* OOPS Concepts
* Java String
* Collections Framework
* Multithreading
* Generics
* Exception Handling
* Stream API
* Lambda Expressions
* Latest Release Features
* Java EE Frameworks - Spring, Hibernate etc

Some of the advanced java concepts are:

* Heap and Stack Memory
* Garbage Collection
* Reflection API
* Thread Deadlock
* Java ClassLoader
* Java Logging API
* Internationalization in Java
* Java Module System
* **3. What is Java used for?**
* We use Java programming everywhere. We use Java to create standalone programs, web applications, and web services. We can create distributed enterprise applications using Java EE frameworks.
* **4. What is Core Java?**
* Java **SE** is also called Core Java. It is the set of libraries that are part of standard java installation. For example, the Collections framework is part of Core Java. But, Servlet/JSP is part of Java Enterprise Edition.

### 7. What is difference between JavaScript and Java?

Some of the key differences between JavaScript and Java are:

* Java is **Object Oriented Programming Language**. But, JavaScript is **an Object Oriented Scripting language.**
* Java code runs in a virtual machine or browser (Applets) where JavaScript **code runs on browser.**
* We have to compile Java **source** **code** to **byte** code before JVM can understand and execute it. **JavaScript code is text based and we don’t need to compile it.**
* We use JavaScript to **perform browser specific tasks**. We use Java to **create standalone utility apps**, **web applications, and web services.**
* JavaScript is **lightweight** whereas we have to **install Java and configure it to run.**

### 9. Who Created Java?

Java was first created by **James Gosling** at Sun Microsystems. The first public version was released in 1996.

Java Platform, Standard Edition 18

Java SE **18.0.** **1.1** is the latest release of Java SE Platform.

### 9. What do you mean by platform independence of Java?

Platform independence means that **you can run the same Java Program in any Operating System**. For example, you can write java program in Windows and run it in Mac OS.

### 10. What is JVM and is it platform independent?

Java Virtual Machine (JVM) is the heart of java programming language. JVM is responsible for converting **byte code into machine-readable code**. JVM **is not platform-independent**, that’s why you have **different JVM for different operating systems**. We can customize JVM with Java Options, such as allocating minimum and maximum memory to JVM. It’s called virtual because it provides an interface that doesn’t depend on the underlying OS.

### 11. What is the difference between JDK and JVM?

**Java Development Kit (JDK**) is for **development** **purposes** and JVM is a **part of it to execute the java programs**.

JDK provides all **the tools, executables, and binaries required to compile**, debug and execute a Java Program. The execution part is **handled by JVM to provide machine independence**.

### 12. What is the difference between JVM and JRE?

**Java Runtime Environment (JRE**) is the **implementation of JVM**. JRE consists **of JVM and java binaries and other classes to execute** any program successfully. JRE doesn’t contain any development tools like java compiler, debugger, etc. If you want to execute any java program, you should have JRE installed.

### 13. Which class is the superclass of all classes?

**java.lang.Object**is the root class for all the java classes and we don’t need to extend it.

### 14. Why Java doesn’t support multiple inheritances?

Java doesn’t support multiple inheritance in classes because of “Diamond Problem”. To know more about diamond problem with example, read [Multiple Inheritance in Java](https://www.digitalocean.com/community/tutorials/multiple-inheritance-in-java).

However multiple inheritances are supported in interfaces. An **interface can extend multiple interfaces because they just declare the methods and implementation will be present in the implementing class. So there is no issue of the diamond problem with interfaces.**

### 15. Why Java is not a pure Object Oriented language?

Java is not said to be pure object-oriented because it supports **primitive types such as int, byte, short, long, etc**. I believe it brings simplicity to the language while writing our code. Java could have **wrapper objects** for the primitive types but just for the representation, they would not have provided any benefit.

As we know, for all the primitive types we have wrapper classes such as Integer, Long etc that provides some additional methods.

### 16. What is the difference between path and classpath variables?

**PATH** is an **environment variable** used by **the operating system to locate the executables**. That’s why when we install Java or want any executable to be found by OS, we need to add the directory location in the PATH variable. If you work on Windows OS, read this post to learn [how to set up PATH variable on Windows](https://www.digitalocean.com/community/tutorials/java-windows-10-download-install).

**Classpath** is specific to Java and used by java **executables to locate class files**. We can provide the classpath location while running the java application and it can be a directory, ZIP files, JAR files, etc.

### 17. What is the importance of the main method in Java?

The main() method is the **entry point of any standalone java application.** The syntax of the main method is public static void main(String args[]).

Java’s main method is **public and static so that Java runtime can access** it **without initializing the** **class**. The input parameter is an array of String through which we can pass runtime arguments to the java program. Check this post to learn [how to compile and run a java program](https://www.digitalocean.com/community/tutorials/java-hello-world-program).

### 18. What is overloading and overriding in Java?

When we **have more than one method with the same name in a single** **class** but the **arguments are different**, then it is called **method** **overloading**.

The overriding concept comes into the picture **with inheritance when** we have **two methods with the same signature**, one in the parent class and another in the child class. We can use [@Override](https://www.digitalocean.com/community/users/override) annotation in the **child class overridden method to make sure if the parent class method is changed, so is the child class.**

### 19. Can we overload the main() method?

**Yes**, we can have **multiple methods with the name “**main” in a single class. However, if we **run the class**, the java runtime environment will look for the main method with syntax as **public static void main(String args[]).**

### 20. Can we have multiple public classes in a java source file?

**We can’t have more** **than one public class** in a single java source file. A single source file can **have multiple classes that are not public.**

### 21. What is a Java Package and which package is imported by default?

Java package is the mechanism to organize the java classes by grouping them. The grouping logic can be based on functionality or modules based. A java class fully classified name contains package and class name. For example, **java.lang.Object** is the fully classified name of Object class that is part of java.lang package.

The java.lang package is imported by **default** and we don’t need to import any class from this package explicitly.

### 22. What are access modifiers in Java?

Java provides access control through public, private and protected access modifier keywords. When none of these are used, it’s called default access modifier.

A java class can only have public or default access modifier. Read [Java Access Modifiers](https://www.digitalocean.com/community/tutorials/java-access-modifiers) to learn more about these in detail.

### 23. What is a final keyword?

The **final** keyword is used with Class to make sure **no other class can extend it.** For example, the **String class is final** and we can’t extend it.

We can use the final keyword with methods to make sure child classes can’t override it.

Java’s final keyword can be used with variables to make sure that it can be assigned only once. However the state of the variable can be changed, for example, we can assign a final variable to an object only once but the object variables can change later on.

Java interface variables **are by default final and static**.

### 24. What is a static keyword?

The static keyword can be used with **class-level variables** to make it **global** i.e all the objects **will share the same variable.**

We can use static keyword with methods also. A static method can access **only static variables of** class and **invoke only static methods** of the class.

Read more in detail at [java static keyword](https://www.digitalocean.com/community/tutorials/static-keyword-in-java).

### 25. What is finally and finalize in java?

The **finally** block is used **with try-catch to put** the code that you want to get executed always, even if an exception is thrown by the try-catch block. finally block is mostly used to release resources created in the try block.

The **finalize**() is a special method in **Object class** that we can **override in our classes**. This method gets called by **the garbage collector when the object is getting garbage collected**. This method is usually **overridden to release system resources** when the object is garbage collected.

### 26. Can we declare a class as static?

We can’t declare a top-level class as static however an **inner class can be declared as static.** If the inner class is declared as static, it’s called a **static nested class**.

The static nested class is the same as any other top-level class and is nested for only packaging convenience.

Read more about inner classes at [java inner class](https://www.digitalocean.com/community/tutorials/java-inner-class).

### 27. What is static import?

If we have to use any static variable or method from other class, usually we import the class and then use the method/variable with class name.

import java.lang.Math;

//inside class

double test = Math.PI \* 5;

Copy

We can do the same thing by importing the static method or variable only and then use it in the class as if it belongs to it.

import static java.lang.Math.PI;

//no need to refer class now

double test = PI \* 5;

Copy

Use of static import can cause confusion, so it’s better to avoid it. Overuse of static import can make your program unreadable and unmaintainable.

### 28. What is try-with-resources in java?

One of the Java 7 features is the try-with-resources statement for **automatic resource management.** Before Java 7, there was no auto resource management and we should explicitly close the resource. Usually, it was done in the finally block of a try-catch statement. This approach used to cause **memory** **leaks** when we forgot to close the resource.

From Java 7, we can **create resources inside try block and use it**. Java takes care of **closing it as soon as try-catch block** gets finished. Read more at [Java Automatic Resource Management](https://www.digitalocean.com/community/tutorials/java-try-with-resources).

// TRY WITH RESOURCE EXAMPLE

static String readFirstLineFromFile(String path) throws IOException {

**try (FileReader fr = new FileReader(path);**

**BufferedReader br = new BufferedReader(fr))** {

return br.readLine();

}

}

//NORMAL TRY BLOCK

static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {

FileReader fr = new FileReader(path);

BufferedReader br = new BufferedReader(fr);

try {

return br.readLine();

} finally {

br.close();

fr.close();

}

}

### 29. What is a multi-catch block in java?

Java 7 one of the improvement was a multi-catch block where **we can catch multiple exceptions** in a single catch block. This makes our code shorter and cleaner when every catch block has a similar code.

If a catch block handles multiple exceptions, you can separate them using a pipe (|) and in this case, the exception parameter (ex) is final, so you can’t change it.

Read more at [Java multi catch block](https://www.digitalocean.com/community/tutorials/java-catch-multiple-exceptions-rethrow-exception).

### 30. What is a static block?

Java static block is the **group of statements that gets executed when the class** is **loaded** into **memory** by Java [ClassLoader](https://www.digitalocean.com/community/tutorials/java-classloader). It is used to initialize static variables of the class. Mostly it’s used to create static resources when class is loaded.

### 31. What is an interface?

Interfaces are core concepts of java programming language and used a lot not only in JDK but also java design patterns, most of the frameworks and tools. **Interfaces provide a way to achieve abstraction in java** and used to **define the contract for the subclasses** to implement.

Interfaces are good for starting point to define Type and create top level hierarchy in our code. Since a java class can implements multiple interfaces, it’s better to use interfaces as super class in most of the cases. Read more at [java interface](https://www.digitalocean.com/community/tutorials/interface-in-java).

### 32. What is an abstract class?

Abstract classes are used in java to create a class **with some default method implementation for subclasses**. An abstract class can have an abstract method without the body and it can have methods with implementation also.

The abstract keyword is used to create a abstract class. Abstract classes can’t be instantiated and mostly used to provide base for sub-classes to extend and implement the abstract methods and override or use the implemented methods in abstract class. Read important points about abstract classes at [java abstract class](https://www.digitalocean.com/community/tutorials/abstract-class-in-java).

### 33. What is the difference between abstract class and interface?

* The abstract keyword is used to create abstract class whereas interface is the keyword for interfaces.
* Abstract classes can have method implementations whereas interfaces can’t.
* A class can extend only one abstract class but it can implement multiple interfaces.
* We can run an abstract class if it has a main() method whereas we can’t run an interface.

Some more differences in detail are at [Difference between Abstract Class and Interface](https://www.digitalocean.com/community/tutorials/difference-between-abstract-class-and-interface-in-java).

### 34. Can an interface implement or extend another interface?

**Interfaces don’t implement another interface**, they **extend** it. Since interfaces can’t have method implementations, there is no issue of diamond problem. That’s why we have multiple inheritances in interfaces i.e an interface can extend multiple interfaces.

From Java 8 onwards, interfaces can have default method implementations. So to handle diamond problem when a common default method is present in multiple interfaces, it’s mandatory to provide implementation of the method in the class implementing them. For more details with examples, read [Java 8 interface changes](https://www.digitalocean.com/community/tutorials/java-8-interface-changes-static-method-default-method).

### 35. What is a Marker interface?

A marker interface is an **empty interface without any method** but used to **force some functionality in implementing** **classes** by Java. Some of the well known marker interfaces are **Serializable and Cloneable.**

### 36. What are Wrapper classes?

Java wrapper classes are the **Object representation of eight primitive types in java**. All the wrapper classes **in java are immutable** and **final**. Java 5 **autoboxing** and **unboxing** allows **easy conversion between primitive types and their corresponding wrapper classes**.

Read more at [Wrapper classes in Java](https://www.digitalocean.com/community/tutorials/wrapper-class-in-java).

### 37. What is Enum in Java?

Enum was introduced in Java 1.5 as a new **type** **whose fields consist of a fixed set of constants**. For example, in Java, we can create Direction as an enum with fixed fields as EAST, WEST, NORTH, SOUTH.

enum is the keyword to create an enum type and similar to the class. Enum constants **are implicitly static and final**. Read more in detail at [java enum](https://www.digitalocean.com/community/tutorials/java-enum).

**private** FoodType isVegeterian;

**public** **enum** FoodType {

***VEG***, ***NONVEG***

}

### 38. What are Java Annotations?

Java Annotations provide information about the **code** and they have **no direct effect on the** code they annotate. Annotations are introduced in Java 5. Annotation is **metadata** **about the program embedded in the program** **itself**. It can be parsed by **the annotation parsing tool** or the **compiler**. We can also specify annotation availability to either compile-time only or till runtime. Java Built-in annotations are [@Override](https://www.digitalocean.com/community/users/override), [@Deprecated](https://www.digitalocean.com/community/users/deprecated) and [@SuppressWarnings](https://www.digitalocean.com/community/users/suppresswarnings). Read more at [java annotations](https://www.digitalocean.com/community/tutorials/java-annotations).

### 39. What is Java Reflection API and why it’s so important?

Java Reflection API provides the ability to inspect and modify the **runtime behavior of java application**. We can inspect a java class, interface, enum and get their methods and field details. Reflection API is an advanced topic and we should avoid it in normal programming. Reflection API usage can break the design pattern such as Singleton pattern by invoking the private constructor i.e violating the rules of access modifiers.

Even though we don’t use Reflection API in normal programming, it’s very important to have. We can’t have any frameworks such as Spring, Hibernate or servers such as Tomcat, JBoss without Reflection API. They invoke the appropriate methods and instantiate classes through reflection API and use it a lot for other processing.

Read [Java Reflection Tutorial](https://www.digitalocean.com/community/tutorials/java-reflection-example-tutorial) to get in-depth knowledge of reflection api.

### 40. What is Composition in java?

Composition is the design technique to implement **has-a relationship** in classes. We can use **Object composition for code** reuse.

Java composition is achieved by using **instance variables that refer to other objects**. The benefit of using composition is that we can control the **visibility of other objects** to client classes and reuse only what we need. Read more with example at [Java Composition](https://www.digitalocean.com/community/tutorials/composition-in-java-example) example.

### 41. What is the benefit of Composition over Inheritance?

One of the best practices of Java programming is to “**favor composition over inheritance**”. Some of the possible reasons are:

* Any change in the superclass might affect subclass even though we might not be using the superclass methods. For example, if we have a method test() in the subclass and suddenly somebody introduces a method test() in the superclass, we will get compilation errors in the subclass. The composition will never face **this issue because we are using only what methods we need**.
* Inheritance exposes **all the superclass methods** **and variables to the client** and if we have no control in designing superclass, it can lead to security holes. Composition **allows us to provide restricted access to the methods and hence more secure**.
* We can get **runtime binding in composition** where **inheritance binds the classes at compile time**. So composition provides flexibility in the invocation of methods.

You can read more about above benefits of composition over inheritance at [java composition vs inheritance](https://www.digitalocean.com/community/tutorials/multiple-inheritance-in-java).

### 42. How to sort a collection of custom Objects in Java?

We need to implement **Comparable interface** to support sorting of custom objects in a collection. The Comparable interface has **compareTo**(T obj) method which is used by sorting methods and by providing this method implementation, we can provide default way to sort custom objects collection.

However, if you want to sort based on different criteria, such as sorting an Employees collection based on salary or age, then we can create Comparator instances and pass it as sorting methodology. For more details read [Java Comparable and Comparator](https://www.digitalocean.com/community/tutorials/comparable-and-comparator-in-java-example).

### 43. What is inner class in java?

We can define a **class inside a class and they are called nested classes.** Any **non-static nested** class is known as an **inner class**. Inner classes are associated with the **object of the class** and they can access **all the variables and methods of the outer class**. Since inner classes are **associated with the instance**, we **can’t** have **any static variables in them.**

We can have local inner class or anonymous inner class inside a class. For more details read [java inner class](https://www.digitalocean.com/community/tutorials/java-inner-class).

### 44. What is anonymous inner class?

A **local inner class without a name is known as an anonymous inner class.** An anonymous class is defined and **instantiated** in a single **statement**. Anonymous inner class always **extend a class or implement an interface**.

Since an anonymous class has no name, it is **not** possible to **define** **a constructor for** an anonymous class. Anonymous inner classes are accessible **only at the point where it is defined**.

### 45. What is Classloader in Java?

Java **Classloader** is the program that **loads byte code program into memory** when we want to **access any class.** We can create our own classloader by extending ClassLoader class and overriding loadClass(String name) method. Learn more at [java classloader](https://www.digitalocean.com/community/tutorials/java-classloader).

### 46. What are different types of classloaders?

There are three types of built-in Class Loaders in Java.

1. **Bootstrap ClassLoader** – It **loads JDK internal classes**, typically loads rt.jar and other core classes.
2. **Extensions Class Loader** – It loads **classes** from the JDK **extensions** **directory**, **usually $JAVA\_HOME/lib/e**xt directory.
3. **System Class Loader** – It loads classes from the **current classpath** that can be set while invoking a program **using -cp or -classpath command-**line options.

### 47. What is ternary operator in java?

Java ternary operator is the only **conditional operator that takes three operands**. It’s a one **liner replacement for if-then-else** statement and used a lot in java programming. We can use **ternary operator if-else conditions or even switch conditions using nested ternary operators**. An example can be found at [java ternary operator](https://www.digitalocean.com/community/tutorials/java-ternary-operator).

### 48. What does super keyword do?

The super keyword can be **used to access the superclass method** when **you have overridden the method in the child class**.

We can use the **super keyword to invoke superclass** **constructors** in child class constructor but in this case, it should be the first statement in the constructor method.

package com.journaldev.access;

public class SuperClass {

public SuperClass(){

}

public SuperClass(int i){}

public void test(){

System.out.println("super class test method");

}

}

Copy

Use of super keyword can be seen in below child class implementation.

package com.journaldev.access;

public class ChildClass extends SuperClass {

public ChildClass(String str){

//access super class constructor with super keyword

super();

//access child class method

test();

//use super to access super class method

super.test();

}

@Override

public void test(){

System.out.println("child class test method");

}

}

Copy

### 49. What is break and continue statement?

We can **use break statement to terminate** **for, while, or do-while loop**. We can use a break statement in the switch statement to **exit** **the** switch case. You can see the example of break statement at [java break](https://www.digitalocean.com/community/tutorials/java-switch-case-string). We can use a break with the label to terminate the nested loops.

The continue **statement skips the current** iteration of a for, while, or do-while loop. We can use the continue statement with **the label to skip the current iteration of the outermost loop.**

### 50. What is this keyword in Java?

**this** keyword provides the **reference to the current object and** it’s mostly **used to make sure that object variables** are used, not **the** **local** **variables having the same name.**

//constructor

public Point(int x, int y) {

this.x = x;

this.y = y;

}

Copy

We can also use this keyword to invoke other constructors from a constructor.

public Rectangle() {

this(0, 0, 0, 0);

}

public Rectangle(int width, int height) {

this(0, 0, width, height);

}

public Rectangle(int x, int y, int width, int height) {

this.x = x;

this.y = y;

this.width = width;

this.height = height;

}

Copy

### 51. What is default constructor?

**No argument constructor of** a class is known as default constructor. When we **don’t define any constructor for** the class, java compiler automatically creates the default **no-args constructor** for the class. If there are **other constructors defined**, then compiler **won’t** **create** **default constructor for us.**

### 52. Can we have try without catch block?

**Yes**, we can have **try-finally statement** and hence avoiding catch block.

### 53. What is Garbage Collection?

Garbage Collection is **the process of looking at heap memory**, identifying which **objects are in use and which are not**, and **deleting the unused objects**. In Java, the **process of deallocating memory is handled automatically by the garbage collector**.

We can run the garbage collector with code **Runtime.getRuntime().gc()** or use utility **method System.gc().** For a detailed analysis of Heap Memory and Garbage Collection, please read [Java Garbage Collection](https://www.digitalocean.com/community/tutorials/java-jvm-memory-model-memory-management-in-java).

### 54. What is Serialization and Deserialization?

We can **convert a Java object to a Stream that is called Serialization**. Once an object is converted to Stream, **it can be saved to file or send over the network or used in socket connections.**

The object should implement a **Serializable** **interface** and we can **use java.io.ObjectOutputStream** to **write objects to file** or to any OutputStream object. Read more at [Java Serialization](https://www.digitalocean.com/community/tutorials/objectoutputstream-java-write-object-file).

The process of **converting stream data created through serialization to Object is called deserialization**. Read more at [Java Deserialization](https://www.digitalocean.com/community/tutorials/objectinputstream-java-read-object-file).

### 55. How to run a JAR file through command prompt?

We can run a jar file using java command but it **requires Main-Class entry** in jar manifest file. Main-Class is the entry point of the jar and used by java command to execute the class. Learn more at [java jar file](https://www.digitalocean.com/community/tutorials/run-jar-file).

### 56. What is the use of System class?

Java System Class is one of **the core classes**. One of the easiest ways to log information for debugging is **System.out.print()** method.

System class **is final** so that we **can’t subclass** and **override its behaviour** through **inheritance**. **System class doesn’t provide any public constructors,** so **we can’t instantiate this class** and that’s **why all of its methods are static**.

Some of the utility methods of System class are for **array copy, get the current time, reading environment variables**. Read more at [Java System Class](https://www.digitalocean.com/community/tutorials/system-class-java-properties-out-println).

### 57. What is instanceof keyword?

We can use the **instanceof** keyword to check if an **object belongs to a class or not**. We should avoid it’s usage as much as possible. Sample usage is:

public static void main(String args[]){

Object str = new String("abc");

if(str instanceof String){

System.out.println("String value:"+str);

}

if(str instanceof Integer){

System.out.println("Integer value:"+str);

}

}

Copy

Since str is of type String at runtime, first if statement evaluates to the true and second one as false.

### 58. Can we use String with switch case?

One of **the Java 7 feature** was improvement of switch case of allow Strings. So if you are **using Java 7 or higher version**, you can use String in **switch-case statements**. Read more at [Java switch-case String example](https://www.digitalocean.com/community/tutorials/java-switch-case-string).

### 59. Java is Pass by Value or Pass by Reference?//check other answers over internet. This maybe wrong

This is a very confusing question, we know that object **variables contain the reference** to the **Objects in heap** space. When we invoke any method, **a copy of these variables is passed and gets stored in the stack memory of the method**. We can test any language whether it’s pass by reference or pass by value through a simple generic swap method, to learn more read [Java is Pass by Value and Not Pass by Reference](https://www.digitalocean.com/community/tutorials/java-is-pass-by-value-and-not-pass-by-reference).

### 60. What is difference between Heap and Stack Memory?

Major difference between **Heap** and **Stack** memory are as follows:

* Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
* Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
* Memory management in the stack is done in a LIFO manner whereas it’s more complex in Heap memory because it’s used globally.

For a detailed explanation with a sample program, read [Java Heap vs Stack Memory](https://www.digitalocean.com/community/tutorials/java-heap-space-vs-stack-memory).

### 61. Java Compiler is stored in JDK, JRE or JVM?

The task of java compiler is to convert java program into bytecode, we have javac executable for that. So it must be stored in JDK, we don’t need it in JRE and JVM is just the specs.

### 62. What will be the output of following programs?

**Context: static method in class**

package com.journaldev.util;

public class Test {

public static String toString(){

System.out.println("Test toString called");

return "";

}

public static void main(String args[]){

System.out.println(toString());

}

}

Copy

**Answer**: The code won’t compile because we **can’t have an Object class method** with the **static** keyword. Note that the Object class has **toString**() method. You will get a compile-**time error as “This static method cannot hide the instance method from Object”.** The reason is that the **static method belongs to the class and** since **every class base is an Object, we can’t** have the same method in the **instance as well as in class.** You won’t get this error if you change the method name from toString() to something else that is not present in the superclass **Object**.

**Context: static method invocation**

package com.journaldev.util;

public class Test {

public static String foo(){

System.out.println("Test foo called");

return "";

}

public static void main(String args[]){

Test obj = null;

System.out.println(obj.foo());

}

}

Copy

**Answer**: Well this is a strange situation. We all have seen **NullPointerException** when we invoke a method on the object that is NULL. But here this program will work and prints “Test foo called”.

The reason for this is the **java compiler code optimization.** Wh**en the java code is compiled to produced byte code, it figures out that foo() is a static method and should be called using class**. So **it changes the method call obj.foo() to Test.foo() and hence** **no NullPointerException.**

# JAVA 8 INTERFACE :

java 8 interface changes include static methods and default methods in interfaces. Prior to Java 8, we could have only method declarations in the interfaces. But from Java 8, we can have **default methods** and **static methods** in the interfaces

Designing interfaces have always been a tough job because if we want to add additional methods in the interfaces, it will require change in all the implementing classes. As interface grows old, the number of classes implementing it might grow to an extent that it’s not possible to extend interfaces. That’s why when designing an application, most of the frameworks provide a base implementation class and then we extend it and override methods that are applicable for our application. Let’s look into the default interface methods and static interface methods and the reasoning of their introduction in Java 8 interface changes.

## **Java Interface Default Method**

For creating a default method in java interface, we need to use “**default**” keyword with the method signature. For example,

package com.journaldev.java8.defaultmethod;

public interface Interface1 {

void method1(String str);

default void log(String str){

System.out.println("I1 logging::"+str);

}

}

Notice that log(String str) is the default method in the Interface1. Now when a class will implement Interface1, it is not mandatory to provide implementation for default methods of interface. This feature will help us in extending interfaces with additional methods, all we need is to provide a default implementation. Let’s say we have another interface with following methods:

package com.journaldev.java8.defaultmethod;

public interface Interface2 {

void method2();

default void log(String str){

System.out.println("I2 logging::"+str);

}

}

We know that Java doesn’t allow us to extend multiple classes because it will result in the “Diamond Problem” where compiler can’t decide which superclass method to use. With the default methods, the diamond problem would arise for interfaces too. Because if a class is implementing both Interface1 and Interface2 and doesn’t implement the common default method, compiler can’t decide which one to chose. Extending multiple interfaces are an integral part of Java, you will find it in the core java classes as well as in most of the enterprise application and frameworks. So to make sure, this problem won’t occur in interfaces, it’s made mandatory to provide implementation for common default methods of interfaces. So if a class is implementing both the above interfaces, it will have to provide implementation for log() method otherwise compiler will throw compile time error. A simple class that is implementing both Interface1 and Interface2 will be:

package com.journaldev.java8.defaultmethod;

public class MyClass implements Interface1, Interface2 {

@Override

public void method2() {

}

@Override

public void method1(String str) {

}

@Override

public void log(String str){

System.out.println("MyClass logging::"+str);

Interface1.print("abc");

}

}

Important points about java interface default methods:

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Java interface default methods has bridge down the differences between interfaces and abstract classes.
3. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.
5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.
7. Java interface default methods are also referred to **as Defender Methods** or **Virtual extension methods.**

## **Java Interface Static Method**

Java interface static method is similar to default method except that we **can’t override them in the implementation** classes. This feature helps us in **avoiding undesired results incase of poor implementation in implementation** classes. Let’s look into this with a simple example.

package com.journaldev.java8.staticmethod;

public interface MyData {

default void print(String str) {

if (!isNull(str))

System.out.println("MyData Print::" + str);

}

static boolean isNull(String str) {

System.out.println("Interface Null Check");

return str == null ? true : "".equals(str) ? true : false;

}

}

Now let’s see an implementation class that is having isNull() method with poor implementation.

package com.journaldev.java8.staticmethod;

public class MyDataImpl implements MyData {

public boolean isNull(String str) {

System.out.println("Impl Null Check");

return str == null ? true : false;

}

public static void main(String args[]){

MyDataImpl obj = new MyDataImpl();

obj.print("");

obj.isNull("abc");

}

}

Note that isNull(String str) is a simple class method, it’s not overriding the interface method. For example, if we will add [@Override annotation](https://www.digitalocean.com/community/tutorials/java-override-method-overriding) to the isNull() method, it will result in compiler error. Now when we will run the application, we get following output.

Interface Null Check

Impl Null Check

If we make the interface method from static to default, we will get following output.

Impl Null Check

MyData Print::

Impl Null Check

Java interface static method is **visible to interface methods only**, if we remove the isNull() method from the MyDataImpl class, we won’t be able to use it for the MyDataImpl object. However like other static methods, **we can use interface static methods using class name**. For example, a valid statement will be:

boolean result = MyData.isNull("abc");

Important points about java interface static method:

1. Java interface static method is part of interface, we **can’t** **use it for implementation class objects.**
2. Java interface static methods are good for providing utility methods, for **example null check, collection sorting** etc.
3. Java interface static method helps us in providing **security** by not allowing implementation classes to override them.
4. We **can’t** define **interface static method for Object class methods**, we will get compiler error as “**This static method cannot hide the instance method from Objec**t”. This is because it’s not allowed in java, since Object is the base class for all the classes and we can’t have one class level static method and another instance method with same signature.
5. We can use java interface static methods to remove **utility** classes such as Collections and move all of it’s static methods to the corresponding interface, that would be easy to find and use.

### Java Functional Interfaces

Before I conclude the post, I would like to provide a brief introduction to Functional interfaces. An interface with exactly one abstract method is known as Functional Interface. A new annotation [@FunctionalInterface](https://www.digitalocean.com/community/users/functionalinterface) has been introduced to mark an interface as Functional Interface. [@FunctionalInterface](https://www.digitalocean.com/community/users/functionalinterface) annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. It’s optional but good practice to use it. Functional interfaces are long awaited and much sought out feature of Java 8 because it enables us to use **lambda expressions** to instantiate them. A new package java.util.function with bunch of functional interfaces are added to provide target types for lambda expressions and method references.