Inheritance

*Java doesn’t support Multiple Inheritance.*

**class** Parent1 {

  // Method inside first parent class

**void** fun() {

    // Print statement if this method is called

    System.out.println("Parent1");

  }

}

// Class 2

// Second Parent Class

**class** Parent2 {

  // Method inside first parent class

**void** fun() {

    // Print statement if this method is called

    System.out.println("Parent2");

  }

}

// Class 3

// Trying to be child of both the classes

**class** Test **extends** Parent1, Parent2 {

  // Main driver method

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

    // Creating object of class in main() method

    Test t = **new** Test();

    // Trying to call above functions of class where

    // Error is thrown as this class is inheriting

    // multiple classes

    t.fun();

  }

}

*Multiple Inheritance is not supported by class because of ambiguity. In the case of interface, there is no ambiguity because the implementation of the method(s) is provided by the implementing class up to Java 7. From Java 8, interfaces also have implementations of methods. So if a class implements two or more interfaces having the same method signature with implementation, it is mandated to implement the method in class also.*

*Abstraction*

* **Abstract classes cannot be instantiated**: An abstract class is a class that cannot be instantiated directly. Instead, it is meant to be extended by other classes, which can provide concrete implementations of its abstract methods.
* **Abstract methods do not have a body**: An abstract method is a method that does not have an implementation. It is declared using the abstract keyword and ends with a semicolon instead of a method body. Subclasses of an abstract class must provide a concrete implementation of all abstract methods defined in the parent class.
* **Abstract classes can have both abstract and concrete methods:** Abstract classes can contain both abstract and concrete methods. Concrete methods are implemented in the abstract class itself and can be used by both the abstract class and its subclasses.
* **Abstract classes can have constructors**: Abstract classes can have constructors, which are used to initialize instance variables and perform other initialization tasks. However, because abstract classes cannot be instantiated directly, their constructors are typically called constructors in concrete subclasses.
* **Abstract classes can contain instance variables**: Abstract classes can contain instance variables, which can be used by both the abstract class and its subclasses. Subclasses can access these variables directly, just like any other instance variables.
* **Abstract classes can implement interfaces:** Abstract classes can implement interfaces, which define a set of methods that must be implemented by any class that implements the interface. In this case, the abstract class must provide concrete implementations of all methods defined in the interface.

**abstract** **class** gfg {

**abstract** **void** printInfo();

}

// Child Class

**class** employee **extends** gfg {

**void** printInfo()

    {

        String name = "aakanksha";

**int** age = 21;

**float** salary = 55552.2F;

        System.out.println(name);

        System.out.println(age);

        System.out.println(salary);

    }

}

// driver Class

**class** base {

    // main function

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

    {

        // object created

        gfg s = **new** employee();

        s.printInfo();

    }

}

final is used to prevent [inheritance](https://www.geeksforgeeks.org/inheritance-in-java/) whereas abstract classes depend upon their child classes for complete implementation. In cases of methods, final is used to prevent [overriding](https://www.geeksforgeeks.org/overriding-in-java/) whereas abstract methods need to be overridden in sub-classes.

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

**Difference between Abstract Class and Interface in Java**

 An abstract class is a class that cannot be instantiated and can contain both abstract and non-abstract methods. An interface, on the other hand, is a contract that specifies a set of methods that a class must implement.

Method implementation: In an abstract class, some methods can be implemented, while others are left abstract, meaning that they have no implementation and must be overridden by concrete subclasses. In contrast, all methods in an interface are by default abstract and must be implemented by any class that implements the interface.

Inheritance: A class can inherit from only one abstract class, but it can implement multiple interfaces. This is because an abstract class represents a type of object, while an interface represents a set of behaviors.

Access modifiers: Abstract classes can have access modifiers such as public, protected, and private for their methods and properties, while interfaces can only have public access.

Variables: An abstract class can have member variables, while an interface cannot.

# Difference Between Data Hiding and Abstraction in Java

[***Abstraction***](https://www.geeksforgeeks.org/abstraction-in-java-2/) Is hiding the internal implementation and just highlight the set of services. It is achieved by using the abstract class and interfaces and further implementing the same. Only necessarily characteristics of an object that differentiates it from all other objects. Only the important details are emphasized and the rest all are suppressed from the user or reader

***Data Hiding***is hiding internal data from outside users. The internal data should not go directly that is outside person/classes is not able to access internal data directly. It is achieved by using an access specifier- a private modifier.

Encapsulation

**// Java program to demonstrate encapsulation**

**class Encapsulate {**

**// private variables declared**

**// these can only be accessed by**

**// public methods of class**

**private String geekName;**

**private int geekRoll;**

**private int geekAge;**

**// get method for age to access**

**// private variable geekAge**

**public int getAge() { return geekAge; }**

**// get method for name to access**

**// private variable geekName**

**public String getName() { return geekName; }**

**// get method for roll to access**

**// private variable geekRoll**

**public int getRoll() { return geekRoll; }**

**// set method for age to access**

**// private variable geekage**

**public void setAge(int newAge) { geekAge = newAge; }**

**// set method for name to access**

**// private variable geekName**

**public void setName(String newName)**

**{**

**geekName = newName;**

**}**

**// set method for roll to access**

**// private variable geekRoll**

**public void setRoll(int newRoll) { geekRoll = newRoll; }**

**}**

**// Class to access variables**

**// of the class Encapsulate**

**public class TestEncapsulation {**

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

**{**

**Encapsulate obj = new Encapsulate();**

**// setting values of the variables**

**obj.setName("Harsh");**

**obj.setAge(19);**

**obj.setRoll(51);**

**// Displaying values of the variables**

**System.out.println("Geek's name: " + obj.getName());**

**System.out.println("Geek's age: " + obj.getAge());**

**System.out.println("Geek's roll: " + obj.getRoll());**

**// Direct access of geekRoll is not possible**

**// due to encapsulation**

**// System.out.println("Geek's roll: " +**

**// obj.geekName);**

**}**

**}**

**Difference between Abstraction and Encapsulation:**

| **Abstraction** | **Encapsulation** |
| --- | --- |
| Abstraction is the process or method of gaining the information. | While encapsulation is the process or method to contain the information. |
| In abstraction, problems are solved at the design or interface level. | While in encapsulation, problems are solved at the implementation level. |
| Abstraction is the method of hiding the unwanted information. | Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside. |
| We can implement abstraction using abstract class and interfaces. | Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public. |
| In abstraction, implementation complexities are hidden using abstract classes and interfaces. | While in encapsulation, the data is hidden using methods of getters and setters. |
| The objects that help to perform abstraction are encapsulated. | Whereas the objects that result in encapsulation need not be abstracted. |
| Abstraction provides access to specific part of data. | Encapsulation hides data and the user can not access same directly (data hiding. |
| Abstraction focus is on “what” should be done. | Encapsulation focus is on “How” it should be done. |

**Polymorphism**

### Types of Inheritance are:

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritances
4. Hybrid inheritance
5. Hierarchical inheritance

Polymorphism is that in which we can perform a task in multiple forms or ways. It is applied to the functions or methods. Polymorphism allows the object to decide which form of the function to implement at compile-time as well as run-time.

Types of Polymorphism are:

1. Compile-time polymorphism (Method overloading)
2. Run-time polymorphism (Method Overriding)

## **Difference between Inheritance and Polymorphism:**

| **S.NO** | **Inheritance** | **Polymorphism** |
| --- | --- | --- |
| 1. | Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class). | Whereas polymorphism is that which can be defined in multiple forms. |
| 2. | It is basically applied to classes. | Whereas it is basically applied to functions or methods. |
| 3. | Inheritance supports the concept of reusability and reduces code length in object-oriented programming. | Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding). |
| 4. | Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance. | Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding). |
| 5. | It is used in pattern designing. | While it is also used in pattern designing. |
| 6. | **Example :**  The class bike can be inherit from the class of two-wheel vehicles, which is turn could be a subclass of vehicles. | **Example :**  The class bike can have method name set\_color(), which changes the bike’s color based on the name of color you have entered. |

[Static vs Dynamic binding](https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/)

* Static binding is done during compile-time while dynamic binding is done during run-time.
* private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

| **Compile Time Polymorphism** | **Run time Polymorphism** |
| --- | --- |
| In Compile time Polymorphism, the call is resolved by the compiler. | In Run time Polymorphism, the call is not resolved by the compiler. |
| It is also known as Static binding, Early binding and overloading as well. | It is also known as Dynamic binding, Late binding and overriding as well. |
| Method overloading is the compile-time polymorphism where more than one methods share the same name with different parameters or signature and different return type. | Method overriding is the runtime polymorphism having the same method with same parameters or signature but associated withcompared, different classes. |
| It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and pointers. |
| It provides fast execution because the method that needs to be executed is known early at the compile time. | It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime. |
| Compile time polymorphism is less flexible as all things execute at compile time. | Run time polymorphism is more flexible as all things execute at run time. |
| Inheritance is not involved. | Inheritance is involved. |

Final

* **Final variables:** When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.
* **Final methods**: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class’s public API and should not be modified by subclasses.
* **Final classes:** When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as is and should not be modified or extended.
* **Initialization:** Final variables must be initialized either at the time of declaration or in the constructor of the class. This ensures that the value of the variable is set and cannot be changed.
* **Performance:** The use of final can sometimes improve performance, as the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed.
* **Security:** final can help improve security by preventing malicious code from modifying sensitive data or behavior.

Static

* **Shared memory allocation**: Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.
* **Accessible without object instantiation:** Static members can be accessed without the need to create an instance of the class. This makes them useful for providing utility functions and constants that can be used across the entire program.
* **Associated with class, not objects:** Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference.
* **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.
* **Can be overloaded, but not overridden**: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

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

**class** Test **implements** Serializable

{

    // Normal variables

**int** i = 10, j = 20;

    // Transient variables

**transient** **int** k = 30;

    // Use of transient has no impact here

**transient** **static** **int** l = 40;

**transient** **final** **int** m = 50;

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

    {

        Test input = **new** Test();

        // serialization

        FileOutputStream fos = **new** FileOutputStream("abc.txt");

        ObjectOutputStream oos = **new** ObjectOutputStream(fos);

        oos.writeObject(input);

        // de-serialization

        FileInputStream fis = **new** FileInputStream("abc.txt");

        ObjectInputStream ois = **new** ObjectInputStream(fis);

        Test output = (Test)ois.readObject();

        System.out.println("i = " + output.i);

        System.out.println("j = " + output.j);

        System.out.println("k = " + output.k);

        System.out.println("l = " + output.l);

        System.out.println("m = " + output.m);

    }

}

The main() method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the main() method by the JVM.