Object Oriented Programming (OOPs) Concept in Java

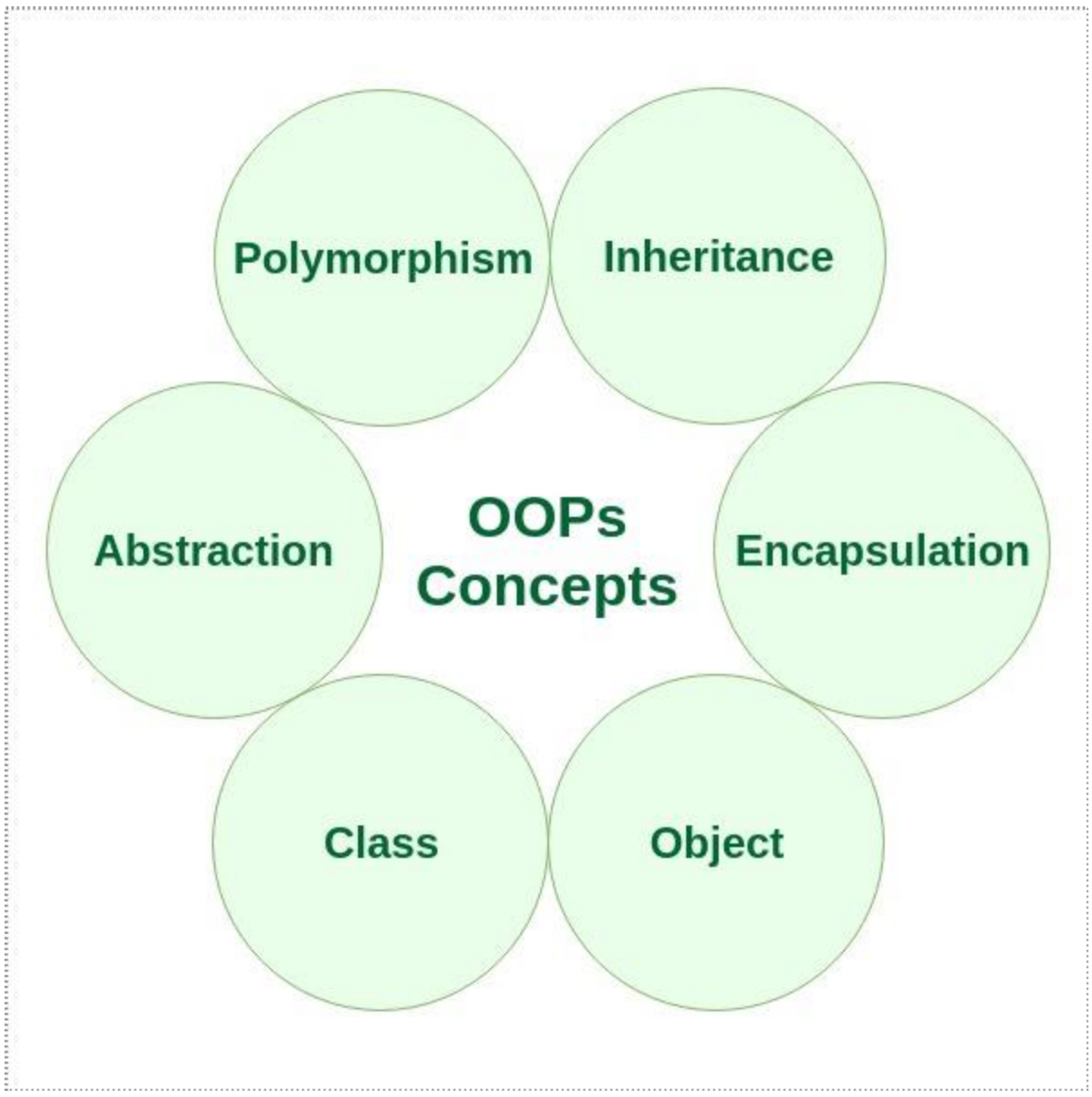
As the name suggests, Object-Oriented Programming or OOPs refers to languages that uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.
Let us do discuss pre-requisite by polishing concepts of methods declaration and passing. Starting off with the method declaration, it consists of six components:

- **Access Modifier: Defines access type of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.**
  - **public: accessible in all class in your application.**
  - **protected: accessible within the package in which it is defined and in its subclass(es)(including subclasses declared outside the package)**
  - **private: accessible only within the class in which it is defined.**
  - **default (declared/defined without using any modifier): accessible within same class and package within which its class is defined.**
- **The return type: The data type of the value returned by the method or void if does not return a value.**
- **Method Name: the rules for field names apply to method names as well, but the convention is a little different.**
- **Parameter list: Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().**
- **Exception list: The exceptions you expect by the method can throw, you can specify these exception(s).**
- **Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations.**

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.
Now with basic prerequisite to step learning 4 pillar of OOPS is as follows. Let us start with learning about the different characteristics of an Object-Oriented Programming language

OOPs Concepts are as follows:

1. **Class**
2. **Object**
3. **Method and method passing**
4. **Pillars of OOPS**
   - **Abstraction**
   - **Encapsulation**
   - **Inheritance**
   - **Polymorphism**
     - **Compile-time polymorphism**
     - **Run-time polymorphism**



A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers: A class can be public or has default access (Refer this for details).**
2. **Class name: The name should begin with a initial letter (capitalized by convention).**
3. **Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.**
4. **Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.**
5. **Body: The class body surrounded by braces, { }.**

Object 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. An object consists of:

1. **State : It is represented by attributes of an object. It also reflects the properties of an object.**
2. **Behavior : It is represented by methods of an object. It also reflects the response of an object with other objects.**
3. **Identity : It gives a unique name to an object and enables one object to interact with other objects.**
4. **Method: A method is a collection of statements that perform some specific task and return result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++ and Python. Methods are time savers and help us to reuse the code without retyping the code.**

Let us now discuss 4 pillars of OOPS:

Pillar 1: 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 ignoring the irrelevant details. The properties and behaviours 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 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.

Pillar 2: Encapsulation

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, 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 is 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.**

Pillar 3: Inheritence

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

Let us discuss some of frequent used important terminologies:

- **Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).**
- **Sub Class: The class that inherits the other class is known as 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.**

Pillar 4: Polymorphism

It refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities.

Note: Polymorphism in Java are mainly of 2 types:

1. ***Overloading***
2. ***Overriding***

Example

- **Java**

```
// Java program to Demonstrate Polymorphism



// This class will contain

// 3 methods with same name,

// yet the program will

// compile & run successfully

public class Sum {

// Overloaded sum().

// This sum takes two int parameters

public int sum(int x, int y)

{

    return (x + y);

}



// Overloaded sum().

// This sum takes three int parameters

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

```
{
    return (x + y + z);
}

// Overloaded sum().
// This sum takes two double parameters
public double sum(double x, double y)
{
    return (x + y);
}

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output:

```
30
60
31.0
```

Classes and Objects in Java

- **Difficulty Level : Medium**
- **Last Updated : 28 Jun, 2021**

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

Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers: A class can be public or has default access (Refer this for details).**
2. **class keyword: class keyword is used to create a class.**
3. **Class name: The name should begin with an initial letter (capitalized by convention).**
4. **Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.**
5. **Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.**
6. **Body: The class body surrounded by braces, { }.**

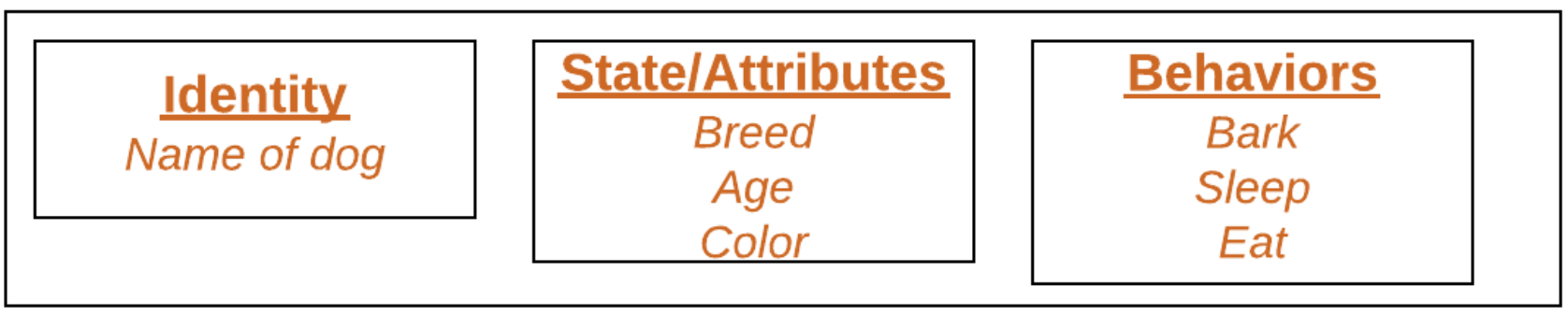
Constructors are used for initializing new objects. Fields are variables that provides the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.
There are various types of classes that are used in real time applications such as nested classes, anonymous classes, lambda expressions.

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. An object consists of :

1. **State: It is represented by attributes of an object. It also reflects the properties of an object.**
2. **Behavior: It is represented by methods of an object. It also reflects the response of an object with other objects.**
3. **Identity: It gives a unique name to an object and enables one object to interact with other objects.**

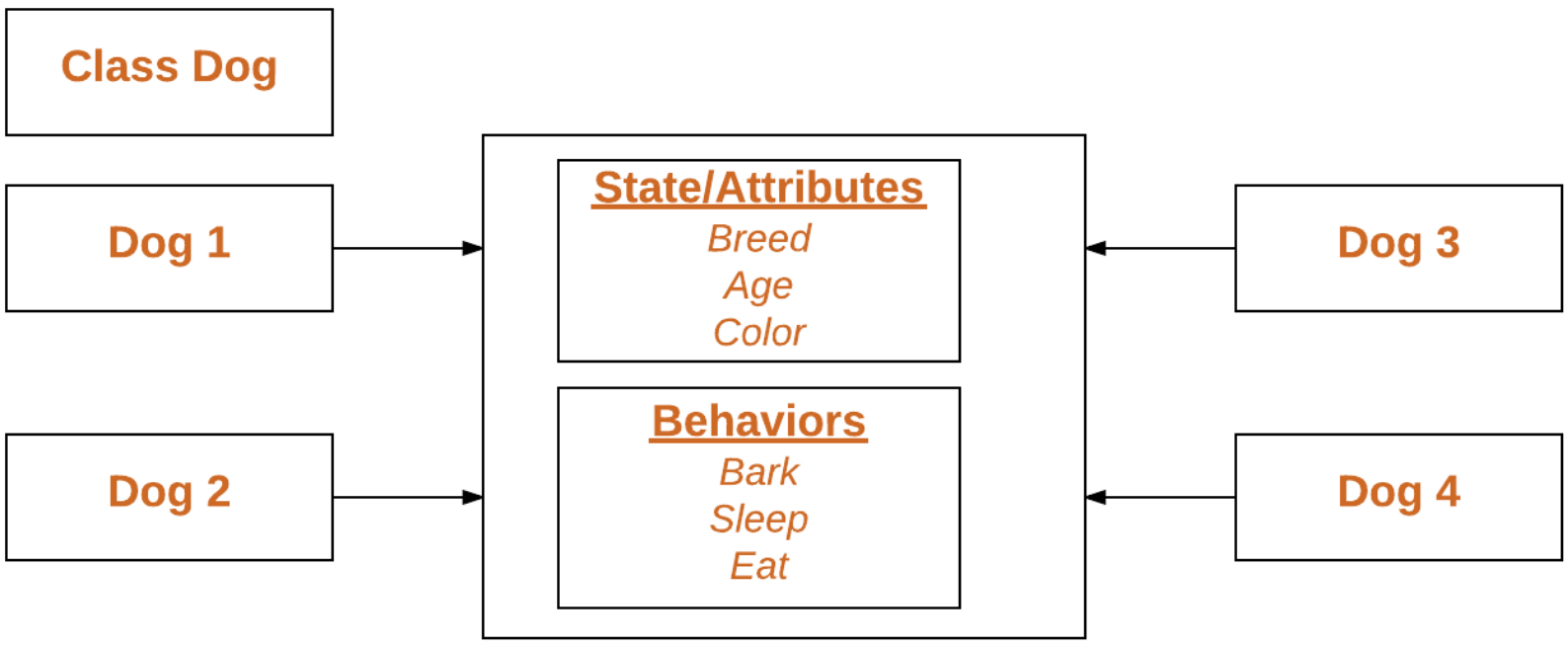
Example of an object: dog



Objects correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product".

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.
Example:



As we declare variables like (type name;). This notifies the compiler that we will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variable, type must be strictly a concrete class name. In general, we can't create objects of an abstract class or an interface.

```
Dog tuffy;
```

If we declare reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

Initializing an object

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

- **Java**

```
// Class Declaration



public class Dog

{

    // Instance Variables

    String name;

    String breed;

    int age;

    String color;



    // Constructor Declaration of Class

    public Dog(String name, String breed,

                   int age, String color)

    {
```

```
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }


    // method 1
    public String getName()
    {
        return name;
    }

    // method 2
    public String getBreed()
    {
        return breed;
    }


    // method 3
    public int getAge()
    {
        return age;
    }

    // method 4
    public String getColor()
    {
        return color;
    }


    @Override
    public String toString()
    {
```

```
    return("Hi my name is "+ this.getName()+
           ".\nMy breed,age and color are " +
           this.getBreed()+"," + this.getAge()+
           ","+ this.getColor());
}

public static void main(String[] args)
{
    Dog tuffy = new Dog("tuffy","papillon", 5, "white");
    System.out.println(tuffy.toString());
}
```

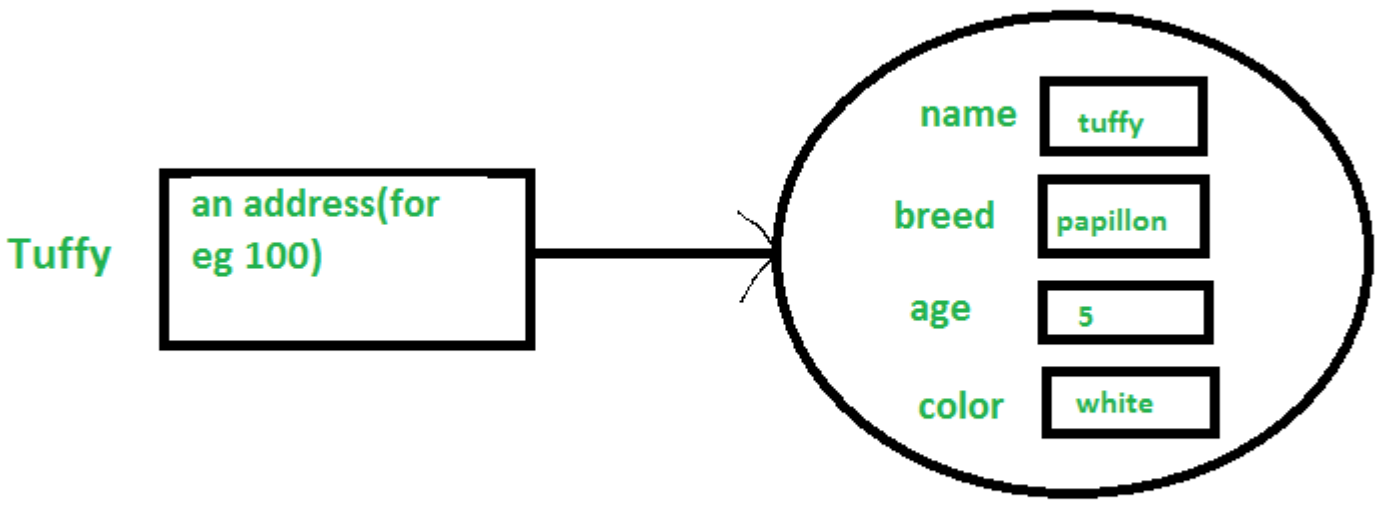
Output:

```
Hi my name is tuffy.
My breed,age and color are papillon,5,white
```

- **This class contains a single constructor. We can recognize a constructor because its declaration uses the same name as the class and it has no return type. The Java compiler differentiates the constructors based on the number and the type of the arguments. The constructor in the *Dog* class takes four arguments. The following statement provides "tuffy","papillon",5,"white" as values for those arguments:**

```
Dog tuffy = new Dog("tuffy","papillon",5, "white");
```

- **The result of executing this statement can be illustrated as :**



Note : All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, also called the default constructor. This default constructor calls the class parent's no-argument constructor (as it contain only one statement i.e super();), or the *Object* class constructor if the class has no other parent (as Object class is parent of all classes either directly or indirectly).

Ways to create object of a class

There are four ways to create objects in java.Strictly speaking there is only one way(by using *new* keyword),and the rest internally use *new* keyword.

- **Using new keyword: It is the most common and general way to create object in java. Example:**

```
// creating object of class Test

Test t = new Test();
```

- **Using Class.forName(String className) method: There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name.We have to give the fully qualified name for a class. On calling new Instance() method on this Class object returns new instance of the class with the given string name.**

```
// creating object of public class Test

// consider class Test present in com.p1 package
Test obj = (Test)Class.forName("com.p1.Test").newInstance();
```

- **Using clone() method: clone() method is present in Object class. It creates and returns a copy of the object.**

```
// creating object of class Test

Test t1 = new Test();

// creating clone of above object

Test t2 = (Test)t1.clone();
```

- **Deserialization: De-serialization is technique of reading an object from the saved state in a file. Refer Serialization/De-Serialization in java**

```
FileInputStream file = new FileInputStream(filename);

ObjectInputStream in = new ObjectInputStream(file);

Object obj = in.readObject();
```

Creating multiple objects by one type only (A good practice)

- **In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case, wastage of memory is less. The objects that are not referenced anymore will be destroyed by Garbage Collector of java. Example:**

```
Test test = new Test();

test = new Test();
```

- **In inheritance system, we use parent class reference variable to store a sub-class object. In this case, we can switch into different subclass objects using same referenced variable. Example:**

```
class Animal {}

class Dog extends Animal {}
class Cat extends Animal {}

public class Test
{
    // using Dog object
    Animal obj = new Dog();

    // using Cat object
    obj = new Cat();
}
```

Anonymous objects

Anonymous objects are the objects that are instantiated but are not stored in a reference variable.

- They are used for immediate method calling.
- They will be destroyed after method calling.
- They are widely used in different libraries. For example, in AWT libraries, they are used to perform some action on capturing an event(eg a key press).
- In the example below, when a key is button(referred by the btn) is pressed, we are simply creating anonymous object of EventHandler class for just calling handle method.

```
btn.setOnAction(new EventHandler()

{

    public void handle(ActionEvent event)

    {

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

    }

});
```

Inheritance in Java

- Difficulty Level : Easy
- Last Updated : 28 Jun, 2021

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.

Important terminology:

- **Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).**
- **Sub 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 keyword used for inheritance is extends.
Syntax :

```
class derived-class extends base-class

{

   //methods and fields

}
```

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

    }

}
```

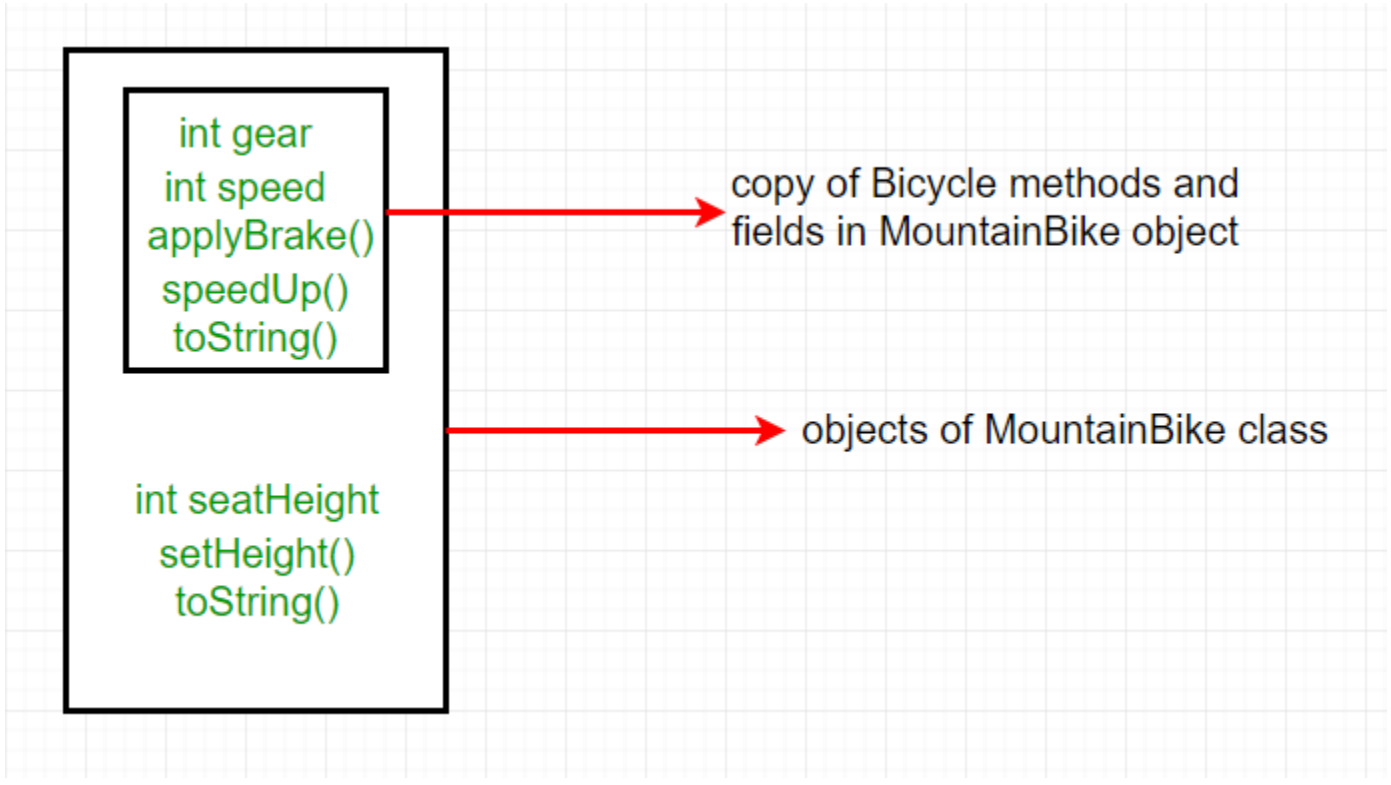
Output

```
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 acquire 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 Java Object Creation of Inherited Class.

Illustrative image of the program:



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

Types of Inheritance in Java

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

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.

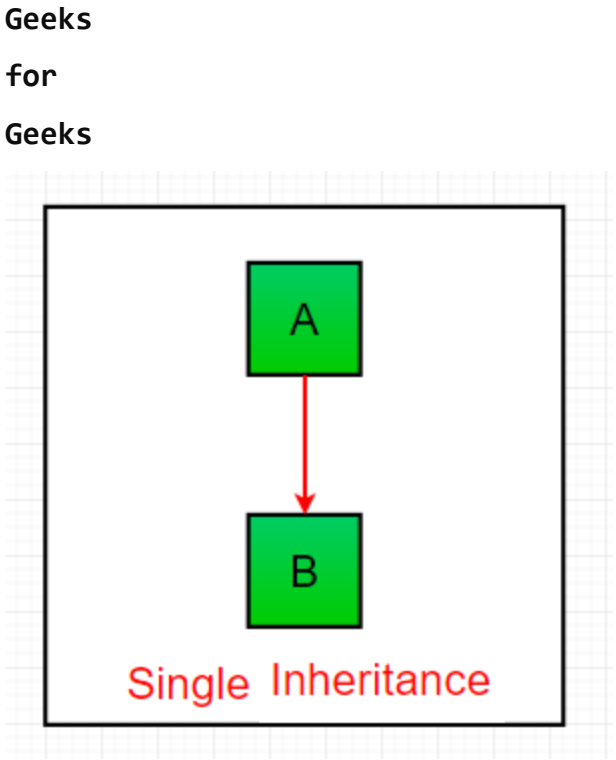
- **Java**

```
// Java program to illustrate the

// concept of single 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"); }

}

// Driver class

public class Main {

    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 act as the base class to other class. 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.

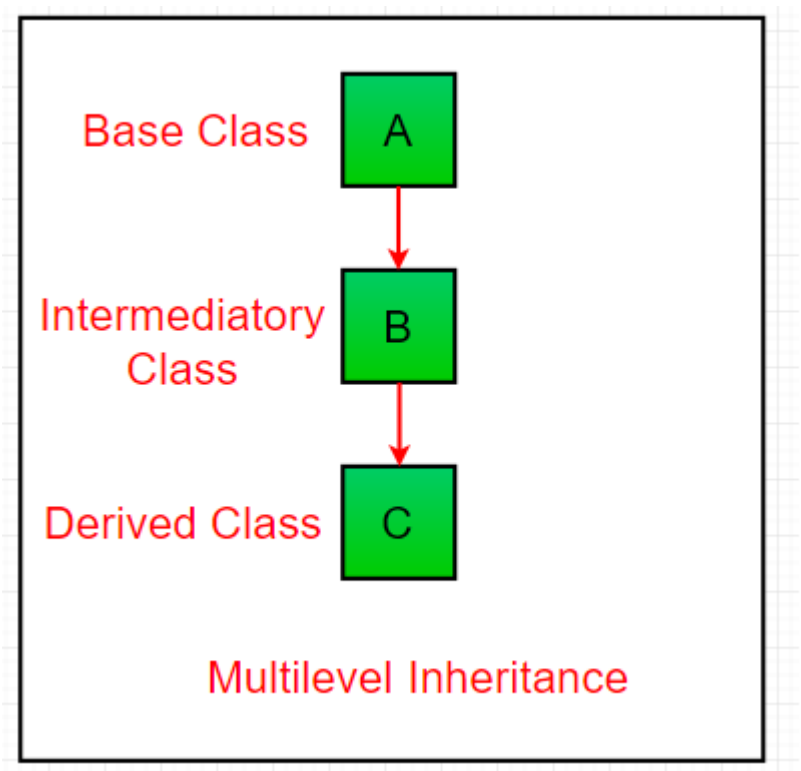
- **Java**

```
// 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 class B, C and D.

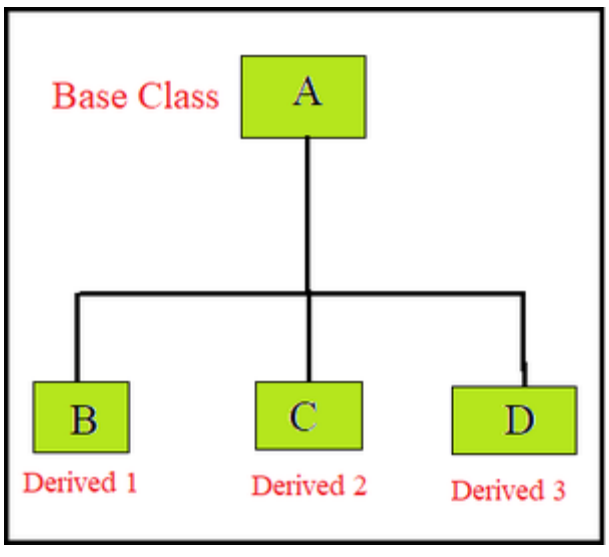
- **Java**

```
// 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
```



Hierarchical Inheritance

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 interface A and B.

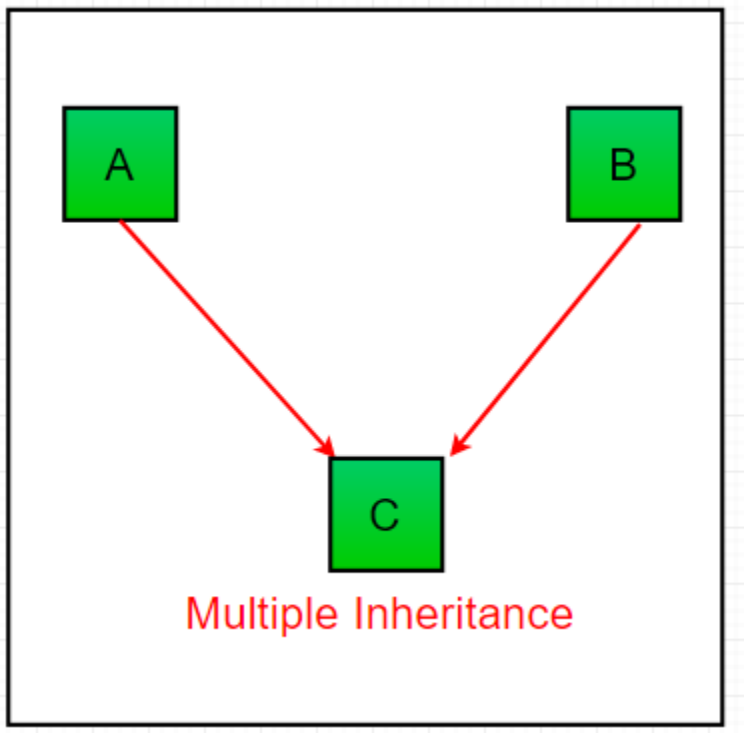
- **Java**

```
// 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 {
```

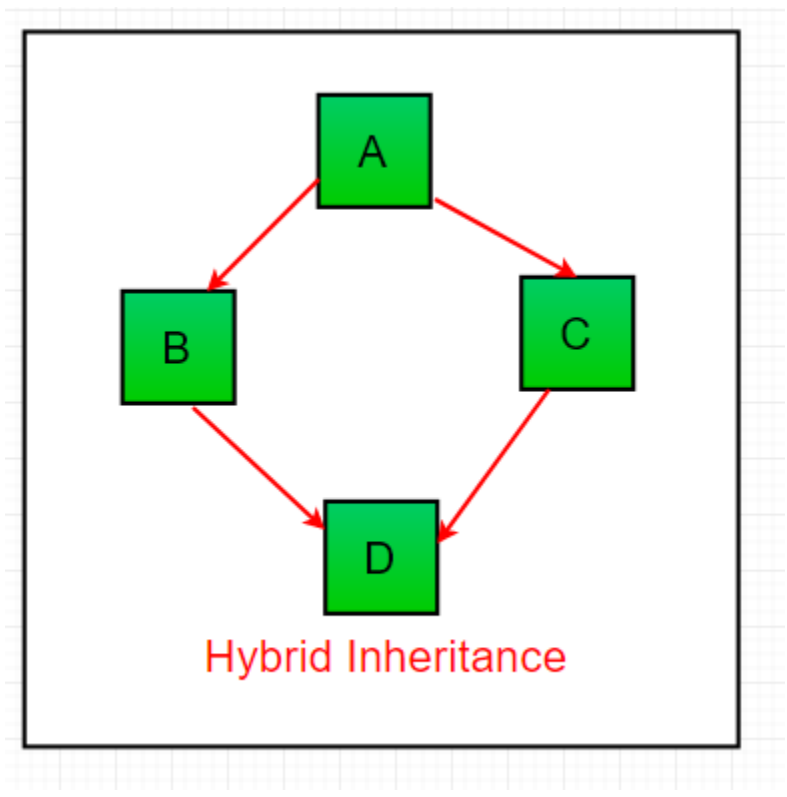
```
    public static void main(String[] args)
    {
        child c = new child();
        c.print_geek();
        c.print_for();
        c.print_geek();
    }
}
```

Output

```
Geeks
for
Geeks
```



5. Hybrid Inheritance(Through Interfaces): 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 is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Important facts about inheritance in Java

- **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.**

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:-

1. **SolarSystem the superclass of Earth class.**
2. **SolarSystem the superclass of Mars class.**
3. **Earth and Mars are subclasses of SolarSystem class.**
4. **Moon is the subclass of both Earth and SolarSystem classes.**

- **Java**

```
class SolarSystem {

}

class Earth extends SolarSystem {

}

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 all 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:

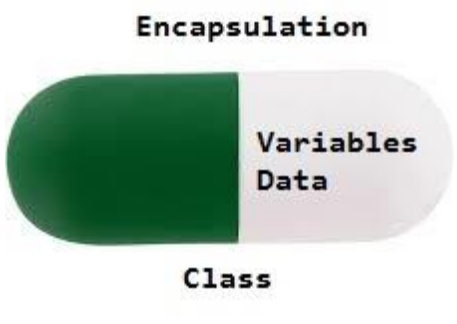
- **The inherited fields can be used directly, just like any other fields.**
- **We can declare new fields in the subclass that are not in the superclass.**
- **The inherited methods can be used directly as they are.**
- **We can write a new *instance* method in the subclass that has the same signature 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.**

Encapsulation in Java

- **Difficulty Level : Easy**
- **Last Updated : 02 Aug, 2021**

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, 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 is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.**
- **As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a combination of data-hiding and abstraction.**
- **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**



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

- **Java**

```
// 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; }
}

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
```

In the above program, the class Encapsulate is encapsulated as the variables are declared as private. The get methods like getAge() , getName() , getRoll() are set as public, these methods are used to access these variables. The setter methods like setName(), setAge(), setRoll() are also declared as public and are used to set 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. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.**
- **Increased Flexibility: We can make the variables of the class read-only or write-only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like 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 like getName(), getAge(), etc. from the above program**
- **Reusability: Encapsulation also improves the re-usability and is easy to change with new requirements.**
- **Testing code is easy: Encapsulated code is easy to test for unit testing.**

Polymorphism in Java

- **Difficulty Level : Easy**
- **Last Updated : 01 Dec, 2020**

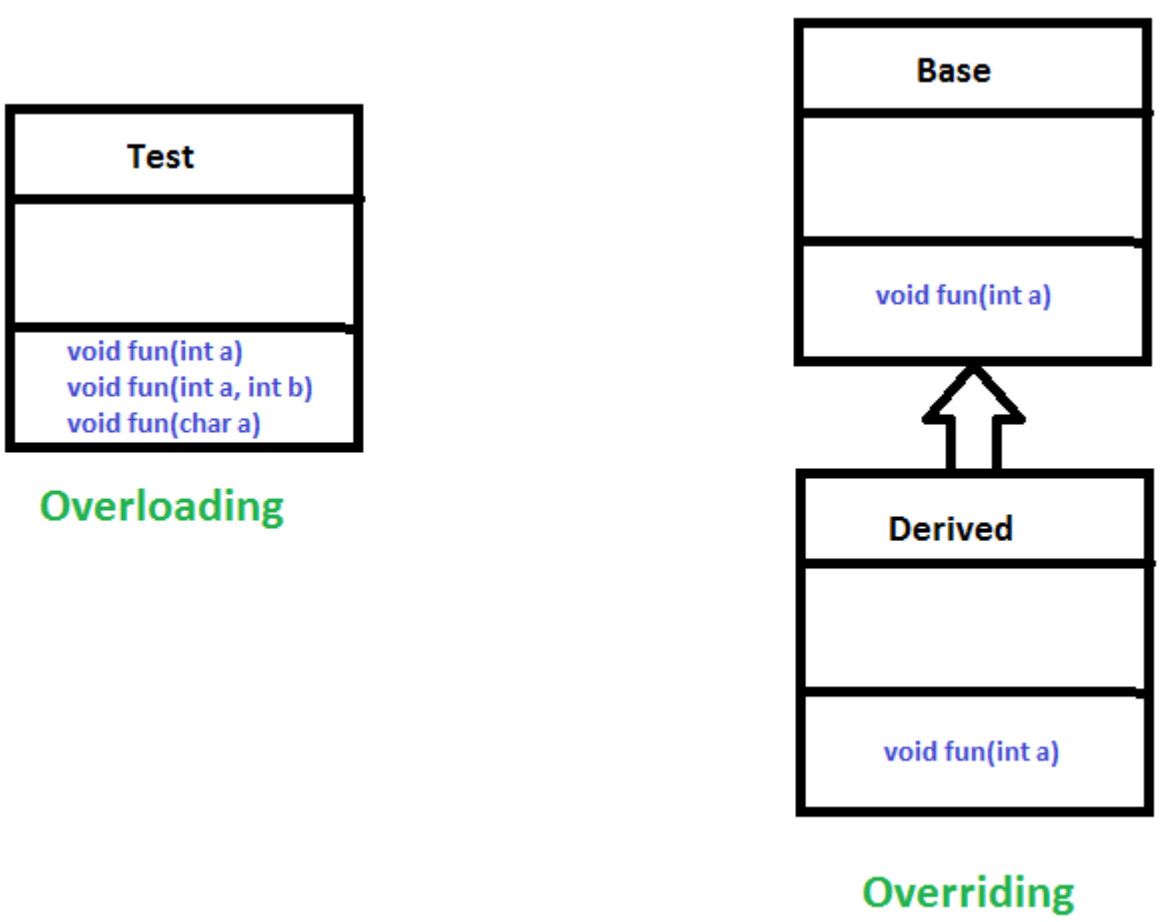
The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.
Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.

In Java polymorphism is mainly divided into two types:

- **Compile time Polymorphism**
- **Runtime Polymorphism**

1. Compile-time polymorphism: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But Java doesn't support the Operator Overloading.



Method Overloading: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.
Example: By using different types of arguments

```
// Java program for Method overloading


class MultiplyFun {


    // Method with 2 parameter
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method with the same name but 2 double parameter
    static double Multiply(double a, double b)
    {
        return a * b;
    }
}


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


        System.out.println(MultiplyFun.Multiply(2, 4));

        System.out.println(MultiplyFun.Multiply(5.5, 6.3));

    }

}
```

Output:

```
8

34.65
```

Example 2: By using different numbers of arguments

```
// Java program for Method overloading
```

```
class MultiplyFun {



    // Method with 2 parameter

    static int Multiply(int a, int b)

    {

        return a * b;

    }



    // Method with the same name but 3 parameter

    static int Multiply(int a, int b, int c)

    {

        return a * b * c;

    }

}



class Main {

    public static void main(String[] args)

    {

        System.out.println(MultiplyFun.Multiply(2, 4));



        System.out.println(MultiplyFun.Multiply(2, 7, 3));

    }

}
```

Output:
8

42

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

```
// Java program for Method overriding



class Parent {
```

```
    void Print()
    {
        System.out.println("parent class");
    }
}

class subclass1 extends Parent {

    void Print()
    {
        System.out.println("subclass1");
    }
}

class subclass2 extends Parent {

    void Print()
    {
        System.out.println("subclass2");
    }
}

class TestPolymorphism3 {
public static void main(String[] args)
{

    Parent a;

    a = new subclass1();
    a.Print();

    a = new subclass2();
    a.Print();
```

```
    }

}
```

Output:

```
subclass1

subclass2
```

Abstraction in Java

- **Difficulty Level : Medium**
- **Last Updated : 17 Sep, 2021**

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 behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

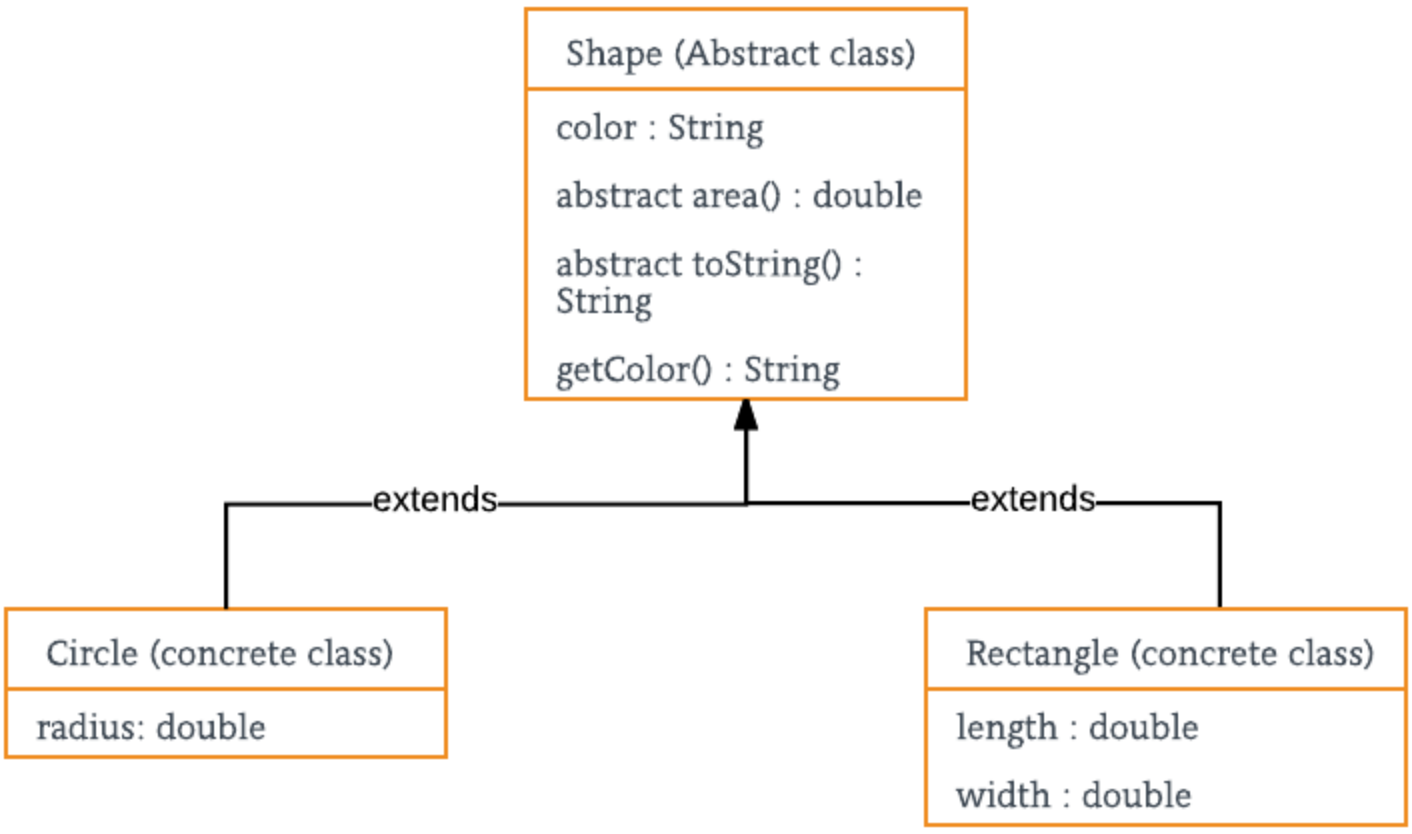
Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the 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 an abstract keyword.**
2. **An abstract method is a method that is declared without 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 the subclass itself abstract.**
5. **Any class that contains one or more abstract methods must also be declared with an 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 parameterized 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 the 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.getColor()
            + "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.getColor()
            + "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 Redand area is : 15.205308443374602
Rectangle color is Yellowand area is : 8.0
```

Encapsulation vs Data Abstraction

1. **Encapsulation is data hiding(information hiding) while Abstraction is detailed hiding(implementation hiding).**
2. **While encapsulation groups together data and methods that act upon the data, data abstraction deal with exposing the interface to the user and hiding the details of implementation.**

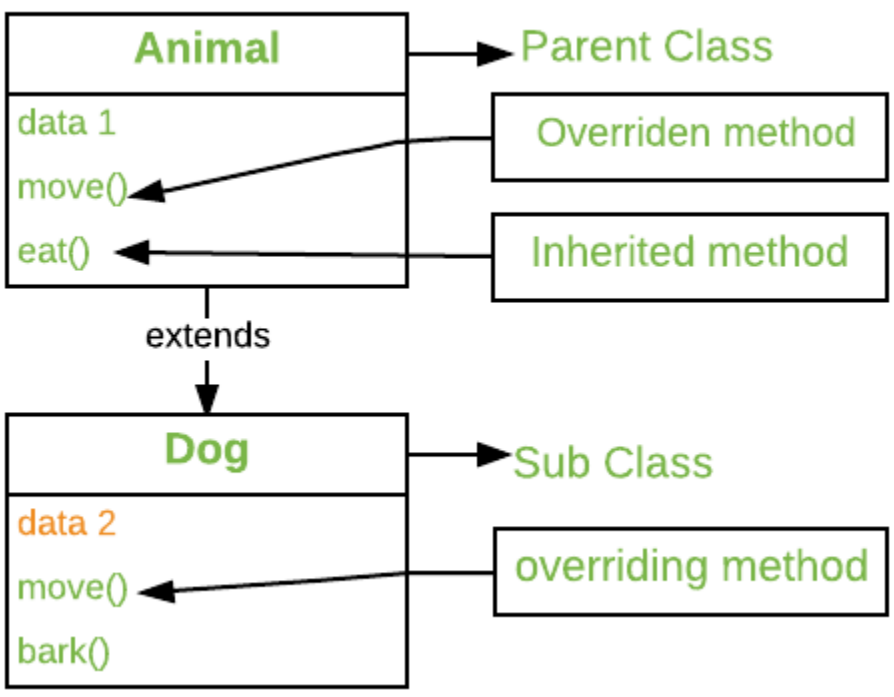
Advantages of Abstraction

1. **It reduces the complexity of viewing the 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.**

Overriding in Java

- **Difficulty Level : Medium**
- **Last Updated : 28 Jun, 2021**

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.



Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

```
// A Simple Java program to demonstrate

// method overriding in java

// Base Class
class Parent {
    void show()
    {
        System.out.println("Parent's show()");
    }
}


// Inherited class
```

```
class Child extends Parent {

    // This method overrides show() of Parent

    @Override

    void show()

    {

        System.out.println("Child's show()");

    }

}



// Driver class

class Main {

    public static void main(String[] args)

    {

        // If a Parent type reference refers

        // to a Parent object, then Parent's

        // show is called

        Parent obj1 = new Parent();

        obj1.show();



        // If a Parent type reference refers

        // to a Child object Child's show()

        // is called. This is called RUN TIME

        // POLYMORPHISM.

        Parent obj2 = new Child();

        obj2.show();

    }

}
```

Output:

```
Parent's show()

Child's show()
```

Rules for method overriding:

1. **Overriding and Access-Modifiers : The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the**

super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

```
// A Simple Java program to demonstrate

// Overriding and Access-Modifiers



class Parent {

    // private methods are not overridden

    private void m1()

    {

        System.out.println("From parent m1()");

    }

    protected void m2()

    {

        System.out.println("From parent m2()");

    }

}



class Child extends Parent {

    // new m1() method

    // unique to Child class

    private void m1()

    {

        System.out.println("From child m1()");

    }



    // overriding method

    // with more accessibility

    @Override

    public void m2()

    {

        System.out.println("From child m2()");

    }

}
```

```
// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Parent();
        obj1.m2();
        Parent obj2 = new Child();
        obj2.m2();
    }
}
```

2. **Output:**
3. `From parent m2()`
4. `From child m2()`
5.
6. **Final methods can not be overridden : If we don't want a method to be overridden, we declare it as final. Please see Using final with Inheritance .**

```
// A Java program to demonstrate that
// final methods cannot be overridden


class Parent {
    // Can't be overridden
    final void show() {}
}
```

```
class Child extends Parent {
    // This would produce error
    void show() {}
}
```

7. **Output:**
8. `13: error: show() in Child cannot override show() in Parent`
9. `    void show() {  }`
10. `             ^`
11. `       overridden method is final`
12. **Static methods can not be overridden(Method Overriding vs Method Hiding) : When you define a static method with same signature as a static method in base class, it is known as method hiding. The following table summarizes what happens when you define a method with the same signature as a method in a super-class.**

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	**Overrides**	**Generates a compile-time error**
Subclass Static Method	**Generates a compile-time error**	**Hides**

```
// Java program to show that
// if the static method is redefined by
// a derived class, then it is not
// overriding, it is hiding

class Parent {
// Static method in base class
// which will be hidden in subclass
static void m1()
{
    System.out.println("From parent "
                       + "static m1()");
}

// Non-static method which will
// be overridden in derived class
void m2()
{
    System.out.println("From parent "
                       + "non-static(instance) m2()");
    }
}

class Child extends Parent {
    // This method hides m1() in Parent
    static void m1()
    {
        System.out.println("From child static m1()");
    }
```

```
    // This method overrides m2() in Parent
    @Override
    public void m2()
    {
        System.out.println("From child "
                           + "non-static(instance) m2()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Child();

        // As per overriding rules this
        // should call to class Child static
        // overridden method. Since static
        // method can not be overridden, it
        // calls Parent's m1()
        obj1.m1();

        // Here overriding works
        // and Child's m2() is called
        obj1.m2();
    }
}
```

Output:

```
From parent static m1()
From child non-static(instance) m2()
```

13. **Private methods can not be overridden : Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass.(See this for details).**

14. The overriding method must have same return type (or subtype) : From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomena is known as <u>covariant return type</u>.

15. Invoking overridden method from sub-class : We can call parent class method in overriding method using <u>super keyword</u>.

```
// A Java program to demonstrate that overridden

// method can be called from sub-class



// Base Class

class Parent {

    void show()

    {

        System.out.println("Parent's show()");

    }

}



// Inherited class

class Child extends Parent {

    // This method overrides show() of Parent

    @Override

    void show()

    {

        super.show();

        System.out.println("Child's show()");

    }

}



// Driver class

class Main {

    public static void main(String[] args)

    {

        Parent obj = new Child();

        obj.show();

    }
```

```
}
```

16. **Output:**
17. `Parent's show()`
18. `Child's show()`
19. **Overriding and constructor : We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).**
20. **Overriding and Exception-Handling : Below are two rules to note when overriding methods related to exception-handling.**
    - **Rule#1 : If the super-class overridden method does not throw an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.**

```
/* Java program to demonstrate overriding when

  superclass method does not declare an exception

*/



class Parent {

    void m1()

    {

        System.out.println("From parent m1()");

    }



    void m2()

    {

        System.out.println("From parent  m2()");

    }

}



class Child extends Parent {

    @Override

    // no issue while throwing unchecked exception

    void m1() throws ArithmeticException

    {

        System.out.println("From child m1()");

    }



    @Override
```

```
        // compile-time error

        // issue while throwin checked exception

        void m2() throws Exception

        {

            System.out.println("From child m2");

        }

    }
```

- **Output:**
- `error: m2() in Child cannot override m2() in Parent`
- `    void m2() throws Exception{ System.out.println("From child m2");}`
- `         ^`
- `  overridden method does not throw Exception`

- **Rule#2 : If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error.Also there is no issue if subclass overridden method is not throwing any exception.**

```
// Java program to demonstrate overriding when

// superclass method does declare an exception



class Parent {

    void m1() throws RuntimeException

    {

        System.out.println("From parent m1()");

    }

}


class Child1 extends Parent {

    @Override

    // no issue while throwing same exception

    void m1() throws RuntimeException

    {

        System.out.println("From child1 m1()");

    }

}

class Child2 extends Parent {

    @Override
```

```
    // no issue while throwing subclass exception
    void m1() throws ArithmeticException
    {
        System.out.println("From child2 m1()");
    }
}
class Child3 extends Parent {
    @Override
    // no issue while not throwing any exception
    void m1()
    {
        System.out.println("From child3 m1()");
    }
}
class Child4 extends Parent {
    @Override
    // compile-time error
    // issue while throwing parent exception
    void m1() throws Exception
    {
        System.out.println("From child4 m1()");
    }
}
```

- **Output:**

```
error: m1() in Child4 cannot override m1() in Parent
    void m1() throws Exception
         ^
  overridden method does not throw Exception
```

21.

22. Overriding and abstract method: Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

23. Overriding and synchronized/strictfp method : The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it's possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vice-versa.

Note :

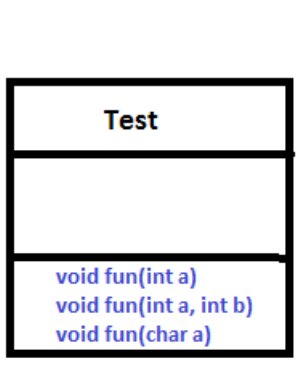
1. **In C++, we need virtual keyword to achieve overriding or Run Time Polymorphism. In Java, methods are virtual by default.**
2. **We can have multilevel method-overriding.**

```
// A Java program to demonstrate
// multi-level overriding


// Base Class
class Parent {
    void show()
    {
        System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    void show() { System.out.println("Child's show()"); }
}


// Inherited class
class GrandChild extends Child {
    // This method overrides show() of Parent
    void show()
    {
        System.out.println("GrandChild's show()");
    }
}


// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new GrandChild();
        obj1.show();
    }
```

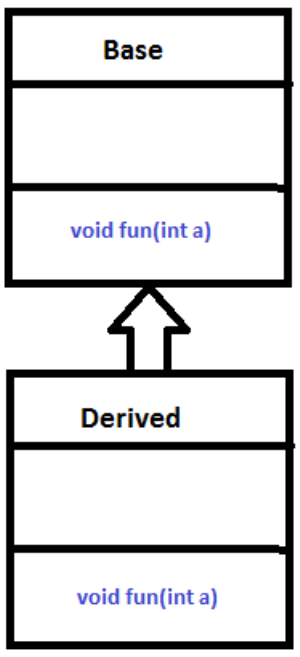
```
}
```

3. **Output:**
4. `GrandChild's show()`
5. **Overriding vs Overloading :**

    1. **Overloading is about same method have different signatures. Overriding is about same method, same signature but different classes connected through inheritance.**



Overloading



Overriding

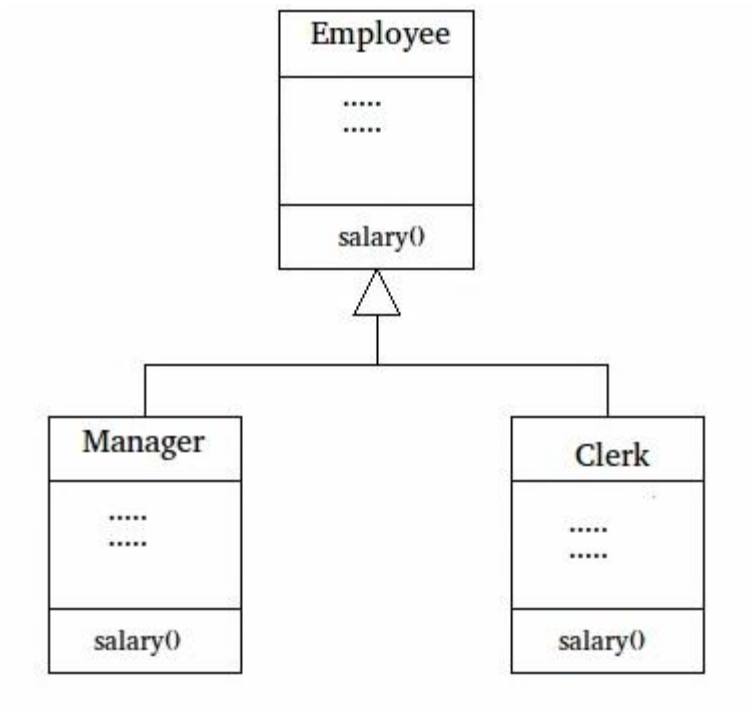
    2. **Overloading is an example of compiler-time polymorphism and overriding is an example of run time polymorphism.**

Why Method Overriding ?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.
Dynamic Method Dispatch is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability to exist code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.
Overridden methods allow us to call methods of any of the derived classes without even knowing the type of derived class object.

When to apply Method Overriding ?(with example)

Overriding and Inheritance : Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.
Let's look at a more practical example that uses method overriding. Consider an employee management software for an organization, let the code has a simple base class Employee, the class has methods like raiseSalary(), transfer(), promote(), .. etc. Different types of employees like Manager, Engineer, ..etc may have their implementations of the methods present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate methods without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its logic in its class, we don't need to worry because if raiseSalary() is present for a specific employee type, only that method would be called.




```
// A Simple Java program to demonstrate application

// of overriding in Java


// Base Class

class Employee {

    public static int base = 10000;

    int salary()

    {

        return base;

    }

}



// Inherited class

class Manager extends Employee {

    // This method overrides salary() of Parent

    int salary()

    {

        return base + 20000;

    }

}



// Inherited class

class Clerk extends Employee {

    // This method overrides salary() of Parent
```

```
    int salary()
    {
        return base + 10000;
    }
}

// Driver class
class Main {
    // This method can be used to print the salary of
    // any type of employee using base class reference
    static void printSalary(Employee e)
    {
        System.out.println(e.salary());
    }

    public static void main(String[] args)
    {
        Employee obj1 = new Manager();

        // We could also get type of employee using
        // one more overridden method.loke getType()
        System.out.print("Manager's salary : ");
        printSalary(obj1);

        Employee obj2 = new Clerk();
        System.out.print("Clerk's salary : ");
        printSalary(obj2);
    }
}
```

Output:

```
Manager's salary : 30000
Clerk's salary : 20000
```

Overloading in Java

- **Difficulty Level : Easy**
- **Last Updated : 17 Feb, 2021**

Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism.

```
// Java program to demonstrate working of method

// overloading in Java.



public class Sum {


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

public int sum(int x, int y)

{

    return (x + y);

}



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

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

{

    return (x + y + z);

}



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

public double sum(double x, double y)

{

    return (x + y);

}



// Driver code

public static void main(String args[])

{
```

```
        Sum s = new Sum();

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

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

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

    }

}
```

Output :

30

60

31.0

Question Arises:
Q. What if the exact prototype does not match with arguments.
Ans.
Priority wise, compiler take these steps:

1. **Type Conversion but to higher type(in terms of range) in same family.**
2. **Type conversion to next higher family(suppose if there is no long data type available for an int data type, then it will search for the float data type).**

Let's take an example to clear the concept:-

```
class Demo {

    public void show(int x)

    {

        System.out.println("In int" + x);

    }

    public void show(String s)

    {

        System.out.println("In String" + s);

    }

    public void show(byte b)

    {

        System.out.println("In byte" + b);

    }

}

class UseDemo {

    public static void main(String[] args)

    {
```

```
byte a = 25;

Demo obj = new Demo();

obj.show(a); // it will go to

// byte argument

obj.show("hello"); // String

obj.show(250); // Int

obj.show('A'); // Since char is

// not available, so the datatype

// higher than char in terms of

// range is int.

obj.show("A"); // String

        obj.show(7.5); // since float datatype

// is not available and so it's higher

// datatype, so at this step their

// will be an error.

}

}
```

What is the advantage?
We don't have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, ... or sum2Int, sum3Int, ... etc.

Can we overload methods on return type?
We cannot overload by return type. This behavior is same in C++. Refer this for details

```
public class Main {

    public int foo() { return 10; }



    // compiler error: foo() is already defined

    public char foo() { return 'a'; }



    public static void main(String args[])

    {

    }

}
```

However, Overloading methods on return type are possible in cases where the data type of the function being called is explicitly specified. Look at the examples below :

```
// Java program to demonstrate the working of method

// overloading in static methods

public class Main {



    public static int foo(int a) { return 10; }

    public static char foo(int a, int b) { return 'a'; }



    public static void main(String args[])

    {

        System.out.println(foo(1));

        System.out.println(foo(1, 2));

    }

}
```

Output:

```
10

a
```

```
// Java program to demonstrate working of method

// overloading in  methods

class A {

    public int foo(int a) { return 10; }

    public char foo(int a, int b) { return 'a'; }

}



public class Main {



    public static void main(String args[])

    {

        A a = new A();

        System.out.println(a.foo(1));

        System.out.println(a.foo(1, 2));
```

```
    }

}
```

Output:

```
10

a
```

Can we overload static methods?
The answer is 'Yes'. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program. Refer this for details.
Can we overload methods that differ only by static keyword?
We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. Refer this for details.
Can we overload main() in Java?
Like other static methods, we can overload main() in Java. Refer overloading main() in Java for more details.

```
// A Java program with overloaded main()
import java.io.*;


public class Test {


    // Normal main()
    public static void main(String[] args)
    {
        System.out.println("Hi Geek (from main)");
        Test.main("Geek");
    }


    // Overloaded main methods
    public static void main(String arg1)
    {
        System.out.println("Hi, " + arg1);
        Test.main("Dear Geek", "My Geek");
    }
    public static void main(String arg1, String arg2)
    {
        System.out.println("Hi, " + arg1 + ", " + arg2);
    }
}
```

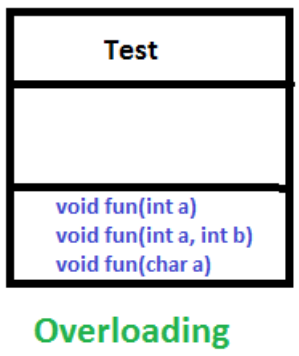
Output :

```
Hi Geek (from main)
```

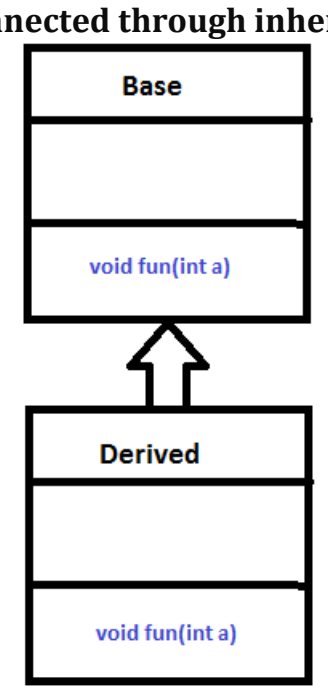
```
Hi, Geek

Hi, Dear Geek, My Geek
```

Does Java support Operator Overloading?
Unlike C++, Java doesn't allow user-defined overloaded operators. Internally Java overloads operators, for example, + is overloaded for concatenation.
What is the difference between Overloading and Overriding?

- **Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.**



Overloading



Overriding

- **Overloading is an example of compiler time polymorphism and overriding is an example of run time polymorphism.**