Material descargable

Programación **orientada a objetos**

egg

Asociación de objetos

En nuestras últimas lecciones, aprendimos a crear nuestras propias clases personalizadas y a instanciar objetos a partir de estas. También usamos el concepto de encapsulamiento declarando propiedades como privadas y a utilizar métodos getters y setters para acceder y modificar esos datos de manera controlada.

Hoy, vamos a introducir una forma en la que los objetos pueden interactuar y estar relacionados entre sí, la **asociación** y sus dos subtipos: la **agregación** y la **composición**.

En el mundo real, las cosas raramente existen de forma aislada, por eso estas relaciones nos permiten representar las interacciones entre objetos.

Asociación

La asociación implica que una clase "*conoce*" a otra y mantiene una referencia a ella, es decir, tiene una propiedad del tipo de esa otra clase. Hasta ahora solo usamos propiedades de clases que ya existen en Java, como String, Integer, Scanner, etc.

La asociación implica usar como propiedades a otras clases personalizadas.

Piensa en una relación entre una clase "**Jugador**" y una clase "**Equipo**". Un **jugador pertenece a un equipo**. Sin embargo, si el equipo se “disuelve”, los jugadores seguirán existiendo.

Veamos un ejemplo:

```java
public class Jugador {
    private String nombre;
    private Equipo equipo;

    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public Equipo getEquipo() {
        return equipo;
    }
    public void setEquipo(Equipo equipo) {
        this.equipo = equipo;
    }
}
```

```java
public class Equipo {
    private String nombre;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

💡 *Podríamos hacer la relación a la inversa, teniendo un arreglo de jugadores en la clase Equipo pero en la práctica eso haría que tengamos que tener un arreglo de jugadores sin equipo para mantener "vivas" a las instancias de jugadores que eliminemos de los arreglos de cada equipo. Además de que los arreglos son de tamaño fijo y se complicaría un poco más si queremos agregar más jugadores que el tamaño original del array.*

Agregación

La agregación es una asociación en la que el ciclo de vida de un objeto no depende del otro.

En Java esto sucede "por defecto", es decir, no hace falta agregar ninguna lógica nueva a nuestras clases Jugador y Equipo en el ejemplo anterior para que la asociación sea de agregación.

Composición

La composición es una asociación en la que el ciclo de vida de una clase depende del otro. Los objetos contenidos no pueden existir independientemente del objeto contenedor. Si el objeto contenedor se destruye, los objetos contenidos también se destruyen.

En nuestro ejemplo los jugadores pertenecen a un Equipo, por lo tanto, el objeto contenedor es Equipo, pero nosotros por razones de practicidad decidimos hacer la relación con un atributo *equipo* dentro de *Jugador*, en lugar de hacer un atributo array de *jugadores* dentro de la clase Equipo.

Si quisiéramos hacer que la asociación sea de composición, deberíamos implementar una lógica en nuestro programa que se asegure de eliminar los objetos "contenidos" cuando se elimina el objeto "contenedor". Pero la lógica de las clases modelo se mantendrá igual.

Veamos un ejemplo:

```
public class Jugador {
    private String nombre;
    private Equipo equipo;

    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public Equipo getEquipo() {
```

```
        return equipo;
    }
    public void setEquipo(Equipo equipo) {
        this.equipo = equipo;
    }
}
```

```
public class Equipo {
    private String nombre;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

```
public class Application {

    private Scanner pepe = new Scanner(System.in);
    private Equipo[] equipos = new Equipo[10];
    private Jugador[] jugadores = new Jugador[110];

    public static void main(String[] args) {
        menu();
    }

    public static void eliminarEquipo() {
        Equipo equipo = seleccionarEquipo();
        eliminarJugadoresConEquipo(equipo);
        for (int i = 0; i < equipos.length; i++) {
            if (equipos[i].equals(equipo)) {
                equipos[i] = null;
            }
        }
        ordernarArray(equipos);
    }

    public static void eliminarJugadoresConEquipo(Equipo equipo) {
```

```
        for (int i = 0; i < jugadores.length; i++) {
            if (jugadores[i].getEquipo.equals(equipo)) {
                jugadores[i] = null;
            }
        }
        ordernarArray(jugadores);
    }
    // ...
}
```

En el ejemplo vemos una parte del código en donde se implementa la lógica para que la relación sea de composición por medio del método *eliminarJugadoresConEquipo()*.