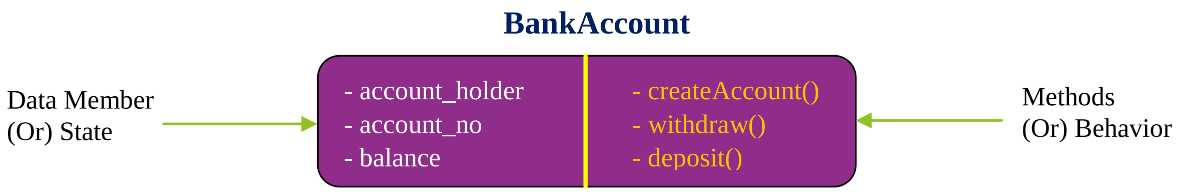
Contents

2.1 What is OOP?

- OOP is a methodology to design a program using classes and objects.
- The main aim of OOP is to bind together the data and the functions that operate on them into single unit.



Advantages of OOP

- a clear structure for the programs
- keep the Java code DRY (Don't Repeat Yourself)
- code easier to maintain, modify and debug
- reusable applications with less code and shorter development time

The four main principles of OOP are inheritance, encapsulation, abstraction, and polymorphism.

2.2 Fundamentals of OOP

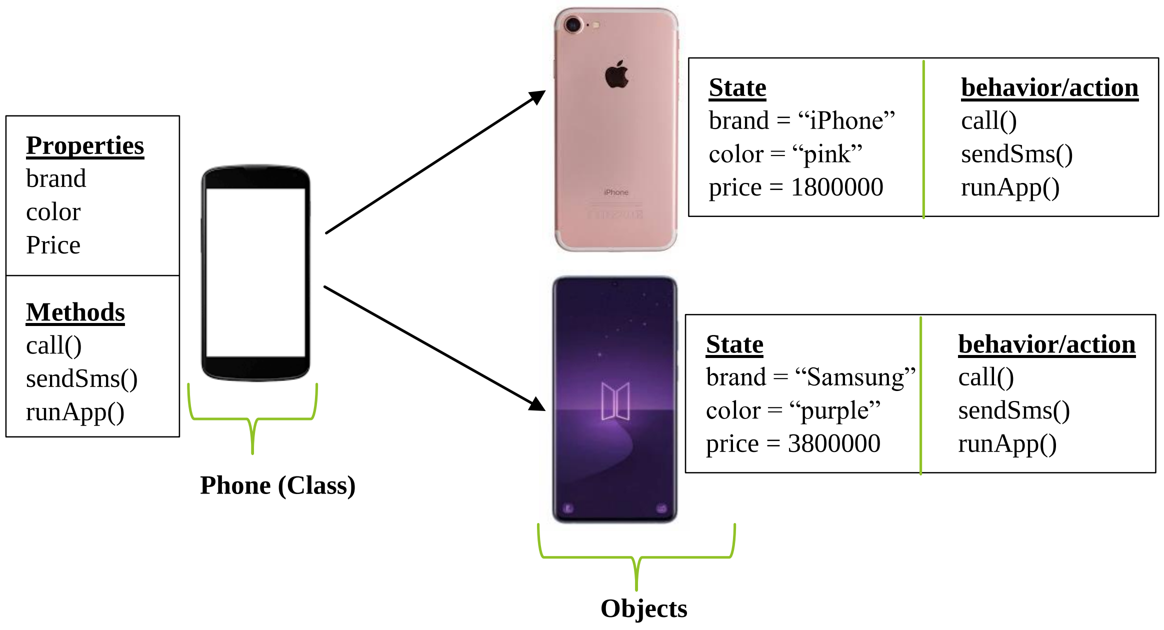
- Classes and Objects are two main fundamental concepts of OOP.

Class

- A class is the blueprint or template for its objects.
- A class would typically contain fields, methods, and a special constructor method.

Object

- An object is a living entity created from the class.
- Each object has its own state (field), behavior (method), and identity.



Declaration of Class in Java

- A class can be declared using the keyword class followed by a class name.
- It typically contains fields, constructors and methods.

General Syntax

```
<access-modifier> class <ClassName>
 {
   // Members of class
     1. Field Declarations
     2. Constructor Declarations
     3. Method Declarations
     4. Block Declarations
 }
```

1. Fields
    - Fields are data member variables of a class that stores data or value.
    - It may be an instance variable or static variable.
2. Constructor
    - A Java constructor is used to create an object.
    - A constructor can be divided into two types - default constructor and user-defined constructor.
3. Method
    - A method defines action or behavior of the class that a class's object can perform based on some data.
    - It may be an instance method or a static method.
4. Block
    - A block is mostly used to change the default values of variables.
    - It may be an instance block or static block.

Example – Sample Java Class Declaration

```
public class Phone {

        //fields
        String brand;
        String color;
        int price;
        //constructor
        public Phone(String brand,String color,int price) {
                this.brand = brand;
                this.color = color;
                this.price = price;
        }

        //methods
        public void call() {
                System.out.println("Phone calling can be made at here!");
        }
        public void sendSms() {
                System.out.println("Message can be sent at here!");
        }
        // ...
}
```

Creating Object

- Objects are created from classes and are called instances of the class.
- The "**new**" keyword is used along with the constructor of the class to create an object.

Syntax

```
ClassName objname; // declare object
objName = new Constructor(); // create object
(Or)
ClassName objname = new Constructor(); // create object
```

Example

```
Phone phone1 = new Phone("iPhone","pink",1800000);
Phone phone2 = new Phone("Samsung","purple",3800000);
Phone phone3 = new Phone("Vivo", "Black", 1000000);
```

Accessing Object's Data

- Instance variables and Instance methods can be accessed by using the dot (.) operator.

Syntax

```
objectName.variable-name; // access its properties
objectName.methodName(); // access its actions
```

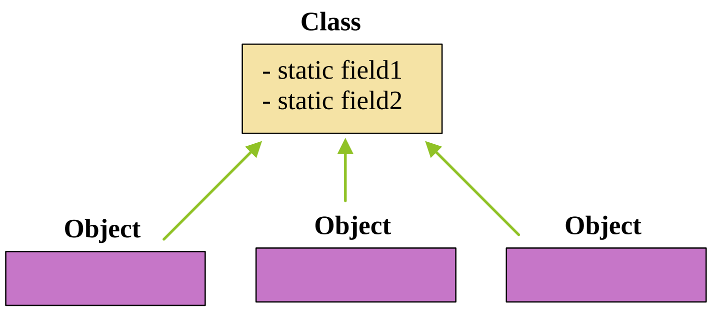
Example

```
System.out.println("--------Phone 1's Info---------");
System.out.format("Brand(%s),Color(%s),Price(%d)\n", phone1. brand, phone1.color,
phone1.price);
phone1.call();
phone1.sendSms();
System.out.println("--------Phone 2's Info---------");
System.out.format("Brand(%s),Color(%s),Price(%d)\n", phone2.brand, phone2.color,
phone2.price);
phone2.call();
phone2.sendSms();
```

Static vs. Instance Field

1. Static Field

- Static variables or fields belong to the class and also call class variables.
- They are located in the class, not in the instances of the class.
- Static variables are shared among all the instances of class.



- A static variable is initialized when the class is loaded at runtime.
- A static variable can be accessed directly by the class name and doesn't need any object(e.g. ***className.variable***)
- Static variable is defined with "**static**" keyword.
- **Syntax – static dataType variable-name;**

Example

```
class Employee{
        static int noOfemployee;
}
```

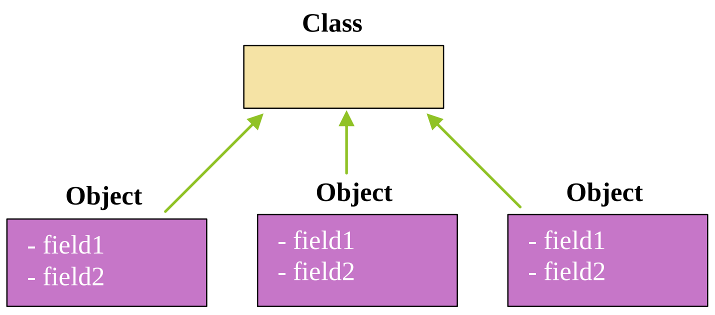
```
Employee.noOfemployee = 50; // accessing static variable

Employee emp1 = new Employee();
Employee emp2 = new Employee();

System.out.println("emp1's no of employee: " + emp1.noOfemployee);
System.out.println("emp2's no of employee: " + emp2.noOfemployee);

emp1.noOfemployee = 30; //change via object( but not recommend)

System.out.println("*** After changing class variable ***");
System.out.println("emp1's no of employee: " + emp1.noOfemployee);
System.out.println("emp2's no of employee: " + emp2.noOfemployee);
```

Output

```
emp1's no of employee: 50
emp2's no of employee: 50
*** After changing class variable ***
emp1's no of employee: 30
emp2's no of employee: 30
```

2. Instance Field

- Instance variables are encapsulated to each instance of a class.
- An instance variable is one per Object, every object has its own copy of instance variable.
- They can only be accessed or invoked through an object reference.



Example

```
class AppException extends Exception{

public AppException(String msg) {
                super(msg);
        }
}
```

```
public class Employee{
        static int noOfemployee = 3; //static variable
        static int nextId = 1; // static variable
        final int empId; //final variable
        String name; //instance variable
        int salary; // instance variable

        public Employee() throws AppException {
                if(nextId > noOfemployee)
                        throw new AppException("Sorry!Limited number had been reached");
                empId = nextId;
                nextId++;
        }
        public void initializeData(String name,int salary) {
                this.name = name;
                this.salary = salary;
        }
        public void changeData(String editName,int editSalary) {
                if(!name.equalsIgnoreCase(editName))
                        name = editName;
                if(salary != editSalary)
                        salary = editSalary;
        }
        public void showData() {
                System.out.println(this.empId + "\t" + this.name + "\t" + this.salary);
        }

}
```

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

 try {
        //create object & initialize data
        Employee emp1 = new Employee();
        emp1.initializeData("Jeon", 900000);
        Employee emp2 = new Employee();
        emp2.initializeData("Honey", 800000);
        Employee emp3 = new Employee();
        emp3.initializeData("Yuki", 1000000);

        //show information
        System.out.println("ID\t Name \tSalary");
        System.out.println("---------------------");
        emp1.showData();
        emp2.showData();
        emp3.showData();
```

Output

```
ID       Name  Salary
---------------------
1        Jeon     900000
2        Honey    800000
3        Yuki     1000000

*** After changing employee 1 ***
1        Jeon     1200000

Sorry! Limited number had been reached
```

```
            emp1.changeData("Jeon", 1200000);
            System.out.println("*** After changing employee 1 ***");
            emp1.showData();

            //create next employee
            Employee emp4 = new Employee(); // exception occurs
            emp4.initializeData("Kyaw Kyaw", 400000);
} catch (AppException e) {
            System.err.println(e.getMessage());
}

}
```

Static vs. Instance Method

1. Static or Class Method

- Similar to static fields, static methods also belong to a class instead of the object.
- We can call them without creating the object of the class (***ClassName.method()***).
- They cannot be overridden, but can be overloaded.
- Static methods are widely used in utility and helper classes.
- They can only access to static members (variables & methods).
- Static methods can't use **this** or **super** keywords.

Syntax:

```
acess_modifier static  return_type   method_name(parameter_list)
  {
    // Method body
  }
```

Example

```
public static void changeNoOfEmployee(int count) {
            noOfemployee = count; // ok
            this.name = ""; //cannot use "this" keyword
            salary = 10000; // cannot access instance variable
            showData(); // cannot invoke instance method
}
System.out.println("Original : " + Employee.noOfemployee);
Employee.changeNoOfEmployee(10); // invoke static method
System.out.println("After Update: " + Employee.noOfemployee);
```

2. **Instance Method**

- Similar to instance fields, instance methods belong to the Object of the class, not to the class.
- An instance method is used to implement behaviors of each object/instance of the class.
- Instance methods can be called after creating the Object of the class (***objectName.method()***).
- Inside instance method, both instance & static data can be accessed.

Example

```
public void viewInformation() {

        System.out.println("Total Employee: " + noOfemployee); //can access static data
        System.out.println("*** Current Employee Info ***");
        showData(); // can access instance data
        System.out.println("Next employee id: " + nextId);

}
public static void main(String[] args) throws AppException
{
        Employee emp1 = new Employee();
        emp1.initializeData("Jeon", 900000);
        emp1.viewInformation();
}
```

Output

```
Total Employee: 3
*** Current Employee Info ***
1         Jeon     900000
Next employee id: 2
```

Static vs. Instance Block

1. **Static Block**

- Static block is generally used to initialize the static variables.
- This block is executed when the class is loaded.
- Only static data can be accessed inside the static block.

2. **Instance Block**

- This block is generally used to initialize the object's data.
- Instance block executes before the constructor for every object creation.
- Both static and instance data can be accessed inside the instance block.

Example

```
public class BlockDemo {

        static final int MIN_LENGTH;
        String phone;

        public BlockDemo(String phone)
        {
                System.out.println("This is constructor block");
                if(phone.length() >= MIN_LENGTH)
                        this.phone = phone;
        }

        { // instance block
                System.out.println("This is instance block!");
                phone = "Invalid number";
        }
        static { // static block
                MIN_LENGTH = 9;
                System.out.println("This is static block!");
        }
}

BlockDemo obj1 = new BlockDemo("09795578841");
System.out.println("Obj1's phone: " + obj1.phone);
System.out.println("***********");
BlockDemo obj2 = new BlockDemo("123");
System.out.println("Obj2's phone: " + obj2.phone);
```

Output

```
This is static block!
This is instance block!
This is constructor block
Obj1's phone: 09795578841
***********
This is instance block!
This is constructor block
Obj2's phone: Invalid number
```

Static block is executed once when class is loaded

Instance block is executed whenever new object is created

Flow of Execution of Program in Java

```
// Members of class
1. Field Declarations
2. Constructor Declarations
3. Method Declarations
4. Block Declarations
```

1. The static variable, static block, and static method are executed first during the loading of the .class file.
2. When the object is created, first, an instance block is executed before the execution of a constructor.
3. After the execution of an instance block, the constructor part will be executed.
4. After the execution of the constructor part, the instance method is executed.

2.3 Constructor

- A constructor is special method that is automatically called by JVM at the time of object creation.
- JVM first allocate the memory for the object and then execute the constructor to initialize the default value to the object's instance fields.
- A constructor in Java cannot be abstract, final, static and synchronized.
- The name of constructor must be the same as the class name.

Declaration syntax

```
Access-modifier ClassName(parameter_list)
{
// Constructor body which is a block of statements
}
```

Two Types of Constructor

1. **Default Constructor**
    - A constructor that has no parameter is known as the default constructor.
    - Default constructor provides the default values to the object like 0, null, etc. depending on the type.

- If we don’t define a constructor in a class, the complier implicitly create default constructor for the class.

2. **Parameterized Constructor**
   - A constructor that has parameters is known as parameterized constructor.
   - This constructor is used when we want to initialize the states of the object with our own default values.

Example

```
class Person{
        String name;
        int age;
        boolean is_single;

        // default constructor
        public Person() {
                // default values for(string, int,boolean) are (null,0,false)
        }
        // parameterized constructor
        public Person(String name,int age,boolean status) {
                this.name = name;
                this.age = age;
                this.is_single = status;
        }
        public void display() {
                System.out.println("Name: " + name);
                System.out.println("Age: " + age);
                System.out.println("Is single: " + ((is_single) ? "yes" : "no"));
                System.out.println("------------------");
        }
}
//Using default constructor
Person obj1 = new Person();
obj1.display();

// using parameterized constructor
Person obj2 = new Person("Jeon", 24, false);
Person obj3 = new Person("Yuki", 30, true);

obj2.display();
obj3.display();
```

Output

```
Name: null
Age: 0
Is single: no
------------------
Name: Jeon
Age: 24
Is single: no
------------------
Name: Yuki
Age: 30
Is single: yes
------------------
```

Copy Constructor

- Copy constructor is a special type of constructor that creates a new object from the existing object.
- This constructor takes an object of the same time as a single argument

Example

```
class Book{
        String title;
        LocalDate publishDate;
        int price;

        public Book(String title,LocalDate pDate,int price) {
                this.title = title;
                this.publishDate = pDate;
                this.price = price;
        }
        public Book(Book other) { // copy constructor
                this.title = other.title;
                this.publishDate = other.publishDate;
                this.price = other.price;
        }
        @Override
        public String toString() {
                String str = "[" + title + "," + publishDate + "," + price + "]";
                return str;
        }
}

LocalDate pubDate = LocalDate.of(1961, 10, 16);

Book book1 = new Book("Detective U San Shar",pubDate,6000);
System.out.println(book1.toString());

Book book2 = new Book(book1);
System.out.println(book2);
```

Output

```
[Detective U San Shar, 1961-10-16, 6000]
[Detective U San Shar, 1961-10-16, 6000]
```

Chained Constructor

- Invoking one constructor from another constructor is called constructor chaining.
  - **this ()** – used to call the constructors within the same class.
  - **super ()** – used to call super class's constructor.

Example – this () constructor

```
public class ConstructorChain {
        public ConstructorChain() {
                this(7);
                System.out.println("This is default constructor");
                //this(7);
        }
        public ConstructorChain(int i) {
                this(7,9);
                System.out.println(i);
        }
        public ConstructorChain(int i, int j) {
                System.out.println("i = " + i + ",j = " + j);
        }
}
```

// compile-time error
// this () must be first statement

Example – super () constructor

```
class Employee{
        private String name;
        public Employee() {

        }
        public Employee(String name) {
                this.name = name;
        }
}
class Teacher extends Employee{
        private String position;
        public Teacher(String name, String position) {
                super(name);
                this.position = position;
                //super(name); // compile-time error
        }
}
```

Private Constructor

- The primary use of private constructors is to restrict object creation from outside the class.

Example

```
class Animal{
        private Animal() {
                // private constructor
        }
}
new Animal(); // compile-time error
```

- A class cannot be extended when a constructor is declared as private.

Example

```
class Dog extends Animal{ // error (can't inherit )


}
```

- Private constructors is mostly used to create a singleton class that can have only one instance. To implement a singleton class, it must have –
  - A private constructor
  - A static field to store its only instance
  - A static method to obtain the instance

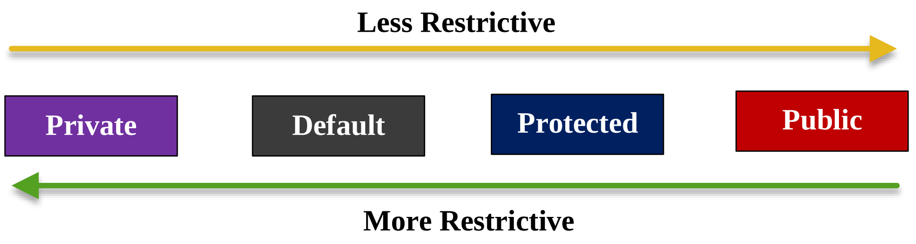
Example

```
class DatabaseConfig {
        String db_name = "employeedb";
        // static field
        private static DatabaseConfig config = null;
        // private constructor
        private DatabaseConfig() {

        }
        // static method
        static DatabaseConfig getInstance() {
                if(config == null)
                        config = new DatabaseConfig();
                return config;
        }
}
```

```
DatabaseConfig obj1 = DatabaseConfig.getInstance();
DatabaseConfig obj2 = DatabaseConfig.getInstance();
System.out.println("obj1's db_name: " + obj1.db_name);
System.out.println("obj2's db_name: " + obj2.db_name);
obj2.db_name = "bankingsystem";
System.out.println("obj1's db_name: " + obj1.db_name);
```

2.4 Access Modifiers

- Access Modifiers define the boundary for accessing the class's members (variables, methods, constructors) and the class itself.
- Four access modifiers: public, private, protected and default (no keyword).



Note – Top-level class can use public or default access modifier only. At the member level, can be use all four.

1. Private

- Accessible only within the class. Private can't be used to local variable.

Example

public class A {	class B{

```
public class A {
        private int number = 10;
        void test() {
                private int a; // error (invalid)
                number = 20; //ok(within class)
        }
}
```
	```
	class B{
	        void test() {
	                A obj = new A();
	                //error (outside class)
	                obj.number = 20;
	        }
	}
	```

2. Default

- Accessible within the same package only and can be inherited to the subclass.

Example

```
package oop.pkg1;

public class A {
    //default(no keyword)
    int number = 10;
    void test() {
        //ok(within class)
        number = 20;
    }
}
```

```
package oop.pkg1;

class B{
    void test() {
        A obj = new A();
        //ok (same package)
        obj.number = 20;
    }
}
```

```
package oop.pkg2;

import oop.pkg1.A;

public class D{
    void test() {
        A obj = new A();
        obj.number = 20; //error (different package)
    }
}
```

3. Protected

- Accessible within package and outside the package but through inheritance only.
- Protected members can be inherited to the subclass.

Example

```
package oop.pkg1;

public class A {
    protected int number = 10;
    void test() {
        //ok (within class)
        number = 20;
    }
}
```

```
package oop.pkg1;

class B{
    void test() {
        A obj = new A();
        //ok (same package)
        obj.number = 20;
    }
}
```

```
package oop.pkg2;

import oop.pkg1.A;

public class D extends A{
    void test() {
```

```
package oop.pkg2;

import oop.pkg1.A;

public class C {
    void test() {
        D obj = new D();
        // ok (via inheritance)
        obj.number = 20;

    }
}
```

```
        A obj = new A();
        // error (not subclass)
        obj.number = 20;

    }
}
```

4. Public

- Accessible everywhere and can also be inherited to subclasses.

Example

```
package oop.pkg1;

public class A {

    public int number = 10;
    void test() {
        //ok (within class)
        number = 20;
    }
}
```

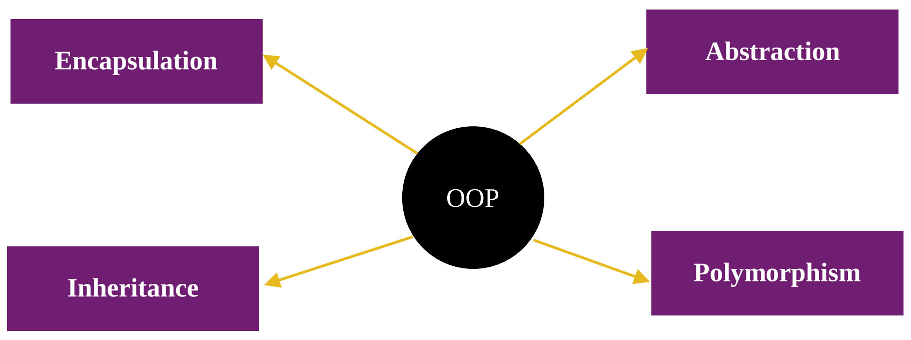
```
package oop.pkg2;

import oop.pkg1.A;

public class C {
    void test() {
        A obj = new A();
        //ok(everywhere)
        obj.number = 20;

    }
}
```

ASSIGNMENT

- Create a BankAccount class that contains
  - Fields
    - accountNo(int)
    - holderName(String)
    - pinNo(String)
    - password(String)
    - balance(int)
  - Constructor
    - 4 arguments constructor to initialize data
  - Methods
    - deposit(int)
    - withdraw(int)
    - changePassword(String)
    - showInfo()
- Inside withdraw () method, if parameter withdraw amount is not enough, InsufficientAmount exception will be fired.
- Inside deposit () method, parameter deposit amount is added to current balance.
- Inside changePassword() method, the following instructions are executed -
  - Get pinNo value from user.
  - Password can be changed only when user entered pinNo value is matched.
  - If the entered pinNo value is not matched to current pinNo, display error message “Invalid pin number!”
- Inside showInfo() method, display all bank account information.

- Create demo class that contain main method. Inside this class, execute the following instructions.
  1. Create BankAccount object using argument constructors.
  2. Test deposit(), withdraw(), changePassword() and showInfo() method
- After calling deposit () or withdraw () method, display the current updated balance.

MAIN FEATURES OF OOP



These are the main OOPs Concepts that you must learn to understand the Object Oriented Programming.

1. Encapsulation

- Binding the data and methods into a single unit (class) and hiding the state or internal representation of an object.
- To do so, declare the fields as private and providing access to them with getter and setter methods.

Example

```
public class Product {

        // ...
        private int barCode;

        public int getBarCode() { //getter
                return barCode;
        }

        public void setBarCode(int barCode) { //setter
                this.barCode = barCode;
        }
        // …
}
```

2. Inheritance

- Reusability of code.
- "Is-a" relationship or "parent-child" relationship between classes. (E.g. student is a person)
- Two terminologies – **super** class and **subclass**.
- Keyword "**extends**" is used to inherit a class.

Example

```
public class Person {
        // …
}
class Student extends Person{
      // …
}
```

Example

```
class Person{
        private String name;
        protected String phone;

        public Person(String name,String phone) {
                this.name = name;
                this.phone = phone;
        }
        void display() {
                System.out.println("Name - " + name);
                System.out.println("Phone - " + phone);
        }
}
class Student extends Person{
        private int rno;

        public Student(int rno,String name,String phone) {
                super(name,phone);
                this.rno = rno;
        }
        public int getRno() {
                return rno;
        }
}

Student obj = new Student(1, "Jeon", "09797768871");
obj.display(); // its parent class's method
System.out.println("Roll number - " + obj.getRno());
```

Output

```
Name - Jeon
Phone - 09797768871
Roll number - 1
```

Note – whenever child class’s constructor is called, its parent class’s default constructor is automatically invoked.

Example

```
class A{
        int i;
        public A() {
                System.out.println("A's default constructor");
        }
        public A(int i) {
                System.out.println("A's argument constructor");
        }
}
class B extends A{
        int j;
        public B() {
                System.out.println("B's default constructor");
        }
        public B(int a,int b) {
                System.out.println("B's argument constructor");
        }
}

B obj1 = new B(); // using default constructor
B obj2 = new B(10,20); // using argument constructor
```

Output

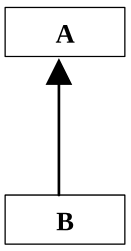
```
A's default constructor
B's default constructor
A's default constructor
B's argument constructor
```

2.1 Types of Inheritance

1. Single Inheritance

- A parent class has only one child class.

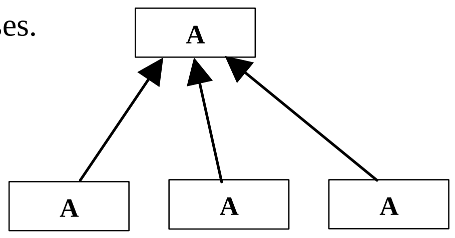
```
class Person{

}
class Student extends Person{

}
```



2. Hierarchical Inheritance

- A parent class has more than one child classes.

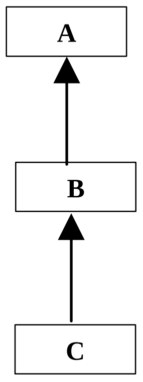
```
class Person{ }
class Student extends Person{ }
class Teacher extends Person{ }
class Programmer extends Person{ }
```



3. Multilevel Inheritance

- Inheritance occurs more than one level.

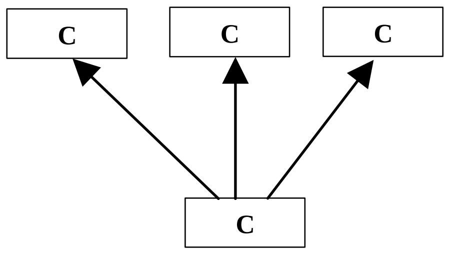
```
class Person{

}
class Employee extends Person{

}
class Programmer extends Employee{

}
```



4. Multiple inheritance

- A child class can inherit from more than one parent classes.
- Java does not support multiple inheritance for classes.

```
class Teacher{

}
class Student{

}
class TeachingAssistant extends Teacher, Student{
// does not support
}
```



2.2 Method Overriding

- Child class can redefine the existing instance method of its parent class.
- Cannot override private, static and final method.

```
class Animal{
        void show() {
                System.out.println("This is show method");
        }
        void sound() { // overridden method
                System.out.println("Some sound");
        }
}
class Cat extends Animal{
        void sound() { // overriding method
                System.out.println("myaung");
        }
}
Cat cat = new Cat();
cat.show();
cat.sound(); // call overriding method
```

Output

```
This is show method
myaung
```

- JVM resolve method overriding at run-time. So it is also called runtime polymorphism.

Example

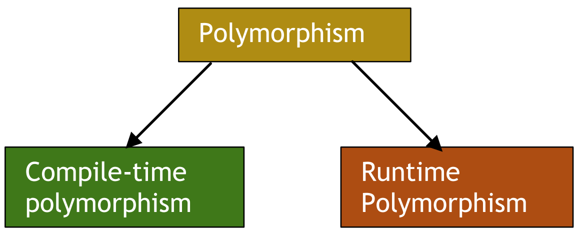
```
class Person{
        private String name;
        public Person(String name) {
                this.name = name;
        }
        public void showInfo() {
                System.out.println("Name: " + name);
        }
}
class Teacher extends Person{
        private String position;
        public Teacher(String name,String pos) {
                super(name);
                this.position = pos;
        }
        @Override
        public void showInfo() {
                super.showInfo();
                System.out.println("Position: " + position);
                System.out.println("------------------");
        }
}

Person p = new Person("James");
p.showInfo(); // call its method
Teacher t = new Teacher("Roly", "Instructor");
t.showInfo();  // call its method

Person p1 = new Teacher("David","Tutor");
p1.showInfo(); // call reference obj's method
```

Output

```
Name: James
Name: Roly
Position: Instructor
------------------
Name: David
Position: Tutor
------------------
```

3. Polymorphism

- A Single interface for multiple implementations depending on conditions.
- E.g. A person may be an employee, customer or passenger.
- Type of Polymorphism
    1. Static or compile-time polymorphism
    2. Dynamic or runtime polymorphism



Compile-time Polymorphism

- Resolved at compile time.
- It is achieved through method overloading in Java.
- **Method overloading** means a class has multiple methods with the same name, but different types/order/number of parameters.

Example

```
class Addition{
        static int add(int a,int b) {

                return (a + b);
        }
        static int add(int[] input) {

                return Arrays.stream(input).sum();
        }
        static float add(float a,float b) {

                return (a + b);
        }
        static String add(String a, String b) {

                return (a + b);
        }
}

System.out.println("100 + 200 = " + Addition.add(100, 200));
System.out.println("23.3 + 20.5 = " + Addition.add(23.3f, 20.5f));
System.out.println("'hello' + 'wold' = " + Addition.add("Hello", "Word"));
System.out.println("int array's sum = " + Addition.add(new int[]{10,20,30,40}));
```

Runtime Polymorphism

- Resolved at runtime.
- It is achieved through method overriding mechanism.
- **Method overriding** means a child class have a method with the same name and same parameters as its parent class, but different implementations

Example

```
class Developer{
        void work() {
                System.out.println("Some work");
        }
}
```

```
class FrontendDeveloper extends Developer{

        @Override
         void work() {
                  System.out.println("doing frontend technologies");
         }
}
class BackendDeveloper extends Developer{

         @Override
         void work() {
                  System.out.println("doing backend technologies.");
         }
}

Developer dev = new Developer();
dev.work();
dev = new BackendDeveloper();
dev.work();
dev = new FrontendDeveloper();
dev.work();
```

Output

```
Some work
doing backend technologies.
doing frontend technologies
```

4. Abstraction

- "**Showing**" only the essential attributes of something and "**hiding**" implementation details that is unnecessary to the user.
- In Java, abstraction is achieved through abstract classes and interfaces.

Abstract Class

- A superclass that cannot be instantiated.
- Abstract classes can have both abstract and concrete methods.
- Abstract methods contain only the method signature.

Syntax

```
abstract class className{
         //abstract method
         abstract return-type methodName(parameters);
         //concrete method
         return-type method1() {

         }
}
```

- Concrete child classes must implement all the abstract methods of the abstract class.

Example

```
public abstract class DatabaseUtil {

	private String db_name = "shopdb";

	public void connectDatabase() {
		System.out.println("Connecting to " + db_name + "...");
	}

	public abstract void insert();
	public abstract void update();
	public abstract boolean delete(int id);
	public abstract Object findById(int id);

}
public class ProductService extends DatabaseUtil{

	@Override
	public void insert() {
		System.out.println("insert into product");

	}
	@Override
	public void update() {
		System.out.println("proudct table update");

	}
	@Override
	public boolean delete(int id) {
		System.out.println("delete from proudct where id = " + id);
		return false;
	}
	@Override
	public Object findById(int id) {
		System.out.println("select * from product where id = " + id);
		return null;
	}

ProductService service = new ProductService();
service.connectDatabase();
service.insert();
service.update();
service.delete(1);
service.findById(10);
}
```

Output

```
Connecting to shopdb...
insert into product
proudct table update
delete from proudct where id = 1
select * from product where id = 10
```

Interface

- An abstract type that contains a collection of methods and constant variables.
- Interface does not have constructor and it cannot be instantiated.
- Interface variables are internally public, static, and final.
- Interface can contains these types of methods.
    - Abstract method – implicitly public and abstract
    - Default method & Static method – introduced in Java 8
    - Private method – added in Java 9

Syntax:

```
interface interface-name {
   // declare constant fields
   // declare methods
       1.  Abstract method
       2.  Default method
       3.  Static method
       4.  Private method
}
```

Example

```
public interface OnClickListener {
        int FONT_SIZE = 12;
        String FONT_FAMILY = "Arial";
        void onClick();
        void onDoubleClick();
}
class Button implements OnClickListener{
        private String name;
        public Button(String name) {
                this.name = name;
        }
        public void display() {
                System.out.println("Font size: " + FONT_SIZE);
                System.out.println("Font family: " + FONT_FAMILY);
        }

        public void onClick() {
                System.out.println(name + " click event");

        }

        public void onDoubleClick() {
```

```
                    System.out.println(name + " double click event");
          }

}

Button btn1 = new Button("btn-login");
Button btn2 = new Button("btn-logout");

btn1.display();
btn1.onClick();
btn1.onDoubleClick();
btn2.onClick();
btn2.onDoubleClick();
```

Output

```
Font size: 12
Font family: Arial
btn-login click event
btn-login double click event
btn-logout click event
btn-logout double click event
```

- Interface allows multiple inheritance.

Example

```
interface onKeyChangedListener{
          void onKeydown();
          void onKeypress();
          void onKeyup();
}
class selectBox implements OnClickListener, onKeyChangedListener{
          public void onKeydown() { }
          public void onKeypress() { }
          public void onKeyup() { }
          public void onClick() {  }
          public void onDoubleClick() { }


}
```

Polymorphism via Interface

```
interface Flying{
          void fly();
}
class Bird implements Flying{

          @Override
          public void fly() {
                    System.out.println("Birds fly with wings");
          }
}
class Airplane implements Flying{
        @Override
        public void fly() {
                System.out.println("Airplanes fly with engine");
        }
}
class Human implements Flying{
        @Override
        public void fly() {
                System.out.println("Human fly with parachute");
        }
}

Flying[] data = new Flying[3] ;
data[0] = new Human();
data[1] = new Bird();
data[2] = new Airplane();

for(var i = 0;i < 3;i++)
        data[i].fly();
```

Output

```
Human fly with parachute
Birds fly with wings
Airplanes fly with engine
```

Default & Static Method

- **Default method**
  - It is used to easily add new method in the existing interface without breaking down the old code.
  - Child class may override the default method.
- **Static Method**
  - It is similar to default method but cannot override.

Example

```
public interface InterfaceTest {

        void method1();

        default void method2() {
                System.out.println("Adding new method.It can be overriden");
        }
        static void method3() {
                System.out.println("Adding new method.It cannot be overriden");
        }
}
class MyClass implements InterfaceTest{

        @Override
        public void method1() {
                System.out.println("This is overriding method");
        }
}

MyClass obj = new MyClass();
obj.method1();
obj.method2();
InterfaceTest.method3();
```

Assignment 1 (Encapsulation)

- Create Student class that contains
  - - studentId(int)
  - - name(String)
  - - mark(int)
  - + getters/setters
  - + Constructors
  - + display () method – display student data
- Create Testing class that contains these instructions
  - Create student object array (length=4) and initialize this array from input user data.
  - Display all student data (calling Display() method)
  - User gives studentId to enquiry the data. If it is found, display the data. Else display message “student id-222 is not found!”.
  - Find the student whose mark is maximum.
  - Find the average marks over all the students.

Assignment 2 (Inheritance)

- Create classes with the following fields and methods and test their methods.

Person Class

- name(String)
- nrcno(String)
- address(String)
- phone(String)

+ Constructors

+ getter/setter

+ showInfo()

+ showIdentificationInfo()

  - which division/state
  - which city
  - what number

Teacher class inherits From Person

- position
- department
- salary

+ Constructors

+ getter/setter

+ promote (String,String) – update position and salary

+ transfer (String) – transfer new department

+ showTeacherInfo() (Hints: Show all information of Teacher)

Assignment 3 (Abstraction)

Define an abstract class named **Shape** containing:

- color(String)

+ getter/setter

+ An abstract method named Area that has a return type double.

Define concrete class named **Rectangle** which is extended from class Shape containing:

- length(int)
- width(int)

+ A full parameterized constructor which initializes all the instance variables of both the classes (by calling setColor method).
+ An overridden method named Area which returns the calculated area of the rectangle.
+ A method named displayArea which displays the area of rectangle.

Define a third class named **Demo** contains a main method that test the methods of the above classes.

Assignment 4 (Polymorphism)

Define an interface named **Shape** containing:

- A method named area which has a return type double.
- A method named volume which has a return type double.

Define another class named **Cube** which implements Shape containing:

- A variable named x of type int.
- Implement the area method which returns the calculated area of the cube (i.e, 6*x*x).
- Implement the volume method which returns the calculated volume of the cube (i.e, x*x*x).

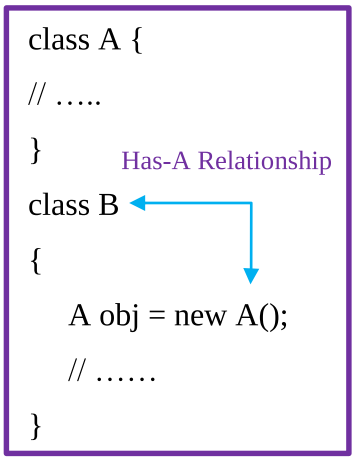
Define another class named **Circle** which implements Shape containing:

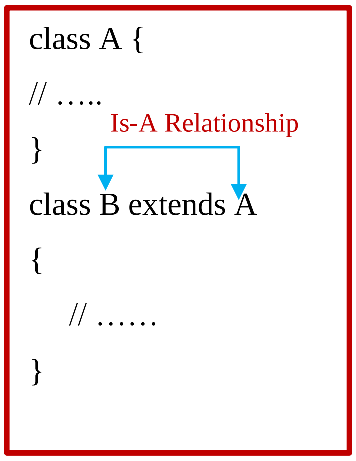
- radius (int)
- Implement the area method which returns the calculated area of the circle (i.e, $A=\pi r$^2).
- Implement the volume method which returns the value 0.

Define a third class named **PolymorphismTest** which contains a main method that demonstrate polymorphism feature. Hints -

- Create Shape array to store shape objects and initialize data.
- Call each object’s different methods.

“IS-A” VS. “HAS-A” RELATIONSHIP

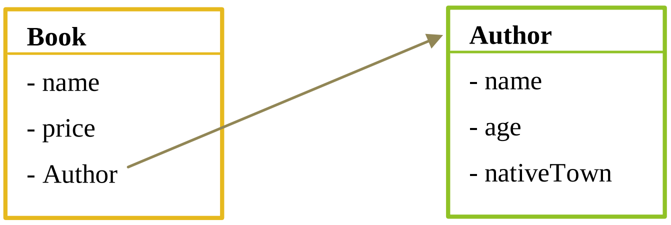

```
class A {
// …..
}
class B
{
    A obj = new A();
    // ……
}
```


```
class A {
// …..
}
class B extends A
{
    // ……
}
```

“Has-A” Relationship

Association relationship between two classes (e.g. Person has an address).

A class contains object of another class.



Example -1

```
class Author{
        String name;
        String nativeTown;
        public Author(String name,String town) {
                this.name = name;
                this.nativeTown = town;
        }
}
class Book{
        String name;
        int price;
        Author author;

        public Book(String name,int price,Author author) {
                this.name = name;
                this.price = price;
                this.author = author;
        }
}
```

Output

```
Book Name: War and Peace
Book Price: 6800
****** Author Details *******
Author Name: Mya Than Tint
Native Town: Myaing
```

Example 2

```
class Product{
        String name;
        int price;
        public Product(String name,int price) {
                this.name = name;
                this.price = price;
        }
}
class SaleRecord{
        int id;
        int saleQty;
        LocalDate saleDate;
        Product product;
        public SaleRecord(int id,Product prod,int qty) {
                this.id = id;
                this.product = prod;
                this.saleQty = qty;
                this.saleDate = LocalDate.now();
        }
        void showData() {
                System.out.print(id + "\t" + saleDate + "\t");
                System.out.print(product.name + "\t" + product.price + "\t");
                System.out.print(saleQty + "\t" + (saleQty * product.price));
                System.out.println();
        }
}
SaleRecord[] records = new SaleRecord[3];

Product prod1 = new Product("Coffee", 3700);
Product prod2 = new Product("Juice", 1500);

records[0] = new SaleRecord(1001, prod1, 10);
records[1] = new SaleRecord(1002, prod2, 5);
records[2] = new SaleRecord(1003, prod1, 6);

System.out.println("No.\tSale Date\tProduct\tPrice\tQty\tSubTotal");
System.out.print\n("--------------------------------");
for(SaleRecord rec: records)
        rec.showData();
```

```
No.     Sale Date       Product Price   Qty     SubTotal
------------------------------------------------------------------------
1001    2021-12-21      Coffee  3700    10      37000
1002    2021-12-21      Juice   1500    5       7500
1003    2021-12-21      Coffee  3700    6       22200
```