Semana 12

Herencia y Polimorfismo

Herencia en Java

Definición

Es el mecanismo en Java por el cual una clase puede heredar las características (campos y métodos) de otra clase. En Java, Herencia significa crear nuevas clases basadas en las existentes. Una clase que hereda de otra clase puede reutilizar los métodos y campos de esa clase. Además, también puede agregar nuevos campos y métodos a su clase actual.

¿Por qué se necesita la herencia?

- **Reutilización del Código:** El código escrito en la Superclase es común a todas las subclases. Las clases secundarias pueden usar directamente el código de la clase principal.
- **Sobreescritura de métodos:** la sobreescritura de métodos solo se puede lograr a través de la herencia. Es una de las formas en que Java logra el polimorfismo de tiempo de ejecución.
- **Abstracción:** El concepto de abstracto donde no tenemos que proporcionar todos los detalles se logra a través de la herencia. La abstracción solo muestra la funcionalidad al usuario.

Términos importantes en herencia

- **Clase:** La clase es un conjunto de objetos que comparten características/comportamiento y propiedades/atributos comunes. La clase no es una entidad del mundo real. Es solo una plantilla, un plano o un prototipo a partir del cual se crean los objetos.
- **Superclase/clase principal:** la clase cuyas características se heredan se conoce como superclase (o clase base o clase principal).
- **Subclase/clase secundaria:** la clase que hereda la otra clase se conoce como subclase (o clase derivada, clase extendida o clase secundaria). La subclase puede agregar sus propios campos y métodos además de los campos y métodos de la superclase.
- **Reutilización:** la herencia admite el concepto de "reutilización", es decir, cuando queremos crear una nueva clase y ya existe una clase que incluye

parte del código que queremos, podemos derivar nuestra nueva clase de la clase existente. Al hacer esto, estamos reutilizando los campos y métodos de la clase existente.

Uso de la Herencia en Java

La palabra reservada **extends** se usa para la herencia en Java. El uso de la palabra reservada **extends** indica que se deriva de una clase existente. En otras palabras, "extiende" se refiere a una mayor funcionalidad.

Sintaxis

```
class derived-class extends base-class
{
   //métodos y campos
}
```

Ejemplos básicos

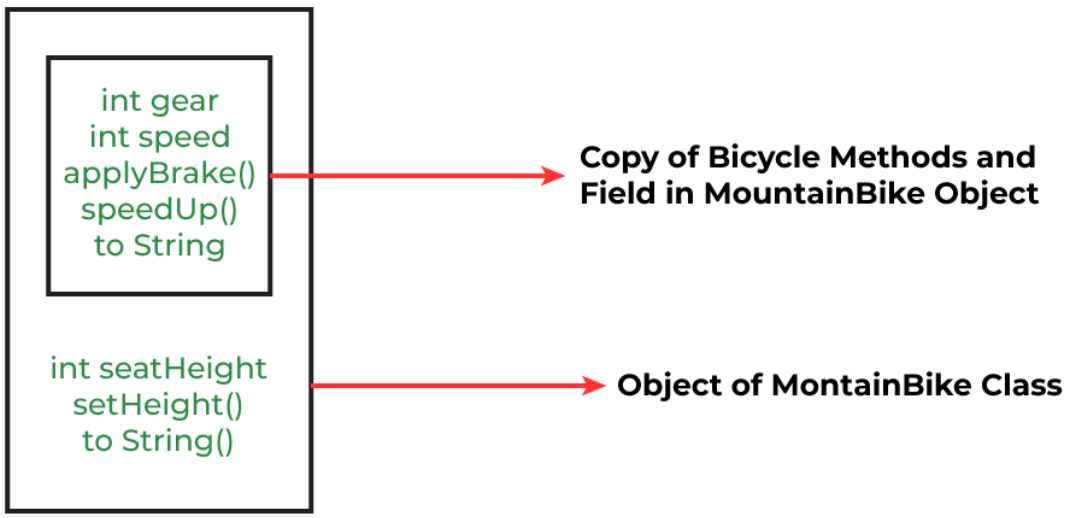
En el siguiente ejemplo de herencia, la clase Bicycle es una clase base, la clase MountainBike es una clase derivada que amplía la clase Bicycle y la clase Test es una clase principal para ejecutar el programa.

```
// Programa en Java para ilustrar el concepto de Herencia

// Clase Base
class Bicycle {
    // la clase Bicicleta tiene dos campos
    public int gear;
    public int speed;

    // La clase de bicicleta tiene un constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // la clase Bicicleta tiene tres métodos
    public void applyBrake(int decrement)
    {
```

```
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
        speed += increment;
    }

    // método toString() para imprimir información de bicicleta
    public String toString()
    {
        return ("Numero de engranajes son " + gear + "\n"
                + "la velocidad de la bici es " + speed);
    }
}

// clase derivada
class MountainBike extends Bicycle {

    // la subclase MountainBike agrega un campo más
    public int seatHeight;

    // la subclase MountainBike tiene un constructor
    public MountainBike(int gear, int speed,
                        int startHeight)
    {
        // invocando el constructor de clase base (bicicleta)
        super(gear, speed);
        seatHeight = startHeight;
    }

    // la subclase MountainBike agrega un método más
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    // Sobreescribiendo el método toString() de Bicycle para imprimir más
información
    @Override public String toString()
    {
        return (super.toString() + "\nla altura del asiento es "
                + seatHeight);
```

```
    }
}

// clase main
public class Test {
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

En el programa anterior, cuando se crea un objeto de la clase MountainBike, una copia de todos los métodos y campos de la superclase adquiere memoria en este objeto. Es por eso que usando el objeto de la subclase también podemos acceder a los miembros de una superclase.



Ejemplo más corto

En el siguiente ejemplo de herencia, la clase Empleado es una clase base, la clase Ingeniero es una clase derivada que amplía la clase Empleado y la clase Test es una clase principal para ejecutar el programa.

```
// Programa conciso en Java para ilustrar la herencia

import java.io.*;
```

```java
// Clase base o Super clase
class Employee {
    int salary = 60000;
}

// Clase derivada o sub clase
class Engineer extends Employee {
    int benefits = 10000;
}

// Clase main
class Test {
    public static void main(String args[])
    {
        Engineer E1 = new Engineer();
        System.out.println("Salario : " + E1.salary
                           + "\nBeneficios : " + E1.benefits);
    }
}
```

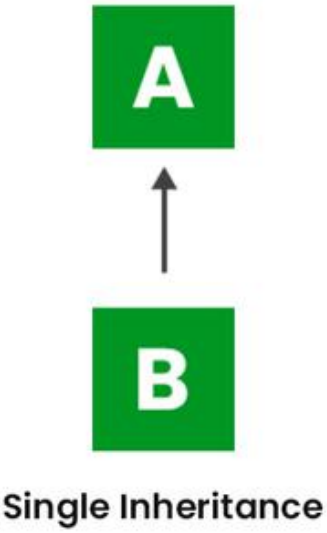
En la práctica, la herencia y el polimorfismo se usan juntos en Java para lograr un rendimiento rápido y legibilidad del código.

Tipos de Herencia en Java

A continuación se muestran los diferentes tipos de herencia compatibles con Java.

1. Herencia Simple

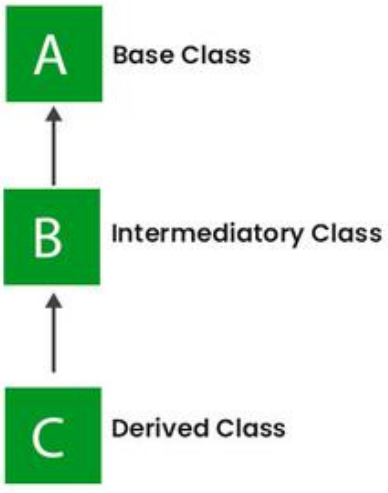
En herencia simple, las subclases heredan las características de una superclase. En la imagen a continuación, la clase A sirve como clase base para la clase B derivada.



Single Inheritance

2. Herencia Multinivel

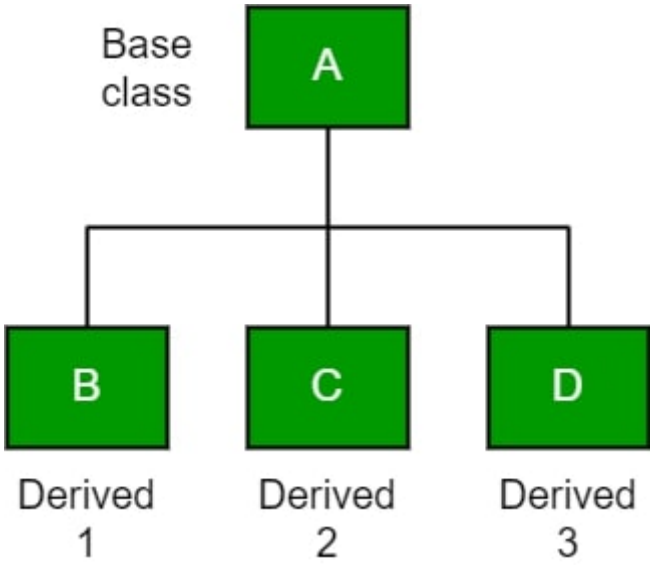
En la herencia multinivel, una clase derivada heredará una clase base y, además de que la clase derivada también actúa como clase base para otras clases. En la siguiente imagen, la clase A sirve como clase base para la clase B derivada, que a su vez sirve como clase base para la clase C derivada. En Java, una clase no puede acceder directamente a los miembros de los abuelos.



Multilevel Inheritance

3. Herencia Jerárquica

En la herencia jerárquica, una clase sirve como superclase (clase base) para más de una subclase. En la siguiente imagen, la clase A sirve como clase base para las clases derivadas B, C y D.



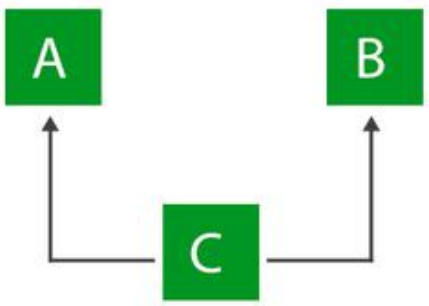
Ejemplo:

```
// Programa en Java para Ilustrar la Herencia Jerárquica

class A {
    public void print_A() { System.out.println("Clase A"); }
}

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

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

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

// Clase main
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();
    }
}
```

4. Herencia múltiple (mediante interfaces)

En las herencias múltiples, una clase puede tener más de una superclase y heredar características de todas las clases principales. Tenga en cuenta que Java no admite herencias múltiples con clases. En Java, podemos lograr múltiples herencias solo a través de Interfaces. En la imagen a continuación, la Clase C se deriva de las interfaces A y B.



Multiple Inheritance

Ejemplo:

```
// Programa Java para ilustrar la herencia múltiple mediante interfaces
import java.io.*;
import java.lang.*;
import java.util.*;
```

```java
interface one {
    public void print_one();
}

interface two {
    public void print_two();
}

interface three extends one, two {
    public void print_one();
}
class child implements three {
    @Override public void print_one()
    {
        System.out.println("Estudiantes");
    }

    public void print_two() { System.out.println("para"); }
}

// Clase principal
public class Main {
    public static void main(String[] args)
    {
        child c = new child();
        c.print_one();
        c.print_two();
        c.print_one();
    }
}
```

Relación es-un

ES-UN es una forma de decir: Este objeto es un tipo de ese objeto. Veamos cómo se usa la palabra clave extends para lograr la herencia.

```java
public class SolarSystem {
}
public class Earth extends SolarSystem {
}
public class Mars extends SolarSystem {
}
```

```
public class Moon extends Earth {
}
```

Ahora, según el ejemplo anterior, en términos orientados a objetos, lo siguiente es cierto:

- SolarSystem es la superclase de la clase Earth.
- SolarSystem es la superclase de la clase Mars.
- Las clases Earth y Mars son subclases de la clase SolarSystem.
- Moon es la subclase de las clases Earth y SolarSystem.

Lo comprobamos con el método **instanceOf**:

```
class SolarSystem {
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
    public static void main(String args[])
    {
        SolarSystem s = new SolarSystem();
        Earth e = new Earth();
        Mars m = new Mars();

        System.out.println(s instanceof SolarSystem);
        System.out.println(e instanceof Earth);
        System.out.println(m instanceof SolarSystem);
    }
}
```

¿Qué se puede hacer en una subclase?

En las subclases podemos heredar miembros tal cual, reemplazarlos, ocultarlos o complementarlos con nuevos miembros:

- Los campos heredados se pueden usar directamente, como cualquier otro campo.
- Podemos declarar nuevos campos en la subclase que no están en la superclase.
- Los métodos heredados se pueden usar directamente tal como están.

- Podemos escribir un nuevo método de instancia en la subclase que tenga la misma firma que el de la superclase, sobreescribiendolo así (como en el ejemplo anterior, se anula el método toString()).
- Podemos escribir un nuevo método estático en la subclase que tenga la misma firma que el de la superclase, ocultándolo así.
- Podemos declarar nuevos métodos en la subclase que no están en la superclase.
- Podemos escribir un constructor de subclase que invoque al constructor de la superclase, ya sea implícitamente o usando la palabra clave super.

Polimorfismo en Java

Ilustración de la vida real Polimorfismo: Una persona al mismo tiempo puede tener diferentes características. Como un hombre al mismo tiempo es padre, esposo y empleado. Entonces, la misma persona posee un comportamiento diferente en diferentes situaciones. Esto se llama polimorfismo.

Definición

El polimorfismo se considera una de las características importantes de la programación orientada a objetos. El polimorfismo nos permite realizar una sola acción de diferentes maneras. En otras palabras, el polimorfismo le permite definir una interfaz y tener múltiples implementaciones. La palabra "poli" significa muchos y "morfos" significa formas, por lo que significa muchas formas.

Tipos de Polimorfismo

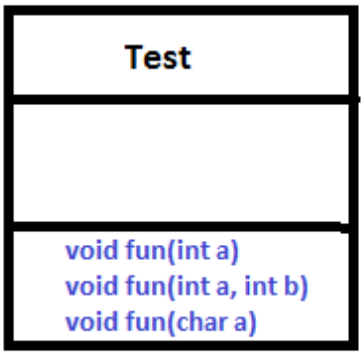
En Java, el polimorfismo se divide principalmente en dos tipos:

- Polimorfismo en tiempo de compilación
- Polimorfismo en tiempo de ejecución

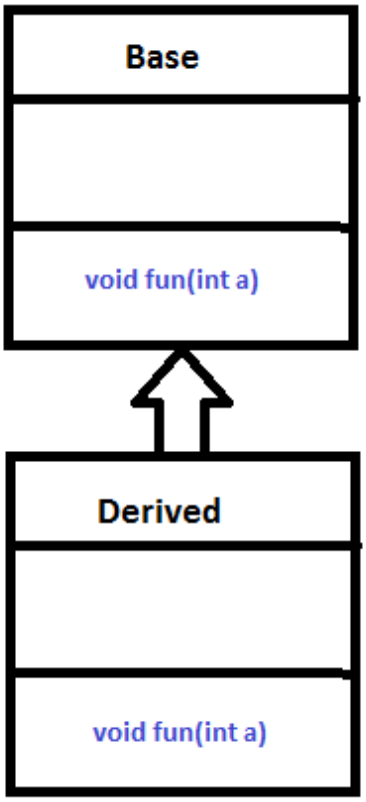
Polimorfismo en tiempo de compilación

También se conoce como polimorfismo estático. Este tipo de polimorfismo se logra mediante sobrecarga de funciones o sobrecarga de operadores.

Importante: Pero Java no admite la sobrecarga de operadores.



Overloading



Overriding

Sobrecarga de Método

Cuando hay varias funciones con el mismo nombre pero diferentes parámetros, se dice que estas funciones están **sobrecargadas**. Las funciones pueden sobrecargarse por cambios en el número de argumentos o/y un cambio en el tipo de argumentos.

Ejemplo:

```
// Programa Java para la sobrecarga de métodos mediante el uso de
diferentes tipos de argumentos

// Clase 1
// Clase Helper
class Helper {

    // Método con 2 parámetros enteros
    static int Multiply(int a, int b)
    {
```

```
        // Devuelve el producto de números enteros
        return a * b;
    }

    // Método 2
    // Con el mismo nombre pero con 2 parámetros dobles
    static double Multiply(double a, double b)
    {

        // Devuelve el producto de números dobles
        return a * b;
    }
}

// Class 2
class Test {

    // Método Main
    public static void main(String[] args)
    {

        // Llamar al método pasando la entrada como en los argumentos
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

Polimorfismo en tiempo de ejecución

También se conoce como envío de método dinámico. Es un proceso en el que una llamada de función al método sobreescrito se resuelve en tiempo de ejecución. Este tipo de polimorfismo se logra mediante la sobreescritura de métodos. La **sobreescritura de métodos**, por otro lado, ocurre cuando una clase derivada tiene una definición para una de las funciones miembro de la clase base. Se dice que esa función base está **sobreescrita**.

Ejemplo:

```
// Programa Java para sobreescritura de métodos

// Clase 1
// Clase Parent
class Parent {
```

```
    // Método de clase Parent
    void Print()
    {

        // Impresión en pantalla
        System.out.println("Clase Parent");
    }
}

// Clase 2
// Subclase 1 de Parent
class subclass1 extends Parent {

    // Método
    void Print() { System.out.println("subclase 1"); }
}

// Clase 3
// Subclase 2 de Parent
class subclass2 extends Parent {

    // Método
    void Print()
    {

        // Impresión
        System.out.println("subclase 2");
    }
}

// Clase 4
// Clase Main
class Test {

    // Método Main
    public static void main(String[] args)
    {

        // Creando objeto de clase 1
        Parent a;
```

```
        // Ahora llamaremos a los métodos de impresión dentro del método
main()

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Explicación: Aquí en este programa, cuando se crea un objeto de una clase secundaria, se llama al método dentro de la clase secundaria. Esto se debe a que el método de la clase principal está sobreescrito por la clase secundaria. Dado que el método se sobreescribe, este método tiene más prioridad que el método principal dentro de la clase secundaria. Entonces, se ejecuta el cuerpo dentro de la clase secundaria.