COM2104: Advanced Programming

WK3 LECTURE: INHERITANCE AND METHOD OVERRIDING

Objectives

- Know what is inheritance and how to create class files with inherited relationship
- Know what is polymorphism and how to apply it
- Know what is method overriding and how to implement it for class files.

INHERITANCE

What is inheritance?

- Inheritance is an important notion about OOP(Object-Oriented Programming).
- In Java, inheritance means creating new classes based on existing ones.
- A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.
- Inheritance is **"IS-A"** relationship.

Important Terminologies Used in Java Inheritance

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

How to Use Inheritance in Java?

- The extends keyword is used for inheritance in Java. Using the extends keyword indicates you derive from an existing class.

Syntax :

```
class DerivedClass extends BaseClass
{
    //methods and fields
}
```

Java Common Inheritance Types

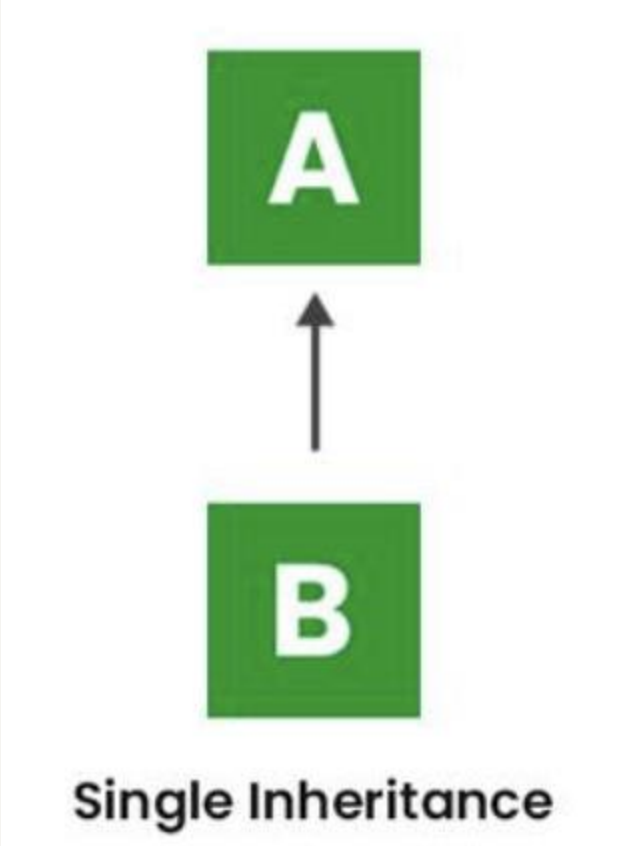
Single Inheritance

Multilevel Inheritance

Hierarchical Inheritance

https://www.geeksforgeeks.org/inheritance-in-java/

Single Inheritance



- In single inheritance, a sub-class is derived from only one super class.
- It inherits the fields and methods of a single-parent class.
- Sometimes, it is also known as **simple inheritance**.
- In the left figure, ‘A’ is a parent class and ‘B’ is a child class. The class ‘B’ inherits all the properties of the class ‘A’.

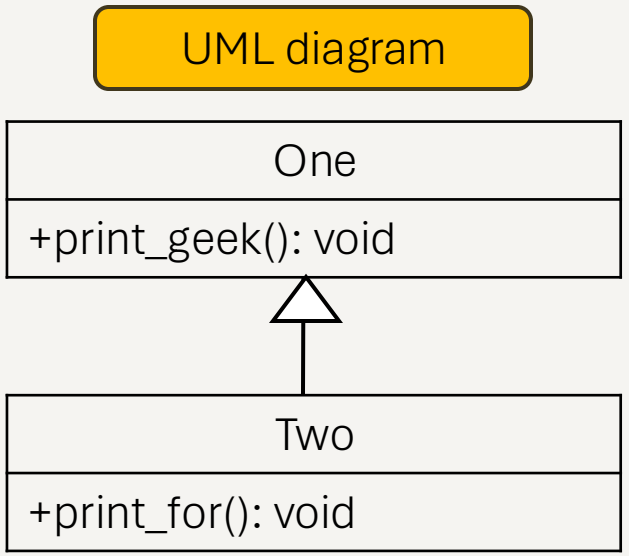
Single Inheritance: one example

```java
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class
class One {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class Two extends One {
    public void print_for() { System.out.println("for"); }
}

// Driver class
public class Main {
      // Main function
    public static void main(String[] args)
    {
        Two g = new Two();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

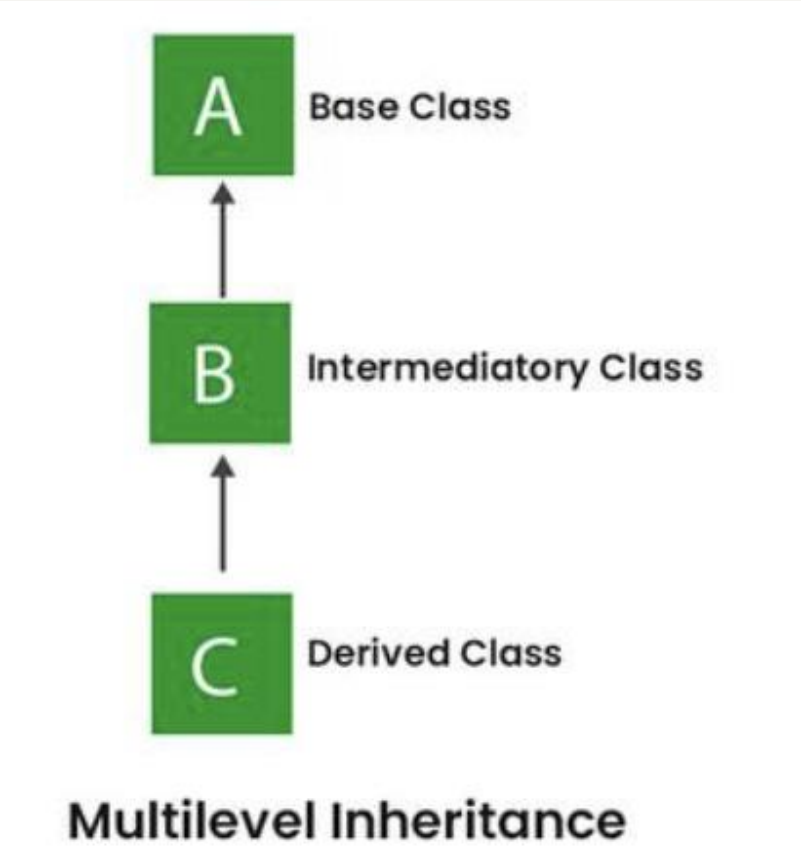
Inheritance

Output

```
Geeks
for
Geeks
```



Multilevel Inheritance



- In Multilevel Inheritance, a derived class will be inheriting an intermediatory class, the intermediatory class will be inheriting a base class.
- In the left image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class **cannot directly** access the grandparent's members.

Multilevel Inheritance: one example

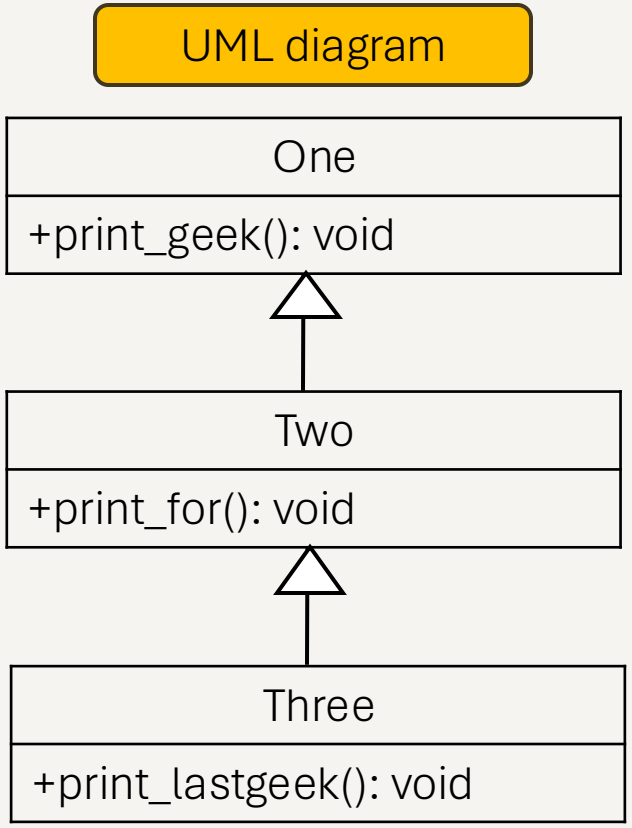
```java
// Importing required libraries
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class One
class One {
    // Method to print "Geeks"
    public void print_geek() {
        System.out.println("Geeks");
    }
}

// Child class Two inherits from class One
class Two extends One {
    // Method to print "for"
    public void print_for() {
        System.out.println("for");
    }
}

// Child class Three inherits from class Two
class Three extends Two {
    // Method to print "Geeks"
    public void print_lastgeek() {
        System.out.println("Geeks");
    }
}

// Driver class
public class Main {
    public static void main(String[] args) {
        // Creating an object of class Three
        Three g = new Three();

        // Calling method from class One
        g.print_geek();

        // Calling method from class Two
        g.print_for();

        // Calling method from class Three
        g.print_lastgeek();
    }
}
```

Output

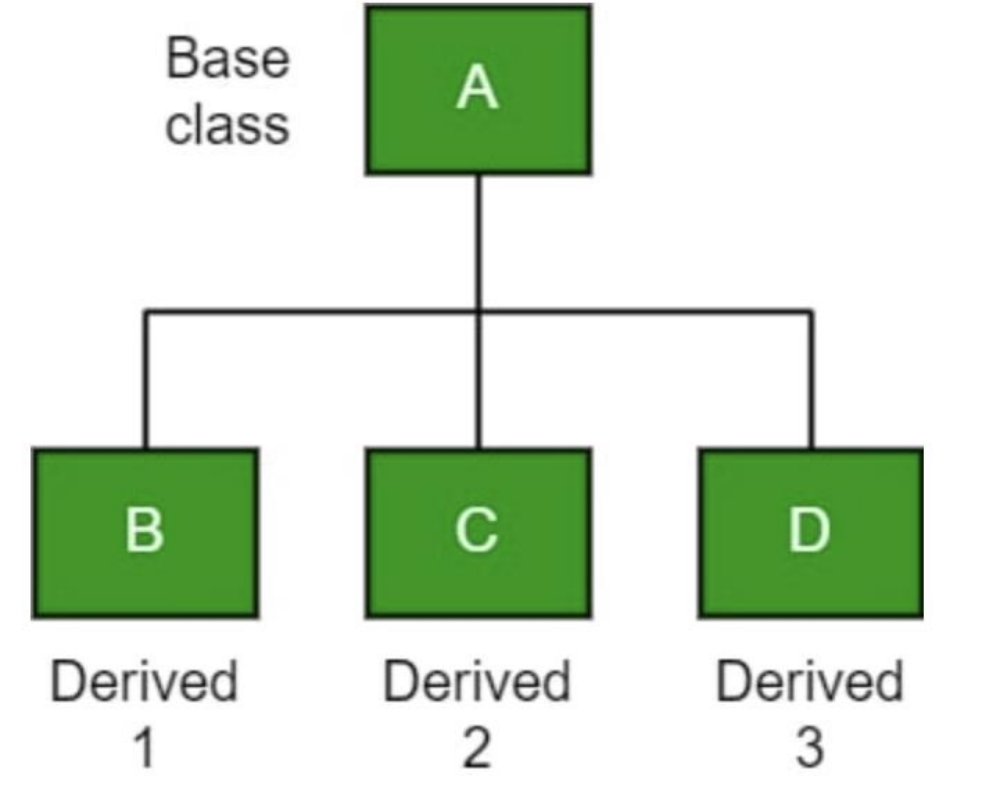
```
Geeks
for
Geeks
```

UML diagram for the above example

```java
// Importing required libraries
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class One
class One {
    // Method to print "Geeks"
    public void print_geek() {
        System.out.println("Geeks");
    }
}

// Child class Two inherits from class One
class Two extends One {
    // Method to print "for"
    public void print_for() {
        System.out.println("for");
    }
}

// Child class Three inherits from class Two
class Three extends Two {
    // Method to print "Geeks"
    public void print_lastgeek() {
        System.out.println("Geeks");
    }
}
```



Hierarchical Inheritance



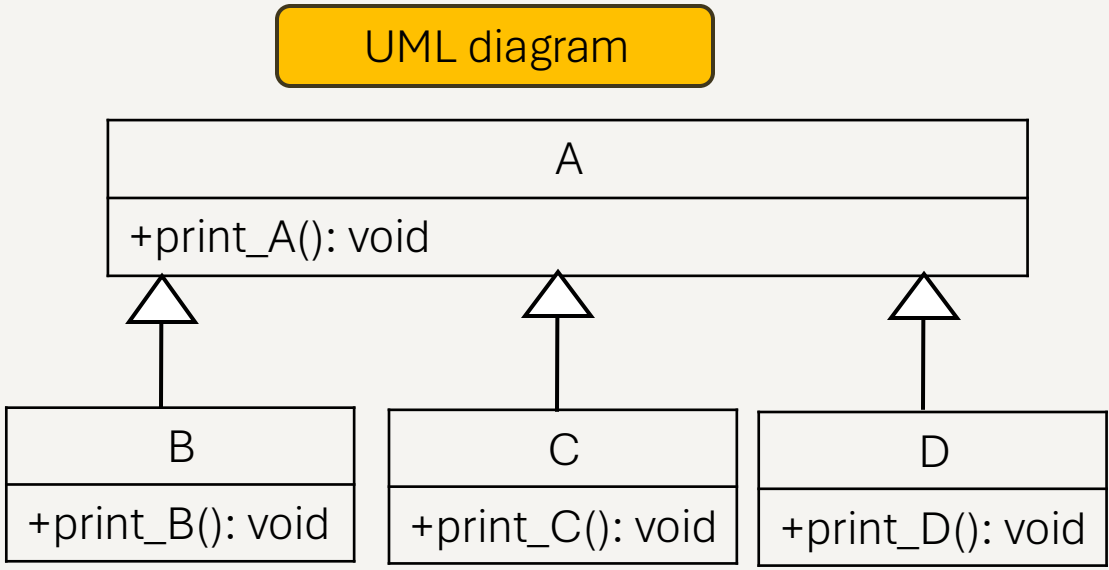
- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclasses.
- In the left image, class A serves as a base class for the derived classes B, C, and D.

Hierarchical Inheritance: One Example

```java
class A {
    public void print_A() { System.out.println("Class A"); }
}

class B extends A {
    public void print_B() { System.out.println("Class B"); }
}

class C extends A {
    public void print_C() { System.out.println("Class C"); }
}

class D extends A {
    public void print_D() { System.out.println("Class D"); }
}

// Driver Class
public class Test {
    public static void main(String[] args)
    {
        B obj_B = new B();
        obj_B.print_A();
        obj_B.print_B();

        C obj_C = new C();
        obj_C.print_A();
        obj_C.print_C();

        D obj_D = new D();
        obj_D.print_A();
        obj_D.print_D();
    }
}
```

Output

```
Class A
Class B
Class A
Class C
Class A
Class D
```



Super keyword

- The super keyword is commonly used to access parent class methods and constructors, enabling a subclass to inherit and reuse the functionality of its superclass.

- The super keyword can be used in three primary contexts:
  - To call the superclass constructor.
  - To access a method from the superclass that has been overridden in the subclass.
  - To access a field from the superclass when it is hidden by a field of the same name in the subclass.

https://www.datacamp.com/doc/java/super

Super keyword for Calling Superclass Constructor

```java
class Animal {
    Animal() {
        System.out.println("Animal constructor called");
    }
}

class Dog extends Animal {
    Dog() {
        super(); // Calls the constructor of Animal class
        System.out.println("Dog constructor called");
    }
}

public class TestSuper {
    public static void main(String[] args) {
        Dog d = new Dog(); // Output: Animal constructor called
                           //         Dog constructor called
    }
}
```

For calling superclass constructor, super() sentence is always the first statement in the subclass constructor.

In this example, the super() call in the Dog constructor invokes the constructor of the Animal class.

Super keyword for Calling Superclass Constructor

```java
class Animal {
    int leg;
    Animal(int legu){
        this.leg = legu;
    }
}

class Dog extends Animal{
    Dog(int legud){
        super(legud);
        System.out.println("Dog constructor called.");
    }
}
public class LectureE1 {

    public static void main(String[] args) {
        Dog d = new Dog(4);
        //Animal constructor gets leg of 4
        //Dog constructor called
    }
}
```

If superclass constructor has attributes, when using super, we should pass new values to these attributes.

Super keyword for Calling Superclass Constructor

```java
class Animal {
    int leg;
    Animal(int legu){
        this.leg = legu;
    }
}

class Dog extends Animal{
    String color;
    Dog(int legud, String colord){
        super(legud);
        this.color = colord;
        System.out.println("Dog constructor called.");
    }
}
public class LectureE1 {

    public static void main(String[] args) {
        Dog d = new Dog(4, "white");
        //Animal constructor gets leg of 4
        //Dog constructor called
    }
}
```

If both superclass and subclass constructors have respective attributes, when using super, we should pass new values for superclass attributes and using the common way to assign new values for subclass attributes.

Super keyword for Accessing Superclass Method

```java
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void sound() {
        super.sound(); // Calls the sound() method of Animal class
        System.out.println("Dog barks");
    }
}

public class TestSuper {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound(); // Output: Animal makes a sound
                   //         Dog barks
    }
}
```

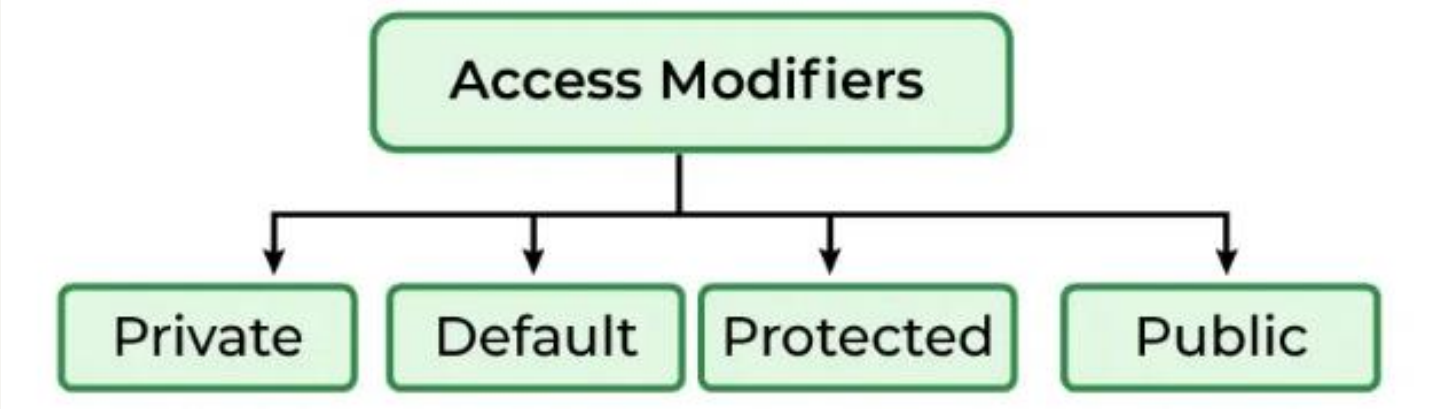
Here, the super.sound() call in the Dog class method sound() invokes the sound() method of the Animal class.

Super keyword for Accessing Superclass Field

```
class Parent {
    int number = 10;
}

class Child extends Parent {
    int number = 20;

    void showNumber() {
        System.out.println(super.number); // Accesses Parent's number
    }
}
```

Access Modifiers in Java

- There are four types of access modifiers in Java:



- https://www.geeksforgeeks.org/access-modifiers-java/

Default Modifiers

- When no access modifier is specified for a class, method, or data member, it is said to be having the **default** access modifier by default.
- The default access modifiers are accessible ***only within the same package***.

Default Modifiers

```java
// default access modifier
package p1;

// Class Geek is having
// Default access modifier
class Geek
{
    void display()
    {
        System.out.println("Hello World!");
    }
}
```

```java
// error while using class from different
// package with default modifier
package p2;
import p1.*;    // importing package p1

// This class is having
// default access modifier
class GeekNew {
    public static void main(String args[]) {

        // Accessing class Geek from package p1
        Geek o = new Geek();

        o.display();
    }
}
```

Error when **Accessing Default Modifier Class** across Packages. In the right example, the program will show the **compile-time error** when we try to access a default modifier class from a different package **p1**.

Private Access Modifier

- The **private access modifier** is specified using the keyword **private**. The methods or data members declared as private are accessible ***only within the class in which they are declared***.
- Any other class of the same package will **not be** able to access these members.

Private Access Modifier

```java
// private access modifier
package p1;

// Class A
class A {
    private void display() {
        System.out.println("GeeksforGeeks");
    }
}

// Class B
class B {
    public static void main(String args[]) {
        A obj = new A();

        // Trying to access private method
        // of another class
        obj.display();
    }
}
```

- In this example, we will create two classes A and B within the same package p1.
- We will declare a method in class A as private and try to access this method from class B. *Error will occur for accessing private method*.

Protected Access Modifier

- The **protected access modifier** is specified using the keyword **protected**.
- The methods or data members declared as protected are ***accessible within the same package or subclasses in different packages***.

Protected Access Modifier

```java
// protected access modifier
package p1;

// Class A
public class A {
    protected void display() {
        System.out.println("GeeksforGeeks");
    }
}
```

```java
// protected modifier
package p2;

// importing all classes
// in package p1
import p1.*;

// Class B is subclass of A
class B extends A {
    public static void main(String args[]) {
        B obj = new B();
        obj.display();
    }
}
```

In this example, we will create two packages, p1 and p2. Class A in p1 has a protected method display. Class B in p2 extends A and accesses the protected method through inheritance by creating an object of class B.

Public Access Modifier

- The **public access modifier** is specified using the keyword **public**.
- The public access modifier has the **widest scope** among all other access modifiers.
- Classes, methods, or data members that are declared as public are ***accessible from everywhere*** in the program. There is no restriction on the scope of public data members.

Public Access Modifier

```java
// public modifier
package p1;

public class A {

public void display() {
        System.out.println("GeeksforGeeks");
    }
}
```

```java
// public access modifier
package p2;

import p1.*;

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

        A obj = new A();
        obj.display();
    }
}
```

Here, the example shows that a **public method** is ***accessible across packages***.

Comparison OF Access Modifiers in Java

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

horizontally

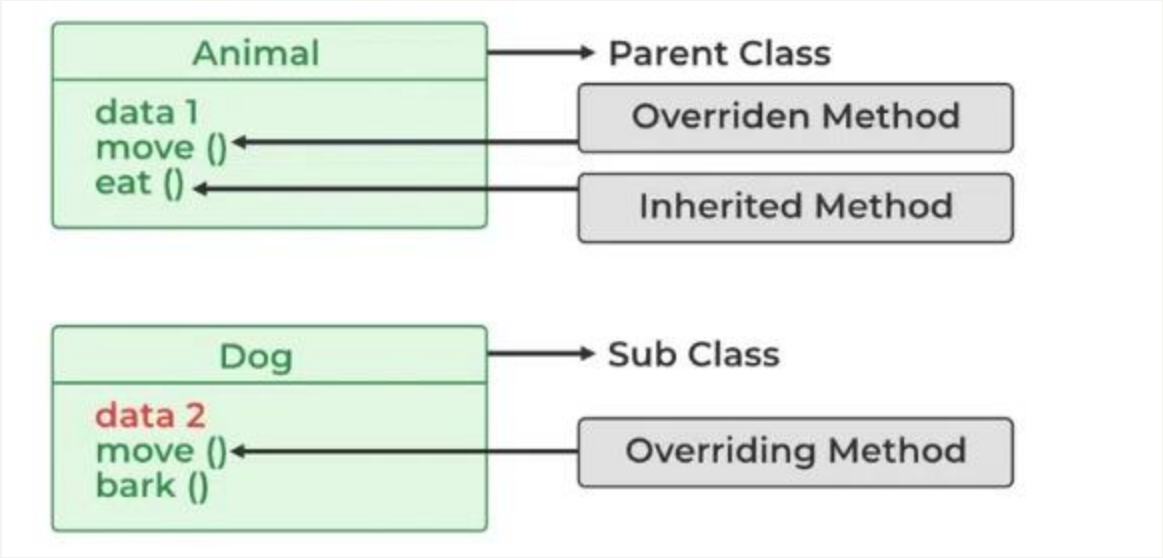
vertically

Java Polymorphism

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
- Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks.
- Method overriding is one way to realize polymorphism.

Method overriding

- In Java, overriding is a feature that allows a subclass/child class to provide a specific implementation of a method that is already provided by its super/parent classes.
- When a method in a subclass has the **same** name, the **same** parameters or signature, and the **same** return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.



- https://www.geeksforgeeks.org/overriding-in-java/

Example of Method Overriding in Java

```java
// Base Class
class Parent {
    void show() { System.out.println("Parent's show()"); }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override void show()
    {
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        // If a Parent type reference refers
        // to a Parent object, then Parent's
        // show is called
        Parent obj1 = new Parent();
        obj1.show();

        // If a Parent type reference refers
        // to a Child object Child's show()
        // is called. This is called RUN TIME
        // POLYMORPHISM.
        Parent obj2 = new Child();
        obj2.show();
    }
}
```

Output

```
Parent's show()
Child's show()
```

Rules for Java Method Overriding

- 1. Overriding and Access Modifiers
- 2. Final methods cannot be overridden
- 3. Static methods cannot be overridden
- 4. Private methods cannot be overridden
- 5. The overriding method must have the same return type (or subtype)
- 6. Invoking overridden method from sub-class

1. Overriding and Access Modifiers

```java
class Parent {
    // private methods are not overridden
    private void m1()
    {
        System.out.println("From parent m1()");
    }

    protected void m2()
    {
        System.out.println("From parent m2()");
    }
}

class Child extends Parent {
    // new m1() method
    // unique to Child class
    private void m1()
    {
        System.out.println("From child m1()");
    }

    // overriding method
    // with more accessibility
    @Override public void m2()
    {
        System.out.println("From child m2()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Parent();
        obj1.m2();
        Parent obj2 = new Child();
        obj2.m2();
    }
}
```

When overriding, we could change the accessibility of methods from lower to higher. Such as, protected in super class → public in subclass.

Output

```
From parent m2()
From child m2()
```

2. Final methods cannot be overridden

```java
// A Java program to demonstrate that
// final methods cannot be overridden

class Parent {
    // Can't be overridden
    final void show() {}
}

class Child extends Parent {
    // This would produce error
    void show() {}
}
```

Output

```
13: error: show() in Child cannot override show() in Parent
    void show() {  }
         ^
  overridden method is final
```

3. Static methods cannot be overridden

```
class Parent {
    // Static method in base class
    // which will be hidden in subclass
    static void m1()
    {
        System.out.println("From parent "
                           + "static m1()");
    }

    // Non-static method which will
    // be overridden in derived class
    void m2()
    {
        System.out.println(
            "From parent "
            + "non - static(instance) m2() ");
    }
}

class Child extends Parent {
    // This method hides m1() in Parent
    static void m1()
    {
        System.out.println("From child static m1()");
    }

    // This method overrides m2() in Parent
    @Override public void m2()
    {
        System.out.println(
            "From child "
            + "non - static(instance) m2() ");
    }
}
```

```
// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Child();

        // As per overriding rules this
        // should call to class Child static
        // overridden method. Since static
        // method can not be overridden, it
        // calls Parent's m1()
        obj1.m1();

        // Here overriding works
        // and Child's m2() is called
        obj1.m2();
    }
}
```

Output

```
From parent static m1()
From child non - static(instance) m2()
```

More explanation of the above example

```
// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Child();

        // As per overriding rules this
        // should call to class Child static
        // overridden method. Since static
        // method can not be overridden, it
        // calls Parent's m1()
        obj1.m1();

        // Here overriding works
        // and Child's m2() is called
        obj1.m2();
    }
}
```

We can create parent instances that are in fact subclasses.

In this case, the instance can call the overridden method in the subclass and ignore the same method in the parent class.

4. Private methods cannot be overridden

```
class SuperClass {
    private void privateMethod()
    {
        System.out.println(
            "This is a private method in SuperClass");
    }

    public void publicMethod()
    {
        System.out.println(
            "This is a public method in SuperClass");
        privateMethod();
    }
}

class SubClass extends SuperClass {
    // This is a new method with the same name as the
    // private method in SuperClass
    private void privateMethod()
    {
        System.out.println(
            "This is a private method in SubClass");
    }

    // This method overrides the public method in SuperClass
    public void publicMethod()
    {
        System.out.println(
            "This is a public method in SubClass");
        privateMethod(); // calls the private method in
                         // SubClass, not SuperClass
    }
}
```

```
public class Test {
    public static void main(String[] args)
    {
        SuperClass obj1 = new SuperClass();
        obj1.publicMethod(); // calls the public method in
                             // SuperClass

        SubClass obj2 = new SubClass();
        obj2.publicMethod(); // calls the overridden public
                             // method in SubClass
    }
}
```

Output

```
This is a public method in SuperClass
This is a private method in SuperClass
This is a public method in SubClass
This is a private method in SubClass
```

5. The overriding method must have the same return type (or subtype)

```
Object x = new Object();
x = "Hello, World";
```

```
class SuperClass {
    public Object method()
    {
        System.out.println(
            "This is the method in SuperClass");
        return new Object();
    }
}

class SubClass extends SuperClass {
    public String method()
    {
        System.out.println(
            "This is the method in SubClass");
        return "Hello, World!";
    }
}

public class Test {
    public static void main(String[] args)
    {
        SuperClass obj1 = new SuperClass();
        obj1.method();

        SubClass obj2 = new SubClass();
        obj2.method();
    }
}
```

New object() can be data with any data types.

Output

```
This is the method in SuperClass
This is the method in SubClass
```

6. Invoking overridden method from sub-class

```java
// A Java program to demonstrate that overridden
// method can be called from sub-class

// Base Class
class Parent {
    void show() { System.out.println("Parent's show()"); }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override void show()
    {
        super.show();
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj = new Child();
        obj.show();
    }
}
```

Calling an overridden method in a superclass within an overridden method in a subclass.

Output

```
Parent's show()
Child's show()
```

toString() method

- All the classes in Java **extend** a parent class known as the **Object class**, and this superclass has methods like equals(), toString(), etc., that are automatically inherited by all the children or subclasses.
- So, if you try to print an object of these classes, the Java compiler internally invokes the default **toString()** method on the object and will generally provide the **string representation** of the object, which will be printed on the console in the format [ClassName@hashcode()], e.g., [Object@19821f].

Default toString() method

The toString() method returns the String representation of the object.

```java
class Student {
    private int id;
    private String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
// Driver class to test the Student class
public class Demo {
    public static void main(String[] args) {
        Student s = new Student(101, "James Bond");
        System.out.println("The student details are: "+s);
    }
}
```

Expected output

```
The student details are: Student@28d93b30
```

In the constructor, we prefer to use this keyword to refer to the attributes of the object. You will learn this keyword mainly in the next class

Overriding toString() method

```java
class Student {
    private int id;
    private String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString() {
        return id + " " + name;
    }
}
// Driver class to test the Student class
public class Demo {
    public static void main(String[] args) {
        Student s = new Student(101, "James Bond");
        System.out.println("The student details are: "+s);
    }
}
```

Expected output

```
The student details are: 101 James Bond
```

After overriding, we could get expected information about one object about invoking toString() method.


End