Entornos de desarrollo
Bloque 2
Tema 7: Clases abstractas e interfaces

by Nacho Iborra

En este tema vamos a introducir dos conceptos importantes muy relacionados con la herencia: las clases abstractas y los interfaces. Su implementación y uso los practicaremos con algunos ejemplos.

2.7.1. Clases abstractas

Una **clase abstracta** es una clase que no se puede instanciar directamente, es decir, no podemos crear objetos de esta clase porque su código y funcionalidad no están completamente definidos. Por ejemplo, supongamos que tenemos una clase llamada `Animal` y especificamos una serie de atributos de todos los animales (como el color, o el número de patas). Sin embargo, hay algunos atributos o métodos, como `hablar` que no pueden ser especificados a menos que sepamos el tipo de animal concreto del que estamos hablando. En este caso podríamos, o incluso deberíamos definir la clase `Animal` como una clase abstracta.

Para definir una clase abstracta en Java, solo debemos añadir la palabra reservada `abstract` antes de `class`. Se pueden especificar atributos, constructores y otros métodos igual que en cualquier otra clase.

```
public abstract class Animal
{
    String color;
    int numberOfLegs;

    public Animal(String color, int numberOfLegs)
    {
        this.color = color;
        this.numberOfLegs = numberOfLegs;
    }

    public String getColor()
    {
        return color;
    }

    public void setColor(String color)
    {
        this.color = color;
    }

    ...
}
```

Además se puedemn añadir métodos *abstractos* si fuera necesario. Un método abstracto es aquel que no está implementado en la clase abstracta. Se añadirá la palabra `abstract` a la cabecera del método y no se pone código. Por ejemplo, podemos añadir un método abstracto a la clase anterior y llamarlo `talk`:

```
public abstract class Animal
{
    String color;
    int numberOfLegs;

    ...

    public abstract void talk();
}
```

Tened en cuenta que **una clase abstacta no necesariamente debe tener métodos abstractos para ser abstracta**, solo los añadiremos si los necesitamos. En este caso, no podemos hacer a un animal hablar a menos que sepamos exactamente el tipo de animal, pero necesitamos que los animales hablen en cuanto son creados, por eso definimos el método `talk` como abstacto para que los descendientes de esta clase lo implementen.

2.7.1.1. Herencia de una clase abstracta

Para crear subtipos de clases abstractas, solo es necesario heredar de ellas. Estos subtipos pueden ser:

- **abstract**, y por lo tanto debemos seguir añadiendo el modificador `abstract` a la clase, y se pueden incluso añadir más métodos abstractos si quisiéramos. Por ejemplo, podemos definir una subclase llamada `Bird` con el método abstracto `fly`:

```
public abstract class Bird extends Animal
{
    public abstract void fly();
}
```

- **concrete**, DEBEMOS implementar (override) todos los métodos abstractos definidos en las clases padres. En nuestro caso, podríamos definir una subclase `Dog` que herede de `Animal` que necesitará implementar el método `talk`, o una subclase `Duck` que herede de `Bird` y deberá implementar tanto el método `talk` como el método `fly`.

```
public class Dog extends Animal
{
    public Dog(String color, int numberOfLegs)
    {
        super(color, numberOfLegs);
```

```
    }

    @Override
    public void talk()
    {
        System.out.println("Woof woof!!");
    }
}

public class Duck extends Bird
{
    public Duck(String color, int numberOfLegs)
    {
        super(color, numberOfLegs);
    }

    @Override
    public void talk()
    {
        System.out.println("Quack quack!!");
    }

    @Override
    public void fly()
    {
        System.out.println("I'm flying like a duck!");
    }
}
```

2.7.1.2. Clases abstractas y polimorfismo

Hasta ahora solo hemos hablado sobre la imposibilidad de crear objetos de una clase abstracta. Sin embargo, se puede definir un objeto de una clase abstracta a partir de alguna de sus subclases. De esta forma se puede asignar un objeto de alguna de sus subclases a una variable de la clase abstracta. Por ejemplo, fijándonos en el ejemplo anterior, no podemos crear un objeto `Animal` porque es una clase abstracta, pero podemos crear un objeto `Dog` y asignarlo a una variable `Animal` gracias al polimorfismo.

```
Animal a1 = new Animal("red", 2);   // Error!!
Animal a2 = new Dog ("white", 4);   // OK
```

> **Ejercicios propuestos:**
>
> **2.7.1.1.** Crea un proyecto en Eclipse llamado *Animals* con una clase principal llamada *Animals* dentro de un package llamado *animals.main*. Añade las clases de los ejemplos anteriores en un package llamado *animals.types*. Define las clases abstractas *Animal* y *Bird* con sus correspondientes subclases *Dog* y *Duck* y cualquier otra que quieras añadir (como *Cat* o *Lion*,

> por ejemplo). Después, define un array de 5 animales (tipo *Animal*) y completa el array con diferentes animales. Después recorre el array y haz a cada animal hablar (`talk`).
>
> **2.7.1.2.** Vuelve al ejercicio 2.6.4.5 del tema anterior (organización cultural) y haz los siguientes cambios (crea una copia de seguridad antes de realizar los cambios):
>
> - Define la clase `ObjetoCultural` como abstracta
> - En el main, define un array de 6 objetos del tipo `ObjetoCultural` y añade tres libros y tres discos de música. Después, muestra por pantalla el array completo.

2.7.2. Interfaces

Un interface se puede considerar como un tipo especial de clase sin código implementado (en realidad, podemos añadir algo de código dentro, pero no es el objetivo de este tema). Por tanto, no pueden ser instaciados directamente tampoco. Los interfaces se usan para definir un conjunto de métodos que necesitan ser implementados por alguna clase cuando hereda del interface.

Por ejemplo, supongamos que tenemos un interface llamado `Shape` que representa cualquier tipo de forma, como círculos, cuadrados etc. No necesitamos guardar ninguna información específica sobre una forma, pero queremos que todas las formas calculen su area y se dibujen. Se podría definir un interfaz de la siguiente forma:

```
public interface Shape
{
    public float calculateArea();
    public void draw();
}
```

De esta forma cualquier clase que "herede" del interface debe implementar estos dos métodos. En realidad, no se hereda de un interfaz, se **implementa** un interfaz, por eso no se usa la palabra reservada *extends* sino `implements`.

```
public class Circle implements Shape
{
    float radius;

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

    @Override
    public float calculateArea()
    {
        return Math.PI * radius * radius;
    }
```

```
    @Override
    public void draw()
    {
        System.out.println("Drawing a circle!");
    }
}
```

2.7.2.1. Herencia o implementación

Como acabamos de ver, ni las clases abstractas ni los interfaces pueden ser instanciados y ambos pueden tener código sin terminar de implementar. Pero... ¿cómo decidir cuál debemos usar en cada ocasión?

- De las clases abstractas se **hereda**, por eso si nos preguntamos si debemos usar una clase abstracta, necesitamos estar seguros de que:
    - Las subclases que definamos con posterioridad **son subtipos** de la clase abstracta.
    - No necesitaremos heredar de nada más (Java solo deja heredar de una clase).
- Los interfaces se **implementan**, y una clase puede implementar tantos interfaces como sean necesarios y al mismo tiempo heredar de una clase.

```
public class MyShape implements Shape, Comparable extends AnotherClass
{
    ...
}
```

El único inconveniente de los interfaces es la (casi) falta de código: no se pueden definir atributos, constructores u otros métodos. Solo se pueden implementar algunos métodos estáticos y otro código especial, y se deja sin implementar un conjunto de métodos para que sean completados por las clases que implementen el interfaz y **actúe como** el interfaz. En el ejemplo anterior, un círculo *Circle* no se considera una forma *Shape* (porque no hereda de ella), pero se puede decir que un *círculo actúa como una forma*, porque *Circle* implementa el interfaz *Shape*.

> **Ejercicios propuestos:**
>
> **2.7.2.1.** Crea un proyecto en Eclipse llamado *Shapes*, con una clase principal llamada *Shapes* en el *package* llamado *shapes*. Después, añade el interfaz *Shape* a este mismo *package* y todas las clases que lo implementan (como *Circle*, *Rectangle* o *Square*) en un subpackage llamado *shapes.types*. Implementad los métodos *calculateArea* y *draw* en todos ellos (únicamente deben mostrar por pantalla un mensaje en cada método *draw* de cada forma). Define un array con 5 formas (tipo *Shape*) y rellénalo con algunos datos (no necesitas pedírselo al usuario). Finalmente recorre el array y calcula el área de cada forma.

2.7.2.2. Ejemplo: ordenar objetos complejos

Podríamos preguntarnos para qué pueden ser útil el uso de interfaces en el trabajo diario. Supongamos que debemos manejar una lista o array de objetos complejos, como objetos de tipo `Person` que contienen nombres y edades:

```
class Person {

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    ...
}

...

public static void main(String[] args) {
    Person[] people = new Person[50];

    people[0] = new Person("Nacho", 40);
    people[1] = new Person("Juan", 70);
    ...
}
```

¿Y si quisiéramos ordenar este array por la edad de las personas en orden descendente? Podríamos definir un método que ordene este array manualmente con alguno de los métodos que ya conocemos (como burbuja etc)

```
for (int i = 0; i < people.length - 1; i++) {
    for (int j = i + 1; j < people.length; j++) {
        if (people[i].getAge() < people[j].getAge()) {
            People aux = people[i];
            people[i] = people[j];
            people[j] = aux;
        }
    }
}
```

Sin embargo, existe una forma más eficiente de realizar esta ordenación, aunque puede que tengamos que escribir más código que con el ejemplo anterior. Hay un par de interfaces disponibles en el core de Java que nos permite ordenar cualquier tipo de objeto. Estos interfaces son: Comparator and Comparable. Ambos tienen un método a implementar:

- Si elegimos el interfaz `Comparable` debemos implementar un método llamado `compareTo`, que recibe un objeto como parámetro y lo compara con el objeto actual (*this*). También devuelve un entero indicando cual deberá ir primero en el array: el objeto actual (valor negativo), el objeto recibido como parámetro (valor positivo) o cero si son iguales. Como estamos trabajando con *this*, necesitaremos implementar la interfaz en la propia clase cuyos objetos queremos ordenar.

- Con el interfaz `Comparator` se necesita implementar un método llamado `compare`. Recibe dos objetos como parámetros, y devuelve un entero indicando cual debe ir primero en la ordenación: el primer parámetro (valor negativo), el segundo parámetro (valor positivo) o cero si son iguales. Esta interfaz suele implementarse en una clase aparte, que se utilizará para ordenar los objetos de otra clase.

Veamos cómo usarlo con la clase `Person`. Primero, necesitamos implementar el interfaz que hayamos elegido. Si elegimos el interfaz `Comparable` debemos implementarlo en la clase `Person` y en el código del método `compareTo` devolveremos un número dependiendo de qué edad es mayor: necesitamos ordenar por edad en orden descendente de forma que devolveremos un valor negativo si el objeto *this* es mayor que el parámetro recibido:

```
class Person implements Comparable<Person> {

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person p) {
        if (this.getAge() > p.getAge())
            return -1;
        else if (p.getAge() > this.getAge())
            return 1;
        else
            return 0;
    }
}
```

Si echamos un vistazo a la API de la clase Integer, existe un método estático llamado `compare` que recibe dos parámetros enteros y devuelve un entero indicando cuál es menor o mayor (negativo si el primero es menor, positivo si el primero es mayor). Así, podemos aprovecharnos de este método para ordenar las personas por edad. Si queremos ordenarlas en orden ascendente, podemos hacer esto:

```
@Override
public int compareTo(Person p) {
```

```
    return Integer.compare(this.getAge(), p.getAge());
}
```

Y si queremos ordenar en orden descendente, simplemente intercambiamos el orden de los parámetros al llamar a la función:

```
@Override
public int compareTo(Person p) {
    return Integer.compare(p.getAge(), this.getAge());
}
```

Existen métodos similares en las clases `Float`, `Character`, `Double`, etc, que permiten comparar fácilmente datos simples de estos tipos.

Después, en el *main*, solo necesitaremos llamar al método `Arrays.sort` de la clase `Arrays` (necesitamos importar la clase `java.util.Arrays`), y entonces nuestro array se ordenará automáticamente:

```
import java.util.Arrays;
...
public static void main(String[] args) {
    Person[] people = new Person[50];
    ... // Fill array
    Arrays.sort(people);
    // Here our array is already sorted by age
}
```

Si elegimos el interfaz `Comparator` normalmente definiremos una clase externa que lo implemente (diferente a la clase `Person`)...

```
public class PersonComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p2.getAge(), p1.getAge());
    }
}
```

Entonces sólo necesitarímos llamar al método `Arrays.sort` con 2 argumentos: el array a ordenar (nuestro array de personas) y un objeto comparador que se usará para ordenarlo (una instancia de la clase `PersonComparator`):

```
import java.util.Arrays;
...
public static void main(String[] args) {
```

```
    Person[] people = new Person[50];
    ... // Fill array
    Arrays.sort(people, new PersonComparator());
    // Here our array is already sorted by age
}
```

De esta forma, podemos dejar la clase `Person` sin cambios, y se usa otra clase para comparar los objetos `Person`. Esta es una forma particularmente útil si no tenemos acceso al código de la clase original para modificarla, o si queremos utilizar una clase anónima como veremos en la siguiente sección.

> **Ejercicios propuestos:**
>
> **2.7.2.2.** Realiza una copia del ejercicio 2.6.1.1 del tema anterior y llámala *SortedVideoGameList*. Después, usa el interfaz `Comparable` o `Comparator` para ordenar el array de videojuegos por precio (en orden descendente) y muestra por pantalla el array ordenado.

2.7.3. Clases anónimas

Dado que las clases abstractas y los interfaces no pueden ser instanciados, siempre necesitaremos definir una subclase que herede o implemente de la clase abstracta o interfaz, y después instanciar esta subclase.

Desde Java 7, existe un atajo para definir estas subclases sin tener que definir un nuevo fuente o una nueva `clase`. Se trata de las clases anónimas.

Una **clase anónima** es una clase sin nombre que se crea en el momento donde se necesite implementar un interfaz, o heredar de una clase abstracta de forma que no necesitemos definir ninguna clase adicional para ello.

Veamos como utilizar una clase anónima con un par de ejemplos. El primero crea una instancia de la clase `Animal` para definir un tipo de animal que no se ha definido con ninguna de de las clases existentes (`Dog` o `Duck`, por ejemplo).

```
Animal strangeAnimal = new Animal("yellow", 2)
{
    @Override
    public void talk()
    {
        System.out.println("Vote for Quimby!");
    }
};

strangeAnimal.talk();
```

Tened en cuenta que solo se define una instancia de `Animal` y dentro de las llaves necesitamos redefinir e implementar todos los métodos abstractos (`talk` en este caso). También se puede definir

calquier atributo o método adicional que se necesite:

```
Animal strangeAnimal = new Animal("yellow", 2)
{
    String name = "Joe Quimby"

    @Override
    public void talk()
    {
        System.out.println("Vote for Quimby!");
    }

    public void anotherMethod()
    {
        System.out.println ("My name is " + name);
    }
};

...
```

Lo que se define de todas formas es un **objeto* que es un subtipo de `Animal` en este caso. No podremos volver a usar este código para definir otro animal más adelante (necesitaríamos duplicar el código).

En cuanto a los interfaces, también se pueden instanciar e implementar sus métodos en una clase anónima. Este ejemplo muestra como se define un subtipo de forma que implementa sus métodos (y alguno más que se podría añadir).

```
Shape irregularShape = new Shape()
{
    @Override
    public float calculateArea()
    {
        return 0.5f;
    }

    @Override
    public void draw()
    {
        System.out.println("Drawing this particular shape!");
    }
};

irregularShape.draw();
System.out.println(irregularShape.calculateArea());
```

Podemos usar clases anónimas en diferentes situaciones. Por ejemplo, podríamos definir directamente el método para ordenar el array de objetos `Person` que hemos visto en ejemplos anteriores:

```
import java.util.Arrays;
...
public static void main(String[] args) {
    Person[] people = new Person[50];
    ... // Fill array
    Arrays.sort(people, new Comparator<Person>()
    {
        @Override
        public int compare(Person p1, Person p2)
        {
            return Integer.compare(p2.getAge(), p1.getAge());
        }
    });
    // Here our array is already sorted by age
}
```

2.7.3.1. Cuando usar clases anónimas

Las clases anónimas son especialmente útiles cuando se necesita definir una instancia concreta de una clase abstracta o interfaz en un punto determinado del código y sólo en ese punto. De esta forma se evita definir una nueva clase asociada a ese código. Sin embargo, si planeamos usar la clase en otros lugares, una clase "normal" sería lo más recomendable para no duplicar código.

> **Ejercicios propuestos:**
>
> **2.7.3.1.** Modifica el ejercicio 2.7.2.1 añadiendo una nueva forma utilizando una clase anónima. Esta forma actuará como un rombo y tendrá dos atributos: diagonal mayor y diagonal menor. Calcula su área aplicando la fórmula adecuada e implementa el método `draw` mostrando por pantalla el mensaje "¡Soy un rombo!". Después añade un objeto de este tipo al array de formas y usa sus métodos.
>
> **2.7.3.2.** Define un `Comparator` en el ejercicio anterior para comparar formas por su área en orden descendente. Debes usar una clase anónima para implementar el comparador. Después ordena el array con este comparador y sácalo por pantalla.