

# Java Basic for Tester

*Java OOP (II)*



- *Java Inheritance*
- *Java Method Overriding*
- *Java super Keyword*
- *Abstract Class & Method*
- *Java Interfaces*
- *Java Polymorphism*
- *Java Encapsulation*

Inheritance is one of the key features of OOP (Object-oriented Programming) that allows us to define a new class from an existing class.

```
class Animal
{
    // eat() method
    // sleep() method
}
class Dog extends Animal
{
    // bark() method
}
```

Inheritance is one of the key features of OOP (Object-oriented Programming) that allows us to define a new class from an existing class.

```
class Animal
{
    // eat() method
    // sleep() method
}
class Dog extends Animal
{
    // bark() method
}
```

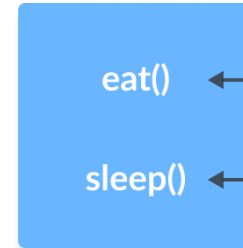
In Java, we use the **extends** keyword to inherit from a class. Here, we have inherited the **Dog** class from the **Animal** class.

```
1 package OOP1.JavaInheritance;
2
3 public class Animal {
4     public void eat() {
5         System.out.println("I can eat");
6     }
7
8     public void sleep() {
9         System.out.println("I can sleep");
10    }
11 }
12
13 class Dog extends Animal {
14     public void bark() {
15         System.out.println("I can bark");
16     }
17 }
18
19 class Main {
20     public static void main(String[] args) {
21
22         Dog dog1 = new Dog();
23         dog1.eat();
24         dog1.sleep();
25         dog1.bark();
26     }
27 }
```

```
Run: Main x
"C:\Program Files\Java\jdk
I can eat
I can sleep
I can bark

Process finished with exit
```

Animal (superclass)



Dog (subclass)

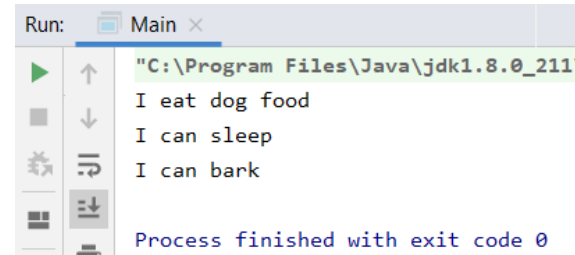


Mainclass

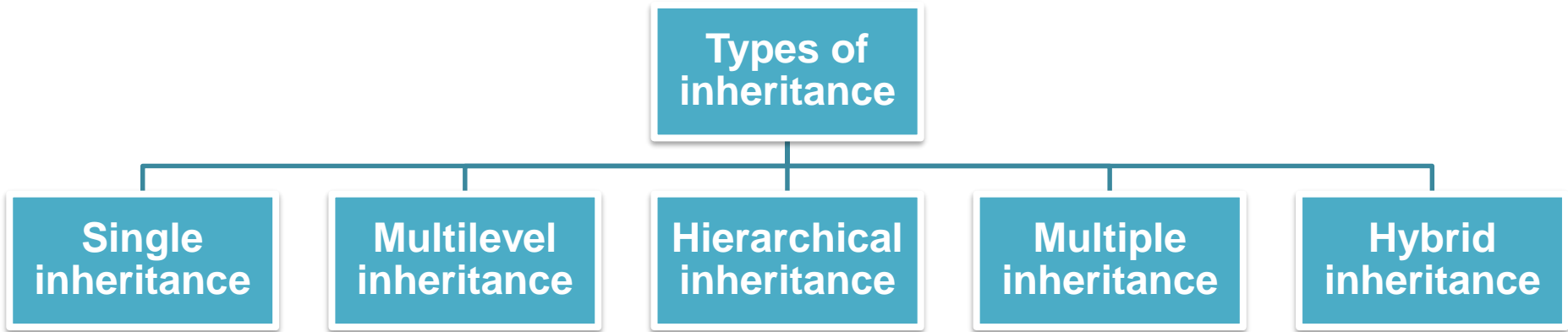


## Java Method overriding

```
1 package OOP1.JavaInheritance;
2
3 public class Animal {
4     public void eat() {
5         System.out.println("I can eat");
6     }
7
8     public void sleep() {
9         System.out.println("I can sleep");
10    }
11 }
12
13 class Dog extends Animal {
14     @Override
15     public void eat() {
16         System.out.println("I eat dog food");
17     }
18     public void bark() {
19         System.out.println("I can bark");
20     }
21 }
22
23 class Main {
24     public static void main(String[] args) {
25
26         Dog dog1 = new Dog();
27         dog1.eat();
28         dog1.sleep();
29         dog1.bark();
30     }
31 }
```



```
Run: Main x
"C:\Program Files\Java\jdk1.8.0_211"
I eat dog food
I can sleep
I can bark
Process finished with exit code 0
```



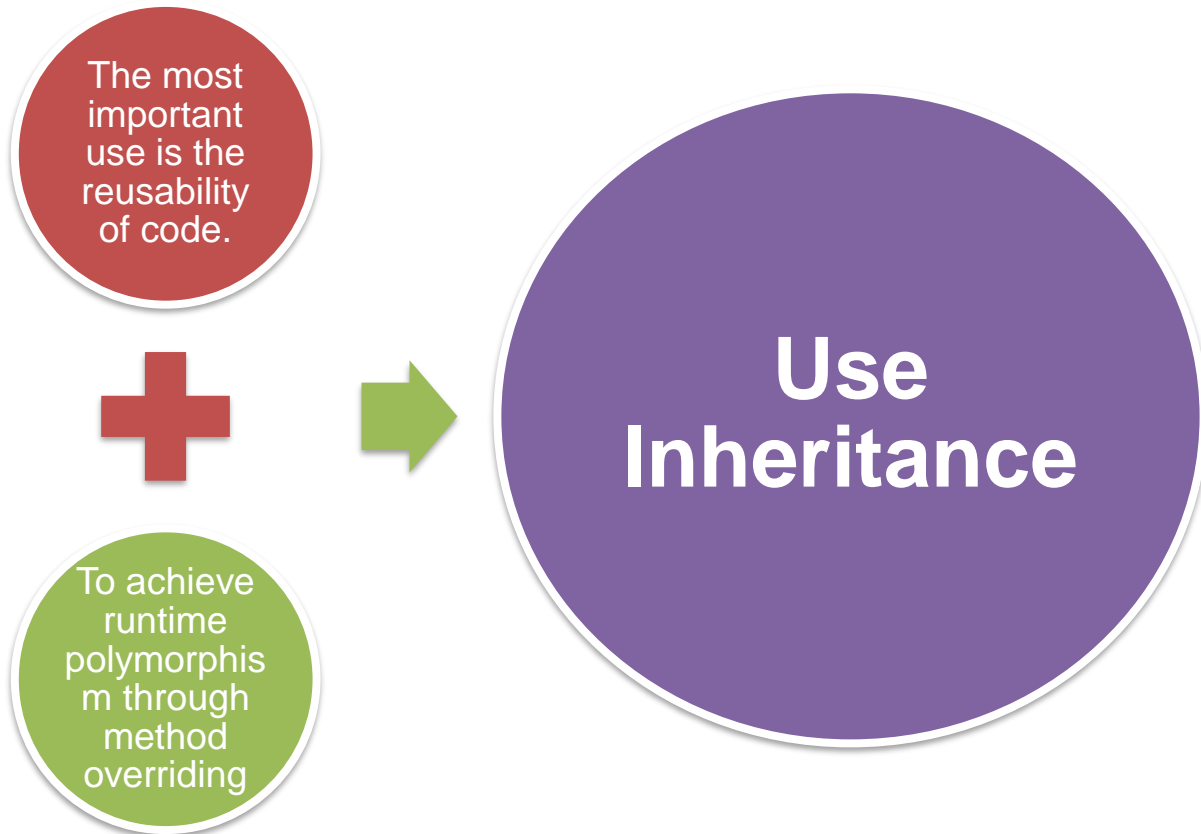
**Single inheritance** - Class B extends from class A only.

**Multilevel inheritance** - Class B extends from class A; then class C extends from class B.

**Hierarchical inheritance** - Class A acts as the superclass for classes B, C, and D.

**Multiple inheritance** - Class C extends from interfaces A and B.

**Hybrid inheritance** - Mix of two or more types of inheritance.





# Java Method Overriding

```
1 package OOP1.JavaMethodOverriding;
2
3 public class Animal {
4     public void displayInfo() {
5         System.out.println("I am an animal.");
6     }
7 }
8
9 class Dog extends Animal {
10     @Override
11     public void displayInfo() {
12         System.out.println("I am a dog.");
13     }
14 }
15
16 class Main {
17     public static void main(String[] args) {
18         Dog d1 = new Dog();
19         d1.displayInfo();
20     }
21 }
```

Run: Main (1) x

```
"C:\Program Files\Java\jdk1.8.0_21
I am a dog.
Process finished with exit code 0
```

Animal (superclass)

displayInfo()

Mainclass

d1.displayInfo()

Dog (subclass)

displayInfo()

## Rule 1

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.

## Rule 2

- We cannot override the method declared as final and static.

## Rule 3

- We should always override abstract methods of the superclass.

The **super** keyword in Java is used in subclasses to access superclass members (**attributes**, **constructors** and **methods**).

## Uses of super keyword

1

- To call methods of the superclass that is overridden in the subclass.

2

- To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.

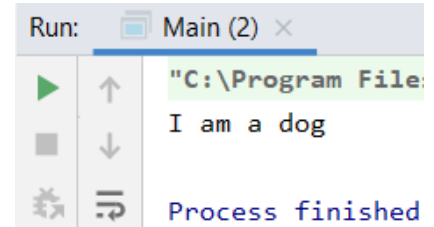
3

- To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

## Access Overridden Methods of the superclass

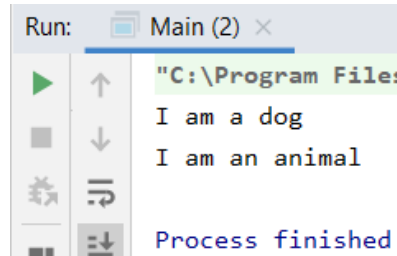
If methods with the same name are defined in both superclass and subclass, the method in the subclass overrides the method in the superclass. This is called method overriding.

```
1 package OOP1.JavaSuper;
2
3 public class Animal {
4     // overridden method
5     public void display(){
6         System.out.println("I am an animal");
7     }
8 }
9 class Dog extends Animal {
10     // overriding method
11     @Override
12     public void display(){
13         System.out.println("I am a dog");
14     }
15     public void printMessage(){
16         display();
17     }
18 }
19 class Main {
20     public static void main(String[] args) {
21         Dog dog1 = new Dog();
22         dog1.printMessage();
23     }
24 }
```

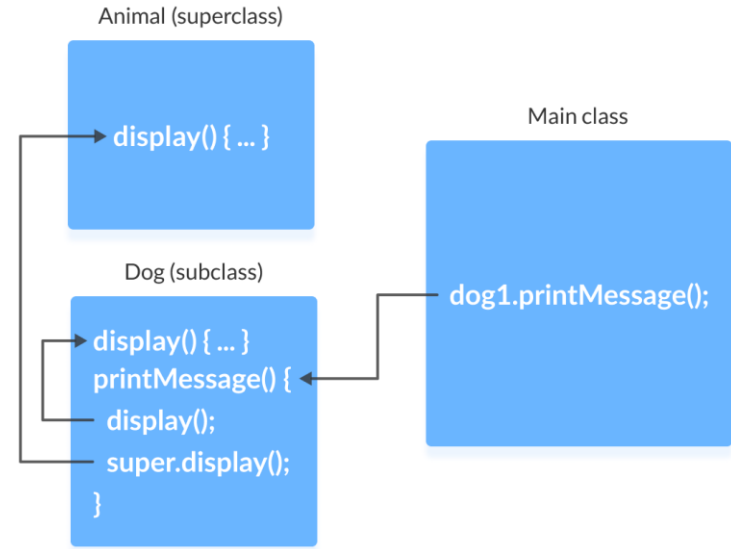


## super to Call Superclass Method

```
1 package OOP1.JavaSuper;
2
3 public class Animal {
4     // overridden method
5     public void display(){
6         System.out.println("I am an animal");
7     }
8 }
9 class Dog extends Animal {
10     // overriding method
11     @Override
12     public void display(){
13         System.out.println("I am a dog");
14     }
15     public void printMessage(){
16         // this calls overriding method
17         display();
18
19         // this calls overridden method
20         super.display();
21     }
22 }
23 class Main {
24     public static void main(String[] args) {
25         Dog dog1 = new Dog();
26         dog1.printMessage();
27     }
28 }
```



```
Run: Main (2) x
"C:\Program Files
I am a dog
I am an animal
Process finished
```



## Java Abstract Class

An abstract class is a class that cannot be instantiated (we cannot create objects of an abstract class). In Java, we use the **abstract** keyword to declare an abstract class.

```
abstract class Animal {  
    //attributes and methods  
}
```

## Java Abstract Method

We use the same keyword `abstract` to create abstract methods. An abstract method is declared without an implementation.

```
abstract void makeSound();
```

An abstract class can contain both abstract and non-abstract methods.

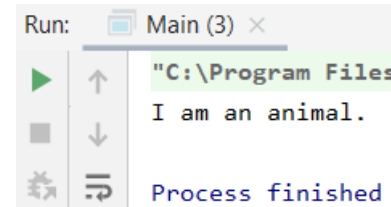
```
1 package OOP1.JavaAbstractClass;
2
3 abstract class Animal {
4     public void displayInfo() {
5         System.out.println("I am an animal.");
6     }
7     abstract void makeSound();
8 }
```



## Inheritance of Abstract Class

An abstract class cannot be instantiated. To access the members of an abstract class, we must inherit it.

```
1 package OOP1.JavaAbstractClass;  
2  
3 abstract class Animal {  
4     public void displayInfo() {  
5         System.out.println("I am an animal.");  
6     }  
7 }  
8  
9 class Dog extends Animal {  
10  
11 }  
12 class Main {  
13     public static void main(String[] args) {  
14         Dog d1 = new Dog();  
15         d1.displayInfo();  
16     }  
17 }
```



## Overriding of Abstract Methods

In Java, it is mandatory to override abstract methods of the superclass in the subclass. It is because the subclass inherits abstract methods of the superclass.

Since our subclass includes abstract methods, we need to override them.

```
1 package OOP1.JavaAbstractClass;
2
3 abstract class Animal {
4     public void eat() {
5         System.out.println("I can eat.");
6     }
7     abstract void makeSound();
8 }
9
10 class Dog extends Animal {
11     @Override
12     public void makeSound() {
13         System.out.println("Bark bark");
14     }
15 }
16 class Main {
17     public static void main(String[] args) {
18         Dog d1 = new Dog();
19         d1.makeSound();
20         d1.eat();
21     }
22 }
```

Run: Main (3) ×

```
"C:\Program File:
Bark bark
I can eat.
Process finished
```

## Why Java Abstraction?



Abstraction is an important concept of object-oriented programming. Abstraction only shows the needed information and all the unnecessary details are kept hidden. This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea.

## Key Points

We use the `abstract` keyword to create abstract classes and methods.

An abstract method doesn't have any implementation (method body).

A class containing abstract methods should also be abstract.

We cannot create objects of an abstract class.

To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.

A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.

We can access the static attributes and methods of an abstract class using the reference of the abstract class.

In Java, an interface defines a set of specifications that other classes must implement.

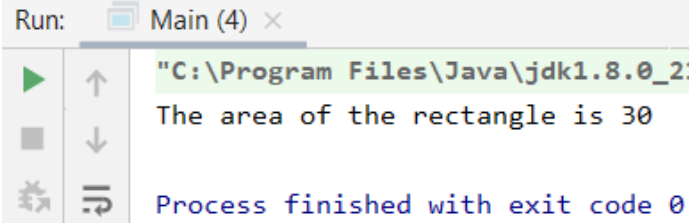
```
1 package OOP1.JavaInterface;
2
3 public interface Polygon {
4     public void getArea();
5 }
```

An interface can include abstract methods and constants.

```
1 package OOP1.JavaInterface;
2
3 public interface Polygon {
4     public static final String color = "blue";
5
6     public void getArea();
7 }
```

Like abstract classes, we cannot create objects of interfaces. However, we can implement interfaces in other classes. In Java, we use the implements keyword to implement interfaces.

```
1 package OOP1.JavaInterface;
2
3 public interface Polygon {
4     void getArea(int length, int breadth);
5 }
6
7 class Rectangle implements Polygon {
8     public void getArea(int length, int breadth) {
9         System.out.println("The area of the rectangle is " + (length * breadth));
10    }
11 }
12
13 class Main {
14     public static void main(String[] args) {
15         Rectangle r1 = new Rectangle();
16         r1.getArea( length: 5, breadth: 6);
17    }
18 }
```



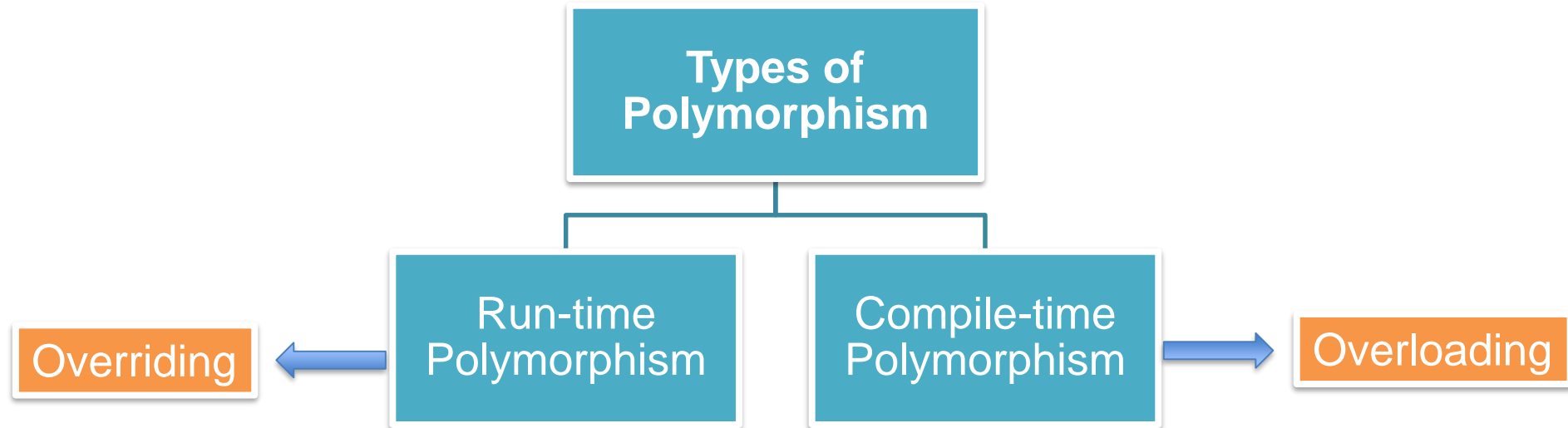
```
Run: Main (4) x
"C:\Program Files\Java\jdk1.8.0_2:
The area of the rectangle is 30
Process finished with exit code 0
```

## Why use Interfaces?



1. Interfaces provide specifications that a class (which implements it) must follow.
2. Similar to abstract classes, interfaces help us to achieve abstraction in Java.
3. Interfaces are also used to achieve multiple inheritance in Java. If a subclass is inherited from two or more classes, it's multiple inheritance.

Polymorphism is an important concept of object-oriented programming. It simply means more than one form. That is, the same entity (method or operator or object) behaves differently in different scenarios.

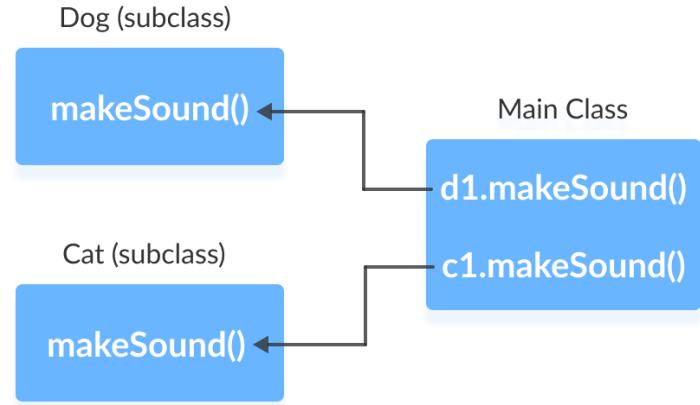
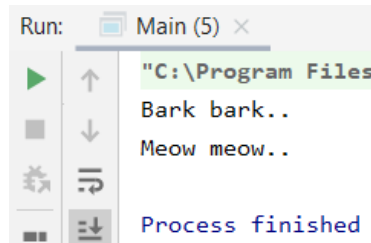




## Run-time Polymorphism

In Java, run-time polymorphism can be achieved through method overriding.

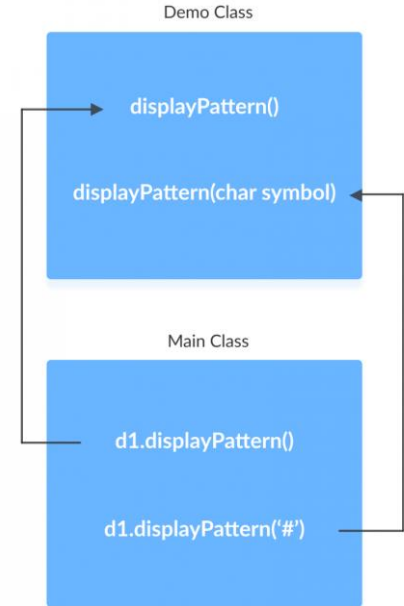
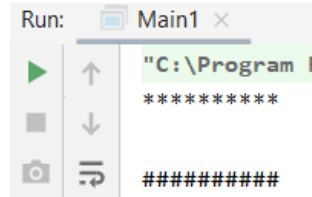
```
1 package OOP1.JavaPolymorphism;
2
3 abstract class Animal {
4     public abstract void makeSound();
5 }
6 class Dog extends Animal {
7     @Override
8     public void makeSound() {
9         System.out.println("Bark bark..");
10    }
11 }
12 class Cat extends Animal {
13     @Override
14     public void makeSound() {
15         System.out.println("Meow meow..");
16    }
17 }
18 class Main {
19     public static void main(String[] args) {
20         Dog d1 = new Dog();
21         d1.makeSound();
22         Cat c1 = new Cat();
23         c1.makeSound();
24     }
25 }
```



## Compile-time Polymorphism

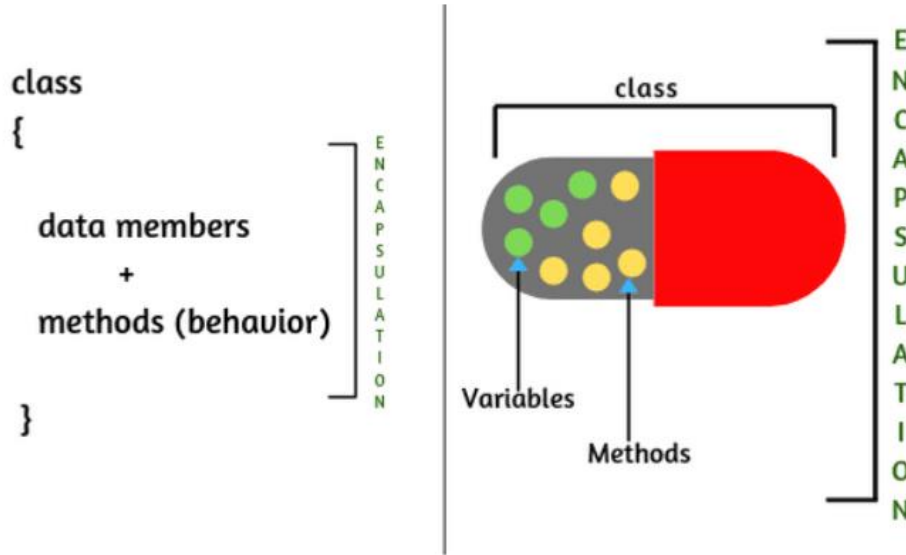
The compile-time polymorphism can be achieved through method overloading and operator overloading in Java.

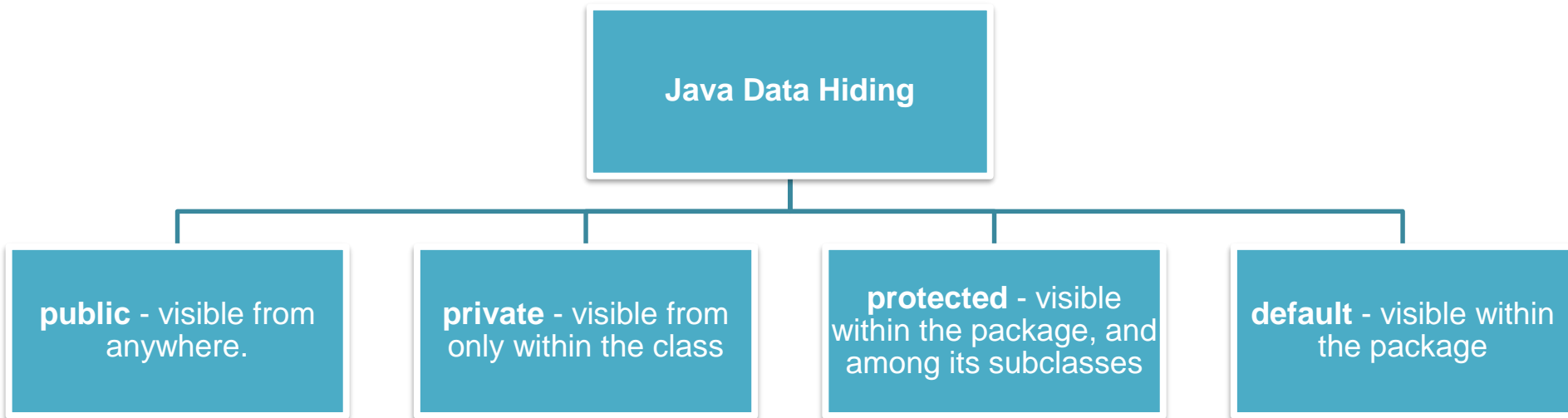
```
1 package OOP1.JavaPolymorphism;
2
3 public class Demo {
4     public void displayPattern(){
5         for(int i = 0; i < 10; i++) {
6             System.out.print("*");
7         }
8     }
9
10    public void displayPattern(char symbol) {
11        for(int i = 0; i < 10; i++) {
12            System.out.print(symbol);
13        }
14    }
15 }
16
17 class Main1 {
18     public static void main(String[] args) {
19         Demo d1 = new Demo();
20         d1.displayPattern();
21         System.out.println("\n");
22         d1.displayPattern(symbol: '#');
23     }
24 }
```



No.	Method Overloading	Method Overriding
1)	Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

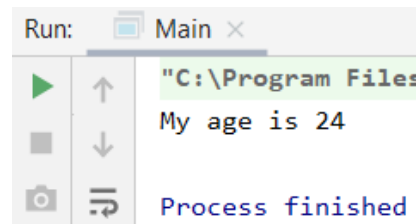
Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of fields and methods inside a single class.





# Java Encapsulation

```
1  package OOP1.JavaEncapsulation;
2
3  public class Person {
4      private int age;
5
6      public int getAge() {
7          return age;
8      }
9
10     public void setAge(int age) {
11         this.age = age;
12     }
13 }
14
15 class Main {
16     public static void main(String[] args) {
17         Person p1 = new Person();
18         p1.setAge(24);
19         System.out.println("My age is " + p1.getAge());
20     }
21 }
```



## Why Encapsulation?



1. In Java, encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.
2. It helps to control the modification of our data fields.

# Thank you

