Asignatura

Programación

UNIVERSAE
Instituto Superior de FP

Asignatura

Programación

UNIDAD 7

Clases avanzadas y utilización de objetos

UNIVERSAE
Instituto Superior de FP

Tipos de relación entre clases

Asociación

- Dos clases tienen un enlace que las une conceptualmente
- Pueden existir sin depender la una de la otra.

Agregación

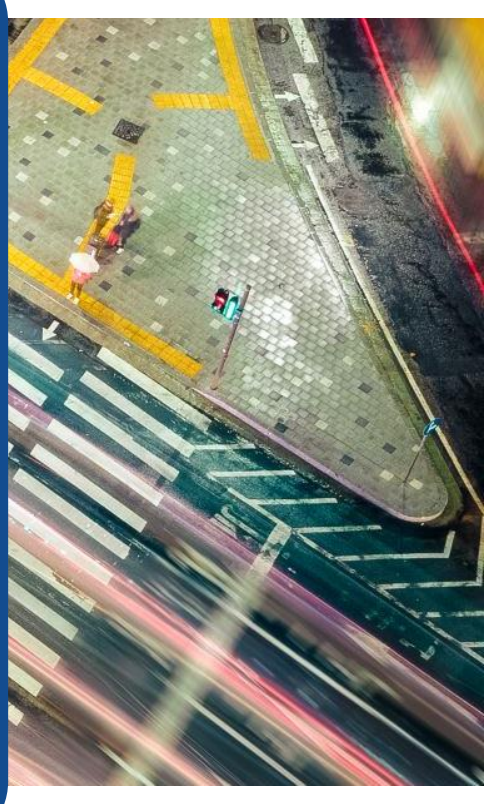
- Existe una relación que implica que una clase necesite otra dentro de su misma funcionalidad.
- Se mantiene la correlación que ambas clases pueden existir sin depender la una de la otra.

Composición

- Existe una relación fuerte.
- Una clase no puede existir, si no existe la otra.

Herencia

- Existe una clase padre, super clase o clase base.
- Las clases que se relacionan son clases hijas o subclases.
- Las clases hijas obtienen toda la funcionalidad de la base padre.



Ejemplo tipos de relaciones

Asociación

```java
public class Zoologico {

	private String nombre;

	public Zoologico(String nombre) {
		this.nombre = nombre;
	}

	public String getNombreZoologico() {
		return nombre;
	}

}
```

```java
public class Animal {

	private String nombre;

	public Animal(String nombre) {
		this.nombre = nombre;
	}

	public String getNombreAnimal() {
		return nombre;
	}

}
```

Composición

```java
public class Zoologico {

    private String nombre;
    private Ubicacion ubicacion = new Ubicacion();

    public Zoologico(String nombre) {
        this.nombre = nombre;
    }

    public String getNombreZoologico() {
        return nombre;
    }

}
```

```java
public class Ubicacion {

    private String ciudad;
    private String direccion;

    public Ubicacion(String ciudad, String direccion) {

        this.ciudad = ciudad;
        this.direccion = direccion;
    }

    public Ubicacion() {
        this.ciudad = "por defecto";
        this.direccion = "por defecto";
    }
}
```

Agregación

```java
public class Animal {

	private String nombre;
	private Animal padre;

	public Animal(Animal padre, String nombre) {
		this.padre = padre;
		this.nombre = nombre;
	}

	public String getNombreAnimal() {
		return nombre;
	}

}
```

Herencia


```java
public class Tigre extends Animal {

    public Tigre() {
        super();
    }
```

Herencia

- Es una característica de la programación orientada de objetos
- Todas las clases comparten conceptos comunes.
- Permite generalizar conceptos.
- Útil para crear jerarquía de clases.

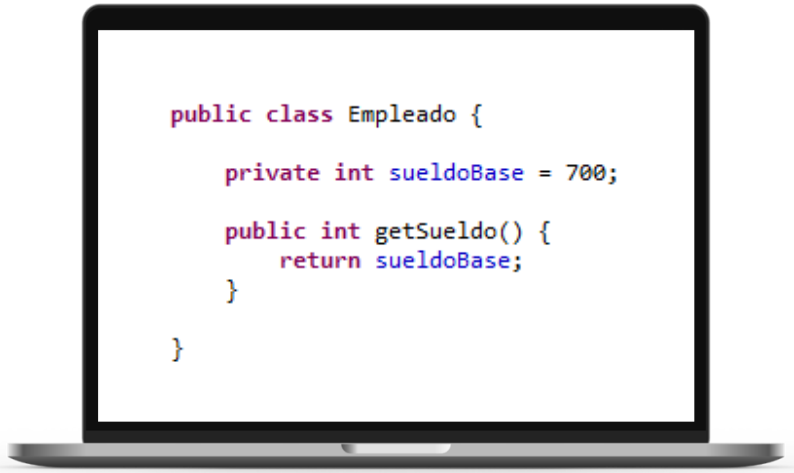
Ventajas

- Reutilizar código
- Facilidad de mantenimiento
- Facilidad de creación de objetos

Procedimiento para establecer la herencia

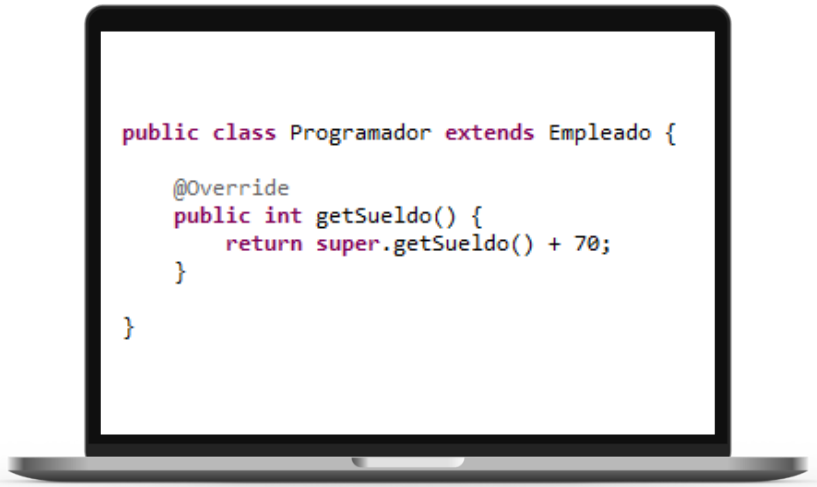
- Definir una clase padre
- Las clases hijas en su cabecera deben de utilizar la palabra reservada **extends** *Nombre de la clase padre*

Clase padre



```java
public class Empleado {

    private int sueldoBase = 700;

    public int getSueldo() {
        return sueldoBase;
    }

}
```

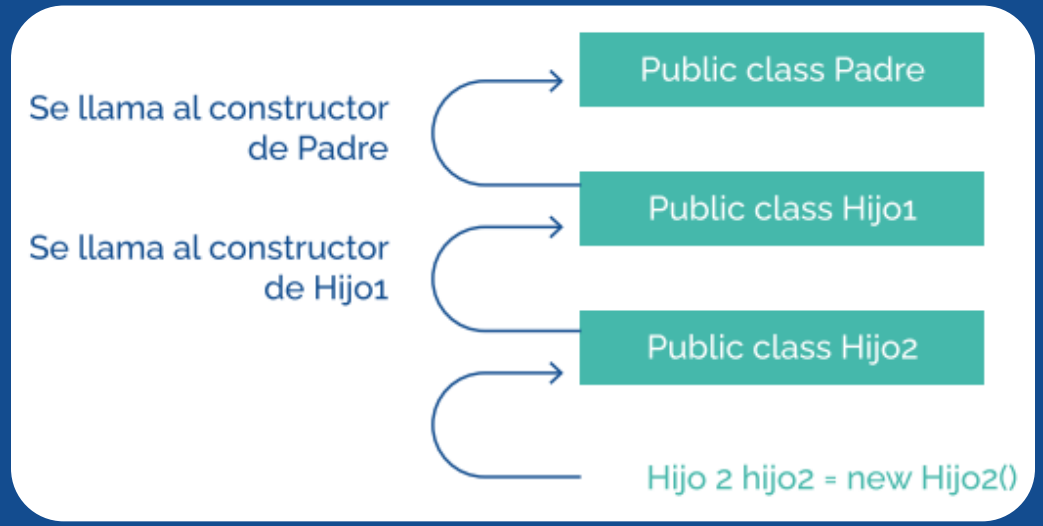
Clase hija



```java
public class Programador extends Empleado {

    @Override
    public int getSueldo() {
        return super.getSueldo() + 70;
    }

}
```

Herencia. Constructores

- Los constructores no se heredan.
- Cada clase tiene que definir su propio constructor.
- En las clases hijas, obligatoriamente hay que llamar al constructor padre.
- Tiene que ser la primera instrucción.
- Por cada instanciación de una clase hija, se produce una llamada en cascada.




```java
public class Empleado {

    private String nombre;

    public Empleado() {

    }

    public Empleado(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

}
```

```java
public class Programador extends Empleado {

    public Programador() {
        super();
    }

    public Programador(String nombre) {
        super(nombre);
    }

}
```

Herencia. Acceso a elementos

Campos

- Si los campos de la clase padre son *public*. (NO RECOMENDABLE)
  - Acceso directo
- Mismo nombre que la clase padre
  - Se usa el de la clase hija. Para usar el del padre, con *super*.
- Si hay mas de un nivel de herencia
  - Usar casting del nivel intermedio para acceder al superior.

```java
public class Empleado {

    public String nombre;
    private String apellido;

    public String getNombre() {
        return nombre;
    }

    public String getApellido() {
        return apellido;
    }

}
```

```java
public class Programador extends Empleado {

    private String nombre;

    public void setNombre() {
        super.nombre = "Jesús";
        this.nombre = "Antonio";
    }
}
```

Métodos

- Si los métodos de la clase padre son *public*
  - Acceso directo
- Mismo nombre que la clase padre
  - Se usa el de la clase hija. Para usar el del padre, con *super*

```java
public class Programador extends Empleado {

    private String nombre;

    public String getNombre() {
        return nombre;
    }

    public String getNombrePadre() {
        return super.getNombre();
    }

}
```

```java
public class Empleado {

    private String nombre;
    private String apellido;

    public String getNombre() {
        return nombre;
    }

    public String getApellido() {
        return apellido;
    }

}
```

Herencia. Sobreescritura de métodos (Override)

- Mismo método que la clase padre
- Necesidad de cambiar su funcionalidad.
- La clase hija necesita declarar el mismo método
- Usar la palabra reservada *@Override*

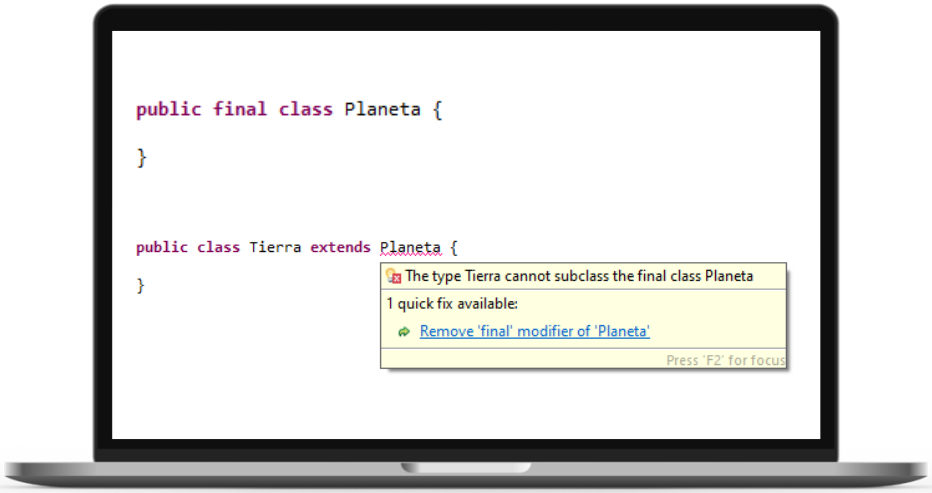
```java
public class Empleado {

    private String nombre;
    private int sueldoBase;

    public Empleado() {
        this.sueldoBase = 700;
    }

    public Empleado(String nombre) {
        this.sueldoBase = 700;
        this.nombre = nombre;
    }

    public int getSueldo() {
        return sueldoBase;
    }

    public String getNombre() {
        return nombre;
    }

}
```

```java
public class Programador extends Empleado {

    public Programador() {
        super();
    }

    public Programador(String nombre) {
        super(nombre);
    }

    @Override
    public int getSueldo() {
        return super.getSueldo() + 70;
    }

}
```

Clases y métodos finales

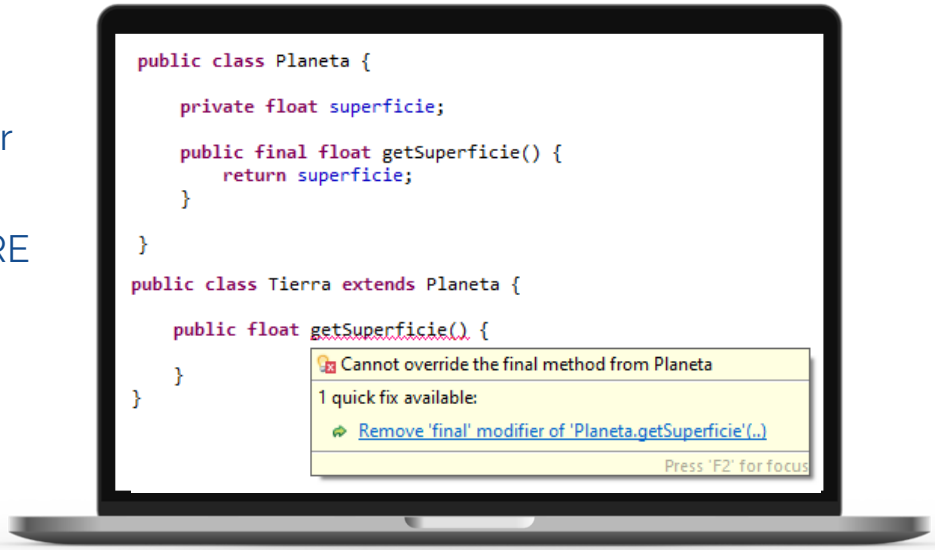
Clases

- Son clases que no se puede heredar de ellas.
- Motivos:
    - Seguridad.
    - Su código ya no evoluciona.
    - Cortar la herencia
- Sintaxis: [visibilidad] **final** class NOMBRE



Métodos

- Permite que los métodos no se puedan sobreescribir
- La funcionalidad solo la establece la clase padre
- Sintaxis: [visibilidad] **final** [void o tipo retorno] NOMBRE



Clases y métodos abstractos

Clases

- Son clases que contienen No se puede instanciar.
- ¿No tiene constructores? Puede tenerlos, para las clases hijas
- Motivos:
  - Conceptos abstractos, que no se pueden representar.
  - Plantilla base para las clases hijas
- Sintaxis: [visibilidad] **abstract** class NOMBRE

```
public abstract class Animal {

    private String nombre;

    public abstract void formaDeComunicarse();

}
```

Métodos

- Solo pueden existir métodos abstractos si la clase es abstracta
- La clase padre solo establece la cabecera del método.
- Las clases hijas obligatoriamente tienen que desarrollar su funcionalidad.
- Sintaxis: [visibilidad] **abstract** [void o tipo retorno] NOMBRE

```
public class Tigre extends Animal {

    @Override
    public void formaDeComunicarse() {
        System.out.println("Ruge, gruñe o ronronea");

    }

}
```

Interfaces

Las interfaces

- Tipo de clase sin funcionalidad.
- Características:
    - Permite organizar y estructurar el código
    - Separa la parte visual a su implementación.
- Contiene:
    - Constantes
    - Métodos sin implementar
- Motivos:
    - Plantilla o esqueleto

Implementación

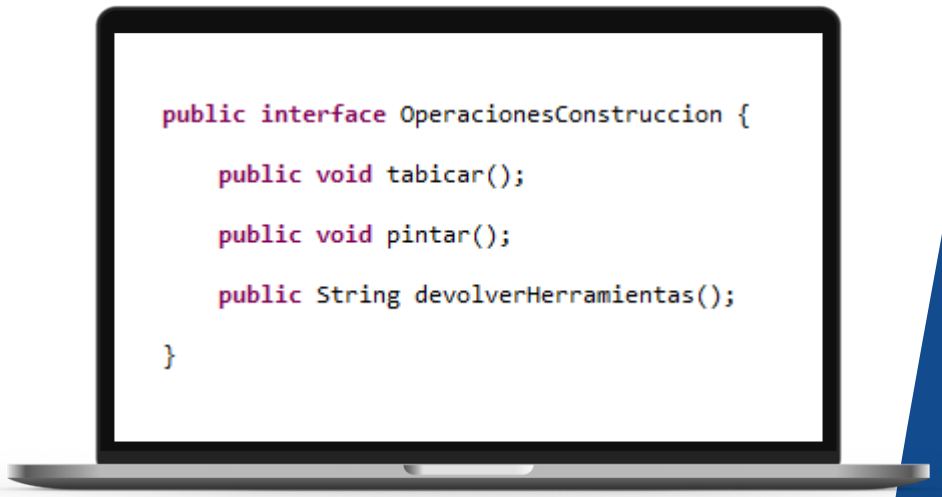
- Para declarar la interfaz

  [visibilidad] ***interface*** *NOMBRE*
- Para implementar la interfaz

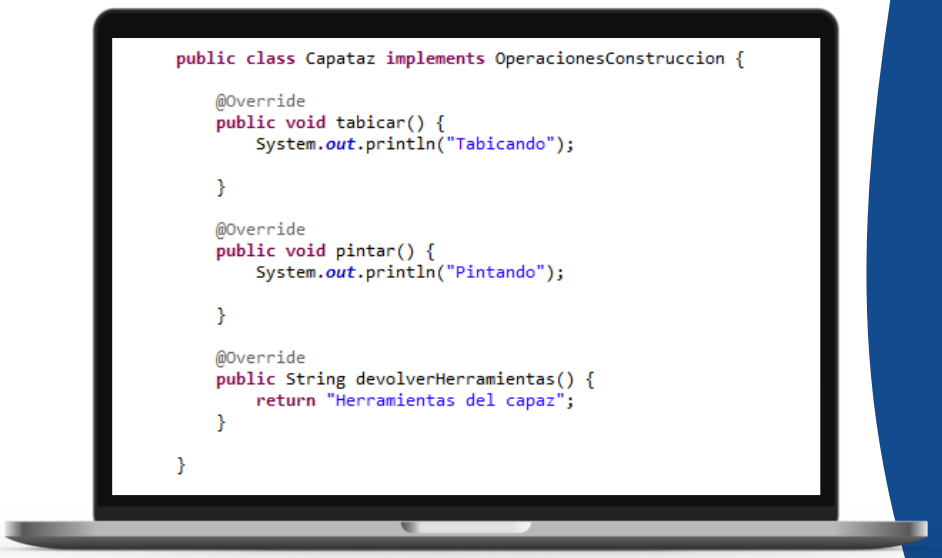
  [visibilidad] class NOMBRE ***implements*** *NOMBRE INTERFAZ*

Herencia múltiple

- Una clase puede heredar e implementar vas de una interfaz



```java
public interface OperacionesConstruccion {

    public void tabicar();

    public void pintar();

    public String devolverHerramientas();

}
```



```java
public class Capataz implements OperacionesConstruccion {

    @Override
    public void tabicar() {
        System.out.println("Tabicando");

    }

    @Override
    public void pintar() {
        System.out.println("Pintando");

    }

    @Override
    public String devolverHerramientas() {
        return "Herramientas del capaz";
    }

}
```

Polimorfirmo

¿Qué es?

- Un objeto puede adoptar diferentes formas según su herencia.
- Afecta en tiempo de ejecución.

Asignación polimorfa

- Se puede crear un objeto del tipo clase padre a partir de una clase hija
- Al revés no es posible
- Ejemplo, *ClasePadre objeto = new ClaseHija();*

Ejecución polimorfa

- La asignación polimorfa permite la invocación correcta de los métodos
- ¿Y si una clase hija tiene un método que no esta en la clase padre?
  Solución, **casting** de la clase hija
- ¿Y si no conocemos que clase hija es?
  Solución, consultar la clase con *instanceof*

Ejemplo. Polimorfismo

```java
public class Empleado {

    private int sueldoBase;

    public Empleado() {
        this.sueldoBase = 700;
    }

    public int getSueldo() {
        return sueldoBase;
    }

}
```

```java
public class Programador extends Empleado {

    @Override
    public int getSueldo() {
        return super.getSueldo() + 70;
    }

}
```

```java
public class Analista extends Empleado {

    @Override
    public int getSueldo() {
        return super.getSueldo() + 500;
    }

    public void analizarProyecto() {
        System.out.println("El analista analiza el proyecto");
    }

}
```

```java
public static void main(String[] args) {

    Empleado empleado1 = new Analista();
    Empleado empleado2 = new Programador();

    System.out.println("Sueldo del analista: " + empleado1.getSueldo());
    System.out.println("Sueldo del programador: " + empleado2.getSueldo());

    if(empleado1 instanceof Analista) {
        ((Analista) empleado1).analizarProyecto();
    }

}
```

```
Sueldo del analista: 1200
Sueldo del programador: 770
El analista analiza el proyecto
```

Resumen

1. Tipos de relación entre clases
2. Ejemplos tipo de relaciones
3. Herencia
4. Herencia. Constructores
5. Herencia. Acceso a los elementos
6. Herencia. Sobreescritura de métodos (Override)
7. Clases y métodos finales
8. Clases y métodos abstractos
9. Interfaces
10. Polimorfirmo
11. Ejemplo. Polimorfirmo

UNIVERSAE

CHANGE YOUR WAY