Object Oriented Programming (OOPs) in Java

What is OOPs 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.
- Provides means of structuring programs so that properties and behaviors are bundled into individual objects.
- **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.

Advantages of object oriented programming:

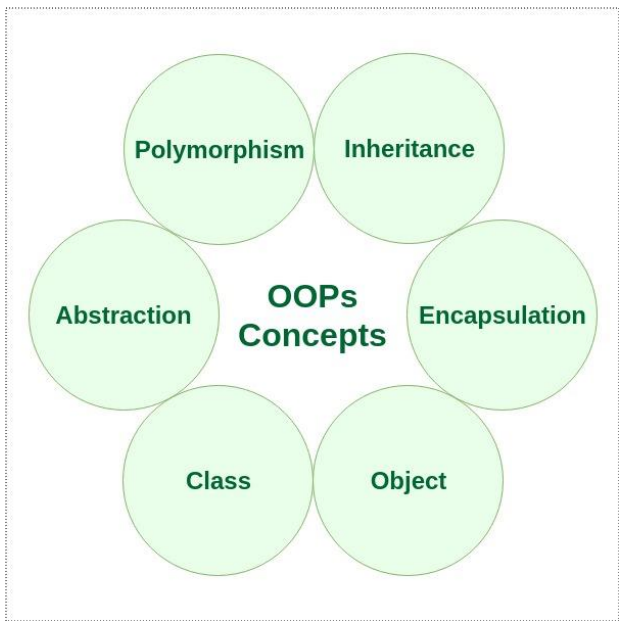
1. **Improved software-development productivity:**
    - Provides improved software-development productivity over traditional procedure-based programming techniques due to below factors.
        a. **Modularity:** Provides separation of duties in object-based program development.
        b. **Extensibility:** Objects can be extended to include new attributes and behaviors.
        c. **Reusability:** Objects can also be reused within an across applications.
2. **Improved software maintainability:**
    - For the reasons mentioned above, it is easier to maintain.
    - Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes.
3. **Faster development:**
    - Reuse enables faster development.
    - Come with rich libraries of objects, and code developed during projects is also reusable in future projects.
4. **Lower cost of development:**
    - Typically, more effort is put into the object-oriented analysis and design, which lowers the overall cost of development.
5. **Higher-quality software:**
    - Faster development of software and lower cost of development allows more time and resources to be used in the verification of the software.
    - Although quality is dependent upon the experience of the teams, it tends to result in higher-quality software.

Disadvantages of object oriented programming:

- **Steep learning curve:**
    - The thought process involved may not be natural for some people, and it can take time to get used to it.
    - It is complex to create programs based on interaction of objects.
    - Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially
- **Larger program size:**
    - Typically involve more lines of code than procedural programs.
- **Slower programs:**
    - Typically slower than procedural programs, as they typically require more instructions to be executed.
- **Not suitable for all types of problems:**
    - There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style.
    - Applying object-oriented programming in those situations will not result in efficient programs.

Major principles of object-oriented programming:

- **Classes, Objects (Instances), Methods**
- **Inheritance**
- **Polymorphism**
- **Encapsulation (Data Hiding)**
- **Abstraction (Detail Hiding)**



Classes, Objects (Instances), Methods

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.

Components of Class

1. **Modifiers:**
   - A class can be public or has default access.
   - It ***can't be declared as private or protected.***
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 { }.

Constructors:

- They 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.

Note: There are various types of classes that are used in real time applications such as `nested classes`, `anonymous classes`, `lambda expressions`.

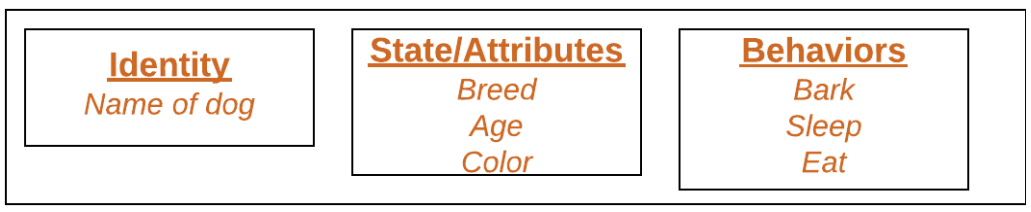
Object

- It is a basic unit of Object Oriented Programming and represents the real life entities.
- A typical Java program creates many objects, which interacts by invoking methods.

Components of Object:

1. **State:**
   - Represented by attributes of an object.
   - Also reflects the properties of an object.
2. **Behavior:**
   - Represented by methods of an object.
   - 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:

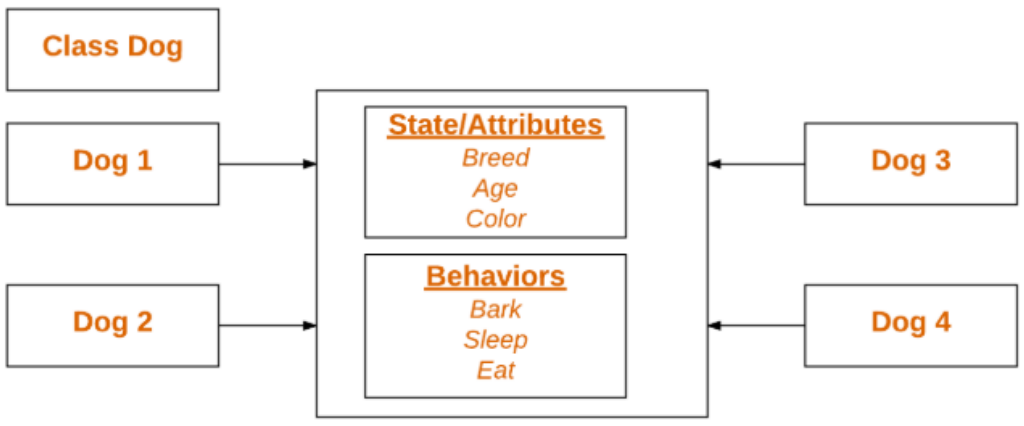


Notes:

- 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 / Instantiating 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.



Important Points:

- When we declare variables like `type name;` , it notifies the compiler that we will use this 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.
- Simply declaring a reference variable does not create an object.
- If we declare reference variable(tuffy) like below, its value will be ***undetermined(null)*** until an object is actually created and assigned to it.

```
Dog tuffy;
```

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.

Example:

```
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){
        // Initializing a Dog object
        Dog tuffy = new Dog("tuffy","papillon", 5, "white");
        System.out.println(tuffy.toString());
    }
}
```

Output:

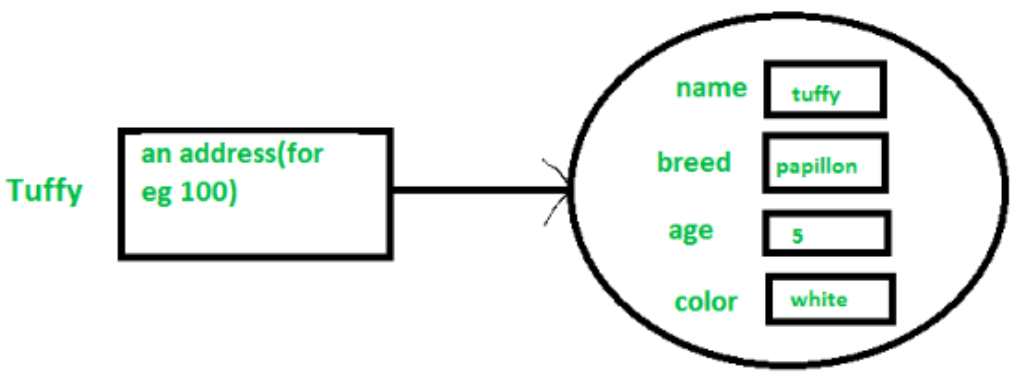
```
Hi my name is tuffy.
My breed,age and color are papillon,5,white
```

About above class

- 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");
```

- Result of above statement is:



Important Points:

- 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).

Methods

- 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.
- Java methods are strictly pass by value.

Components of Method

1. **Access Modifier:**
    - Defines access type of the method i.e. from where it can be accessed in our application.
    - In Java, there 4 type of the access specifiers.
        - **public:** accessible in all class in the 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.
2. **The return type:**

    - The data type of the value returned by the method or void if does not return a value.
3. **Method Name:**
    - The rules for field names apply to method names as well, but the convention is a little different.
4. **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, we must use empty parentheses ().
5. **Exception list:**
    - The exceptions we expect by the method can throw, we can specify these exception(s).
6. **Method body:**
    - The code we need to be executed to perform our intended operations enclosed between braces.

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.

Inheritance

What is Inheritance ?

- It is mechanism by which one class is allowed to inherit the features(fields and methods) of another class.
- The keyword used for inheritance is **extends**.

Important terminologies:

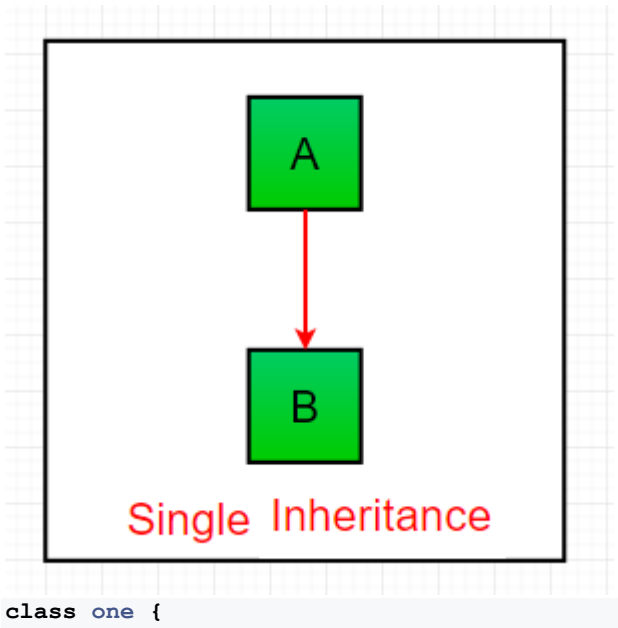
- **Super Class:**
    - The class whose features are inherited is known as superclass(or a base class or a parent class).
- **Sub Class:**
    - The class that inherits the other class is known as subclass(or a derived class, extended class, or child class).
    - The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:**
    - Inheritance supports the concept of “reusability”.
    - That means 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.

```
class DerivedClass extends BaseClass {
   //methods and fields
}
```

Types of Inheritance

1. Single Inheritance :

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




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

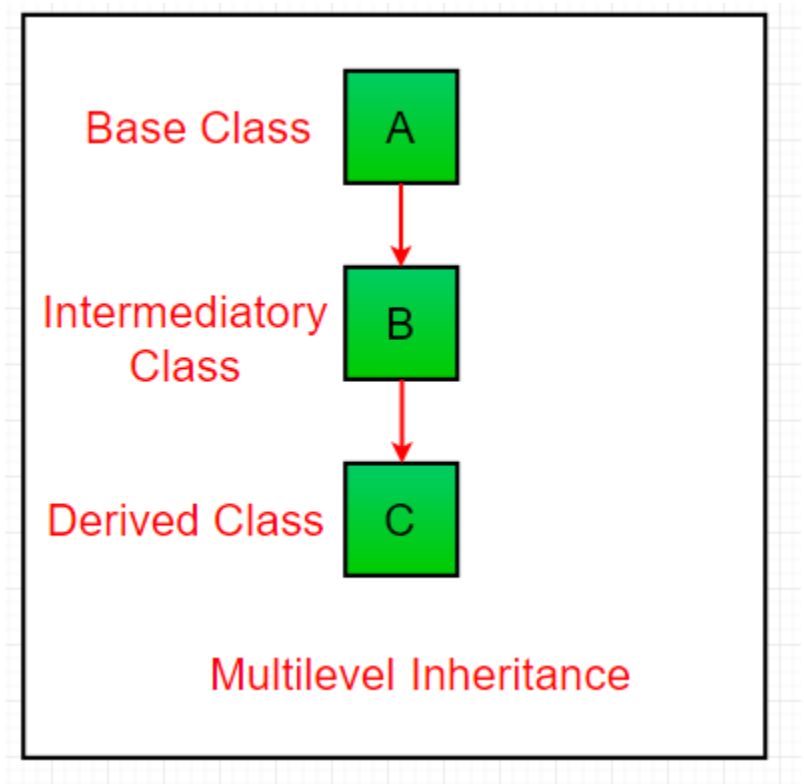
```
        }
}

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

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

Output:

```
for
Geeks
```

2. Multilevel Inheritance

- In Multilevel Inheritance, a derived class will be inheriting a base class and the derived class also act as the base class to other class.
- In below image, the 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.




```
// Java program to illustrate the concept of Multilevel inheritance
class one {
        public void print_geek() {
                System.out.println("Geeks 1");
        }

  public void print_hello() {
                System.out.println("Hello from GrandParent");
        }
}

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

class three extends two {
        public void print_geek() {
                System.out.println("Geeks 3");
        }
}

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

Output:

```
Geeks 3
for
Hello from GrandParent
```

Method Resolution:

- Java starts finding the method from the current class and goes onto super classes one by one going level above.

- As all class is subclass of java **Object Class** it seraches till there and if not found gives ***cannot find symbol error***.

Accessing Grandparent's member in Java using super

- In Java, a class cannot directly access the grandparent's members.

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

class Parent extends Grandparent {
    public void Print() {
        System.out.println("Parent");
    }
}

class Child extends Parent {
    public void Print() {
        super.super.Print();  // Trying to access Grandparent's Print()
        System.out.println("Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output:

```
Compiler Error
```

Important Points:

- There is error in line `super.super.print();`.
- In Java, a class cannot directly access the grandparent's members. It is allowed in C++ though.
- In C++, we can use scope resolution operator (::) to access any ancestor's member in inheritance hierarchy.
- ***In Java, we can access grandparent's members only through the parent class.***

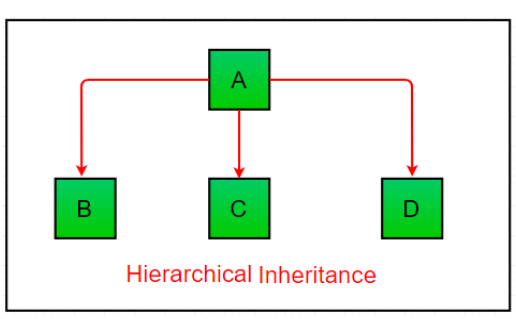
```
class Grandparent {
    public void Print() {
        System.out.println("Grandparent");
    }
}

class Parent extends Grandparent {
    public void Print() {
        super.Print();
        System.out.println("Parent");
    }
}

class Child extends Parent {
    public void Print() {
        super.Print();
        System.out.println("Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output:

```
Grandparent
Parent
Child
```

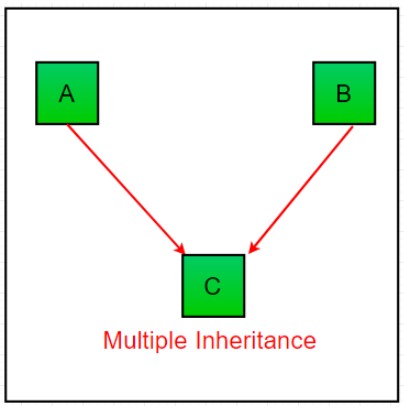
3. Hierarchical Inheritance :

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



4. Multiple Inheritance (Through Interfaces) :

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



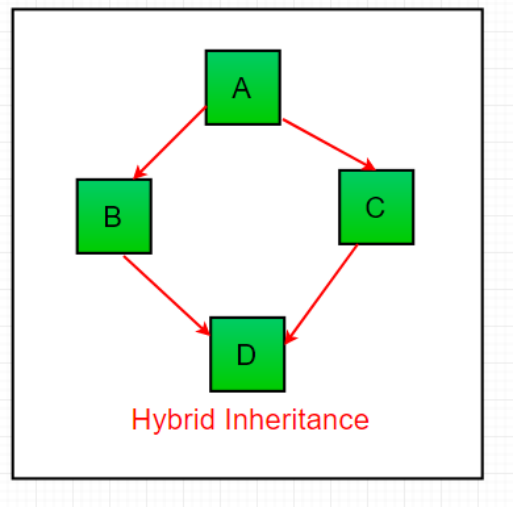

```
interface one {
        public void print_geek();
}

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");
        }
}

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

Output:

```
Geeks
for
```

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 inheritance with classes, the ***hybrid inheritance is also not possible with classes***.
- In java, we can ***achieve hybrid inheritance only through interfaces***.



Important Points 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 ***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 inheritance*** with classes.
    - Although with interfaces, multiple inheritance is 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.

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 by ***overriding*** it.
- 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***.

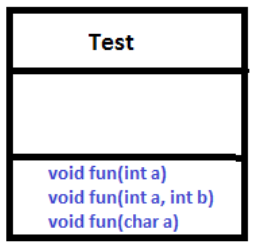
Polymorphism

What is Polymorphism ?

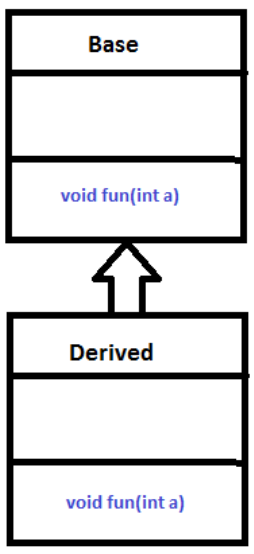
- Polymorphism allows us to perform a single action in different ways.
- Ability of OOPs programming languages to differentiate between entities with the same name efficiently.
- This is done by Java with the help of the signature and declaration of these entities.

Types Polymorphism

- **Compile time Polymorphism**
    - **Method Overloading**
    - **Operator Overloading**
- **Runtime Polymorphism**
    - **Method Overriding**



Overloading



Overriding

Compile Time Polymorphism

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

```
// Overloaded sum with different number of parameters and differnt argument types
public class Sum {
  public int sum(int x, int y) {
    return (x + y);
  }

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

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

  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
```

Operator Overloading

- **In java, only "+" operator can be overloaded:**
  - To add integers
  - To concatenate strings

```
class OperatorOVERDDN {
    void operator(String str1, String str2) {
        String s = str1 + str2;
        System.out.println("Concatinated String - " + s);
    }

    void operator(int a, int b) {
        int c = a + b;
        System.out.println("Sum = " + c);
    }
}

class Main {
    public static void main(String[] args) {
        OperatorOVERDDN obj = new OperatorOVERDDN();
        obj.operator(2, 3);
        obj.operator("joe", "now");
    }
}
```

Output:

```
Sum = 5
Concatinated String - joenow
```

Runtime Polymorphism

- 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.

Method Overriding

- Allows a subclass to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- That parent class function is said to be **overridden**.
- A superclass reference variable can refer to a subclass object k/a as **upcasting**.
- Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs k/a **dynamic dispatching**.

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

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

```
}

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 obj;
        obj = new Parent();
        obj.show();

        // If a Parent type reference refers to a Child object, then Child's show() is called.
        // This is RUN TIME POLYMORPHISM.
        // Also known as Dynamic Dispatch (here child's show() method, above parent's show() method)
        obj = new Child();
        obj.show();
    }
}
```

Output:

```
Parent's show()
Child's show()
```

Note: In Java, we can override methods only, not the variables(data members), **so runtime polymorphism cannot be achieved by data members**.

```
class A {
    int x = 10;
}

class B extends A {
    int x = 20;
}

public class Test {
    public static void main(String args[]) {
        A a = new B(); // object of type B

        // Data member of class A will be accessed
        System.out.println(a.x);
    }
}
```

Output:

```
10
```

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.***

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.
- ***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.

2. 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.

```
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 :- protected to public
    @Override
    public void m2() {
        System.out.println("From child m2()");
    }
}

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

Output:

```
From parent m2()
From child m2()
```

3. Final methods can not be overridden :

- If we don't want a method to be overridden, we declare it as **final**.

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

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

4. Static methods can not be overridden(Method Overriding vs Method Hiding) :

- When we defines a static method with same signature as a static method in base class, it is known as **Method Hiding**.
- Below table summarizes what happens when we 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

5. The overriding method must have same return type (or subtype) :

- It is possible to have different return type but it should be sub-type of parent's return type.
- This phenomena is known as **covariant return type**.

6. Invoking overridden method from sub-class :

- We can call parent class method in overriding method using **super keyword**.

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

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

class Main {
        public static void main(String[] args) {
                Parent obj = new Child();
                obj.show();
        }
}
```

7. 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.

8. 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 throws an exception, subclass overriding method can only throws the **unchecked exception**, throwing checked exception will lead to compile-time error.

```
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 throwing checked exception
    void m2() throws Exception {
        System.out.println("From child m2");
    }
}
```

    - **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.

```
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()");
    }
}
```

9. 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.

10. Overriding and synchronized/strictfp method :

- The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding.
- That means it’s possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vice-versa.

Encapsulation (Data Hiding)

What is Data 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.
- In other words, 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.
- In encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

How Encapsulation is achieved ?

- Encapsulation can be achieved by Declaring all the variables in the class as private.
- And then writing public methods in the class to set and get the values of variables.

```
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;
    }
}

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 that how the class is storing values in the variables. He only knows 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 as read-only or write-only depending on our requirement. If we wish to make the variables as 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 easy to change with new requirements.

- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Abstraction (Detail Hiding)

What is Data Abstraction ?

- It is the property by virtue of which only the essential details are displayed to the user.
- The non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.
- It may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.
- ***Example:*** 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.

How Abstraction is achieved ?

- In java, abstraction is achieved by **interfaces** and **abstract classes**.
- We can achieve 100% abstraction using interfaces.

Advantages of Abstraction:

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

Encapsulation vs Data Abstraction

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

Abstract Classes

What are abstract classes ?

- A class that ***contains one or more abstract methods*** and must be declared with **abstract keyword**.
- It is used to ***achieve abstraction to a certain extent*** the full abstraction is provided by **interfaces**.
- An abstract class may or may not have all abstract methods, some of them can be concrete methods
- A method defined abstract must always be redefined/overrrided in subclass or either make subclass itself abstract.
- There can be no object of an abstract class *i.e.* an abstract class can not be directly instantiated with ***new operator***.
- An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

What are abstract methods ?

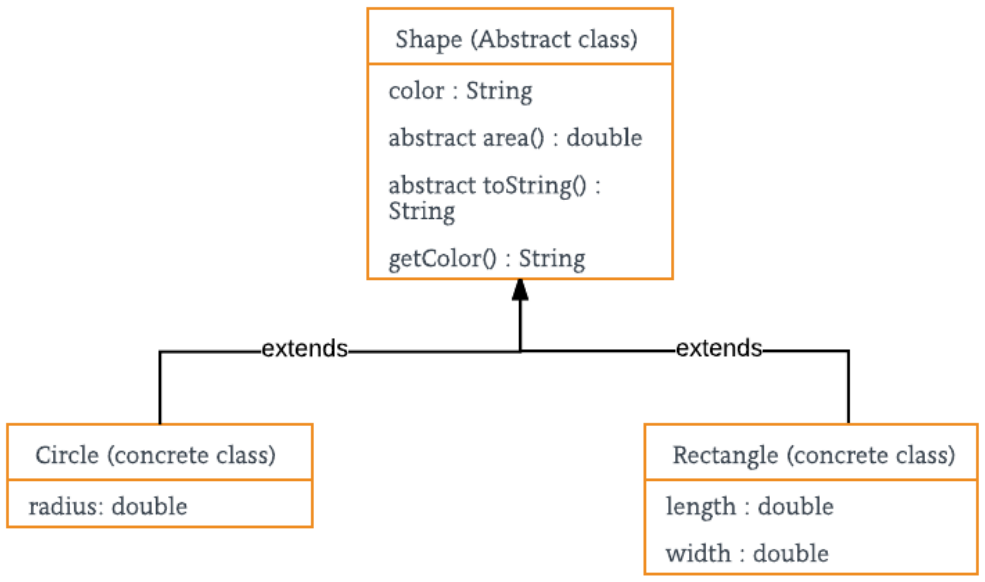
1. An abstract method is a method that is declared without an implementation.
2. It contains only signature of the mehod.

When to use abstract classes and abstract methods ?

- 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.

Situation Example:

- 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 we want to calculate the area of a shape.
- The type hierarchy embodies both the similarities and differences between the shapes.




```java
abstract class Shape  {
    String color;

    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have 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 Redand area is : 15.205308443374602
Rectangle color is Yellowand area is : 8.0
```

Properties of Abstract Classes

Property-1: An instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base {
    abstract void fun();
}
class Derived extends Base {
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
        // Below line will cause compiler error as thetries to create an instance of abstract class.
        // Base b = new Base();

        // But we can have references of Base type.
        Base b = new Derived();
        b.fun();
    }
}
```

Output:

```
Derived fun() called
```

Property-2: Constructor of abstract class is called when an instance of a inherited class is created.

```
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
       Derived d = new Derived();
    }
}
```

Output:

```
Base Constructor Called
Derived Constructor Called
```

Property-3: We can have an abstract class without any abstract method, this allows to create classes that cannot be instantiated, but can only be inherited.

```
abstract class Base {
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base { }

class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.fun();
    }
}
```

Output:

```
Base fun() called
```

Property-4: Abstract classes can also have final methods (methods that cannot be overridden).

```
abstract class Base {
    final void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base {}

class Main {
    public static void main(String args[]) {
       Base b = new Derived();
       b.fun();
    }
}
```

Output:

```
Base fun() called
```

Interfaces

What is an Interface?

- Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
- Interfaces specify what a class must do and not how.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

Basic Syntax:

```
interface <interface_name> {
    // declare constant fields
    // declare methods that are abstract by default.
}
```

Why do we use interface ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve **multiple inheritance** .

Important Points about Interfaces

- Interface can't be instantiated 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 extends another interface or interfaces (more than one interface) .
- A class that implements interface must implements all the methods in interface.
- ***All the methods are public and abstract, and all the fields are public, static, and final.***
- It is used to achieve **multiple inheritance** and **loose coupling**.

Why use interfaces when we have abstract classes?

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

Example:

```
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");
    }

    public static void main (String[] args) {
        TestClass t = new TestClass();
        t.display();
        System.out.println(a);
    }
}
```

Output:

```
Geek
10
```

A real World Example:

```
interface Vehicle {
    // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

class Bicycle implements Vehicle{
    int speed;
    int gear;
     // to change gear
    @Override
    public void changeGear(int newGear){
        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){
        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed + " gear: " + gear);
    }
}

class Bike implements Vehicle {
    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){
        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){
        speed = speed + increment;
```

```
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed + " gear: " + gear);
    }

}
class GFG {
    public static void main (String[] args) {
        // creating an 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
```

New features added in interfaces in Java 8

1. Default implementation for interface methods

Prior to Java 8, interface could not define implementation. We can now add default implementation for interface methods.

This default implementation has special use and does not affect the intention behind interfaces.

Reason:

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

```
interface In1 {
    final int a = 10;
    default void display() {
        System.out.println("hello");
    }
}

class TestClass implements In1 {
    public static void main (String[] args) {
        TestClass t = new TestClass();
        t.display();
    }
}
```

Output:

```
hello
```

2. Defining static methods in interface

We can now define static methods in interfaces which can be called independently without an object.

Note:- these methods are not inherited.

```
interface In1 {
    final int a = 10;
    default void display() {
        System.out.println("hello");
    }
}

class TestClass implements In1 {
    public static void main (String[] args) {
        In1.display();
    }
}
```

Output:

```
hello
```

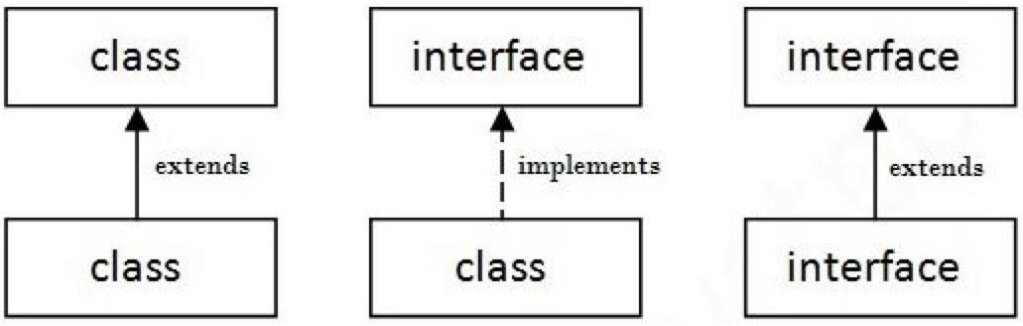
New features added in interfaces in Java 9

From Java 9 onwards, interfaces can contain following also

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

Implements vs. Extends

- Both keywords are used when creating your own new class in the Java language.
- **Implements** is for when we are implementing an interface.
- **Extends** is for inheriting from a base class for extending something that already exists by adding more functionality to it or overriding something already existing.



- To solve problem of multiple-inheritance in Java we can only have one super class, but we can implement multiple interfaces.