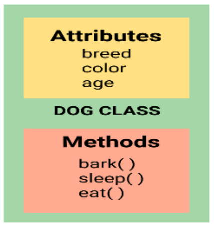
Introduction to Object Oriented Programming

Classes and Objects are basic concepts of Object Oriented Programming which revolve around real-life entities.

Class

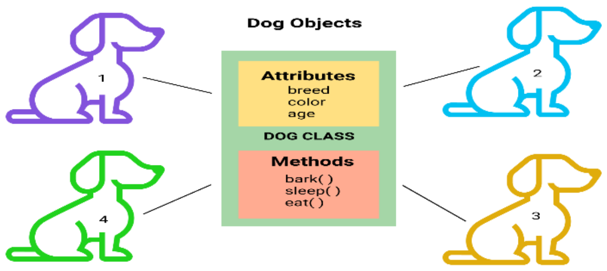
A class is a user-defined blueprint or prototype from which real-world objects are created. It represents the set of properties or methods that are common to all objects of one type.
We can call class as a collection of objects, which is a logical entity and does not take space in the memory.
Example : Dog Class can be represented as shown in the diagram below.



Object

It is a basic unit of Object Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods.
Objects having memory addresses and take up some space. They can interact with each other without knowing the data and code.
An object consists of:-

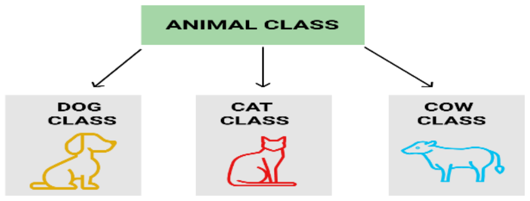
- **State** : It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior** : It is represented by the methods of an object. It also reflects the response of an object with other objects.
- **Identity** : It gives a unique name to an object and enables one object to interact with other objects.
  Example : Dog objects created from Dog Class



Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

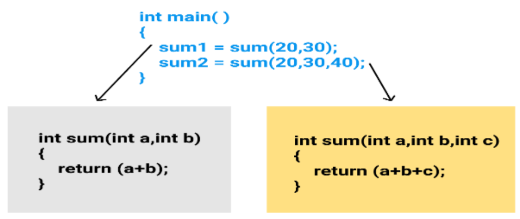
- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.
  Example : Dog, Cat, and Cow can the be derived classes of Animal base class.



Polymorphism

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.
A real-life example of polymorphism is that a person at the same time can have different characteristics. Like a man at the same time can be a father, a husband, and an employee. So the same person posses different behavior in different situations. This is called polymorphism.
Example: Suppose we have to write a function to add some integers, some times there are 2 integers and some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will

be called according to parameters.



```
int main( )
{
  sum1 = sum(20,30);
  sum2 = sum(20,30,40);
}

int sum(int a,int b)
{
  return (a+b);
}

int sum(int a,int b,int c)
{
  return (a+b+c);
}
```

Abstraction

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. **Ex: A car is viewed as a car rather than its individual components.**
Data Abstraction may also be defined as the process of identifying only the required characteristics of an object and ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.
Consider a **real-life example** of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying the brakes will stop the car, but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes, etc in the car. This is what abstraction is.

Advantages of Abstraction

- It reduces the complexity of viewing things.
- Avoids code duplication and increases reusability.
- Helps to increase the security of an application or program as only the important details are provided to the user.

Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from the other classes, so it is also known as **data-hiding**.
- Encapsulation can be achieved by declaring all the variables in the class as private and writing public methods in the class to set and get the values of the variables.

Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. He only knows that we are passing the values to a setter method and that the variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class as read-only or write-only, depending on our requirements. If we wish to make the variables as read-only then we have to omit the setter methods such as setName(), setAge() etc. or if we wish to make the variables as write-only then we have to omit the get methods such as getName(), getAge(), etc.
- **Reusability:** Encapsulation also improves the re-usability and makes the code easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Encapsulation vs Data Abstraction

- Encapsulation is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
- While encapsulation groups together the data and the methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of the implementation.

Encapsulation with Examples

Encapsulation is defined as the wrapping up of the data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

```
// Java program to demonstrate encapsulation
```

```java
public 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;
    }
}
```

In the above program, the class EncapsulateDemo is encapsulated as the variables are declared as private. The get methods - getAge(), getName(), and getRoll() are set as public. These methods are used to access the variables - geekAge, geekName, and geekRoll respectively. The setter methods - setName(), setAge(), and setRoll() are also declared as public and are used to set the values of the respective variables.

The program to access variables of the class EncapsulateDemo is shown below:

```java
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);
    }
}
```

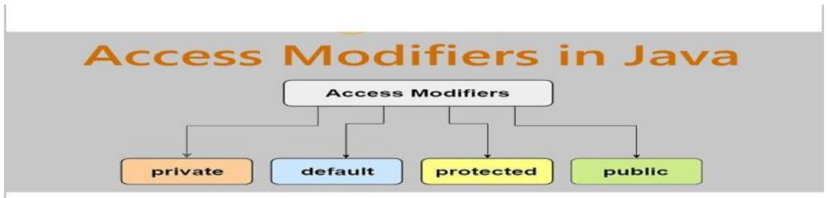
Output:

```
Geek's name: Harsh
Geek's age: 19
Geek's roll: 51
```

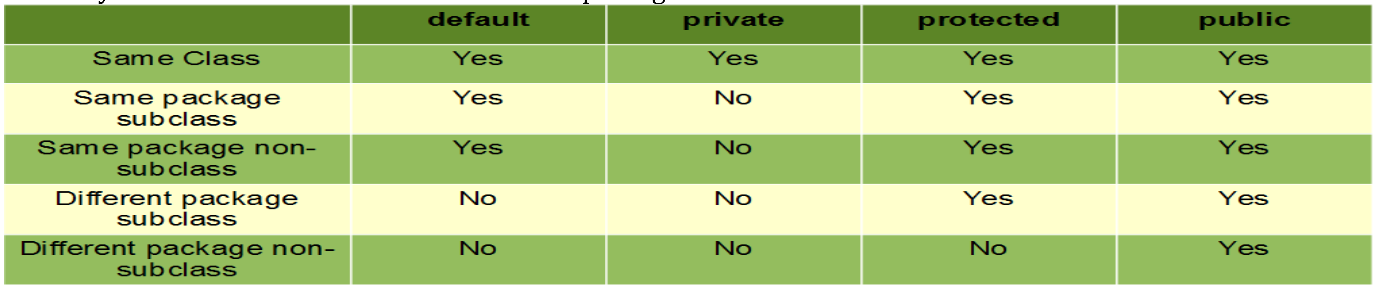
Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. He only knows that we are passing the values to a setter method and the variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirements. If we wish to make the variables as read-only then we have to omit the setter methods such as setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods such as getName(), getAge(), etc. from the above program
- **Reusability:** Encapsulation also improves the re-usability and makes the code easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Access Modifiers in Java

- 

- In Java, access modifiers are used to specify the accessibility of classes, methods, and variables within a program. There are four access modifiers in Java:
- **Public:** Public access modifier is the least restrictive modifier and allows classes, methods, and variables to be accessed from anywhere in the program. Any class or method that is marked as public can be accessed from any other class or method in the program.
- **Private:** Private access modifier is the most restrictive modifier and allows classes, methods, and variables to be accessed only within the same class in which they are defined. Any class or method that is marked as private cannot be accessed from any other class or method in the program.
- **Protected:** Protected access modifier allows classes, methods, and variables to be accessed within the same package or in a subclass outside of the package. Any class or method that is marked as protected can be accessed from any other class or method within the same package or in a subclass outside of the package.
- **Default:** Default access modifier is the one that is used when no access modifier is specified. It allows classes, methods, and variables to be accessed within the same package. Any class or method that is marked as default can be accessed from any other class or method within the same package.



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Here are some examples of how access modifiers can be used in Java:

```
    public class MyClass {
public int publicVar;
private int privateVar;
protected int protectedVar;
int defaultVar;

public void publicMethod() {
    // method code here
}

private void privateMethod() {
    // method code here
}

protected void protectedMethod() {
    // method code here
}

void defaultMethod() {
    // method code here
```

In this example, MyClass is a class with four variables and four methods, each with a different access modifier. The public variable and method can be accessed from any part of the program, the private variable and method can only be accessed within the same class, the protected variable and method can be accessed within the same package or in a subclass outside of the package, and the default variable and method can be accessed only within the same package.

This Reference in Java

In Java, the "this" keyword is a reference to the current object, i.e., the object on which the current method or constructor is being invoked. It is an implicit reference, meaning that it is automatically created by the Java runtime environment and is available within any non-static method or constructor of a class.
Here are some common uses of the "this" keyword in Java:
1.To access instance variables: When a method or constructor is invoked on an object, the "this" keyword can be used to refer to the object's instance variables. For example:
Java
```
public class MyClass {
    private int x;

    public void setX(int x) {
        this.x = x; // "this" refers to the current object
```
In this example, the "setX" method takes an integer parameter and assigns it to the instance variable "x" using the "this" keyword.
2.To call another constructor: When a constructor is overloaded in a class, one constructor can call another using the "this" keyword. This is useful when one constructor needs to perform some common initialization that is shared by all constructors. For example:
```
public class MyClass {
    private int x;
    private int y;

    public MyClass() {
        this(0, 0); // call the other constructor with x=0 and y=0
    }

    public MyClass(int x, int y) {
        this.x = x;
        this.y = y;
```
In this example, the default constructor of MyClass calls the other constructor with x=0 and y=0 using the "this" keyword.
3. To return the current object: A method can return the current object using the "this" keyword. This is useful when chaining method calls on the same object. For example:
Java
```
public class MyClass {
    private int x;

    public MyClass setX(int x) {
        this.x = x;
        return this; // return the current object
```
In this example, the "setX" method sets the value of x and returns the current object using the "this" keyword. This allows multiple method calls to be chained together, like this:
```
MyClass obj = new MyClass();
obj.setX(10).setX(20);
```

Final Keyword in Java

In Java, the "final" keyword is used to denote a constant or an unmodifiable entity. Once a variable, method or class is declared as final, it cannot be modified or overridden.
Here are some common uses of the "final" keyword in Java:
1.Final variables: Final variables are constants that cannot be changed once they are initialized. This is useful for defining values that should not be modified, such as mathematical constants or configuration parameters. For example:
Java
```
public class MyClass {
    private final int x = 10; // x is a final variable

    public void method() {
        // x = 20; // Error: cannot assign a value to final variable x
```
In this example, the variable "x" is declared as final and initialized to 10. The "method" method cannot modify the value of "x" because it is final.
2.Final methods: Final methods are methods that cannot be overridden by a subclass. This is useful for defining methods that should not be changed or modified, such as utility methods or methods with important functionality. For example:
```
public class MyClass {
    public final void method() {
        // method code here
public class MySubClass extends MyClass {
    // cannot override the "method" method because it is final
```
In this example, the "method" method is declared as final in the superclass "MyClass". The subclass "MySubClass" cannot override this method because it is final.

3.Final classes: Final classes are classes that cannot be subclassed. This is useful for defining classes that should not be extended or modified, such as utility classes or classes with important functionality. For example:
Java
```
public final class MyClass {
    // class code here
public class MySubClass extends MyClass {
    // Error: cannot extend final class MyClass
```
In this example, the "MyClass" class is declared as final, which means it cannot be subclassed. The "MySubClass" class cannot extend "MyClass" because it is final.

Static Members in Java

In Java, a static member is a member of a class that belongs to the class itself, rather than to any instance of the class. This means that there is only one copy of a static member shared by all instances of the class, and it can be accessed using the class name rather than an object reference.
Here are some common uses of static members in Java:
1.Static variables: Static variables are class-level variables that are shared by all instances of the class. They are often used to define constants or to maintain state that is shared across all instances of the class. For example:
Java
```
public class MyClass {
    private static int count = 0; // count is a static variable
    public MyClass() {
        count++;
    }
    public static int getCount() {
        return count;
```
In this example, the "count" variable is declared as static and is incremented each time a new instance of "MyClass" is created. The "getCount" method is also declared as static and returns the current count.
2. Static methods: Static methods are class-level methods that can be called without creating an instance of the class. They are often used to define utility methods or factory methods that do not depend on instance-specific state. For example:
Java
```
public class MyClass {
    public static int add(int a, int b) {
        return a + b;
```
In this example, the "add" method is declared as static and can be called using the class name rather than an instance of "MyClass".
3. Static blocks: Static blocks are used to initialize static variables or to perform any other static initialization that needs to be done. They are executed when the class is loaded into memory, before any instances of the class are created. For example:
Java
```
public class MyClass {
    private static int x;
    static {
        x = 10;
```
In this example, the "x" variable is declared as static and is initialized to 10 in the static block.
In summary, static members in Java are class-level members that are shared by all instances of the class. They include static variables, static methods, and static blocks, and are often used to define constants, utility methods, or to maintain state that is shared across all instances of the class.

Inheritance in Java

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Why Do We Need Java Inheritance?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

Important Terminologies Used in Java Inheritance

- **Class:** Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- **Super Class/Parent Class:** The class whose features are inherited is known as a superclass(or a base class or a parent class).

- **Sub Class/Child Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to Use Inheritance in Java?

The **extends keyword** is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, "extends" refers to increased functionality.

Syntax :

```
class derived-class extends base-class  {     //methods and fields   }
```

Inheritance in Java Example

Example: In the below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class that extends the Bicycle class and class Test is a driver class to run the program.

Java

```
// Java program to illustrate the
// concept of inheritance

// base class
class Bicycle {
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return ("No of gears are " + gear + "\n"
                + "speed of bicycle is " + speed);
    }
}

// derived class
class MountainBike extends Bicycle {

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear, int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    // overriding toString() method
    // of Bicycle to print more info
    @Override public String toString()
    {
        return (super.toString() + "\nseat height is "
                + seatHeight);
    }
}

// driver class
```

```
public class Test {
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

Output

```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

In the above program, when an object of MountainBike class is created, a copy of all methods and fields of the superclass acquires memory in this object. That is why by using the object of the subclass we can also access the members of a superclass.
Please note that during inheritance only the object of the subclass is created, not the superclass. For more, refer to Java Object Creation of Inherited Class.

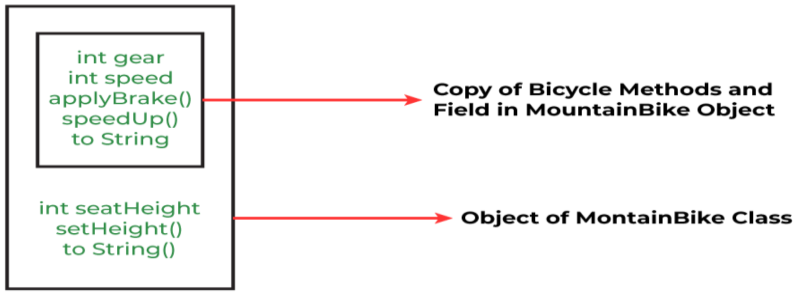
Example 2: In the below example of inheritance, class Employee is a base class, class Engineer is a derived class that extends the Employee class and class Test is a driver class to run the program.

```
// Java Program to illustrate Inheritance (concise)

import java.io.*;

// Base or Super Class
class Employee {
    int salary = 60000;
}

// Inherited or Sub Class
class Engineer extends Employee {
    int benefits = 10000;
}

// Driver Class
class Gfg {
    public static void main(String args[])
    {
        Engineer E1 = new Engineer();
        System.out.println("Salary : " + E1.salary
                           + "\nBenefits : " + E1.benefits);
    }
}
```

Output

```
Salary : 60000
Benefits : 10000
```

Illustrative image of the program:



int gear
int speed
applyBrake()
speedUp()
to String
int seatHeight
setHeight()
to String()
Copy of Bicycle Methods and Field in MountainBike Object
Object of MontainBike Class

In practice, inheritance, and polymorphism are used together in Java to achieve fast performance and readability of code.

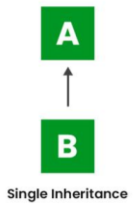
Java Inheritance Types

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

1. Single Inheritance

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



Single inheritance

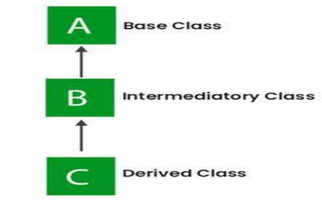
```
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class
class one {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class two extends one {
    public void print_for() { System.out.println("for"); }
}

// Driver class
public class Main {
      // Main function
    public static void main(String[] args)
    {
        two g = new two();
        g.print_geek();
        g.print_for();
        g.print_geek();
```

Output

```
Geeks
for
Geeks
```

2. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



Multilevel Inheritance

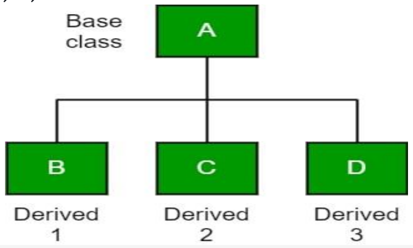
```
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

class one {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class two extends one {
    public void print_for() { System.out.println("for"); }
}

class three extends two {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

// Drived class
public class Main {
    public static void main(String[] args)
    {
        three g = new three();
        g.print_geek();
        g.print_for();
        g.print_geek();
```

Output

```
Geeks
for
Geeks
```

3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.



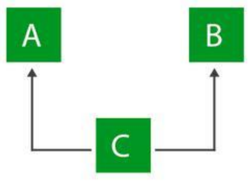
```
// Java program to illustrate the
// concept of Hierarchical  inheritance

class A {
    public void print_A() { System.out.println("Class A"); }
}

class B extends A {
    public void print_B() { System.out.println("Class B"); }
}

class C extends A {
    public void print_C() { System.out.println("Class C"); }
}

class D extends A {
    public void print_D() { System.out.println("Class D"); }
}

// Driver Class
public class Test {
    public static void main(String[] args)
    {
        B obj_B = new B();
        obj_B.print_A();
        obj_B.print_B();

        C obj_C = new C();
        obj_C.print_A();
        obj_C.print_C();

        D obj_D = new D();
        obj_D.print_A();
        obj_D.print_D();
```

Output

```
Class A
Class B
Class A
Class C
Class A
Class D
```

4. Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



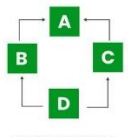
Multiple Inheritance

```
// Java program to illustrate the
// concept of Multiple inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

interface one {
    public void print_geek();
}

interface two {
    public void print_for();
}

interface three extends one, two {
    public void print_geek();
}
class child implements three {
    @Override public void print_geek()
    {
        System.out.println("Geeks");
    }

    public void print_for() { System.out.println("for"); }
}

// Drived class
public class Main {
    public static void main(String[] args)
    {
        child c = new child();
        c.print_geek();
        c.print_for();
        c.print_geek();
```
Output

```
Geeks
for
Geeks
```

5. Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance. However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.



Hybrid Inheritance

Java IS-A type of Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

Java

```
public class SolarSystem {
}
public class Earth extends SolarSystem {
}
public class Mars extends SolarSystem {
}
public class Moon extends Earth {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true:-

- SolarSystem is the superclass of Earth class.
- SolarSystem is the superclass of Mars class.
- Earth and Mars are subclasses of SolarSystem class.
- Moon is the subclass of both Earth and SolarSystem classes.

Java

```
class SolarSystem {
}
class Earth extends SolarSystem {
}
```

```java
class Mars extends SolarSystem {
}
public class Moon extends Earth {
    public static void main(String args[])
    {
        SolarSystem s = new SolarSystem();
        Earth e = new Earth();
        Mars m = new Mars();

        System.out.println(s instanceof SolarSystem);
        System.out.println(e instanceof Earth);
        System.out.println(m instanceof SolarSystem);
```

Output

```
true
true
true
```

What Can Be Done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in the example above, *toString()* method is overridden).
- We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Advantages Of Inheritance in Java:

1. Code Reusability: Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
2. Abstraction: Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.
3. Class Hierarchy: Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.
4. Polymorphism: Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways.

Disadvantages of Inheritance in Java:

1. Complexity: Inheritance can make the code more complex and harder to understand. This is especially true if the inheritance hierarchy is deep or if multiple inheritances is used.
2. Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.

Conclusion

Let us check some important points from the article are mentioned below:

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritances with classes. Although with interfaces, multiple inheritances are supported by Java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

FAQs in Inheritance

1. What is Inheritance Java?

Inheritance is a concept of OOPs where one class inherits from another class that can reuse the methods and fields of the parent class.

2. What are the 4 types of inheritance in Java?

There are Single, Multiple, Multilevel, and Hybrid.

3. What is the use of extend keyword?

Extend keyword is used for inheriting one class into another.

4. What is an example of inheritance in Java?
A real-world example of Inheritance in Java is mentioned below:
Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes.

Super keyword in Java

In Java, the "super" keyword is used to refer to the parent class of a class or to call a parent class's constructor or method. Here are some common uses of the "super" keyword in Java:

1. Call parent class constructor: When a class is inherited, the subclass must call the constructor of its parent class to initialize the inherited fields. This is done using the "super" keyword in the subclass constructor. For example:

```
public class ParentClass {
    public ParentClass(int x) {
        // parent class constructor code here
public class ChildClass extends ParentClass {
    public ChildClass(int x, int y) {
        super(x); // call parent class constructor
        // child class constructor code here
```

In this example, the "ChildClass" subclass calls the constructor of its "ParentClass" superclass using the "super" keyword.
2. Call parent class method: A subclass can call a method of its parent class using the "super" keyword. This is useful when the subclass overrides a parent class method but still wants to call the original implementation. For example:

```
public class ParentClass {
    public void method() {
        // parent class method code here
  public class ChildClass extends ParentClass {
    public void method() {
        super.method(); // call parent class method
        // child class method code here
```

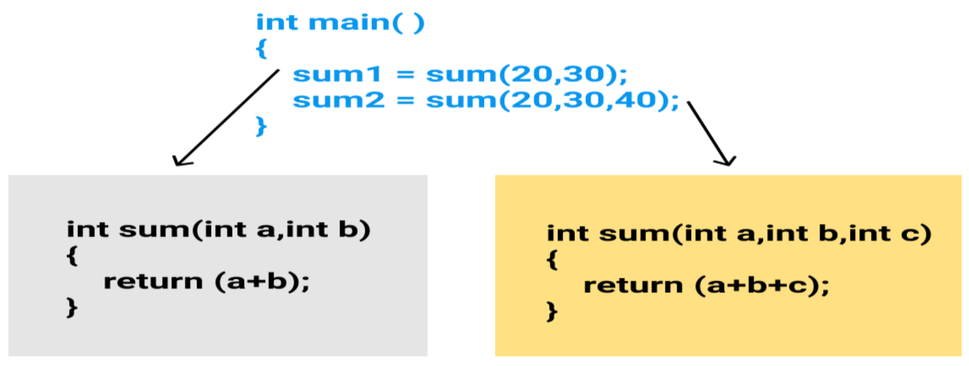
In this example, the "ChildClass" subclass overrides the "method" method of its "ParentClass" superclass but still calls the parent class's implementation using the "super" keyword.
3. Access parent class fields: A subclass can access a field of its parent class using the "super" keyword. This is useful when the subclass needs to access a field that is hidden by a field with the same name in the subclass. For example:

```
public class ParentClass {
    public int x = 10;
public class ChildClass extends ParentClass {
    public int x = 20;

    public void method() {
        System.out.println(super.x); // access parent class field
```

In this example, the "ChildClass" subclass has a field "x" that hides the "x" field of its "ParentClass" superclass. The "method" method of "ChildClass" uses the "super" keyword to access the parent class's "x" field.

Polymorphism with Examples

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.
A real-life example of polymorphism is that a person at the same time can have different characteristics. Like a man at the same time can be a father, a husband, and an employee. So the same person posses different behavior in different situations. This is called polymorphism.
Example: Suppose we have to write a function to add some integers, some times there are 2 integers and some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



```
int main( )
{
    sum1 = sum(20,30);
    sum2 = sum(20,30,40);
}
```

```
int sum(int a,int b)
{
    return (a+b);
}
```

```
int sum(int a,int b,int c)
{
    return (a+b+c);
}
```

Compile Time Polymorphism or Method Overloading

Early binding or Compile time polymorphism can be achieved by resolving the methods. The compiler resolves which method should be used based on the signature and number of parameters at compile time only.

Method overloading increases the **readability** and **maintainability** of the Java Program as we don't need to write separate method names performing same operations on different numbers or types of arguments.

- **Method Overloading: Changing number of Parameters**

```
class Addition
{
    static int sum(int a, int b)
    {
       return (a+b);
    }
    static int sum(int a, int b, int c)
    {
       return (a+b+c);
    }

}

class Test
{

    public static void main(String[] args)
    {
        //Don't need to create objects
        //We have defined sum Method as static
        System.out.println(Addition.sum(5,5));
        System.out.println(Addition.sum(5,5,5));
```

Output:

```
10
15
```

- **Method Overloading: Changing data type of Parameters**

Java

```
class Addition
{
    static int sum(int a, int b)
    {
       return (a+b);
    }
    static double sum(double a, double b)
    {
       return (a+b);
    }

}

class Test
{

    public static void main(String[] args)
    {
        //Don't need to create objects
        //We have defined sum Method as static
        System.out.println(Addition.sum(5,5));
        System.out.println(Addition.sum(5.2,5.5));
```

Output:

```
10
10.7
```

- **Why method overloading is not possible by changing the return type of method only?**

If we only change the return type, then method overloading cannot happen because there will be ambiguity between the methods having the same number and data type of parameters. Let's have a look at an example,

Java

```
class Addition
{
    static int sum(int a, int b)
    {
       return (a+b);
    }
    static double sum(int a, int b)
    {
       return (a+b);

class Test
{

    public static void main(String[] args)
    {
        //Don't need to create objects
        //We have defined sum Method as static
        System.out.println(Addition.sum(5,5));
        // Ambiguity which method to invoke !!
```

Output:

```
Compile Time Error: method sum(int, int) is already defined in class Addition
```

- **Can we Overload Java main() method?**
  Yes, We can write many main methods with different signatures but **JVM** invokes only the main method which receives String Array as an argument. Let's have a look at an example.
  Java

```
class Test
{

    public static void main(String[] args)
    {
        System.out.println("Main method with String [ ] arguments");
    }
    public static void main(String args)
    {
        System.out.println("Main method with String argument");
    }
    public static void main()
    {
        System.out.println("Main method with no argument");
```

Output:

```
Main method with String [ ] arguments
```

Run time Polymorphism or Method Overriding

In method overriding actual method is **resolved at run time**. When we have the same name method in Parent and Children class, then this conflict is resolved at the run time.
Method overriding is used to provide a **specific implementation** of a method that is already provided by its superclass.
Two rules must be followed to override the method:

1. Methods in parent and children class must have the same name.
2. Methods in parent and children class must have the same signature(data type, number of parameters).

- **Real-World Example of Method Overriding**
  There are several types of employees in the Company and they are having different bonuses based on their designation, Let's see how this can be implemented through method overriding.

```
class Employee
{
    int bonus()
    {
       return 500;
    }}

class Programmer extends Employee
{

    int bonus()
    {
       return 1000;
    }}
class Manager extends Employee
{
       int bonus()
    {
       return 2000;
    }}
class Test
{
    public static void main(String [] args)
    {
         Programmer p = new Programmer();
         Manager m = new Manager();
         System.out.println("Programmer's Bonus : "+ p.bonus());
         System.out.println("Manager's Bonus : "+ m.bonus());
    }}
```

Output:

```
Programmer's Bonus: 1000
Manager's Bonus: 2000
```

- **Can we override static method?**
  We can not override the static method because static methods do not belong to objects, it resides in the Class memory area and We can not override class methods as they are shared among all the objects.
- **Can we override the main method?**
  We can not override the main method because main is a static method.

Difference between Method Overloading and Method Overriding

Method Overloading	Method Overriding
Method overloading increases the readability of the program.	Method overriding is used for specific implementation of children class's object method already defined in super class.
In method overloading parameters must be different.	In method overriding parameters must be the same.
Method overloading is done between the same class methods.	Method overriding is done between children and parent class method.
Method overloading is compile time polymorphism.	Method overriding is run time polymorphism.

Abstraction with Examples

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object, ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

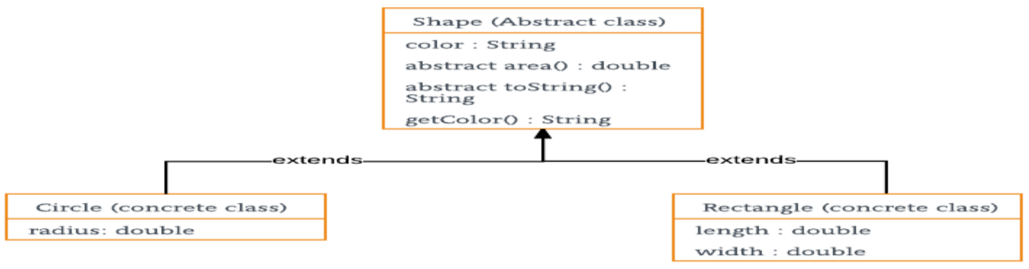
Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes, etc in the car. This is what abstraction is.
In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Abstract classes and Abstract methods :

1. An abstract class is a class that is declared with abstract keyword.
2. An abstract method is a method that is declared without an implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
7. An abstract class can have parametrized constructors and the default constructor is always present in an abstract class.

When to use abstract classes and abstract methods with an example

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
Consider a classic "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape" and each shape has a color, size and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle and so on - each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



```java
// Java program to illustrate the
// concept of Abstraction
abstract class Shape
{
    String color;
        // these are abstract methods
    abstract double area();
    public abstract String toString();
      // Abstract class can have a constructor
    public Shape(String color) {
        System.out.println("Shape constructor called");
        this.color = color;
    }
        // this is a concrete method
    public String getColor() {
        return color;
    }
}
class Circle extends Shape
{
    double radius;
       public Circle(String color,double radius) {
        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }
    @Override
    double area() {
        return Math.PI * Math.pow(radius, 2);
    }
    @Override
    public String toString() {
        return "Circle color is " + super.color +
                       " and area is : " + area();     }
}
class Rectangle extends Shape{

    double length;
    double width;
    public Rectangle(String color,double length,double width) {
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override
    double area() {
        return length*width;
    }
    @Override
    public String toString() {
        return "Rectangle color is " + super.color +
                           " and area is : " + area();
    }}
public class Test
{
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
```
Output:
```
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Red and area is : 15.205308443374602
Rectangle color is Yellow and area is : 8.0
```
Advantages of Abstraction

1. It reduces the complexity of viewing things.
2. Avoids code duplication and increases reusability.
3. Helps to increase the security of an application or program as only important details are provided to the user.

Interfaces in Java

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, without body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, the class must be declared abstract.
- A Java library example is the Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Syntax :

```
interface <interface_name> {
    // 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 the 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 interface use **implements** keyword.

Why do we use interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in the case of class, but by using interface it can achieve multiple inheritance.
- 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 that the abstract classes may contain non-final variables, whereas variables in interface are final, public, and static.

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

To implement an interface we use the keyword: implement

```
// 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
```

A real-world example:

Let's consider the example of vehicles like bicycles, cars, bike………, they have common functionalities. So we make an interface and put all these common functionalities and let the Bicycle, Bike, Car ….etc implement all these functionalities in their own class in their own way.

```
import java.io.*;

interface Vehicle {
```

```java
    // 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 inatance 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
```

Important points about interface

- We can't create an instance(an interface can't be instantiated) of an interface, but we can make reference of it that refers to the object of its implementing class.
- A class can implement more than one interface.
- An interface can extend another interface or interfaces (more than one interface).
- A class that implements an 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 inheritance.
- It is used to achieve loose coupling.

Abstract Class vs Interface in Java

Abstraction: Hiding the internal implementation of the feature and only showing the functionality to the users. i.e. what it works (showing), how it works (hiding). Both abstract class and interface are used for abstraction.

Abstract class vs Interface

1. **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
2. **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
3. **Type of variables:** Abstract class can have final, non-final, static, and non-static variables. Interface has only static and final variables.
4. **Implementation:** Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.
5. **Inheritance vs Abstraction:** A Java interface can be implemented using keyword the "implements" and an abstract class can be extended using the keyword "extends".
6. **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members such as private, protected, etc.

When to use what?

Consider using **abstract classes** if any of these statements apply to your situation:

- In java application, there are some related classes that need to share some lines of code then you can put these lines of code within an abstract class and this abstract class should be extended by all these related classes.
- You can define non-static or non-final field(s) in abstract class so that via a method you can access and modify the state of the object to which they belong.
- You can expect that the classes that extend an abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

Consider using **interfaces** if any of these statements apply to your situation:

- It is total abstraction, All methods declared within an interface must be implemented by the class(es) that implements this interface.
- A class can implement more than one interface. It is called multiple inheritance.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.