Unidad 7

Clases avanzadas y utilización de objetos

Programación

7

Índice

Introducción

En la anterior unidad vimos el concepto de clase y como representaba un objeto en el mundo real sin interacción alguna con otras clases. Un programa tiene una clase raíz encargada de poner en marcha la aplicación y a partir de ese punto va creando objetos que tienen una relación común. Ahora se profundizará en la relación que hay entre clases, su encapsulamiento y ocultación al resto del programa con el objetivo de facilitar su comprensión y la reutilización de código.

Al finalizar esta unidad

- Asimilaremos que es la herencia y en qué momentos implementarla.
- Conoceremos aspectos más avanzados, clases abstractas e interfaces.
- Conoceremos que es el polimorfismo y su aplicación.

7.1. Asociación, agregación y composición de clases

Para que un programa tenga sentido y pueda funcionar adecuadamente no basta con tener solo la definición de las clases que representan los elementos reales de la lógica de nuestro programa. Es necesario definir la relación que tienen cada clase dentro del programa para que esté conectado y tenga un diseño lógico y sólido. Dependiendo de la lógica que vaya a tener el programa podemos identificar cuatro tipos de relaciones:

- **Asociación.** La relación de asociación ocurre cuando dos clases tienen un enlace que las une conceptualmente, pero pueden existir sin depender la una a la otra. Por ejemplo, Un zoológico y un animal.

```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;
	}
}
```

Imagen 1. Ejemplo de asociación

Como se ve en el ejemplo, un zoológico puede existir sin necesidad de tener un animal, y viceversa. En el código de la imagen no existe ningún campo interno de cada clase que haga referencia a la otra. En otra parte del programa se puede enlazar si un animal pertenece a un zoológico.

- **Agregación.** Es una asociación entre dos clases, la diferencia es que una de las clases declara a la otra en su interior. Se mantiene la característica que pueden existir por sí misma sin necesidad de la otra.

```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;
	}

}
```

Imagen 2. Ejemplo de agregación

Un animal puede existir sin tener un padre. Un ejemplo, nos encontramos un animal y no podemos conocer quienes son sus padres.

- **Composición.** Es una asociación que ocurre cuando una clase no tiene sentido sin otra, ambas son dependientes. Se puede apreciar cuando una clase declara en su interior como tipo de atributo a otra clase y se instancia siempre.

```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";
    }
}
```

Imagen 3. Ejemplo de composición

- **Herencia.** La relación se define entre clase padre y clase hija. La clase padre tienen toda la funcionalidad y las clases heredarán toda la funcionalidad de la clase padre. Ejemplo, existen muchos animales en la vida real con diferentes características, se puede representar con una clase cada tipo y que la clase padre (Animal) defina las características comunes de todos los tipos.

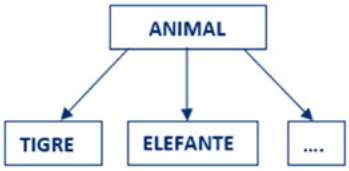


Imagen 4. Relación de herencia

7.2. Herencia

Es una característica de la programación orientada a objetos que permite generalizar conceptos y crear una jerarquía de clases. Ahora se definirá la clase padre, base, ancestra o superclase que tendrá unas características comunes para las clases hijas, extendidas, derivadas o subclase que heredaran como propias.

La ventaja de aplicar la herencia permite definir en la clase padre todo el código común y las clases hijas heredar toda la funcionalidad a excepción de la parte declarada privada y de los constructores del padre. De esta forma se reduce el código ya que no es necesario replicar los atributos y métodos de la clase padre a cada clase hija.

En java solo es posible hacer una herencia simple, de una única clase y en forma de jerarquía. No es posible la herencia múltiple, que una clase herede de dos clases diferentes.

Para heredar hay que usar la palabra reservada extends al lado del nombre de la clase.

```java
public class Hijo extends Padre { … }
```

Si una clase no va a tener funcionalidad de padre y no se quiere que se herede de ella hay que utilizar la palabra reservada final.

```java
public final class Clase {  …  }
```

7.3. Constructores y herencia

En herencia los constructores no se pueden heredan directamente si no de forma indirecta. Esto quiere decir, que cada clase tiene que definir sus constructores obligatoriamente. Una vez se instancien los objetos desde la clase hija, se realizará una llamada en cascada desde abajo hasta llegar a la clase padre.

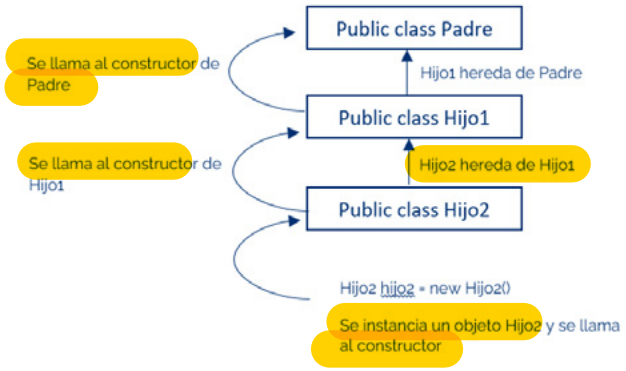


Imagen 5. Flujo de llamada a constructores

Al definir los constructores de la clase hija, lo primero que hay que hacer de forma obligatoria es invocar al constructor de la clase padre en la primera línea del código. Como se indicó, los constructores no se heredan, entonces para poder invocar al constructor de la clase padre hay que usar la palabra reservada super. Dependiendo si la llamada es al constructor por defecto o con parámetros, se usará de la siguiente forma, super() o super(parametro1, parametro2, ...). Posteriormente se añadirá todo el código del constructor de la clase hija.

```
public class Animal {

    private String nombre;
    private Animal padre;

    public Animal() {
        // Constructor por de-
        fecto
    }

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

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

}

public class Tigre extends Ani-
mal {

    public Tigre() {
        super();
    }

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

    public Tigre(String nombre,
    Tigre padre) {
        super(nombre,padre);
    }

}
```

7.4. Acceso a campos de la superclase

A diferencia de los constructores, que no se heredan y se tiene que forzar la llamada del constructor padre obligatoriamente, los atributos si son heredables, pero puede darse diferentes casuísticas cuando hay que acceder a los campos:

- Los campos de la clase padre son visibles, es decir, no se han declarado privados. Podremos acceder a ellos desde las clases hijas hasta un nivel.
- El identificador del campo padre tiene el mismo identificador del campo hija. El campo padre se oculta y solo será visible el campo de la clase hija. Para acceder al campo padre tendremos que utilizar la palabra reservada super.
- Si hay más de un nivel de jerarquía de herencia. La clase de nivel más bajo para poder acceder a los campos de clases superiores a un nivel tienen que hacer cast del nivel intermedio para hacer uso de los campos.

7.5. Acceso a métodos de la superclase

Como ya hemos visto con la palabra reservada super. podemos llamar a los constructores y campos de la clase padre. De la misma forma se puede emplear para llamar a los métodos de la clase padre siempre y cuando sean visibles.

7.6. Sobreescritura de métodos (Override)

Cuando una clase hija hereda un método, pero quiere aplicar una funcionalidad diferente a la definida por la clase padre hay que sobrescribir ese método, no es necesario crear un método nuevo por tener una funcionalidad diferente.

Se pueden sobrescribir cualquier método de la clase padre a excepción de los constructores, ya hemos visto que los constructores no se heredan y obligatoriamente se tiene invocar según el orden de llamada.

Para sobrescribir un método de la clase padre, hay que volver a crearlo con el mismo nombre, parámetros y retorno que la clase padre y además se tendrá que añadir encima de la cabecera del método la palabra reservada @override.

Al sobrescribir el método este se modificará para la clase concreta, pero el método original seguirá estando disponible para la clase padre u otras clases que hereden.

```
public class Animal {

	public void identificarse() {
		System.out.println(“Soy un animal”);
	}

}

public class Tigre extends Animal {

	@Override
	public void identificarse() {
		super.identificarse();
		System.out.println(“y un tigre”);
	}

}

public static void main(String[] args) {

	Tigre tigre = new Tigre();

	tigre.identificarse();

}
```

Resultado

Soy un animal

y un tigre

7.7. Clases y métodos abstractos

Cuando tenemos una clase que no tiene sentido crear objeto de ella porque su representación contiene conceptos abstractos, hablamos de una clase abstracta y sirve exclusivamente para la herencia. Una clase abstracta forma como plantilla para indicar como pueden definirse sus métodos en las clases hijas. Las clases hijas puede cambiar los métodos sobrescribiéndolos.

Una clase abstracta no se puede instanciar objetos de ella, con lo cual, no contiene constructores.

Para crear una clase abstracta se añadirá la palabra reservada abstract antes de la palabra reservada class

```
public abstract class Animal { .. }
```

7.7.1. Métodos abstractos

Hasta ahora habíamos visto que una clase hija puede sobrescribir cualquier método que herede del padre, cuando necesitaba cambiar su funcionalidad. Ahora veremos como desde una clase padre puede forzar a que las clases hija tengan que sobrescribir obligatoriamente un método, para ello vamos a usar los métodos abstractos. Esta funcionalidad es útil cuando se aprecia que hay una acción común entre varios elementos pero cada uno de ellos tiene diferente funcionalidad.

La clase contenedora del método abstracto obligatoriamente tiene que ser una clase abstracta, de lo contrario el compilador daría error.

Para declarar un método abstracto se añadirá al comienzo de la cabecera del método la palabra reservada abstract. Y no se definirá ningún cuerpo del método, serán las clases hijas quienes definan la funcionalidad.

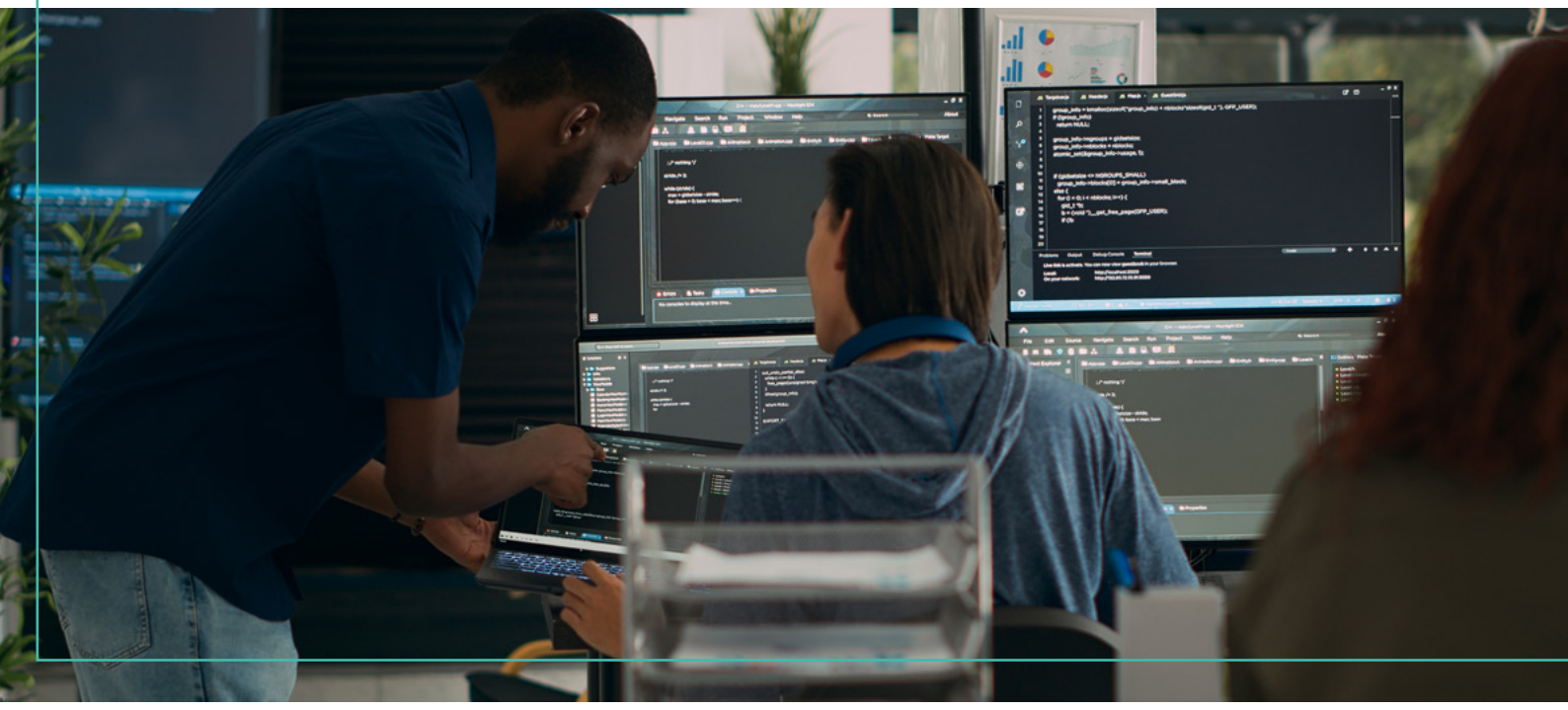
```
public abstract class Animal {

	abstract void formaDeComunicarse();

}

public class Tigre extends Animal {

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

	}

}

public class Tortuga extends Animal {

	@Override
	void formaDeComunicarse() {
		System.out.println("Emiten sonidos");

	}

}
```

7.8. Clases y métodos finales

Las clases finales, indican que de una clase no es posible heredar de ella, esto es gracias a indicar con la palabra reservada final justo delante de class. El motivo para declarar una clase final puede ser por motivos de seguridad o porque su código ya no puede evolucionar.

7.8.1. Métodos finales

Igual que sucede en las clases, podemos definir un método final para que las clases hijas no puedan sobrescribir su funcionalidad.



7.9. Interfaces

Una interfaz es un prototipo de plantilla, que proporciona un conjunto de métodos que simplifican la funcionalidad de una clase. A diferencia de una clase abstracta, una interfaz solo puede tener variables constantes y métodos sin implementar.

Las principales características de las interfaces son:

- Organización y estructuración del código
- Separación de la parte visual a su implementación.

Para declarar una interfaz se utilizará la palabra reservada interface. Y se crea un fichero igual que una clase. La sintaxis de la declaración es la siguiente:

```
public interface Identificador [extends Interface1, Interface2, _] {
// Constantes
// Cabecera de los métodos
}
```

Para hacer uso de la interfaz la clase implementadora que tiene que llamar a la interfaz mediante la palabra reservada implements.

```
public class Identificador implements IdentificadorInterfaz { }
```

Un ejemplo de una interfaz y su implementación:

```
public interface OperacionesConstruccion {

	public void tabicar();

	public void pintar();

	public String devolverHerramientas();

}

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”;
	}

}
```

7.9.1. Pseudoherencia múltiple

Como se explicó anteriormente en Java no es posible tener herencia múltiple. Esto no es del todo correcto porque se puede simular la herencia múltiple con las interfaces. Una clase hija puede heredar de la clase padre y a la vez implementar una interfaz.

En el siguiente ejemplo se puede ver como la clase Tigre hereda de Animal y a su vez implementa la interfaz AccionesCaza que tiene las acciones de caza de un animal.

```
public class Tigre extends Animal implements AccionesCaza { … }
```

7.10. Polimorfismo

El polimorfismo es la capacidad para ciertos objetos que tienen una relación entre si adoptar diferentes formas en tiempo de ejecución.

Cuando instanciamos un objeto, lo hacemos de la siguiente forma: Tipo identificador = new Tipo(), haciendo estático que el tipo de la parte izquierda de la igualdad sea igual al mismo tiene que la parte derecha de la igualdad. Ahora veremos que en java permite la relajación de tipos y gracias al polimorfismo la instanciación de un objeto pueda ser dinámica en el tiempo.

7.10.1. Asignación polimorfa

En la herencia hemos visto que existe una jerarquía de clase. Cada clase padre tiene la generalización de todas sus clases hijas, con lo cual, la clase padre puede ser instanciada a partir de cualquiera de sus hijas. Ejemplo, TipoPadre identificador = new TipoHija(); Al revés no se podría, una clase hija no puede contener la instanciación de una clase padre. A esta funcionalidad se le denomina asignación polimorfa.

7.10.2. Ejecución polimorfa

¿Qué sucede cuando se ejecuta el programa y se llama a un método de una variable con asignación polimórfica? El polimorfismo garantiza que se ejecute correctamente el método de la clase real y no la contenedora (padre) de aquellos métodos que existen por igual en ambas clases.

En cambio, cuando la clase real tiene métodos exclusivos que no existen en la clase padre no podremos invocarlos. Para poder hacer la invocación de estos métodos previamente habrá que realizar un casting a la clase real para volver a su origen.

Durante la ejecución también puede suceder que no sepamos la clase real que contiene la instancia contenedora. Este problema se soluciona con el operador instaceof para comprobar si un objeto es instancia de un tipo concreto. Ejemplo de uso del operador, identificador instanceof Tipo

En el siguiente ejemplo se puede ver uso del polimorfismo y las herramientas de casting y el operador instaceof.

```
public class Empleado {

	private int sueldoBase = 700;

	public int getSueldo() {
		return sueldoBase;
	}

}

public class Analista extends Empleado {

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

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

}

public class Programador extends Empleado {

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

}

public class EjemploPolimorfismo {

	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();
	}

	}

}
```

Resultado:

Sueldo del analista: 1200

Sueldo del programador: 770

www.universae.com