**Final Keyword In Java**

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes.

Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,

* The final variable cannot be reinitialized with another value
* The final method cannot be overridden
* The final class cannot be extended

## 1. Java final Variable

In Java, we cannot change the value of a final variable. For example,

class Main {

public static void main(String[] args) {

// create a final variable

final int AGE = 32;

// try to change the

System.out.println("Age: " + AGE);

} final variable

AGE = 45;

}

## 2. Java final Method

Java, the final method cannot be overridden by the child class. For example

class FinalDemo {

// create a final method

public final void display() {

System.out.println("This is a final method.");

}

}

class Main extends FinalDemo {

// try to override final method

public final void display() {

System.out.println("The final method is overridden.");

}

public static void main(String[] args) {

Main obj = new Main();

obj.display();

}

}

We have tried to override the final method in the Main class. When we run the program, we will get a compilation error with the following message.

display () in Main cannot override display() in FinalDemo

public final void display() {

^

overridden method is final

## 3. Java final Class

In Java, the final class cannot be inherited by another class. For example,

// create a final class

final class FinalClass {

public void display() {

System.out.println("This is a final method.");

}

}

// try to extend the final class

class Main extends FinalClass {

public void display() {

System.out.println("The final method is overridden.");

}

public static void main(String[] args) {

Main obj = new Main();

obj.display();

}

}

Java instanceof Operator

The instanceof operator in Java is used to check whether an object is an instance of a particular class or not.

Its syntax is

objectName instanceOf className;

Here, if objectName is an instance of className, the operator returns true. Otherwise, it returns false.

### Example: Java instanceof

class chkInstanceEx {

public static void main(String[] args) {

// create a variable of string type

String name = "Program for instanceof";

// checks if name is instance of String

boolean result1 = name instanceof String;

System.out.println("name is an instance of String: " + result1);

// create an object of Main

chkInstanceEx obj = new chkInstanceEx ();

// checks if obj is an instance of Main

boolean result2 = obj instanceof Main;

System.out.println("obj is an instance of Main: " + result2);

}

}

**Output**

name is an instance of String: true

obj is an instance of Main: true

**Encapsulation**

**Encapsulation in Java** is a process of wrapping code and data together into a single unit

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

### Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

### Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

//A Java class which is a fully encapsulated class.

//It has a private data member and getter and setter methods.

**package** com.javatpoint;

**public** **class** Student{

//private data member

**private** String name;

//getter method for name

**public** String getName(){

**return** name;

}

//setter method for name

**public** **void** setName(String name){

**this**.name=name

}

}

//A Java class to test the encapsulated class.

**package** com.javatpoint;

**class** Test{

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

//creating instance of the encapsulated class

Student s=**new** Student();

//setting value in the name member

s.setName("vijay");

//getting value of the name member

System.out.println(s.getName());

}

}

### Read-Only class

/A Java class which has only getter methods.

public class Student{

//private data member

private String college="AKG";

//getter method for college

public String getCollege(){

return college;

}

}

### Write-Only class

//A Java class which has only setter methods.

public class Student{

//private data member

private String college;

//getter method for college

public void setCollege(String college){

this.college=college;

}

}

### Another Example of Encapsulation in Java

//A Account class which is a fully encapsulated class.

//It has a private data member and getter and setter methods.

**class** Account {

//private data members

**private** **long** acc\_no;

**private** String name,email;

**private** **float** amount;

//public getter and setter methods

**public** **long** getAcc\_no() {

**return** acc\_no+” ”+name+amount+email;

}

**public** **void** setAcc\_no(**long** acc\_no) {

**this**.acc\_no = acc\_no;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** **float** getAmount() {

**return** amount;

}

**public** **void** setAmount(**float** amount) {

**this**.amount = amount;

}

}

*File: TestAccount.java*

//A Java class to test the encapsulated class Account.

**public** **class** TestEncapsulation {

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

    //creating instance of Account class

    Account acc=**new** Account();

    //setting values through setter methods

    acc.setAcc\_no(7560504000L);

    acc.setName("SonoJaisree");

    acc.setEmail("sonjaisree@gmail.com");

    acc.setAmount(500000f);

    //getting values through getter methods

    System.out.println(acc.getAcc\_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());

}

}

**Output:**

7560504000 SonoJaisree sonojaisree@gmail.com 500000.0

**Access Modifiers**

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

### Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

**Simple example of private access modifier**

**class** A{

**private** **int** data=40;

**private** **void** msg(){System.out.println("Hello java");}

}

**public** **class** Simple{

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

   A obj=**new** A();

   System.out.println(obj.data);//Compile Time Error

  obj.msg();//Compile Time Error

   }

}

### 2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

//save by A.java

**package** pack;

**class** A{

**void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B{

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

  A obj = **new** A();//Compile Time Error

   obj.msg();//Compile Time Error

 }

}

### 3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

**package** pack;

**public** **class** A{

**protected** **void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B **extends** A{

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

   B obj = **new** B();

   obj.msg();

  }

}

Output:Hello

### 4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

//save by A.java

**package** pack;

**public** **class** A{

**public** **void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B{

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

   A obj = **new** A();

   obj.msg ();

  }

}

# strictfp keyword in java

**strictfp** is a modifier that stands for strict floating-point which was not introduced in the base version of java as it was introduced in Java version 1.2. It is used in java for restricting floating-point calculations and ensuring the same result on every platform while performing operations in the floating-point variable.   
Floating-point calculations are platform-dependent i.e. different output(floating-point values) is achieved when a class file is run on different platforms(16/32/64 bit processors). To solve this type of issue, strictfp keyword was introduced in JDK 1.2 version by following [IEEE 754](https://en.wikipedia.org/wiki/IEEE_floating_point) standards for floating-point calculations.

### Legal code for strictfp keyword

The strictfp keyword can be applied on methods, classes and interfaces.

**strictfp** **class** A{}//strictfp applied on class  

**strictfp** **interface** M{}//strictfp applied on interface

class A{

strictfp void m(){}//strictfp applied on method

}

// Java program to illustrate strictfp modifier

// Usage in Classes

// Main class

class GFG {

// Method 1

// Calculating sum using strictfp modifier

public strictfp double sum()

{

double num1 = 10e+10;

double num2 = 6e+08;

// Returning the sum

return (num1 + num2);

}

// Method 2

// Main driver method

public static void main(String[] args)

{

// Creating object of class in main() method

GFG t = new GFG();

// Here we have error of putting strictfp and

// error is not found public static void main method

System.out.println(t.sum());

}

}

***Note:****strictfp modifier is used with classes, interfaces, and methods only but is not applicable to apply with variables.*

### Illegal code for strictfp keyword

The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

Object class in Java

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

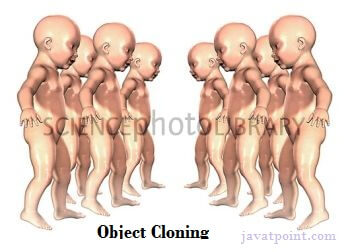
Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

1. Object obj=getObject();//we don't know what object will be returned from this method

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

**Object Cloning:**

# Object Cloning in Java

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

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

Object obj = new A();

Object obj1 = new A();

If(obj.equals(obj1))

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

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

**class** Student **implements** Cloneable{

**int** rollno;

String name;

Student18(**int** rollno,String name){

**this**.rollno=rollno;

**this**.name=name;

}

**public** Object clone()**throws** CloneNotSupportedException{

**return** **super**.clone();

}

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

**try**{

Student18 s1=**new** Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);

System.out.println(s2.rollno+" "+s2.name);

}**catch**(CloneNotSupportedException c){}

}

}

### Downcasting

**Upcasting** is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the **"ClassCastException"**. Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.

//Parent class

class Parent {

    String name;

    // A method which prints the data of the parent class

**void** showMessage()

    {

        System.out.println("Parent method is called");

    }

}

// Child class

class Child extends Parent {

**int** age;

    // Performing overriding

    @Override

    void showMessage()

    {

        System.out.println("Child method is called");

    }

}

**public** **class** Downcasting{

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

    {

        Parent p = **new** Child();

        p.name = "Shubham";

          Bank b = new SBI(); //upcasting

SBI s = (SBI)b // Downcasting

        // Performing Downcasting Implicitly

        //Child c = new Parent(); // it gives compile-time error

        // Performing Downcasting Explicitly

        Child c = (Child)p;

        c.age = 18;

        System.out.println(c.name);

        System.out.println(c.age);

        c.showMessage();

    }

}

**Note:**

In Java, we rarely use **Upcasting**. We use it when we need to develop a code that deals with only the parent class. **Downcasting** is used when we need to develop a code that accesses behaviors of the child class.