Chapter 03 The Four Pillars of OOP

The whole OOP concept depends on four main ideas, which are known as the pillars of OOP. These four pillars are as follows:

- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

1. Inheritances

The word inheritance means receiving something from something else. In real life, we might talk about a child inheriting a house from his or her parents. In that case, the child has the same power over the house that his parents had.

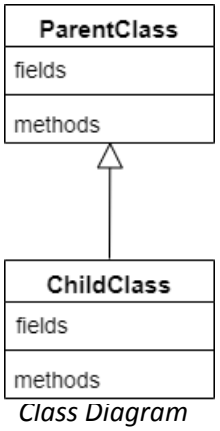
In OOP, one class is allowed to inherit the features (data fields and methods) of another class. This is called **inheritance**. Inheritance provides **reusability** of a code, i.e., when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the data fields and methods of the existing class.

In inheritance, there are two kinds of classes:

- **Parent class** (aka. superclass or base class): The class whose features are inherited.
- **Child class** (aka. subclass or derived class): The class that inherits another class. The child class can add its own data fields and methods in addition to the parent class data fields and methods. Also, a child class can also provide its specific implementation to the methods of the parent class.

Syntax: Here's the syntax of the inheritance:

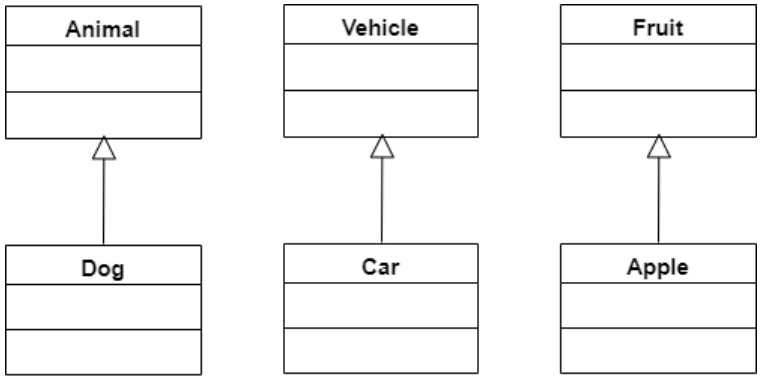
```
// define a parent class
class ParentClass {
    // fields
    // methods
}

// define a chile class
class ChildClass extends ParentClass {  // inheritance
    // fields
    // methods
}
```



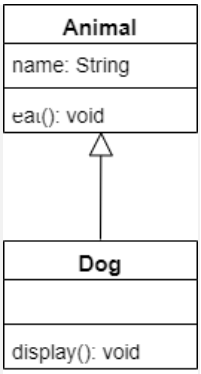
Class Diagram

Inheritance is an **is-a relationship**. That is, we use inheritance only if there exists an **is-a relationship** between two classes. For example,

`Dog` is an `Animal`, so `Dog` can inherit from `Animal`
`Car` is a `Vehicle`, so `Car` can inherit from `Vehicle`
`Apple` is a `Fruit`, so `Apple` can inherit from `Fruit`



Example 01: Using Inheritance




```java
class Animal{

    String name;

    void eat(){
        System.out.println("The animal is eating");
    }
}
class Dog extends Animal {   // inherit from Animal
    // new method in child class
    void display() {
        // access name field of the parent class
        System.out.println("Its name is" + name);
    }
}
class Program {
    public static void main(String[] args){
        // create an object of the child class
        Dog dog1 = new Dog();

        // access parent class data field and method
        dog1.name = "Lucky";
        dog1.eat();

        // call child class method
        dog1.display();
    }
}
```

Output:

```
The dog is eating
Its name is Lucky
```

Constructors are not inherited in Java. But it is always called every time the child class's object is created.

Example 02: A program that shows that the parent constructor is called every creation of a child class's object.

```
class ParentClass {
    ParentClass(){
        System.out.println("Parent Constructor is called");
    }
}
class ChildClass extends ParentClass {
    ChildClass() {
        System.out.println("Child Constructor is called");
    }
}
class Program {
    public static void main(String[] args){
        ChildClass child1 = new ChildClass();
    }
}
```

Output:

```
Parent Constructor is called
Child Constructor is called
```

Again, when a child class object is to be created, the parent class's constructor is invoked. The compiler will be aware of only the default constructor. So, if the parent does not have a default constructor or a no-arg constructor, the compiler will throw an error.

Example 03: A program that shows that the parent's default constructor is needed.

```
class ParentClass {

    String name;

    ParentClass(){
         System.out.println("Parent Constructor is called");
    }
    ParentClass(String name){
        this.name = name;
        System.out.println("The parent's name is " + name);
    }
}
```

```
class ChildClass extends ParentClass {
    ChildClass(String name) {
        System.out.println("The child's name is " + name);
    }
}
class Program {
    public static void main(String[] args){
        ChildClass child1 = new ChildClass("John");
    }
}
```

Output:

```
Parent Constructor is called
The child's name is John
```

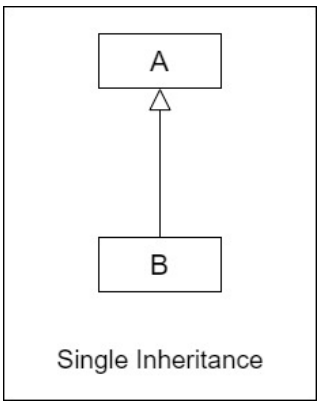
2. Types of Inheritances

There are five types of inheritances:

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

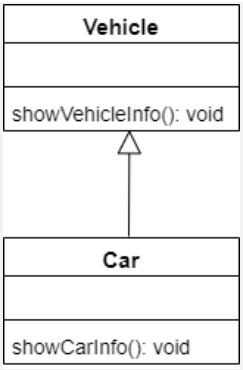
2.1 Single Inheritance

In single inheritance, a child class inherits the features of a parent class. In a diagram below, the class A serves as a parent class for the child class B.



Example 04: Single Inheritance

```
// Parent class
class Vehicle {
    void showVehicleInfo(){
        System.out.println("This is a vehicle");
    }
}
// Child class
class Car extends Vehicle {
    void showCarInfo(){
        System.out.println("This is a car");
    }
}
class Program {
    public static void main(String[] args){
        // Create an object of Car
        Car car = new Car();

        // Access Vehicle's info using the car object
        car.showVehicleInfo();

        // Access Car's info using its own car object
        car.showCarInfo();
    }
}
```

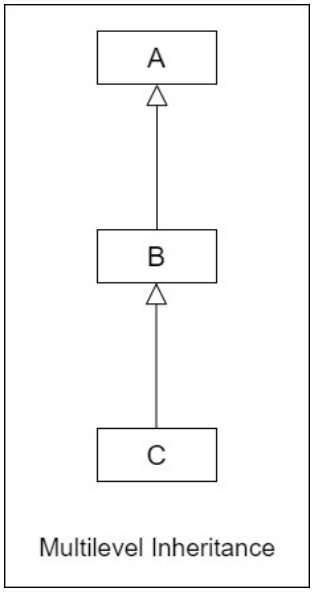


Output:

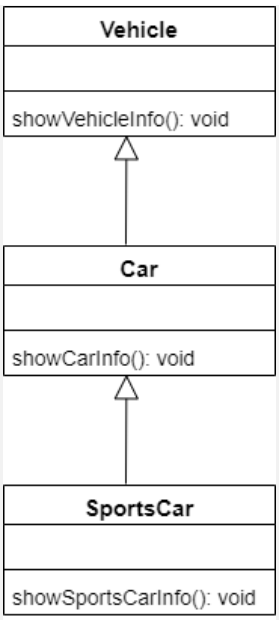
```
This is a vehicle
This is a car
```

2.2 Multilevel Inheritance

In Multilevel Inheritance, a child class inherits from a parent class and as well as the child class also acts as a parent class to another class. In below diagram, class A serves as a parent class for the child class B, which in turn serves as a parent class for the child class C.



Example 05: Multilevel Inheritance



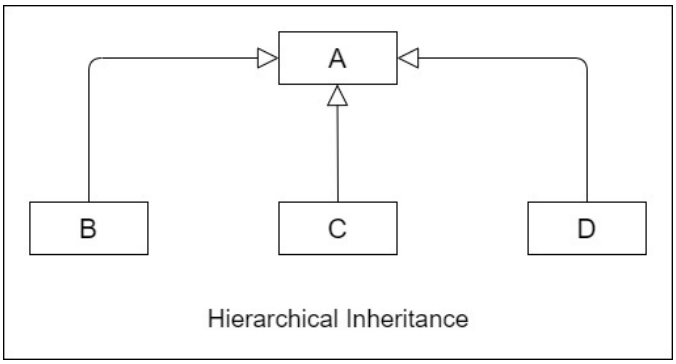

```java
// Parent class
class Vehicle {
    void showVehicleInfo(){
        System.out.println("This is a vehicle");
    }
}
// Child class
class Car extends Vehicle {
    void showCarInfo(){
        System.out.println("This is a car");
    }
}
// Child class
class SportsCar extends Car {
    void showSportsCarInfo(){
        System.out.println("This is a sport car");
    }
}
class Program {
    public static void main(String[] args){
        // Create an object of SportCar
        SportsCar sport_car = new SportsCar();

        // Access Vehicle's info using the sport car object
        sport_car.showVehicleInfo();

        // Access Car's info using the sport car object
        sport_car.showCarInfo();

        // Access SportCar's info using its own sport car object
        sport_car.showSportsCarInfo();
    }
}
```

Output:

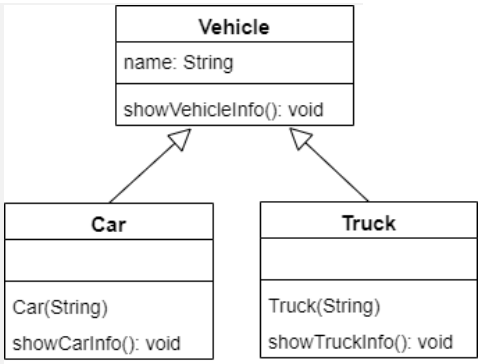
```
This is a vehicle
This is a car
This is a sport car
```

2.3 Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a parent class for more than one child class. In below diagram, class A serves as a parent class for the child class B, C, and D.



Example 06: Hierarchical Inheritance




```java
// Parent class

class Vehicle {
    String name;

    void showVehicleInfo(){
        System.out.println("Vehicle's name is " + name);
    }
}
// Child class 1
class Car extends Vehicle {
    Car(String name){
        this.name = name;
    }
    void showCarInfo(){
        System.out.println("Car's name is " + name);
    }
}
// Child class 2
class Truck extends Vehicle {
    Truck(String name){
        this.name = name;
    }
    void showTruckInfo(){
        System.out.println("Truck's name is " + name);
    }
}
```

```java
class Program {
    public static void main(String[] args){
        // Create an object of Car
        Car car = new Car("BMW");

        // Create an object of Truck
        Truck truck = new Truck("Ford");

        // Display the car info
        car.showVehicleInfo();
        car.showCarInfo();

        // Display the truck info
        truck.showVehicleInfo();
        truck.showTruckInfo();
    }
}
```

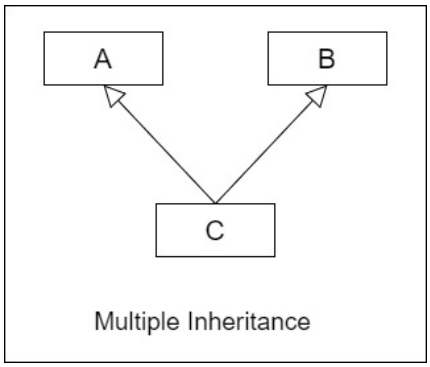
Output:

```
Vehicle's name is BMW
Car's name is BMW
Vehicle's name is Ford
Truck's name is Ford
```

2.4 Multiple Inheritance

In Multiple inheritance, one class can have more than one parent classes and inherit features from all parent class classes.

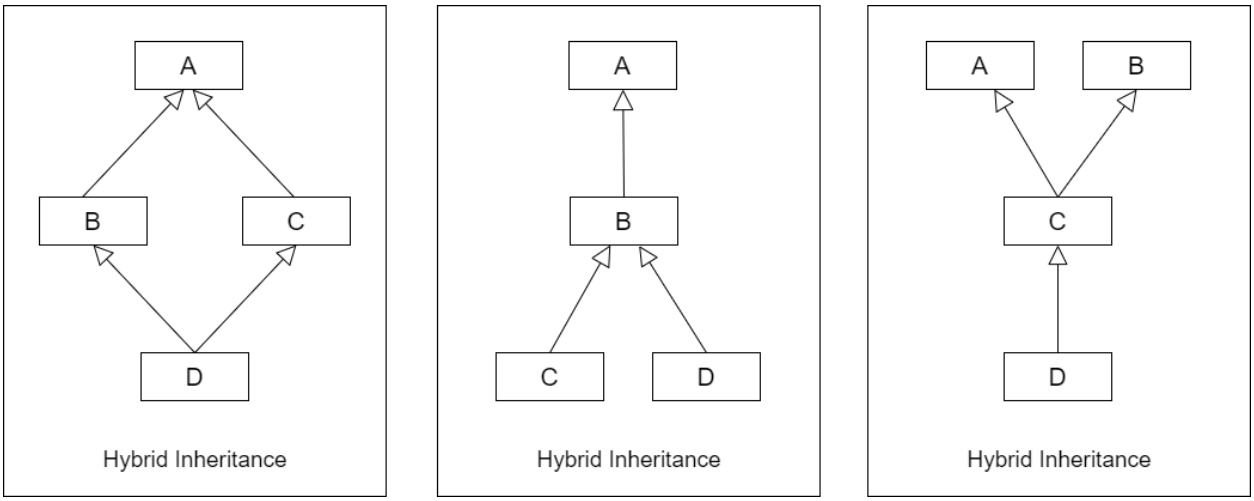
Unlike some other popular OOP languages like C++ or Python, Java does not provide support for multiple inheritance using classes. Instead, Java uses better ways through which we can achieve the same result as multiple inheritances. This will be presented in the later section.



Multiple Inheritance

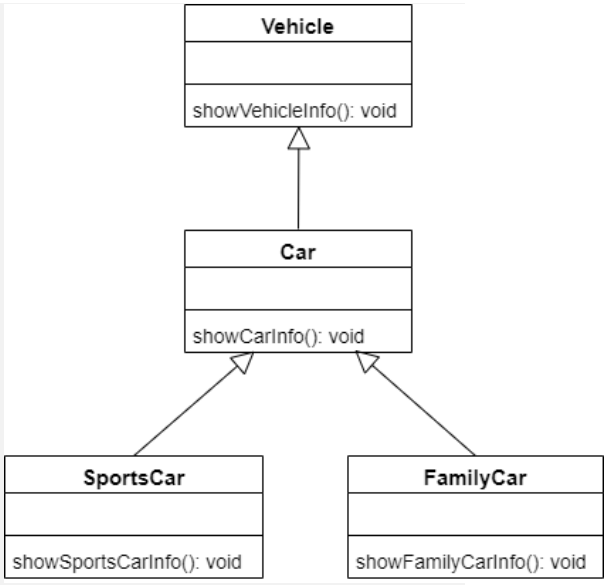
2.5 Hybrid Inheritance

When inheritance consists of multiple different types of inheritance is called **Hybrid Inheritance**. Below are some examples:



Example 07: Hybrid Inheritance

```
class Vehicle {

    void showVehicleInfo(){
        System.out.println("This is a vehicle");
    }
}
class Car extends Vehicle {
    void showCarInfo(){
        System.out.println("This is a car");
    }
}
class SportsCar extends Car {
    void showSportsCarInfo(){
        System.out.println("This is a sports car");
    }
}
class FamilyCar extends Car {
    void showFamilyCarInfo(){
        System.out.println("This is a family car");
    }
}
```



```
class Program {
    public static void main(String[] args){
        // Create objects
        SportsCar sportsCar = new SportsCar();

        // Display the sports car info
        sportsCar.showVehicleInfo();
        sportsCar.showCarInfo();
        sportsCar.showSportsCarInfo();
    }
}
```

Output:

```
This is a vehicle
This is a car
This is a sports car
```

3. The `super` Keyword

The **`super`** keyword in Java is used in a child class to access parent class members (fields, constructors and methods).

Uses of super keyword

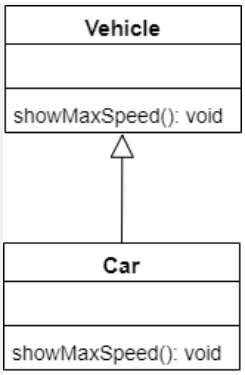
- To call methods of the parent class that is overridden in the child class.
- To access fields of the parent class if both superclass and child class have fields with the same name.
- To explicitly call parent class's constructor from the child class's constructor.

3.1 Method Overriding

In inheritance, all members available in the parent class are by default available in the child class. If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional methods in the child class. This concept is called **method overriding**.

When a child class method has the same name, same parameters, and same return type as a method in its parent class, then the method in the child is said to override the method in the parent class.

Example 08: We create two classes named Vehicle and Car. The class Car extends from the class Vehicle so, all members of the Vehicle class are available in the Car class. In addition to that, the Car class redefined the method `getSpeed()`.




```
class Vehicle {
    int getSpeed(){
        return 100;
    }
}
class Car extends Vehicle {
    // overridden the implementation of Vehicle class
    int getSpeed(){
        return 200;
    }
    void showMaxSpeed(){
        System.out.println("Max speed is " + getSpeed() + "km/hour");
    }
}
class Program {
    public static void main(String[] args){
        Car car = new Car();
        car.showMaxSpeed();
    }
}
```

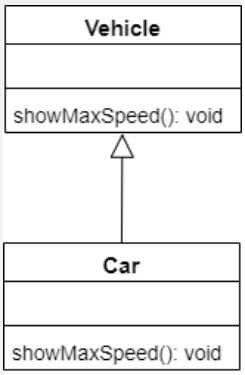
Output:

```
Max speed is 200 km/hour
```

Since `getSpeed()` is defined in both the classes, the method of child class `Car` overrides the method of parent class `Vehicle`. Hence, the `getSpeed()` of the child class is called.

What if the overridden method of the parent class has to be called? In this case, we use can use the **super** keyword to do it.

Example 09: Using the **super** keyword to call/access the parent class's members.

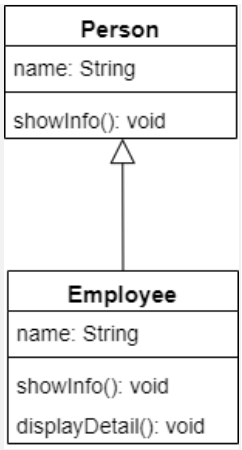



```
class Vehicle {
    int getSpeed(){
        return 100;
    }
}
class Car extends Vehicle {
    // overridden the implementation of Vehicle class
    int getSpeed(){
        return 200;
    }
    void showMaxSpeed(){
        System.out.println("Max speed is " + super.getSpeed() + "km/hour");
    }
}
class Program {
    public static void main(String[] args){
        Car car = new Car();
        car.showMaxSpeed();
    }
}
```

Output:

```
Max speed is 100 km/hour
```

Example 10: Another example of using the **`super`** keyword to access to parent class's members.




```
// Parent class
class Person {
    String name = "John";

    void showInfo(){
        System.out.println("This is a person");
    }
}
// Child class
class Employee extends Person {
    String name = "Jackie";

    void showInfo(){
        System.out.println("This is an employee");
    }
    void displayDetail(){
        super.showInfo();  // call parent's method
        System.out.println("Person name:" + super.name); // access parent's data
                                                         // field
        showInfo();
        System.out.println("Employee name:" + name);
    }
}
class Program {
    public static void main(String[] args){
        Employee employee = new Employee();
        employee.displayDetail();
    }
}
```

Output:

```
This is a person
Person name: John
This is an employee
Employee name: Jackie
```

As we know, when a child class's object is created, its default constructor of the parent class is automatically called. We can also do this explicitly by calling the parent class's constructor from the child class constructor using `super()` which is a special form of the `super` keyword. `super()` can be used only inside the child class constructor and must be the first statement.

Example 11: Use of `super()` to explicitly call the parent class's constructor.

```
class Animal {
    Animal() {
        System.out.println("This is an animal");
    }
}
class Dog extends Animal {
    Dog() {
        // calling the constructor of the parent class
        super();

        System.out.println("This is a dog");
    }
}
class Program {
    public static void main(String[] args) {
        Dog dog1 = new Dog();
    }
}
```

Output

```
This is an animal
This is a dog
```

Example 12: Use of `super()` to explicitly call the parent class's parameterized constructor.

```
class Animal {
    // default or no-arg constructor
    Animal() {
        System.out.println("This is an animal");
    }
    // parameterized constructor
    Animal(String type) {
        System.out.println("Type: " + type);
    }
}
class Dog extends Animal {
    // default constructor
    Dog(){
        // call parameterized constructor of the parent class
        super("Animal");

        System.out.println("This is a dog");
    }
}
class Program {
    public static void main(String[] args) {
        Dog dog1 = new Dog();
    }
}
```

Output

```
Type: Animal
This is a dog
```

5. Encapsulations

In OOP, encapsulation refers to binding related fields and methods together. Encapsulation allows us to restrict access to fields and methods directly and prevent accidental data modification from outside of class by creating private data fields and methods within a class.

If you are working with the class and dealing with sensitive data, providing access to all fields used within the class publicly is not a good choice.

Let's say you have a `BankAccount` class. For security reasons, you might need to, for example, make some class members such as `accountBalance`, `taxRate` and `getBalanceAfterTax()` private in order to do some validation checks before allowing these members to be accessed from outside the class.

5.1. Access Modifiers

Encapsulation can be used to achieve data hiding by declaring the data members and methods of a class either as private or protected.

Access Modifiers specify the accessibility (visibility) of classes, interfaces, fields, methods, constructors, and the setter methods.

There are four types of access modifiers available in Java:

- Default – No keyword required: declarations are visible only within the package (package private).
- Private: declarations are visible within the class only.
- Protected: declarations are visible within the package or all child classes.
- Public: declarations are visible everywhere.

The visibility of these modifiers increases in this order:

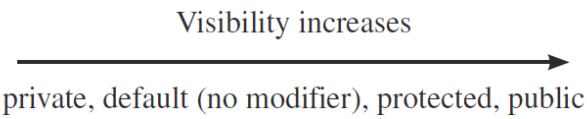


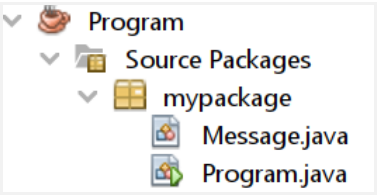
TABLE 11.2 Data and Methods Visibility

Modifier on Members in a Class	*Accessed from the Same Class*	*Accessed from the Same Package*	*Accessed from a Subclass in a Different Package*	*Accessed from a Different Package*
Public	✓	✓	✓	✓
Protected	✓	✓	✓	–
Default (no modifier)	✓	✓	–	–
Private	✓	–	–	–

5.1.1 Default Access Modifier

If we do not explicitly specify any access modifier for classes, methods, fields, etc., then by default the default access modifier is considered.

Example 13: Default access modifier.



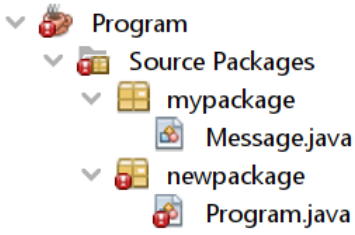

```
// Message.java
package mypackage;

class Message {
    void show(){
        System.out.println("This is a message");
    }
}
// Program.java
package mypackage;

class Program {
    public static void main(String[] args) {
        Message msg = new Message();
        msg.show();
    }
}
```

Output

```
This is a message
```

Here, the `Message` class has the default access modifier. And the class is visible to all the classes that belong to the `mypackage` package.

However, if we try to use the `Message` class in another class outside of `mypackage`, as shown in the image below, we will get a compilation error.



5.1.2 Public Access Modifier

When methods, fields, classes, and so on are declared `public`, then we can access them from anywhere. The public access modifier has no scope restriction.

Example 14: Public access modifier.

```
// Animal.java
public class Animal {
    // public field
    public int legCount;

    // public method
    public void display() {
        System.out.println("I am an animal.");
        System.out.println("I have " + legCount + " legs.");
    }
}
//Program.java
class Program {
    public static void main(String[] args) {
        // accessing the public class
        Animal animal = new Animal();

        // accessing the public variable
        animal.legCount = 4;
        // accessing the public method
        animal.display();
    }
}
```

Output

```
This is an animal.
It has 4 legs.
```

5.1.2 Private Access Modifier

We can hide the members of a class from the its outside by marking them `private`. When variables and methods are declared private, they cannot be accessed outside of the class. For example,

Example 15: Private Access Modifier.

```
//Data.java
private class Data {
    // private field
    private String name;

    private void displayName(){
        Data d = new Data();

        System.out.print(name);
    }
}
//Program.java
class Program {
    public static void main(String[] args) {
        // create an object of Data
        Data data1 = new Data();  // Error

        // access private variable and field from another class
        data1.name = "John";  // Error

    }
}
```

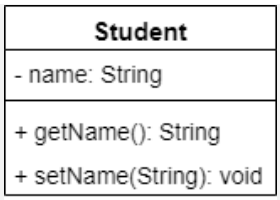
5.1.2.1 Getter and Setter Methods

What if we need to access those private variables? In this case, we can use the **getter** and **setter** methods (aka. accessors and mutators).

We use getters and setters when we want to avoid direct access to private variables, or when we want add validation logic for modifying values of data fields.

To modify a data member, we call the setter method, and to read a data member, we call the getter method.

Example 16: Using getters and setter.



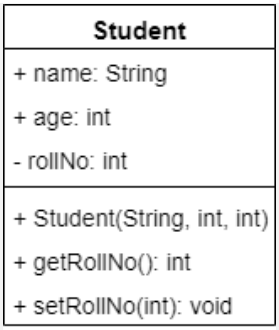

```
// Student.java
public class Student {
    private String name;

    // getter method
    String getName(){
        return name;
    }
    //setter method
    void setName(String name){
        this.name = name;
    }
}
// Program.java
class Program {
    public static void main(String[] args) {
        Student student = new Student();

        // set the name using setter
        student.setName("John");

        // get name and age using getter
        System.out.println("Hello " + student.getName());
    }
}
```

Output:

```
Hello John
```

Let's take a look at another example that shows how to use encapsulation to implement data hiding and apply additional validation before modifying the values of an object's fields.

Example 17: Data hiding and conditional logic for modifying the values of an object's fields.




```
// Student.java
public class Student {
    public String name;
    public int age;
    private int rollNo;

    public Student(String name, int age, int rollNo){
        this.name = name;
        this.age = age;
        this.rollNo = rollNo;
    }
    int getRollNo(){
        return rollNo;
    }
    void setRollNo(int rollNo){
        if(rollNo > 50)
            System.out.println("Invalid roll no. Please use " +
                "correct roll number");
        else
            this.rollNo = rollNo;
    }
    void showInfo(){
        System.out.println("Student Details:" + name + " " + rollNo);
    }
}
// Program.java
class Program {
    public static void main(String[] args) {
        Student student = new Student("Lucy", 20, 15);

        student.setRollNo(60);
    }
}
```

Output

```
Invalid roll no. Please use correct roll number
```

5.1.2.2 Using Private Members in a Parent Class

You would not always want all members of a parent class to be inherited by its child class. In this case, you can make those members private so that they will not be available to the child class.

Example 18: This example show that a child class does not inherit the private members from its parent classes.

```
// Animal.java
public class Animal {
    public String type;
    private int legCount;

    public void displayName() {
        System.out.println("This is a " + type);
    }
    private void displayLegs() {
        System.out.println("It has " + legCount + " legs.");
    }
}
// Dog.java
public class Dog extends Animal {

}
// Program.java
public class Program {
    public static void main(String[] args) {
        Dog dog1 = new Dog();
        dog1.type = "Dog";
        dog1.legCount = 4;   // Error

        dog1.displayName();
        dog1.displayLegs();  // Error
    }
}
```

5.1.3 Protected Access Modifier

When members are declared `protected`, we can access them within the same package as well as from child classes.

Example 19: Protected Access Modifier

```
// Animal.java
public class Animal {
    protected void display() {
        System.out.println("This is an animal");
    }
}
// Dog.java
public class Dog extends Animal {
    public static void main(String[] args) {
        Dog dog = new Dog();

        dog.display();
    }
}
```

Output:

```
This is an animal
```

Note: For class diagrams:

- `+` is used for public member
- `-` is used for private member
- `#` is used for protect member
- `static` variable is underlined

6. Abstraction

Abstraction is the process of hiding certain details and showing only essential information to the user. This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea.

A practical example of abstraction can be motorbike brakes. We know what brake does. When we apply the brake, the motorbike will stop. However, the working of the brake is kept hidden from us. The major advantage of hiding the working of the brake is that now the manufacturer can implement brake differently for different motorbikes, however, what brake does will be the same.

In Java, abstraction can be achieved with either **abstract classes** or **interfaces**.

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class). An abstract class can have both abstract and regular methods.

Abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the child class.

The `abstract` keyword is used for classes and methods. For example:

```
abstract class AbstractClass {
    // abstract method
    abstract void method1();

    // regular method
    void method2() {
        System.out.println("This is regular method");
    }
}
```

Note: abstract classes might or might not have abstract methods. Also, if a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error.

Though abstract classes cannot be instantiated, we can create child classes from it. We can then access members of the abstract class using the object of the child class.

If the abstract class includes any abstract method, then all the child classes inherited from the abstract class must override/implement the abstract method.

Example 20: Using abstract class and abstract methods.

```
// AbstractClass.java
abstract class AbstractClass {
    abstract void method1();  // abstract method
    void method2(){  // regular method
        System.out.println("This is a regular method");
    }
}
// ChildClass.java
public class ChildClass extends AbstractClass {
    // implement abstract method
    public void method1(){
        System.out.println("Abstract method is implemented");
    }
}
// Program.java
public class Program {
    public static void main(String[] args){
        ChildClass obj = new ChildClass();

        obj.method1();
        obj.method2();
    }
}
```

Output:

```
Abstract method is implemented
This is a regular method
```

Example 21: Another example of using abstract class and abstract methods.

```
// Vehicle.java
public abstract class Vehicle {
    public abstract String GetInfo();
}
// Car.java
public class Car extends Vehicle {
    private String company = "Toyota";
    private String model = "Camry";
    private int year = 2020;

    public String GetInfo(){
        return "Car Information: " + company + " " + model + " " + year;
    }
}
// Bicycle.java
public class Bicycle extends Vehicle {
    private String company = "Toyota";
    private String model = "BMW";
    private int year = 2021;

    public String GetInfo(){
        return "Bicycle Information: " + company + " " + model + " " + year;
    }
}
// Program.java
public class Program {
    public static void main(String[] args){
        Bicycle bike1 = new Bicycle();
        System.out.println(bike1.GetInfo());

        Car car1 = new Car();
        System.out.println(car1.GetInfo());
    }
}
```

Output

```
Bicycle Information: Toyota BMW 2021
Car Information: Toyota Camry 2020
```

In the above example, we have an abstract class called `Vehicle`. It has one abstract method, called `GetInfo`(). As it is an abstract method, this has to be implemented by the classes that inherit the abstract class. Our `Bicycle` and `Car` classes inherit the `Vehicle` abstract class, so they have to implement the abstract method `GetInfo`(). If you take a look at the implementation of these methods in the two classes, you will see that the implementation is different, which is due to abstraction.

An abstract class can have constructors like the regular class. But if the abstract classes cannot be instantiated, why do they have constructors?

A constructor of the abstract class does not actually construct the object, it can be used to initialize fields. We also know that abstract classes may contain fields and sometimes they need to be initialized somehow by using the constructor.

Like a regular class, the constructors of the abstract class are always invoked when a concrete child class is instantiated. And we can access the constructor of an abstract class from the subclass using the `super` keyword.

Example 22: A program to illustrate Concept of Constructors in Abstract Class.

```
// Content.java
public abstract class Content {
    int num;

    // Constructor of abstract class
    public Content(int num){
        this.num = num;
    }
    // Abstract method of abstract class
    abstract int multiply(int val);
}
// ChildClass.java
public class ChildClass extends Content {

    // Constructor of Child class
    ChildClass(){
        super(10); // Use super keyword refers to parent class
    }
    public int multiply(int val){
        return num * val;
    }
}
// Program.java
public class Program {
    public static void main(String[] args){
        ChildClass obj = new ChildClass();

        // Calling abstract method of abstract class
        System.out.println(obj.multiply(5));
    }
}
```

Output

```
50
```

7. Interface

An interface is a **fully** abstract class. It includes a group of abstract methods. We use the `interface` keyword to create an interface in Java.

```
interface Animal {
    // interface method does not have a body
    void animalSound();
    void run();
}
```

A class is a blueprint, which means it contains the members and methods that the instantiated objects will have. An interface can also be categorized as a blueprint, but unlike a class, an interface does not have any method implementation. Interfaces are more like a guideline for classes that inherit the interface.

The main features of interfaces in Java are as follows:

- Interfaces cannot have any concrete method.
- Like abstract classes, we cannot create objects of interfaces.
- A class can implements multiple interfaces.
- An Interface can extend other interfaces. If any class implements an interface, it must provide implementations for all the abstract methods of the interface and also all the abstract methods of the other interfaces that this interface has extended.
- The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members. In other words, Interface fields are `public`, `static` and `final` by default, and the methods are `public` and `abstract`.

Example 23: Using interfaces. To inherit an interface, use `implements` keyword. The body of the interface method is implemented by its child classes.

```
// BankAccount.java
public interface BankAccount {
    String name = "ABC Bank";

    void debit(double amount);
    void credit(double amount);
}
// Acount.java
public class Account implements BankAccount {
    // implementation of abstract methods
    public void debit(double amount){
        System.out.println(amount + "$ has been debited from your account!");
    }
    public void credit(double amount){
        System.out.println(amount + "$ has been credited to your account!");
    }
    public void displayBankName(){
        // name = "AAA"; //error: cannot assign a value to final variable "name"
        System.out.println(name);
    }
}
```

```java
// Program.java
public class Program {
    public static void main(String[] args){
        Account account1 = new Account();

        account1.displayBankName();
        account1.debit(1000);
        account1.credit(2000);
    }
}
```

Output

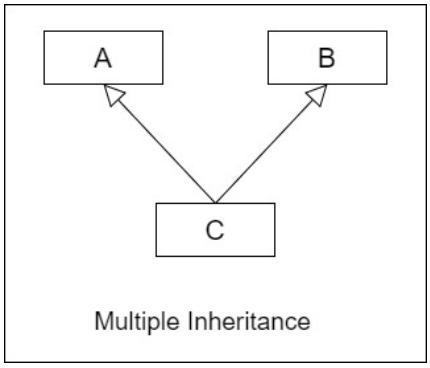
```
ABC Bank
1000$ has been debited from your account!
2000$ has been credited to your account!
```

In the interface, the methods are more like guidelines or requirements for the classes that will inherit this interface. If any class inherits this interface, then the class has to implement the method body. If the class does not implement any of the methods of the interface, the compiler will throw an error that the class has not implemented all the methods of the interface. This is a great use of the OOP concept of inheritance.

In the above example, the `Account` class has inherited the `BankAccount` interface and this is why the two methods, `debit`() and `credit`(), have to be implemented.

7.1 Multiple Inheritance (Through Interfaces)

Java does not support "multiple inheritance" (a class can only inherit from one parent class). However, it can be achieved with interfaces, because the class can inherit multiple interfaces. In the image below, Class C inherits the interface A and B.



Example 24: Multiple Inheritance (Through Interfaces)

```
// Parent.java
public class Parent {
    public void DisplayParent(){
        System.out.println("Here are my parents");
    }
}
// Person.java
interface Person {
    public void DisplayPerson();
}
// Child.java
class Child extends Parent implements Person {
    public void DisplayPerson(){   // implements the Person interface's method
        System.out.println("Here is person");
    }
    public void DisplayChild(){
        System.out.println("I am a child");
    }
}
// Program.java
public class Program {
    public static void main(String[] args){
        Child child = new Child();

        child.DisplayChild();
        child.DisplayParent();
        child.DisplayPerson();
    }
}
```

Output

```
I am a child
Here are my parents
Here is person
```

7.2 Default Methods in Java Interfaces

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods. Default methods are inherited like ordinary methods.

To declare default methods inside interfaces, we use the `default` keyword. For example,

```
public default void getSides() {
    // body
}
```

Example 25: Using default methods.

```
// Polygon.java
interface Polygon {
    void getArea();

    // default method
    default void getSides() {
        System.out.println("I can get sides of a polygon");
    }
}
// Square.java
class Square implements Polygon {
    public void getArea() {
        int length = 5;
        int area = length * length;

        System.out.println("The area of the square is " + area);
    }
}
// Rectangle.java
class Rectangle implements Polygon {
    public void getArea() {
        int length = 6;
        int breadth = 5;
        int area = length * breadth;

        System.out.println("The area of the rectangle is " + area);
    }
    // overrides the default method
    public void getSides() {
        System.out.println("I have 4 sides");
    }
}
// Program.java
public class Program {
    public static void main(String[] args){
        // create an object of Rectangle
        Rectangle r1 = new Rectangle();
        r1.getArea();
        r1.getSides();

        // create an object of Square
        Square s1 = new Square();
        s1.getArea();
        s1.getSides();
    }
}
```

Output:

```
The area of the rectangle is 30
I have 4 sides
The area of the square is 25
I can get sides of a polygon
```

In the above example, the `Rectangle` class overrides the `getSides()` method, while the `Square` class does not. Now, while calling the `getSides()` method using the Rectangle object, the overridden method is called. However, in the case of the Square object, the default method is called.

8. Polymorphism

Polymorphism is an important concept of object-oriented programming. It simply means more than one form. There are two types of polymorphism: Static Polymorphism and Dynamic Polymorphism.

8.1. Static Polymorphism

Static polymorphism means defining multiple methods with the same name but with different parameters. Using static polymorphism, we can perform different tasks with the same method name by passing different parameters.

Static Polymorphism:

- is also known as Compile Time Polymorphism
- the role of a method is determined at compilation time.
- is achieved by **method overloading**.

Example 26: Method overloading.

```
// Calculator.java
public class Calculator {
    public int Add(int a, int b){
        return a + b;
    }
    public double Add(double a, double b){
        return a + b;
    }
    public int Add(int a, int b, int c){
        return a + b + c;
    }
}
```

```java
// Program.java
public class Program {
    public static void main(String[] args){
        Calculator calculator = new Calculator();

        System.out.println(calculator.Add(2, 3));
        System.out.println(calculator.Add(1.7, 4.8));
        System.out.println(calculator.Add(2, 3, 4));
    }
}
```

Output

```
5
6.5
9
```

Here, we can see that we have three methods with the same name, `Add`. As the parameters of those methods are different, methods are allowed to have the same name by the compiler. This is called **method overloading** and this is Static Polymorphism.

8.2. Dynamic Polymorphism

Dynamic polymorphism means overriding a parent class method in the child class. Using dynamic polymorphism, we can override a parent class method in the child class by creating a method with the same name and parameters to perform a different task.

Dynamic Polymorphism:

- is also known as Runtime Polymorphism
- the role of a method is determined at runtime.
- is achieved by **method overriding**.

Method overriding is also like method overloading but it performed with parent class and child class.

Example 27: Method overriding

```java
// Language.java
public class Language {
    public void displayInfo() {
        System.out.println("Common English Language");
    }
}
// Java.java
class Java extends Language {
    public void displayInfo() {
        System.out.println("Java Programming Language");
    }
}
```

```java
// Program.java
public class Program {
    public static void main(String[] args){
        // create an object of Java class
        Java j1 = new Java();
        j1.displayInfo();

        // create an object of Language class
        Language l1 = new Language();
        l1.displayInfo();
    }
}
```

Output

```
Java Programming Language
Common English Language
```

In the above example, the use of `displayInfo()` is to print the information. However, it is printing different information in Language and Java. Hence it is an example for polymorphism.

8.3 Preventing Extending and Overriding

Neither a final class nor a final method can be extended. A final data field is a constant.

You may occasionally want to prevent classes from being extended. In such cases, use the final modifier to indicate a class is final and cannot be a parent class. For example, the `Math` class and `String`, are also final classes.

For example, the following class `MyClass` is final and cannot be extended:

```java
public final class MyClass {

}
```

You also can define a method to be final; a final method cannot be overridden by its child classes. For example, the following method `myMethod` is `final` and cannot be overridden:

```java
public class Test {
    public final void myMethod(){

    }
}
```

Exercises

1. Design a class named `Person` and its two child classes named `Student` and `Employee`. Make `FacultyMember` and `Staff` a subclasse of `Employee`. A person has a name, address, phone number, and e-mail address. A student has a class status (freshman, sophomore, junior, or senior). Define the status as a constant. An employee has an office and salary, and date hired. Use the `MyDate` class (contains `day`, `month`, and `year`) to create an object for date hired. A faculty member has office hours and a position. A staff has a title. Override the `toString`() method in each class to display the class name and the person's name.

   Write a test program that creates a Person, Student, Employee, Faculty, and Staff, and invokes their `toString`() methods.
2. A Student is an object in a university management System. Analyze the fields and the methods (at least three for each) that should be included in the `Student` abstract class. This `Student` abstract class also contains a method called `takeExam`() which will be implemented in the child classes `PhdStudent` and `GradStudent` in which Phd students take exam by giving their final defense presentations while the graduate students give written papers.
3. Create an `Animal` class with a `makeSound`() method. Then, create `Dog` and `Cat` classes that inherit from `Animal`. The `makeSound`() method should be overridden in each child class to make the appropriate animal sound. Finally, create a `Zoo` class that takes a collection of `Animal` objects and calls their respective `speak`() methods.
4. Create a class called `BankAccount` that encapsulates all the data and behavior necessary to manage a bank account. The class should have private variables for the account number, balance, and account holder's name. It should have methods for creating accounts, transferring money between accounts, checking the account balance, and generating reports about the bank's financial status. Make sure that the methods only allow valid transactions (e.g., you cannot withdraw more than the available balance).
5. Create a `Student` class that encapsulates all the data and behavior necessary to manage a student's academic record. The class should have private variables for the student's name, ID, classes taken, and grades. It should also have methods for adding and dropping classes, as well as calculating the student's overall GPA.
6. Create a `Person` class with fields like `name`, `age`, and `gender`. Then, create `Employee` and `Customer` classes that inherit from `Person`. The `Employee` class should have fields like `employeeID`, `jobTitle`, and `salary`, while the `Customer` class should have fields like `customerID`, `address`, and `phoneNumber`.

7. Create a `Library` class that abstracts the concept of a library. The class should have methods for adding and removing books, checking out and returning books, and generating reports about the library's collection.
8. Create a `Game` class that models a simple game. The class should have private variables for the game state and public methods for `starting`(), `pausing`(), and `stopping`() the game. Then, create two child classes called `SinglePlayerGame` and `MultiplayerGame` that inherit from the `Game` class. The `SinglePlayerGame` class should have a method for playing against the computer, while the `MultiplayerGame` class should have a method for playing against other players. Finally, create a `GameLauncher` class that takes a game object and launches the game.

Reference

[1] Y. Daniel Liang. 'Introduction to Java Programming', 11e – 2019

[2] https://www.programiz.com/java-programming