

## Classes and Objects

Class in Java is a template or a blueprint for creating Objects, and it defines the attributes and behaviors of Objects of a certain type. On the other hand, an Object is an Instance of a Class, representing a real-world entity with its behavior and state.

### **Object Definitions:**

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

### **Initialize object in Java:**

1. By reference variable
2. By method
3. By constructor

## Message Passing

in OOP is a mechanism for objects to communicate and interact with each other by sending messages. It involves invoking methods on objects, which can lead to the exchange of information, execution of a specific behavior, or modification of an object's state.

Message passing involves sending messages from one object to another, triggering a specific behavior or action. These messages are typically in the form of method invocations, where one object invokes a method on another object to request an operation or exchange information. This communication model enables objects to collaborate and work together to accomplish tasks.

## Getters and Setters

The concept of getters and setters supports the concept of data hiding. Because other objects should not directly manipulate data within another object, the getters and setters provide

controlled access to an object's data. Getters and setters are sometimes called accessor methods and Mutator methods, respectively.

### **What exactly is a class?**

- A class is a blueprint for an object.
- When you instantiate an object, you use a class as the basis for how the object is built.
- An object cannot be instantiated without a class. A class can be thought of as a sort of higher-level data type.
- A class defines the attributes and behaviors that all objects created with this class will possess. Classes are pieces of code.

### **Encapsulation and data hiding**

Encapsulation is defined by the fact that objects contain both the attributes and behaviors.

Data hiding is a major part of encapsulation.

For data hiding to work, all attributes should be declared as private. Thus, attributes are never part of the interface. Only the public methods are part of the class interface. Declaring an attribute as public breaks the concept of data hiding.

### **Composition**

It is natural to think of objects as containing other objects. A television set contains a tuner and video display. A computer contains video cards, keyboards, and drives. Although the computer can be considered an object unto itself, the drive is also considered a valid object. In fact, you could open up the computer and remove the drive and hold it in your hand. Both the computer and the drive are considered objects. It is just that the computer contains other objects—such as drives. In this way, objects are often built, or composed, from other objects. this is composition.

### **Encapsulation**

Encapsulating the data and behavior into a single object is of primary importance in OO development. A single object contains both its data and behaviors and can hide what it wants from other objects.

### **Inheritance**

A class can inherit from another class and take advantage of the attributes and methods defined by the superclass.

### **Polymorphism**

Polymorphism means that similar objects can respond to the same message in different ways. For example, you might have a system with many shapes. However, a circle, a square, and a

star are each drawn differently. Using polymorphism, you can send each of these shapes the same message (for example, Draw), and each shape is responsible for drawing itself.

### **Composition**

Composition means that an object is built from other objects.

### **Constructors**

- In OO languages, constructors are methods that share the same name as the class and have no return type.
- The most important function of a constructor is to initialize the memory allocated when the new keyword is encountered.
- If the class provides no explicit constructor, a default constructor will be provided.
- Overloading allows a programmer to use the same method name over and over, as long as the signature of the method is different each time.
- The first thing that happens inside the constructor is that the constructor of the class's superclass is called.
- If there is no explicit call to the superclass constructor, the default is called automatically. Then each class attribute of the object is initialized. These are the attributes that are part of the class definition (instance variables), not the attributes inside the constructor or any other method (local variables). Then the rest of the code in the constructor executes.

Scope:

There are three types of attributes:

- Local attributes
- Object attributes
- Class attributes

#### **Local attributes:**

Local attributes are owned by a specific method.

```
public class Number {  
    public method1() {  
        int count;  
    }  
    public method2() {
```

```
}
```

```
}
```

The method method1 contains a local variable called count. This integer is accessible only inside method1. The method method2 has no idea that the integer count even exists.

### **Object attribute:**

There are many design situations in which an attribute must be shared by several methods within the same object.

```
public class Number {  
    int count; // available to both method1 and method2  
  
    public method1() {  
        count = 1;  
    }  
  
    public method2() {  
        count = 2;  
    }  
}
```

In this case, the class attribute count is declared outside the scope of both method1 and method2. However, it is within the scope of the class. Thus, count is available to both method1 and method2.

### **Class attributes:**

it is possible for two or more objects to share attributes.

```
public class Number {  
    static int count;  
  
    public method1() {  
    }  
}
```

}

By declaring count as static, this attribute is allocated a single piece of memory for all objects instantiated from the class. Thus, all objects of the class use the same memory location for count. Essentially, each class has a single copy, which is shared by all objects of that class.

**Question: Do you think putting data and method for that data brings any improvement in designing a solution? Write exactly three points.**

1. **Encapsulation:** Bundling data and methods together promotes encapsulation, making it easier to manage and protect data integrity by controlling access to the data and operations that manipulate it.
2. **Modularity:** Grouping related data and methods improves modularity, allowing for clearer organization and easier maintenance of code. This facilitates code reuse and simplifies debugging and testing.
3. **Abstraction:** Integrating data and methods supports abstraction, enabling developers to focus on essential functionalities without needing to know the underlying implementation details. This promotes higher-level thinking and enhances code readability and maintainability.

### **Encapsulation versus abstraction**

- Encapsulation wraps code and data for unnecessary information, while Abstraction presents only useful data.
- Encapsulation focuses on how it should get done, while Abstraction focuses on what should get done.
- The Abstraction technique hides complexity by providing a more abstract picture, and the Encapsulation technique hides work within the system so it can get changed in the future.
- Abstraction allows you to separate the program into many independent parts, while Encapsulation enables you to change it as per your requirement.
- Abstraction addresses design problems, while Encapsulation addresses implementation problems.
- Abstraction hides unnecessary details in code, while Encapsulation makes it easier for developers to organize the entire program.
- Encapsulation focuses on how it should get done, while Abstraction focuses on what should get done.

- The Abstraction technique hides complexity by providing a more abstract picture, and the Encapsulation technique hides work within the system so it can get changed in the future.
- Abstraction allows you to separate the program into many independent parts, while Encapsulation enables you to change it as per your requirement.
- Abstraction addresses design problems, while Encapsulation addresses implementation problems.
- Abstraction hides unnecessary details in code, while Encapsulation makes it easier for developers to organize the entire program.

#### **Example of copy constructor**

```
public class Person {  
    private String name;  
    private int age;  
  
    // Constructor  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // Copy Constructor  
    public Person(Person original) {  
        this.name = original.name;  
        this.age = original.age;  
    }  
  
    // Getter methods  
    public String getName() {
```

```
        return name;
    }

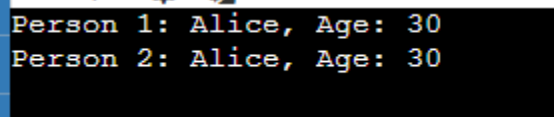
    public int getAge() {
        return age;
    }

    public static void main(String[] args) {
        // Creating an object using the constructor
        Person person1 = new Person("Alice", 30);

        // Creating a copy of the object using the copy constructor
        Person person2 = new Person(person1);

        // Displaying the details of both objects
        System.out.println("Person 1: " + person1.getName() + ", Age: " +
            person1.getAge());

        System.out.println("Person 2: " + person2.getName() + ", Age: " +
            person2.getAge());
    }
}
```



```
Person 1: Alice, Age: 30
Person 2: Alice, Age: 30
```

**Private and static method difference :**

In Java, private variables are instance variables that can only be accessed within the same class in which they are declared. They are not visible or accessible from outside the class.

Static variables, however, are class variables shared among all class instances. They are also known as class variables. A single copy of a static variable is created and shared among all class objects, regardless of how many objects are created from the class.

## **Collections framework**

Java collection provides classes and interfaces for us to be able to write code quickly and efficiently.

### **Why do we need collections?**

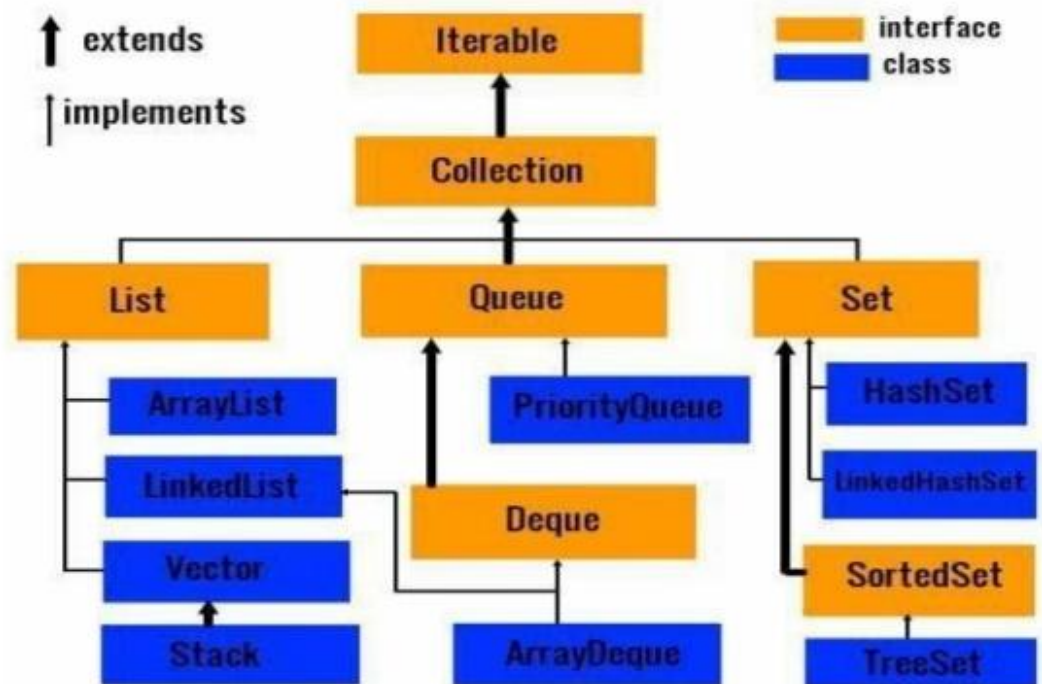
We need collections for efficient storage and better manipulation of data in java.

Few commonly used collections are :

- ArrayList - for variable size collection
- Set – for distinct collection
- Stack – a LIFO structure
- Hashmap – for storing key values



## Hierarchy of Collection Framework



### Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about the Java ArrayList class are:

1. Java ArrayList class can contain duplicate elements.
2. Java ArrayList class maintains insertion order.
3. Java ArrayList class is non-synchronized.
4. Java ArrayList allows random access because the array works on an index basis.
5. In the Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.

## Constructors of Java ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

## Object Cloning in Java

Object **cloning** is a way to create an exact copy of an object. The clone() method of the Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

1. **protected** Object clone() **throws** CloneNotSupportedException

### Why use clone() method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

## Advantage of Object cloning

- You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
- It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
- Clone() is the fastest way to copy array.

## Disadvantage of Object cloning

- To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
- We have to implement a cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.

Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.

Object.clone() doesn't invoke any constructor so we don't have any control over object construction.

- If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

Example:

```
class Student18 implements Cloneable{
    int rollNo;
    String name;

    Student18(int rollNo,String name){
        this.rollNo=rollNo;
```

```

    this.name=name;
}

public Object clone()throws CloneNotSupportedException{
    return super.clone();
}

public static void main(String args[]){
    try{
        Student18 s1=new Student18(101,"amit");

        Student18 s2=(Student18)s1.clone();

        System.out.println(s1.rollNo+" "+s1.name);
        System.out.println(s2.rollNo+" "+s2.name);

    }catch(CloneNotSupportedException c){}

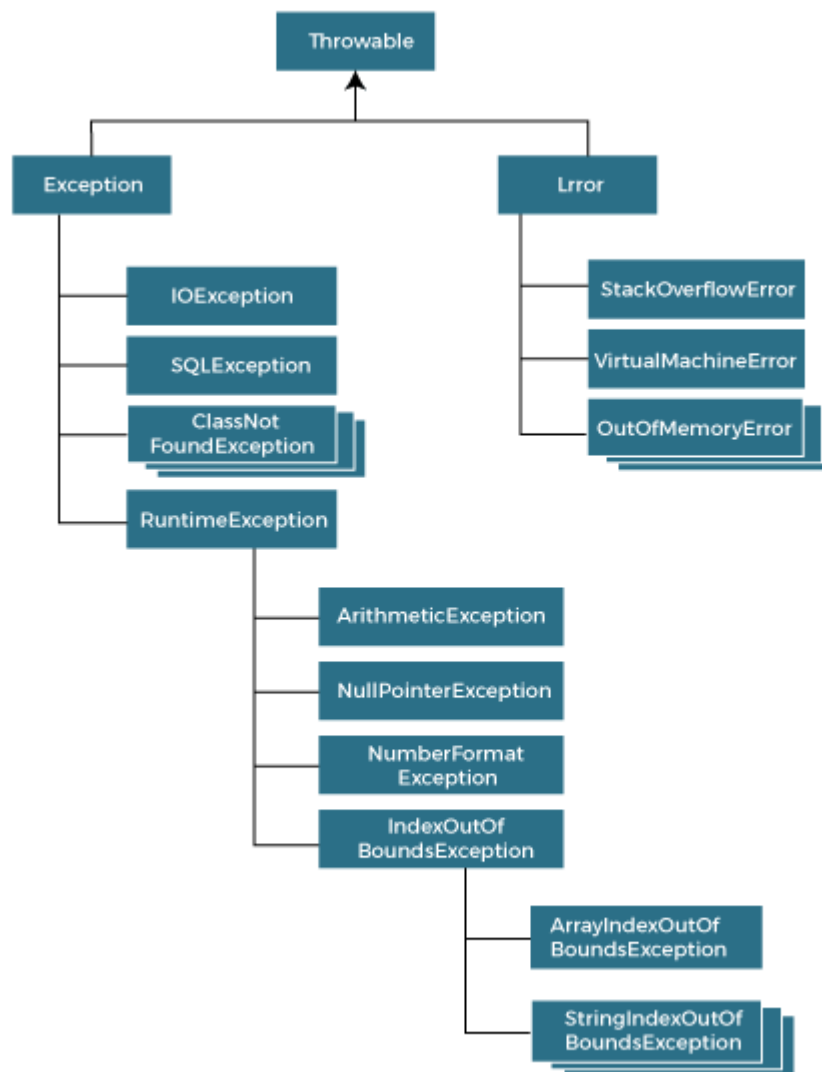
}
}

```

## Exception Handling in Java

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.





### 1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

## Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

## Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

## Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.

- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

## Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing a connection, etc.
- The important statements to be printed can be placed in the finally block.

**Rule:** For each try block there can be zero or more catch blocks, but only one finally block.

there are a few rare scenarios where the `finally` block may not execute:

1. **JVM Termination:** If the JVM terminates while the `try` or `catch` block is executing (for example, by calling `System.exit(int)`), the `finally` block will not execute.

```
try {  
    System.exit(0);  
} finally {  
    System.out.println("This will not be printed.");  
}
```

2. **Infinite Loop or Endless Wait in the Try Block:** If the code in the `try` block enters an infinite loop or the thread gets stuck indefinitely (e.g., waiting for an external resource that never becomes available), the `finally` block may never be reached.

```
try {  
    while (true) {  
        // Infinite loop  
    }  
} finally {  
    System.out.println("This will not be printed.");  
}
```



## Java throw keyword

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

If we throw a checked exception using the throw keyword, it is a must to handle the exception using catch block or the method must declare it using the throws declaration.

## Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives information to the programmer that there may be an exception. So, the programmer should provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If any unchecked exception occurs such as NullPointerException, it is programmer's fault that he is not checking the code before it is used.

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used to throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.

4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare an unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, the subclass overridden method can declare the same, subclass exception or no exception but cannot declare parent exception.

Feature	Error Handling	Exception Handling
Mechanism	Return codes, flags	Try-catch blocks, throwing exceptions
Complexity	Can lead to scattered code	More structured and centralized
Readability	Manual checks, less readable	Clear separation of logic and error handling
Maintenance	Harder to maintain	Easier to maintain
Propagation	Manual propagation	Automatic propagation
Granularity	Limited granularity	Fine-grained control

## Techniques for Error Handling in Procedural Programming

### 1. Return Codes:

- Functions return specific values to indicate success or failure.
- The calling code must check these return values and handle errors appropriately.

## 2. **Global Error Variables:**

- Use a global variable to store the error status.
- Functions set this global variable when an error occurs.

## 3. **Error Flags:**

- Use flag variables to indicate the occurrence of an error.
- Set these flags within functions and check them in the main logic.

## 4. **special Return Values:**

- Functions return a special value (e.g., `NULL`, `-1`) to indicate an error.

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int* allocate_array(int size) {  
    if (size <= 0) {  
        return NULL; // Indicate an error  
    }  
    return (int*)malloc(size * sizeof(int));  
}
```

```
int main() {  
    int *array = allocate_array(-5);  
    if (array == NULL) {  
        printf("Error: Invalid array size\n");  
    } else {  
        // Use the array  
    }  
}
```

```
        free(array);  
    }  
    return 0;  
}
```

## For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

```
//Java Program to print the array elements using for-each loop  
class Testarray1{  
    public static void main(String args[]){  
        int arr[]={3,3,4,5};  
        //printing array using for-each loop  
        for(int i:arr)  
            System.out.println(i);  
    }  
}
```

```
//Java Program to demonstrate the way of passing an array
//to method.
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
if(min>arr[i])
min=arr[i];

System.out.println(min);
}

public static void main(String args[]){
int a[]={3,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}}
```

```

//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};

//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3]; //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++){
{
c[i][j]+=a[i][k]*b[k][j];
}
//end of k loop
System.out.print(c[i][j]+ " "); //printing matrix element
}
//end of j loop
System.out.println();//new line
}
}}

```

## Interfaces Changes In Java 8

The Java 8 release introduces or allows us to have static and default methods in the interfaces. Using default methods in an interface, the developers can add more methods to the interfaces. This way they do not disturb or change the classes that implement the interface.

Java 8 also allows the interface to have a static method. Static methods are the same as those we define in classes. Note that the static method cannot be overridden by the class that implements the interface.

The introduction of static and default methods in the interface made it easier to alter the interfaces without any problems and also made interfaces easier to implement.

## Static Method In Interface In Java

Interfaces can also have methods that can have definitions. These are the static methods in the interface. The static methods are defined inside the interface and they cannot be overridden or changed by the classes that implement this interface.

```
//interface declaration
interface TestInterface {
    // static method definition
    static void static_print() {
        System.out.println("TestInterface::static_print ()");
    }
    // abstract method declaration
    void nonStaticMethod(String str);
}

// Interface implementation
class TestClass implements TestInterface {
    // Override interface method
    @Override
    public void nonStaticMethod(String str) {
        System.out.println(str);
    }
}

public class Main{
    public static void main(String[] args) {
        TestClass classDemo = new TestClass();

        // Call static method from interface
        TestInterface.static_print();

        // Call overridden method using class object
        classDemo.nonStaticMethod("TestClass::nonStaticMethod ()");
    }
}
```

## Interface Default Method

As already mentioned, interfaces before Java 8 permitted only abstract methods. Then we would provide this method implementation in a separate class. If we had to add a new method to the interface, then we have to provide its implementation code in the same class.

Hence if we altered the interface by adding a method to it, the implementation class also would change.

This limitation was overcome by Java 8 version that allowed the interfaces to have default methods. The default methods in a way provide backward compatibility to the existing interfaces and we need not alter the implementation class. The default methods are also known as “virtual extension method” or “defender methods”.

Default methods are declared by using the keyword “default” in the declaration. The declaration is followed by the definition of the method. We can override the default method as it is available to the class that implements the interface.

In the same way, we can invoke it using the implementation class object from the interface directly without overriding it.

```
interface TestInterface {
    // abstract method
    public void cubeNumber(int num);

    // default method
    default void print()
    {
        System.out.println("TestInterface :: Default method");
    }
}

class TestClass implements TestInterface {
    // override cubeNumber method
    public void cubeNumber(int num)
    {
        System.out.println("Cube of given number " + num+ ":" + num*num*num);
    }
}

class Main{
    public static void main(String args[])
    {
        TestClass obj = new TestClass();
        obj.cubeNumber(5);

        // call default method print using class object
        obj.print();
    }
}
```



### *When to use an Interface:*

- Interfaces are mainly used when we have a small concise functionality to implement.
- When we are implementing APIs and we know they won't change for a while, then that time we go for interfaces.
- Interfaces allow us to implement multiple inheritance. Hence when we need to implement multiple inheritance in our application, we go for interfaces.
- When we have a wide range of objects, again interfaces are a better choice.
- Also when we have to provide a common functionality to many unrelated classes, still interfaces are used.