# Assertions in Java

An assertion allows testing the correctness of any assumptions that have been made in the program. An assertion is achieved using the **assert**statement in Java. While executing assertion, it is believed to be true. If it fails, JVM throws an error named **AssertionError.**It is mainly used for testing purposes during development.

The **assert**statement is used with a Boolean expression and can be written in **two different ways**.

**First way:**

*assert expression;*

**Second way:**

*assert expression1 : expression2;*

**Example:**

|  |
| --- |
| // Java program to demonstrate syntax of assertion  **import** java.util.Scanner;    **class** Test {  **public** **static** **void** main(String args[])      {  **int** value = 15;  **assert** value >= 20 : " Underweight";          System.out.println("value is " + value);      }  } |

**Output**

value is 15

#### After enabling assertions:

**Output:**

Exception in thread "main" java.lang.AssertionError: Underweight

**Enabling Assertions**

By default, assertions are disabled. We need to run the code as given. The syntax for enabling assertion statement in Java source code is:

**java –ea** Test

Or

**java –enableassertions** Test

Here, Test is the file name.

**Disabling Assertions**

The syntax for disabling assertions in java is:

**java –da** Test

Or

**java –disableassertions** Test

Here, Test is the file name.

**Why use Assertions**

Wherever a programmer wants to see if his/her assumptions are wrong or not.

* To make sure that an unreachable-looking code is actually unreachable.
* To make sure that assumptions written in comments are right.

if ((x & 1) == 1) {

}

else // x must be even

{

assert (x % 2 == 0);

}

* To make sure the default switch case is not reached.
* To check the object’s state.
* At the beginning of the method
* After method invocation.

**Assertion Vs Normal Exception Handling**

Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state a code expects before it starts running or the state after it finishes running. Unlike normal exception/error handling, assertions are generally disabled at run-time.

**Where to use Assertions**

* Arguments to private methods. Private arguments are provided by the developer’s code only and the developer may want to check his/her assumptions about arguments.
* Conditional cases.
* Conditions at the beginning of any method.

**Where not to use Assertions**

* Assertions should not be used to replace error messages
* Assertions should not be used to check arguments in the public methods as they may be provided by the user. Error handling should be used to handle errors provided by users.
* Assertions should not be used on command line arguments.

**Example:**

|  |
| --- |
| // Java program to demonstrate assertion in Java  **public** **class** Example {  **public** **static** **void** main(String[] args)      {  **int** age = 14;  **assert** age <= 18 : "Cannot Vote";          System.out.println("The voter's age is " + age);      }  } |

**Output**

The voter's age is 14

## ****Mutable String****

## ****What is a mutable string in java?****

**Immutable**means unchanging over time or unable to be changed. Whenever we create a string object of the **String class**, it is by default created immutable in nature. If we change the value of the string, the **JVM**creates a new object.

**Mutable** means changing over time or that can be changed. In a **mutable string**, we can change the value of the string and **JVM**doesn’t create a new object. In a **mutable string**, we can change the value of the string in the same object.  
To create a **mutable string in java**, Java has two classes **[StringBuffer](https://javagoal.com/string-in-java/" \l "14" \t "_blank)**and [**StringBuilder**](https://javagoal.com/string-in-java/#10)where the **String class** is used for the **immutable string**.

## ****How to create a mutable string in java?****

To create a **mutable string**, we can use **[StringBuffer](https://javagoal.com/string-in-java/" \l "14" \t "_blank)**and [**StringBuilder class**](https://javagoal.com/string-in-java/#10). Both classes create a **mutable object** of string but which one we should use totally depends on the scenario.

Suppose you want to work in a multithreading environment and the string should be thread-safe then you should use the **StringBuffer class.**On the other hand, if you don’t want a multithreading environment then you can use **StringBuilder** is not.  
But when you consider performance first then **StringBuilder**is better in terms of performance as compared to **StringBuffer**.

Let’s see how to create a string by **StringBuffer**and **StringBuilder**class.

public class MutableString

{

public static void main (String[] args)

{

StringBuffer str1 = new StringBuffer("JavaGoal");

StringBuilder str2 = new StringBuilder("Learning");

System.out.println("Value of str1 before change :" + str1);

System.out.println("Value of str2 before change :" + str2);

str1.append(".com");

str2.append(" website");

System.out.println("Value of str1 after change :" + str1);

System.out.println("Value of str2 after change :" + str2);

}

}

***Output:****Value of str1 before change :JavaGoal  
Value of str2 before change :Learning  
Value of str1 after change :JavaGoal.com  
Value of str2 after change :Learning website*

In the above example, we are creating two mutable strings one is from **StringBuffer**and another from **StringBuilder**. Here we are changing the value of string after creation. We can see we do not need to assign the value to assign in the same object, unlike immutable strings.

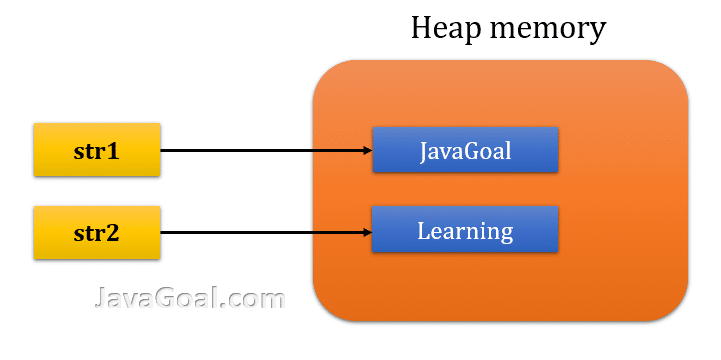
**How does mutable string work in memory?**

In the above example, you have seen how to create a mutable string. Let’s see how it is working in memory.

StringBuffer str1 = new StringBuffer("JavaGoal");

StringBuilder str2 = new StringBuilder("Learning");

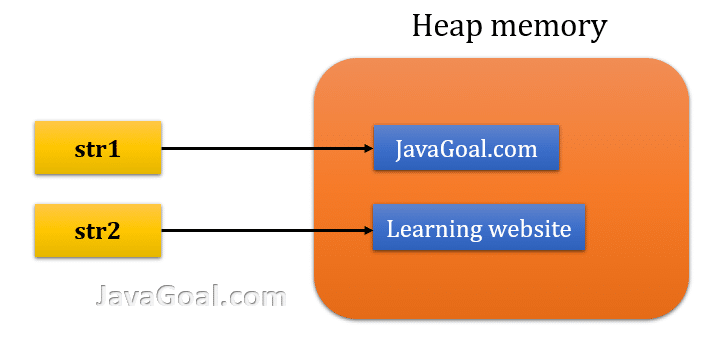
After execution of the above line, the **JVM** will create two objects in memory and return the reference to the variable.



str1.append(".com");

str2.append(" website");

After execution of the above line, the **JVM**will change the value in existing objects. It will not create new objects.



* **Package**

**Package** in [Java](https://www.geeksforgeeks.org/java/) is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

* Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
* Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
* Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
* Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.  
We can reuse existing classes from the packages as many time as we need it in our program.

**How packages work?**

Package names and directory structure are closely related. For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present inside *college*. Also, the directory *college* is accessible through [CLASSPATH](https://en.wikipedia.org/wiki/Classpath_(Java)) variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.

**Package naming conventions :**Packages are named in reverse order of domain names. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

**Adding a class to a Package :** We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.

**Subpackages:**Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.  
**Example :**

import java.util.\*;

**util** is a subpackage created inside **java** package.

**Accessing classes inside a package**

Consider following two statements :

// import the Vector class from util package.

import java.util.vector;

// import all the classes from util package

import java.util.\*;

* First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
* Second statement imports all the classes from **util** package.

// All the classes and interfaces of this package

// will be accessible but not subpackages.

import package.\*;

// Only mentioned class of this package will be accessible.

import package.classname;

// Class name is generally used when two packages have the same

// class name. For example in below code both packages have

// date class so using a fully qualified name to avoid conflict

import java.util.Date;

import my.package.Date;

// Java program to demonstrate accessing of members when

// corresponding classes are imported and not imported.

import java.util.Vector;

public class ImportDemo

{

public ImportDemo()

{

// java.util.Vector is imported, hence we are

// able to access directly in our code.

Vector newVector = new Vector();

// java.util.ArrayList is not imported, hence

// we were referring to it using the complete

// package.

java.util.ArrayList newList = new java.util.ArrayList();

}

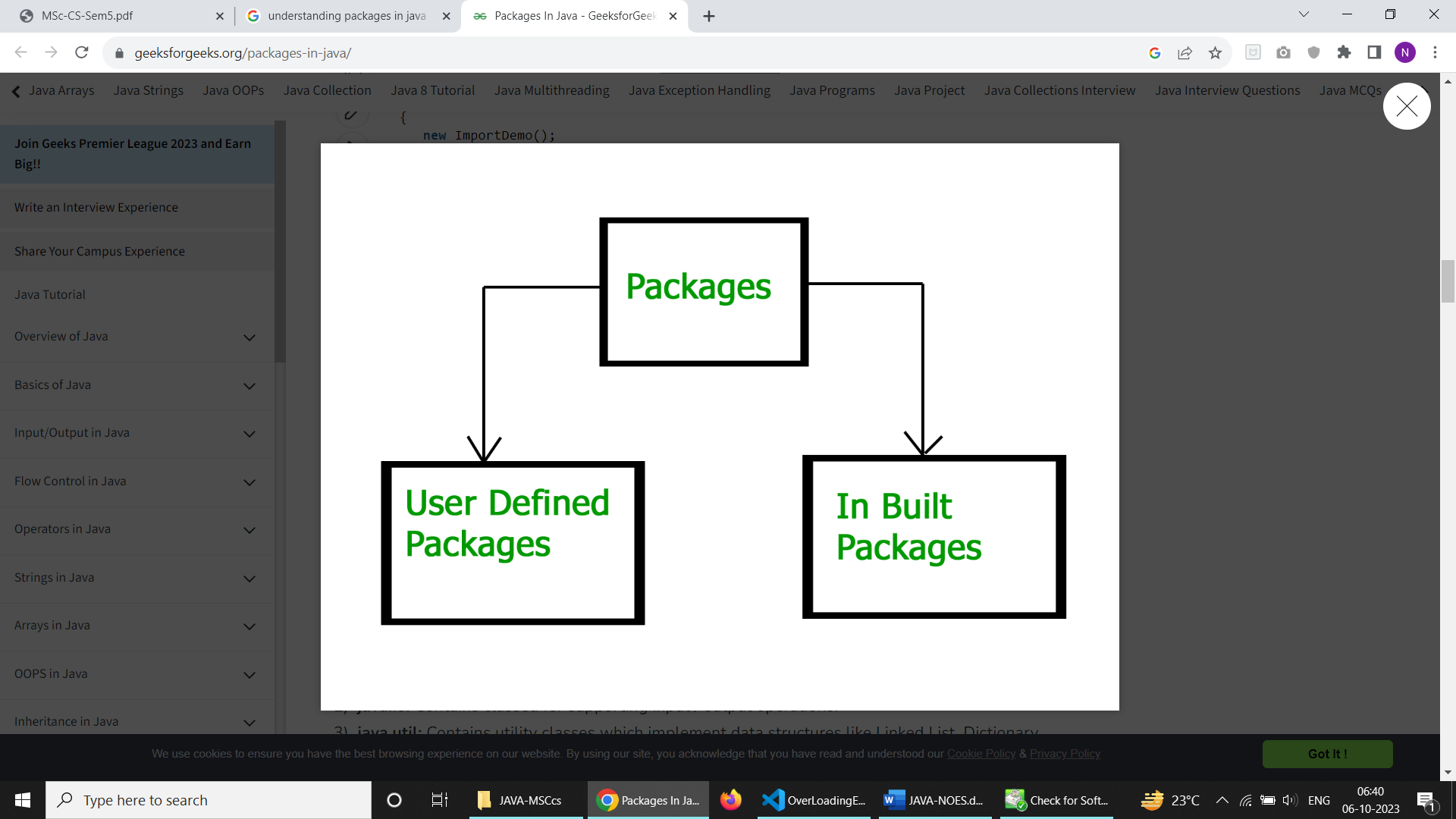
public static void main(String arg[])

{

new ImportDemo();

}

}



**Built-in Packages**  
These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:  
1) **java.lang:**Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.  
2) **java.io:**Contains classed for supporting input / output operations.  
3) **java.util:**Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.  
4) **java.applet:**Contains classes for creating Applets.  
5) **java.awt:**Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).  
6) **java.net:**Contain classes for supporting networking operations.

**User-defined packages**  
These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

// Name of the package must be same as the directory

// under which this file is saved

package myPackage;

public class MyClass

{

public void getNames(String s)

{

System.out.println(s);

}

}

/\* import 'MyClass' class from 'names' myPackage \*/

import myPackage.MyClass;

public class PrintName

{

public static void main(String args[])

{

// Initializing the String variable

// with a value

String name = "HelloThere:)";

// Creating an instance of class MyClass in

// the package.

MyClass obj = new MyClass();

obj.getNames(name);

}

}

**Note :** **MyClass.java** must be saved inside the **myPackage** directory since it is a part of the package.

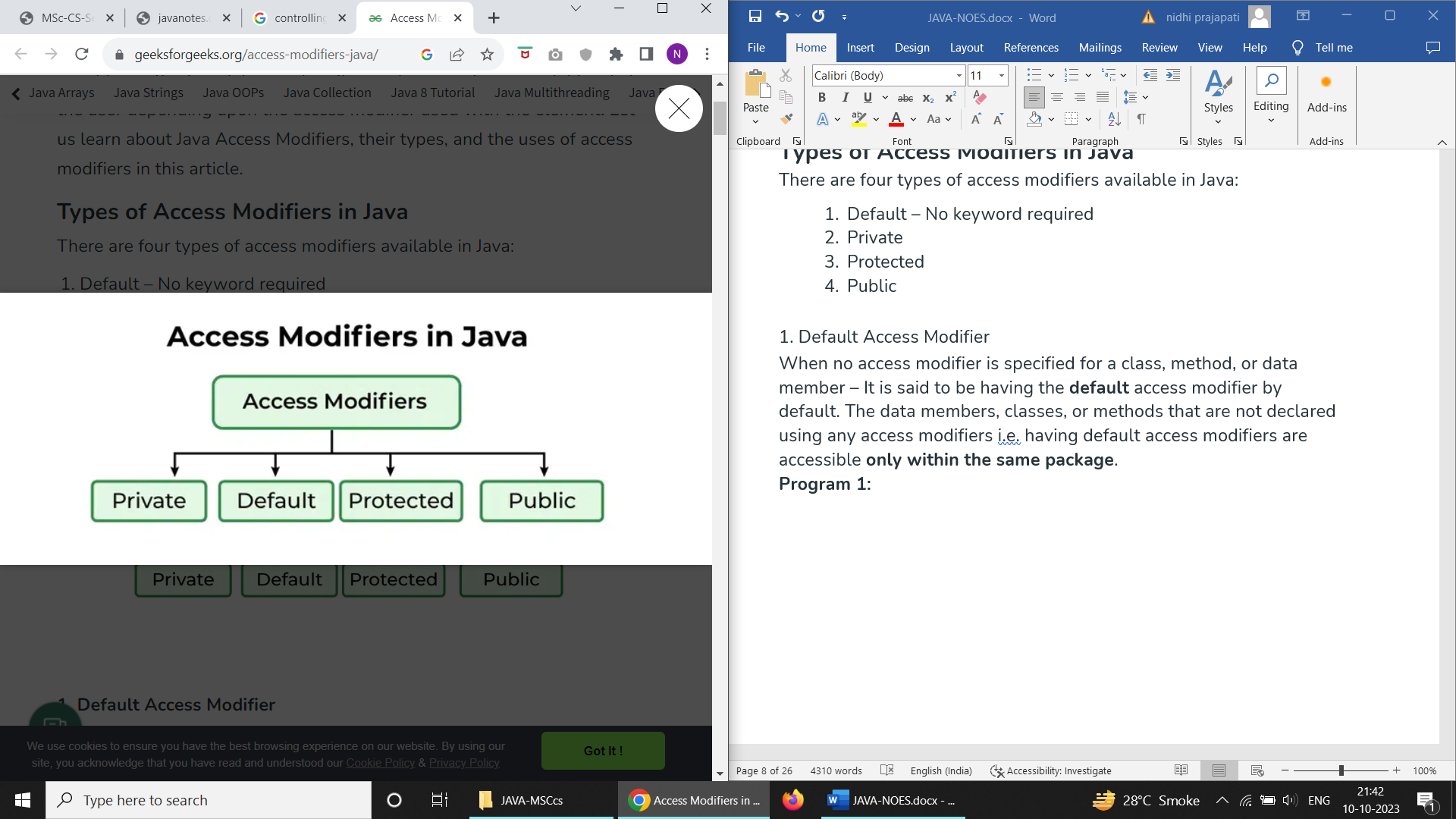
# Access Modifiers

in Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

## Types of Access Modifiers in Java

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public



### **1. Default Access Modifier**

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**.

**Program 1:**

// Java program to illustrate default modifier

**package** p1;

// Class Geek is having Default access modifier

**class** Geek

{

**void** display()

    {

        System.out.println("Hello World!");

    }

}

**Program 2:**

|  |
| --- |
| // default modifier  **package** p2;  **import** p1.\*;    // This class is having default access modifier  **class** GeekNew  {  **public** **static** **void** main(String args[])      {          // Accessing class Geek from package p1          Geek obj = **new** Geek();            obj.display();      }  } |

**Output:**

Compile time error

### **2. Private Access Modifier**

The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared.

* Any other **class of**the **same package will not be able to access** these members.
* Top-level classes or interfaces can not be declared as private because
  + private means “only visible within the enclosing class”.
  + protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

// private modifier

**package** p1;

**class** A

{

**private** **void** display()

    {

        System.out.println("GeeksforGeeks");

    }

}

**class** B

{

**public** **static** **void** main(String args[])

    {

        A obj = **new** A();

        // Trying to access private method

        // of another class

        obj.display();

    }

}

**Output:**

error: display() has private access in A

obj.display();

### **3. Protected Access Modifier**

The protected access modifier is specified using the keyword **protected**.

The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

**Program 1:**

// protected modifier

**package** p1;

// Class A

**public** **class** A

{

**protected** **void** display()

    {

        System.out.println("GeeksforGeeks");

    }

}

**Program 2:**

// protected modifier

**package** p2;

**import** p1.\*; // importing all classes in package p1

// Class B is subclass of A

**class** B **extends** A

{

**public** **static** **void** main(String args[])

{

    B obj = **new** B();

    obj.display();

}

}

**Output:**

GeeksforGeeks

### **Public Access modifier**

The public access modifier is specified using the keyword **public**.

* The public access modifier has the **widest scope** among all other access modifiers.
* Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

**Program 1:**

|  |
| --- |
| // public modifier  **package** p1;  **public** **class** A  {  **public** **void** display()      {          System.out.println("GeeksforGeeks");      }  } |

**Program 2:**

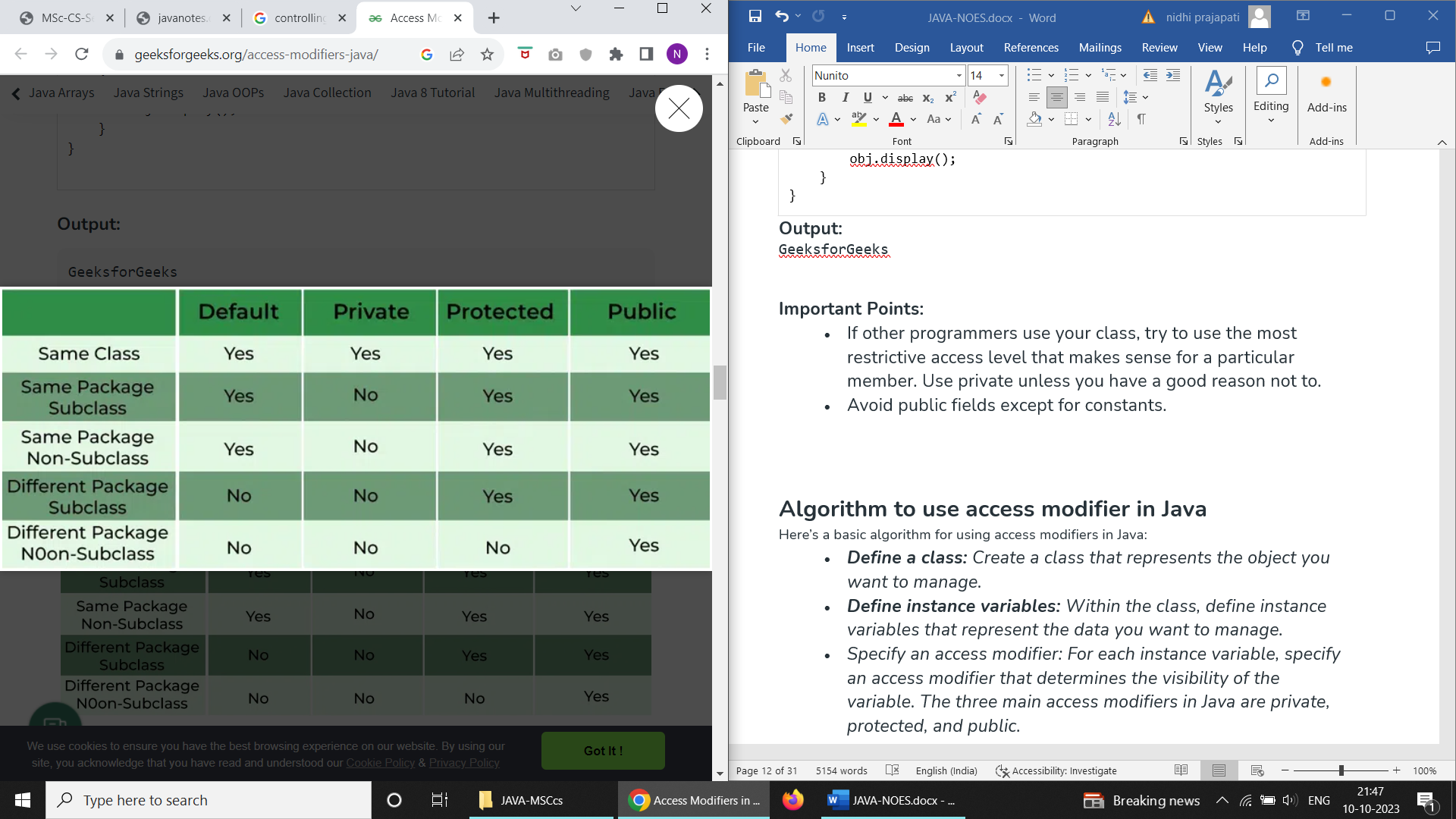
|  |
| --- |
| **package** p2;  **import** p1.\*;  **class** B {  **public** **static** **void** main(String args[])      {          A obj = **new** A();          obj.display();      }  } |

**Output:**

GeeksforGeeks

**Important Points:**

* If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
* Avoid public fields except for constants.



## Algorithm to use access modifier in Java

#### Here’s a basic algorithm for using access modifiers in Java:

* ***Define a class:****Create a class that represents the object you want to manage.*
* ***Define instance variables:****Within the class, define instance variables that represent the data you want to manage.*
* *Specify an access modifier: For each instance variable, specify an access modifier that determines the visibility of the variable. The three main access modifiers in Java are private, protected, and public.*
* ***Use private for variables that should only be accessible within the class:****If you want to prevent access to a variable from outside the class, use the private access modifier. This is the most restrictive access modifier and provides the greatest level of encapsulation.*
* ***Use protected for variables that should be accessible within the class and its subclasses****: If you want to allow access to a variable from within the class and its subclasses, use the protected access modifier. This is less restrictive than private and provides some level of inheritance.*
* ***Use public for variables that should be accessible from anywhere****: If you want to allow access to a variable from anywhere, use the public access modifier. This is the least restrictive access modifier and provides the least amount of encapsulation.*
* ***Use accessor and mutator methods to manage access to the variables:****In order to access and modify the variables, use accessor (getter) and mutator (setter) methods, even if the variables have a public access modifier. This provides a level of abstraction and makes your code more maintainable and testable.*
* ***In this exampl****e, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.*

# **Nested Classes in Java**

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 the below example, the class NestedClass does not exist independently of the class OuterClass.
* A nested class has access to the members, including private members, of the class in which it is nested. But the enclosing class does not have access to the member 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:
  1. **static nested class:**Nested classes that are declared static are called static nested classes.
  2. **inner class:**An inner class is a non-static nested class.

**Syntax:**

class OuterClass

{

...

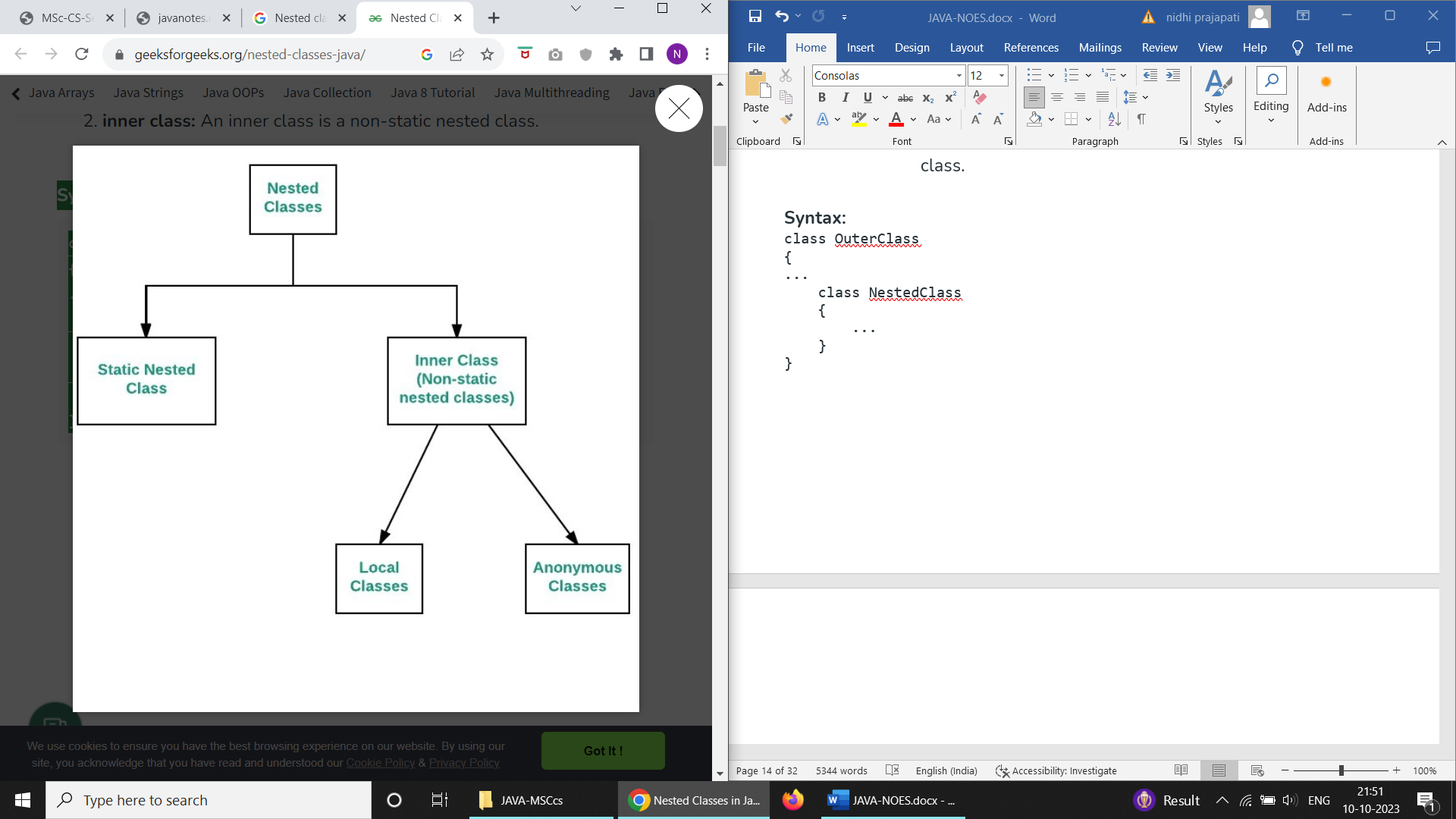
class NestedClass

{

...

}

}



## ****Static nested classes****

In the case of normal or regular inner classes, without an outer class object existing, there cannot be an inner class object. i.e., an object of the inner class is always strongly associated with an outer class object. But in the case of static nested class, Without an outer class object existing, there may be a static nested class object. i.e., an object of a static nested class is not strongly associated with the outer class object. As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference. They are accessed using the enclosing class name.

OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject =

new OuterClass.StaticNestedClass();

#### Below is the implementation of the above method:

|  |
| --- |
| // a static nested class    // outer class  **class** OuterClass {      // static member  **static** **int** outer\_x = 10;      // instance(non-static) member  **int** outer\_y = 20;      // private member  **private** **static** **int** outer\_private = 30;      // static nested class  **static** **class** StaticNestedClass {  **void** display()          {              // can access static member of outer class              System.out.println("outer\_x = " + outer\_x);              // can access private static member of              // outer class              System.out.println("outer\_private = "                                 + outer\_private);              // The following statement will give compilation              // error as static nested class cannot directly              // access non-static members              // System.out.println("outer\_y = " + outer\_y);                // Therefore create object of the outer class                // to access the non-static member                OuterClass out = **new** OuterClass();                System.out.println("outer\_y = " + out.outer\_y);              }      }  }  // Driver class  **public** **class** StaticNestedClassDemo {  **public** **static** **void** main(String[] args)      {          // accessing a static nested class          OuterClass.StaticNestedClass nestedObject              = **new** OuterClass.StaticNestedClass();          nestedObject.display();      }  } |

**Output**

outer\_x = 10

outer\_private = 30

outer\_y = 20

## ****Inner classes****

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

**There are two special kinds of inner classes :**

1. [Local inner classes](https://www.geeksforgeeks.org/local-inner-class-java/)
2. [Anonymous inner classes](https://www.geeksforgeeks.org/anonymous-inner-class-java/)

|  |
| --- |
| 1. // a inner class 3. // outer class 4. **class** OuterClass { 5. // static member 6. **static** **int** outer\_x = 10; 8. // instance(non-static) member 9. **int** outer\_y = 20; 11. // private member 12. **private** **int** outer\_private = 30; 14. // inner class 15. **class** InnerClass { 16. **void** display() 17. { 18. // can access static member of outer class 19. System.out.println("outer\_x = " + outer\_x); 21. // can also access non-static member of outer 22. // class 23. System.out.println("outer\_y = " + outer\_y); 25. // can also access a private member of the outer 26. // class 27. System.out.println("outer\_private = " 28. + outer\_private); 29. } 30. } 31. } 33. // Driver class 34. **public** **class** InnerClassDemo { 35. **public** **static** **void** main(String[] args) 36. { 37. // accessing an inner class 38. OuterClass outerObject = **new** OuterClass(); 40. OuterClass.InnerClass innerObject 41. = outerObject.**new** InnerClass(); 43. innerObject.display(); 44. } 45. } |

**Output**

outer\_x = 10

outer\_y = 20

outer\_private = 30

### **Comparison between a normal or regular class and a static nested class**

| **S.No.** | **Normal/Regular inner class** | **Static nested class** |
| --- | --- | --- |
| 1. | Without an outer class object existing, there cannot be an inner class object. That is, the inner class object is always associated with the outer class object. | Without an outer class object existing, there may be a static nested class object. That is, a static nested class object is not associated with the outer class object. |
| 2. | As the main() method can’t be declared, the regular inner class can’t be invoked directly from the command prompt. | As the main() method can be declared, the static nested class can be invoked directly from the command prompt. |
| 3. | Both static and non-static members of the outer class can be accessed directly. | Only a static member of an outer class can be accessed directly. |

# **finalize() Method**

The Java **finalize() method** of [Object class](https://www.geeksforgeeks.org/object-class-in-java/) is a method that the [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/) always calls just before the deletion/destroying the object which is eligible for Garbage Collection to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection, or we can say resource de-allocation. Remember, it is not a reserved keyword. Once the finalize() method completes immediately, Garbage Collector destroys that object.

**Finalization:**Just before destroying any object, the garbage collector always calls finalize() method to perform clean-up activities on that object. This process is known as Finalization in Java.

***Note:****The Garbage collector calls the finalize() method only once on any object.*

**Syntax:**

protected void finalize throws Throwable{}

Since the Object class contains the finalize method hence finalize method is available for every java class since Object is the superclass of all java classes. Since it is available for every java class, Garbage Collector can call the finalize() method on any java object.

**Why finalize() method is used?**

finalize() method releases system resources before the garbage collector runs for a specific object. JVM allows finalize() to be invoked only once per object.

**How to override finalize() method?**

The finalize method, which is present in the Object class, has an **empty implementation**. In our class, clean-up activities are there. Then we have to **override this method** to define our clean-up activities.

In order to [Override this method](https://www.geeksforgeeks.org/overriding-in-java/), we have to define and call finalize within our code explicitly.

// overriding of finalize() method

import java.lang.\*;

// Defining a class demo since every java class

// is a subclass of predefined Object class

// Therefore demo is a subclass of Object class

public class demo {

protected void finalize() throws Throwable

{

try {

System.out.println("inside demo's finalize()");

}

catch (Throwable e) {

throw e;

}

finally {

System.out.println("Calling finalize method"

+ " of the Object class");

// Calling finalize() of Object class

super.finalize();

}

}

// Driver code

public static void main(String[] args) throws Throwable

{

// Creating demo's object

demo d = new demo();

// Calling finalize of demo

d.finalize();

}

}

**Output:**

inside demo's finalize()

Calling finalize method of the Object class

# **Inheritance in Java**

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.

### **Why use inheritance in java**

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

### **Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

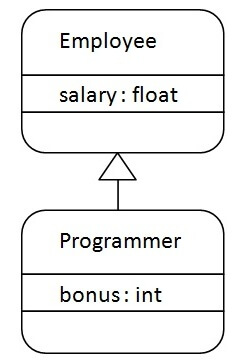
### **The syntax of Java Inheritance**

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

### **Java Inheritance Example**



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }

Programmer salary is:40000.0

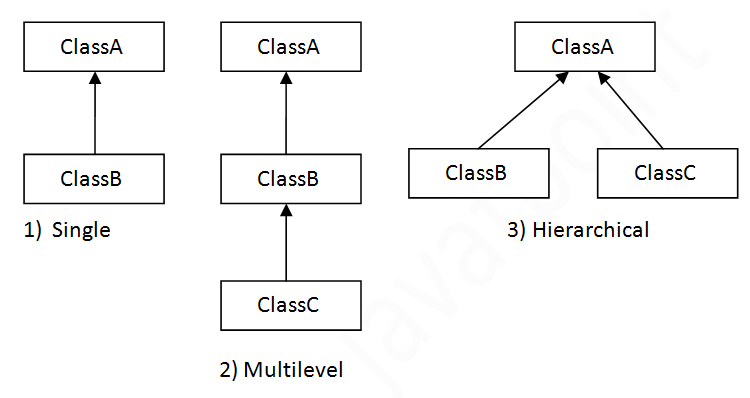
Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

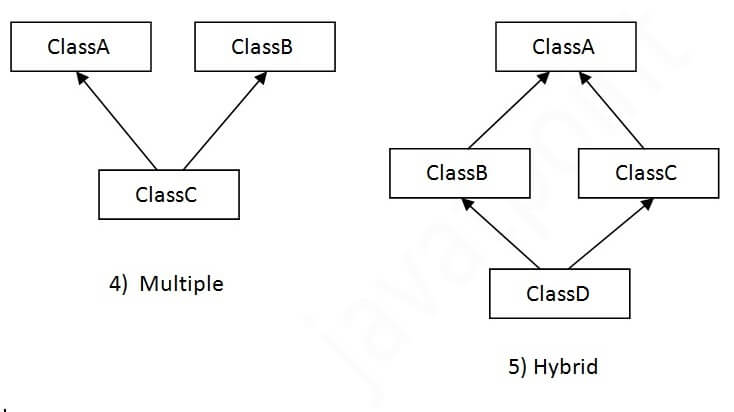
## Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



When one class inherits multiple classes, it is known as multiple inheritance.



Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

*File: TestInheritance.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** TestInheritance{
8. **public** **static** **void** main(String args[]){
9. Dog d=**new** Dog();
10. d.bark();
11. d.eat();
12. }}

Output:

barking...

eating...

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

*File: TestInheritance2.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

Output:

weeping...

barking...

eating...

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

*File: TestInheritance3.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** meow(){System.out.println("meowing...");}
9. }
10. **class** TestInheritance3{
11. **public** **static** **void** main(String args[]){
12. Cat c=**new** Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}

Output:

meowing...

eating...

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

1. **class** A{
2. **void** msg(){System.out.println("Hello");}
3. }
4. **class** B{
5. **void** msg(){System.out.println("Welcome");}
6. }
7. **class** C **extends** A,B{//suppose if it were
9. **public** **static** **void** main(String args[]){
10. C obj=**new** C();
11. obj.msg();//Now which msg() method would be invoked?
12. }
13. }

Compile Time Error

# Polymorphism in Java

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

**Real-life Illustration Polymorphism**: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

## What is Polymorphism in Java?

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

## ****Types of Java polymorphism****

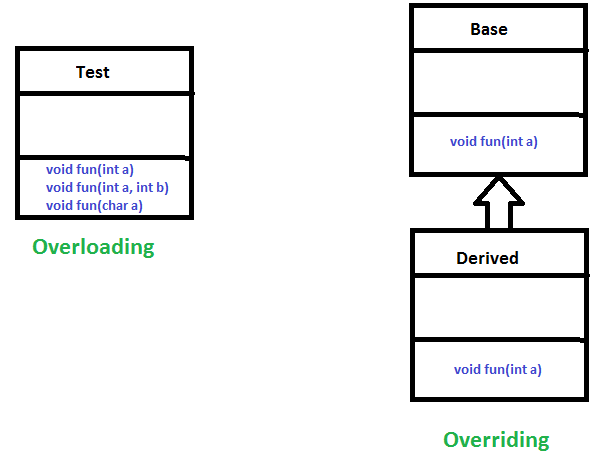
In Java polymorphism is mainly divided into two types:

* Compile-time Polymorphism
* Runtime Polymorphism

## Compile-Time Polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

***Note:****But Java doesn’t support the Operator Overloading.*



### **Method Overloading**

When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

**Example 1:**

* Java

|  |
| --- |
| // Java Program for Method overloading  // By using Different Types of Arguments    // Class 1  // Helper class  **class** Helper {        // Method with 2 integer parameters  **static** **int** Multiply(**int** a, **int** b)      {            // Returns product of integer numbers  **return** a \* b;      }        // Method 2      // With same name but with 2 double parameters  **static** **double** Multiply(**double** a, **double** b)      {            // Returns product of double numbers  **return** a \* b;      }  }    // Class 2  // Main class  **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {            // Calling method by passing          // input as in arguments          System.out.println(Helper.Multiply(2, 4));          System.out.println(Helper.Multiply(5.5, 6.3));      }  } |

**Output**

8

34.65

**Example 2:**

* Java

|  |
| --- |
| // Java program for Method Overloading  // by Using Different Numbers of Arguments    // Class 1  // Helper class  **class** Helper {      // Method 1      // Multiplication of 2 numbers  **static** **int** Multiply(**int** a, **int** b)      {  **return** a \* b; }      // Method 2  **static** **int** Multiply(**int** a, **int** b, **int** c)      {  **return** a \* b \* c; }  }  // Class 2  // Main class  **class** GFG {  **public** **static** **void** main(String[] args)      {          // Calling method by passing          // input as in arguments          System.out.println(Helper.Multiply(2, 4));          System.out.println(Helper.Multiply(2, 7, 3));      }  } |

**Output**

8

42

## ****Subtypes of Compile-time Polymorphism:****

### **i. Function Overloading**

It is a feature in C++ where multiple functions can have the same name but with different parameter lists. The compiler will decide which function to call based on the number and types of arguments passed to the function.

### **ii. Operator Overloading**

It is a feature in C++ where the operators such as +, -, \*, etc. can be given additional meanings when applied to user-defined data types.

### **iii. Template**

it is a powerful feature in C++ that allows us to write generic functions and classes. A template is a blueprint for creating a family of functions or classes.

## [Runtime Polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/)

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding., on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

**Example**

* Java

|  |
| --- |
| // Java Program for Method Overriding  // Class 1  // Helper class  **class** Parent {  **void** Print()      { System.out.println("parent class"); }  }  // Class 2  // Helper class  **class** subclass1 **extends** Parent {  **void** Print() { System.out.println("subclass1"); }  }  // Class 3  // Helper class  **class** subclass2 **extends** Parent {  **void** Print()      { System.out.println("subclass2"); }  }  // Class 4  // Main class  **class** GFG {  **public** **static** **void** main(String[] args)      {          Parent a;          a = **new** subclass1();          a.Print();          a = **new** subclass2();          a.Print();      }  } |

**Output**

subclass1

subclass2

#### Explanation of the above code:

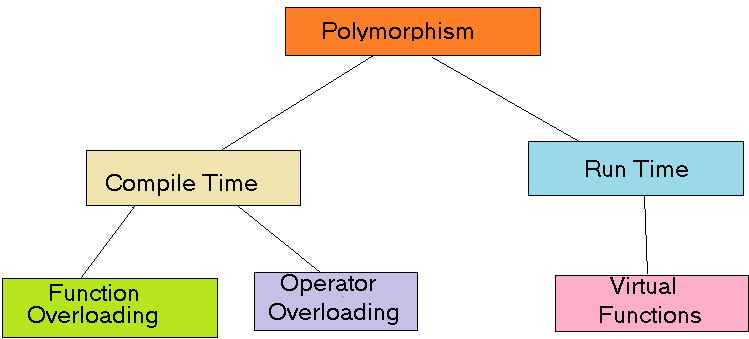
Here in this program, When an object of a child class is created, then the method inside the child class is called. This is because The method in the parent class is overridden by the child class. Since The method is overridden, This method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

## ****Subtype of Run-time Polymorphism****

### **i. Virtual functions**

It allows an object of a derived class to behave as if it were an object of the base class. The derived class can override the virtual function of the base class to provide its own implementation. The function call is resolved at runtime, depending on the actual type of the object.

**Diagram –**



Polymorphism in Java is a concept that allows objects of different classes to be treated as objects of a common class. It enables objects to behave differently based on their specific class type.

### Advantages of Polymorphism in Java

1. Increases code reusability by allowing objects of different classes to be treated as objects of a common class.
2. Improves readability and maintainability of code by reducing the amount of code that needs to be written and maintained.
3. Supports dynamic binding, enabling the correct method to be called at runtime, based on the actual class of the object.
4. Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.

### Disadvantages of Polymorphism in Java

1. Can make it more difficult to understand the behavior of an object, especially if the code is complex.
2. This may lead to performance issues, as polymorphic behavior may require additional computations at runtime.

# Abstract Class

In Java, abstract class is declared with the abstract keyword. It may have both abstract and non-abstract methods(methods with bodies). An abstract is a Java modifier applicable for classes and methods in Java but not for Variables. In this article, we will learn the use of abstract classes in Java.

## What is Abstract Class in Java?

Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the “abstract” keyword in its class definition.

### **Illustration of Abstract class**

abstract class Shape   
{  
 int color;  
 // An abstract function  
 abstract void draw();  
}

In Java, the following some important observations about abstract classes are as follows:

1. An instance of an abstract class can not be created.
2. Constructors are allowed.
3. We can have an abstract class without any abstract method.
4. There can be a **final method** in abstract class but any abstract method in class(abstract class) can not be declared as final  or in simpler terms final method can not be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”
5. We can define static methods in an abstract class
6. We can use the **abstract keyword** for declaring ***top-level classes (Outer class) as well as inner classes*** as abstract
7. If a**class** contains at least **one abstract method**then compulsory should declare a class as abstract
8. If the**Child class** is unable to provide implementation to all abstract methods of the**Parent class**then we should declare that **Child class as abstract**so that the next level Child class should provide implementation to the remaining abstract method

## Examples of Java Abstract Class

### 1. Example of Abstract Class that has Abstract method

**Below is the implementation of the above topic:**

|  |
| --- |
| // Abstract class  **abstract** **class** Sunstar {  **abstract** **void** printInfo();  }  // Abstraction performed using extends  **class** Employee **extends** Sunstar {  **void** printInfo()      {          String name = "avinash";  **int** age = 21;  **float** salary = 222.2F;          System.out.println(name);          System.out.println(age);          System.out.println(salary);      }  }  // Base class  **class** Base {  **public** **static** **void** main(String args[])      {          Sunstar s = **new** Employee();          s.printInfo();      }  } |

**Output**

avinash

21

222.2

### 2. Abstract Class having constructor, data member, and methods

Elements abstract class can have

* data member
* abstract method
* method body (non-abstract method)
* constructor
* main() method.

**Below is the implementation of the above topic:**

* Java

|  |
| --- |
| // Java Program to implement Abstract Class  // having constructor, data member, and methods  **import** java.io.\*;  **abstract** **class** Subject {      Subject() {        System.out.println("Learning Subject");      }  **abstract** **void** syllabus();  **void** Learn(){            System.out.println("Preparing Right Now!");      }  }  **class** IT **extends** Subject {  **void** syllabus(){      System.out.println("C , Java , C++");    }  }  **class** GFG {  **public** **static** **void** main(String[] args) {          Subject x=**new** IT();            x.syllabus();            x.Learn();      }  } |

**Output**

Learning Subject

C , Java , C++

Preparing Right Now!

## Properties of Abstract class

Let us elaborate on these observations and do justify them with help of clean java programs as follows.

### **Observation 1**

In Java, just like in C++ an instance of an abstract class cannot be created, we can have references to abstract class type though. It is as shown below via the clean Java program.

**Example**

* Java

|  |
| --- |
| // Java Program to Illustrate  // that an instance of Abstract  // Class can not be created  // Class 1  // Abstract class  **abstract** **class** Base {  **abstract** **void** fun();  }    // Class 2  **class** Derived **extends** Base {  **void** fun()      {          System.out.println("Derived fun() called");      }  }  // Class 3  // Main class  **class** Main {      // Main driver method  **public** **static** **void** main(String args[])      {          // Uncommenting the following line will cause          // compiler error as the line tries to create an          // instance of abstract class. Base b = new Base();          // We can have references of Base type.          Base b = **new** Derived();          b.fun();      }  } |

**Output**

Derived fun() called

### **Observation 2**

Like C++, an **abstract class** can contain **constructors** in Java. And a constructor of an abstract class is called when an instance of an inherited class is created. It is as shown in the program below as follows:

**Example:**

* Java

|  |
| --- |
| // Java Program to Illustrate Abstract Class  // Can contain Constructors  // Class 1  // Abstract class  **abstract** **class** Base {      // Constructor of class 1      Base()      {          // Print statement          System.out.println("Base Constructor Called");      }      // Abstract method inside class1  **abstract** **void** fun();  }  // Class 2  **class** Derived **extends** Base {      // Constructor of class2      Derived()      {          System.out.println("Derived Constructor Called");      }      // Method of class2  **void** fun()      {          System.out.println("Derived fun() called");      }  }    // Class 3  // Main class  **class** GFG {      // Main driver method  **public** **static** **void** main(String args[])      {          // Creating object of class 2          // inside main() method          Derived d = **new** Derived();          d.fun();      }  } |

**Output**

Base Constructor Called

Derived Constructor Called

Derived fun() called

### **Observation 3**

In Java, we can have ***an abstract class without any abstract method***. This allows us to ***create classes that cannot be instantiated but can only be inherited***. It is as shown below as follows with help of a clean java program.

**Example:**

* Java

|  |
| --- |
| // Java Program to illustrate Abstract class  // Without any abstract method  // Class 1  // An abstract class without any abstract method  **abstract** **class** Base {      // Demo method. This is not an abstract method.  **void** fun()      {          // Print message if class 1 function is called          System.out.println(              "Function of Base class is called");      }  }    // Class 2  **class** Derived **extends** Base {      // This class only inherits the Base class methods and      // properties  // Class 3  **class** Main {      // Main driver method  **public** **static** **void** main(String args[])      {          // Creating object of class 2          Derived d = **new** Derived();          // Calling function defined in class 1 inside main()          // with object of class 2 inside main() method          d.fun();      }  } |

**Output**

Function of Base class is called

### **Observation 4**

Abstract classes can also have ***final*** methods (methods that cannot be overridden)

**Example:**

* Java

|  |
| --- |
| // Java Program to Illustrate Abstract classes  // Can also have Final Methods  // Class 1  // Abstract class  **abstract** **class** Base {  **final** **void** fun()      {          System.out.println("Base fun() called");      }  }    // Class 2  **class** Derived **extends** Base  }  // Class 3  // Main class  **class** GFG {      // Main driver method  **public** **static** **void** main(String args[])      {          {              // Creating object of abstract class              Base b = **new** Derived();              // Calling method on object created above              // inside main method              b.fun();          }      }  } |

**Output**

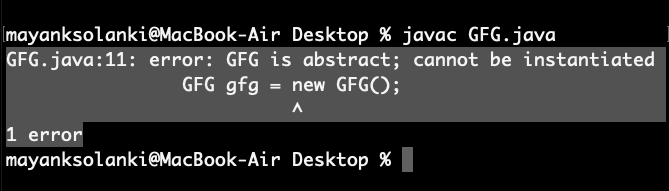
Base fun() called

### **Observation 5**

For any abstract java class we are not allowed to create an object i.e., for an abstract class instantiation is not possible.

|  |
| --- |
| // Java Program to Illustrate Abstract Class  // Main class  // An abstract class  **abstract** **class** GFG {      // Main driver method  **public** **static** **void** main(String args[])            // Trying to create an object          GFG gfg = **new** GFG();      }  } |

**Output:**



### **Observation 6**

Similar to the interface ***we can define static methods in an abstract class*** that***can be called independently without an object.***

|  |
| --- |
| // Java Program to Illustrate  // Static Methods in Abstract  // Class Can be called Independently  // Class 1  // Abstract class  **abstract** **class** Helper {      // Abstract method  **static** **void** demofun()      {          System.out.println("Geeks for Geeks");      }  }  // Class 2  // Main class extending Helper class  **public** **class** GFG **extends** Helper {      // Main driver method  **public** **static** **void** main(String[] args)      {          // Calling method inside main()          // as defined in above class          Helper.demofun();      }  } |

**Output**

Geeks for Geeks

### **Observation 7**

We can use the **abstract keyword** for declaring top-level classes (Outer class) as well as inner classes as abstract

|  |
| --- |
| **import** java.io.\*;  **abstract** **class** B {      // declaring inner class as abstract with abstract      // method  **abstract** **class** C {  **abstract** **void** myAbstractMethod();      }  }  **class** D **extends** B {  **class** E **extends** C {          // implementing the abstract method  **void** myAbstractMethod()          {              System.out.println(                  "Inside abstract method implementation");          }      }  }    **public** **class** Main {    **public** **static** **void** main(String args[])      {          // Instantiating the outer class          D outer = **new** D();          // Instantiating the inner class          D.E inner = outer.**new** E();          inner.myAbstractMethod();      }  } |

**Output**

Inside abstract method implementation

### **Observation 8**

If a **class contains at least one abstract method** then **compulsory that we should declare the class as abstract** otherwise we will get a compile-time error ,If a class contains at least one abstract method then, implementation is not complete for that class, and hence it is not recommended to create an object so in order to restrict object creation for such partial classes we use**abstract keyword.**

|  |
| --- |
| /\*package whatever //do not write package name here \*/  **import** java.io.\*;  // here if we remove the abstract  // keyword then we will get compile  // time error due to abstract method  **abstract** **class** Demo {  **abstract** **void** m1();  }  **class** Child **extends** Demo {  **public** **void** m1()      {        System.out.print("Hello");      }  }  **class** GFG {  **public** **static** **void** main(String[] args)      {          Child c = **new** Child();          c.m1();      }  } |

**Output**

Hello

### **Observation 9**

If the**Child**class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method.

* Java

|  |
| --- |
| // Java Program to demonstrate  // Observation  **import** java.io.\*;  **abstract** **class** Demo {  **abstract** **void** m1();  **abstract** **void** m2();  **abstract** **void** m3();  }  **abstract** **class** FirstChild **extends** Demo {  **public** **void** m1() {        System.out.println("Inside m1");      }  }  **class** SecondChild **extends** FirstChild {  **public** **void** m2() {        System.out.println("Inside m2");      }  **public** **void** m3() {        System.out.println("Inside m3");      }  }  **class** GFG {  **public** **static** **void** main(String[] args)      {          // if we remove the abstract keyword from FirstChild          // Class and uncommented below obj creation for          // FirstChild then it will throw          // compile time error as did't override all the          // abstract methods            // FirstChild f=new FirstChild();          // f.m1();          SecondChild s = **new** SecondChild();          s.m1();          s.m2();          s.m3();      }  } |

**Output**

Inside m1

Inside m2

Inside m3

In C++, if a class has at least one [pure virtual function](https://www.geeksforgeeks.org/pure-virtual-functions-and-abstract-classes/), then the class becomes abstract. Unlike C++, in Java, a separate keyword abstract is used to make a class abstract.

## Conclusion

Points to remember from this article are mentioned below:

* An abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties.
* An abstract class can be created using “abstract” keywords.
* We can have an abstract class without any abstract method.

# **Java Type Casting**

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

* **Widening Casting** (automatically) - converting a smaller type to a larger type size  
  byte -> short -> char -> int -> long -> float -> double
* **Narrowing Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char -> short -> byte

## Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

### **Example**

public class Main {

public static void main(String[] args) {

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

}

}

## Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

### **Example**

public class Main {

public static void main(String[] args) {

double myDouble = 9.78d;

int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt); // Outputs 9

}

}

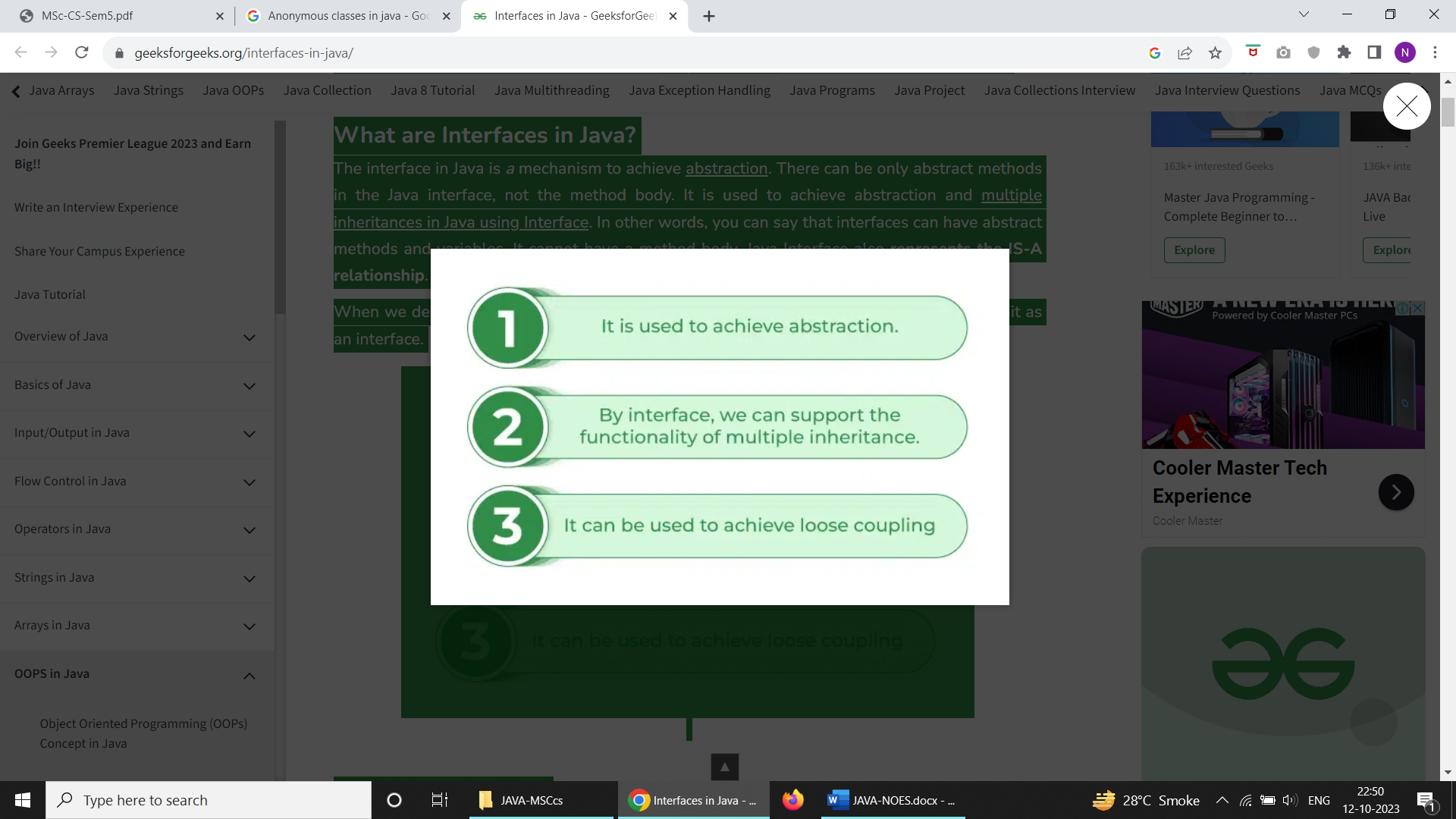
# Interface

An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.

## What are Interfaces in Java?

The interface in Java is a mechanism to achieve [abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2/). There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and [multiple inheritances in Java using Interface](https://www.geeksforgeeks.org/java-and-multiple-inheritance/). In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also **represents the IS-A relationship**.

When we decide on a type of entity by its behavior and not via attribute we should define it as an interface.



### **Syntax for Java Interfaces**

interface {  
  
 // declare constant fields  
 // declare methods that abstract   
 // by default.   
}

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement the interface use the implements keyword.

### Uses of Interfaces in Java

*Uses of Interfaces in Java are mentioned below:*

* *It is used to achieve total abstraction.*
* *Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.*
* *Any class can extend only 1 class but can any class implement an infinite number of interface.*
* *It is also used to achieve loose coupling.*
* *Interfaces are used to implement abstraction.*

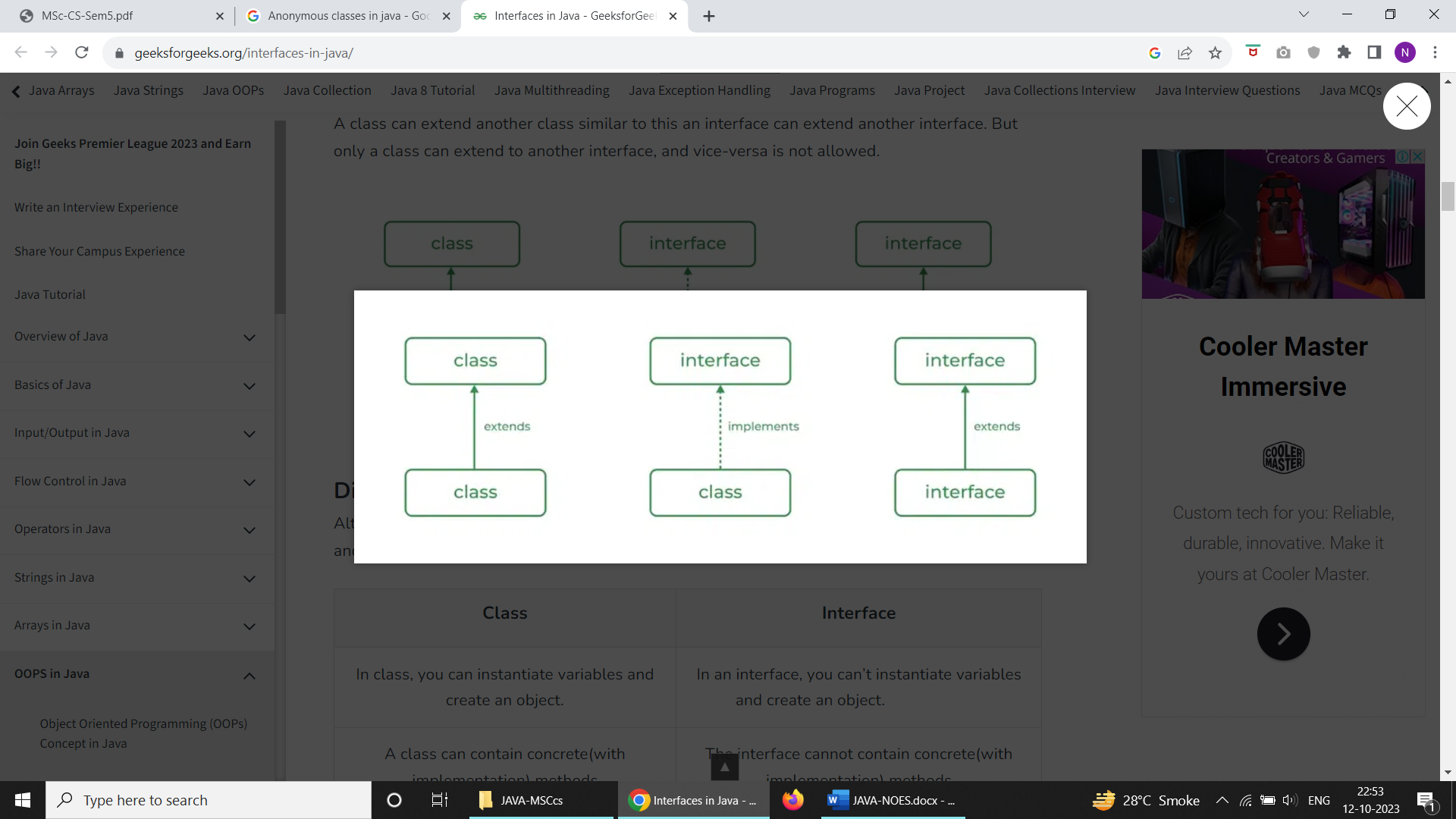
**So the question arises why use interfaces when we have abstract classes?**

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public, and static.

// A simple interface  
  
interface Player  
{  
 final int id = 10;  
 int move();  
}

## Relationship Between Class and Interface

A class can extend another class similar to this an interface can extend another interface. But only a class can extend to another interface, and vice-versa is not allowed.



## Difference Between Class and Interface

Although Class and Interface seem the same there have certain differences between Classes and Interface. The major differences between a class and an interface are mentioned below:

| **Class** | **Interface** |
| --- | --- |
| In class, you can instantiate variables and create an object. | In an interface, you can’t instantiate variables and create an object. |
| A class can contain concrete(with implementation) methods | The interface cannot contain concrete(with implementation) methods |
| The access specifiers used with classes are private, protected, and public. | In Interface only one specifier is used- Public. |

***Implementation:****To implement an interface we use the keyword****implements***

* Java

|  |
| --- |
| // Java program to demonstrate working of  // interface    **import** java.io.\*;    // A simple interface  **interface** In1 {        // public, static and final  **final** **int** a = 10;        // public and abstract  **void** display();  }    // A class that implements the interface.  **class** TestClass **implements** In1 {        // Implementing the capabilities of      // interface.  **public** **void** display(){        System.out.println("Geek");      }        // Driver Code  **public** **static** **void** main(String[] args)      {          TestClass t = **new** TestClass();          t.display();          System.out.println(a);      }  } |

**Output**

Geek

10

## Java Interfaces Examples

Let’s consider the example of vehicles like bicycles, cars, bikes, etc they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car, etc implement all these functionalities in their own class in their own way.

**Below is the implementation of the above topic:**

|  |
| --- |
| // Java program to demonstrate the  // real-world example of Interfaces    **import** java.io.\*;    **interface** Vehicle {        // all are the abstract methods.  **void** changeGear(**int** a);  **void** speedUp(**int** a);  **void** applyBrakes(**int** a);  }    **class** Bicycle **implements** Vehicle{    **int** speed;  **int** gear;        // to change gear      @Override  **public** **void** changeGear(**int** newGear){            gear = newGear;      }        // to increase speed      @Override  **public** **void** speedUp(**int** increment){            speed = speed + increment;      }        // to decrease speed      @Override  **public** **void** applyBrakes(**int** decrement){            speed = speed - decrement;      }    **public** **void** printStates() {          System.out.println("speed: " + speed              + " gear: " + gear);      }  }    **class** Bike **implements** Vehicle {    **int** speed;  **int** gear;        // to change gear      @Override  **public** **void** changeGear(**int** newGear){            gear = newGear;      }        // to increase speed      @Override  **public** **void** speedUp(**int** increment){            speed = speed + increment;      }        // to decrease speed      @Override  **public** **void** applyBrakes(**int** decrement){            speed = speed - decrement;      }    **public** **void** printStates() {          System.out.println("speed: " + speed              + " gear: " + gear);      }    }  **class** GFG {    **public** **static** **void** main (String[] args) {            // creating an instance of Bicycle          // doing some operations          Bicycle bicycle = **new** Bicycle();          bicycle.changeGear(2);          bicycle.speedUp(3);          bicycle.applyBrakes(1);            System.out.println("Bicycle present state :");          bicycle.printStates();            // creating instance of the bike.          Bike bike = **new** Bike();          bike.changeGear(1);          bike.speedUp(4);          bike.applyBrakes(3);            System.out.println("Bike present state :");          bike.printStates();      }  } |

**Output**

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear: 1

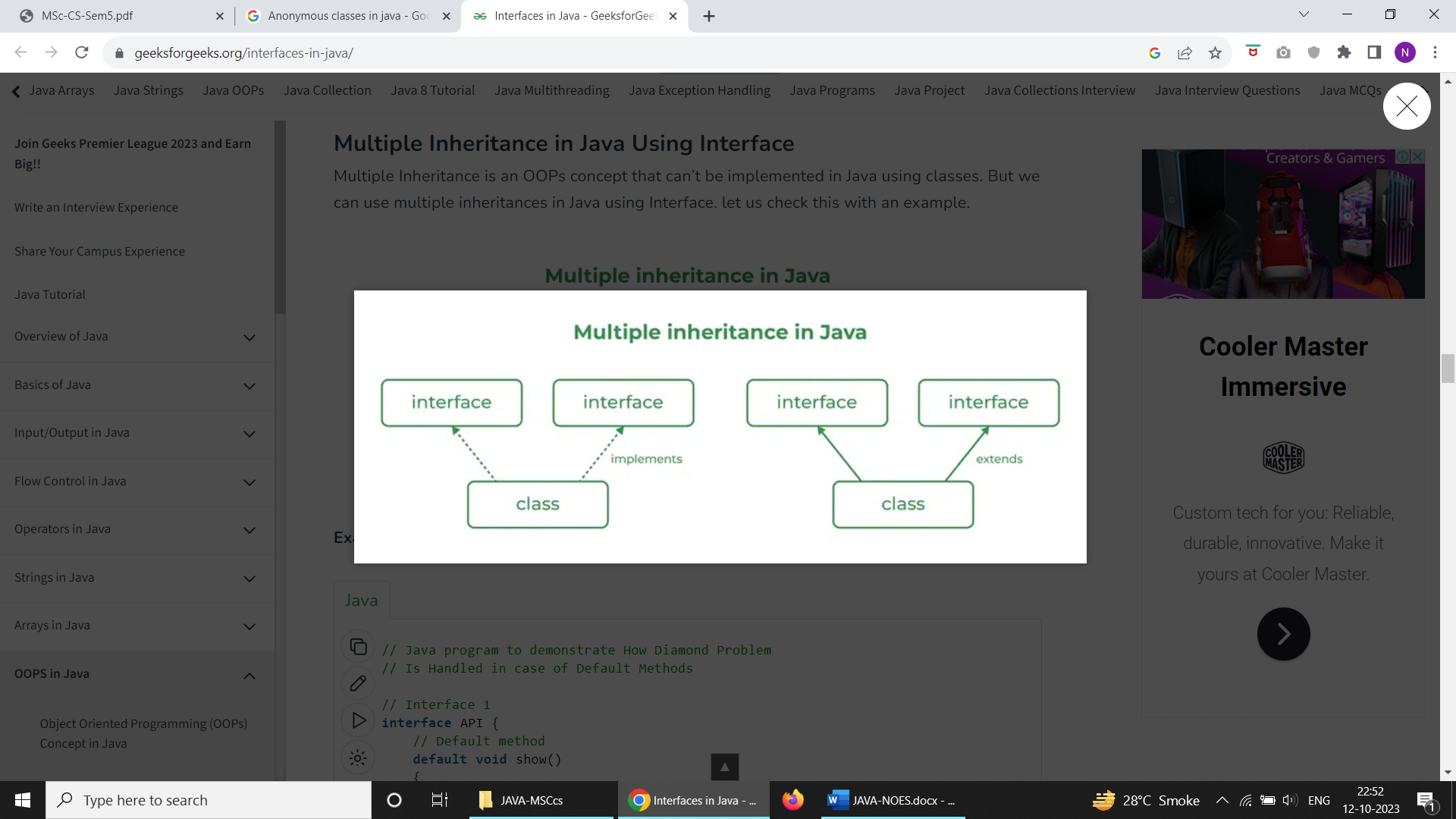
## Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

1. Without bothering about the implementation part, we can achieve the security of the implementation.
2. In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

## Multiple Inheritance in Java Using Interface

Multiple Inheritance is an OOPs concept that can’t be implemented in Java using classes. But we can use multiple inheritances in Java using Interface. let us check this with an example.



**Example:**

* Java

|  |
| --- |
| // Java program to demonstrate How Diamond Problem  // Is Handled in case of Default Methods    // Interface 1  **interface** API {      // Default method  **default** **void** show()      {            // Print statement          System.out.println("Default API");      }  }    // Interface 2  // Extending the above interface  **interface** Interface1 **extends** API {  }    // Interface 3  // Extending the above interface  **interface** Interface2 **extends** API {  }    // Main class  // Implementation class code  **class** TestClass **implements** Interface1, Interface2 {      // Main driver method  **public** **static** **void** main(String args[])      {          // Creating object of this class          // in main() method          TestClass d = **new** TestClass();            // Now calling the function defined in interface 1          // from whom Interface 2and 3 are deriving          d.show();      }  } |

**Output**

Default API

## New Features Added in Interfaces in JDK 8

There are certain features added to Interfaces in JDK 8 update mentioned below:

***1.***Prior to JDK 8, the interface could not define the implementation. We can now add default implementation for interface methods. This default implementation has a special use and does not affect the intention behind interfaces.

Suppose we need to add a new function to an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

**Below is the implementation of the above point:**

* Java

|  |
| --- |
| // Java program to show that interfaces can  // have methods from JDK 1.8 onwards    **interface** In1  {  **final** **int** a = 10;  **default** **void** display()      {          System.out.println("hello");      }  }    // A class that implements the interface.  **class** TestClass **implements** In1  {      // Driver Code  **public** **static** **void** main (String[] args)      {          TestClass t = **new** TestClass();          t.display();      }  } |

**Output**

hello

***2.***Another feature that was added in JDK 8 is that we can now define static methods in interfaces that can be called independently without an object.

**Note:** these methods are not inherited.

* Java

|  |
| --- |
| // Java Program to show that interfaces can  // have methods from JDK 1.8 onwards    **interface** In1  {  **final** **int** a = 10;  **static** **void** display()      {          System.out.println("hello");      }  }    // A class that implements the interface.  **class** TestClass **implements** In1  {      // Driver Code  **public** **static** **void** main (String[] args)      {          In1.display();      }  } |

**Output**

hello

## Extending Interfaces

One interface can inherit another by the use of keyword extends. When a class implements an interface that inherits another interface, it must provide an implementation for all methods required by the interface inheritance chain.

**Program 1:**

* Java

|  |
| --- |
| **interface** A {  **void** method1();  **void** method2();  }  // B now includes method1 and method2  **interface** B **extends** A {  **void** method3();  }  // the class must implement all method of A and B.  **class** gfg **implements** B {  **public** **void** method1()      {          System.out.println("Method 1");      }  **public** **void** method2()      {          System.out.println("Method 2");      }  **public** **void** method3()      {          System.out.println("Method 3");      }  } |

**Program 2:**

* Java

|  |
| --- |
| **interface** Student  {  **public** **void** data();    }  **class** avi **implements** Student  {  **public** **void** data ()      {          String name="avinash";  **int** rollno=68;          System.out.println(name);          System.out.println(rollno);      }  }  **public** **class** inter\_face  {  **public** **static** **void** main (String args [])      {          avi h= **new** avi();          h.data();      }  } |

**Output**

avinash

68

In a Simple way, the interface contains multiple abstract methods, so write the implementation in implementation classes. If the implementation is unable to provide an implementation of all abstract methods, then declare the implementation class with an abstract modifier, and complete the remaining method implementation in the next created child classes. It is possible to declare multiple child classes but at final we have completed the implementation of all abstract methods.

In general, the development process is step by step:

*1. Level 1-  interfaces: It contains the service details.*

*2. Level 2 – abstract classes: It contains partial implementation.*

*3. Level 3 – implementation classes: It contains all implementation.*

*4. Level 4 – Final Code / Main Method: It have access of all interfaces data.*

**Example:**

|  |
| --- |
| // Java Program for  // implementation Level wise  **import** java.io.\*;  **import** java.lang.\*;  **import** java.util.\*;    // Level 1  **interface** Bank {  **void** deposit();  **void** withdraw();  **void** loan();  **void** account();  }    // Level 2  **abstract** **class** Dev1 **implements** Bank {  **public** **void** deposit()      {          System.out.println("Your deposit Amount :" + 100);      }  }    **abstract** **class** Dev2 **extends** Dev1 {  **public** **void** withdraw()      {          System.out.println("Your withdraw Amount :" + 50);      }  }    // Level 3  **class** Dev3 **extends** Dev2 {  **public** **void** loan() {}  **public** **void** account() {}  }    // Level 4  **class** GFG {  **public** **static** **void** main(String[] args)      {          Dev3 d = **new** Dev3();          d.account();          d.loan();          d.deposit();          d.withdraw();      }  } |

**Output**

Your deposit Amount :100

Your withdraw Amount :50

## New Features Added in Interfaces in JDK 9

From Java 9 onwards, interfaces can contain the following also:

1. Static methods
2. Private methods
3. Private Static methods

## Important Points in Java Interfaces

In the article, we learn certain important points about interfaces as mentioned below:

* *We can’t create an instance(interface can’t be instantiated) of the interface but we can make the reference of it that refers to the Object of its implementing class.*
* *A class can implement more than one interface.*
* *An interface can extend to another interface or interface (more than one interface).*
* *A class that implements the interface must implement all the methods in the interface.*
* *All the methods are public and abstract. And all the fields are public, static, and final.*
* *It is used to achieve multiple inheritances.*
* *It is used to achieve loose coupling.*
* *Inside the Interface not possible to declare instance variables because by default variables are****public static final.***
* *Inside the Interface, constructors are not allowed.*
* *Inside the interface main method is not allowed.*
* *Inside the interface, static, final, and private methods declaration are not possible.*

## Frequently Asked Questions in Interfaces

### 1. What is a marker or tagged interface?

*Tagged Interfaces are interfaces without any methods they serve as a marker without any capabilities.*

### 2. **How many Types of interface in Java?**

*Types of interfaces in Java are mentioned below:*

1. *Functional Interface*
2. *Marker interface*

### 3.  Why multiple inheritance is not supported through class in Java?

*Multiple Inheritance is not supported through class in Java so to avoid certain challenges like Ambiguity and diamond problems.*

# Native Keyword

The native keyword in Java is applied to a method to indicate that the method is implemented in native code using JNI (Java Native Interface). The native keyword is a modifier that is applicable **only for methods**, and we can’t apply it anywhere else. The methods which are implemented in C, C++ are called native methods or foreign methods.

The native modifier indicates that a method is implemented in platform-dependent code, often seen in C language. Native modifier indicates that a method is implemented in platform-dependent code, often in C.

### Main objectives of the native keyword

* To improve the performance of the system.
* To achieve machine level/memory level communication.
* To use already existing legacy non-java code.

***Conclusion:****Java application scan call code is written in C, C++, or assembler.*

## Create Native Method

The steps for the creation of native methods are as follows:

1. Write java code
2. Compile the Java code.
3. Create C header(.h file)
4. Create C stubs file (using tool: Java HEdge)
5. Write C code
6. Create a shared code library (DLL)
7. Run application

### **Implementation of**Native keyword in Java

Let us first take random Java code that contains the native method and later we will be compiling it.  We are done with the above two steps. For steps 3 and 4 we will be using the existing .exe known as java HEdge” in order to create a C header and C stub file.

Now we will insert(write) our C code(or use) and later using DLL, we will be creating objects of the same inside our application( Main example1A) and later calling the native methods thereby within the java program.

**Example 1-A:**Application

|  |
| --- |
| // Java Program to Illustrate Native Keyword  // Inside DLL named: NameOfDLLFile    // Main class  // NativeDemo  **class** GFG {        // Method 1  **public** **static** **void** main(String[] args)      {    **int** var;            // Here we will not be having body of this method          // in our java code here          NameOfDLLFile obj = **new** NameOfDLLFile();            obj.var = **null**;            System.out.println("Before native method: var = "                             + var);            obj.test();            System.out.println("After native method: var = "                             + var);      }        // Native method  **public** **native** **void** test()      {    **static**          {                // We will be loading body from DLL file              // It has to be present in DLL file              System.loadLibrary("NameOfDLLFile");                // Above C code in loaded in the JVM          }      }  } |

For the above program C code that is shared in DLL is as follows:

**Example 1-B:** Support to the above example

* C++

|  |
| --- |
| // C++ Program to Be Shared In DLL to Illustrate  // Native Method in Java    // Importing required libraries  #include <iostream>    **using** **namespace** std;  // Method 2  // Native  **void** test(**int** var) { cout << var; }    // Method 1  // Main driver method  **int** main()  {        test(10);    **return** 1;  } |

***Note:****DLL is named as can be perceived from program 1A:****NameOfDLLFile***

**Output:**

Before native method: var = null  
After native method: var = 10

Do remember that there are certain **important points about native keywords**, which are as follows:

* For native methods, implementation is already available in old languages like C, and C++ and we are not responsible for providing an implementation. Hence native method declaration should end with ; (semi-colon).
* We can’t declare a native method as [abstract](https://www.geeksforgeeks.org/abstract-classes-in-java/).
* We can’t declare a native method as [strictfp](https://www.geeksforgeeks.org/strictfp-keyword-java/) because there is no guarantee that old languages (C, C++) follow IEEE 754 standards. Hence native strictfp combination is an illegal combination for methods.
* The main advantage of native keywords is performance improvement, but the main disadvantage of native keywords is that it breaks the platform-independent nature of Java.

***Note:****Do go through strictfp keyword of java as is one of the concept which even very good java developer is unaware of.*

In this section, we explain how to declare a native method in Java and how to generate the corresponding C/C++ function prototype.

**Syntax:**Declaring Native Methods

private native String getLine(String prompt);

**Syntax:**From the Native Language Side

javah -jni Prompt  
JNIEXPORT jstring JNICALL Java\_Prompt\_getLine(JNIEnv \*, jobject, jstring);

# Anonymous Inner Class

[Nested Classes in Java](https://www.geeksforgeeks.org/nested-classes-java/) is prerequisite required before adhering forward to grasp about anonymous Inner class. It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overriding methods of a class or interface, without having to actually subclass a class.

***Tip:****Anonymous inner classes are useful in writing implementation classes for listener interfaces in graphics programming.*

The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

**Syntax:**

// Test can be interface,abstract/concrete class

Test t = new Test()

{

// data members and methods

public void test\_method()

{

........

........

}

};

Now let us do discuss the difference between regular class(normal classes) and Anonymous Inner class

* A normal class can implement any number of interfaces but the anonymous inner class can implement only one interface at a time.
* A regular class can extend a class and implement any number of interfaces simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time.
* For regular/normal class, we can write any number of constructors but we can’t write any constructor for anonymous Inner class because the anonymous class does not have any name and while defining constructor class name and constructor name must be same.

**Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class**

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

* An anonymous class has access to the members of its enclosing class.
* An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
* Like a nested class, a declaration of a type (such as a variable) in anonymous class shadows any other declarations in the enclosing scope that have the same name.

Anonymous classes also have the same restrictions as local classes with respect to their members:

* We cannot declare static initializers or member interfaces in an anonymous class.
* An anonymous class can have static members provided that they are constant variables.

***Note:****We can declare the following in anonymous classes as follows:*

* *Fields*
* *Extra methods (even if they do not implement any methods of the supertype)*
* *Instance initializers*
* *Local classes*

**Ways:**

Anonymous inner classes are generic created via below listed two ways as follows:

1. Class (may be abstract or concrete)
2. Interface

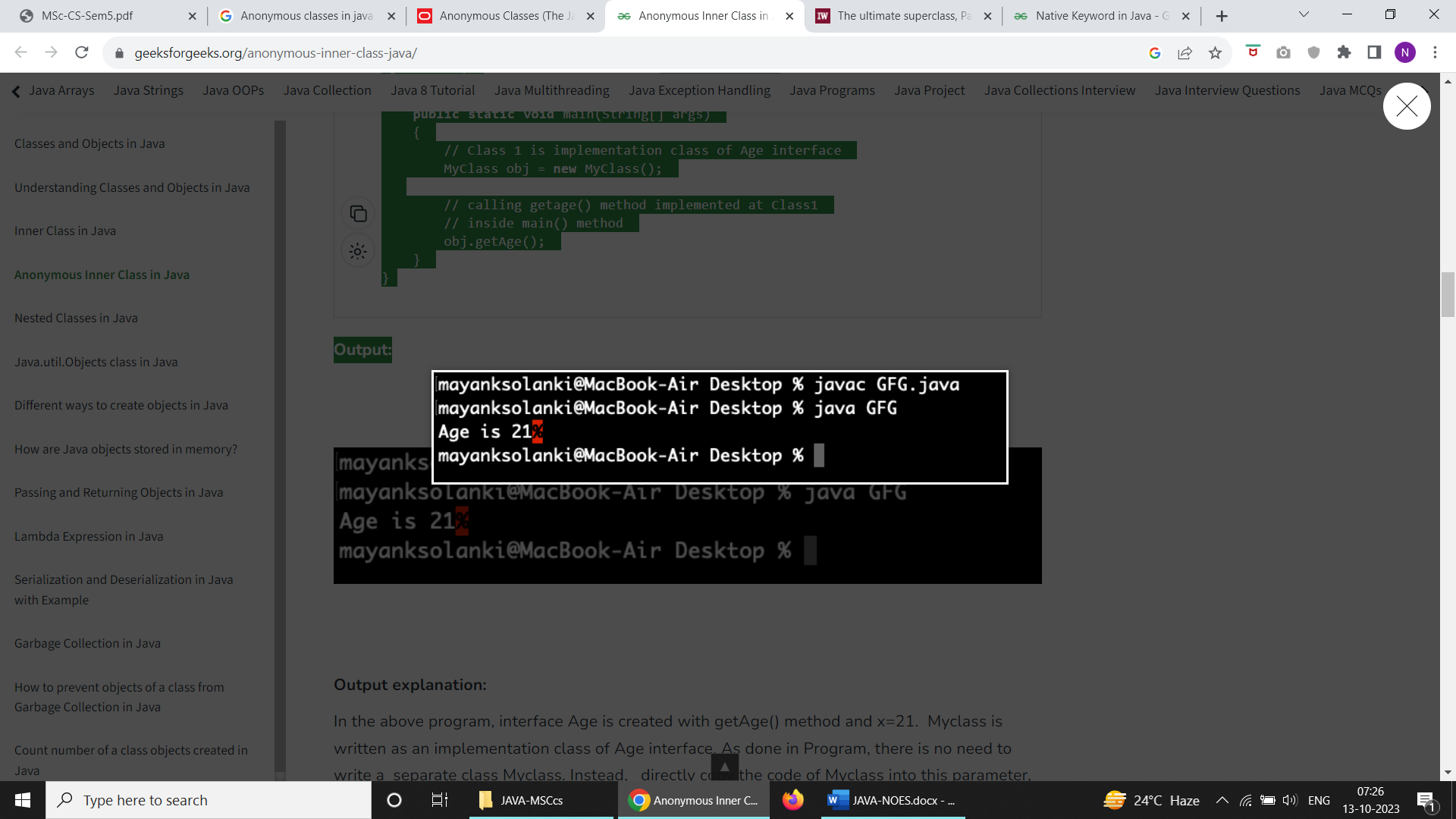
Now let us take an example with which we will understand anonymous inner class, let us take a simple program

**Example**

* Java

|  |
| --- |
| // Java program to demonstrate Need for  // Anonymous Inner class    // Interface  **interface** Age {        // Defining variables and methods  **int** x = 21;  **void** getAge();  }    // Class 1  // Helper class implementing methods of Age Interface  **class** MyClass **implements** Age {        // Overriding getAge() method      @Override **public** **void** getAge()      {          // Print statement          System.out.print("Age is " + x);      }  }    // Class 2  // Main class  // AnonymousDemo  **class** GFG {      // Main driver method  **public** **static** **void** main(String[] args)      {          // Class 1 is implementation class of Age interface          MyClass obj = **new** MyClass();            // calling getage() method implemented at Class1          // inside main() method          obj.getAge();      }  } |

**Output:**



**Output explanation:**

In the above program, interface Age is created with getAge() method and x=21.  Myclass is written as an implementation class of Age interface. As done in Program, there is no need to write a  separate class Myclass. Instead,   directly copy the code of Myclass into this parameter, as shown here:

Age oj1 = new Age()

{

@Override

public void getAge()

{

System.out.print("Age is " + x);

}

};

*Here, an object to Age is not created but an object of Myclass is created and copied in the entire class code as shown above. This is possible only with anonymous inner class. Such a class is called ‘anonymous inner class’, so here we call ‘Myclass’ as anonymous inner class.*

**Example:**

* Java

|  |
| --- |
| // Java Program to Demonstrate Anonymous inner class    // Interface  **interface** Age {  **int** x = 21;  **void** getAge();  }    // Main class  **class** AnonymousDemo {        // Main driver method  **public** **static** **void** main(String[] args)      {            // Myclass is hidden inner class of Age interface          // whose name is not written but an object to it          // is created.          Age oj1 = **new** Age() {                @Override **public** **void** getAge()              {                  // printing  age                  System.out.print("Age is " + x);              }          };            oj1.getAge();      }  } |

**Output**

Age is 21

### Types of Anonymous Inner Class

Based on declaration and behavior, there are 3 types of anonymous Inner classes:

1. Anonymous Inner class that extends a class
2. Anonymous Inner class that implements an interface
3. Anonymous Inner class that defines inside method/constructor argument

**Type 1:**Anonymous Inner class that extends a class

We can have an anonymous inner class that extends a class. For example, we know that we can create a thread by extending a Thread class. Suppose we need an immediate thread but we don’t want to create a class that extends [Thread class](https://www.geeksforgeeks.org/java-lang-thread-class-java/) all the time. With the help of this type of Anonymous Inner class, we can define a ready thread.

**Example**

* Java

|  |
| --- |
| // Java program to illustrate creating an immediate thread  // Using Anonymous Inner class that extends a Class    // Main class  **class** MyThread {        // Main driver method  **public** **static** **void** main(String[] args)      {          // Using Anonymous Inner class that extends a class          // Here a Thread class          Thread t = **new** Thread() {                // run() method for the thread  **public** **void** run()              {                  // Print statement for child thread                  // execution                  System.out.println("Child Thread");              }          };            // Starting the thread          t.start();            // Displaying main thread only for readability          System.out.println("Main Thread");      }  } |

**Output**

Main Thread

Child Thread

**Type 2:**Anonymous Inner class that implements an interface

We can also have an anonymous inner class that implements an interface. For example, we also know that by implementing Runnable interface we can create a Thread. Here we use an anonymous Inner class that implements an interface.

**Example**

|  |
| --- |
| // Java program to illustrate defining a thread  // Using Anonymous Inner class that implements an interface    // Main class  **class** MyThread {        // Main driver method  **public** **static** **void** main(String[] args)      {          // Here we are using Anonymous Inner class          // that implements a interface i.e. Here Runnable          // interface          Runnable r = **new** Runnable() {                // run() method for the thread  **public** **void** run()              {                  // Print statement when run() is invoked                  System.out.println("Child Thread");              }          };            // Creating thread in main() using Thread class          Thread t = **new** Thread(r);            // Starting the thread using start() method          // which invokes run() method automatically          t.start();            // Print statement only          System.out.println("Main Thread");      }  } |

**Output**

Main Thread

Child Thread

**Type 3:**Anonymous Inner class that defines inside method/constructor argument

Anonymous inner classes in method/constructor arguments are often used in graphical user interface (GUI) applications. To get you familiar with syntax lets have a look at the following program that creates a thread using this type of Anonymous Inner class

**Example**

* Java

|  |
| --- |
| // Java program to illustrate defining a thread  // Using Anonymous Inner class that define inside argument    // Main class  **class** MyThread {      // Main driver method  **public** **static** **void** main(String[] args)      {          // Using Anonymous Inner class that define inside          // argument          // Here constructor argument          Thread t = **new** Thread(**new** Runnable() {    **public** **void** run()              {                  System.out.println("Child Thread");              }          });            t.start();            System.out.println("Main Thread");      }  } |

**Output**

Main Thread

Child Thread

However, constructors can not be declared in an anonymous class.

# final, finally and finalize in Java

[**final keyword**](https://www.geeksforgeeks.org/final-keyword-java/)

final (lowercase) is a reserved keyword in java. We can’t use it as an identifier, as it is reserved. We can use this keyword with variables, methods, and also with classes. The final keyword in java has a different meaning depending upon whether it is applied to a variable, class, or method.

**final with Variables:**The value of the variable cannot be changed once initialized.

**class** A {

**public** **static** **void** main(String[] args)

    {

**int** a = 5;

**final** **int** b = 6;

        a++;

        // Immediately gives Compile Time error.

        b++;

    }

}

If we declare any variable as final, we can’t modify its contents since it is final, and if we modify it then we get Compile Time Error.

**final with Class:** The class cannot be subclassed. Whenever we declare any class as final, it means that we can’t extend that class or that class can’t be extended, or we can’t make a subclass of that class.

**final** **class** RR {

**public** **static** **void** main(String[] args)

    {

**int** a = 10;

    }

}

// here gets Compile time error that

// we can't extend RR as it is final.

**class** KK **extends** RR {

    // more code here with main method

}

**final with Method:** The method cannot be overridden by a subclass. Whenever we declare any method as final, then it means that we can’t override that method.

**class** QQ {

**final** **void** rr() {}

**public** **static** **void** main(String[] args)

    {

    }

}

**class** MM **extends** QQ {

    // Here we get compile time error

    // since can't extend rr since it is final.

**void** rr() {}

}

***Note :****If a class is declared as final as****by default****all of the methods present in that class are automatically final, but****variables are not****.*

EXAMPLE :

|  |
| --- |
| **final** **class** G {      // by default it is final.  **void** h() {}      // by default it is not final.  **static** **int** j = 30;  **public** **static** **void** main(String[] args)      {          // See modified contents of variable j.          j = 36;          System.out.println(j);      }  } |

**Output**:

36

[**finally keyword**](https://www.geeksforgeeks.org/final-keyword-java/)

Just as final is a reserved keyword, so in the same way finally is also a reserved keyword in java i.e, we can’t use it as an identifier. The **finally** keyword is used in association with a [try/catch block](https://www.geeksforgeeks.org/flow-control-in-try-catch-finally-in-java/) and guarantees that a section of code will be executed, even if an exception is thrown. The final block will be executed after the try and catch blocks, but before control transfers back to its origin. finally is executed even if try block has return statement.

|  |
| --- |
| **class** Geek {      // A method that throws an exception and has finally.      // This method will be called inside try-catch.  **static** **void** A()      {  **try** {              System.out.println("inside A");  **throw** **new** RuntimeException("demo");          }  **finally**          {              System.out.println("A's finally");          }      }        // This method also calls finally. This method      // will be called outside try-catch.  **static** **void** B()      {  **try** {              System.out.println("inside B");  **return**;          }  **finally**          {              System.out.println("B's finally");          }      }    **public** **static** **void** main(String args[])      {  **try** {              A();          }  **catch** (Exception e) {              System.out.println("Exception caught");          }          B();      }  } |

**Output:**

inside A

A's finally

Exception caught

inside B

B's finally

***Note :****If a class is declared as final as****by default****all of the methods present in that class are automatically final, but****variables are not****.*

**Case 1: Exceptions do not occur in the program**

|  |
| --- |
| // Java program to illustrate finally in  // Case where exceptions do not  // occur in the program  **class** B {  **public** **static** **void** main(String[] args)      {  **int** k = 55;  **try** {              System.out.println("In try block");  **int** z = k / 55;          }  **catch** (ArithmeticException e) {              System.out.println("In catch block");              System.out.println("Dividing by zero but caught");          }  **finally**          { System.out.println("Executes whether exception occurs or not"); }      }  } |

**Output**:

In try block

Executes whether exception occurs or not

Here the above exception does not occur but still, finally, the block executes since finally is meant to execute whether an exception occurs or not. **The flow of the Above Program**: First it starts from the main method and then goes to try block and in the try, since no exception occurs, the flow doesn’t go to catch block hence flow goes directly from try to finally block.

**Case 2: Exception occurs and corresponding catch block matches**

|  |
| --- |
| // Java program to illustrate finally in  // Case where exceptions occur  // and match in the program  **class** C {  **public** **static** **void** main(String[] args)      {  **int** k = 66;  **try** {              System.out.println("In try block");  **int** z = k / 0;              System.out.println("Flow doesn't came here");          }  **catch** (ArithmeticException e) {              System.out.println("In catch block");              System.out.println("Dividing by zero but caught");          }  **finally**          { System.out.println("Executes whether an exception occurs or not"); }      }  } |

**Output**:

In try block

In catch block

Dividing by zero but caught

Executes whether an exception occurs or not

Here, the above exception occurs, and the corresponding catch block is found but still, finally, the block executes since finally is meant to execute whether an exception occurs or not or whether the corresponding catch block is found or not. **The flow of the Above Program**: First, starts from the main method and then goes to try block, and in the try, an Arithmetic exception occurs, and the corresponding catch block is also available, so flow goes to catch block. After that flow doesn’t go to try block again since once an exception occurs in try block then flow **doesn’t**come back again to try block. After that finally, execute since finally is meant to execute whether an exception occurs or not or whether a corresponding catch block is found or not.

**Case 3: Exception occurs and the corresponding catch block is not found/match**

|  |
| --- |
| // Java program to illustrate finally in  // Case where exceptions occur  // and do not match any case in the program  **class** D {  **public** **static** **void** main(String[] args)      {  **int** k = 15;  **try** {              System.out.println("In try block");  **int** z = k / 0;          }  **catch** (NullPointerException e) {              System.out.println("In catch block");              System.out.println("Dividing by zero but caught");          }    **finally**          { System.out.println("Executes whether an exception occurs or not"); }      }  } |

**Output**:

In try block

Executes whether an exception occurs or not

Exception in thread "main":java.lang.ArithmeticException:

/ by zero followed by stack trace.

Here above exception occurs and the corresponding catch block is not found/matched but still finally block executes since finally is meant to execute whether an exception occurs or not or whether the corresponding catch block is found/matched or not. **The flow of the Above Program**: First starts from the main method and then goes to try block and in try an Arithmetic exception occurs, and the corresponding catch block is **not** available, so flow **doesn’t** go to catch block. After that flow doesn’t go to try block again since once an exception occurs in try block then flow **doesn’t**come back again to try block. After that finally, execute since finally is meant to execute whether an exception occurs or not or whether the corresponding catch block is found/matches or not.

**Case 4: finally block doesn’t get execute irrespective of the exception that occurs**

|  |
| --- |
| // Java program to illustrate finally in  // case where finally doesnt get executed  **class** E{  **public** **static** **void** main(String[] args){  **try**{        System.out.println("In try block");        System.exit(0);      }  **catch**(ArithmeticException e){        System.out.println("In catch block");      }  **finally**{        System.out.println("finally block");      }    }  } |

**Output:**

In try block

Here in the above program, finally, the block doesn’t execute. There is only one situation where finally block won’t be executed when we are using **System.exit(0)** method. When we are using System.exit(0) then JVM itself shutdown, hence in this case finally block won’t be executed. Here, the number within the parenthesis is known as the status code. Instead of zero, we can take any integer value where zero means normal termination, and non-zero means abnormal termination. Whether it is zero or non-zero, there is no change in the result and the effect is the same with respect to the program.

**Application of finally block**: So basically the use of finally block is **resource deallocation**. This means all the resources such as Network Connections, and Database Connections, which we opened in the try block is needed to be closed so that we won’t lose our resources as opened. So those resources are needed to be closed in the final block.

### **Finalize method**

It is a **method** that the [**Garbage Collector**](https://www.geeksforgeeks.org/garbage-collection-java/) always calls just **before** the deletion/destroying of the object which is eligible for Garbage Collection, so as to perform **clean-up activity**. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection, or we can say resource de-allocation. Remember, it is **not** a reserved keyword. Once the finalized method completes immediately Garbage Collector destroys that object. finalize method is present in the Object class and its syntax is:

**protected void finalize throws Throwable{}**

Since the Object class contains the finalize method, hence finalize method is available for every Java class since Object is the superclass of all Java classes. Since it is available for every java class hence Garbage Collector can call finalize method on **any Java object** Now, the finalize method which is present in the Object class, has an empty implementation, in our class clean-up activities are there, then we have to **override** this method to define our own clean-up activities. Cases related to finalizing method:

**Case 1:** The object which is eligible for Garbage Collection, that object’s corresponding class finalize method is going to be executed

|  |
| --- |
| **class** Hello {  **public** **static** **void** main(String[] args)      {          String s = **new** String("RR");          s = **null**;          // Requesting JVM to call Garbage Collector method          System.gc();          System.out.println("Main Completes");      }      // Here overriding finalize method  **public** **void** finalize()      {          System.out.println("finalize method overridden");      }  } |

**Output**:

Main Completes

***Note****: Here above output came only****Main Completes****and****not****“finalize method overridden” because Garbage Collector call finalize method on that class object which is eligible for Garbage collection. Here above we have done->****s = null****and ‘s’ is the object of String class, so String class finalize method is going to be called and not our class(i.e, Hello class). So we modify our code like->*

Hello s = new Hello();

s = null;

Now our class i.e, Hello class finalize method is called.

**Output**:

finalize method overridden

Main Completes

So basically, Garbage Collector calls finalize method on that class object which is eligible for Garbage collection. So if a String object is eligible for Garbage Collection, then **the String** class finalize method is going to be called and **not the Hello class** finalize method.

**Case 2:**We can call finalize method Explicitly then it will be executed just like a normal method call, but the object won’t be deleted/destroyed

|  |
| --- |
| **class** Bye {  **public** **static** **void** main(String[] args)      {          Bye m = **new** Bye();          // Calling finalize method Explicitly.          m.finalize();          m.finalize();          m = **null**;          // Requesting JVM to call Garbage Collector method          System.gc();          System.out.println("Main Completes");      }      // Here overriding finalize method  **public** **void** finalize()      { System.out.println("finalize method overridden"); }  } |

**Output**:

finalize method overridden

//call by programmer but object won't gets destroyed.

finalize method overridden

//call by programmer but object won't gets destroyed.

Main Completes

finalize method overridden

//call by Garbage Collector just before destroying the object.

***Note****: As finalize is a method and not a reserved keyword, so we can call finalize method****Explicitly****, then it will be executed just like normal method call, but object won’t be deleted/destroyed.*

**Case 3:**

**Part a)**If the programmer calls finalize method while executing finalize method, some unchecked exception rises.

|  |
| --- |
| **class** Hi {  **public** **static** **void** main(String[] args)      {          Hi j = **new** Hi();          // Calling finalize method Explicitly.          j.finalize();          j = **null**;          // Requesting JVM to call Garbage Collector method          System.gc();          System.out.println("Main Completes");      }      // Here overriding finalize method  **public** **void** finalize()      {          System.out.println("finalize method overridden");          System.out.println(10 / 0);      }  } |

**Output**:

exception in thread "main" java.lang.ArithmeticException:

/ by zero followed by stack trace.

So the **key point** is: If the programmer calls finalize method while executing finalize method some unchecked exception rises, then JVM terminates the program abnormally by rising an exception. So in this case, the program termination is **Abnormal**.

**Part b)**If the garbage Collector calls finalize method while executing finalize method, some unchecked exception rises.

|  |
| --- |
| **class** RR {  **public** **static** **void** main(String[] args)      {          RR q = **new** RR();          q = **null**;          // Requesting JVM to call Garbage Collector method          System.gc();          System.out.println("Main Completes");      }      // Here overriding finalize method  **public** **void** finalize()      {          System.out.println("finalize method overridden");          System.out.println(10 / 0);      }  } |

**Output**

Main Completes

finalize method overridden

So the **key point** is if Garbage Collector calls finalize method while executing finalize method some unchecked exception rises then JVM **ignores** that exception and the rest of the program will be continued normally. So in this case the program termination is **Normal** and not abnormal.

**Important points:**

* There is no guarantee about the time when finalize is called. It may be called any time after the object is not being referred anywhere (can be garbage collected).
* JVM does not ignore all exceptions while executing finalize method, but it ignores **only**[**unchecked exceptions**](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/). If the corresponding catch block is there, then JVM won’t ignore any corresponding catch block and will be executed.
* System.gc() is just a request to JVM to execute the Garbage Collector. It’s up to JVM to call Garbage Collector or not. Usually, JVM calls Garbage Collector when there is not enough space available in the Heap area or when the memory is low.

**CHAPTER – 3 Exceptions, File Handling & I/O Operations, Threads**

**Exception Handling** in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

## What are Java Exceptions?

**In Java, Exception** is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program’s instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

### **Major reasons why an exception Occurs**

* Invalid user input
* Device failure
* Loss of network connection
* Physical limitations (out-of-disk memory)
* Code errors
* Opening an unavailable file

**Errors** represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

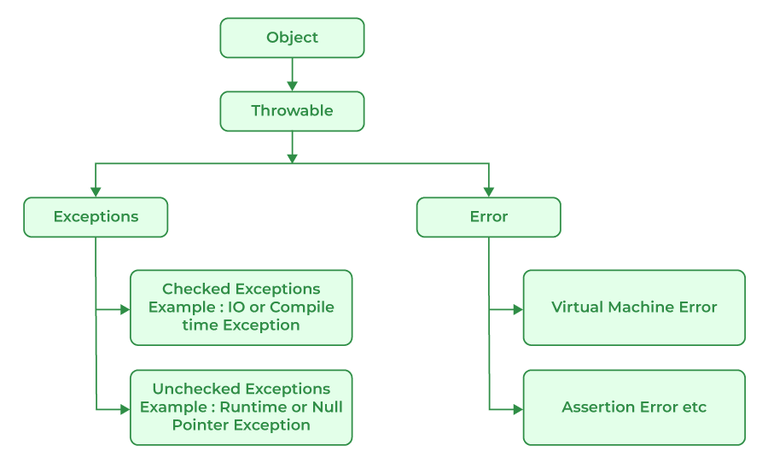
### Difference between Error and Exception

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

* **Error:**An Error indicates a serious problem that a reasonable application should not try to catch.
* **Exception:**Exception indicates conditions that a reasonable application might try to catch.

### Exception Hierarchy

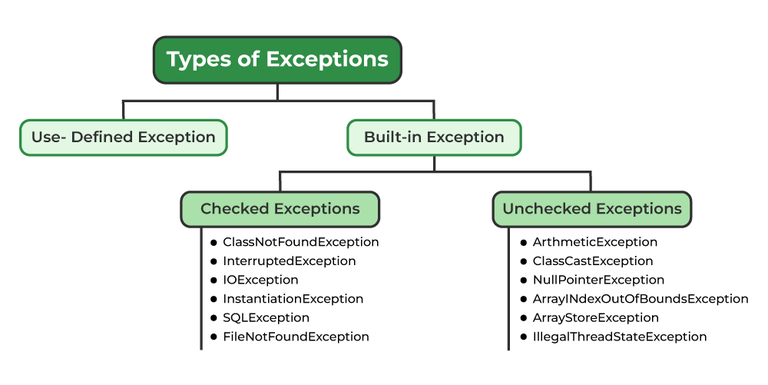
All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system([JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



*Java Exception Hierarchy*

## Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



**Exceptions can be categorized in two ways:**

1. **Built-in Exceptions**
   * Checked Exception
   * Unchecked Exception
2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

### **1. Built-in Exceptions**

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

* **Checked Exceptions:**Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
* **Unchecked Exceptions:**The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn’t handle or declare it, the program would not give a compilation error.

***Note:****For checked vs unchecked exception, see*[*Checked vs Unchecked Exceptions*](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/)

### **2. User-Defined Exceptions:**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called ‘user-defined Exceptions’.

The ***advantages of Exception Handling in Java***are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

**Methods to print the Exception information:**

#### **1. printStackTrace()**

This method prints exception information in the format of the Name of the exception: description of the exception, stack trace.

**Example:**

|  |
| --- |
| //program to print the exception information using printStackTrace() method  **import** java.io.\*;  **class** GFG {  **public** **static** **void** main (String[] args) {  **int** a=5;  **int** b=0;  **try**{            System.out.println(a/b);          }  **catch**(ArithmeticException e){          e.printStackTrace();        }      }  } |

**Output**

java.lang.ArithmeticException: / by zero  
at GFG.main(File.java:10)

#### **2. toString()**

The toString() method prints exception information in the format of the Name of the exception: description of the exception.

**Example:**

|  |
| --- |
| //program to print the exception information using toString() method  **import** java.io.\*;  **class** GFG1 {  **public** **static** **void** main (String[] args) {  **int** a=5;  **int** b=0;  **try**{            System.out.println(a/b);          }  **catch**(ArithmeticException e){          System.out.println(e.toString());        }      }  } |

**Output**

java.lang.ArithmeticException: / by zero

#### **3. getMessage()**

The getMessage() method prints only the description of the exception.

**Example:**

|  |
| --- |
| //program to print the exception information using getMessage() method  **import** java.io.\*;  **class** GFG1 {  **public** **static** **void** main (String[] args) {  **int** a=5;  **int** b=0;  **try**{            System.out.println(a/b);          }  **catch**(ArithmeticException e){          System.out.println(e.getMessage());        }      }  } |

**Output**

/ by zero

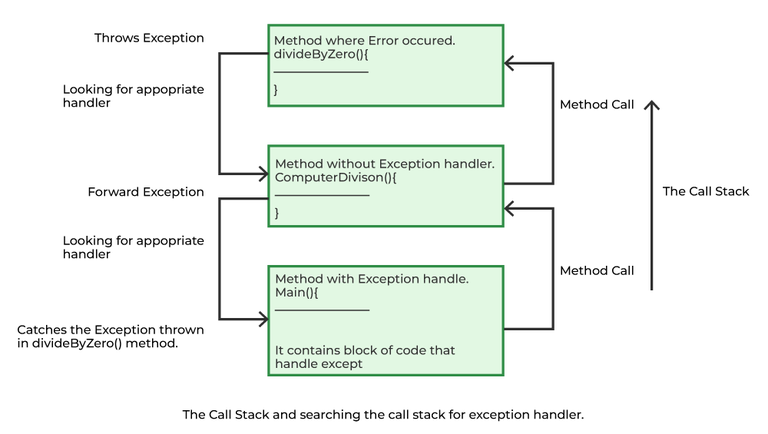
## How Does JVM Handle an Exception?

**Default Exception Handling:**Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of methods is called **Call Stack**. Now the following procedure will happen.

* The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.
* The run-time system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.
* If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object thrown matches the type of exception object it can handle.
* If the run-time system searches all the methods on the call stack and couldn’t have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program **abnormally**.

Exception in thread "xxx" Name of Exception : Description  
... ...... .. // Call Stack

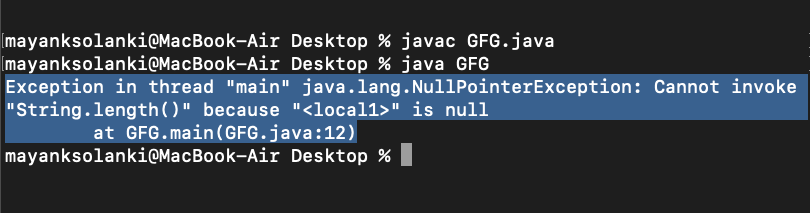
Look at the below diagram to understand the flow of the call stack.



**Illustration:**

|  |
| --- |
| // Java Program to Demonstrate How Exception Is Thrown  // Class  // ThrowsExecp  **class** GFG {      // Main driver method  **public** **static** **void** main(String args[])      {          // Taking an empty string          String str = **null**;          // Getting length of a string          System.out.println(str.length());      }  } |

**Output**



Let us see an example that illustrates how a run-time system searches for appropriate exception handling code on the call stack.

**Example:**

|  |
| --- |
| // Java Program to Demonstrate Exception is Thrown  // How the runTime System Searches Call-Stack  // to Find Appropriate Exception Handler  // Class  // ExceptionThrown  **class** GFG {      // Method 1      // It throws the Exception(ArithmeticException).      // Appropriate Exception handler is not found      // within this method.  **static** **int** divideByZero(**int** a, **int** b)      {            // this statement will cause ArithmeticException          // (/by zero)  **int** i = a / b;  **return** i;      }        // The runTime System searches the appropriate      // Exception handler in method also but couldn't have      // found. So looking forward on the call stack  **static** **int** computeDivision(**int** a, **int** b)      {  **int** res = 0;          // Try block to check for exceptions  **try** {              res = divideByZero(a, b);          }          // Catch block to handle NumberFormatException          // exception Doesn't matches with          // ArithmeticException  **catch** (NumberFormatException ex) {              // Display message when exception occurs              System.out.println(                  "NumberFormatException is occurred");          }  **return** res;      }        // Method 2      // Found appropriate Exception handler.      // i.e. matching catch block.  **public** **static** **void** main(String args[])      {  **int** a = 1;  **int** b = 0;          // Try block to check for exceptions  **try** {  **int** i = computeDivision(a, b);          }          // Catch block to handle ArithmeticException          // exceptions  **catch** (ArithmeticException ex) {              // getMessage() will print description              // of exception(here / by zero)              System.out.println(ex.getMessage());          }      }  } |

**Output**

/ by zero

## How Programmer Handle an Exception?

**Customized Exception Handling:**Java exception handling is managed via five keywords: **try**, **catch**, [**throw**](https://www.geeksforgeeks.org/throw-throws-java/), [**throws**](https://www.geeksforgeeks.org/throw-throws-java/), and **finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

***Tip:****One must go through*[*control flow in try catch finally block for better understanding.*](https://www.geeksforgeeks.org/flow-control-in-try-catch-finally-in-java/)

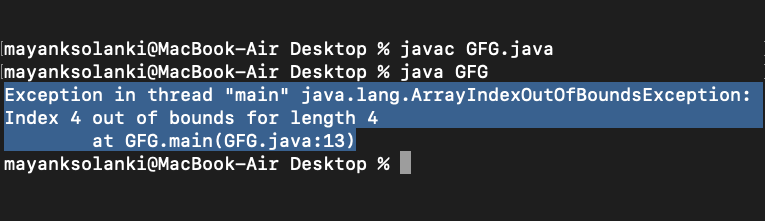
### Need for try-catch clause(Customized Exception Handling)

Consider the below program in order to get a better understanding of the try-catch clause.

**Example:**

|  |
| --- |
| // Java Program to Demonstrate  // Need of try-catch Clause  // Class  **class** GFG {      // Main driver method  **public** **static** **void** main(String[] args)      {          // Taking an array of size 4  **int**[] arr = **new** **int**[4];          // Now this statement will cause an exception  **int** i = arr[4];          // This statement will never execute          // as above we caught with an exception          System.out.println("Hi, I want to execute");      }  } |

**Output**



**Output explanation:**In the above example, an array is defined with size i.e. you can access elements only from index 0 to 3. But you trying to access the elements at index 4(by mistake) that’s why it is throwing an exception. In this case, JVM terminates the program**abnormally**. The statement System.out.println(“Hi, I want to execute”); will never execute. To execute it, we must handle the exception using try-catch. Hence to continue the normal flow of the program, we need a try-catch clause.

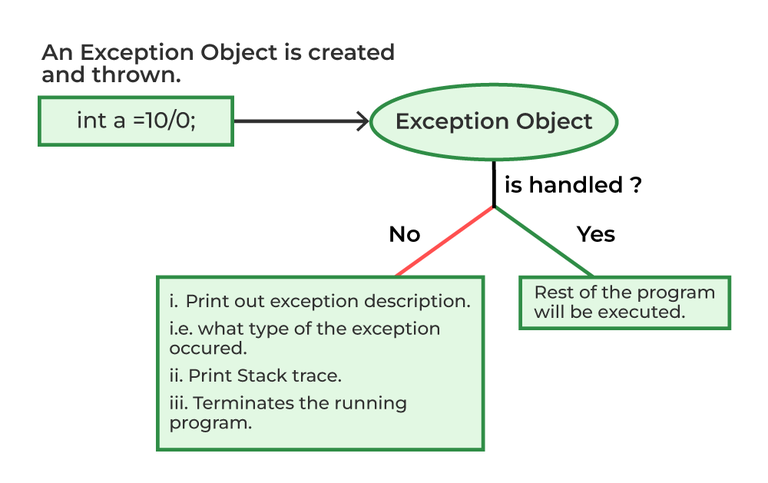
### **How to Use the Try-catch Clause?**

try {  
 // block of code to monitor for errors  
 // the code you think can raise an exception  
} catch (ExceptionType1 exOb) {  
 // exception handler for ExceptionType1  
} catch (ExceptionType2 exOb) {  
 // exception handler for ExceptionType2  
}  
// optional  
finally { // block of code to be executed after try block ends   
}

Certain key points need to be remembered that are as follows: 

* In a method, there can be more than one statement that might throw an exception, So put all these statements within their own **try** block and provide a separate exception handler within their own **catch** block for each of them.
* If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate the exception handler, we must put a **catch** block after it. There can be more than one exception handler. Each **catch** block is an exception handler that handles the exception to the type indicated by its argument. The argument, ExceptionType declares the type of exception that it can handle and must be the name of the class that inherits from the **Throwable** class.
* For each try block, there can be zero or more catch blocks, but **only one** final block.
* The finally block is optional. It always gets executed whether an exception occurred in try block or not. If an exception occurs, then it will be executed after **try and catch blocks.** And if an exception does not occur, then it will be executed after the **try** block. The finally block in Java is used to put important codes such as clean-up code e.g., closing the file or closing the connection.
* If we write System.exit in the try block, then finally block will not be executed.

The summary is depicted via visual aid below as follows:



# Types of Exception

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



### [**Built-in Exceptions**](https://www.geeksforgeeks.org/built-exceptions-java-examples/)**:**

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException:**It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException:**This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:**This Exception is raised when a file is not accessible or does not open.
5. **IOException:**It is thrown when an input-output operation failed or interrupted
6. **InterruptedException:**It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:**It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:**It is thrown when accessing a method that is not found.
9. **NullPointerException:**This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException:**This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:**This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:**It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :**This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :**This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

### **Examples of Built-in Exception**

**A. Arithmetic exception**

|  |
| --- |
| **class** ArithmeticException\_Demo  {  **public** **static** **void** main(String args[])      {  **try** {  **int** a = 30, b = 0;  **int** c = a/b;  // cannot divide by zero              System.out.println ("Result = " + c);          }  **catch**(ArithmeticException e) {              System.out.println ("Can't divide a number by 0");          }      }  } |

**Output**

Can't divide a number by 0

**B. NullPointer Exception**

|  |
| --- |
| **class** NullPointer\_Demo  {  **public** **static** **void** main(String args[])      {  **try** {              String a = **null**; //null value              System.out.println(a.charAt(0));          } **catch**(NullPointerException e) {              System.out.println("NullPointerException..");          }      }  } |

**Output**

NullPointerException..

**C. StringIndexOutOfBound Exception**

|  |
| --- |
| **class** StringIndexOutOfBound\_Demo  {  **public** **static** **void** main(String args[])      {  **try** {              String a = "This is like chipping "; // length is 22  **char** c = a.charAt(24); // accessing 25th element              System.out.println(c);          }  **catch**(StringIndexOutOfBoundsException e) {              System.out.println("StringIndexOutOfBoundsException");          }      }  } |

**Output**

StringIndexOutOfBoundsException

**D. FileNotFound Exception**

|  |
| --- |
| **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.io.FileReader;  **class** File\_notFound\_Demo {    **public** **static** **void** main(String args[])  {  **try** {              // Following file does not exist              File file = **new** File("[E://file.txt](file:///E:\file.txt)");              FileReader fr = **new** FileReader(file);          } **catch** (FileNotFoundException e) {             System.out.println("File does not exist");          }      }  } |

**Output:**

File does not exist

**E. NumberFormat Exception**

|  |
| --- |
| **class**  NumberFormat\_Demo  {  **public** **static** **void** main(String args[])      {  **try** {              // "akki" is not a number  **int** num = Integer.parseInt ("akki") ;                System.out.println(num);          } **catch**(NumberFormatException e) {              System.out.println("Number format exception");          }      }  } |

**Output**

Number format exception

**F. ArrayIndexOutOfBounds Exception**

|  |
| --- |
| **class** ArrayIndexOutOfBound\_Demo  {  **public** **static** **void** main(String args[])      {  **try**{  **int** a[] = **new** **int**[5];              a[6] = 9; // accessing 7th element in an array of                        // size 5          }  **catch**(ArrayIndexOutOfBoundsException e){              System.out.println ("Array Index is Out Of Bounds");          }      }  } |

**Output**

Array Index is Out Of Bounds

**G. IO Exception**

|  |
| --- |
| **class** IOException\_Demo {    **public** **static** **void** main(String[] args)      {            // Create a new scanner with the specified String          // Object          Scanner scan = **new** Scanner("Hello Geek!");            // Print the line          System.out.println("" + scan.nextLine());            // Check if there is an IO exception          System.out.println("Exception Output: "                             + scan.ioException());            scan.close();      }  } |

**Output:**

Hello Geek!

Exception Output: null

**H. NoSuchMethod Exception**

|  |
| --- |
| **public** **class** NoSuchElementException\_Demo {    **public** **static** **void** main(String[] args)      {            Set exampleleSet = **new** HashSet();            Hashtable exampleTable = **new** Hashtable();            exampleleSet.iterator().next();            //accessing Set            exampleTable.elements().nextElement();            //accessing Hashtable              // This throws a NoSuchElementException as there are          // no elements in Set and HashTable and we are          // trying to access elements      }  } |

**I. IllegalArgumentException:**This program, checks whether the person is eligible for voting or not. If the age is greater than or equal to 18 then it will not throw any error. If the age is less than 18 then it will throw an error with the error statement.

Also, we can specify “throw new IllegalArgumentException()” without the error message. We can also specify Integer.toString(variable\_name) inside the IllegalArgumentException() and It will print the argument name which is not satisfied the given condition.

|  |
| --- |
| **import** java.io.\*;    **class** GFG {  **public** **static** **void** print(**int** a)      {  **if**(a>=18){            System.out.println("Eligible for Voting");            }  **else**{    **throw** **new** IllegalArgumentException("Not Eligible for Voting");              }        }  **public** **static** **void** main(String[] args) {           GFG.print(14);      }  } |

**Output :**

Exception in thread "main" java.lang.IllegalArgumentException: Not Eligible for Voting

at GFG.print(File.java:13)

at GFG.main(File.java:19)

**J. IllegalStateException:**This program, displays the addition of numbers only for Positive integers. If both the numbers are positive then only it will call the print method to print the result otherwise it will throw the IllegalStateException with an error statement. Here, the method is not accessible for non-positive integers.

Also, we can specify the “throw new IllegalStateException()” without the error statement.

|  |
| --- |
| **import** java.io.\*;    **class** GFG {  **public** **static** **void**  print(**int** a,**int** b)       {           System.out.println("Addition of Positive Integers :"+(a+b));       }    **public** **static** **void** main(String[] args) {  **int** n1=7;  **int** n2=-3;  **if**(n1>=0 && n2>=0)       {           GFG.print(n1,n2);       }  **else**       {  **throw** **new** IllegalStateException("Either one or two numbers are not Positive Integer");       }      }  } |

**Output :**

Exception in thread "main" java.lang.IllegalStateException: Either one or two numbers are not Positive Integer

at GFG.main(File.java:20)

**k. ClassNotFound Exception :**

|  |
| --- |
| **public** **class** ClassNotFoundException\_Demo  {  **public** **static** **void** main(String[] args) {  **try**{              Class.forName("Class1");   // Class1 is not defined          }  **catch**(ClassNotFoundException e){              System.out.println(e);              System.out.println("Class Not Found...");          }      }  } |

**Output**

java.lang.ClassNotFoundException: Class1

Class Not Found...

### **User-Defined Exceptions**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, the user can also create exceptions which are called ‘user-defined Exceptions’.

The following steps are followed for the creation of a user-defined Exception.

* The user should create an exception class as a subclass of the Exception class. Since all the exceptions are subclasses of the Exception class, the user should also make his class a subclass of it. This is done as:

class MyException extends Exception

* We can write a default constructor in his own exception class.

MyException(){}

* We can also create a parameterized constructor with a string as a parameter.   
  We can use this to store exception details. We can call the superclass(Exception) constructor from this and send the string there.

MyException(String str)

{

super(str);

}

* To raise an exception of a user-defined type, we need to create an object to his exception class and throw it using the throw clause, as:

MyException me = new MyException(“Exception details”);

throw me;

* The following program illustrates how to create your own exception class MyException.
* Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
* In main() method, the details are displayed using a for-loop. At this time, a check is done if in any account the balance amount is less than the minimum balance amount to be apt in the account.
* If it is so, then MyException is raised and a message is displayed “Balance amount is less”.

**Example**

|  |
| --- |
| **class** MyException **extends** Exception  {      //store account information  **private** **static** **int** accno[] = {1001, 1002, 1003, 1004};    **private** **static** String name[] =                   {"Nish", "Shubh", "Sush", "Abhi", "Akash"};    **private** **static** **double** bal[] =           {10000.00, 12000.00, 5600.0, 999.00, 1100.55};        // default constructor      MyException() {    }        // parameterized constructor      MyException(String str) { **super**(str); }        // write main()  **public** **static** **void** main(String[] args)      {  **try**  {              // display the heading for the table              System.out.println("ACCNO" + "\t" + "CUSTOMER" +                                             "\t" + "BALANCE");                // display the actual account information  **for** (**int** i = 0; i < 5 ; i++)              {                  System.out.println(accno[i] + "\t" + name[i] +                                                 "\t" + bal[i]);                    // display own exception if balance < 1000  **if** (bal[i] < 1000)                  {                      MyException me =  **new** MyException("Balance is less than 1000");  **throw** me;                  }              }          } //end of try    **catch** (MyException e) {              e.printStackTrace();          }      }  } |

Runtime Error

MyException: Balance is less than 1000

at MyException.main(fileProperty.java:36)

**Output:**

ACCNO CUSTOMER BALANCE

1001 Nish 10000.0

1002 Shubh 12000.0

1003 Sush 5600.0

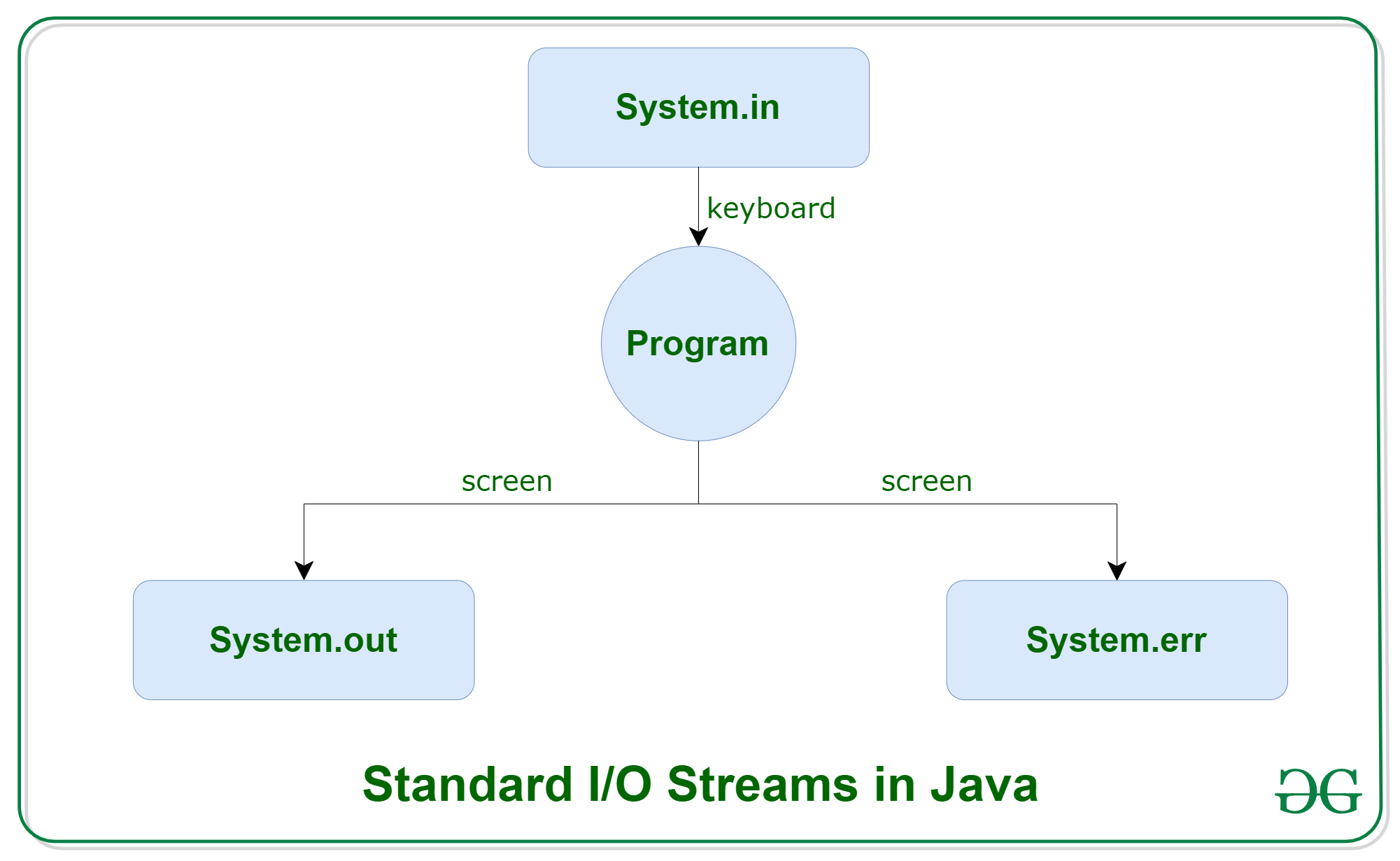
1004 Abhi 999.0

# Java IO : Input-output in Java

[Java](https://www.geeksforgeeks.org/java/) brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.



Before exploring various input and output streams lets look at **3 standard or default streams** that Java has to provide which are also most common in use:



1. [System.in](https://www.geeksforgeeks.org/java-lang-system-class-java/)**:** This is the **standard input stream** that is used to read characters from the keyboard or any other standard input device.
2. [System.out](https://www.geeksforgeeks.org/java-lang-system-class-java/)**:** This is the **standard output stream** that is used to produce the result of a program on an output device like the computer screen.

Here is a list of the various print functions that we use to output statements:

* + [print()](https://www.geeksforgeeks.org/difference-between-print-and-println-in-java/)**:** This method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here.  
    **Syntax:**

System.out.print(parameter);

**Example:**

|  |
| --- |
| // Java code to illustrate print()  **import** java.io.\*;    **class** Demo\_print {  **public** **static** **void** main(String[] args)      {            // using print()          // all are printed in the          // same line          System.out.print("GfG! ");          System.out.print("GfG! ");          System.out.print("GfG! ");      }  } |

**Output:**

GfG! GfG! GfG!

* + [println()](https://www.geeksforgeeks.org/difference-between-print-and-println-in-java/)**:** This method in Java is also used to display a text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.  
    **Syntax:**

System.out.println(parameter);

**Example:**

|  |
| --- |
| // Java code to illustrate println()  **import** java.io.\*;  **class** Demo\_print {  **public** **static** **void** main(String[] args)      {          // using println()          // all are printed in the          // different line          System.out.println("GfG! ");          System.out.println("GfG! ");          System.out.println("GfG! ");      }  } |

**Output:**

GfG!

GfG!

GfG!

* + [printf()](https://www.geeksforgeeks.org/formatted-output-in-java/)**:** This is the easiest of all methods as this is similar to printf in C. Note that System.out.print() and System.out.println() take a single argument, but printf() may take multiple arguments. This is used to format the output in Java.  
    **Example:**

|  |
| --- |
| // A Java program to demonstrate working of printf() in Java  **class** JavaFormatter1 {  **public** **static** **void** main(String args[])      {  **int** x = 100;          System.out.printf(              "Printing simple"                  + " integer: x = %d\n",              x);            // this will print it upto          // 2 decimal places          System.out.printf(              "Formatted with"                  + " precision: PI = %.2f\n",              Math.PI);    **float** n = 5.2f;            // automatically appends zero          // to the rightmost part of decimal          System.out.printf(              "Formatted to "                  + "specific width: n = %.4f\n",              n);            n = 2324435.3f;            // here number is formatted from          // right margin and occupies a          // width of 20 characters          System.out.printf(              "Formatted to "                  + "right margin: n = %20.4f\n",              n);      }  } |

* + **Output:**
  + Printing simple integer: x = 100
  + Formatted with precision: PI = 3.14
  + Formatted to specific width: n = 5.2000
  + Formatted to right margin: n = 2324435.2500

1. [System.err](https://www.geeksforgeeks.org/java-lang-system-class-java/)**:** This is the **standard error stream** that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

This stream also uses all the 3 above-mentioned functions to output the error data:

* + print()
  + println()
  + printf()

**Example:**

|  |
| --- |
| // Java code to illustrate standard  // input output streams    **import** java.io.\*;  **public** **class** SimpleIO {    **public** **static** **void** main(String args[])  **throws** IOException      {            // InputStreamReader class to read input          InputStreamReader inp = **null**;            // Storing the input in inp          inp = **new** InputStreamReader(System.in);            System.out.println("Enter characters, "                             + " and '0' to quit.");  **char** c;  **do** {              c = (**char**)inp.read();              System.out.println(c);          } **while** (c != '0');      }  } |

**Input:**

GeeksforGeeks0

**Output:**

Enter characters, and '0' to quit.

G

e

e

k

s

f

o

r

G

e

e

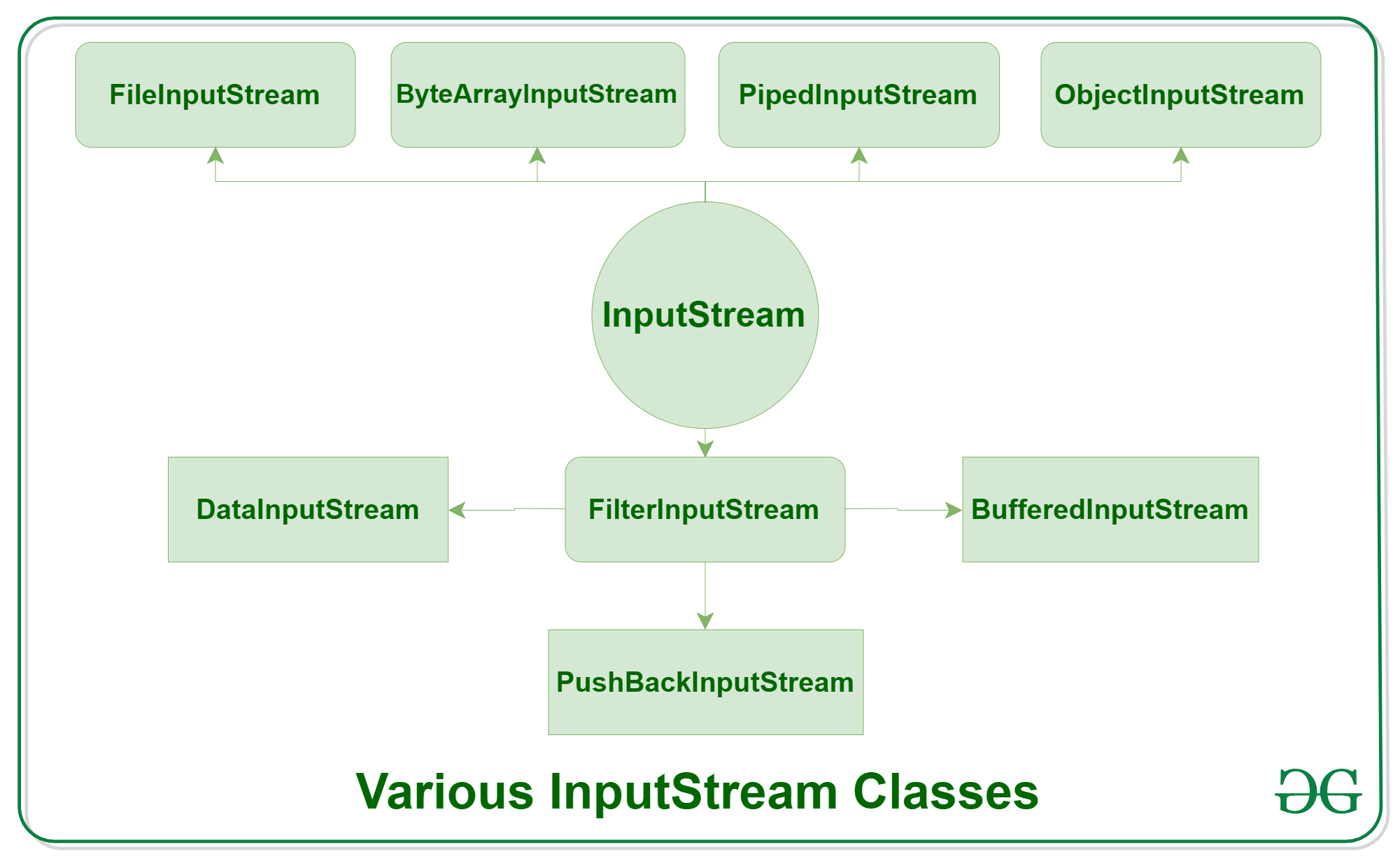
k

s

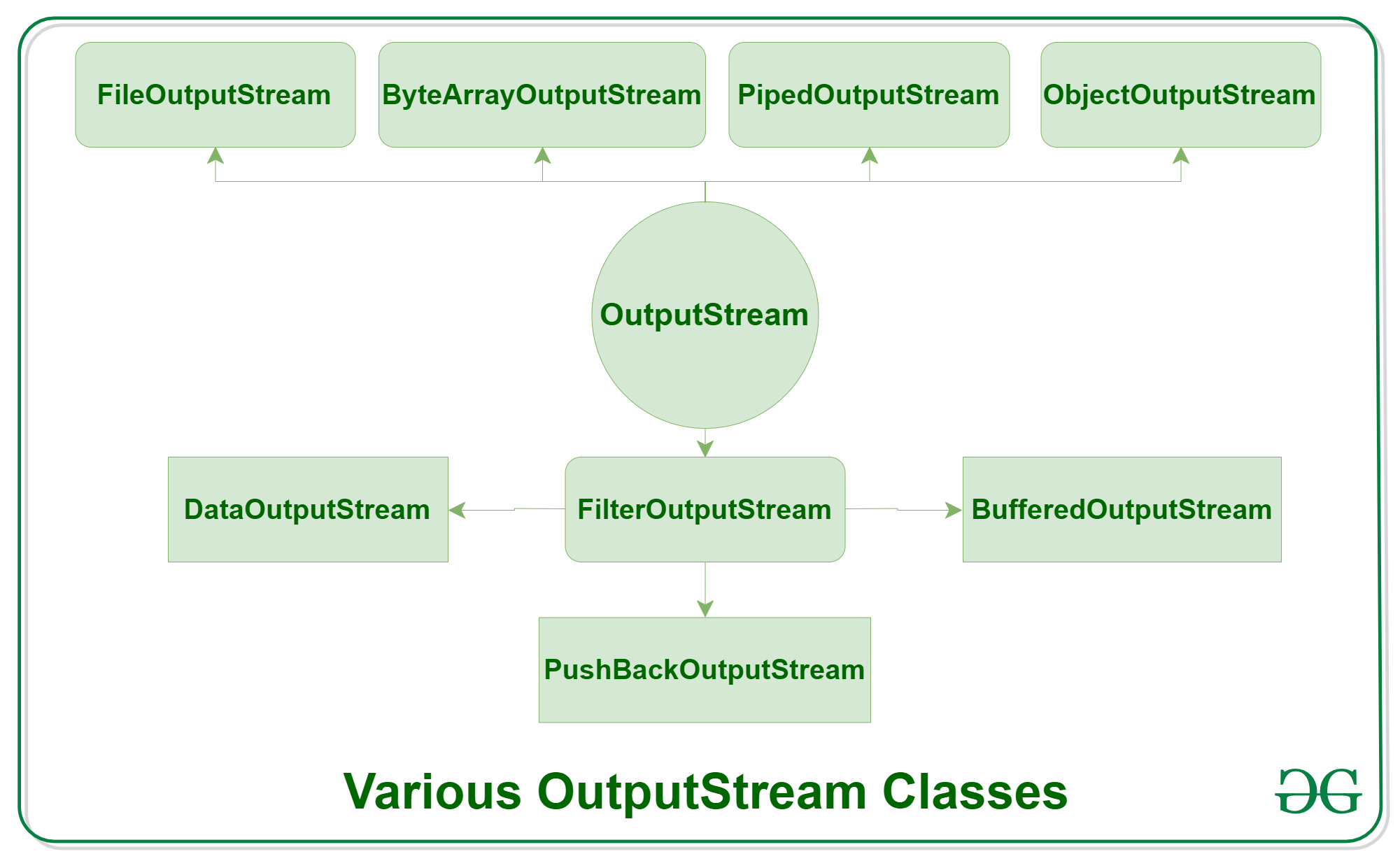
0

**Types of Streams:**

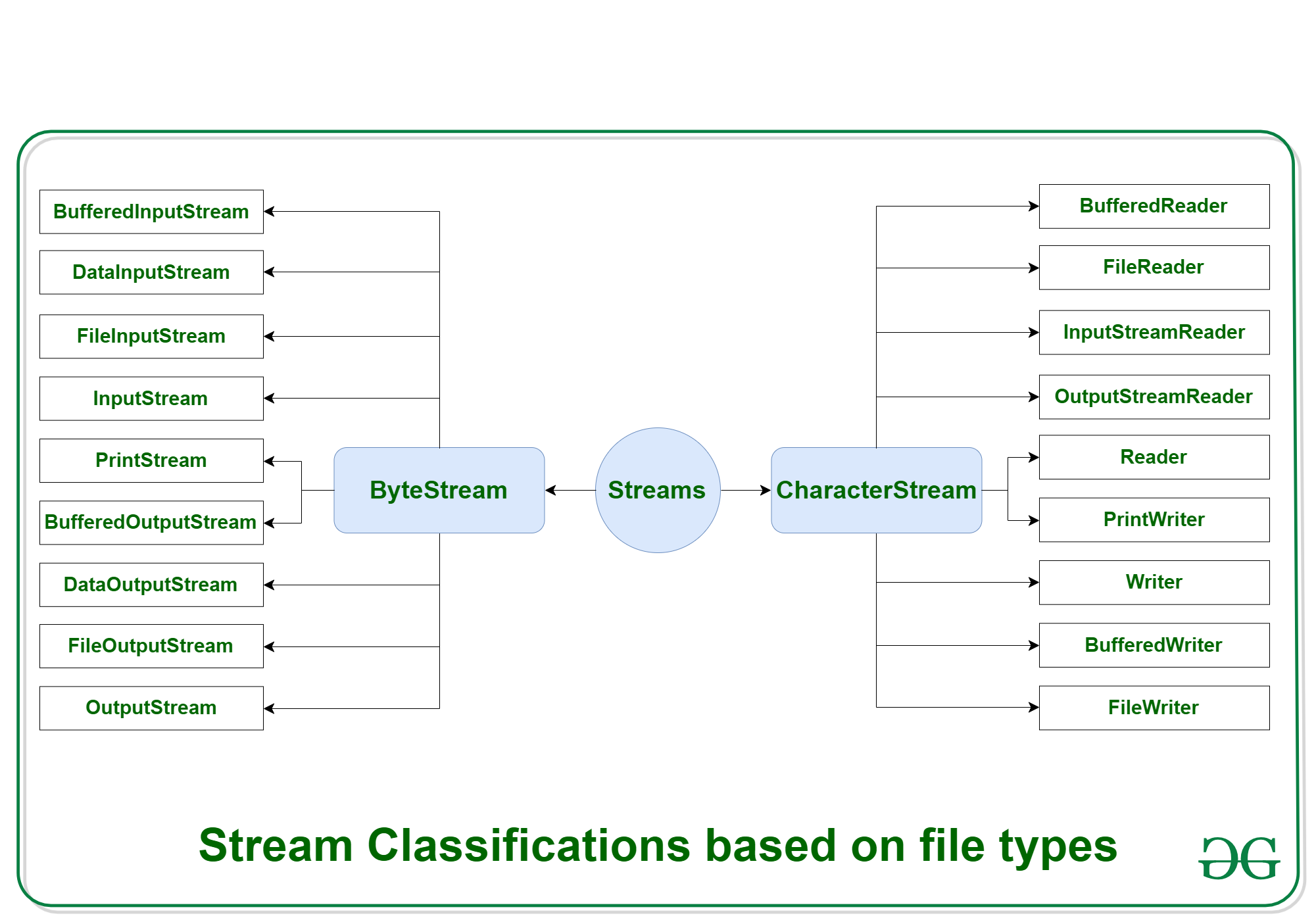
* **Depending on the type of operations**, streams can be divided into two primary classes:
* [Input Stream:](https://www.geeksforgeeks.org/java-io-inputstream-class-in-java/) These streams are used to read data that must be taken as an input from a source array or file or any peripheral device. For eg., FileInputStream,ufferedInputStream,ByteArrayInputSStream etc.



* [Output Stream:](https://www.geeksforgeeks.org/java-io-outputstream-class-java/) These streams are used to write data as outputs into an array or file or any output peripheral device. For eg., FileOutputStream, BufferedOutputStream, ByteArrayOutputStream etc.



* **Depending on the types of file**, Streams can be divided into two primary classes which can be further divided into other classes as can be seen through the diagram below followed by the explanations.



* 1. **ByteStream:** This is used to process data byte by byte (8 bits). Though it has many classes, the FileInputStream and the FileOutputStream are the most popular ones. The FileInputStream is used to read from the source and FileOutputStream is used to write to the destination. Here is the list of various ByteStream Classes:

| **Stream class** | **Description** |
| --- | --- |
| [BufferedInputStream](https://www.geeksforgeeks.org/java-io-bufferedinputstream-class-java/) | It is used for Buffered Input Stream. |
| [DataInputStream](https://www.geeksforgeeks.org/java-io-datainputstream-class-java-set-1/) | It contains method for reading java standard datatypes. |
| [FileInputStream](https://www.geeksforgeeks.org/java-io-fileinputstream-class-java/) | This is used to reads from a file |
| [InputStream](https://www.geeksforgeeks.org/java-io-inputstream-class-in-java/) | This is an abstract class that describes stream input. |
| [PrintStream](https://www.geeksforgeeks.org/java-io-printstream-class-java-set-1/) | This contains the most used print() and println() method |
| [BufferedOutputStream](https://www.geeksforgeeks.org/java-io-bufferedoutputstream-class-java/) | This is used for Buffered Output Stream. |
| [DataOutputStream](https://www.geeksforgeeks.org/dataoutputstream-in-java/) | This contains method for writing java standard data types. |
| [FileOutputStream](https://www.geeksforgeeks.org/creating-a-file-using-fileoutputstream/) | This is used to write to a file. |
| [OutputStream](https://www.geeksforgeeks.org/java-io-outputstream-class-java/) | This is an abstract class that describe stream output. |

* 1. **Example:**

|  |
| --- |
| // Java Program illustrating the  // Byte Stream to copy  // contents of one file to another file.  **import** java.io.\*;  **public** **class** BStream {  **public** **static** **void** main(          String[] args) **throws** IOException      {            FileInputStream sourceStream = **null**;          FileOutputStream targetStream = **null**;    **try** {              sourceStream                  = **new** FileInputStream("sorcefile.txt");              targetStream                  = **new** FileOutputStream("targetfile.txt");                // Reading source file and writing              // content to target file byte by byte  **int** temp;  **while** ((                         temp = sourceStream.read())                     != -1)                  targetStream.write((**byte**)temp);          }  **finally** {  **if** (sourceStream != **null**)                  sourceStream.close();  **if** (targetStream != **null**)                  targetStream.close();          }      }  } |

* 1. **Output:**
  2. Shows contents of file test.txt
  3. **CharacterStream:** In Java, characters are stored using Unicode conventions (Refer this for details). Character stream automatically allows us to read/write data character by character. Though it has many classes, the FileReader and the FileWriter are the most popular ones. FileReader and FileWriter are character streams used to read from the source and write to the destination respectively. Here is the list of various CharacterStream Classes:

| **Stream class** | **Description** |
| --- | --- |
| [BufferedReader](https://www.geeksforgeeks.org/java-io-bufferedreader-class-java/) | It is used to handle buffered input stream. |
| [FileReader](https://www.geeksforgeeks.org/file-handling-java-using-filewriter-filereader/) | This is an input stream that reads from file. |
| [InputStreamReader](https://www.geeksforgeeks.org/java-io-inputstreamreader-class/) | This input stream is used to translate byte to character. |
| OutputStreamReader | This output stream is used to translate character to byte. |
| [Reader](https://www.geeksforgeeks.org/java-io-reader-class-java/) | This is an abstract class that define character stream input. |
| [PrintWriter](https://www.geeksforgeeks.org/java-io-printwriter-class-java-set-1/) | This contains the most used print() and println() method |
| [Writer](https://www.geeksforgeeks.org/java-io-writer-class-java/) | This is an abstract class that define character stream output. |
| [BufferedWriter](https://www.geeksforgeeks.org/io-bufferedwriter-class-methods-java/) | This is used to handle buffered output stream. |
| [FileWriter](https://www.geeksforgeeks.org/file-handling-java-using-filewriter-filereader/) | This is used to output stream that writes to file. |
|  |  |

* 1. **Example:**

|  |
| --- |
| // Java Program illustrating that  // we can read a file in a human-readable  // format using FileReader    // Accessing FileReader, FileWriter,  // and IOException  **import** java.io.\*;  **public** **class** GfG {  **public** **static** **void** main(          String[] args) **throws** IOException      {          FileReader sourceStream = **null**;  **try** {              sourceStream                  = **new** FileReader("test.txt");                // Reading sourcefile and              // writing content to target file              // character by character.  **int** temp;  **while** ((                         temp = sourceStream.read())                     != -1)                  System.out.println((**char**)temp);          }  **finally** {              // Closing stream as no longer in use  **if** (sourceStream != **null**)                  sourceStream.close();          }      }  } |

# Java.io.File Class in Java

Java File class is Java’s representation of a file or directory pathname. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. Java File class contains several methods for working with the pathname, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

* It is an abstract representation of files and directory pathnames.
* A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
* First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
* Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

### **How to Create a File Object?**

A File object is created by passing in a string that represents the name of a file, a String, or another File object. For example,

File a = new File("/usr/local/bin/geeks");

This defines an abstract file name for the geeks file in the directory /usr/local/bin. This is an absolute abstract file name.

### Fields in File Class in Java

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| **pathSeperator** | String | the character or string used to separate individual paths in a list of file system paths. |
| **pathSeperatorChar** | Char | the character used to separate individual paths in a list of file system paths. |
| **separator** | String | default name separator character represented as a string. |
| **separatorChar** | Char | default name separator character. |

### Constructors of Java File Class

* **File(File parent, String child):**Creates a new File instance from a parent abstract pathname and a child pathname string.
* **File(String pathname):**Creates a new File instance by converting the given pathname string into an abstract pathname.
* **File(String parent, String child):**Creates a new File instance from a parent pathname string and a child pathname string.
* **File(URI uri):**Creates a new File instance by converting the given file: URI into an abstract pathname.

### Methodsof File Class in Java

| **S. No.** | **Method** | **Description** | **Return Type** |
| --- | --- | --- | --- |
| 1. | [**canExecute()**](https://www.geeksforgeeks.org/file-canexecute-method-in-java-with-examples/) | Tests whether the application can execute the file denoted by this abstract pathname. | boolean |
| 2. | [**canRead()**](https://www.geeksforgeeks.org/file-canread-method-in-java-with-examples/) | Tests whether the application can read the file denoted by this abstract pathname. | boolean |
| 3. | [**canWrite()**](https://www.geeksforgeeks.org/file-canwrite-method-in-java-with-examples/) | Tests whether the application can modify the file denoted by this abstract pathname. | boolean |
| 4. | **compareTo(File pathname)** | Compares two abstract pathnames lexicographically. | int |
| 5. | [**createNewFile()**](https://www.geeksforgeeks.org/file-createnewfile-method-in-java-with-examples/) | Atomically creates a new, empty file named by this abstract pathname. | boolean |
| 6. | [**createTempFile(String prefix, String suffix)**](https://www.geeksforgeeks.org/file-createtempfile-method-in-java-with-examples/) | Creates an empty file in the default temporary-file directory. | File |
| 7. | [**delete()**](https://www.geeksforgeeks.org/files-delete-method-in-java-with-examples/) | Deletes the file or directory denoted by this abstract pathname. | boolean |
| 8. | **equals(Object obj)** | Tests this abstract pathname for equality with the given object. | boolean |
| 9. | [**exists()**](https://www.geeksforgeeks.org/file-exists-method-in-java-with-examples/) | Tests whether the file or directory denoted by this abstract pathname exists. | boolean |
| 10. | [**getAbsolutePath()**](https://www.geeksforgeeks.org/file-getabsolutepath-method-in-java-with-examples/) | Returns the absolute pathname string of this abstract pathname. | String |
| 11. | [**list()**](https://www.geeksforgeeks.org/file-list-method-in-java-with-examples/) | Returns an array of strings naming the files and directories in the directory. | String[] |
| 12. | [**getFreeSpace()**](https://www.geeksforgeeks.org/file-getfreespace-method-in-java-with-examples/) | Returns the number of unallocated bytes in the partition. | long |
| 13. | [**getName()**](https://www.geeksforgeeks.org/file-getname-method-in-java-with-examples/) | Returns the name of the file or directory denoted by this abstract pathname. | String |
| 14. | [**getParent()**](https://www.geeksforgeeks.org/file-getparent-method-in-java-with-examples/) | Returns the pathname string of this abstract pathname’s parent. | String |
| 15. | [**getParentFile()**](https://www.geeksforgeeks.org/file-getparentfile-method-in-java-with-examples/) | Returns the abstract pathname of this abstract pathname’s parent. | File |
| 16. | [**getPath()**](https://www.geeksforgeeks.org/file-getpath-method-in-java-with-examples/) | Converts this abstract pathname into a pathname string. | String |
| 17. | [**setReadOnly()**](https://www.geeksforgeeks.org/file-setreadonly-method-in-java-with-examples/) | Marks the file or directory named so that only read operations are allowed. | boolean |
| 18. | [**isDirectory()**](https://www.geeksforgeeks.org/file-isdirectory-method-in-java-with-examples/) | Tests whether the file denoted by this pathname is a directory. | boolean |
| 19. | [**isFile()**](https://www.geeksforgeeks.org/file-isfile-method-in-java-with-examples/) | Tests whether the file denoted by this abstract pathname is a normal file. | boolean |
| 20. | [**isHidden()**](https://www.geeksforgeeks.org/file-ishidden-method-in-java-with-examples/) | Tests whether the file named by this abstract pathname is a hidden file. | boolean |
| 21. | [**length()**](https://www.geeksforgeeks.org/file-length-method-in-java-with-examples/) | Returns the length of the file denoted by this abstract pathname. | long |
| 22. | [**listFiles()**](https://www.geeksforgeeks.org/file-listfiles-method-in-java-with-examples/) | Returns an array of abstract pathnames denoting the files in the directory. | File[] |
| 23. | [**mkdir()**](https://www.geeksforgeeks.org/file-mkdir-method-in-java-with-examples/) | Creates the directory named by this abstract pathname. | boolean |
| 24. | [**renameTo(File dest)**](https://www.geeksforgeeks.org/file-renameto-method-in-java-with-examples/) | Renames the file denoted by this abstract pathname. | boolean |
| 25. | [**setExecutable(boolean executable)**](https://www.geeksforgeeks.org/file-setexecutable-method-in-java-with-examples/) | A convenience method to set the owner’s execute permission. | boolean |
| 26. | [**setReadable(boolean readable)**](https://www.geeksforgeeks.org/file-setreadable-function-in-java-with-examples/) | A convenience method to set the owner’s read permission. | boolean |
| 27. | [**setReadable(boolean readable, boolean ownerOnly)**](https://www.geeksforgeeks.org/file-setreadable-function-in-java-with-examples/) | Sets the owner’s or everybody’s read permission. | boolean |
| 28. | [**setWritable(boolean writable)**](https://www.geeksforgeeks.org/file-setwritable-method-in-java-with-examples/) | A convenience method to set the owner’s write permission. | boolean |
| 29. | **toString()** | Returns the pathname string of this abstract pathname. | String |
| 30. | **toURI()** | Constructs a file URI that represents this abstract pathname. | URI |

### Java File Class Examples

**Example 1:**Program to check if a file or directory physically exists or not.

|  |
| --- |
| **import** java.io.File;    // Displaying file property  **class** fileProperty {  **public** **static** **void** main(String[] args)      {            // accept file name or directory name through          // command line args          String fname = args[0];            // pass the filename or directory name to File          // object          File f = **new** File(fname);            // apply File class methods on File object          System.out.println("File name :" + f.getName());          System.out.println("Path: " + f.getPath());          System.out.println("Absolute path:"                             + f.getAbsolutePath());          System.out.println("Parent:" + f.getParent());          System.out.println("Exists :" + f.exists());    **if** (f.exists()) {              System.out.println("Is writable:"                                 + f.canWrite());              System.out.println("Is readable" + f.canRead());              System.out.println("Is a directory:"                                 + f.isDirectory());              System.out.println("File Size in bytes "                                 + f.length());          }      }  } |

**Output**

File name :file.txt

Path: file.txt

Absolute path:C:\Users\akki\IdeaProjects\codewriting\src\file.txt

Parent:null

Exists :true

Is writable:true

Is readabletrue

Is a directory:false

File Size in bytes 20

**Example 2:**Program to display all the contents of a directory

Here we will accept a directory name from the keyboard and then display all the contents of the directory. For this purpose, list() method can be used as:

String arr[]=f.list();

In the preceding statement, the list() method causes all the directory entries copied into the array *arr[]*. Then pass these array elements arr[i] to the File object and test them to know if they represent a file or directory.

|  |
| --- |
| **import** java.io.BufferedReader;  **import** java.io.File;  **import** java.io.IOException;  **import** java.io.InputStreamReader;    // Displaying the contents of a directory  **class** Contents {  **public** **static** **void** main(String[] args)  **throws** IOException      {          // enter the path and dirname from keyboard          BufferedReader br = **new** BufferedReader(  **new** InputStreamReader(System.in));            System.out.println("Enter dirpath:");          String dirpath = br.readLine();          System.out.println("Enter the dirname");          String dname = br.readLine();            // create File object with dirpath and dname          File f = **new** File(dirpath, dname);            // if directory exists,then  **if** (f.exists()) {              // get the contents into arr[]              // now arr[i] represent either a File or              // Directory              String arr[] = f.list();                // find no. of entries in the directory  **int** n = arr.length;                // displaying the entries  **for** (**int** i = 0; i < n; i++) {                  System.out.println(arr[i]);                  // create File object with the entry and                  // test if it is a file or directory                  File f1 = **new** File(arr[i]);  **if** (f1.isFile())                      System.out.println(": is a file");  **if** (f1.isDirectory())                      System.out.println(": is a directory");              }              System.out.println(                  "No of entries in this directory " + n);          }  **else**              System.out.println("Directory not found");      }  } |

**Output**

Enter dirpath:

C:\Users\akki\IdeaProjects\

Enter the dirname

codewriting

.idea

: is a directory

an1.txt

: is a file

codewriting.iml

: is a file

file.txt

: is a file

out

: is a directory

src

: is a directory

text

: is a file

No of entries in this directory 7

# FileWriter and FileReader

Java FileWriter and FileReader classes are used to write and read data from text files (they are [Character Stream](https://www.geeksforgeeks.org/character-stream-vs-byte-stream-java/) classes). It is recommended **not** to use the FileInputStream and FileOutputStream classes if you have to read and write any textual information as these are Byte stream classes.

**FileWriter**  
FileWriter is useful to create a file writing characters into it.

* This class inherits from the OutputStream class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
* FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
* FileWriter creates the output file if it is not present already.

**Constructors:**

* **FileWriter(File file) –** Constructs a FileWriter object given a File object.
* **FileWriter (File file, boolean append) –** constructs a FileWriter object given a File object.
* **FileWriter (FileDescriptor fd) –** constructs a FileWriter object associated with a file descriptor.
* **FileWriter (String fileName) –** constructs a FileWriter object given a file name.
* **FileWriter (String fileName, Boolean append) –** Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

**Methods:**

* **public void write (int c) throws IOException –** Writes a single character.
* **public void write (char [] stir) throws IOException –** Writes an array of characters.
* **public void write(String str)throws IOException –** Writes a string.
* **public void write(String str,** **int off,** **int len)throws IOException –** Writes a portion of a string. Here off is offset from which to start writing characters and len is the number of characters to write.
* **public void flush() throws IOException** flushes the stream
* **public void close() throws IOException** flushes the stream first and then closes the writer.

Reading and writing take place character by character, which increases the number of I/O operations and affects the performance of the system.**BufferedWriter** can be used along with FileWriter to improve the speed of execution.  
The following program depicts how to create a text file using FileWriter

|  |
| --- |
| // Creating a text File using FileWriter  **import** java.io.FileWriter;  **import** java.io.IOException;  **class** CreateFile  {  **public** **static** **void** main(String[] args) **throws** IOException      {          // Accept a string          String str = "File Handling in Java using "+                  " FileWriter and FileReader";            // attach a file to FileWriter          FileWriter fw=**new** FileWriter("output.txt");            // read character wise from string and write          // into FileWriter  **for** (**int** i = 0; i < str.length(); i++)              fw.write(str.charAt(i));            System.out.println("Writing successful");          //close the file          fw.close();      }  } |

**FileReader**

FileReader is useful to read data in the form of characters from a ‘text’ file.

* This class inherited from the InputStreamReader Class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
* FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

**Constructors:**

* **FileReader(File file) –**Creates a FileReader , given the File to read from
* **FileReader(FileDescripter fd) –** Creates a new FileReader , given the FileDescripter to read from
* **FileReader(String fileName) –**Creates a new FileReader , given the name of the file to read from

**Methods:**

* **public int read () throws IOException –** Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.
* **public int read(char[] cbuff) throws IOException –** Reads characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.
* **public abstract int read(char[] buff, int off, int len) throws IOException –**Reads characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.   
  Parameters:   
  cbuf – Destination buffer   
  off – Offset at which to start storing characters   
  len – Maximum number of characters to read
* **public void close() throws IOException** closes the reader.
* **public long skip(long n) throws IOException –**Skips characters. This method will block until some characters are available, an I/O error occurs, or the end of the stream is reached.   
  Parameters:   
  n – The number of characters to skip

The following program depicts how to read from the ‘text’ file using FileReader

|  |
| --- |
| // Reading data from a file using FileReader  **import** java.io.FileNotFoundException;  **import** java.io.FileReader;  **import** java.io.IOException;  **class** ReadFile  {  **public** **static** **void** main(String[] args) **throws** IOException      {          // variable declaration  **int** ch;            // check if File exists or not          FileReader fr=**null**;  **try**          {              fr = **new** FileReader("text");          }  **catch** (FileNotFoundException fe)          {              System.out.println("File not found");          }            // read from FileReader till the end of file  **while** ((ch=fr.read())!=-1)              System.out.print((**char**)ch);            // close the file          fr.close();      }  } |

# Java.io.RandomAccessFile Class

## Introduction

The **Java.io.RandomAccessFile** class file behaves like a large array of bytes stored in the file system.Instances of this class support both reading and writing to a random access file.

## Class declaration

Following is the declaration for **Java.io.RandomAccessFile** class −

public class RandomAccessFile

extends Object

implements DataOutput, DataInput, Closeable

## Class constructors

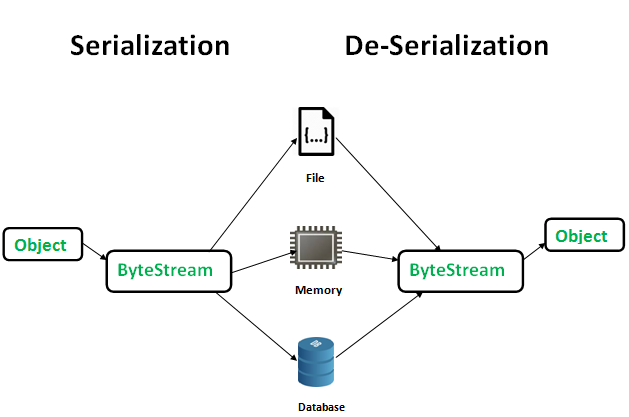
|  |  |
| --- | --- |
| **Sr.No.** | **Constructor & Description** |
| 1 | **RandomAccessFile(File file, String mode)**  This creates a random access file stream to read from, and optionally to write to, the file specified by the File argument. |
| 2 | **RandomAccessFile(File file, String mode)**  This creates a random access file stream to read from, and optionally to write to, a file with the specified name. |

## Class methods

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | [**void close()**](https://www.tutorialspoint.com/java/io/randomaccessfile_close.htm)  This method Closes this random access file stream and releases any system resources associated with the stream. |
| 2 | [**FileChannel getChannel()**](https://www.tutorialspoint.com/java/io/randomaccessfile_getchannel.htm)  This method returns the unique FileChannel object associated with this file. |
| 3 | [**FileDescriptor getFD()**](https://www.tutorialspoint.com/java/io/randomaccessfile_getfd.htm)  This method returns the opaque file descriptor object associated with this stream. |
| 4 | [**long getFilePointer()**](https://www.tutorialspoint.com/java/io/randomaccessfile_getfilepointer.htm)  This method returns the current offset in this file. |
| 5 | [**long length()**](https://www.tutorialspoint.com/java/io/randomaccessfile_length.htm)  This method returns the length of this file. |
| 6 | [**int read()**](https://www.tutorialspoint.com/java/io/randomaccessfile_read.htm)  This method reads a byte of data from this file. |
| 7 | [**int read(byte[] b)**](https://www.tutorialspoint.com/java/io/randomaccessfile_read_byte.htm)  This method reads up to *b.length* bytes of data from this file into an array of bytes. |
| 8 | [**int read(byte[] b, int off, int len)**](https://www.tutorialspoint.com/java/io/randomaccessfile_read_byte_len.htm)  This method reads up to *len* bytes of data from this file into an array of bytes. |
| 9 | [**boolean readBoolean()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readboolean.htm)  This method reads a boolean from this file. |
| 10 | [**byte readByte()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readbyte.htm)  This method reads a signed eight-bit value from this file. |
| 11 | [**char readChar()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readchar.htm)  This method reads a character from this file. |
| 12 | [**double readDouble()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readdouble.htm)  This method reads a double from this file. |
| 13 | [**float readFloat()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readfloat.htm)  This method reads a float from this file. |
| 14 | [**void readFully(byte[] b)**](https://www.tutorialspoint.com/java/io/randomaccessfile_readfully_byte.htm)  This method reads b.length bytes from this file into the byte array, starting at the current file pointer. |
| 15 | [**void readFully(byte[] b, int off, int len)**](https://www.tutorialspoint.com/java/io/randomaccessfile_readfully_byte_len.htm)  This method reads exactly len bytes from this file into the byte array, starting at the current file pointer. |
| 16 | [**int readInt()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readint.htm)  This method reads a signed 32-bit integer from this file. |
| 17 | [**String readLine()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readline.htm)  This method reads the next line of text from this file. |
| 18 | [**long readLong()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readlong.htm)  This method reads a signed 64-bit integer from this file. |
| 19 | [**short readShort()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readshort.htm)  This method reads a signed 16-bit number from this file. |
| 20 | [**int readUnsignedByte()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readunsignedbyte.htm)  This method reads an unsigned eight-bit number from this file. |
| 21 | [**int readUnsignedShort()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readunsignedshort.htm)  This method reads an unsigned 16-bit number from this file. |
| 22 | [**String readUTF()**](https://www.tutorialspoint.com/java/io/randomaccessfile_readutf.htm)  This method reads in a string from this file. |
| 23 | [**void seek(long pos)**](https://www.tutorialspoint.com/java/io/randomaccessfile_seek.htm)  This method sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs. |
| 24 | [**void setLength(long newLength)**](https://www.tutorialspoint.com/java/io/randomaccessfile_setlength.htm)  This method sets the length of this file. |
| 25 | [**int skipBytes(int n)**](https://www.tutorialspoint.com/java/io/randomaccessfile_skipbytes.htm)  This method attempts to skip over n bytes of input discarding the skipped bytes. |
| 26 | [**void write(byte[] b)**](https://www.tutorialspoint.com/java/io/randomaccessfile_write_byte.htm)  This method writes b.length bytes from the specified byte array to this file, starting at the current file pointer. |
| 27 | [**void write(byte[] b, int off, int len)**](https://www.tutorialspoint.com/java/io/randomaccessfile_write_byte_len.htm)  This method writes len bytes from the specified byte array starting at offset off to this file. |
| 28 | [**void write(int b)**](https://www.tutorialspoint.com/java/io/randomaccessfile_write.htm)  This method writes the specified byte to this file. |
| 29 | [**void writeBoolean(boolean v)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writeboolean.htm)  This method writes a boolean to the file as a one-byte value. |
| 30 | [**void writeByte(int v)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writebyte.htm)  This method writes a byte to the file as a one-byte value. |
| 31 | [**void writeBytes(String s)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writebytes.htm)  This method writes the string to the file as a sequence of bytes. |
| 32 | [**void writeChar(int v)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writechar.htm)  This method writes a char to the file as a two-byte value, high byte first. |
| 33 | [**void writeChars(String s)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writechars.htm)  This method writes a string to the file as a sequence of characters. |
| 34 | [**void writeDouble(double v)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writedouble.htm)  This method converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first. |
| 35 | [**void writeFloat(float v)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writefloat.htm)  This method converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first. |
| 36 | [**void writeInt(int v)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writeint.htm)  This method writes an int to the file as four bytes, high byte first. |
| 37 | [**void writeLong(long v)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writelong.htm)  This method writes a long to the file as eight bytes, high byte first. |
| 38 | [**void writeShort(int v)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writeshort.htm)  This method writes a short to the file as two bytes, high byte first. |
| 39 | [**void writeUTF(String str)**](https://www.tutorialspoint.com/java/io/randomaccessfile_writeutf.htm)  This method writes a string to the file using modified UTF-8 encoding in a machine-independent manner. |

# Serialization and Deserialization in Java

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.   
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. 

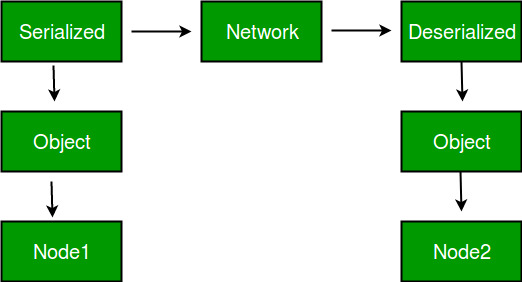
public final Object readObject()

throws IOException,

ClassNotFoundException

**Advantages of Serialization**

1. To save/persist state of an object.
2. To travel an object across a network.



Only the objects of those classes can be serialized which are implementing **java.io.Serializable** 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.

**Points to remember**

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. Example :

class A implements Serializable{

// B also implements Serializable

// interface.

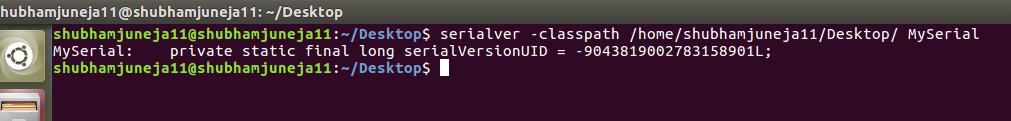
B ob=new B();

}

**SerialVersionUID** The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an **InvalidClassException**.

A Serializable class can declare its own UID explicitly by declaring a field name. It must be static, final and of type long. 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. It is also recommended to use private modifier for UID since it is not useful as inherited member. **serialver** The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.

You can run the following command to get serialVersionUID serialver [-classpath classpath] [-show] [classname…]



**Example 1:**

|  |
| --- |
| // Java code for serialization and deserialization  // of a Java object  **import** java.io.\*;    **class** Demo **implements** java.io.Serializable  {  **public** **int** a;  **public** String b;        // Default constructor  **public** Demo(**int** a, String b)      {  **this**.a = a;  **this**.b = b;      }    }    **class** Test  {  **public** **static** **void** main(String[] args)      {          Demo object = **new** Demo(1, "geeksforgeeks");          String filename = "file.ser";            // Serialization  **try**          {              //Saving of object in a file              FileOutputStream file = **new** FileOutputStream(filename);              ObjectOutputStream out = **new** ObjectOutputStream(file);                // Method for serialization of object              out.writeObject(object);                out.close();              file.close();                System.out.println("Object has been serialized");            }    **catch**(IOException ex)          {              System.out.println("IOException is caught");          }              Demo object1 = **null**;            // Deserialization  **try**          {              // Reading the object from a file              FileInputStream file = **new** FileInputStream(filename);              ObjectInputStream in = **new** ObjectInputStream(file);                // Method for deserialization of object              object1 = (Demo)in.readObject();                in.close();              file.close();                System.out.println("Object has been deserialized ");              System.out.println("a = " + object1.a);              System.out.println("b = " + object1.b);          }    **catch**(IOException ex)          {              System.out.println("IOException is caught");          }    **catch**(ClassNotFoundException ex)          {              System.out.println("ClassNotFoundException is caught");          }        }  } |

**Output :**

Object has been serialized

Object has been deserialized

a = 1

b = geeksforgeeks

**Example 2:**

|  |
| --- |
| // Java code for serialization and deserialization  // of a Java object  **import** java.io.\*;    **class** Emp **implements** Serializable {  **private** **static** **final** **long** serialversionUID =                                   129348938L;  **transient** **int** a;  **static** **int** b;      String name;  **int** age;        // Default constructor  **public** Emp(String name, **int** age, **int** a, **int** b)      {  **this**.name = name;  **this**.age = age;  **this**.a = a;  **this**.b = b;      }    }    **public** **class** SerialExample {  **public** **static** **void** printdata(Emp object1)      {            System.out.println("name = " + object1.name);          System.out.println("age = " + object1.age);          System.out.println("a = " + object1.a);          System.out.println("b = " + object1.b);      }    **public** **static** **void** main(String[] args)      {          Emp object = **new** Emp("ab", 20, 2, 1000);          String filename = "shubham.txt";            // Serialization  **try** {                // Saving of object in a file              FileOutputStream file = **new** FileOutputStream                                             (filename);              ObjectOutputStream out = **new** ObjectOutputStream                                             (file);                // Method for serialization of object              out.writeObject(object);                out.close();              file.close();                System.out.println("Object has been serialized\n"                                + "Data before Deserialization.");              printdata(object);                // value of static variable changed              object.b = 2000;          }    **catch** (IOException ex) {              System.out.println("IOException is caught");          }            object = **null**;            // Deserialization  **try** {                // Reading the object from a file              FileInputStream file = **new** FileInputStream                                           (filename);              ObjectInputStream in = **new** ObjectInputStream                                           (file);                // Method for deserialization of object              object = (Emp)in.readObject();                in.close();              file.close();              System.out.println("Object has been deserialized\n"                                  + "Data after Deserialization.");              printdata(object);                // System.out.println("z = " + object1.z);          }    **catch** (IOException ex) {              System.out.println("IOException is caught");          }    **catch** (ClassNotFoundException ex) {              System.out.println("ClassNotFoundException" +                                  " is caught");          }      }  } |

**Output:**

Object has been serialized

Data before Deserialization.

name = ab

age = 20

a = 2

b = 1000

Object has been deserialized

Data after Deserialization.

name = ab

age = 20

a = 0

b = 2000

Description for Output: You have seen while deserializing the object the values of a and b has changed. 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.

Transient Vs Final:  
**final** variables will be participated into serialization directly by their values.  
Hence declaring a final variable as transient there is no use.  
//the compiler assign the value to final variable

example:

final int x= 10;

int y = 20;

System.out.println(x);// compiler will replace this as System.out.println(10)->10

because x is final.

System.out.println(y);//20

**Example 3:**

|  |
| --- |
| //java code for final with transient    **import** java.io.\*;      **class** Dog **implements** Serializable{  **int** i=10;  **transient** **final** **int** j=20;  }  **class** GFG {  **public** **static** **void** main (String[] args)**throws** IOException,ClassNotFoundException      {            Dog d1=**new** Dog();            //Serialization started            System.out.println("serialization started");            FileOutputStream fos= **new** FileOutputStream("abc.ser");            ObjectOutputStream oos=**new** ObjectOutputStream(fos);            oos.writeObject(d1);            System.out.println("Serialization ended");              //Deserialization started            System.out.println("Deserialization started");            FileInputStream fis=**new** FileInputStream("abc.ser");            ObjectInputStream ois=**new** ObjectInputStream(fis);            Dog d2=(Dog) ois.readObject();            System.out.println("Deserialization ended");            System.out.println("Dog object data");            //final result            System.out.println(d2.i+"\t" +d2.j);      }  } |

**Output**

serialization started

Serialization ended

Deserialization started

Deserialization ended

Dog object data

10 20

#### 

**Practicals ::**

**1)**

public class ClassProgram{

 int puppyAge;

 public ClassProgram(String name){

 // This constructor has one parameter, name.

 System.out.println("Name chosen is :" + name );

 }

 public void setAge( int age ){

 puppyAge = age;

 }

 public int getAge( ){

 System.out.println("Puppy's age is :" + puppyAge );

 return puppyAge;

 }

 public static void main(String []args){

 /\* Object creation \*/

 ClassProgram myPuppy = new ClassProgram( "tommy" );

 /\* Call class method to set puppy's age \*/

 myPuppy.setAge( 2 );

 /\* Call another class method to get puppy's age \*/

 myPuppy.getAge( );

 /\* You can access instance variable as follows as well \*/

 System.out.println("Variable Value :" + myPuppy.puppyAge );

 }

}

**2)**

public class CLDemo {

    public static void main(String[] args) {

        System.out.println("You entered " + args.length

                + " command-line arguments");

        if (args.length > 0) {

            System.out.println("They were:");

            for (int i = 0; i < args.length; i++)

                System.out.println(" " + args[i]);

        }

    } // end main()

} // end class CLDemo

**3)**

public class EnumDemo {

    // Define two enum types -- remember that the definitions

    // go OUTSIDE The main() routine!

    enum Day {

        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

    }

    enum Month {

        JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC

    }

    public static void main(String[] args) {

        Day tgif; // Declare a variable of type Day.

        Month libra; // Declare a variable of type Month.

        tgif = Day.FRIDAY; // Assign a value of type Day to tgif.

        libra = Month.OCT; // Assign a value of type Month to libra.

        System.out.print("My sign is libra, since I was born in ");

        System.out.println(libra); // Output value will be: OCT

        System.out.print("That is the ");

        System.out.print(libra.ordinal());

        System.out.println("-th month of the year.");

        System.out.println(" (Counting from 0, of course!)");

        System.out.print("Is not it nice to get to ");

        System.out.println(tgif); // Output value will be: FRIDAY

        System.out.println(tgif + " is the " + tgif.ordinal()

                + "-th day of the week.");

        // You can concatenate enum values onto Strings!

    }

}

**4)**

// Java program to illustrate creating an array

// of integers, puts some values in the array,

// and prints each value to standard output.

// Syntax:

// 1) data type[] arrName;

// 2) datatype arrName[];

// 3) datatype [] arrName;

class Arrays {

    public static void main(String[] args)

    {

        // declares an Array of integers.

        int[] arr;

        // allocating memory for 5 integers.

        arr = new int[5];

        // initialize the first elements of the array

        arr[0] = 10;

        // initialize the second elements of the array

        arr[1] = 20;

        // so on...

        arr[2] = 30;

        arr[3] = 40;

        arr[4] = 50;

        // accessing the elements of the specified array

        for (int i = 0; i < arr.length; i++)

            System.out.println("Element at index " + i

                            + " : " + arr[i]);

    }

}

// OUTPUT :

// Element at index 0 : 10

// Element at index 1 : 20

// Element at index 2 : 30

// Element at index 3 : 40

// Element at index 4 : 50

**5)**

// Java program to demonstrate

// passing of array to method

// Passing Arrays to Methods

// Like variables, we can also pass arrays to methods. For example, the below program passes the array to method sum to calculate the sum of the array’s values.

public class PassingOfArray {

    // Driver method

    public static void main(String args[])

    {

        int arr[] = { 3, 1, 2, 5, 4 };

        // passing array to method m1

        sum(arr);

    }

    public static void sum(int[] arr)

    {

        // getting sum of array values

        int sum = 0;

        for (int i = 0; i < arr.length; i++)

            sum += arr[i];

        System.out.println("sum of array values : " + sum);

    }

}

// Output

// sum of array values : 15

**6)**

// Returning Arrays from Methods

// As usual, a method can also return an array. For example, the below program returns an array from method m1.

// Java program to demonstrate

// return of array from method

class ArrayFromMethod {

    // Driver method

    public static void main(String args[])

    {

        int arr[] = m1();

        for (int i = 0; i < arr.length; i++)

            System.out.print(arr[i] + " ");

    }

    public static int[] m1()

    {

        // returning array

        return new int[] { 1, 2, 3 };

    }

}

// Output

// 1 2 3

**7)**

// Java program to illustrate creating

// an array of objects

class Student {

    public int roll\_no;

    public String name;

    Student(int roll\_no, String name)

    {

        this.roll\_no = roll\_no;

        this.name = name;

    }

}

// Elements of the array are objects of a class Student.

public class ArrayStudent{

    public static void main(String[] args)

    {

        // declares an Array of integers.

        Student[] arr;

        // allocating memory for 5 objects of type Student.

        arr = new Student[5];

        // initialize the first elements of the array

        arr[0] = new Student(1, "aman");

        // initialize the second elements of the array

        arr[1] = new Student(2, "vaibhav");

        // so on...

        arr[2] = new Student(3, "shikar");

        arr[3] = new Student(4, "dharmesh");

        arr[4] = new Student(5, "mohit");

        // accessing the elements of the specified array

        for (int i = 0; i < arr.length; i++)

            System.out.println("Element at " + i + " : "

                            + arr[i].roll\_no + " "

                            + arr[i].name);

    }

}

// Output :

// Element at 0 : 1 aman

// Element at 1 : 2 vaibhav

// Element at 2 : 3 shikar

// Element at 3 : 4 dharmesh

// Element at 4 : 5 mohit

**8)**

// Java program to illustrate creating

// an array of objects

class Student

{

    public String name;

    Student(String name)

    {

        this.name = name;

    }

    @Override

    public String toString(){

        return name;

    }

}

// Elements of the array are objects of a class Student.

public class ArrayStudent2

{

    public static void main (String[] args)

    {

        // declares an Array and initializing the elements of the array

        Student[] myStudents = new Student[]{new Student("Dharma"),new Student("sanvi"),new Student("Rupa"),new Student("Ajay")};

        // accessing the elements of the specified array

        for(Student m:myStudents){

            System.out.println(m);

        }

    }

}

// Output :

// Dharma

// sanvi

// Rupa

// Ajay

**9)**

// Multidimensional Arrays:

// Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other arrays. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

// datatype [][] arrayrefvariable;

// or

// datatype arrayrefvariable[][];

// int[][] intArray = new int[10][20]; //a 2D array or matrix

// int[][][] intArray = new int[10][20][10]; //a 3D array

public class multiDimensional {

    public static void main(String args[])

    {

        // declaring and initializing 2D array

        int arr[][]

            = { { 2, 7, 9 }, { 3, 6, 1 }, { 7, 4, 2 } };

        // printing 2D array

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++)

                System.out.print(arr[i][j] + " ");

            System.out.println();

        }

    }

}

//OUTPUT

// 2 7 9

// 3 6 1

// 7 4 2

**10)**

// Java program to demonstrate

// Class Objects for Arrays

class ObjectsForArrays {

    public static void main(String args[])

    {

        int intArray[] = new int[3];

        byte byteArray[] = new byte[3];

        short shortsArray[] = new short[3];

        // array of Strings

        String[] strArray = new String[3];

        System.out.println(intArray.getClass());

        System.out.println(

            intArray.getClass().getSuperclass());

        System.out.println(byteArray.getClass());

        System.out.println(shortsArray.getClass());

        System.out.println(strArray.getClass());

    }

}

// Output

// class [I

// class java.lang.Object

// class [B

// class [S

// class [Ljava.lang.String;

// Explanation:

// The string “[I” is the run-time type signature for the class object “array with component type int.”

// The only direct superclass of an array type is java.lang.Object.

// The string “[B” is the run-time type signature for the class object “array with component type byte.”

// The string “[S” is the run-time type signature for the class object “array with component type short.”

// The string “[L” is the run-time type signature for the class object “array with component type of a Class.” The Class name is then followed.

**11)**

// Java program to demonstrate

// cloning of one-dimensional arrays

class CloningArray {

    public static void main(String args[])

    {

        int intArray[] = { 1, 2, 3 };

        int cloneArray[] = intArray.clone();

        // will print false as deep copy is created

        // for one-dimensional array

        System.out.println(intArray == cloneArray);

        for (int i = 0; i < cloneArray.length; i++) {

            System.out.print(cloneArray[i] + " ");

        }

    }

}

// Output

// false

// 1 2 3

// Java program to demonstrate

// cloning of multi-dimensional arrays

// class CloningArray {

//  public static void main(String args[])

//  {

//      int intArray[][] = { { 1, 2, 3 }, { 4, 5 } };

//      int cloneArray[][] = intArray.clone();

//      // will print false

//      System.out.println(intArray == cloneArray);

//      // will print true as shallow copy is created

//      // i.e. sub-arrays are shared

//      System.out.println(intArray[0] == cloneArray[0]);

//      System.out.println(intArray[1] == cloneArray[1]);

//  }

// }

// Output

// false

// true

// true

**12)**

// Java Program to demonstrate

// Default Constructor

import java.io.\*;

// Driver class

// class ConstructorExamples {

//  // Default Constructor

//  ConstructorExamples() { System.out.println("Default constructor"); }

//  // Driver function

//  public static void main(String[] args)

//  {

//      ConstructorExamples hello = new ConstructorExamples();

//  }

// }

// Output

// Default constructor

// ---------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------------------------------

// Java Program for Parameterized Constructor

// class Sample {

//  // data members of the class.

//  String name;

//  int id;

//  Sample(String name, int id)

//  {

//      this.name = name;

//      this.id = id;

//  }

// }

// class ConstructorExamples {

//  public static void main(String[] args)

//  {

//      // This would invoke the parameterized constructor.

//      Sample obj1 = new Sample("avinash", 68);

//      System.out.println("Name :" + obj1.name

//                      + " and Id :" + obj1.id);

//  }

// }

//Output :

// Name :avinash and Id :68

// ---------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------------------------------

// Java Program to illustrate constructor overloading

// using same task (addition operation ) for different

// types of arguments.

import java.io.\*;

class Sample {

    // constructor with one argument

    Sample(String name)

    {

        System.out.println("Constructor with one "

                        + "argument - String : " + name);

    }

    // constructor with two arguments

    Sample(String name, int age)

    {

        System.out.println(

            "Constructor with two arguments : "

            + " String and Integer : " + name + " " + age);

    }

    // Constructor with one argument but with different

    // type than previous..

    Sample(long id)

    {

        System.out.println(

            "Constructor with one argument : "

            + "Long : " + id);

    }

}

class ConstructorExamples {

    public static void main(String[] args)

    {

        // Creating the objects of the class named 'Sample'

        // by passing different arguments

        // Invoke the constructor with one argument of

        // type 'String'.

        Sample obj2 = new Sample("Shikhar");

        // Invoke the constructor with two arguments

        Sample obj3 = new Sample("Dharmesh", 26);

        // Invoke the constructor with one argument of

        // type 'Long'.

        Sample obj4 = new Sample(325614567);

    }

}

//Output :

// Constructor with one argument - String : Shikhar

// Constructor with two arguments :  String and Integer : Dharmesh 26

// Constructor with one argument : Long : 325614567

**13)**

// Copy Constructor in Java

// Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

// Java Program for Copy Constructor

import java.io.\*;

class Sample {

    // data members of the class.

    String name;

    int id;

    // Parameterized Constructor

    Sample(String name, int id)

    {

        this.name = name;

        this.id = id;

    }

    // Copy Constructor

    Sample(Sample obj2)

    {

        this.name = obj2.name;

        this.id = obj2.id;

    }

}

class CopyConstructor {

    public static void main(String[] args)

    {

        // This would invoke the parameterized constructor.

        System.out.println("First Object");

        Sample obj1 = new Sample("avinash", 68);

        System.out.println("Sample Name :" + obj1.name

                        + " and Sampl Id :" + obj1.id);

        System.out.println();

        // This would invoke the copy constructor.

        Sample obj2 = new Sample(obj1);

        System.out.println(

            "Copy Constructor used Second Object");

        System.out.println("Sample Name :" + obj2.name

                        + " and Sample Id :" + obj2.id);

    }

}

// OUTPUT:

// First Object

// Sample Name :avinash and Sampl Id :68

// Copy Constructor used Second Object

// Sample Name :avinash and Sample Id :68

**14)**

// Java program to demonstrate working of method

// overloading in Java

public class OverLoadingExample {

    // Overloaded sum(). This sum takes two int parameters

    public int sum(int x, int y) { return (x + y); }

    // Overloaded sum(). This sum takes three int parameters

    public int sum(int x, int y, int z)

    {

        return (x + y + z);

    }

    // Overloaded sum(). This sum takes two double

    // parameters

    public double sum(double x, double y)

    {

        return (x + y);

    }

    // Driver code

    public static void main(String args[])

    {

        OverLoadingExample s = new OverLoadingExample();

        System.out.println(s.sum(10, 20));

        System.out.println(s.sum(10, 20, 30));

        System.out.println(s.sum(10.5, 20.5));

    }

}

// Output:

// 30

// 60

// 31.0

//----------------------------------------------------------------------------------------------

// 1. Changing the Number of Parameters

// Method overloading can be achieved by changing the number of parameters while passing to different methods.

// Java Program to Illustrate Method Overloading

// By Changing the Number of Parameters

// Importing required classes

// import java.io.\*;

// Class 1

// Helper class

// class Product {

//  // Method 1

//  // Multiplying two integer values

//  public int multiply(int a, int b)

//  {

//      int prod = a \* b;

//      return prod;

//  }

//  // Method 2

//  // Multiplying three integer values

//  public int multiply(int a, int b, int c)

//  {

//      int prod = a \* b \* c;

//      return prod;

//  }

// }

// Class 2

// Main class

// class OverLoadingExample {

//  // Main driver method

//  public static void main(String[] args)

//  {

//      // Creating object of above class inside main()

//      // method

//      Product ob = new Product();

//      // Calling method to Multiply 2 numbers

//      int prod1 = ob.multiply(1, 2);

//      // Printing Product of 2 numbers

//      System.out.println(

//          "Product of the two integer value :" + prod1);

//      // Calling method to multiply 3 numbers

//      int prod2 = ob.multiply(1, 2, 3);

//      // Printing product of 3 numbers

//      System.out.println(

//          "Product of the three integer value :" + prod2);

//  }

// }

//Output:

// Product of the two integer value :2

// Product of the three integer value :6

//-----------------------------------------------------------------------------------------------------

// 2. Changing Data Types of the Arguments

// In many cases, methods can be considered Overloaded if they have the same name but have different parameter types, methods are considered to be overloaded.

// Java Program to Illustrate Method Overloading

// By Changing Data Types of the Parameters

// Importing required classes

// import java.io.\*;

// class Product {

//  // Multiplying three integer values

//  public int Prod(int a, int b, int c)

//  {

//      int prod1 = a \* b \* c;

//      return prod1;

//  }

//  // Multiplying three double values.

//  public double Prod(double a, double b, double c)

//  {

//      double prod2 = a \* b \* c;

//      return prod2;

//  }

// }

// class OverLoadingExample {

//  public static void main(String[] args)

//  {

//      Product obj = new Product();

//      int prod1 = obj.Prod(1, 2, 3);

//      System.out.println(

//          "Product of the three integer value :" + prod1);

//      double prod2 = obj.Prod(1.0, 2.0, 3.0);

//      System.out.println(

//          "Product of the three double value :" + prod2);

//  }

// }

//Output:

// Product of the three integer value :6

// Product of the three double value :6.0

//----------------------------------------------------------------------------------------------------

// 3. Changing the Order of the Parameters of Methods

// Method overloading can also be implemented by rearranging the parameters of two or more overloaded methods. For example, if the parameters of method 1 are (String name, int roll\_no) and the other method is (int roll\_no, String name) but both have the same name, then these 2 methods are considered to be overloaded with different sequences of parameters.

// Java Program to Illustrate Method Overloading

// By changing the Order of the Parameters

// import java.io.\*;

// class Student {

//  // Method 1

//  public void StudentId(String name, int roll\_no)

//  {

//      System.out.println("Name :" + name + " "

//                      + "Roll-No :" + roll\_no);

//  }

//  // Method 2

//  public void StudentId(int roll\_no, String name)

//  {

//      // Again printing name and id of person

//      System.out.println("Roll-No :" + roll\_no + " "

//                      + "Name :" + name);

//  }

// }

// class OverLoadingExample {

//  // Main function

//  public static void main(String[] args)

//  {

//      // Creating object of above class

//      Student obj = new Student();

//      // Passing name and id

//      // Note: Reversing order

//      obj.StudentId("Spyd3r", 1);

//      obj.StudentId(2, "Kamlesh");

//  }

// }

// //Output:

// Name :Spyd3r Roll-No :1

// Roll-No :2 Name :Kamlesh

**15)**

package package\_name;

public class PackageTwo {

    public void methodClassTwo(){

        System.out.println("Hello there i am ClassTwo");

    }

}

package package\_one;

public class PackageOne {

    public void methodClassOne() {

        System.out.println("Hello there its ClassOne");

    }

}

import package\_one.PackageOne;

import package\_name.PackageTwo;

public class PackageTesting {

    public static void main(String[] args){

        PackageTwo a = new PackageTwo();

        PackageOne b = new PackageOne();

        a.methodClassTwo();

        b.methodClassOne();

    }

}

// output:

// Hello there i am ClassTwo

// Hello there its ClassOne