**Notes**

* A constructor call **must** always return a new object by design.
* Java ensures that any enum value is instantiated only once in a Java program. By default, the Enum instance is thread-safe.
* If the class is declared public, then the default constructor is public; if the class is declared protected, then the default constructor is protected
* JDK is the one which is used to write Java programs and compile the code
* JRE is the one that runs our code
* JVM is part of JRE. It’s not platform-independent. It consists of 3 components ClassLoader, Runtime Memory, and Execution Engine.
* If JVM doesn’t find the class it throws ***NoClassDefFoundError*** or ***ClassNotFoundException***.
* Variables start with a letter, underscore, or dollar sign. A variable name cannot be started with a number. No spaces or special characters are allowed in the variable. Java keywords cannot be used in a variable name. Uppercase characters are distinct from lowercase characters since Java is a case-sensitive language. Hence john's age is different from John’s Age.
* Constructor cannot be declared as final, or static.
* We can handle exception handling in static blocks. Static block can have try and catch. Static block provides the option for exception handling
* ***transient*** keyword is used if we don’t want some object property to be converted to stream in serializing the object. Similarly, ***static variable*** values are also not serialized since they belong to the class and not an object.
* The ***volatile*** keyword in Java is used to mark a Java variable as “being stored in main memory”. Every thread that accesses a volatile variable will read it from main memory, and not from the CPU cache. This way, all threads see the same value for the volatile variable.
* By implementing ***validateObject()*** method in **ObjectInputValidation** interface, we can add some business validations to the class variables to make sure that the data integrity is not harmed during serialization.
* ***Interface*** can extend another interface (Interface interfaceB extends interfaceA )
* Variables created outside a lambda expression cannot be modified inside it.
* Java Enum values are globally accessible same as static variable.
* Inner class which is ***private static*** can be accessed by the outer class, other classes can’t access it.
* Till Java 1.6, try block should be followed by either catch or finally block but from Java 7 we can have only try with resource block with out catch & finally blocks
* ***Static methods*** can not be overridden. But can be overloaded since they are resolved using static binding by the compiler at compile time.
* Keys and value can’t be primitive datatype. Key in Hashmap is valid if it implements hashCode() and equals() method , it should also be immutable (immutable custom object ) so that hashcode and equality remains constant. Value in hashmap can be any wrapper class, custom objects, arrays, any reference type or even null . For example***, Hashmap can have array as value but not as key.***
* **Map.put()** 🡪 Returns the value to which this map previously associated the key, or null if the map contained no mapping for the key.

**Advantages of Method Overloading**

* Cleanliness of code and increases readability of the program.
* Reduces the execution time because the binding is done in compilation time itself.

**Method Overloading vs Method Overriding**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Threads**

* The purpose of ***start()*** method is to create a separate call stack for the thread. A separate call stack is created by it, and then run() is called by JVM.
* If we have multiple threads from 1 object then synchronization will handle the data racing or bad output. If multiple threads are assigned from multiple objects then that may result in bad output or data racing.
* When we have multiple threads from a single object than we can use synchronization to lock the object to pass 1 thread at a time and prevent bad output.
* The ***join()*** method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution.
* If multiple threads are assigned from multiple objects than we can not use synchronization. In such cases providing lock to the object will result in data racing. we need to use static synchronization and provide lock to the class. The class will select 1 object at a time. The object in turn will choose 1 thread and pass it to the sensitive area.
* Difference between Synchronized and Static Synchronized in Java

|  |  |
| --- | --- |
| **Synchronized** | **Static Synchronized** |
| It requires an object-level lock. | It requires a class-level lock. |
| Its method need not be declared static. | Its method needs to be declared static. |
| It is used regularly. | It is not used regularly. |
| A different instance is created for each object. | Only one instance for the entire program. |

* Java synchronized keyword cannot be used for constructors and variables.
* The Object class in java contains three final methods that allow threads to communicate about the lock status of a resource. These methods are ***wait(), notify()*** and ***notifyAll()***
* **jstack**: Java comes with jstack tool through which we can generate thread dump for a java process. This is a two step process.

1. Find out the PID of the java process using ps -eaf | grep java command
2. Run jstack tool as jstack PID to generate the thread dump output to console, you can append thread dump output to file using command “jstack PID >> mydumps.tdump”

* Java 8 has introduced **jcmd** utility. You should use this instead of jstack if you are on Java 8 or higher. Command to generate thread dump using jcmd is ***jcmd PID Thread.print.***
* Java *java.util.Timer* is a utility class that can be used to schedule a thread to be executed at certain time in future. Java Timer class can be used to schedule a task to be run one-time or to be run at regular intervals. java.util.TimerTask is an abstract class that implements Runnable interface and we need to extend this class to create our own TimerTask that can be scheduled using java Timer class
* Java Callable and Future are used a lot in multithreaded programming. Java 5 introduced ***java.util.concurrent.Callable*** interface in concurrency package that is similar to Runnable interface but it can return any Object and able to throw Exception.
* The argument value for milliseconds in Thread.sleep(ms) cannot be negative. Otherwise, it throws IllegalArgumentException.
* Java **BlockingQueue** doesn’t accept null values and throw NullPointerException if you try to store null value in the queue. Java BlockingQueue implementations are thread-safe. Java provides several BlockingQueue implementations such as ***ArrayBlockingQueue***, ***LinkedBlockingQueue***, ***PriorityBlockingQueue***, ***SynchronousQueue*** etc.

**Access Specifiers**

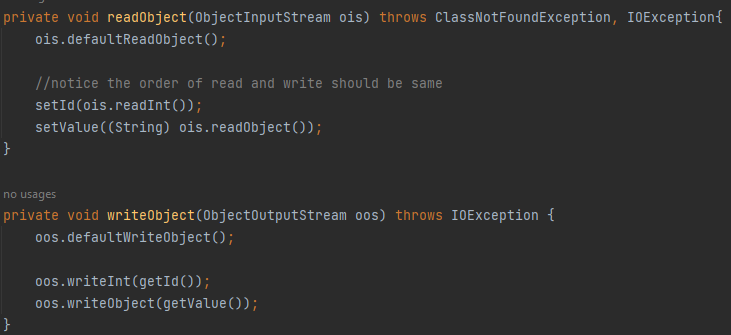


Inner class:

* Classes are not the only place where the inner classes can be defined. We can also define an inner class inside a block level code, such as within a for loop or within a method.
* Inner class has access to all the members in the outer class, but members in the outer class are not accessible in the inner class. It can be accessed only by creating objects of Inner class in the outer class.

**Serialization**

* ***transient*** keyword is used if we don’t want some object property to be converted to stream in serializing the object. Similarly, ***static variable*** values are also not serialized since they belong to the class and not an object.
* There are four methods that we can provide in the class to change the serialization behavior
* **readObject(ObjectInputStream ois)**: If this method is present in the class, ObjectInputStream readObject() method will use this method for reading the object from stream.
* **writeObject(ObjectOutputStream oos):** If this method is present in the class, ObjectOutputStream writeObject() method will use this method for writing the object to stream. One of the common usage is to obscure the object variables to maintain data integrity.
* **Object writeReplace():** If this method is present, then after the serialization process this method is called and the object returned is serialized to the stream.
* **Object readResolve():** If this method is present, then after the deserialization process, this method is called to return the final object to the caller program. One of the uses of this method is to implement a Singleton pattern with Serialized classes.
* If we need to extend a class that doesn’t implement a Serializable interface, we have to use ***readObject()*** and ***writeObject()*** methods to serialize the Superclass also.

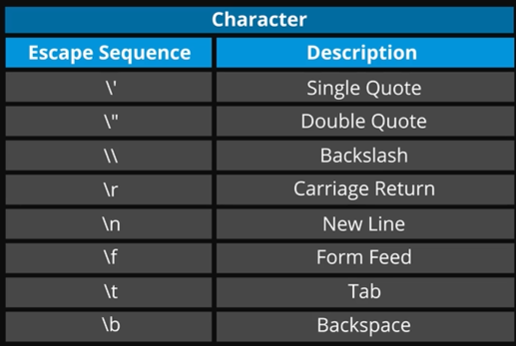


* **ObjectInputValidation** Interface - By implementing the ***validateObject()*** method, we can put some business validations to make sure that the data integrity is not harmed.
* **Java Serialization Proxy pattern** is a way to achieve greater security with Serialization. In this pattern, ***an inner private static class*** is used as a proxy class for serialization purposes. This class is designed in a way to maintain the state of the main class. This pattern is implemented by properly implementing ***readResolve()*** and ***writeReplace()*** methods.
* **Externalization** serves the purpose of custom Serialization, where we can decide what to store in the stream.
* **Externalizable** interface present in java.io, is used for Externalization which ***extends Serializable interface***. It consists of two methods that we have to override to write/read objects into/from the stream which are
* void readExternal(ObjectInput in)
* void writeExternal(ObjectOutput out)

**Cloning**

* There are 3 types of cloning: shallow copy, deep copy and lazy copy
* **Shallow Copy**
* Whenever we use default implementation of clone method we get shallow copy of object means it creates new instance and copies all the field of object to that new instance and returns it as object type, we need to explicitly cast it back to our original object. This is shallow copy of the object.
* clone() method of the object class support shallow copy of the object. If the object contains primitive as well as non primitive or reference type variable in shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves.
* That’s why the name shallow copy or shallow cloning in Java. If only primitive type fields or Immutable objects are there then there is no difference between shallow and deep copy in Java.

**Escape Sequences**



**Exceptions**

* Some Important Points
* We can’t have catch or finally clause without a try statement.
* A try statement should have either catch block or finally block, it can have both blocks. We can use only try also when we use try with resources
* We can’t write any code between try-catch-finally blocks.
* We can have multiple catch blocks with a single try statement.
* try-catch blocks can be nested similar to if-else statements.
* We can have only one finally block with a try-catch statement.
* **Order of Exception:** If you have multiple catch blocks for a single try and if the exceptions classes of them belong to the same hierarchy, You need to make sure that the catch block that catches the exception class of higher-level is at last at the last in the order of catch blocks.
* The catch block catching the Exception object should be placed at last in the order of the catch blocks.

**Maps**

* Keys and value can’t be primitive datatype. Key in Hashmap is valid if it implements hashCode() and equals() method , it should also be immutable (immutable custom object ) so that hashcode and equality remains constant. Value in hashmap can be any wrapper class, custom objects, arrays, any reference type or even null . For example, **Hashmap can have array as value but not as key.**
* HashMap in Java implements Serializable, Cloneable, Map<K, V> interfaces.Java HashMap extends AbstractMap<K, V> class. The direct subclasses are ***LinkedHashMap*** and ***PrinterStateReasons***.

**Questions**

* **Why non-static variables cannot be referenced from a static method in Java?**
  + For the non-static variable, there is a need for an object instance to call the variables. We can also create multiple objects by assigning different values for that non-static variable. So, different objects may have different values for the same variable. So there will be ambiguity for the compiler to understand the value.
* **Why Java Interfaces Cannot Have Constructor But Abstract Classes Can Have?**
* Methods present in the interface are only declared not defined, As there is no implementation of methods, there is no need of making objects for calling methods in the interface and thus no point in having a constructor in it.
* A constructor is used to initialize non-static data members and as there are no non-static data members in the interface, there is no need of a constructor
* If we try to create a constructor inside the interface, the compiler will give a compile-time error.
* The main purpose of the constructor is to initialize the newly created object. In abstract class, we have an instance variable, abstract methods, and non-abstract methods. We need to initialize the non-abstract methods and instance variables, therefore abstract classes have a constructor.
* If we don’t provide any constructor the compiler will add a default constructor in an abstract class.
* The constructor inside the abstract class can only be called during constructor chaining i.e. when we create an instance of sub-classes. This is also one of the reasons abstract class can have a constructor.
* **Why a Constructor cannot be final, static or abstract in Java?**
* As we know, constructors are not inherited in java. Therefore, constructors are not subject to hiding or overriding. When there is no chance of constructor overriding, there is no chance of modification also. So constructors cannot be ***final***.
* We know static keyword belongs to a class rather than the object of a class. A constructor is called when an object of a class is created, so no use of the static constructor. Another thing is that if we will declare static constructor then we cannot access/call the constructor from a subclass. Because we know static is allowed within a class but not by a subclass. So constructors cannot be ***static***.
* If we are declaring a constructor as abstract as we have to implement it in a child class, but we know a constructor is called implicitly when the new keyword is used so it can’t lack a body and also it cannot be called as a normal method. Also, if we make a constructor abstract then we have to provide the body later. But we know constructor cannot be overridden so providing body is impossible. So constructors cannot be ***abstract***.
* **What are Nested classes in Java?**
* Nested classes are divided into two categories namely ***static*** and ***non-static***. Nested classes that are declared static are called static nested classes. Non-static nested classes are called inner classes.
* There are two kinds of classes in Java, one is called a ***top-level class*** and the other is called a ***nested class***. As the name suggested top-level class is a class that is declared in ‘.java’ file.
* A nested class is declared inside another class. The class which enclosed nested class is known as Outer class.
* We can not make a top-level class static. You can only make nested classes either static or non-static. If you make a nested class non-static then it also referred to as Inner class.
* **What are the differences between Static and Non-Static Nested Class in Java?**
* The static inner class can access the static members of the outer class directly. But, to access the instance members of the outer class you need to instantiate the outer class.
* Nested static class doesn’t need a reference of the Outer class but a nonstatic nested class or Inner class requires an Outer class reference.
* A non-static nested class has full access to the members of the class within which it is nested. A static nested class does not have a reference to a nesting instance, so a static nested class cannot invoke non-static methods or access non-static fields of an instance of the class within which it is nested.
* Another difference between static and non-static nested class is that you cannot access non-static members e.g. method and field into nested static class directly. If you do you will get errors like “nonstatic member cannot be used in the static context”. While the Inner class can access both static and non-static members of the Outer class.
* **What is volatile keyword in java?**
* The volatile keyword in Java is used to indicate that a variable's value may be modified by multiple threads simultaneously. It ensures that the variable is always read from and written to the main memory, rather than from thread-specific caches, ensuring visibility across threads.
* This keyword guarantees that any change made to the variable is immediately visible to all threads, preventing potential inconsistencies and data races. However, volatile does not provide atomicity or synchronization, so additional synchronization mechanisms should be used in conjunction with it when necessary.
* The volatile keyword cannot be used with classes or methods, it only applies to variables
* **What is the use of serialVersionUID?**
* If serialVersionUID is not present in the class and if we deserialize the object and if there was a change in the class after serialization we will get exception ***as java.io.InvalidClassException: com.journaldev.serialization.Employee; local class incompatible: stream classdesc serialVersionUID = -6470090944414208496, local class serialVersionUID = -6234198221249432383***
* **Can we have try and catch block inside a static block?**
* Yes, we can.
* **On what order we should catch the exceptions?**
* If you have multiple catch blocks for a single try and if the exception classes belong to the same hierarchy, We need to make sure that the catch block that catches the exception class of higher-level is at last at the last in the order of catch blocks. The catch block catching the **Exception** object should be placed at last in the order of the catch blocks.