OOP Concepts in Java

OOP stands for Object-Oriented Programming.
It's a programming paradigm based on the concept of "objects", which can contain data and methods.

The core OOPs concepts

1. Object
2. Class
3. Abstraction
4. Encapsulation
5. Inheritance
6. Polymorphism

1. Object

The Object is the real-time entity having some state and behavior. In Java, Object is an instance of the class having the instance variables like the **state** of the object and the **methods** as the behavior of the object. The object of a class can be created by using the `new` keyword in Java Programming language.

A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

I found various Object Definitions:

1. An object is a real-world entity.

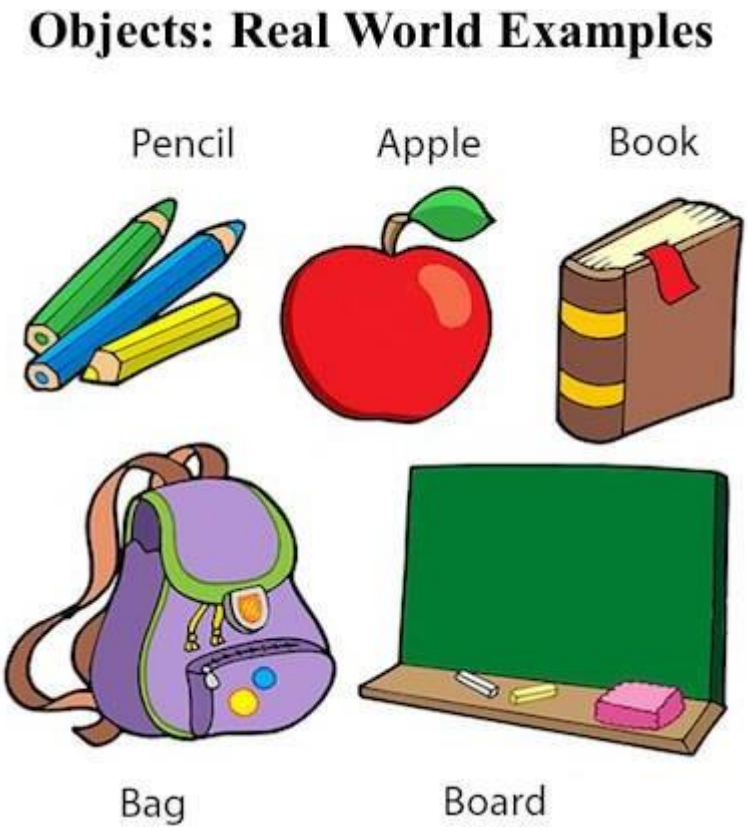
2. An object is a runtime entity.

3. The object is an entity which has state and behavior.

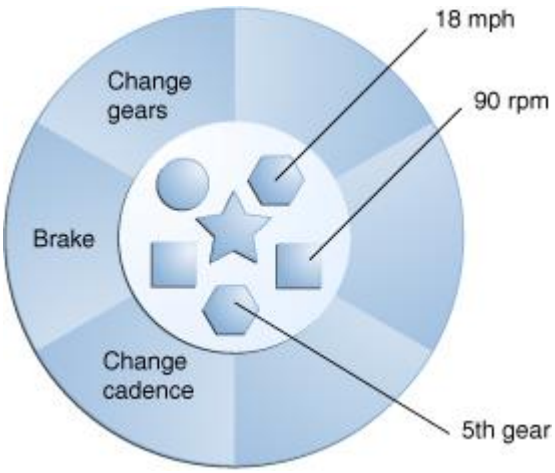
4. The object is an instance of a class.

Real-world examples

- Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). **Chair, Bike, Marker, Pen, Table, Car, Book, Apple, Bag** etc. It can be physical or logical (tangible and intangible).



- **Bicycles** also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes).



How to Declare, Create and Initialize an Object in Java

A class is a blueprint for Object, you can create an object from a class. Let's take Student class and try to create Java object for it.

Let's create a simple *Student* class which has *name* and *college* fields. Let's write a program to create declare, create and initialize a *Student* object in Java.

```
package net.javaguides.corejava.oops;

public class Student {
    private String name;
    private String college;

    public Student(String name, String college) {
        super();
        this.name = name;
        this.college = college;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    public String getCollege() {
        return college;
    }

    public void setCollege(String college) {
        this.college = college;
    }

    public static void main(String[] args) {

        Student student = new Student("Ramesh", "BVB");
        Student student2 = new Student("Prakash", "GEC");
        Student student3 = new Student("Pramod", "IIT");
    }
}
```

From the above program, the *Student* objects are:

```
Student student = new Student("Ramesh", "BVB");
Student student2 = new Student("Prakash", "GEC");
Student student3 = new Student("Pramod", "IIT");
```

Each of these statements has three parts (discussed in detail below):

Declaration: The code *Student student;* declarations that associate a variable name with an object type.

Instantiation: The *new* keyword is a Java operator that creates the object.

Initialization: The *new* operator is followed by a call to a constructor, which initializes the new object.

Declaring a Variable to Refer to an Object

General syntax:

```
type name;
```

This notifies the compiler that you will use a *name* to refer to data whose type is a *type*. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

From the above program, we can declare variables to refer to an object as:

```
Student student;
Student student2;
Student student3;
```

Instantiating a Class

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 object constructor.

For example:

```
Student student = new Student("Ramesh", "BVB");
Student student2 = new Student("Prakash", "GEC");
Student student3 = new Student("Pramod", "IIT");
```

Note that we have used a *new* keyword to create *Student* objects.

Initializing an Object

The *new* keyword is followed by a call to a constructor, which initializes the new object. For example:

```
Student student = new Student("Ramesh", "BVB");
Student student2 = new Student("Prakash", "GEC");
Student student3 = new Student("Pramod", "IIT");
```

From above code will call below constructor in **Student** class.

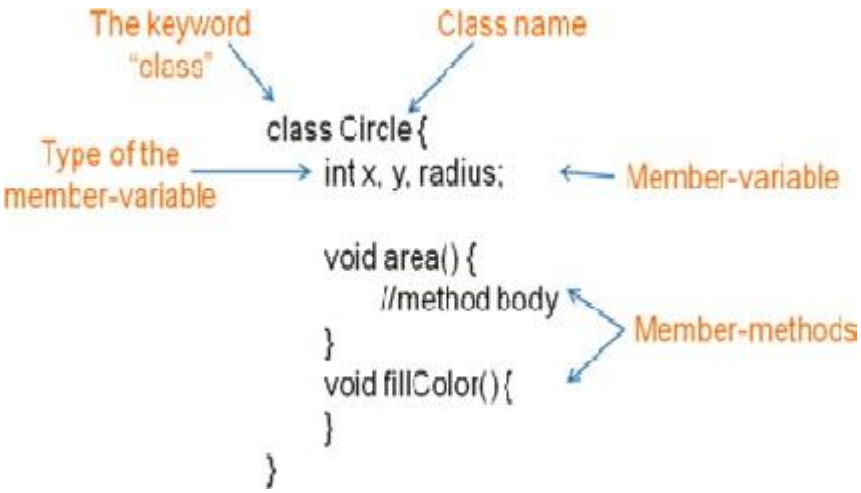
```
public class Student {
    private String name;
    private String college;

    public Student(String name, String college) {
         super();
         this.name = name;
         this.college = college;
    }
}
```

2. Class

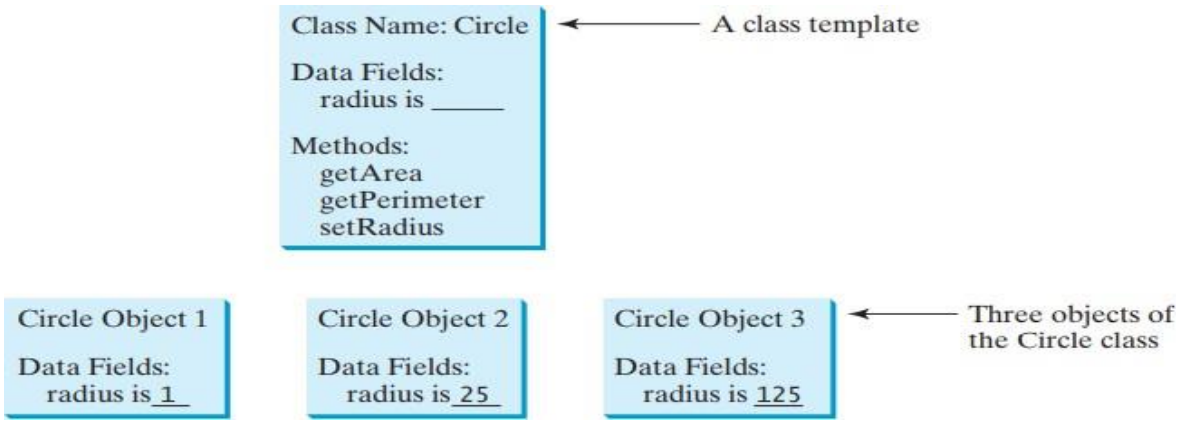
A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. In short, a class is the **specification or template of an object**.

A real-world example is **Circle**. Let's look at an example of a class and analyze its various parts in a below diagram. This example declares the class **Circle**, which has the member-variables x, y, and radius of type Integer and the two member-methods, `area()`and `fillColor()`.



A class is a template for creating objects

Below diagram shows a **Circle** class which is a template to create three objects:



Implementation with Example - Creating Student Class

Let's demonstrate how to create *Class* in Java with an example. Here is a **Student** class:

```
package net.javaguides.corejava.oops;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class Student {
    private String name = "Ramesh";
    private String college = "ABC";

    public Student() {
        super();
    }

    public Student(String name, String college) {
        super();
        this.name = name;
        this.college = college;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCollege() {
        return college;
    }

    public void setCollege(String college) {
        this.college = college;
    }
}
```

3. Abstraction

Abstraction means hiding lower-level details and exposing only the essential and relevant details to the users.

Real-world examples

1. The first example, let's consider a **Car,** which abstracts the internal details and exposes to the driver only those details that are relevant to the interaction of the driver with the **Car**.

2. The second example, consider an **ATM** Machine; All are performing operations on the ATM machine like cash withdrawal, money transfer, retrieve mini-statement...etc. but we can't know internal details about ATM.

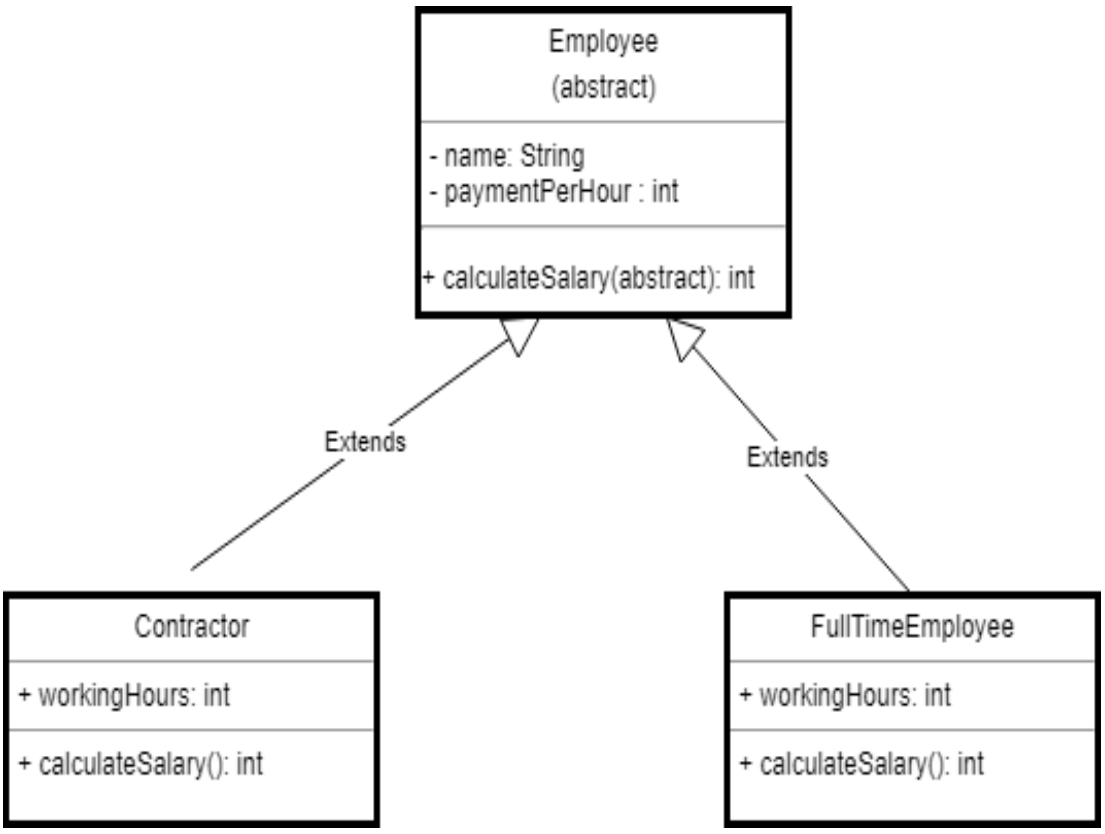
Real Life Example of Abstraction

Implementation with Example

Example - Employee, Contractor, and FullTimeEmployee Example

In this example, we create an abstract **Employee** class and which contains abstract `calculateSalary()` method. Let the subclasses extend **Employee** class and implement a `calculateSalary()` method.

Let's create **Contractor** and **FullTimeEmployee** classes as we know that the salary structure for a contractor and full-time employees are different so let these classes to override and implement a `calculateSalary()` method.



Let's write source code by looking into an above class diagram.

Step 1: Let's first create the superclass **Employee**. Note the usage of abstract keyword in this class definition. This marks the class to be abstract, which means it can not be instantiated directly. We define a method called *calculateSalary()* as an abstract method. This way you leave the implementation of this method to the inheritors of the **Employee** class.

```
public abstract class Employee {

    private String name;
    private int paymentPerHour;

    public Employee(String name, int paymentPerHour) {
        this.name = name;
        this.paymentPerHour = paymentPerHour;
    }

    public abstract int calculateSalary();
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getPaymentPerHour() {
        return paymentPerHour;
    }
    public void setPaymentPerHour(int paymentPerHour) {
        this.paymentPerHour = paymentPerHour;
    }
}
```

Step 2: The **Contractor** class inherits all properties from its parent abstract **Employee** class but have to provide its own implementation to *calculateSalary()* method. In this case, we multiply the value of payment per hour with given working hours.

```
public class Contractor extends Employee {

    private int workingHours;
    public Contractor(String name, int paymentPerHour, int workingHours) {
        super(name, paymentPerHour);
        this.workingHours = workingHours;
    }
    @Override
    public int calculateSalary() {
        return getPaymentPerHour() * workingHours;
    }
}
```

Step 3: The **FullTimeEmployee** also has its own implementation of *calculateSalary()* method. In this case, we just multiply by a constant value of **8** hours.

```
public class FullTimeEmployee extends Employee {
    public FullTimeEmployee(String name, int paymentPerHour) {
        super(name, paymentPerHour);
    }
    @Override
    public int calculateSalary() {
        return getPaymentPerHour() * 8;
    }
}
```

Step 4: Let's create a **AbstractionDemo** class to test implementation of *Abstraction* with below code:

```
public class AbstractionDemo {

    public static void main(String[] args) {

        Employee contractor = new Contractor("contractor", 10, 10);
        Employee fullTimeEmployee = new FullTimeEmployee("full time employee", 8);
        System.out.println(contractor.calculateSalary());
        System.out.println(fullTimeEmployee.calculateSalary());
    }
}
```

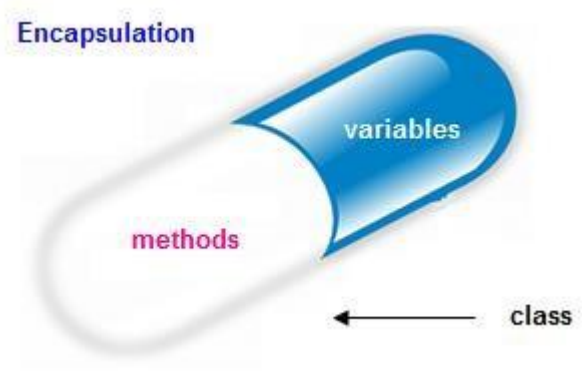
4. Encapsulation

Encapsulation is a process of wrapping of data and methods in a single unit is called encapsulation.

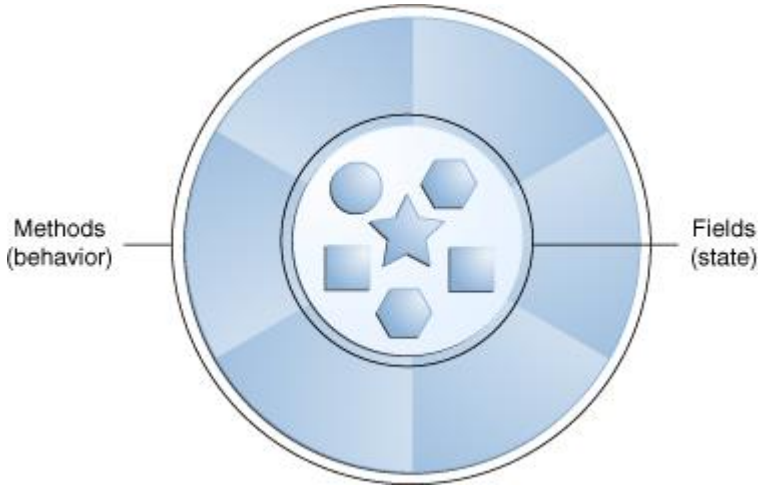
In OOP, data and methods operating on that data are combined together to form a single unit, which is referred to as a **Class**.

Real-world examples

1. **Capsule**, it is wrapped with different medicines. In a capsule, all medicine is encapsulated inside a capsule.

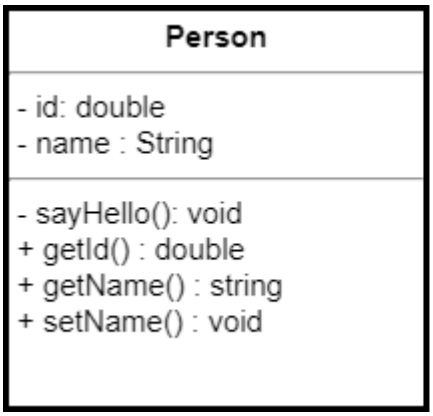


2. A Java class is an example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.



Implementation with Example

Consider below *Person* class diagram, the *id* and *name* parameters should not be accessed directly outside *Person* class - achieved by private declaration.



The **id** and **name** parameters should not be accessed directly outside Person class achieved by private declaration

- Symbol for private
+ Symbol for public

Let's create a *Person* class to demonstrate the use of encapsulation in Java.

Step 1: Create a *Person* class.

```
public class Person {

    private double id;
    private String name;

    public Person() {
        // Only Person class can access and assign
        id = Math.random();
        sayHello();
    }

    // This method is protected for giving access within Person class only
    private void sayHello() {
        System.out.println("Hello - " + getId());
    }

    public double getId() {
        return id;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Step 2: Let's test the **Encapsulation** via a *main()* method.

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

        Person p1 = new Person();
        p1.setName("Ramesh");
        /*
         * Note: As id and name are encapsulated in Person class, those cannot be
accessed
         * directly here. Also there is no way to assign id in this
         * class. Try to uncomment below code and you would find compile time
error.
         */
        // p1.id = "123";
        // p1.name = "this will give compile time error";
        // p1.sayHello(); // You can't access this method, as it is private to
Person

        System.out.println("Person 1 - " + p1.getId() + " : " + p1.getName());
    }
}
```

4. Difference between Abstraction and Encapsulation

Abstraction and *Encapsulation* in Java are two important Object-oriented programming concept and they are completely different to each other.

- **Encapsulation** is a process of binding or wrapping the data and the codes that operate on the data into a single entity. This keeps the data safe from outside interface and misuse.
- **Abstraction** is the concept of hiding irrelevant details. In other words, make the complex system simple by hiding the unnecessary detail from the user.
- **Abstraction** is implemented in Java using interface and abstract class while **Encapsulation** is implemented using private, package-private and protected access modifiers.
- **Abstraction** solves the problem in the design level. Whereas **Encapsulation** solves the problem in the implementation level.

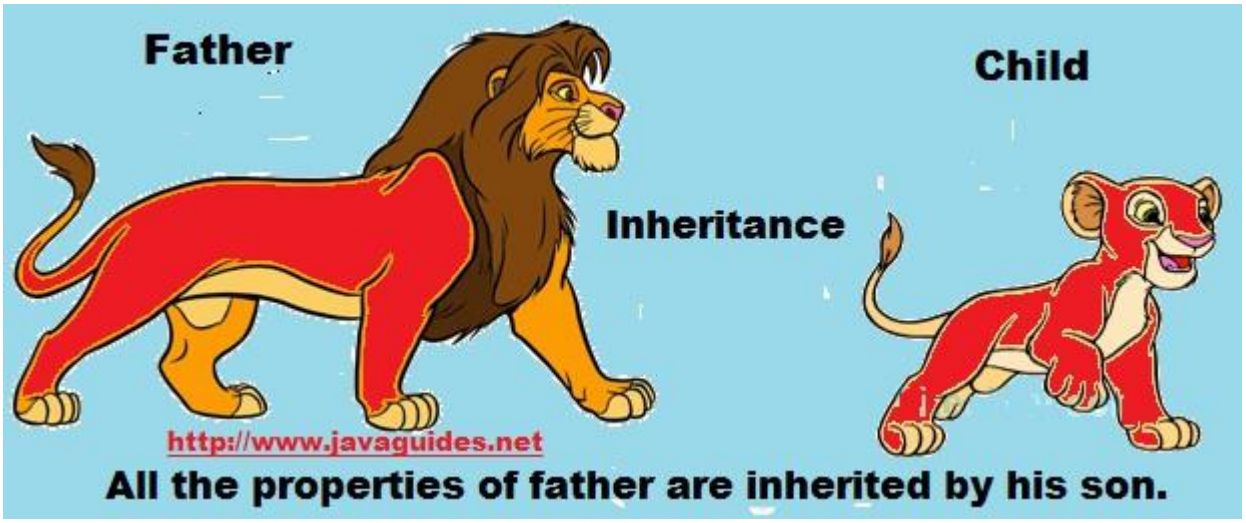
5. Inheritance

The Inheritance is a process of obtaining the data members and methods from one class to another class, plus can have its own is known as inheritance. It is one of the fundamental features of **object-oriented programming**.

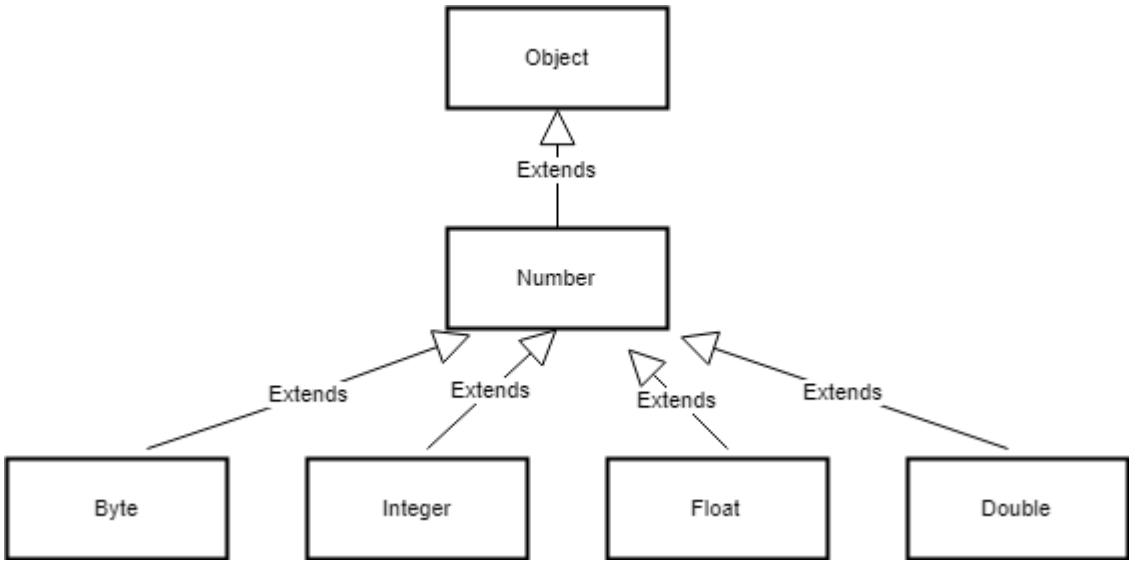
Inheritance - IS-A relationship between a superclass and its subclasses.
Super Class: The class whose features are inherited is known as a superclass (or a base class or a parent class).
Sub Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Real-world example

1. The real-life example of inheritance is child and parents, all the properties of a father are inherited by his son.



2. In the Java library, you can see extensive use of inheritance. Below figure shows a partial inheritance hierarchy from a **java.lang** library. The **Number** class abstracts various numerical (reference) types such as `Byte`, `Integer`, `Float`, `Double`, `Short`, and `BigDecimal`.



Implementation with Example

Example 1: Let's understand inheritance by example.

Dog class is inheriting behavior and properties of *Animal* class and can have its own too.

```
/**
 * Test class for inheritance behavior - Dog class is inheriting behavior
 * and properties of Animal class and can have its own too.
 *
 *
 */
public class Inheritance {

    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.setId(123); // inherited from super class
        dog.sound(); // overrided behavior of sub class
    }
}

/**
 * This is parent (also called as super or base) class Animal
 *
 */
class Animal {
    int id;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
```

```
    public void sound() {
        System.out.println("By default it is mute");
    }
}

/**
 * This is subclass (also called as derived or child or extended) Dog which is
extending Animal
 *
 */
class Dog extends Animal {

    // Own behavior
    private void bark() {
        System.out.println("Dog '" + getId() + "' is barking");
    }

    // Overriding super class behavior
    @Override
    public void sound() {
        bark();
    }
}
```

The keyword used for inheritance is *extends*.
Syntax :

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

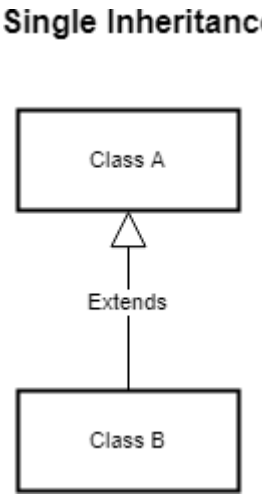
Example 2: In this example, the *Programmer* is the subclass and *Employee* is the superclass. The relationship between the two classes is the *Programmer* IS-A Employee. It means that *Programmer* is a type of *Employee*.

```
class Employee{
    float salary=40000;
}
class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}
```

Types of Inheritance in Java

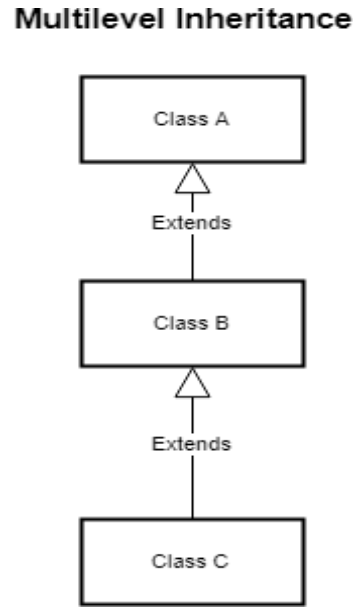
Below are the different types of inheritance which are supported by Java.

1. Single Inheritance



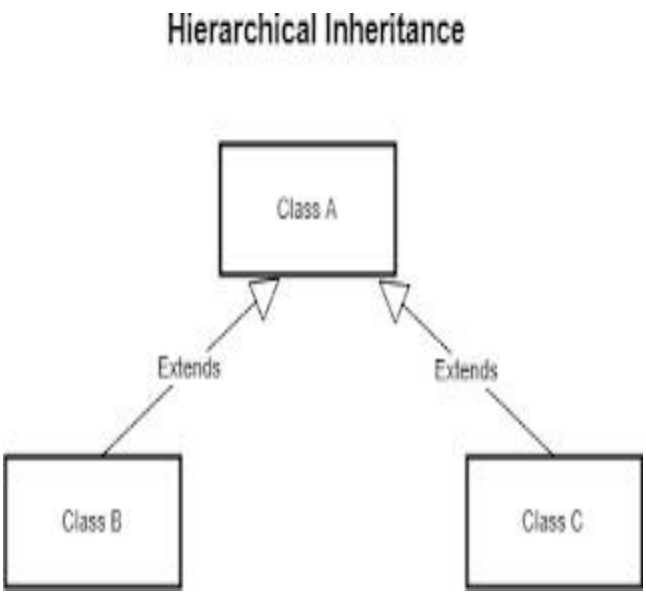
In single inheritance, subclasses inherit the features of one superclass. In the image below, class **A** serves as a base class for the derived class **B**.

2. Multilevel Inheritance



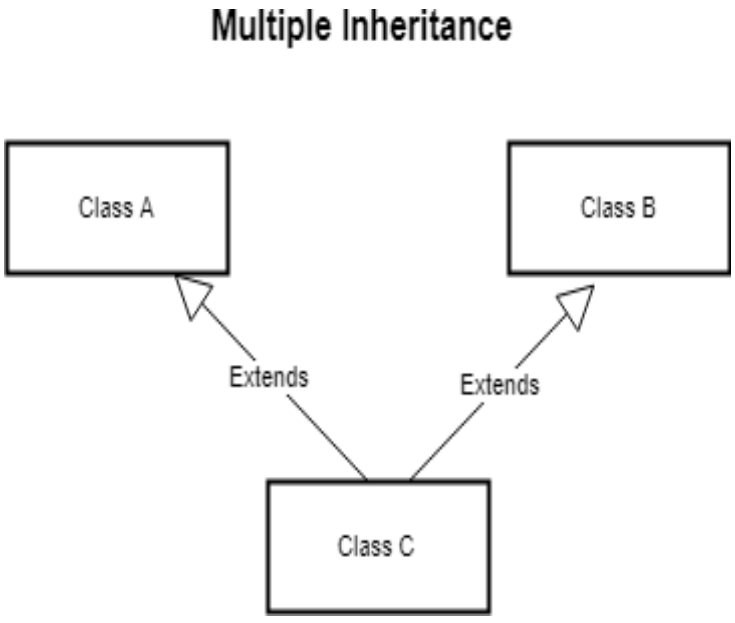
In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, class **A** serves as a base class for the derived class **B**, which in turn serves as a base class for the derived class **C**. In Java,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

3. Hierarchical Inheritance



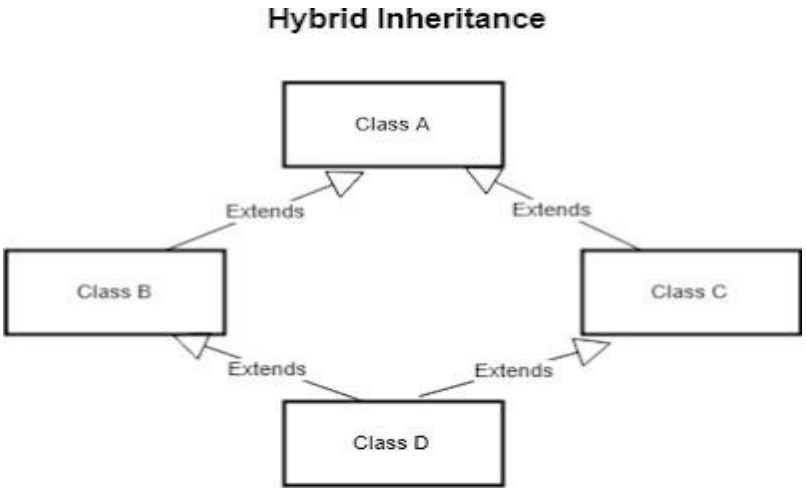
In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In below image, class **A** serves as a base class for the derived class **B** and class **C**.

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 class **A** and class **B**.

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

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where **A**, **B**, and **C** are three classes. The **C** class inherits **A** and **B** classes. If **A** and **B** classes have the same method and you call it from child class object, there will be ambiguity to call a method of **A** or **B**class.

Since compile-time errors are better than runtime errors, java renders compile-time error if you inherit 2 classes. So whether you have the same method or different, there will be compile time error now.

```
class A{
    void msg(){System.out.println("Hello");}
}
class B{
    void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

     Public Static void main(String args[]){
          C obj=new C();
          obj.msg();//Now which msg() method would be invoked?
     }
}
```

6. Polymorphism

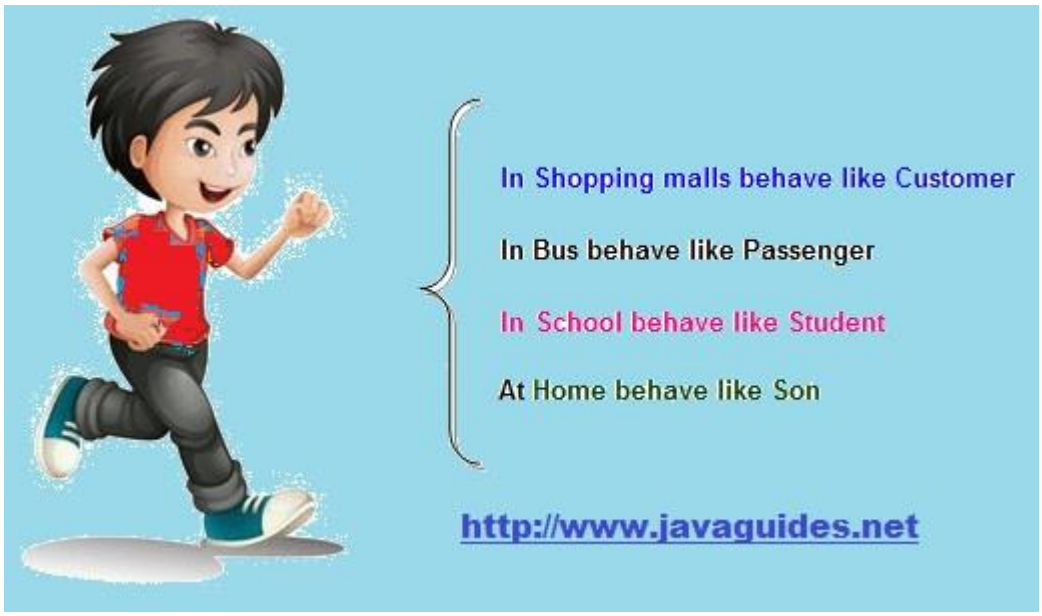
The process of representing one form in multiple forms is known as **Polymorphism**.

Different definitions of Polymorphism are:

1. `Polymorphism` let us perform a single action in different ways.
2. `Polymorphism` allows you to define one interface and have multiple implementations
3. We can create functions or reference variables which behaves differently in a different programmatic context.
4. `Polymorphism` means many forms.

A real-world example of polymorphism

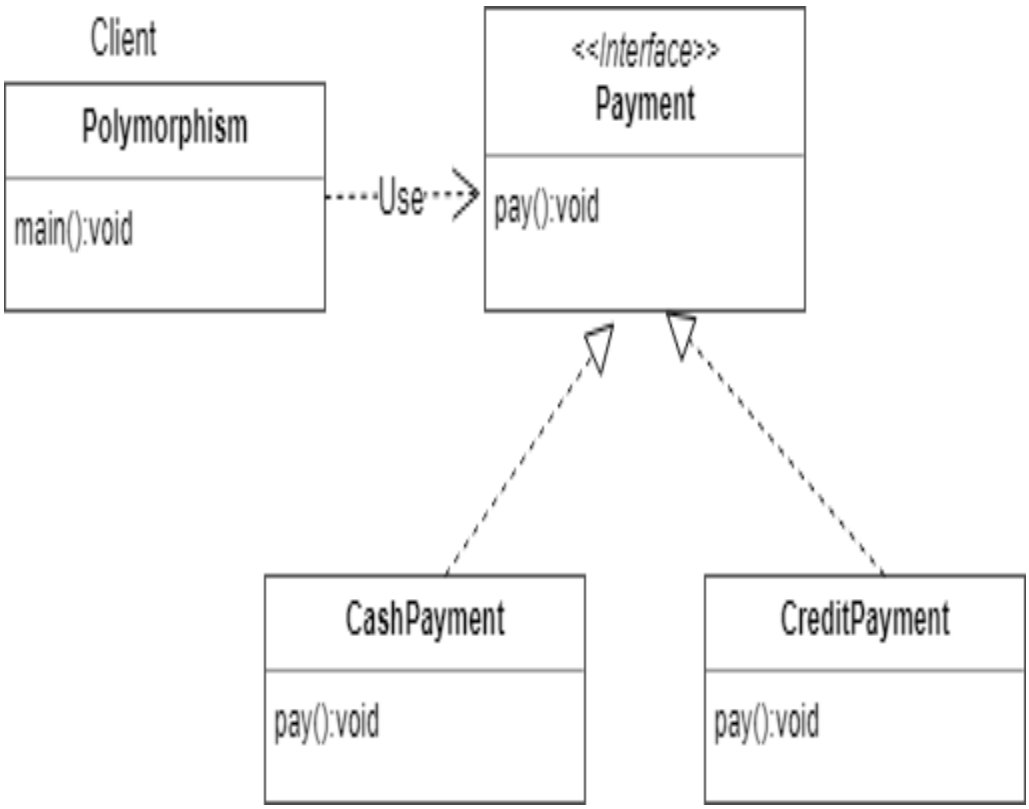
Suppose if you are in a classroom that time you behave like a **student**, when you are in the market at that time you behave like a **customer**, when you at your home at that time you behave like a **son** or **daughter**, Here one person present in different-different behaviors.



Implementation with Example

Example: Payment Processing Example

In this **Payment Processing Example**, applying runtime polymorphism and it can have many forms at runtime. Refer below source code the single payment "p" instance can be used to pay by cash and credit card, payment *p* instance takes many forms here.



Let's start coding by looking into above class diagram.

1. Create Payment interface
2. Create **CashPayment** class which implements Payment interface
3. Create **CreditPayment** class which implements Payment interface
4. Let's create a **Polymorphism** class with main() method for testing.

```java
public class Polymorphism {

    public static void main(String[] args) {
        // Here the runtime polymorphism fundamental is not applied,
        // as it is of single CashPayment form
        CashPayment c = new CashPayment();
        c.pay();

        // Now the initialization is done using runtime polymorphism and
        // it can have many forms at runtime
        // Single payment "p" instance can be used to pay by cash and credit card
        Payment p = new CashPayment();
        p.pay(); // Pay by cash

        p = new CreditPayment();
        p.pay(); // Pay by creditcard
    }

}

/**
 * This represents payment interface
 */
interface Payment {
    public void pay();
}

/**
 * Cash IS-A Payment type
 *
 * @author tirthalp
 *
 */
class CashPayment implements Payment {

    // method overriding
    @Override
    public void pay() {
        System.out.println("This is cash payment");
    }

}

/**
 * Creditcard IS-A Payment type
 */
class CreditPayment implements Payment {

    // method overriding
    @Override
    public void pay() {
        System.out.println("This is credit card payment");
    }

}
```

Types of Polymorphism in Java

1. **Compile time polymorphism** or **method overloading** or **static banding**
2. **Runtime polymorphism** or **method overriding** or **dynamic binding**

When a type of the object is determined at a compiled time(by the compiler), it is known as *`static binding`*.

When a type of the object is determined at run-time, it is known as *`dynamic binding`*.

1. Compile time Polymorphism

If the class contains two or more methods having the same name and different arguments then it is *method overloading*.

The compiler will resolve the call to a correct method depending on the actual number and/or types of the passed parameters The advantage of method overloading is to increases the readability of the program.

Method Overloading: changing no. of arguments
In this example, we have created two methods, first, add() method performs the addition of two numbers and a second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create an instance for calling methods.

```
class Adder {
    static int add(int a, int b) {
        return a + b;
    }
    static int add(int a, int b, int c) {
        return a + b + c;
    }
}
class TestOverloading1 {
    public static void main(String[] args) {
        System.out.println(Adder.add(11, 11));
        System.out.println(Adder.add(11, 11, 11));
    }
}
```

In this example, we have created two methods that differ in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder {
    static int add(int a, int b) {
        return a + b;
    }
    static double add(double a, double b) {
        return a + b;
    }
}
class TestOverloading2 {
    public static void main(String[] args) {
        System.out.println(Adder.add(11, 11));
        System.out.println(Adder.add(12.3, 12.6));
    }
}
```

2. Runtime Polymorphism

Runtime polymorphism is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable. Let's first understand the upcasting before *`Runtime Polymorphism`*.

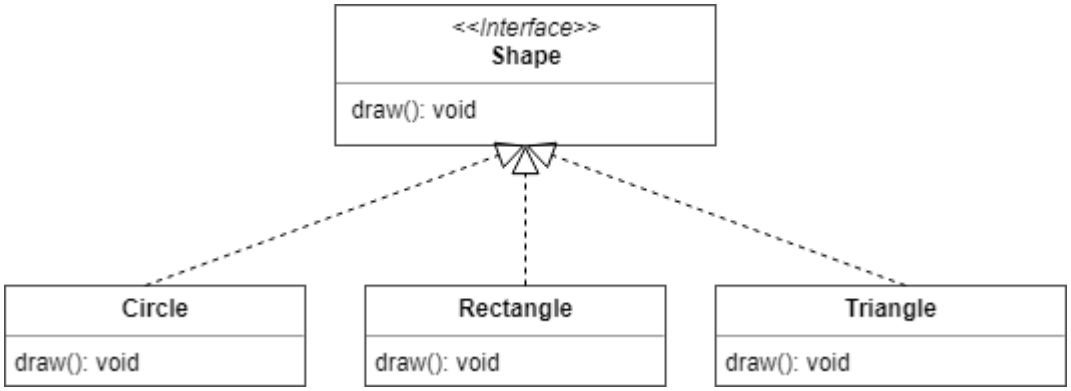
Upcasting

When reference variable of *`Parent`* class refers to the object of *`Child`* class, it is known as upcasting.

For example:

```
class A {}
class B extends A {}
class Demo {
    public static void main(String[] args) {
        A a = new B(); //upcasting
    }
}
```

Java Runtime Polymorphism Example: Shape Example




```
class Shape {
    void draw() {
        System.out.println("drawing...");
    }
}
class Rectangle extends Shape {
    void draw() {
        System.out.println("drawing rectangle...");
    }
}
class Circle extends Shape {
    void draw() {
        System.out.println("drawing circle...");
    }
}
class Triangle extends Shape {
    void draw() {
        System.out.println("drawing triangle...");
    }
}
class TestPolymorphism2 {
    public static void main(String args[]) {
        Shape s;
        s = new Rectangle();
        s.draw();
        s = new Circle();
        s.draw();
        s = new Triangle();
        s.draw();
    }
}
```

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

Java Runtime Polymorphism with Data Member

The method is overridden not applicable data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member *speedlimit*, we are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of *Parent* class always.

```
class Bike {
    int speedlimit = 90;
}
class Honda extends Bike {
    int speedlimit = 150;

    public static void main(String args[]) {
        Bike obj = new Honda();
        System.out.println(obj.speedlimit); //90
    }
}
```

Output:

```
90
```

Let's try the below scenario: Here the *BabyDog* is not overriding the *eat()* method, so *eat()* method of Dog class is invoked. Note that if we are not using @Override annotation in this example.

```
class Animal {
    void eat() {
        System.out.println("animal is eating...");
    }
}
class Dog extends Animal {
    void eat() {
        System.out.println("dog is eating...");
    }
}
class BabyDog1 extends Dog {
    public static void main(String args[]) {
        Animal a = new BabyDog1();
        a.eat();
    }
}
```

Output:

```
dog is eating...
```

Polymorphism - Method Overloading vs Method Overriding

Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
In java, method overloading can't be done by changing only the return type of method. *Return type can be same/different* in overloading, but you must change the parameter.	*Return type must be same or covariant (changing return type to subclass type)* in method overriding.