M.3611.056.003 - Curso de Introducción a la programación con JAVA (Presencia Virtual Online - Cádiz)

POO3: Herencia, Polimorfismo, Abstracción

Eduardo Corral Muñoz
eoi_eduardo@corral.es

Programación Orientada a Objetos 3

Índice

— 01

Herencia

Herencia

- En la vida real, los descendientes heredan de sus ascendientes características como rasgos físicos, color de ojos, de pelo, altura, complexión, color de piel, … y comportamientos como habilidades (deporte), gustos, preferencias …
- En Java, la herencia también es una relación de parentesco entre dos clases, una es padre y la otra hija. La hija hereda los atributos y métodos del padre (y de todos los ascendientes).
- Es el mecanismo OOP para establecer relaciones jerárquicas entre clases. Es uno de sus fundamentos.
- Una subclase, clase hija, hereda atributos y métodos de todos sus ancestros.
- Es una de las bases de reutilización de código.
- La herencia solo es aplicable cuando existe una relación “familiar” entre las clases.

Herencia

- El concepto de herencia se divide en dos grupos:
  - **subclass** – subclase (child) – la clase que hereda de otra
  - **superclass** - superclase (parent) – de la que se hereda
- Permite la reusabilidad del código. Reutilizamos atributos y métodos de una clase existente en una clase nueva.
- Si la superclase necesita proteger sus atributos se utiliza el modificador **protected** en lugar de **private** para poder heredarlo.
- Si deseamos evitar que una clase sea heredable utilizaremos la palabra clave final.
- En los diagrmas UML se indica la herencia con la “flecha blanca”.

Herencia

Para heredar de una clase se usa la palabra clave **extends**.

```
class Vehiculo { // Clase Padre = Superclase
  protected String marca = "Ford";
  public void claxon() {
    System.out.println("Tuut, tuut!");
  }
}

class Coche extends Vehiculo { // Clase hija = Subclase
  private String modelo = "Mustang";
  public static void main(String[] args) {
    Coche miCoche = new Coche();
    miCoche.claxon();
    System.out.println(miCoche.marca + " " + miCoche.modelo);
  }
}
```

Herencia

- Ejemplos de herencia

```
class Persona { … }

// Hijos de la clase Persona
class Alumno extends Persona { … }
class Profesor extends Persona { … }
class Director extends Persona { … }

// Nietos de la clase Persona, hijos de la clase Alumno
class AlumnoIngenieria extends Alumno { … }
class AlumnoErasmus extends Alumno { … }
```

Herencia

- La creación de jerarquías de clase, herencia, puede hacerse de dos maneras diferentes:
  - **Generalización**

    Hemos creado dos o más clases que tienen en común algunos atributos y/o métodos, refactorizamos el código y llevamos los elementos comunes a una clase “padre” y “extendemos” las otras (hijas) con los elementos no comunes. (Alumno, Profesor, Director → Persona)
  - **Especialización**

    Partimos de una clase ya creada con anterioridad (padre) y creamos las clases que la extienden (hijas) añadiendo atributos o métodos que “especializan” la clase anterior. (Persona → Alumno)

— 02

Heredando métodos Polimorfismo

Polimorfismo

- Polimorfismo = múltiples formas.
- Se produce cuando tenemos múltiples clases que están relacionadas entre sí por herencia.
- El polimorfismo utiliza los métodos heredados de una superclase para realizar diferentes tareas mediante sobrecarga de métodos, añadiendo nuevos parámetros y/o cambiando el tipo del propio método. Esto nos permite realizar un sola acción de diferentes maneras.
- Herencia y polimorfismo permiten la reusabilidad del código. Reutilizamos atributos y métodos de una clase existente en una clase nueva sin alterarlos en la clase padre.

Polimorfismo

```
// superclase
public class Animal{
	public void sonido(){
		System.out.println(“El animal emite un sonido”);
	}
}
// subclases
class Cerdo extends Animal{
	public void sonido(){
		System.out.println(“El cerdo dice: güi güi”);
	}
}
class Perro extends Animal{
	public void sonido(){
		System.out.println(“El perro dice: guau guau”);
	}
}
```

Polimorfismo

```
// main
class SonidoAnimal {
	public static void main(String[] args) {
		Animal miAnimal = new Animal();
		Cerdo miCerdo = new Cerdo();
		Perro miPerro = new Perro();

		miAnimal.sonido();
		miCerdo.sonido();
		miPerro.sonido();
	}
}
```

super - constructores

- La palabra reservada **super** permite acceder al método constructor de una clase padre (superclase) desde un clase hija (subclase). Debe invocarse como primera línea del constructor de la clase hija.

```
// superclase
public class Persona{
	protected String nombre;
	protected String apellido;

	// constructor padre
	public Persona(String nombre, String apellido){
		this.nombre = nombre;
		this.apellido = apellido;
	}
}
```

super - constructores

```
// subclase
public class Alumno extends Persona{
	private String centro;

	// constructor hija
	public Alumno(String nombre, String apellido, String centro ){
		super(nombre, apellido);
		this.centro = apellido;
	}
}
```

super - métodos

- La palabra reservada **super** también nos permite acceder a los métodos de una clase padre (superclase) desde un clase hija (subclase). Debe invocarse como primera línea del método de la clase hija.

```
// superclase
public class Persona{
        protected String nombre;
        protected String apellido;

        // constructor padre
        public metodoPersona(){
                ....
        }
}
```

Sobreescritura de métodos

- La herencia también nos permite sobreescribir métodos heredados del padre, redefiniéndolos en la clase hija sin afectar a la clase padre.

```
// superclase
public class Persona{
        private String nombre;
        private String apellido;

        // método padre
        public void metodoDelPadre(){
                …
        }
}
```

```
// subclase
public class Alumno extends Persona{
	private String centro;
	private Float calificacion;

	// método que usa el método del padre
	public void metodoDelHijo(){
		// Otras instrucciones
		super.metodoDelPadre();
		// Otras instrucciones
	}
```

Sobreescritura de métodos

- Empleamos la notación **@Override** (anular, ignorar) indicando que sutituimos el método del padre por la nueva definición en el hijo.

```
// subclase
public class Alumno extends Persona{
	private String centro;
	private Float calificacion;

	// método con el mismo nombre que el del padre
	@Override
	public void metodoDelPadre(){
		// Otra función diferente
	}
}
```

Sobreescritura de métodos

- Con **@Override** y **super** podemos “ampliar” la funcionalidad del método original, incorporando el método padre y añadiendo, antes o después de invocarlo, nuevas instrucciones.

```
// subclase
public class Alumno extends Persona{
    private String centro;
    private Float calificacion;

    // método con el mismo nombre que el del padre
    @Override
    public void metodoDelPadre(){
        // Otras instrucciones
        super.metodoDelPadre();
        // Otras instrucciones
    }
}
```

Sobreescritura de métodos

```
// subclase
public class Persona{
        …
        @Override
        public String nombreCompleto(){
                return this.nombre + “ “ + this.apellido;
        }
}
public class Alumno extends Persona{
        …
        @Override
        public String nombreCompleto(){
                return super.nombreCompleto() + this.centro;
        }
}
```

Restringir la herencia

- Podemos impedir la herencia mediante el modificador **final**, con éste modificador no es posible crear subclases.

```
final public class Persona{
    …
}
```

- Podemos impedir también la sobreescritura de métodos con **final.**

```
public class Persona{

    …
    final public void metodoDelPadre();
}
```

— 03

Abstracción
Clases abstractas

Abstracción

- La abstracción de datos es el proceso de ocultar determinados detalles y mostrar al usuario solamente la información esencial.
- Se puede conseguir con clases e interfaces abstractos.
- El modificador **abstract** es un modificador de no-acceso utilizado para clases y métodos.
- **abstract class**: clase restringida con la que no se pueden crear objetos, debe ser heredada por otra clase para acceder. Si se intenta instanciar un objeto con ellas, se produce un error.
- **abstract method**: solo se pueden usar en clases abstractas, no tienen cuerpo hasta que se define en la subclase donde ha sido heredado.
- Una clase abstracta puede contener tanto métodos abstractos como normales.

Clases abstractas

- Es una clase que representa un concepto genérico (abstracto) que no debe ser ejemplarizado.
- Tenemos una jerarquía en las que todas la clases tienen un método común, pero se implementa de forma diferente en cada una de ellas, lo sobreescribimos en todas ellas.
- Una clase abstracta está diseñada para ser una superclase.
- No pueden ser instanciadas, solo heredadas.

Clases abstractas

- Creando una clase abstracta

```
abstract class Persona{
        protected String nombre;
        protected String apellido;
        protected String email;
        …
        // métodos abstractos
        abstract public void registro();
        abstract public void editar();
        abstract public void borrar();

        // métodos normales
        public String nombeCompleto(){
                return this.nombre + “ “ + this.apellido;
        }
}
```

Clases abstractas

- Creando otras clases abstractas

```
abstract class figuraGeometrica{
    // métodos abstractos
    abstract public void perimetro();
    abstract public void area();
    abstract public void dibujar();
}

abstract class Mamiferos{
    private String habitat;
    private String color;
    …
    abstract public void sonido();
    abstract public void comer();
    abstract public void dormir();
}
```

Clases abstractas

- Creando subclases desde un clase abstracta

```
class Triangulo extends figuraGeometrica{
    @Override
    public void perimetro(){
        // cálculo del perímetro del triangulo
    }

    @Override
    public void area() (){
        // cálculo del área del triangulo
    }
    …
}
```

— 04

Clases internas

Clases internas – Inner Class

- Podemos crear clases anidadas (clases dentro clases).
- El objetivo de las clases anidadas es agruparlas, haciendo el código más fácil de entender y mantener.
- Para acceder a una clase interna, creamos un objeto de la clase externa y, a continuación, creamos un objeto de la clase interna.

Clases internas – Inner Class

Clases anidadas:

```
class OuterClass {
      int x = 10;

      class InnerClass {
            int y = 5;
      }
}

public class Main {
      public static void main(String[] args) {
            OuterClass myOuter = new OuterClass();
            OuterClass.InnerClass myInner = new myOuter.new InnerClass();
            System.out.println(myInner.y + myOuter.x);
      }
}
```

Clases internas privadas

- Al contrario que las clases “habituales”, las internas pueden ser **private** o **protected**.
- Usamos **private** si no queremos que un objeto exterior acceda a la clase interna.
- Usamos **protected** si queremos limitar el acceso al interior de la clase externa.

Clases internas privadas

Clases privadas:

```
class OuterClass {
      int x = 10;

      private class InnerClass {
            int y = 5;
      }
}

public class Main {
      public static void main(String[] args) {
            OuterClass myOuter = new OuterClass();
            OuterClass.InnerClass myInner = new myOuter.new InnerClass();
            System.out.println(myInner.y + myOuter.x);
      }
}
```

Clases internas estáticas

- Una clase interna también puede ser **static**, se puede acceder a ella sin tener que crear un objeto de la clase externa.
- Una clase interna static no tiene acceso a los elementos de la clase externa.

Clases internas estáticas

- Clases estáticas:

```
class OuterClass {
      int x = 10;

      static class InnerClass {
            int y = 5;
      }
}

public class Main {
      public static void main(String[] args) {
            OuterClass.InnerClass myInner = new OuterClass.InnerClass();
            System.out.println(myInner.y);
      }
}
```

Acceso a una clase externa desde una interna

- Una de las ventajas de las clases internas es que pueden acceder a los atributos o métodos de la clase externa.

```
class OuterClass {
	int x = 10;
	class InnerClass {
		public int myInnerMethod() {
			return x;
		}
	}
}
public class Main {
	public static void main(String[] args) {
		OuterClass myOuter = new OuterClass();
		OuterClass.InnerClass myInner = myOuter.new InnerClass();
		System.out.println(myInner.myInnerMethod());
	}
}
```