Serialization& Deserailization

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the java.io.Serializable interface.

The ObjectOutputStream class contains writeObject() method for serializing an Object.

public final void writeObject(Object obj)

throws IOException

The ObjectInputStream class contains readObject() method for deserializing an object.

public final Object readObject()

throws IOException,

ClassNotFoundException

Marker Interface : Serializable is a marker interface (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.

2. Only non-static data members are saved via Serialization process.

3. Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.

4. Constructor of object is never called when an object is deserialized.

5. Associated objects must be implementing Serializable interface.

i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

The reason being a was marked as transient and b was static.

In case of transient variables:- A variable defined with transient keyword is not serialized during serialization process.This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0).

In case of static Variables:- A variable defined with static keyword is not serialized during serialization process.This variable will be loaded with current value defined in the class during deserialization.

What is Garbage Collections ?

When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

But in Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying unreachable objects. The garbage collector is the best example of the Daemon thread as it is always running in the background.

There are generally four ways to make an object eligible for garbage collection.

#Nullifying the reference variable

#Re-assigning the reference variable

#An object created inside the method

#Island of Isolation

Using System.gc() method: System class contain static method gc() for requesting JVM to run Garbage Collector.

Using Runtime.getRuntime().gc() method: Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

ust before destroying an object, Garbage Collector calls finalize() method on the object to perform cleanup activities. Once finalize() method completes, Garbage Collector destroys that object.

protected void finalize() throws Throwable

Finalize () method is present in Throable Class

In case of a singleton class, the reference of the only object created can be stored in a static reference. Since static members are stored in class area (a memory segment), their lifetime spans the lifetime of the program --> How to not delete an Object

How to prevent garbage collection using finalize?

The finalize method of a class can be overridden to preserve the reference of the object that is about to be deleted. The following program demonstrates, how:

// Java code to demonstrate how to prevent garbage collection

// of an object using finalize method

class A {

static A y;

void f()

{

A x = new A();

}

pubic void finalize()

{

y = this; // Putting the reference id

// of the current object

// into the static variable y

In Java, inner class refers to the class that is declared inside class or interface which were mainly introduced, to sum up, same logically relatable classes as Java is purely object-oriented so bringing it closer to the real world. Now geeks you must be wondering why they were introduced?

There are certain advantages associated with inner classes are as follows:

Making code clean and readable.

Private methods of the outer class can be accessed, so bringing a new dimension and making it closer to the real world.

Optimizing the code module.

Types of Inner Classes

There are basically four types of inner classes in java.

Nested Inner Class

Method Local Inner Classes

Static Nested Classes

Anonymous Inner Classes

Nested Inner Class :

It can access any private instance variable of the outer class. Like any other instance variable, we can have access modifier private, protected, public, and default modifier. Like class, an interface can also be nested and can have access specifiers.

Example :

// Helper classes

class Outer {

// Class 2

// Simple nested inner class

class Inner {

// show() method of inner class

public void show()

{

// Print statement

System.out.println("In a nested class method");

}

}

}

// Class 2

// Main class

class Main {

// Main driver method

public static void main(String[] args)

{

// Note how inner class object is created inside

// main()

Outer.Inner in = new Outer().new Inner();

// Calling show() method over above object created

in.show();

}

}

*We can not have a static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself. For example, the following program doesn’t compile.*

**Type 2:**Method Local Inner Classes

Inner class can be declared within a method of an outer class

Method Local inner classes can’t use a local variable of the outer method until that local variable is not declared as final.

*Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in method local inner class.*

**Type 3:**Static Nested Classes

Static nested classes are not technically inner classes. They are like a static member of outer class.

// Java Program to Illustrate Static Nested Classes

// Importing required classes

import java.util.\*;

// Class 1

// Outer class

class Outer {

// Method

private static void outerMethod()

{

// Print statement

System.out.println("inside outerMethod");

}

// Class 2

// Static inner class

static class Inner {

public static void display()

{ // Print statement

System.out.println("inside inner class Method");

// Calling method inside main() method

outerMethod();

}

}

}

// Class 3

// Main class

class GFG {

// Main driver method

public static void main(String args[])

{

Outer.Inner obj = new Outer.Inner();

// Calling method via above instance created

obj.display();

}

}

Nested Classes :

In Java, it is possible to define a class within another class, such classes are known as *nested* classes. They enable you to logically group classes that are only used in one place, thus this increases the use of [encapsulation](https://www.geeksforgeeks.org/encapsulation-in-java/), and creates more readable and maintainable code.

* The scope of a nested class is bounded by the scope of its enclosing class. Thus in below example, class *NestedClass* does not exist independently of class *OuterClass*.
* A nested class has access to the members, including private members, of the class in which it is nested. The reverse is also true i.e., the enclosing class can access the members of the nested class.
* A nested class is also a member of its enclosing class.
* As a member of its enclosing class, a nested class can be declared *private*, *public*, *protected*, or *package private*(default).
* Nested classes are divided into two categories.
* **static nested class :** Nested classes that are declared *static* are called static nested classes.
* **inner class :**An inner class is a non-static nested class.

Java.util. Object Class:

There are 9 utility methods in Object Class

They can easily handle Null Pointer Exception and customize Null Pointer Exception method .

1. toString(Onject o)
2. **toString(Object o, String nullDefault)** : This method is overloaded version of above method. It returns the result of calling toString() method on the first argument if the first argument is not null and returns the second argument otherwise.
3. **equals(Object a,Object b)** : This method true if the arguments are equal to each other and false otherwise. Consequently, if both arguments are null, true is returned and if exactly one argument is null, false is returned. Otherwise, equality is determined by using the equals() method of the first argument.
4. **deepEquals(Object a,Object b)** :This method returns true if the arguments are deeply equal to each other and false otherwise. Two null values are deeply equal. If both arguments are arrays, the algorithm in [Arrays.deepEquals](http://contribute.geeksforgeeks.org/geek/) is used to determine equality. Otherwise, equality is determined by using the equals method of the first argument.
5. **requireNonNull(T obj)** : This method checks that the specified object reference is not null. This method is designed primarily for doing parameter validation in methods and constructors
6. **requireNonNull(T obj,String message)** : This method is overloaded version of above method with customized message printing if obj is null
7. **int hashCode(Object o)** : This method returns the hash code of a non-null argument and 0 for a null argument.
8. **int hash(Object… values)** : This method generates a hash code for a sequence of input values. The hash code is generated as if all the input values were placed into an array, and that array were hashed by calling Arrays.hashCode(Object[]).  
   This method is useful for implementing Object.hashCode() on objects containing multiple fields. For example, if an object that has three fields, x, y, and z, one could write:

@Override

public int hashCode() {

return Objects.hash(x, y, z);

}

**Note**: When a single object reference is supplied, the returned value does not equal the hash code of that object reference. This value can be computed by calling hashCode(Object).

1. **compare(T a,T b,Comparator c)** : As usual, this method returns 0 if the arguments are identical and c.compare(a, b) otherwise. Consequently, if both arguments are null 0 is returned.

Note that if one of the arguments is null, a NullPointerException may or may not be thrown depending on what ordering policy.

From Java 8, 3 new Methods included :

public static boolean isNull([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html" \o "class in java.lang) obj)

Returns true if the provided reference is null otherwise returns false.

public static boolean nonNull([Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html" \o "class in java.lang) obj)

Returns true if the provided reference is non-null otherwise returns false.

public static <T> T requireNonNull(T obj,

[Supplier](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html)<[String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html)> messageSupplier)

Checks that the specified object reference is not null and throws a customized [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html" \o "class in java.lang) if it is.

Unlike the method [requireNonNull(Object, String)](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html" \l "requireNonNull-T-java.lang.String-), this method allows creation of the message to be deferred until after the null check is made.

Q . How many ways to create Object ??

There are 5 ways :

1. Using new keyword
2. Using new instance
3. Using clone() method
4. Using deserialization
5. Using newInstance() method of Constructor class

New Keyword: A a=new A();

New Instance : If we know the name of the class & if it has a public default constructor we can create an object **Class.forName**. We can use it to create the Object of a Class. Class.forName actually loads the Class in Java but doesn’t create any Object. To create an Object of the Class you have to use the new Instance Method of the Class.

Clalss Name abc

Class cls=Class.forName(abc);

That class Is Referred , now to create a Instance

Abc obj= (abc)cls.newInstance();

Clone (): Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor. In order to use the clone() method on an object we need to implement [Cloneable](https://www.geeksforgeeks.org/cloneable-interface-in-java/) and define the [*clone() method*](https://www.geeksforgeeks.org/clone-method-in-java-2/) in it.

Like Class is xyz:

Xyz o1=new xyz();

Xyz o2=(xyz)o1.clone();

Deerialinzation :

Whenever we serialize and then deserialize an object, JVM creates a separate object. In **deserialization**, JVM doesn’t use any constructor to create the object. To deserialize an object we need to implement the Serializable interface in the class.

**New Instance() Method :**There is one newInstance() method in the**[java.lang.reflect.Constructor class](https://www.geeksforgeeks.org/java-lang-reflect-constructor-class-in-java/)** which we can use to create objects. It can also call the parameterized constructor, and private constructor by using this newInstance() method. Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class internally uses newInstance() method of Constructor class.

***Note: When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does. That’s why we said that java is***[***strictly pass-by-value***](https://www.geeksforgeeks.org/g-fact-31-java-is-strictly-pass-by-value/)***.***

***NOTE:***

**transient** is a variables modifier used in [serialization](https://www.geeksforgeeks.org/serialization-in-java/). At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use **transient** keyword. When JVM comes across **transient**keyword, it ignores original value of the variable and save default value of that variable data type.

**transient** keyword plays an important role to meet security constraints. There are various real-life examples where we don’t want to save private data in file. Another use of **transient** keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc.  
Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use **transient** keyword with private confidential fields of a class during serialization.

Volatile Keyword :

class SharedObj

{

// volatile keyword here makes sure that

// the changes made in one thread are

// immediately reflect in other thread

static **volatile** int sharedVar = 6;

}

Note that volatile should not be confused with the static modifier. static variables are class members that are shared among all objects. There is only one copy of them in the main memory.

**volatile vs synchronized:** Before we move on let’s take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility**: It means that changes made by one thread to shared data are visible to other threads.

Java’s synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modify the value of the shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other threads trying to enter the block at the same time will be blocked and put to sleep.

In some cases, we may only desire visibility and not atomicity. The use of synchronized in such a situation is overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features.

**strictfp** is a modifier that stands for strict floating-point which was not introduced in the base version of java as it was introduced in Java version 1.2. It is used in java for restricting floating-point calculations and ensuring the same result on every platform while performing operations in the floating-point variable Not allowed with variable , only allowed with Classes , interface and Methods .

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8/) can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***,***Comparable*** are some of the examples of functional interfaces.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

class Test {

    public static void main(String args[])

    {

        // create anonymous inner class object

        new Thread(new Runnable() {

            @Override public void run()

            {

                System.out.println("New thread created");

            }

        }).start();

    }

}

From Java 8:

/ Java program to demonstrate Implementation of

// functional interface using lambda expressions

class Test {

    public static void main(String args[])

    {

        // lambda expression to create the object

        new Thread(() -> {

            System.out.println("New thread created");

        }).start();

    }

}

**Java SE 8 included four main kinds of functional interfaces**which can be applied in multiple situations. These are:

1. Consumer
2. Predicate
3. Function
4. Supplier

**1. Consumer**

The consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument. The consumer interface has no return value. It returns nothing. There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments.

Consumer<Integer> consumer = (value) -> System.out.println(value);

#### 2. Predicate

In scientific logic, a function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate. Similarly, in the java programming language, a predicate functional interface of java is a type of function which accepts a single value or argument and does some sort of processing on it, and returns a boolean (True/ False) answer. The implementation of the Predicate functional interface also encapsulates the logic of filtering (a process that is used to filter stream components on the base of a provided predicate) in Java.

public interface Predicate<T> {

   boolean test(T t);

}

#### 3. Function

A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing. There are many versions of Function interfaces because a primitive type can’t imply a general type argument, so we need these versions of function interfaces. Many different versions of the function interfaces are instrumental and are commonly used in primitive types like double, int, long

#### 4.Supplier

The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output. This type of functional interface is generally used in the lazy generation of values. Supplier functional interfaces are also used for defining the logic for the generation of any sequence. For example – The logic behind the Fibonacci Series can be generated with the help of the Stream.generate method, which is implemented by the Supplier functional Interface.

**Lambda Expression**

Lambda expressions basically express instances of [functional interfaces](https://www.geeksforgeeks.org/functional-interfaces-java/) (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces

lambda expressions are added in Java 8 and provide below functionalities.

* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.

**Stream in Java :**

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
   List number = Arrays.asList(2,3,4,5);  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

1. A stream consists of source followed by zero or more intermediate methods combined (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
2. Stream is used to compute elements as per the pipelined methods without altering the original value of the object.
3. s