1. Refuerzo de Java

Anexo I. Enumeraciones, anotaciones y más sobre herencia

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

1. Introducción

En este anexo se verán algunos conceptos adicionales relacionados con las relaciones entre clases. En primer lugar, se obtendrá una descripción general rápida de dos conceptos sencillos: cómo establecer enumeraciones en Java y qué son las anotaciones y cómo usarlas.

Luego, se explorarán algunos de los conceptos vistos en esta parte de la unidad sobre relaciones entre clases, como las clases, herencia, clases abstractas, interfaces o genéricos. Se creará un ejemplo y se verá cómo implementarlo utilizando diferentes enfoques, luego aprenderás cuándo usar cada uno.

2. Enumeraciones y anotaciones

2.1. Enumeraciones en Java

Cuando se necesita usar un conjunto de constantes como los nombres de los meses, días de la semana, colores, etc, es mejor usar una enumeración que, por ejemplo, una lista, o crear una cadena o constantes enteras. Las enumeraciones se pueden iterar como una lista y son un conjunto limitado de valores de solo lectura.

En Java, las enumeraciones son clases. Eso significa que pueden tener un constructor (siempre privado, por lo que funciona como *Singleton* y no puede crear más de una instancia a la vez → https://es.wikipedia.org/wiki/Singleton) y métodos. También pueden implementar interfaces (y todos sus métodos asociados).

En una enumeración, todos los valores deben estar definidos al principio de la misma, separados por comas y generalmente en mayúsculas (son como constantes).

```
public enum MyColours {
   RED, GREEN, BLUE, BLACK, WHITE;
}
```

Como ya se ha comentado, se pueden implementar un constructor y diferentes métodos. Se deberá llamar al constructor al definir los valores, en cada uno (cada valor será una instancia de la enumeración):

```
public enum MyColours {
   RED("FF0000"),
   GREEN("00FF00"),
   BLUE("0000FF"),
   BLACK("000000"),
   WHITE("FFFFFF");

   private final String rgbValue;

   private MyColours(String rgbValue) {
      this.rgbValue = rgbValue;
   }

   public String getRgbValue() {
      return rgbValue;
   }
}
```

Se puede acceder a un valor y sus métodos de la siguiente manera:

```
System.out.println(MyColours.RED.getRgbValue());
//Will print → FF0000.
```

Puedes usar el método ***values()*** para obtener un *array* e iterar a través de todos los valores:

```
for(MyColours col : MyColours.values()) {
   System.out.println(col.name() + " -> " + col.getRgbValue());
}
```

2.2. Anotaciones

Las anotaciones son meta-instrucciones y no se compilan como el resto del código. Estas proporcionan información útil al compilador y a diferentes herramientas como: herramientas de generación automática de código, herramientas de documentación (JavaDoc), *Testing Frameworks* (JUnit,…), ORM (Hibernate,….), etc.

Se colocan antes de una clase, un campo o la definición de un método, afectando a lo que sigue. Empiezan con el carácter '@'. Algunos ejemplos de anotaciones que interpreta el compilador son:

- **@Deprecated** → Se aplica a un método y muestra una advertencia cada vez que un programador quiere usar este método en su código, indicándole que debe dejar de utilizarlo y usar alguna funcionalidad nueva en su lugar.
- **@Override** → Informa al compilador que el método está sobrecargado. No es necesario usar esta anotación, pero si lo haces y el método no existe en la clase padre (no sobrecargado), el compilador mostrará un error. Se han visto algunos ejemplos de esta anotación hablando de herencia.
- **@FunctionalInterface** → Informa al compilador que la interfaz que tiene esta anotación es funcional y, por lo tanto, solo tiene un método para implementar.
- **@SuppressWarnings("type")** → Desactiva un tipo de advertencia dentro de un método, algunos ejemplos de tipos de advertencia son *"unused"* (variables no utilizadas), *"deprecated"* (uso de métodos obsoletos), …

3. Más sobre la herencia

Se acaba de hablar de algunos conceptos básicos en materia de herencia, como clases padre, clases hijo, clases abstractas, interfaces ... y algunos otros conceptos avanzados como clases anónimas o genéricos. Para acostumbrase a estos conceptos, se verá cómo aplicarlos con un ejemplo sencillo.

3.1. Presentando el ejemplo

Se trabajará con varios tipos de formas geométricas. Se definirá una clase padre o una interfaz llamada *Shape*, con una estructura básica, y luego se implementarán algunas subclases (formas geométricas concretas), y se calculará el área de cada una de ellas.

3.2. Utilizar clases “normales”

La primera, y quizás la forma más fácil para resolver el problema es crear clases “normales”. En primer lugar, se definirá una clase padre llamada ***Shape*** con un atributo protegido llamado ***area*** (*float*), y un método llamado ***getArea()*** que calculará el área de la forma:

```
public class Shape {
   protected double area;

   public Shape() {
   }

   public double getArea() {
      return 0;
   }
}
```

Luego, se definirán tres tipos de forma: *Triangle*, *Rectangle* y *Circle*, en tres clases diferentes que heredarán de la clase *Shape*, añadiendo los atributos correspondientes y sobrecargando el método *getArea()*.

La clase *Triangle* tendrá dos nuevos atributos: *base* y altura (*height*). El área será calculada como *base * altura / 2*:

```
public class Triangle extends Shape {
    private double height, base;

    public Triangle(double height, double base) {
        this.height = height;
        this.base = base;
    }

    @Override
    public double getArea() {
        return height * base / 2;
    }
}
```

La clase *Rectangle* también tendrá los atributos *base* y altura (*height*), y el área se calculará como *base * altura*:

```
public class Rectangle extends Shape {
    private double height, base;

    public Rectangle(double height, double base) {
        this.height = height;
        this.base = base;
    }

    @Override
    public double getArea() {
        return height * base;
    }
}
```

La clase *Circle* tendrá el radio (*radius*) como único atributo, y su área se calculará como *PI * radio*^{2}:

```java
public class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

Se podrían probar estas clases en un programa principal. Se puede utilizar algún polimorfismo para verificar si los resultados son los esperados:

```java
public class Main {

    public static void main(String[] args) {
        Shape s = new Rectangle(4, 2);
        Circle c = new Circle(5);
        Triangle t = new Triangle(4, 2);

        System.out.printf("Shape area: %.2f\n", s.getArea());
        System.out.printf("Circle area: %.2f\n", c.getArea());
        System.out.printf("Triangle area: %.2f\n", t.getArea());
    }
}
```

Si ejecutas la aplicación, verás un resultado como este en la consola:

```
Shape area: 8,00
Circle area: 78,54
Triangle area: 4,00
```

En las siguientes secciones, se mejorará este proyecto añadiendo algunas características nuevas.

3.3. Usando una clase abstracta

Si te fijas en la clase *Shape* del ejemplo anterior, no sirve de nada implementar el método *getArea()* en ella, ya que no se sabe con qué forma se trabajará y, por tanto, no se sabe qué valor debe devolverse, por eso se devuelve 0.

Se puede definir la clase *Shape* como abstracta, y luego establecer el método *getArea()* como un método abstracto que debe implementarse en las subclases. La nueva clase *Shape* quedaría así:

```java
public abstract class Shape {
    protected double area;

    public Shape() {
    }

    public abstract double getArea();
}
```

El resto de clases seguirán igual. La única diferencia con el ejemplo anterior es, que ahora no puede crearse un objeto de la clase *Shape*, ya que es abstracta:

```
Shape s = new Shape(); // ERROR
```

Por lo general, las clases abstractas son muy útiles cuando se tienen varios subtipos de objetos que implementan un método que no tiene solución en la clase padre. Por ejemplo, si se tiene un método llamado *hablar()* en una clase *Animal*, no se sabría qué poner hasta que se sepa el tipo concreto de animal que se está instanciando. Si es un *Perro*, entonces se diría “Gua!”, si es un gato, se diría “Miau!”, etc. Por tanto, el método *hablar()* sería abstracto en la clase *Animal*, y se implementaría dependiendo del tipo concreto de animal en las subclases.

3.4. Usando una interfaz

Observa de nuevo la clase *Shape*. El atributo *area* no es útil, porque se calcula el área automáticamente en cada método *getArea()* de cada subclase. Si no hay atributos en la clase padre, y no se necesita tener ningún método implementado, se puede cambiar esta clase a una interfaz, de modo que solo se pongan los métodos abstractos en ella. Y ahora la clase *Shape* puede ser una interfaz, y se vería así (cambia el nombre a *IShape* en el proyecto, para que se pueda preservar la clase abstracta *Shape* antigua para recordarlo):

```
public interface IShape {
    public double getArea();
}
```

Todas las subclases deberán ahora implementar esta interfaz, en lugar de extender de la clase *Shape*:

```
public class Triangle implements IShape {
    // The rest remains the same.

public class Rectangle implements IShape {
    // The rest remains the same.

public class Circle implements IShape {
    // The rest remains the same.
```

En este caso, se está ahorrando algo de código, ya que no se necesita un constructor. En la aplicación principal, se puede instanciar cada tipo de objeto y calcular su área. Si se quiere crear una instancia del objeto *IShape*, deberá hacerse desde una de sus clases implementadas:

```
public class Main {

    public static void main(String[] args) {
        IShape s = new Rectangle(4, 2);
        Circle c = new Circle(5);
        Triangle t = new Triangle(4, 2);

        System.out.printf("Shape area: %.2f\n", s.getArea());
        System.out.printf("Circle area: %.2f\n", c.getArea());
        System.out.printf("Triangle area: %.2f\n", t.getArea());
    }
}
```

3.5. Usando clases anónimas

Las clases anónimas son muy útiles cuando no quieres crear un archivo fuente separado para una clase que implementa una interfaz o extiende una clase abstracta. En este caso, se puede simplemente crear una instancia de la clase abstracta, o interfaz, cuando quieras utilizarla e implementar los métodos abstractos dentro del bloque de código.

Fíjate en la clase principal creada. Se ha definido un objeto de cada tipo (rectángulo, triángulo y círculo) y se calculan sus áreas. ¿Qué pasa si se quiere definir un solo objeto de otro tipo (por ejemplo, un cuadrado)? Se podría definir una clase *Square* con su propio método *getArea()*, y luego instanciar un objeto *Square*, o utilizar una clase anónima en el método *main* de la aplicación con un método *getArea()* que calcule el área del cuadrado (*lado * lado*):

```
public class Main {

    public static void main(String[] args) {
        IShape s = new Rectangle(4, 2);
        Circle c = new Circle(5);
        Triangle t = new Triangle(4, 2);

        IShape square = new IShape() {
            double side = 3;

            @Override
            public double getArea() {
                return side * side;
            }
        };

        System.out.printf("Shape area: %.2f\n", s.getArea());
        System.out.printf("Circle area: %.2f\n", c.getArea());
        System.out.printf("Triangle area: %.2f\n", t.getArea());
        System.out.printf("Square area: %.2f\n", square.getArea());
    }
}
```

Ten en cuenta que se acaba de definir un tipo de clase pequeña dentro del *main*. Tiene su propio atributo (*side*) y sobrecarga el método *getArea()*. Esta clase no tiene nombre (no se ha escrito ninguna *"public class YYY"*), y solo se puede usar una vez (si se necesita definir otro cuadrado con una longitud de lado diferente y calcular el área, se tendría que crear otra clase anónima con sus propios atributos y método *getArea()*).

3.5.1. ¿Cuándo usar clases "normales" o clases anónimas?

A partir del ejemplo anterior, es posible que hayas deducido que a veces es mejor crear una clase "normal" y, a veces, es mejor usar una clase anónima. ¿Cuándo usar cada una?

- Se usaran clases "normales" para implementar una interfaz o extender una clase abstracta cuando se necesite usarla en más de un lugar del código. En este caso, usar clases anónimas significaría repetir código muchas veces a lo largo del programa.
- Por otro lado, cuando solo se necesita usar una instancia de una clase abstracta, o interfaz, en un lugar concreto del código una vez, se pueden usar clases anónimas.

3.6. Más sobre genéricos

Al visitar la página web de referencia para ArrayList de Java …

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

… verás que esta clase se define como ***ArrayList<E>***. El símbolo ***<E>*** es una notación utilizada para definir una clase genérica, una clase que el compilador no conoce hasta que se instancie el objeto. Otras notación comunes son ***<T>***, ***<S>***, ***<U>***, etc. En general, una letra mayúscula entre **< >**.

Si se crea un genérico, por defecto puede ser cualquier clase, por lo que las únicas propiedades y métodos que el compilador permitirá usar son los heredados desde la clase ***Object*** (todas las clases en Java heredan de *Object*):

```
public class GenericExample<T> {
    private T generic;

    public GenericExample(T generic) {
        this.generic = generic;
    }

    public void showType() {
        System.out.println(generic.getClass().getName().toString());
        // We can't use for example .substring()
        // since <T> can be anything.
    }

    public T getGeneric() {
        return generic;
    }
}
```

Se define el tipo de <*T*> cuando se instancia un objeto de *GenericExample*:

```
GenericExample<String> genEx = new GenericExample<>("Hello world!");
genEx.showType(); // → java.lang.String
/* Out of the class, in this context we can use a String method with the generic
object because the compiler knows that the generic is a string */
System.out.println(genEx.getGeneric().length());
```

Se puede especificar que el genérico debe ser un subtipo de clase o implementar una interfaz. En este caso, como ocurre con el polimorfismo, se puede utilizar la super-clase o los métodos de la interfaz:

```
public class GenericExample<T extends Store> {
    …
    public void welcome() {
        generic.welcome(); // generic can use Store methods
    }
    …
}

// MAIN
GenericExample<String> genEx = new GenericExample<>("Hello world!"); // ERROR
GenericExample<LiquorStore> genEx = new GenericExample<>(new LiquorStore()); // OK
```

Se puede utilizar más de un genérico en una clase:

```
public class GenericExample<T extends Store, E extends Classroom> {
}
```

3.6.1. Un ejemplo introductorio

Imagina que tienes una clase llamada *Inventory* que puede almacenar elementos (cualquier objeto que herede de la clase *Item*). Observa la diferencia en este caso entre usar genéricos para definir esa clase y usar polimorfismo.

La apariencia general de la clase *Item* y algunas de sus subclases serían así:

```
public class Item {
    private float price;
    private int weight;

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }
}

public class Potion extends Item {
    public void drink() {
        System.out.println("Gulp gulp gulp.");
    }
}

public class Weapon extends Item {
    private int damage;

    public int getDamage() {
        return damage;
    }

    public void setDamage(int damage) {
        this.damage = damage;
    }
}
```

Si se utiliza polimorfismo para tratar con una lista de objetos *Item* en la clase *Inventory*, se tendría algo como esto:

```
public class Inventory {
    private List<Item> items = new ArrayList<>();

    public void addItem(Item item) {
        items.add(item);
    }

    public Item getItem(int index) {
        return items.size() > index?items.get(index):null;
```

```
    }
}

// MAIN
Inventory inv = new Inventory();
inv.addItem(new Potion()); // Index 0
inv.addItem(new Weapon()); // Index 1

// returns an Item, usually we don't know which type
Item it = inv.getItem(0);

if(it instanceof Potion) {
    ((Potion) it).drink();
} else if(it instanceof Weapon) {
    System.out.println("Damage: " + ((Weapon) it).getDamage());
}
```

Si se utilizan genéricos para manejar la misma lista, se obtendría esto:

```
public class Inventory<T extends Item> {
    private List<T> items = new ArrayList<>();

    public void addItem(T item) {
        items.add(item);
    }

    public T getItem(int index) {
        return items.size() > index?items.get(index):null;
    }
}

// MAIN
Inventory<Item> inv = new Inventory<>(); // Same behaviour as before!
Inventory<Potion> potInv = new Inventory<>(); // Only allows potions
potInv.addItem(new Potion()); // OK
potInv.addItem(new Weapon()); // ERROR, <T> must be a Potion

Potion pot = potInv.getItem(0); // Compiler knows is a Potion. No casting needed.
```

En resumen, siempre que se quiera poder usar más de un tipo de objeto dentro de una instancia de clase, se puede usar polimorfismo (un inventario con diferentes tipos de elementos), aunque todavía se pueden usar genéricos para esto. Cuando se quiera la posibilidad de utilizar solo una clase a la vez, definida cuando se instancia un objeto, solo puede hacerse con genéricos (se puede crear un inventario que solo permita pociones, además de solo armas, etc, usando el misma clase para todos).

También se puede declarar un genérico (o más de uno) solo para un método específico, de la misma manera que se hizo con la clase:

```
public <U extends T> void add(U item) {
    items.add(item);
}
```

3.6.2. De vuelta a nuestro ejemplo

Volviendo al ejemplo de las figuras. Se definirá una clase que use genéricos para comparar dos formas geométricas entre ellas. Imprimirá el área más grande de ambas formas. El código fuente de esta clase sería así:

```
public class ShapeComparator<T extends IShape, U extends IShape> {
   private T shape1;
   private U shape2;

   ShapeComparator(T shape1, U shape2) {
      this.shape1 = shape1;
      this.shape2 = shape2;
   }

   public T getShape1() {
      return shape1;
   }

   public void setShape1(T shape1) {
      this.shape1 = shape1;
   }

   public U getShape2() {
      return shape2;
   }

   public void setShape2(U shape2) {
      this.shape2 = shape2;
   }

   public void compare() {
      double area1 = shape1.getArea();
      double area2 = shape2.getArea();

      if(area1 > area2) {
         System.out.printf("Shape 1 wins: %.2f vs %.2f\n", area1, area2);
      } else if(area2 > area1) {
         System.out.printf("Shape 2 wins: %.2f vs %.2f\n", area2, area1);
      } else {
         System.out.printf("Tie: %.2f\n", area1);
      }
   }
}
```

Ten en cuenta que la clase recibe dos clases que implementan la interfaz *IShape* como parámetros entre los símbolos <...>. En el constructor, se asigna cada parámetro a un atributo interno (*shape1* será de clase *T* o de cualquier subtipo, y *shape2* será de clase *U* o cualquier subtipo). Luego, se definen los *getters* y *setters* de cada atributo, y un método *compare* que compará las áreas de las figuras y mostrará la más grande.

Se pueden utilizar objetos de esta clase en el programa principal para comparar figuras:

```
public class Main {
    public static void main(String[] args) {
        …

        ShapeComparator<Triangle, Rectangle> triRecComp =
                new ShapeComparator<>(
                        new Triangle(9.65, 5.5), new Rectangle(6.7, 5));
        triRecComp.compare();

        ShapeComparator<Rectangle, Circle> recCirComp =
                new ShapeComparator<>(
                        new Rectangle(9.65, 5.5), new Circle(9));
        recCirComp.compare();
    }
}
```

El primer objeto creado en esta prueba (*triRecComp*) solo aceptará objetos de tipo *Triangle* (o cualquier subtipo) como su primer parámetro, y de tipo *Rectangle* (o cualquier subtipo) como segundo. Algo parecido ocurre con el segundo objeto (*recCirComp*): acepta *Rectangle* como primer parámetro y *Circle* como segundo parámetro.

Ejercicio 1

Crea un proyecto llamado ***AnimalConversation*** que contendrá las siguientes clases (incluida una clase principal):

- Una clase abstracta llamada ***Animal***. Tendrá un atributo protegido llamado *name* (*String*, asignada en el constructor) y un método abstracto llamado ***talk()***, que devolverá un *String* con el sonido que produce este animal.
- Las clase ***Dog***, ***Cat*** y ***Sheep*** que heredarán de la clase *Animal* e implementan el método abstracto *talk()* (los perros devolverán *"Wof wof"*, los gatos devolverán *"Meooow"* y las ovejas devolverán *"Beeee"*).
- Una clase llamada ***AnimalConversation*** que contendrá dos objetos (*animal1* y *animal2*), derivados de *Animal*. Utiliza genéricos para definir el tipo de estos dos objetos (cada objeto puede ser de un tipo diferente pero siempre derivado de *Animal*). Esta clase tendrá un método llamado *chat()*, que mostrará en consola el mensaje de *animal1* (llamando a su método *talk()*), y luego de *animal2*.
- En el ***main***, instancia dos o más *AnimalConversations*, cada una para comparar diferentes tipos de animales (uno puede comparar solo perro vs gato, otro gato vs oveja, etc). Según cómo definas los genéricos en cada instancia, solo te permitirá asignar u obtener (implementa *getters* y *setters*) ese tipo de animal y no otros.