1. Refuerzo de Java

Parte I. Relaciones entre clases

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

1. Introducción

En esta unidad, se revisarán algunos conceptos básicos orientados a objetos que deberías haber aprendido durante el primer curso, por ejemplo, la herencia, composición, polimorfismo, interfaces, gestión de colecciones, etc, junto con algunos conceptos avanzados aportados por las últimas versiones de Java, como expresiones lambda y flujos.

Cuando se define un proyecto, se necesitan crear algunas clases que representen las diferentes entidades que componen el programa. Esas clases, o entidades, tienen que interactuar entre sí (comunicarse) de diferentes formas. Algunas de ellas serán atributos de otras clases, mientras que otras clases compartirán una estructura común, con sus diferencias particulares. En la primera parte de esta unidad, se verán algunas relaciones diferentes que se pueden establecer entre las clases de un proyecto, y también se aprenderá cuándo aplicar cada tipo de relación.

A continuación, la segunda parte de la unidad, se centrará en tres conceptos:

- En primer lugar, se verá cómo tratar las colecciones en Java. Se explicarán varios tipos de colecciones (listas, mapas y conjuntos) y diferentes implementaciones de cada tipo. Se practicará con un par de ejercicios para que aprendas cuándo usar cada tipo de colección.
- Luego, se hará una descripción general de la gestión de excepciones. Este es un poderoso método de manejo de errores, en el que se capturarán los diferentes errores que puede tener un programa, y luego se controlará el mensaje de error que produce el programa.
- Finalmente, se aprenderán diferentes formas de leer y escribir información desde/hacia archivos, y algunas clases útiles para explorar la estructura de directorios u obtener información de archivos.

Para finalizar esta unidad, en su tercera parte se verán algunas características interesantes de Java relacionadas con la programación funcional, como expresiones lambda y *streams*.

Además, hay más contenido que se cubrirá en los anexos proporcionados: uso de enumeraciones, anotaciones, nuevas clases como ensambladores de cadenas, nuevas I/O y funcionalidades de colección…

2. Asociación entre clases

La asociación es una relación entre dos clases, en la que una de ellas utiliza la otra, es decir, un objeto de una de las clases es un atributo de la otra clase. Por lo general, se representa en el código con una referencia al objeto contenido, o una colección de esos objetos. En el siguiente ejemplo, podemos ver una clase ***Classroom*** que contiene / usa una colección (Lista) de objetos del tipo ***Student***:

```
public class Classroom {
    private List<Student> studentList = new ArrayList<>();
}
```

Se puede establecer una relación de **Tiene** entre estas dos clases. En este ejemplo, se podría decir que una ***Classroom*** tiene una lista de ***Students***, definiéndose entre ellas una agregación o composición.

2.1. Composición vs agregación

Existen dos tipos especiales de asociaciones: composiciones y agregaciones. En ambos, una de las clases se considera como un todo y la otra es parte de ese todo. Pero… ¿cómo distinguir entre composición y agregación? Veámoslo con un ejemplo sencillo:

- **Composición**: se utiliza cuando un objeto es parte indivisible de otro objeto. Por ejemplo, un habitación (*Room*) es parte de un casa (*House*), y solo de esa casa, un cuadrado es parte de un tablero de ajedrez, etcétera. La característica principal de este tipo de relación es que cuando se destruye el objeto principal, también se destruyen todos los objetos que están relacionados con él (composición). Esta relación se representa así en un diagrama de clases:

- **Agregación**: se usa cuando un objeto es parte de otro objeto (o tal vez parte de dos o más objetos) y puede existir sin el objeto que lo contiene. Un ejemplo de esto sería un jugador (*Player*), quien es parte de un equipo (*Team*), o tal vez más, o un estudiante, quien pertenece en un aula (o más). En estos casos, cuando el equipo o el aula ya no existe, los jugadores y los estudiantes continúan existiendo y pueden unirse a otro equipo / aula. Se representa esta relación así en un diagrama de clases:

2.2. Composición y agregación en la práctica

En la práctica, la forma en que se defina la agregación o composición depende del lenguaje de programación que se esté utilizando. Pero, en general, si no se puede acceder al atributo interno que hace la composición o agregación desde fuera de la clase que lo contiene, entonces tenemos una composición. De lo contrario, tenemos una agregación. Observa el siguiente ejemplo:

Se define una clase coche (*Car*) que tiene un objeto de tipo motor (*Engine*). Si se quiere definir una composición entre estas dos clases, se haría de esta forma:

```
class Car {
    private final Engine engine;

    Car(EngineParams params) {
        engine = new Engine(params);
    }
}
```

Ten en cuenta que se crea el objeto *Engine* dentro de la clase *Car* mediante el uso de algunos parámetros especificados en el objeto *EngineParams*. En este caso, si el objeto *Car* es destruido, entonces el objeto *Engine* también será destruido. Por tanto, esta es una composición.

Sin embargo, si se necesita definir una agregación entre la clase *Car* y la clase *Engine*, se hará de la siguiente forma:

```
class Car {
    private Engine engine;

    void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

En este caso, se está usando un objeto externo de tipo *Engine* para crear el objeto interno *Engine* del coche, por lo que el motor puede existir sin el coche: si se destruye el coche, el motor externo que usamos en el constructor seguirá existiendo. Esto puede resultar útil si se quiere utilizar el motor en otro coche, una vez destruido el anterior.

3. Herencia y polimorfismo

Se utilizará la **herencia** cuando se quiera crear una nueva clase que toma todas las características de otra, agregando sus particulares. Por ejemplo, si tenemos una clase **Animal** con un conjunto de atributos (nombre, peso …) y métodos, se puede heredar de ella para crear una nueva clase llamada **Perro**, que también tendrá todas estas características, y se pueden agregar algunas adicionales, como un método *ladrar().*

3.1. Herencia básica en Java

Cuando se desee que una clase herede las características (públicas y protegidas) de otra clase, se utilizará la palabra reservada ***extends***, la clase hija ***extends*** (hereda) de la clase padre. Fíjate en el siguiente ejemplo:

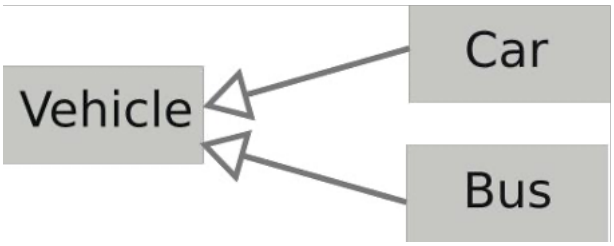
```
public class ComputerClassroom extends Classroom {

}
```

Todos los atributos y métodos (públicos y protegidos) de la clase principal se heredan y pueden ser utilizados por la clase descendiente. Por ejemplo, si se tiene una clase tienda (*Store*) con un método de bienvenida, se puede heredar de esta clase y utilizar este método:

```java
public class Store {
    public void welcome() {
        System.out.println("Welcome to our store!");
    }
}

public class LiquorStore extends Store {
}

// MAIN
LiquorStore lqStore = new LiquorStore();
lqStore.welcome();

// Result
// “Welcome to our store!"
```

3.1.1. Cómo identificar la herencia. Relación Es-Un

En secciones anteriores se ha visto cómo identificar una composición o agregación, encontrando una relación **Tiene** entre las clases involucradas. Cuando se habla de herencia, se identifica con una relación **Es-Un**, por lo que una clase es un subtipo de otra clase. En otras palabras, comparte las características del antepasado e introduce algunas nuevas. Un ejemplo de esto es un coche, que **es un** subtipo de vehículo. Otro podría ser un mamífero, que **es un** subtipo de animal. O un aula de informática, que **es un** subtipo de aula que también tiene ordenadores. Esta es la forma de representar esta relación en un diagrama de clases:



3.1.2. Sobrecargar métodos y llamar a la clase principal

Respecto a la herencia, se puede **sobrecargar** (*override*) un método de la clase padre en la clase hija, lo que significa que se puede redefinir su funcionalidad en lugar de mantener la misma que la del padre. Aún así, se puede llamar al método del padre original desde el método de la clase hija, o llamar al constructor del padre, usando la palabra reservada ***super***.

```java
public class LiquorStore extends Store {
   @Override
   public void welcome() {
      System.out.println("If you are younger than 18, go back home!");
   }
}

// MAIN
LiquorStore lqStore = new LiquorStore();
lqStore.welcome();

// Result:
// “Welcome to our store!” → super method call
// "If you are younger than 18, go back home!"
```

Ten en cuenta que utiliza una palabra llamada ***@Override***. Esta anotación indica que hay un método llamado ***welcome*** en la clase padre, y se queire redefinir su comportamiento. Puedes obtener más información sobre las anotaciones en el **Anexo I**.

A continuación, puedes ver otro ejemplo llamando a ***super*** desde el constructor:

```
public class Classroom {
    private int maxStudents;

    public Classroom(int maxStudents) {
        this.maxStudents = maxStudents;
    }
    public int getMaxStudents() {
        return maxStudents;
    }
}

public class ComputerClassroom extends Classroom{
    private int numComputers;

    public ComputerClassroom(int maxStudents, int numComputers) {
        super(maxStudents);
        this.numComputers = numComputers;
    }
    public int getNumComputers() {
        return numComputers;
    }
}

// MAIN
ComputerClassroom compClass = new ComputerClassroom(30, 20);
System.out.println("Students max.: " + compClass.getMaxStudents()
        + ", computers: " + compClass.getNumComputers());

// Result:
// "Students max.: 30, computers: 20"
```

3.1.3. Herencia múltiple

Una clase solo puede heredar de una clase principal en Java. Otros lenguajes como C++ permiten que una clase herede de varias clases al mismo tiempo. Se puede pensar que una clase solo puede ser un subtipo de otra clase, y esa es la base de la herencia única. Sin embargo, la herencia múltiple deja la puerta abierta a clases que pueden ser subtipos de varias clases.

Por ejemplo, la clase **MesaDeMadera** podría ser un subtipo de **Muebles**, pero también un subtipo de **CosasDeMadera**. En Java, se tendrá que elegir qué clase (Muebles o CosasDeMadera) será la clase padre, mientras que en C++, se podría heredar de ambas. Sin embargo, se verá más adelante que queda una puerta trasera para permitir un tipo de herencia múltiple: interfaces.

3.2. Clases abstractas

No todos los métodos tienen que implementarse en una clase. Se puede definir una clase abstracta que no se puede instanciar y tenga uno o más métodos abstractos, que no tienen implementación dentro de ellos. Una clase que hereda de una clase abstracta debe implementar sus métodos abstractos. De lo contrario, también debe ser una clase abstracta.

```java
public abstract class Store {
    public abstract void welcome();
}

public class LiquorStore extends Store {
    @Override
    public void welcome() {
        System.out.println("Welcome to our liquor store. "
            + "If you are younger than 18, go back home!");
    }
}

// MAIN
Store store = new Store(); → ERROR
LiquorStore liqStore = new LiquorStore(); → OK
```

3.3. Interfaces

Si bien en Java no se puede heredar de más de una clase al mismo tiempo, aún se puede implementar algún tipo de herencia múltiple usando ***interfaces***. Una interfaz es como una clase abstracta, pero con algunas restricciones. Solo pueden contener métodos públicos abstractos (***public abstract***) y constantes estáticas (aunque no se utilizará la palabra *abstract*, ya que todos los métodos son abstractos por definición dentro de una interfaz).

```java
public interface ISayHello {
   public void sayHello();
}

public interface ISayGoodbye {
   public void sayGoodbye();
}
```

Cuando se desea heredar de una interfaz, se utiliza la palabra ***implements*** antes del nombre de la interfaz y después la cláusula ***extends*** (si la hubiera). Se puede implementar más de una interfaz utilizando la coma como separador.

```java
public class Student implements ISayHello, ISayGoodbye {

    @Override
    public void sayGoodbye() {
        System.out.println("Hello, I'm a student!");
    }

    @Override
    public void sayHello() {
        System.out.println("Goodbye people!");
    }
}
```

Como sucede con las clases abstractas, no se puede instanciar un objeto desde una interfaz. Se debe implementar en una clase no abstracta para poder instanciar un objeto que tenga sus métodos.

```java
ISayHello hello = new IsayHello(); → ERROR
Student st1 = new Student(); → OK
```

Al implementar una interfaz, deben implementarse todos sus métodos. Se puede decir que una interfaz define un comportamiento para todas las clases que lo implementan.

Además, una interfaz puede heredar los métodos de otra interfaz (usando la palabra ***extends*** en este caso), para añadir más métodos abstractos a implementar:

```
public interface ISayThings extends ISayHello, ISayGoodbye {
    // Includes methods from ISayHello and ISayGoodbye.
}
```

3.3.1. Interfaces funcionales

Una **interfaz funcional** es una interfaz con un solo método para implementar. Normalmente están marcados con la anotación ***@FunctionalInterface*** para que el compilador sepa que únicamente contiene un método. Por ejemplo, la interfaz *java.util.Comparator* es una interfaz funcional:

```
@FunctionalInterface
public interface Comparator<T>
```

Más adelante se verá que este tipo de interfaces están estrechamente relacionadas con las expresiones lambda.

3.4. Clases anónimas

Para simplificar las cosas, cuando solo se quiere implementar uno o dos métodos de una clase o interfaz abstracta y nada más, no es necesario crear una clase (y su archivo) solo para hacer esto. Desde Java 7, se puede utilizar lo que se conoce como una clase anónima. En lugar de declarar una nueva clase, se instanciará directamente un objeto basado en la interfaz o clase que se quiera extender y, se implementarán todos los métodos obligatorios y extras que se necesitan en la instanciación misma.

```
// MAIN
ISayThings sayThings = new ISayThings() {
    private String name = "Peter";

    @Override
    public void sayHello() {
        System.out.println("Hello, my name is " + name);
    }
    @Override
    public void sayGoodbye() {
        System.out.println("Goodbye friends!");
    }
};
```

También se puede crear una instancia de una clase anónima para implementar métodos que faltan en una clase abstracta (los otros métodos permanecerán como están definidos, a menos que se anulen):

```
public abstract class Store {
    public abstract void welcome();
}
// MAIN
Store store = new Store() {
   @Override
   public void welcome() {
      System.out.println("Welcome to our anonymous store!");
   }
};
```

Ejercicio 1

Crea un proyecto llamado "Stores", e implementa las siguientes clases:

- Una clase abstracta llamada ***Store*** como el que se muestra en la subsección 3.2. Tendrá un método abstracto llamado ***welcome***, dos nuevos campos privados (***cash*** → *double* y ***drinkPrice*** → *double*) y un constructor que recibirá el *drinkPrice* e inicializará el *cash* a 0.0. También deberás implementar un método llamado ***payDrinks(int numOfDrinks)*** que sumará a la variable *cash* el pago (número de bebidas * precio).
- Una clase llamada ***LiquorStore*** que extenderá de la clase *Store*. Implementará el método ***welcome*** como se muestra en la subsección 3.2, y tendrá un nuevo campo privado llamado ***tax*** → *int*. Su constructor recibirá los valores de *drinkPrice* y *tax*, llamando al constructor del padre cuando sea necesario. Esta clase también sobrecargará ***payDrinks(int)*** llamando primero al método del padre (pagar sin impuestos) y después añadir el valor del impuesto al campo de *cash*.
- Una clase principal con el método ***main***. Este método deberá:
  - Crear un objeto *LiquorStore* con *drinkPrice* = 8,95 € y *tax* = 20%. Pagará 10 bebidas e imprimirá el efectivo para ver si su valor es 107,40 € (imprime 2 números decimales).
  - Crear una instancia *Store* usando una clase anónima. El precio de la bebida será de 8,95 € y el método de bienvenida dirá "Welcome to anonymous store!, Our drink price is XX €" (donde XX será el valor del atributo *drinkPrice*, con 2 decimales). Llamará al método *welcome* y también se pará por 10 bebidas. El efectivo resultante debe ser 89,50 €).
- Implementa *getters* y *setters* cuando sea necesario.

3.5. Polimorfismo

En la Programación Orientada a Objetos, el **polimorfismo** puede significar cosas diferentes, pero en este caso se hablará de **polimorfismo de subtipo**. Esto significa que un objeto puede comportarse como si fuera una instancia de una clase extendida o una interfaz implementada. Luego, un objeto tienda (*Store*) se puede instanciar como un subtipo tienda de licores (*LiquorStore*)… pero solo se podrá llamar a los métodos que estén presente en el tipo padre.

```
public abstract class Store {
    public abstract void welcome();
}

public class LiquorStore extends Store {

   @Override
   public void welcome() {
      System.out.println("Welcome to our liquor store. "
            + "If you are younger than 18, go back home!");
   }

   public void buyLiquor() {
      System.out.println("Do you want beer, wine, or vodka?");
```

```
   }
}

// MAIN
Store store = new LiquorStore();
store.buyLiquor(); // ERROR. We only can access Store methods
store.welcome(); // OK, and executes LiquorStore implementation
```

Se puede convertir una variable que hace referencia a una clase extendida a su clase original o cualquier otra clase extendida o interfaz implementada en cualquier momento. Se puede usar la instrucción ***instanceof*** para averiguar si realmente es una instancia del subtipo que creemos que es.

```
if (store instanceof LiquorStore) {
   ((LiquorStore) store).buyLiquor(); // OK
}
```

Si el parámetro de un método es una interfaz o una clase de la que heredan otras clases, se puede pasar cualquier clase de subtipo como valor. Se deberá tener en cuenta que solo se podrá utilizar el comportamiento de la interfaz o la super-clase (a menos que se haga un *casting* como antes):

```
public class Student implements ISayHello, ISayGoodbye {
   … // Implemented methods
    public void enterStore(Store store) {
        sayHello();
        store.welcome();
    }
}

// MAIN
LiquorStore liqStore = new LiquorStore();
Student student = new Student();
student.enterStore(liqStore); // Will be treated as a Store inside
```

Ejercicio 2

Crea un proyecto llamado ***KillEnemies***. Deberás crear lo siguiente en su interior:

- Interfaz ***Character***: Con un método llamado *isEnemy()* que devolverá un booleano.
- Clase ***Friend***: implementará *Character*. *IsEnemy()* devolverá falso.
- Clase ***Enemy***: implementará *Character*. *IsEnemy()* devolverá verdadero. También deberás crear un método llamado *kill()* que mostrará el mensaje *"Ahhhggg, you killed me, bastard!"*.
- La **clase principal** contendrá el método ***main***. Deberás crear un ***ArrayList*** de 10 personajes (5 amigos y 5 enemigos), luego, usando ***Collections.shuffle(List)***, desordenarás el orden de los elementos. Tendrás que recorrer todos los personajes comprobando si son enemigos, y si lo son, los matas (llamando al método *kill()*). Un ejemplo de la salida podría ser:

```
Character 0 is an enemy! kill it!
Ahhhggg, you killed me, bastard!
Character 1 is an enemy! kill it!
Character 2 is a friend! :-)
…
```

3.6. Usar genéricos

Los genéricos es un concepto con el que ya deberías estar familiarizado. Por ejemplo, al utilizar cualquier tipo de colección, como un ***ArrayList***, te permitirá crear una lista para almacenar objetos de algún tipo específico. Deberás especificar el tipo de objetos cuando crees la variable:

```
ArrayList <Store> stores ; // Holds objects which are a Store or inherit from it
```

Esto se logra con genéricos. Un genérico es un tipo especial que representa una clase o interfaz desconocida. El compilador solo lo sabrá cuando se instancie la variable que tenga un genérico, en ese momento deberás especificar el tipo del genérico.