynamic binding). Method overloading is an example of static polymorphism, while method overriding is an example of dynamic polymorphism.

**Inheritance**

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

**Multiple inheritance in Classes and Interfaces in java**

We all know that Java won't support[multiple inheritance](http://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html) to avoid diamond problem. So there is no way of extending multiple classes in java. That ends the discussion about multiple inheritance in classes. But still Java allows multiple inheritance in Interfaces. That raises the question about the same diamond problem in Interface as well right?  
consider the below case where one interface extending multiple Interfaces.

interface TestInter extends InterfaceA, InterfaceB {

// methods of all extended interface.

}

The above code have no errors or problems and works fine. But what we are trying to question is what happens if any of the interfaces having same method?? For example, first interface has a method *test ()*

interface InterfaceA {

testMethod();

}

And also the second interface have a method with same name

interface InterfaceB {

testMethod();

}

And now the interface *TestInter*extending these interfaces which having same method names. Java won't complain about that and it runs fine.. Again the question raises like, what happens if a class implement that TestInter?. Normally that class needs to implement all the methods inside the interface implemented and the methods of that extended interface(s). Having two interfaces same method, which method needs to implement and which interfaces method calls when calling a method on *TestInter*instance ??? The answer is that JVM is smart enough to know about this and it manages it. Let see what happens for that method with same name.

interface InterfaceA {

void testMethod();

}

interface InterfaceB {

void testMethod();

}

interface TestInter extends InterfaceA, InterfaceB {

// methods of TestInter.

}

class TestClass implements TestInter {@

Override

public void testMethod() {

//you see here

//method from A and B

//only one method overridden.

}

}

The above compiles successfully and we won't face any issues with it further as well. Yes, a single implementation works for the both methods. Single implementation belongs to the both interfaces. Hence there is no **diamond problem**. A test case would be as below to test that. 

InterfaceA intera = new TestClass();

intera.testMethod(); // the same implementation

InterfaceB interb = new TestClass();

interb.testMethod(); //the same implementation

**Arrays vs ArrayLists**

Array is a fixed length data structure whereas ArrayList is a variable length Collection class. We cannot change length of array once created in Java but ArrayList can be changed.

We cannot store primitives in ArrayList, it can only store objects. But array can contain both primitives and objects in Java. Since Java 5, primitives are automatically converted in objects which is known as auto-boxing.

**Explain Generics in Java?**

Generics were included in Java language to provide stronger type checks, by allowing the programmer to define, which classes can be used with other classes

In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types. ([Official Java Documentation](https://docs.oracle.com/javase/tutorial/java/generics/why.html))

This means that, for example, you can define:

List<Integer> list = new ArrayList<>();

And let the compiler take care of noticing, if you put some object, of type other than Integer into this list and warn you.

**How is String class implemented?**

The String class is immutable, so that once it is created a String object cannot be changed. The String class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

**Why was it made immutable? / What does it means to say that a String is immutable?**

String was made immutable to prevent malicious manipulation of data, when, for example, user login or other sensitive data is being send to a server.

It means that once created, String object's char[] (its' containing value) is declared final and, therefore, it can not be changed during runtime.

**What is the difference between an Integer and int?**

int is a primitive data type (with boolean, byte, char, short, long, float and double), while Integer (with Boolean, Byte, Character, Short,Long, Float and Double) is a [wrapper](https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html) class that encapsulates primitive data type, while providing useful methods to perform different tasks with it.

**What is Autoboxing and Unboxing?**

Autoboxing and Unboxing is the process of automatic wrapping (putting in a box) and unwrapping (getting the value out) of primitive data types, that have "wrapper" classes. So int and Integer can (almost always) be used interchangeably in Java language, meaning a method void giveMeInt(int i) { ... } can take int as well as Integer as a parameter.

**Typecast in Java**

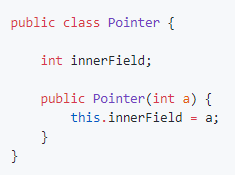
In Java, you can use casts to polymorph one class into another, compatible one. For example:

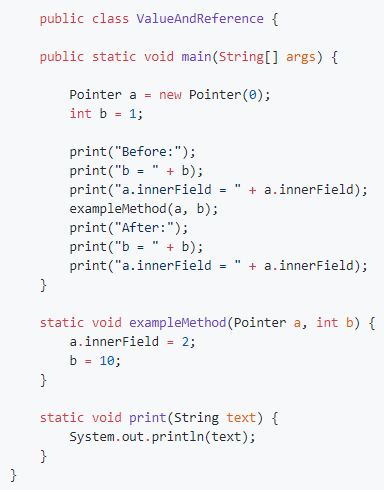
* long i = 10l;
* int j = (int) i;
* long k = j;

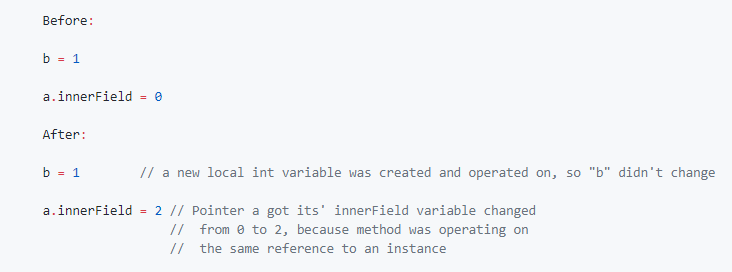
Here we see, that, while **narrowing** (long i -> int j) requires an explicit cast to make sure the programmer realizes, that there may be some data or precision loss, **widening** (int j -> long k) does not require an explicit cast, because there can be no data loss (long can take larger numbers than int allows).

**Do objects get passed by reference or value in Java? Elaborate on that.**

In Java all primitives and objects are passed by value, meaning that their copy will be manipulated in the receiving method. But there is a caveat - when you pass an object reference into a method, a *copy of this reference* is made, so it still points to the same object. This means, that any changes that you make to the insides of this object are retained, when the method exits.





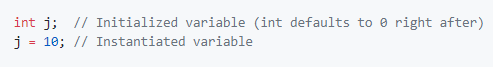


**What is the difference between instantiation and initialization of an object?**

Initialization is the process of the memory allocation, when a new variable is created. Variables should be explicitly given a value, otherwise they may contain a random value that remained from the previous variable that was using the same memory space. To avoid this problem, Java language assigns default (right after initialization) values to some data types:

* + boolean defaults to false;
  + byte defaults to 0;
  + short defaults to 0;
  + int defaults to 0;
  + long defaults to 0L;
  + char defaults to \u0000;
  + float defaults to 0.0f;
  + double defaults to 0.0d;
  + object defaults to null.

Instantiation is the process of explicitly assigning definitive value to a declared variable:



**What the difference between local, instance and class variables?**

**Local variables** exist only in methods that created them, they are stored separately in their respected Thread Stack (for more information, see question about Java Memory Model) and cannot have their reference passed outside of the method scope.

**Instance variables** are the ones, that are declared in classes and their value can be different from one instance of the class to another, but they always require that class' instance to exist.

**Class variables** are those, that are marked with static keyword in their class' body. They can only have one value across all instances of that class (changing it in one place will change it in their class and, therefore, in all instances) and can even be retrieved without that class' instance (if their access modifier allows it).

**What is garbage collector? How does it work?**

All objects are allocated on the heap area managed by the JVM. As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory.

**What is Java Memory Model? What contracts does it guarantee? How are its' Heap and Stack organized?**

<http://tutorials.jenkov.com/java-concurrency/java-memory-model.html>

**What is memory leak and how does Java handle it?**

The standard definition of a memory leak is a scenario that occurs when **objects are no longer being used by the application, but the Garbage Collector is unable to remove them from working memory** – because they’re still being referenced. As a result, the application consumes more and more resources – which eventually leads to a fatal *OutOfMemoryError*.

**What are strong, soft, weak and phantom references in Java?**

Java provides two different types/classes of *Reference Objects*: **strong** and **weak**. Weak Reference Objects can be further divided into *soft* and *phantom*. Let's go point by point.

**Strong Reference Object**

StringBuilder builder = new StringBuilder();

This is the default type/class of Reference Object, if not differently specified: builder is a strong Reference Object. This kind of reference makes the referenced object not eligible for GC. That is, whenever an object is referenced by a *chain of strong Reference Objects*, it cannot be garbage collected.

**Weak Reference Object**

WeakReference<StringBuilder> weakBuilder = new WeakReference<StringBuilder>(builder);

Weak Reference Objects are not the default type/class of Reference Object and to be used they should be explicitly specified like in the above example. This kind of reference makes the reference object eligible for GC. That is, in case the only reference reachable for the StringBuilder object in memory is, actually, the weak reference, then the GC is allowed to garbage collect the StringBuilder object. When an object in memory is reachable only by Weak Reference Objects, it becomes automatically eligible for GC.

**Levels of Weakness**

Two different levels of weakness can be enlisted: *soft* and *phantom*.

A *soft* Reference Object is basically a weak Reference Object that remains in memory a bit more: normally, it resists GC cycle until memory is available and there is no risk of OutOfMemoryError (in that case, it can be removed).

On the other hand, a *phantom* Reference Object is useful only to know exactly when an object has been effectively removed from memory: normally they are used to fix *weird finalize() revival/resurrection behavior*, since they actually do not return the object itself but only help [in keeping track of their memory presence](https://community.oracle.com/blogs/enicholas/2006/05/04/understanding-weak-references).

Weak Reference Objects are ideal to implement cache modules. In fact, a sort of automatic eviction can be implemented by allowing the GC to clean up memory areas whenever objects/values are no longer reachable by strong references chain. An example is the [WeakHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/WeakHashMap.html) retaining weak keys.

**What does the keyword synchronized mean?**

The synchronized keyword is all about different threads reading and writing to the same variables, objects and resources. This is not a trivial topic in Java, but here is a quote from Sun:

synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.

*In a very, very small nutshell:* When you have two threads that are reading and writing to the same 'resource', say a variable named foo, you need to ensure that these threads access the variable in an atomic way. Without the synchronized keyword, your thread 1 may not see the change thread 2 made to foo, or worse, it may only be half changed. This would not be what you logically expect.

**What is a ThreadPoolExecutor?**

A thread pool manages a pool of worker threads (the exact number varies depending upon how it’s implementation).

A task queue holds tasks waiting to be executed by any one of the idle threads in the pool.

The **ThreadPoolExecutor** executes a given task using one of its threads from the thread pool. It’s a powerful task execution framework as it supports **task addition in a queue, task cancellation, and task prioritization.**

ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(

int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue

);

**corePoolSize:**The minimum number of threads to keep in the pool. Initially there are zero threads in the pool. But as tasks are added to the queue, new threads are created.

**maximumPoolSize:** The maximum number of threads allowed in the pool.

**keepAliveTime:**When the number of threads is greater than the core, the noncore threads (excess idle threads) will wait for a new tasks, and if they don’t get one within the time defined by this parameter, they will terminate.

**unit:**The unit of time for **keepAliveTime**.

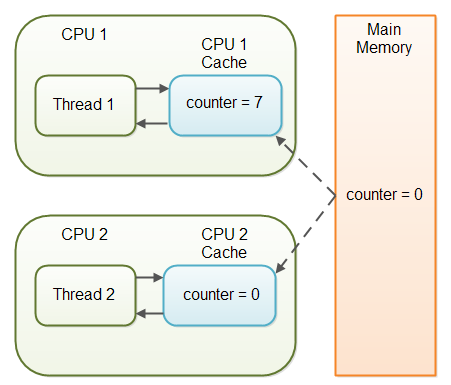
**workQueue:**The task queue, which will only hold runnable tasks. It will have to be a [BlockingQueue](http://developer.android.com/reference/java/util/concurrent/BlockingQueue.html" \t "_blank).

**What is volatile modifier?**

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Imagine too, that only Thread 1 increments the counter variable, but both Thread 1 and Thread 2 may read the counter variable from time to time.

If the counter variable is not declared volatile there is no guarantee about when the value of the countervariable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

The Java volatile keyword is intended to address variable visibility problems. By declaring the countervariable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory.

**How does the try{} catch{} finally{} works?**

You can attach a finally-clause to a try-catch block. The code inside the finally clause will always be executed, even if an exception is thrown from within the try or catch block. If your code has a return statement inside the try or catch block, the code inside the finally-block will get executed before returning from the method.

No matter whether an exception is thrown or not inside the try or catch block the code inside the finally-block is executed.

If an exception is thrown inside a finally block, and it is not caught, then that finally block is interrupted just like the try-block and catch-block is.

**What is serialization?**

Serialization is the process of converting an object into a stream of bytes in order to store an object into memory, so that it can be recreated at a later time, while still keeping the object's original state and data. In Android you may use either the Serializable, Externalizable (implements Serializable) or Parcelable interfaces.

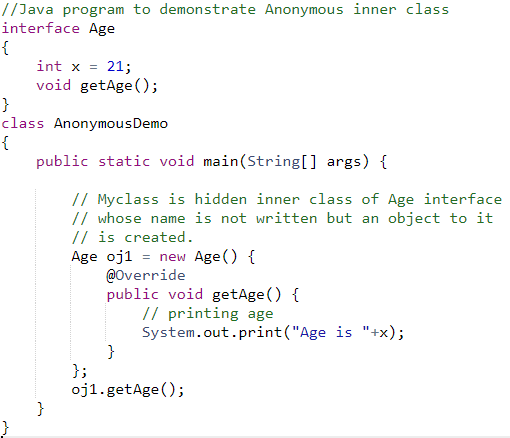
While Serializable is the easiest to implement, Externalizable may be used if you need to insert custom logic into the process of serialization (although it is almost never used nowadays as it is considered a relic from early versions of Java). But it is highly recommended to use Parcelable in Android instead, as Parcelable was created exclusively for Android and it performs about 10x faster than Serializable, because Serializable uses reflection, which is a slow process and tends to create a lot of temporary objects and it may cause garbage collection to occur more often.

**What is transient modifier?**

**Java transient** keyword is used in serialization. If you define any data member as transient, it will not be serialized.

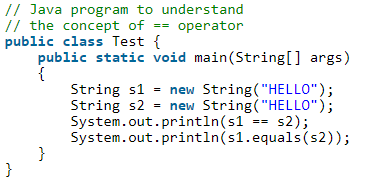
**What are anonymous classes?**

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name.



**What is the difference between using == and .equals on an object?**

We can use == operators for reference comparison (**address comparison**) and .equals() method for **content comparison**. In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.



**What is the hashCode() and equals() used for?**

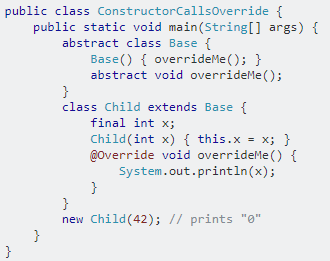
***hashcode():***a method provided by **java.lang.Object**that returns an integer representation of the object memory address. By default, this method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.

You **must override** hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.

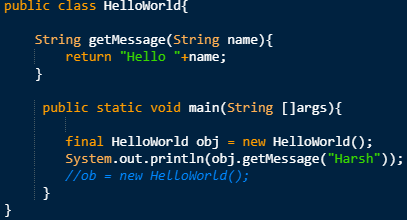
**Why would you not call abstract method in constructor?**

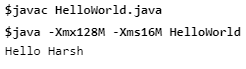
There are a few more restrictions that a class must obey to allow inheritance. **Constructors must not invoke overridable methods**, directly or indirectly. If you violate this rule, program failure will result. The superclass constructor runs before the subclass constructor, so the overriding method in the subclass will be invoked before the subclass constructor has run. If the overriding method depends on any initialization performed by the subclass constructor, the method will not behave as expected.



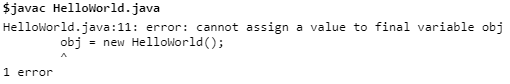
Here, when Base constructor calls overrideMe, Child has not finished initializing the final int x, and the method gets the wrong value. This will almost certainly lead to bugs and errors.

**When would you make an object value final?**





After uncommenting the commented lines, the following error occurs.



'obj' is a reference to an object, and the real object is on the heap. the reference tells you how to get there.

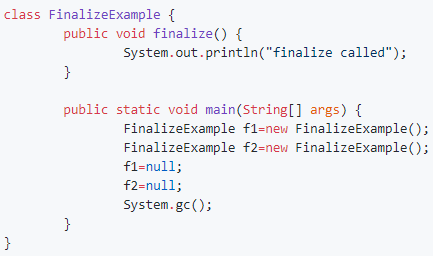
You can kind of think of it as an index card that holds an address. Normally, you write the address in pencil. you can erase or change it. when you say "final HelloWorld obj", you are writing the address in ink - it cannot be changed or erased.

**What are these final, finally and finalize keywords?**

**final** is a keyword in the java language. It is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

**finally** is a code block and is used to place important code, it will be executed whether exception is handled or not.

**finalize** is a method used to perform clean up processing just before object is garbage collected.

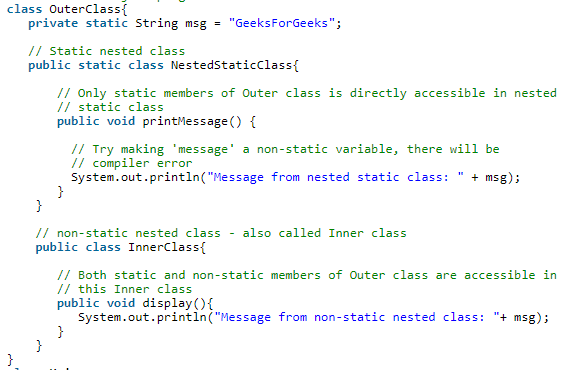


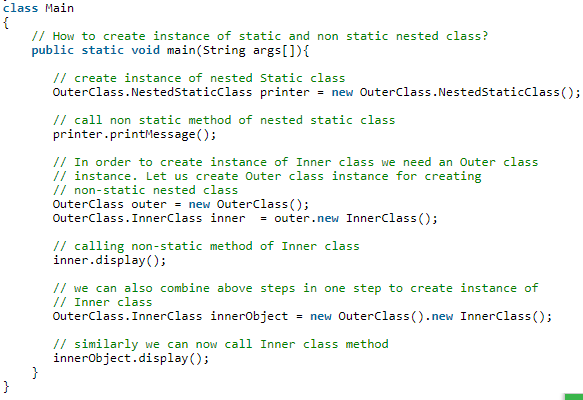
**What does the static word mean in Java?**

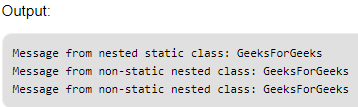
In case of static variable it means that this variable (its' value or the object it references) spans across all instances of enclosing class (changing it in one instance affects all others), while in case of static methods it means that these methods can be invoked without an instance of their enclosing class.

**Can a class be static in Java?**

The answer is YES, we can have static class in java. Java allows us to define a class within another class. Such a class is called a nested class. The class which enclosed nested class is known as Outer class. In java, we can’t make Top level class static. ***Only nested classes can be static***.







**Can a static method be overridden in Java?**

While child class can override a static method with another static method with the same signature (return type can be downcasted), it is not truly overridden - it becomes "hidden", but both methods can still be accessed under right circumstances (see question about overloading/overriding above).

**When is a static block run?**

Code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class).

**What is reflection?**

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

The required classes for reflection are provided under java.lang.reflect package.

Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.

Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

Reflection can be used to get information about –

1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
3. **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

**What is Dependency Injection? Can you name few libraries? Have you used any?**

Dependency injection is a very powerful technique, where you relay the task of providing object with its' dependencies on instances of other objects (OOP Composition, [Wikipedia](https://en.wikipedia.org/wiki/Object_composition?oldformat=true)) to a separate class. This allows for fewer constructors, setters, factories and builders as all those functions are taken care of by the DI framework that you use. Also, and it may seem as a minor advantage, but if you use DI framework you need not worry about going through the project and changing all of (example names) YourCustomInterface customInterfaceObject = new YourCustomClass(); to a new implementaion, as long as your new class (in place of YourCustomClass) still implements CustomInterface - you can just tweak the DI factory class to produce new class and voila - this new class will be automatically instantiated throughout your code. This allows for better maintenence and control over the program. Another example of DI usage is unit-testing - it allows to conveniently inject all needed dependencies and keep the amount of written code at a lower level.

One of the most popular libraries for DI for Android is Dagger 2.

**Difference between StringBuffer and StringBuilder?**

StringBuffer is the **thread safe** utility class to perform several operations on Strings. It contains append() and insert() methods that are widely used to perform operation on Strings in a multi-thread environment. If you will check the source code, most of its functions are synchronized for thread safety.

Since most of the String operations, for example concatenation happens in a single thread environment, Java 1.5 introduced another utility class **StringBuilder** to perform similar operations but doesn’t provide thread safety. If you will look into its source code, all the methods are unsynchronized. This is the most important point for StringBuffer vs StringBuilder.

StringBuilder is faster than StringBuffer because of no synchronization.

**What is Java NIO?**

Java NIO (New IO) is an alternative IO API for Java (from Java 1.4), meaning alternative to the standard[**Java IO**](http://tutorials.jenkov.com/java-io/index.html) and [**Java Networking**](http://tutorials.jenkov.com/java-networking/index.html) API's. Java NIO offers a different way of working with IO than the standard IO API's.

In the standard IO API you work with byte streams and character streams. In NIO you work with **channels** and **buffers**. Data is always read from a channel into a buffer or written from a buffer to a channel. A Channel is a bit like a stream. From the Channel data can be read into a Buffer. Data can also be written from a Buffer into a Channel.

Java NIO enables you to do **non-blocking** IO. For instance, a thread can ask a channel to read data into a buffer. While the channel reads data into the buffer, the thread can do something else. Once data is read into the buffer, the thread can then continue processing it. The same is true for writing data to channels.

Java NIO contains the concept of "**selectors**". A selector is an object that can monitor multiple channels for events (like: connection opened, data arrived etc.). Thus, a single thread can monitor multiple channels for data.