

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.

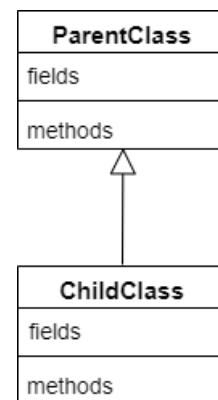
In inheritance, there are two kinds of classes:

- **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 child class
class ChildClass extends ParentClass // inheritance
{
    // fields
    // methods
}
```



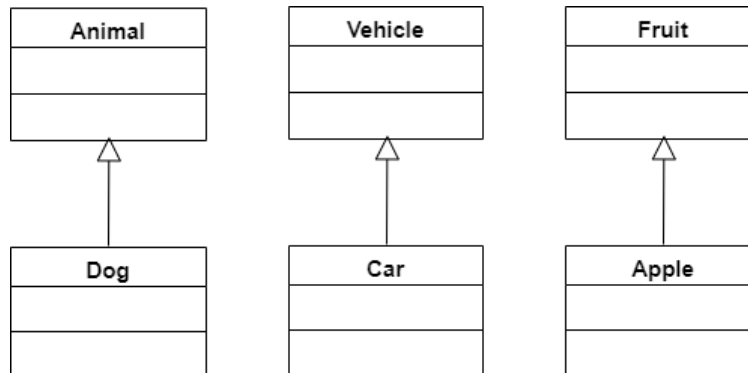
Class Diagram

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**



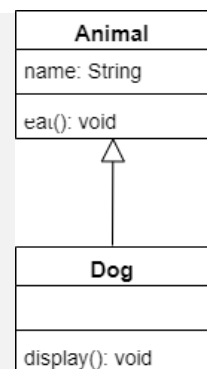
Example 01: Using Inheritance

```
class Animal{
    String name;

    void eat(){
        System.out.println("The dog 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

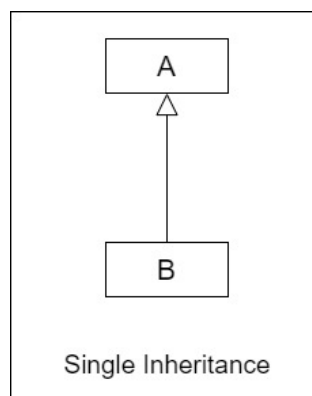
2. Types of Inheritances

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 02: Single Inheritance

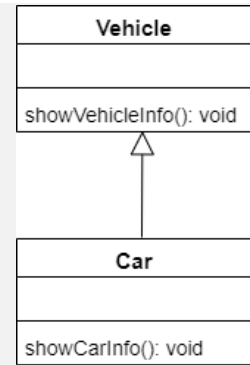
```
// Parent class
class Vehicle {
    void showVehicleInfo(){
        System.out.println("This is a vehicle");
    }
}

// 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();
    }
}
```

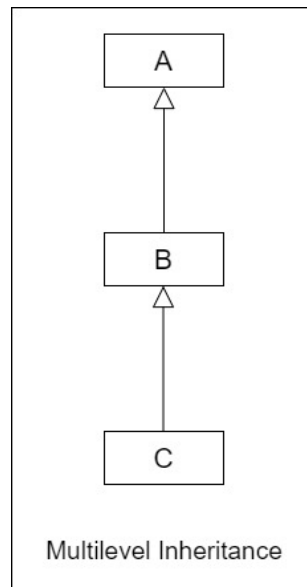


Output:

```
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.

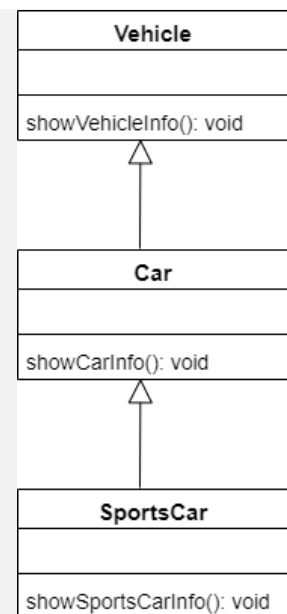


Example 03: Multilevel Inheritance

```
// 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");
    }
}

// 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:

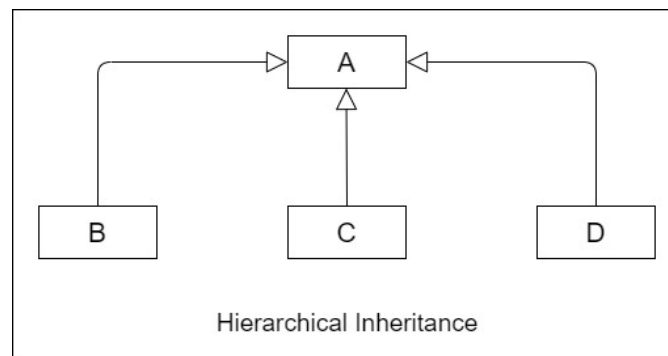
```

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 04: Hierarchical Inheritance

```
// 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);
    }
}

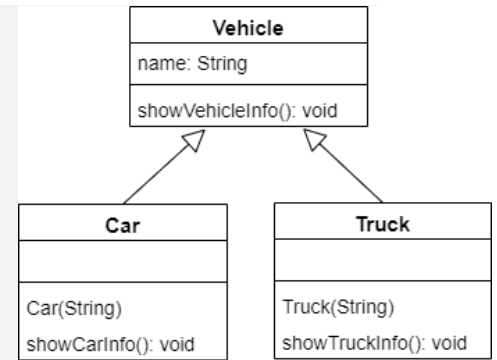
// Child class 2
class Truck extends Vehicle {
    Truck(String name){
        this.name = name;
    }
    void showTruckInfo(){
        System.out.println("Truck's name is " + name);
    }
}

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
```

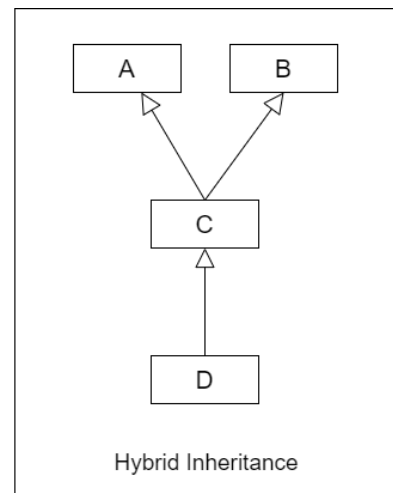
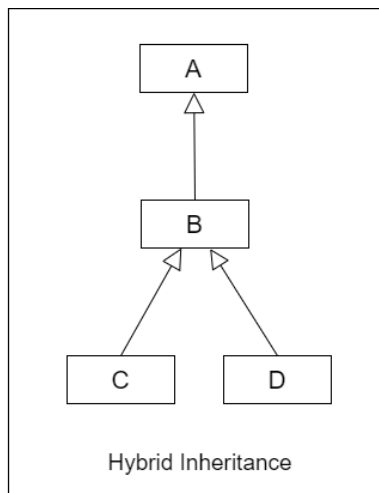
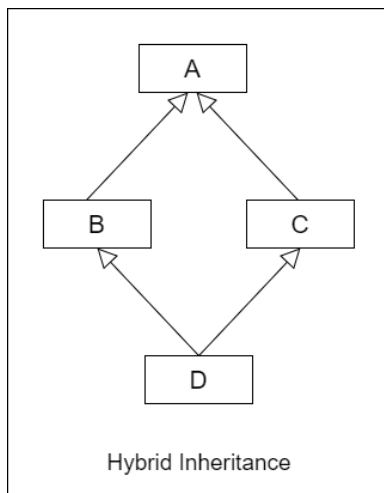
2.4 Multiple Inheritance

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 chapter.

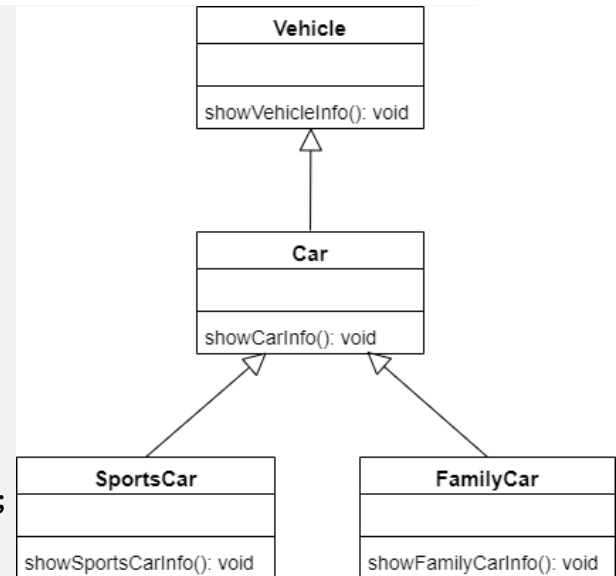
2.5 Hybrid Inheritance

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



Example 05: 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 superclass that is overridden in the subclass.
- To access fields of the superclass if both superclass and subclass have fields with the same name.
- To explicitly call parent class's constructor from the child class's constructor.

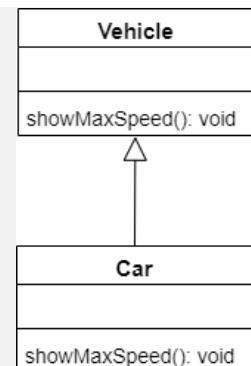
3.1 Method Overriding

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 06: 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:

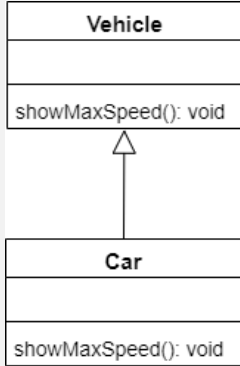
Max speed is 200 km/hour

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 can use the `super` keyword to do it.

Example 07: 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();
    }
}
```



```
classDiagram
    class Vehicle {
        showMaxSpeed() void
    }
    class Car {
        showMaxSpeed() void
    }
    Vehicle <|-- Car
```

Output:

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

Example 08: Another example of using the **super** keyword to access to parent class's members.

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

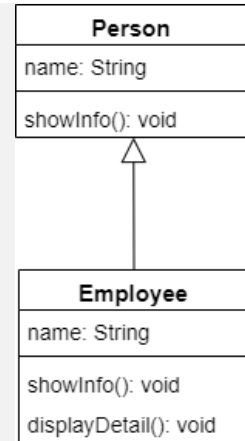
    void showInfo(){
        System.out.println("This is a person");
    }
}

// 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 09: 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 10: 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.

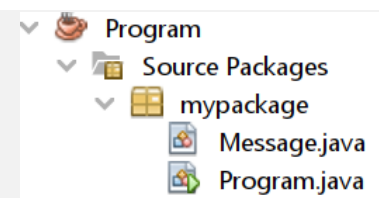
5.1.1 Default Access Modifier

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

Example 11: Default access modifier.

```
// Message.java
package mypackage;

class Message {
    void show(){
        System.out.println("This is a message");
    }
}
```



```
// Program.java
package mypackage;

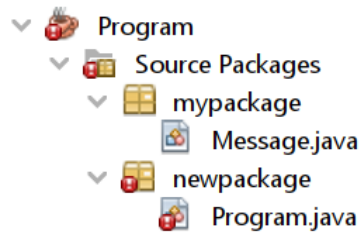
class Program {
    public static void main(String[] args) {
        Message msg = new Message();
        msg.show();
    }
}
```

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 12: 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 13: Private Access Modifier.

```
private class Data {
    // private field
    private String 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.

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 14: 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;
    }
}
```

Student
- name: String
+ getName(): String
+ setName(String): void

```
// 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 15: 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);
    }
}
```

Student
+ name: String
+ age: int
- rollNo: int
+ Student(String, int, int)
+ getRollNo(): int
+ setRollNo(int): void

```
// 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 16: 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 17: 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

Reference

- [1] Y. Daniel Liang. 'Introduction to Java Programming', 11e – 2019
- [2] <https://www.programiz.com/java-programming>