Tema 2: Diseño de tipos

Autor: Miguel Toro, Mariano González. **Revisores**: Fermín Cruz, Pepe Riquelme, Toñi Reina.
Última modificación: 15/2/2022.

Tabla de contenidos

1. Esquema de diseño de tipos

En este capítulo vamos a ver el esquema que seguiremos para el diseño de tipos. Un buen diseño de tipos es básico para que los programas sean comprensibles y fáciles de mantener. Veremos algunas pautas para este diseño y algunos ejemplos que puedan servir de guía.

Nuestra plantilla para el diseño de un tipo será la siguiente:

NombreDeTipo
Propiedades:
NombreDePropiedad: Tipo, Consultable o no, Modificable o no, Derivada o no
…
Constructores:
Constructor 1 …
Restricciones:
Restricción 1 …
Operaciones:
…
Criterio de Igualdad
Representación como cadena
Orden natural (si lo tiene)

En este esquema, el criterio de igualdad y la representación como cadena están relacionados con el tipo *Object,* mientras que el orden natural está relacionado con el tipo *Comparable*. Estos tipos los veremos con más detalles en los siguientes apartados.

2. Clases

Las **clases** son las unidades de la POO que permiten definir los detalles del **estado interno** de un objeto (mediante los **atributos**), calcular las **propiedades** de los objetos a partir de los atributos e implementar las **funcionalidades** ofrecidas por los objetos (a través de los **métodos**)

Para implementar una clase partimos de la interfaz o interfaces que un objeto debe ofrecer y que han sido definidas previamente. En la clase se dan los detalles del estado y de los métodos. Decimos que la clase implementa (representado por la palabra **implements**) la correspondiente interfaz (o conjunto de ellas).

2.1 Estructura de una clase

Para escribir una clase en Java tenemos que ajustarnos al siguiente patrón:

```
[Modificadores] class NombreClase [extends ...] [implements ...] {
        [atributos]
        [métodos]
}
```

El patrón lo tenemos que completar con el nombre de la clase, y, de manera opcional, una serie de modificadores, una clausula ***extends***, una clausula ***implements*** y una serie de atributos y métodos.

En nuestra metodología de diseño, para ir detallando la clase nos guiamos por la interfaz que implementa. Como convención, haremos que el nombre de la clase que implementa una interfaz sea el mismo de la interfaz, seguido de Impl. Por ejemplo, una clase que implemente la interfaz *Punto* se llamará *PuntoImpl.*

2.2 Atributos

Los atributos sirven para establecer los detalles del **estado interno** de los objetos. Son un conjunto de variables, cada una de un tipo, y con determinadas restricciones sobre su visibilidad exterior.

Como una primera guía dentro de nuestra metodología, la clase tendrá un atributo por cada propiedad no derivada (o básica) y será del mismo tipo que esta propiedad. Aunque más adelante aprenderemos otras posibilidades, restringiremos el acceso a los atributos para que sólo sean visibles desde dentro de la clase. Esto implica que a la hora de declarar los atributos tenemos que anteponer la palabra reservada ***private*** a la declaración del mismo. Por ejemplo, para definir los atributos de una clase que implemente la interfaz *Punto* escribiríamos las dos líneas de código siguiente*:*

```
private Double x;
private Double y;
```

Como regla general, a los atributos les damos el mismo nombre, pero comenzando en minúscula, de la propiedad que almacenan. Solo definimos atributos para guardar los valores de las propiedades que sean básicas. Las propiedades derivadas, es decir, aquellas que se pueden calcular a partir de otras, no tienen un atributo asociado en general.

La declaración de un atributo tiene un **ámbito**, que va desde la declaración hasta el fin de la clase, incluido el cuerpo de los métodos.

La sintaxis para declarar los atributos responde al siguiente patrón:

[Modificadores] **tipo** Identificador [= valor inicial]*;*

Como puede verse, un atributo puede tener, de forma opcional, un posible valor inicial.

2.3 Métodos

Los métodos son unidades de programa que indican la forma concreta de **consultar o modificar** las propiedades de un objeto determinado. La sintaxis para los métodos corresponde al siguiente patrón:

[Modificadores] *TipoDeRetorno nombreMétodo* ([parámetros formales]) {
    Cuerpo;
}

Un método tiene dos partes: **cabecera** (o signatura o firma) y **cuerpo.** La **cabecera** está formada por el nombre del método, los parámetros formales y sus tipos, y el tipo que devuelve. Cada parámetro formal supone una nueva declaración de variable cuyo ámbito es el cuerpo del método. El **cuerpo** está formado por declaraciones, expresiones y otro código necesario para indicar de forma concreta qué hace el método. Las variables que se declaran dentro de un método se denominan variables locales y su ámbito es desde la declaración hasta el final del cuerpo del método.

Note que en una interfaz sólo aparecen las cabeceras de los métodos. Sin embargo, en una clase deben aparecer tanto la cabecera como el cuerpo. En una clase, al igual que ocurría con las interfaces, puede haber métodos con el mismo nombre pero diferente cabecera. Pero, al igual que se indicó en la declaración de interfaces, no puede haber dos métodos con el mismo nombre, los mismos parámetros formales y distinto tipo de retorno.

De forma general, hay dos tipos de métodos: **observadores** y **modificadores**. Los métodos observadores no modifican el estado del objeto. Es decir, no modifican los atributos. O lo que es lo mismo, los atributos no pueden aparecer, dentro de su cuerpo, en la parte izquierda de una asignación. Normalmente serán métodos observadores aquellos cuyo nombre empiece por *get*. Los métodos modificadores modifican el estado del objeto. Los atributos aparecen, dentro de su cuerpo, en la parte izquierda de una asignación. Normalmente todos los métodos que empiecen por *set* son modificadores. Ejemplos de un método modificador *(setX*) y otro consultor (*getX*) son:

```
public void setX(Double x1) {
      x = x1;
}
```

←Cabecera

} Cuerpo

```
public Double getX() {
      return x;
}
```

←Cabecera

} Cuerpo

Hay unos métodos especiales que llamaremos **métodos constructores** o simplemente **constructores.** Son métodos que tienen el mismo nombre que la correspondiente clase. Sirven para crear objetos nuevos y establecer el estado inicial de los objetos creados. Ejemplos de constructores de la clase *PuntoImpl* son:

```
public PuntoImpl (Double x1, Double y1) {
      this.x = x1;
      this.y = y1;
}
public PuntoImpl() {
      this.x = 0.;
      this.y = 0.;
}
```

Como vemos, los dos métodos constructores tienen el mismo nombre (que debe ser el de la clase) pero diferente cabecera. Con todo lo anterior podemos construir una clase.

Veamos ya la clase completa que implementa la interfaz Punto:

```
public class PuntoImpl implements Punto {

	// Atributos
	private Double x;
	private Double y;

	// Constructores
	public PuntoImpl (Double x1, Double y1) {
		this.x = x1;
		this.y = y1;
	}
	public PuntoImpl() {
		this.x = 0.;
		this.y = 0.;
	}

	// Observadores
	public Double getX() {
		return x;
	}
	public Double getY() {
		return y;
	}

	// Modificadores
	public void setX(Double x1) {
		x = x1;
	}
	public void setY(Double y1) {
		y = y1;
	}

	//Otras operaciones
	public Double getDistanciaAOtroPunto(Punto p) {
		Double dx = this.getX() - p.getX();
		Double dy = this.getY() - p.getY();
		return Math.sqrt(dx * dx + dy * dy);
	}
}
```

La palabra *this* es una palabra reservada y representa el objeto que estamos implementando. Así, *this.x* es el atributo *x* del objeto que estamos implementando. De la misma forma, `this.getX()` es el método *getX* invocado sobre el objeto que estamos implementando y, sin embargo, *p.getX()* es la invocación del método *getX* sobre el objeto *p*.

Cuando dentro del cuerpo de una clase, en la invocación de un método (o atributo) que se hace sobre el objeto *this* no hay ambigüedad, podemos quitar la palabra *this*. Por lo tanto, si no hay ambigüedad, las expresiones siguientes son equivalentes dentro de una clase dada:

- *this.getX()* es equivalente a *getX()*
- *this.getDistanciaAOtroPunto(origen)* es equivalente a *getDistanciaAOtroPunto(origen)*

Con la palabra *this* y el operador punto podemos formar expresiones correctas dentro de una clase.

3. Herencia

La **herencia** es una propiedad por la que se establecen relaciones que podemos denominar **padre-hijo entre interfaces o entre clases.** La herencia se representa mediante la cláusula **extends.**

En Java la herencia entre interfaces puede ser múltiple: un hijo puede tener uno o varios padres. La interfaz hija tiene todas las signaturas (métodos) declaradas en los padres. No se puede ocultar ninguna.

La sintaxis para la **interfaz** con herencia responde al siguiente patrón:

```
[Modificadores] interface NombreInterfazHija extends NombreInterfazPadre
[,…] {
       [signaturas de métodos]
}
```

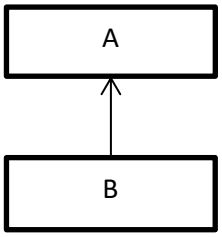
Mediante **extends** se indica que la interfaz que estamos definiendo combina otras interfaces padres y, posiblemente, añade algunos métodos más. Cuando una interfaz extiende a otra decimos que el tipo definido por la interfaz hija es un **subtipo** del tipo definido por la interfaz padre.

Entre tipo y subtipo hay una importante propiedad. Sea una variable *b* de un tipo *B,* y otra *a* de otro tipo *A*. Si *B* es un subtipo de *A,* entonces la variable *b* puede ser asignada a la variable *a* pero no al revés.

```
A a;
B b;
a = b;    // correcto
b = a;    // incorrecto
```

3.1 Grafo de tipos

Mediante el **grafo de tipos** representamos los tipos y sus relaciones. El grafo de tipos nos permite visualizar rápidamente los tipos que hemos diseñado y su relación de subtipado. En el ejemplo anterior tendríamos el siguiente grafo:



En este gráfico decimos que B es un **subtipo** de A, y A es un **supertipo** de B. El grafo de tipos es útil, también, para visualizar el conjunto de **tipos ofrecidos por un objeto**. El conjunto de tipos ofrecidos por un objeto está formado por todos los supertipos del tipo asociado a la clase que creó el objeto. En este conjunto siempre se añade un tipo proporcionado por Java y que es ofrecido por todos los objetos: el tipo *Object.*

Junto a esta relación entre los tipos hay una más que no se suele representar en el grafo de tipos. Es la **relación de uso**. Un tipo *A* usa a otro *B* cuando declara algún parámetro formal, tipo

de retorno, atributo o variable local del tipo *B*. Por ejemplo, pensemos en un tipo Círculo con dos propiedades, el centro y el radio. La primera propiedad es de tipo Punto, así que podemos decir que el tipo Círculo usa el tipo Punto.

4. El tipo Object

En Java existe una clase especial llamada *Object*. Todas las clases que definamos y las ya definidas heredan de *Object*, es decir, implícitamente *Object* es un tipo al que extienden todas las clases Java. Como cualquier tipo, *Object* proporciona una serie de métodos públicos. Aunque tiene más métodos, en este tema nos vamos a centrar solamente en tres de ellos: *equals*, *hashCode* y *toString*. Veremos sus propiedades, las restricciones entre ellos y la forma de rediseñarlos para que se ajusten a nuestras necesidades. La signatura de estos métodos es:

```
boolean equals(Object o);
int hashCode();
String toString();
```

Como el tipo *Object* es ofrecido por todos los objetos que creemos, los métodos anteriores están disponibles en todos los objetos, es decir, todos los objetos heredan los métodos *equals*, *hashCode* y *toString* de la clase *Object*. Estos métodos tienen definido un comportamiento por defecto. Veamos para qué se utiliza cada uno de estos métodos:

- El método *equals(Object o)* se utiliza para decidir si el objeto es igual al que se le pasa como parámetro.
- El método *hashCode()* devuelve un entero, que es el código *hash* del objeto. Todo objeto tiene, por lo tanto, un código *hash* asociado.
- El método *toString()* devuelve una cadena de texto que es la representación exterior del objeto. Cuando el objeto se muestre en la consola tendrá el formato indicado por su método *toString* correspondiente.

4.1 Propiedades y restricciones de los métodos equals, hashCode y toString

Todos los objetos ofrecen estos tres métodos. Por lo tanto es necesaria una buena comprensión de sus propiedades y un buen diseño de los mismos.

Propiedades de equals:

- **Reflexiva:** Un objeto es igual a sí mismo. Es decir, para cualquier objeto *x,* distinto de *null*, se debe cumplir que *x.equals(x)* es *true* y *x.equals(null)* es *false.*
- **Simétrica**: Si un objeto es igual a otro, el segundo también es igual al primero. Es decir, para dos objetos cualesquiera *x* e *y,* distintos de *null,* se debe cumplir que *x.equals(y) => y.equals(x).* Múltiples invocaciones de *x.equals(y)* deben devolver el mismo resultado si el estado de *x* e *y* no ha cambiado.
- **Transitiva:** Si un objeto es igual a otro, y este segundo es igual a un tercero, el primero también será igual al tercero. Es decir para tres objetos *x, y, z,* distintos de *null,* se debe cumplir que *x.equals(y) && y.equals(z) => x.equals(z).*

Propiedades de equals/toString:

- Si dos objetos son iguales, sus representaciones en forma de cadena también deben serlo. Es decir, para dos objetos cualesquiera *x e y,* distintos de *null,* se debe cumplir que *x.equals(y) => x.toString().equals(y.toString()).*

Propiedades de equals/hashCode:

- Si dos objetos son iguales, sus códigos *hash* tienen que coincidir. La inversa no tiene por qué ser cierta. Es decir, para dos objetos cualesquiera *x* e *y,* distintos de *null,* se debe cumplir que *x.equals(y) => x.hashCode() == y.hashCode().* Sin embargo, no se exige que dos objetos no iguales produzcan códigos *hash* desiguales, aunque hay que ser consciente de que se puede ganar mucho en eficiencia si en la mayoría de los casos objetos distintos tienen códigos hash distintos.

Las propiedades y restricciones anteriores son muy importantes. Si no se cumplen, el programa que diseñemos puede que no funcione adecuadamente. Todos los tipos ofrecidos en la API de Java tienen un diseño adecuado de esos tres métodos.

Veamos como ejemplo la implementación del tipo Punto. La plantilla del diseño del tipo se completaría de la siguiente forma:

Tipo Punto

…
Representación como cadena: (coordenada X, coordenada Y)
Criterio de igualdad: dos puntos son iguales si sus coordenadas X son iguales y sus coordenadas Y son iguales.

Teniendo esto en cuenta, los métodos *toString*, *equals* y *hashCode* que deberían incluirse en la clase PuntoImpl serían los siguientes:

```
public String toString() {
	String s;
	s = "(" + getX() + ", " + getY() + ")";
	return s;
}

public boolean equals(Object o) {
	boolean r = false;
	if (o instanceof Punto) {
		Punto p = (Punto) o;
		r = this.getX().equals(p.getX()) &&
			this.getY().equals(p.getY());
	}
	return r;
}

public int hashCode() {
	return getX().hashCode() * 31 + getY().hashCode();
}
```

4.2 Igualdad e Identidad

El operador de igualdad en Java es ==. Este operador devuelve un tipo *boolean* y toma dos operandos del mismo tipo (para ello es posible que haya que hacer alguna conversión automática).

El operador de igualdad tiene un significado muy preciso según se aplique entre tipos primitivos o tipos objeto. Para aclarar ese significado debemos introducir el concepto de identidad y su diferencia con la igualdad.

De manera general, dos objetos son **iguales** cuando los valores de sus propiedades observables son iguales. Por otro lado, dos objetos son **idénticos** cuando al modificar una propiedad observable de uno de ellos, se produce una modificación en la del otro y viceversa. De lo anterior, se deduce que **identidad implica igualdad**, pero igualdad no implica identidad. Es decir, la identidad de un objeto permanece inalterada aunque cambien sus propiedades. Dos objetos no idénticos pueden tener las mismas propiedades y entonces son iguales. Dos objetos idénticos siempre tienen las mismas propiedades.

Los valores de los tipos primitivos pueden tener igualdad, pero no tienen identidad. El operador *new* crea objetos con una nueva identidad. El operador de asignación entre tipos objeto asigna la identidad del objeto de la derecha al objeto de la izquierda. Después de asignar el objeto *b* al *a* (*a = b;*) ambos objetos son idénticos y, por lo tanto, si modificamos las propiedades de *a* quedan modificadas las de *b* y al revés. Entre tipos primitivos el operador de asignación asigna valores. Si *m* y *n* son de tipos primitivos, entonces la sentencia *m = n;* asigna el valor de *n* a *m*. Después de la asignación, *m* y *n* tienen el mismo valor, pero si modificamos el valor de *m,* no queda modificado el de *n* porque los elementos de tipos primitivos no tienen identidad, sólo tienen valor.

El operador == aplicado entre tipos primitivos decide si los valores de los operandos son iguales.

```
int i = 7;
int j = 4;
int k = j;
boolean a = (i == j);     // a es false
boolean b = (k == j);     // b es true
```

El valor de a es *false.* Se comparan los valores de tipos primitivos y estos valores son distintos. Después de asignar j a k sus valores son iguales luego b es true.

Este mismo operador aplicado entre tipos objeto decide si ambos objetos son idénticos o no. Para decidir si los objetos son iguales utilizamos el método equals, que se invoca mediante la expresión o1.*equals(o2) y* que devuelve un valor tipo *boolean*:

```
Punto p1 = new PuntoImpl(1.0, 2.0);
Punto p2 = new PuntoImpl(1.0, 2.0);
boolean c = (p1 == p2);          // c es false
boolean d = p1.equals(p2);       // d es true
```

Los objetos *p1* y *p2* han sido creados con dos identidades distintas (cada vez que llamamos al operador new se genera una nueva identidad) y no han sido asignados entre sí (no son idénticos) pero son iguales porque sus propiedades los son. Por ello *c* es *false* y *d* es *true*.

Veamos otro ejemplo:

```
Punto p1 = new PuntoImpl(1.0, 1.0);
```

```
Punto p2 = new PuntoImpl(3.0, 1.0);
Punto p3 = p1;
p1.setX(3.0);
boolean a = (p3 == p1);   // a es true
boolean b = (p3 == p2);   // b es false
Double x1 = p3.getX();    // x1 vale 3.0
```

El valor de *a* es true porque *p1* ha sido asignado a *p3* y por lo tanto tienen la misma identidad. Por ello al modificar la propiedad X en p1 queda modificada en p3 y por lo tanto x1 vale 3.0.

Cuando un objeto es inmutable no pueden cambiarse sus propiedades. Cualquier operación sobre el objeto que cambie las mismas producirá como resultado un nuevo objeto (con una nueva identidad) con los valores adecuados de las propiedades.

```
Integer a = 3;
Integer b = a;
b++;
boolean e = (a == b);     // e es false
```

Al ser el tipo *Integer* inmutable, el valor de *e* es false. En la línea 3, con la expresión *b++*, se crea un objeto nuevo (con una nueva identidad) que se asigna a *b*. Si se elimina la línea 3 (*b++;*) entonces el valor de e será *true*.

5. El tipo Comparable

El tipo *Comparable* define un orden sobre los objetos de un tipo creado por el usuario, al que llamaremos orden natural. Java ya proporciona un orden natural para los tipos envoltura como Integer y Double o el tipo inmutable String permitiendo que, por ejemplo, se puedan ordenar cuando forman parte de una lista.

El tipo *Comparable* está definido como:

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

El tipo *Comparable* se compone de un único método, de nombre *compareTo*. El tipo *Comparable* usa un tipo genérico (T) y establece el orden natural sobre los objetos de un tipo dado.

- El método *compareTo* compara dos objetos *o1* y *o2* y devuelve un entero que es:
- Negativo si *o1* es menor que *o2*
- Cero si *o1* es igual a *o2*
- Positivo si *o1* es mayor que *o2*

El orden natural definido debe ser coherente con la definición de igualdad. Si *equals* devuelve *true, compareTo* debe devolver cero. Aquí también incluimos, tal como se recomienda en la documentación de Java, la inversa. Es decir, que si *compareTo* devuelve cero, entonces *equals* devuelve *true*. Esto lo podemos enunciar diciendo que la expresión siguiente es verdadera para cualquier par de objetos *x* e y: (x.compareTo(y) == 0) ⇔ (x.equals(y)).

En general, para implementar el método *compareTo* usaremos los métodos *compareTo* de las propiedades involucradas en la igualdad o algunas otras derivadas. Un orden natural adecuado puede ser comparar en primer lugar por una propiedad elegida arbitrariamente, si resulta cero comparar por la segunda propiedad, etc.

Veamos un ejemplo con un nuevo tipo, Persona, ya que en el tipo Punto no tiene sentido definir un orden natural (no todos los tipos admiten un orden). El diseño de este tipo Persona es el siguiente:

Tipo Persona

- Propiedades:
  - **DNI**: String, consultable.
  - **Nombre**: String, consultable.
  - **Apellidos**: String, consultable.
  - **Fecha de nacimiento**: LocalDate, consultable.
  - **Edad**: Integer, consultable, derivada.
- Representación como cadena: apellidos, nombre (DNI)
- Criterio de igualdad: dos personas son iguales si sus DNI son iguales, sus apellidos son iguales y sus nombres son iguales.
- Criterio de orden natural: por apellidos, nombre y DNI.

Como se puede ver, el orden natural para el tipo Persona es compatible con la igualdad, ya que tiene en cuenta las mismas propiedades escogidas para definir el criterio de igualdad.

Lo primero que debemos hacer es indicar que la interfaz del tipo Persona extiende la interfaz Comparable:

```
public interface Persona extends Comparable<Persona> {
	String getDNI();
	String getNombre();
	String getApellidos();
	LocalDate getFechaNacimiento();
	Integer getEdad();
}
```

Ahora ya podemos implementar en la clase PersonaImpl el método *compareTo* definido en la interfaz Comparable, de acuerdo con la definición del orden natural para el tipo Persona. El método quedaría así:

```
public int compareTo(Persona p) {
	int r;
	if (p == null) {
		throw new NullPointerException();
	}
	r = getApellidos().compareTo(p.getApellidos());
	if (r == 0) {
		r = getNombre().compareTo(p.getNombre());
		if (r == 0) {
			r = getDNI().compareTo(p.getDNI());
		}
	}
	return r;
}
```

Veamos por último un test donde se usa el método *compareTo* para comparar dos objetos de tipo Persona:

```
public TestCompareTo {
	public static void main(String[] args) {
		Persona p1 = new PersonaImpl("222222222R", "John", "Doe",
			LocalDate.of(2000, 10, 11));
		Persona p2 = new PersonaImpl("111111111A", "Jane", "Doe",
			LocalDate.of(2001, 5, 17));

		if (p1.compareTo(p2) > 0) {
			System.out.println(p1 + " va después de " + p2);
		}
	}
}
```

6. Restricciones y excepciones

6.1 Restricciones

En el esquema de diseño de tipos aparece la palabra 'Restricciones' junto a cada propiedad. Con esto nos referimos a que pueden existir ciertas restricciones sobre los valores que pueden tomar las propiedades. Por ejemplo, la duración de una canción no puede ser negativa, la fecha de llegada de un vuelo no puede ser anterior a la fecha de salida, o el nombre de una persona no puede estar vacío.

Cuando se crea un objeto, hay que comprobar que se cumplen todas las restricciones que existan sobre sus propiedades. También hay que comprobarlas cuando se intente modificar una propiedad del objeto que tenga una restricción. Por tanto, los lugares de la clase donde hay que chequear las restricciones son los métodos constructores y los modificadores.

Si se incumple una restricción, el programa debe detener su ejecución, informando al usuario de la restricción incumplida. Para ello contamos en Java con un mecanismo: las excepciones.

6.2 Excepciones

Las excepciones son, junto con las sentencias de control vistas anteriormente, otro mecanismo de control. Es decir, son un instrumento para romper y gestionar el orden en que se evalúan las sentencias de un programa.

Se denomina **excepción** a un evento que ocurre durante la ejecución de un programa, y que indica una situación normal o anormal que hay que gestionar. Por ejemplo, una división por cero o el acceso a un fichero no disponible en el disco. Estos eventos pueden ser de dos grandes tipos: eventos generados por el propio sistema y eventos generados por el programador. En ambos casos hay que gestionar el evento. Es decir, decidir qué sentencias se ejecutan cuando el evento ocurre.

Cuando se produce un evento como los comentados más arriba, decimos que se ha **disparado una excepción**. Cuando tomamos decisiones después de haberse producido un evento decimos que **gestionamos la excepción**.

Cuando se dispara (o se lanza, o se eleva) una excepción tras la ocurrencia de un evento, se crean unos objetos que transportan la información del evento. A estos objetos también los llamamos excepciones, y Java dispone de un conjunto predefinido de ellos.

6.3 Lanzamiento de excepciones

Cuando se lanza una excepción, se interrumpe el flujo del programa. El lanzamiento de una excepción suele tener la siguiente estructura:

```
if (condicion_de_lanzamiento) {
	throw new tipo_excepcion("Texto informativo");
}
```

Como vemos, hay una sentencia *if* que evalúa una condición y, si es cierta, se lanza la excepción. Llamaremos **condición de lanzamiento** a la condición que controla el lanzamiento de la excepción. La condición de lanzamiento es una expresión lógica que depende de los parámetros del método y de las propiedades del objeto. La cláusula *throw* crea un objeto del tipo adecuado y dispara la excepción.

Veamos un ejemplo con el tipo Persona. Una restricción de este tipo consiste en que la propiedad nombre debe tener siempre un valor. Por tanto, si intentamos asignar al nombre de una persona una cadena vacía, se incumplirá la restricción y deberá lanzarse una excepción. El constructor de *PersonaImpl* quedaría de la siguiente manera:

```
public PersonaImpl(String dni, String nombre, String apellidos, LocalDate
		fechaNacimiento) {
	if (nombre.equals("")) {
		throw new IllegalArgumentException(
			"El nombre no puede estar vacío");
	}
	this.dni = dni;
	this.nombre = nombre;
	this.apellidos = apellidos;
	this.fechaNacimiento = fechaNacimiento;
}
```

La restricción hay que chequearla en dos lugares: en el constructor (o constructores, si hay más de uno), y en el método modificador de la propiedad, si lo hay. Suponiendo que la propiedad nombre fuese modificable, y que por tanto hubiese un método *setNombre* como el siguiente,

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

entonces habría que comprobar que, al modificar el nombre de la persona, el nuevo nombre no es una cadena vacía, y el método *setNombre* quedaría así:

```
public void setNombre(String nombre) {
	if (nombre.equals("")) {
		throw new IllegalArgumentException(
			"El nombre no puede estar vacío");
	}
	this.nombre = nombre;
}
```

Si se invoca en la clase *TestPersona* al método *p.setNombre("")*, siendo p un objeto de tipo Persona, se mostrará en la consola un mensaje como el siguiente:

```
Exception in thread "main" java.lang.IllegalArgumentException: El nombre no
puede estar vacío
	at fp.persona.PersonaImpl.setNombre(PersonaImpl.java:30)
	at fp.persona.test.TestPersona.main(TestPersona.java:12)
```

Podemos ver que el código que chequea la restricción se repite en los dos métodos que deben comprobarla. Para evitar esto, podemos crear un método interno de la clase que se encargue de hacer el chequeo, e invocarlo desde ambos métodos. El código quedaría entonces así:

```
public PersonaImpl(String dni, String nombre, String apellidos, LocalDate
		fechaNacimiento) {
	checkNombre(nombre);
	this.dni = dni;
	this.nombre = nombre;
	this.apellidos = apellidos;
	this.fechaNacimiento = fechaNacimiento;
}
public void setNombre(String nombre) {
	checkNombre(nombre);
	this.nombre = nombre;
}
private void checkNombre(String nombre) {
	if (nombre.equals("")) {
		throw new IllegalArgumentException(
			"El nombre no puede estar vacío");
	}
}
```

El método *checkNombre* se declara como `private` porque es un método que se utiliza internamente en la clase, y no forma parte de la interfaz del tipo Persona.

6.4 Gestión de excepciones

En el ejemplo anterior, el programa finaliza lanzando una excepción y mostrando el correspondiente mensaje de error en la consola. Podemos gestionar la excepción de forma que el programa finalice de una forma más controlada. Para ello, en lugar de lanzar la excepción, la podemos **capturar**, es decir, detectar que se ha producido, y tomar las acciones necesarias para que el programa finalice de la forma más adecuada posible. Para ello usamos la cláusula *try/catch*. Veamos cómo sería en el ejemplo anterior.

```
try {
	Persona p = new PersonaImpl("12345678Z", "", "Nadie",LocalDate.now());
	System.out.println(p);
} catch (IllegalArgumentException e) {
	System.out.println("Excepción capturada: nombre vacío");
}
System.out.println("Programa finalizado");
```

Cuando se ejecute el test, por consola aparecerá el siguiente resultado:

```
Excepción capturada: nombre vacío
Programa finalizado
```

Como puede verse, dentro del bloque *try* se intenta crear una persona con el nombre vacío, lo cual provoca el lanzamiento de la excepción. El bloque *catch* **captura** esta excepción antes de que se aborte el programa, y muestra el mensaje "Excepción capturada: nombre vacío". El programa continúa su ejecución con la instrucción que sigue al bloque, mostrando el mensaje "Programa finalizado".

7. Constructor a partir de String

Con el método *toString()* convertimos un objeto en su representación como cadena de caracteres. Muchos tipos necesitan la operación inversa, es decir, construir un objeto a partir de la información contenida en una cadena de caracteres. Con esta funcionalidad pretendemos poder construir, por ejemplo, objetos de tipo Punto a partir de cadenas de la forma "(5.3, -2.1)". Para poder implementar esta funcionalidad necesitamos que el tipo correspondiente tenga un constructor que tome como único parámetro una cadena de caracteres.

Los tipos envoltura y los tipos enumerados disponen de unos métodos, *valueOf*, que convierten una cadena de caracteres en un valor del tipo correspondiente. Por ejemplo,

```
Integer a = Integer.valueOf("+35");
Double b = Double.valueOf("-4.57");
Color c = Color.valueOf("ROJO");
```

Estos métodos elevan la excepción *NumberFormatException* o *IllegalArgumentException* si el formato de la cadena no es el adecuado. En los casos que diseñemos nosotros, se elevará la excepción *IllegalArgumentException*.

Para diseñar constructores para tipos que tomen una cadena, necesitamos analizar la cadena y partirla en trozos adecuados. En el caso del tipo punto, la cadena *"(5.3, -2.1)"* tenemos que partirla en las subcadenas "5.3", que dará lugar a la coordenada *x*, y "-2.1", que dará lugar a la coordenada *y*. Esto lo podemos hacer con el método *split* del tipo *String*.

Si queremos que el tipo Punto, junto con su funcionalidad ya ofrecida, pueda crear objetos a partir de una cadena de caracteres, entonces la clase que lo implemente debe tener un constructor de la forma:

```
public PuntoImpl(String s) {
	String s1 = s.replace("(", "").replace(")", "");
	String[] sp = s1.split(",");
	if (sp.length != 2) {
		throw new IllegalArgumentException(
			"Cadena con formato no válido");
	}
	x = Double.valueOf(sp[0].trim());
	y = Double valueOf(sp[1].trim());
}
```

En primer lugar, se eliminan los posibles caracteres adicionales que no formen parte de los valores de las propiedades (en este caso los paréntesis), lo cual hacemos mediante el método *replace* del tipo String. A continuación, se utiliza el método *split*, también del tipo String, para

trocear la cadena según la expresión regular definida en el parámetro. En este caso se parte la cadena tomando como separador el carácter coma, con lo cual se deben obtener dos trozos, correspondientes a las coordenadas x e y, si la cadena está bien formada. Estos trozos son cadenas de caracteres, y es necesario convertirlos a valores de tipo Double para almacenarlos en los atributos correspondientes. Para ello, usamos el método *valueOf* del tipo Double, eliminando previamente los posibles espacios en blanco al comienzo y al final de la cadena mediante el método *trim* del tipo String.

Veamos un segundo ejemplo con el tipo Persona, donde como recordamos hay una restricción sobre la propiedad nombre. Suponiendo que la representación como cadena de un objeto de tipo Persona tiene un formato como el del ejemplo "111111111A, Jane, Doe, 17/5/2001", al trocear la cadena deberemos obtener cuatro trozos, correspondientes al DNI, nombre, apellidos y fecha de nacimiento, en ese orden. El constructor a partir de String quedaría de la siguiente manera:

```
public PersonaImpl(String s) {
	String[] sp = s.split(",");
	if (sp.length != 4) {
		throw new IllegalArgumentException(
			"Cadena con formato no válido");
	}
	String nombre = sp[1].trim();
	checkNombre(nombre);
	this.dni = sp[0].trim();
	this.nombre = nombre;
	this.apellidos = sp[2].trim();
	this.fechaNacimiento = LocalDate.parse(sp[3].trim(),
		DateTimeFormatter.ofPattern("d/M/y"));
}
```

Veamos ahora el caso de un tipo que tiene una propiedad que a su vez es de un tipo también definido por nosotros. Sea por ejemplo un tipo Circulo con dos propiedades: el radio del círculo, que es de tipo Double, y el centro del círculo, que es de tipo Punto. Suponiendo que la representación como cadena de un objeto de tipo Circulo tiene un formato como el del ejemplo “(2.0, -3.5); 5.5”, al trocear la cadena por el carácter ‘;’ obtendremos dos trozos, siendo el primero el centro y el segundo el radio del círculo. Como se puede ver, el primer trozo coincide con la representación como cadena de un Punto, por lo cual podemos construir un punto a partir de él usando el constructor a partir de String del tipo Punto. Con esto, el constructor a partir de String del tipo Circulo sería el siguiente:

```
public CirculoImpl(String s) {
	String[] sp = s.split(";");
	if (sp.length != 2) {
		throw new IllegalArgumentException(
			"Cadena con formato no válido");
	}
	Double radio = Double.valueOf(sp[1].trim());
	checkRadio(radio); // El radio del círculo debe ser positivo
	this.centro = new PuntoImpl(sp[0].trim());
	this.radio = radio;
}
```

En general, los pasos que hay que realizar son los siguientes:

1. Eliminar caracteres innecesarios (sólo a veces), usando el método *replace*.
2. Trocear la cadena mediante el método *split*.
3. Si el número de trozos obtenido no es el esperado, lanzar la excepción.
4. Para cada propiedad que tenga alguna restricción, declarar una variable local del tipo de la propiedad e inicializarla a partir del trozo en cuestión. Usaremos para ello el constructor de String del tipo, y siempre el método *trim* sobre el trozo.
5. Invocar a los métodos *checkers* pasándoles las variables anteriores.
6. Inicializar los atributos, bien a partir de las variables anteriores, bien usando el constructor de String del tipo adecuado, y aplicando el método *trim* sobre el trozo.

8. Records

En los apartados anteriores hemos visto cómo construir una clase para implementar un tipo dado por sus propiedades y operaciones. Esta clase contiene unos métodos básicos, como el constructor, los métodos observadores y modificadores, o los métodos *equals*, *hashCode* y *toString*. Si nos fijamos en estos métodos, podemos ver que su código sigue siempre un patrón similar; no tenemos más que comparar los métodos que hemos escrito para el tipo Punto y el tipo Persona: de unos a otros, solo cambian los atributos, siendo el resto idéntico.

Vemos pues que el diseño del tipo contiene mucho código que es mecánico, y por tanto fácil de automatizar. De hecho, los IDE suelen ofrecer la opción de crear automáticamente estos métodos a partir de los atributos de la clase. Consciente de ello, en la versión 14 de Java se introduce un nuevo elemento, **record**, cuyo objetivo es simplificar el diseño de un tipo inmutable para almacenar datos simples, como los resultados de consultas a una base de datos, o la información obtenida a partir de un servicio o un archivo CSV. Al ser inmutables, los records no tienen métodos modificadores.

Para ver cómo funciona, veamos cómo se implementaría el tipo Persona utilizando un record:

```
public record Persona(String DNI, String nombre, String apellidos,
            LocalDate fechaNacimiento) {
}
```

Como se aprecia, basta con indicar el tipo y el nombre de cada atributo. Con esto, se crea un record con esos atributos y con unos métodos predefinidos: un constructor con parámetros, un método observador por cada propiedad, y los métodos *equals*, *hashCode* y *toString*. Podemos hacer entonces lo siguiente:

```
Persona p = new Persona("11111111A", "Jane", "Doe", LocalDate.of(2001,5,17));
System.out.println("Nombre completo: " + p.nombre() + " " + p.apellidos());
```

Se puede observar que los métodos observadores tienen el mismo nombre que el atributo correspondiente. En cuanto a la igualdad, el método *equals* devuelve *true* si los objetos son del mismo tipo y coinciden los valores de todos sus atributos.

Es posible modificar estos métodos. Por ejemplo, se puede modificar el constructor por defecto para chequear posibles restricciones:

```
public record Persona(String DNI, String nombre, String apellidos,
		LocalDate fechaNacimiento) {

	public Persona {
		checkNombre(nombre);
	}

	private void checkNombre(String nombre) {
		if (nombre.equals("")) {
			throw new IllegalArgumentException(
				"El nombre no puede estar vacío");
		}
	}
}
```

También se pueden añadir nuevos métodos. Por ejemplo, se puede añadir un nuevo constructor, se pueden añadir métodos para las propiedades derivadas, o se puede hacer que el tipo sea Comparable, añadiendo un método compareTo. Con todo esto, el record nos podría quedar de esta forma:

```
public record Persona(String DNI, String nombre, String apellidos,
		LocalDate fechaNacimiento) implements Comparable<Persona> {

	public Persona {
		checkNombre(nombre);
	}

	public Persona(String nombre, String apellidos) {
		this("", nombre, apellidos, LocalDate.now());
	}

	private void checkNombre(String nombre) {
		if (nombre.equals("")) {
			throw new IllegalArgumentException(
				"El nombre no puede estar vacío");
		}
	}

	public Integer edad() {
		return fechaNacimiento.until(LocalDate.now()).getYears();
	}

	public int compareTo(Persona p) {
		int r;
		if (p == null) {
			throw new NullPointerException();
		}
		r = apellidos.compareTo(p.apellidos());
		if (r == 0) {
			r = nombre.compareTo(p.nombre());
			if (r == 0) {
				r = DNI.compareTo(p.DNI());
			}
		}
		return r;
	}
}
```

Lo que no se puede añadir son métodos modificadores, ya que, como se ha dicho antes, **los records definen tipos inmutables**, es decir, aquellos cuyas propiedades no cambian una vez creados. No podríamos usar por tanto un record para el tipo Punto, ya que sus coordenadas se pueden modificar.

En resumen, el elemento record nos proporciona una forma simple y rápida de implementar un tipo, siempre que el tipo sea inmutable. El esquema básico de un record se puede enriquecer añadiendo métodos. En este punto, hemos de tener en cuenta que la idea del record es simplificar el diseño; si necesitamos añadir muchos métodos, posiblemente sea mejor idea utilizar una clase.

9. Diseño de una clase sin interfaz

Hemos visto que en el diseño de un tipo hay dos piezas de código separadas: la interfaz y la clase. La interfaz define las operaciones que se pueden realizar con el tipo, y la clase establece cómo se realizan estas operaciones. Una misma interfaz puede implementarse mediante más de una clase, ya que puede haber distintas formas de realizar una misma operación.

Sin embargo, existe la posibilidad de utilizar solo una pieza de código, la clase, para diseñar un tipo. En este caso, la clase contiene tanto la definición como la implementación del tipo. Veamos, por ejemplo, cómo quedaría el tipo Persona implementado con una clase:

```
public class Persona {

	private String dni, nombre, apellidos;
	private LocalDate fechaNacimiento;

	public Persona(String dni, String nombre, String apellidos,
			LocalDate fechaNacimiento) {
		checkNombre(nombre);
		this.dni = dni;
		this.nombre = nombre;
		this.apellidos = apellidos;
		this.fechaNacimiento = fechaNacimiento;
	}

	private void checkNombre(String nombre) {
		if (nombre.equals("")) {
			throw new IllegalArgumentException(
				"El nombre no puede estar vacío");
		}
	}

	public String getDNI() {
		return dni;
	}

	public String getNombre() {
		return nombre;
	}
```

```
	public String getApellidos() {
		return apellidos;
	}

	public LocalDate getFechaNacimiento() {
		return fechaNacimiento;
	}

	public Integer getEdad() {
		return fechaNacimiento.until(LocalDate.now()).getYears();
	}

	public String toString() {
		return "Persona [dni=" + dni + ", nombre=" + nombre + ",
			apellidos=" + apellidos + ",
			fechaNacimiento=" + fechaNacimiento + "]";
	}
}
```

Como vemos, damos a la clase el mismo nombre del tipo. Y usaremos este nombre para crear los objetos. Por ejemplo:

```
Persona p = new Persona("11111111A", "Jane", "Doe", LocalDate.of(2001,5,17));
System.out.println("Nombre completo: " + p.getNombre() + " " +
	p.getApellidos());
```

Por tanto, se puede implementar un tipo creando directamente una clase, sin necesidad de definir una interfaz. Esto tiene el inconveniente de que se pierde la separación entre la definición y la implementación del tipo, pero por otro lado el código resulta más simple. Y si solo vamos a tener una implementación del tipo, es suficiente con la clase.

Un caso particular de clases que no implementan una interfaz son las llamadas **clases de utilidad**. Son clases que contienen una colección de métodos que realizan unas tareas útiles en un determinado dominio. Estas clases normalmente no tienen atributos, sino solo un conjunto de métodos static.

Un ejemplo de clase de utilidad es la clase *Math* de la API de Java. Esta clase contiene métodos que realizan operaciones matemáticas, como la raíz cuadrada, el seno y el coseno, el valor absoluto, etc.

También podemos definir nuestras propias clases de utilidad. Por ejemplo, la clase de utili0dad *Checkers* contiene métodos para chequear los valores de los atributos de un objeto:

```
public class Checkers {

	public static void check(String textoRestriccion, Boolean condicion) {
	   if (!condicion) {
	      throw new IllegalArgumentException(
	         Thread.currentThread().getStackTrace()[2].getClassName()
	         + "." +
	         Thread.currentThread().getStackTrace()[2].getMethodName()
	         + ": " + textoRestriccion);
	   }
	}
```

```
        public static void checkNoNull(Object... parametros) {
            for (int i = 0; i < parametros.length; i++) {
                if (parametros[i] == null) {
                    throw new IllegalArgumentException(
                        Thread.currentThread().getStackTrace()[2].getClassName()
                        + "." +
                        Thread.currentThread().getStackTrace()[2].getMethodName()
                        + ": el parámetro " + (i + 1) + " es nulo");
                }
            }
        }
}
```

Podemos invocar un método de esta clase en el constructor de la clase Persona, para chequear que el nombre de la persona no es una cadena vacía:

```
public class Persona {

        private String dni, nombre, apellidos;
        private LocalDate fechaNacimiento;

        public Persona(String dni, String nombre, String apellidos,
                        LocalDate fechaNacimiento) {
                Checkers.check("El nombre no puede estar vacío",
                        !nombre.equals(""));
                this.dni = dni;
                this.nombre = nombre;
                this.apellidos = apellidos;
                this.fechaNacimiento = fechaNacimiento;
        }

        ...
```

Al usar el método *check* de la clase *Checkers*, evitamos tener que escribir el método *checkNombre*. Y esto lo podemos hacer en cualquier clase en la que tengamos que chequear el valor de un atributo, ahorrándonos escribir un método check específico para cada clase.

En resumen, el uso de una clase de utilidad nos permite reutilizar métodos de uso frecuente, evitando tener que reescribirlos una y otra vez. Sin embargo, no conviene abusar de este tipo de clases, ya que no siguen el principio de la orientación a objetos sino el imperativo, y además corren el riesgo de convertirse en un simple contenedor de métodos sin relación entre sí. Siempre hay que pensar antes si esos métodos encajan en una clase normal con sus propiedades y operaciones.

En este tema hemos visto tres formas diferentes de diseñar un tipo en Java: mediante una interfaz y una clase, únicamente mediante una clase, y mediante un record. Ante esto, nos podemos preguntar cuándo usar cada una de estas formas de diseñar un tipo. Una recomendación puede ser la siguiente:

- **Interfaz + Clase**: cuando hay más de una forma de implementar el tipo.
- **Clase**: cuando solo vamos a tener una implementación del tipo.
- **Record**: cuando el tipo es inmutable y almacena datos simples.