Tema 1: Introducción a Java

Autor: Miguel Toro, Mariano González. **Revisor**: Fermín Cruz. **Última modificación**: 24/2/2022.

Tabla de contenidos

1. Conceptos básicos de la P.O.O.

Los **Lenguajes Orientados a Objetos** son los que están basados en la modelización mediante objetos. Estos objetos tendrán la **información** y las **capacidades** necesarias para resolver los problemas en que participen. Por ejemplo, una persona tiene un nombre, una fecha de nacimiento, unos estudios…; un vehículo tiene un dueño, una matrícula, una fecha de matrícula…; un círculo tiene un centro y un radio…. Para una persona tenemos la capacidad de calcular su edad, para un vehículo la de saber en qué fecha tiene que pasar la ITV, y para un círculo el cálculo de su área.

1.1 Programación Orientada a Objetos

La POO (Programación Orientada a Objetos) es una forma de construir programas de ordenador donde las entidades principales son los objetos. Está basada en la forma que tenemos los humanos de concebir objetos, distinguir unos de otros mediante sus **propiedades** y asignarles **funciones** o capacidades. Estas dependerán de las propiedades que sean **relevantes** para el problema que se quiere resolver.

Los elementos básicos de la POO son:

- **Objeto**
- **Interfaz**
- **Clase**
  - **Atributos** (almacenan las propiedades)
  - **Métodos** (consultan o actualizan las propiedades)
- **Paquete**

1.2 Objeto

Los objetos tienen una identidad, unas propiedades, un estado y una funcionalidad asociada:

- Cada objeto tiene una **identidad** que lo hace único y lo distingue del resto de objetos del mismo tipo. Puede haber varios objetos con el mismo estado pero cada uno con su identidad propia; en ese caso se dice que los objetos son iguales, pero no idénticos.
- Las **propiedades** son las características observables de un objeto desde el exterior del mismo. Pueden ser de diversos tipos (números enteros, reales, textos, booleanos, etc.). Estas propiedades pueden estar ligadas unas a otras y pueden ser modificables desde el exterior o no. Las propiedades de un objeto pueden ser básicas y derivadas. Son **propiedades derivadas** las que dependen de las demás. El resto son **propiedades básicas o no derivadas.**
- El **estado** indica cuál es el valor de sus propiedades en un momento dado.
- La **funcionalidad** de un objeto se ofrece a través de un conjunto de **métodos**. Los métodos actúan sobre el estado del objeto (pueden consultar o modificar las propiedades) y son el mecanismo de comunicación del objeto con el exterior.

La **encapsulación** es un concepto clave en la POO y consiste en **ocultar** la forma en que se almacena la información que determina el estado del objeto. Esto conlleva la obligación de que toda la **interacción con** el objeto se haga a través de ciertos **métodos** implementados con

ese propósito (se trata de **ocultar** información **irrelevante** para quien utiliza el objeto). Las propiedades de un objeto sólo serán accesibles para consulta o modificación a través de sus **métodos**. La encapsulación permite definir de forma estricta las responsabilidades de cada objeto, lo que facilita la depuración de las aplicaciones cuando se detectan errores.

1.3 Interfaz

Una interfaz es un elemento de la POO que permite, a priori, establecer **cuáles** son las propiedades y ***qué*** *se puede hacer con un objeto de un determinado tipo*, pero no se preocupa de saber **cómo** se hace.

Formalmente una interfaz (**interface**) contiene, principalmente, las **signaturas** de los métodos que nos permiten consultar y/o cambiar las propiedades de los objetos, así como del resto de operaciones que definen la funcionalidad. Denominamos **signatura de un método** a su nombre, el número y el tipo de los parámetros que recibe y el tipo que devuelve en su caso. Llamaremos a este tipo devuelto **tipo de retorno**. A las signaturas de un método las llamaremos también **cabecera del método** o **prototipo.**

Cada vez que declaramos una interfaz estamos declarando un tipo nuevo. Con este tipo podemos declarar variables y, con ellas, construir expresiones.

Supongamos que queremos modelar un objeto que llamaremos Punto (y que representará un punto del plano). Las propiedades que nos interesan de los objetos de tipo Punto son las siguientes:

- X, que representa su coordenada X, es de tipo Real, y es consultable y modificable
- Y, que representa su coordenada Y, es de tipo Real, y es consultable y modificable

La interfaz que representa el tipo Punto es la siguiente:

```
public interface Punto {
	Double getX();
	Double getY();
	void setX(Double x1);
	void setY(Double y1);
}
```

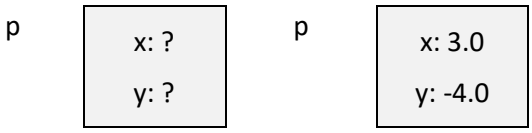
Cuando definimos una interfaz estamos definiendo un tipo nuevo con el cual podemos declarar nuevas entidades. A estas entidades declaradas de esta forma las llamaremos objetos. Llamaremos objetos, en general, a las variables cuyos tipos vengan definidos por una interfaz o una clase. En el siguiente ejemplo se declara la variable *p,* que representa un objeto de tipo punto:

```
Punto p;
```

La variable *p* puede combinarse con los métodos de la interfaz mediante el operador punto (.) para formar expresiones. Estas expresiones, cuando están bien formadas, tienen un tipo y un valor. Desde otro punto de vista podemos decir que el método ha sido **invocado** o **llamado** sobre el objeto *p*. Por ejemplo:

```
p.setX(3.0);
p.setY(-4.0);
```

Las dos expresiones anteriores están bien formadas porque p ha sido declarada de tipo Punto. Ambas son de tipo *void*. Indican que los métodos *setX* y *setY* han sido invocados sobre el objeto p. Con ello conseguimos que las propiedades *X* e *Y* del objeto pasen a ser (3.0,-4.0). Ambas son **expresiones con efectos laterales** porque al ser evaluadas cambia el estado del objeto *p*.



Si queremos consultar las propiedades de *p*, invocamos a los métodos *get*:

```
p.getX()
p.getY()
```

Las expresiones anteriores son expresiones bien formadas. Ambas son de tipo *Double* y sus valores son los valores de las propiedades *X* e *Y*, respectivamente, guardadas en el estado el objeto en un momento dado. Otra forma de decirlo es: al invocar a los métodos *getX()* o *getY()* sobre una entidad *p* de tipo *Punto* obtenemos valores de tipo *Double*. Estos son los valores de las propiedades *X* e *Y* guardados en el estado de *p*. Las dos son expresiones sin efectos laterales porque al ser evaluadas no cambian el estado del objeto.

Los objetos, las consultas a los mismos y, en general, la invocación de alguno de sus métodos sobre un objeto pueden formar parte de expresiones. Por ejemplo:

```
p.getX() * p.getX() + p.getY() * p.getY();
```

es una expresión de tipo *Double* que calcula la suma del cuadrado de la coordenada X y el cuadrado de la coordenada Y del punto *p* .

El punto (.) es, por lo tanto, un operador que combina un objeto con sus métodos y que indica que el correspondiente método se ha invocado sobre el objeto. Después del operador punto (.) aparece el nombre del método y cada **parámetro formal** (que aparecen en la declaración) es sustituido (en la **invocación**) por una expresión que tiene el mismo tipo o un tipo compatible. El conjunto de estas expresiones que sustituyen a los parámetros formales se llaman **parámetros reales**. Una variable, el operador punto (.), un método y sus correspondientes parámetros reales forman una expresión. El tipo de esa expresión es el tipo de retorno del método.

Diseñemos el tipo Punto de una forma más general. Podemos incluir otras operaciones en el tipo, además de las de consulta y modificación de propiedades:
Punto:

- Propiedades:
  - X, Real, Consultable, modificable
  - Y, Real, Consultable, modificable
- Operaciones:
  - Double getDistanciaAOtroPunto(Punto p): devuelve la distancia euclídea a otro punto.

La interfaz asociada es:

```
public interface Punto {
	Double getX();
	Double getY();
	void setX(Double x1);
	void setY(Double y1);
	Double getDistanciaAOtroPunto(Punto p);
}
```

1.4 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**)

Normalmente para implementar una clase partimos de una o varias 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).

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

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

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.*

Los atributos sirven para establecer los detalles del **estado interno** de los objetos. Son un conjunto de variables, cada una de un tipo. Los métodos son unidades de programa que indican la forma concreta de **consultar o modificar** las propiedades de un objeto determinado, o bien realizan operaciones sobre el objeto.

1.5 Otros conceptos y ventajas de la POO

Para hacer programas sin errores es muy importante la facilidad de depuración del entorno o lenguaje con el que trabajemos. Después de hechos los programas deben ser mantenidos, lo que puede implicar la modificación o extensión de las funcionalidades de los objetos. La capacidad para **mantener** un programa está relacionada con el buen diseño de sus módulos, la encapsulación de los conceptos relevantes, la posibilidad de reutilización y la cantidad de software reutilizado y la comprensibilidad del mismo. Todas estas características son más fáciles de conseguir usando lenguajes orientados a objetos y en particular Java.

- **Modularidad**: es la característica por la cual un programa de ordenador está compuesto de partes separadas a las que llamamos módulos. La modularidad es una característica importante para la escalabilidad y comprensión de programas, además

de ahorrar trabajo y tiempo en el desarrollo. En Java las unidades modulares son fundamentalmente las clases que se pueden agregar, junto con las interfaces, en unidades modulares mayores que son los paquetes.

- **Encapsulación**: es la capacidad de ocultar los detalles de implementación. En concreto los detalles de implementación del estado de un objeto y los detalles de implementación del cuerpo de los métodos. Esta capacidad se consigue en Java mediante la separación entre interfaces y clases, y mediante la declaración de atributos privados en las clases, como veremos más adelante. Los clientes de un objeto, es decir, los otros objetos que utilizan a ese objeto, interactúan con él a través del contrato ofrecido (interfaz). El código de los métodos queda oculto a los clientes del objeto.
- **Reutilización**: una vez implementada una clase de objetos, puede ser usada por otros programadores ignorando detalles de implementación. Las técnicas de reutilización pueden ser de distintos tipos. Una de ellas es la herencia, que se verá más adelante.
- **Facilidad de comprensión o legibilidad:** es la facilidad para entender un programa. La legibilidad de un programa aumenta si está bien estructurado en módulos y se ha usado la encapsulación adecuadamente.
- **Cohesión**: se consigue cuando cada objeto tiene perfectamente definida su funcionalidad (y por tanto, sus responsabilidades dentro de la aplicación). Cuanto más precisa y atómica sea la funcionalidad de cada objeto y de cada método de cada objeto, más cohesiva es una solución. Esto redunda en mayor facilidad de depuración y extensibilidad.
- **Bajo acoplamiento**: el acoplamiento es la dependencia entre los objetos. Dos objetos están muy acoplados si hacer cambios en las propiedades o la funcionalidad de uno de ellos repercute en tener que cambiar muchos detalles del otro. Si existen muchas dependencias entre todos los tipos de objetos, se hace muy difícil realizar cualquier cambio a la aplicación. Es por tanto deseable conseguir un bajo acoplamiento, para lo cual se emplean técnicas como los patrones de diseño, que no estudiaremos en esta asignatura.

1.6 Estructura de un programa en Java

Un programa en Java está formado por un conjunto de declaraciones de tipos, interfaces y clases. Un programa puede estar en dos modos distintos. En el **modo de compilación** está cuando estamos escribiendo las clases e interfaces. En este modo, a medida que vamos escribiendo, el entorno va detectado si las expresiones que escribimos están bien formadas. Si el entorno no detecta errores entonces diremos que el programa ha compilado bien y por lo tanto está listo para ser ejecutado. Se llama modo de compilación porque el encargado de detectar los errores en nuestros programas, además de preparar el programa para poder ser ejecutado, es otro programa que denominamos comúnmente **compilador**. En el **modo de ejecución** se está cuando queremos obtener los resultados de un programa que hemos escrito previamente. Decimos que ejecutamos el programa. Para poder ejecutar un programa este debe haber compilado con éxito previamente y tener un método especial denominado método principal (también denominado programa principal). En otro caso no lo podremos ejecutar.

En el modo de ejecución pueden aparecer nuevos errores. Los errores que pueden ser detectados en el modo de compilación y los que se detectan en el modo de ejecución son diferentes. Hablaremos de ellos más adelante. Cuando queremos ejecutar un programa en Java éste empieza a funcionar en la clase concreta elegida de entre las que tengan un método de nombre *main*. Es decir, un programa Java empieza a ejecutarse por el método *main* de una clase seleccionada.

Las interfaces y clases diseñadas más arriba forman un programa. Necesitamos una que tenga un método *main*. En el ejemplo siguiente se presenta una que sirve para comprobar el buen funcionamiento del tipo Punto.

```
package test;
import punto.*;

public class TestPunto {
	public static void main(String[] args) {
		Punto p = new PuntoImpl(2.0, 3.0);
		System.out.println("Punto:" + p);
		p.setX(3.0);
		System.out.println("Punto:" + p);
		p.setY(2.0);
		System.out.println("Punto:" + p);
	}
}
```

El programa empieza a ejecutarse el método *main* de la clase seleccionada que en este caso es *TestPunto*. Esta ejecución sólo puede llevarse a cabo cuando han sido eliminados todos los posibles errores en tiempo de compilación y por lo tanto las expresiones están bien formadas.

En la línea

```
Punto p = new PuntoImpl(2.0, 3.0);
```

Se declara p como un objeto de tipo *Punto* y se construye un objeto nuevo, con estado `(2.0, 3.0)`, mediante un constructor de la clase *PuntoImpl*. El nuevo objeto construido es asignado a *p,* por lo que *p* pasa a ser un objeto con estado `(2.0, 3.0)`.

Las instrucciones System.out.println(…) permiten mostrar en pantalla las cadenas de texto que se pasan como parámetros entre los paréntesis (se verá más adelante en este tema).

En la línea

```
p.setX(3.0);
```

se invoca al método *setX(Double X),* con parámetro real 3.0, sobre el objeto *p* para modificar en su estado la propiedad *X* al valor 3.

Los resultados esperados en la consola son:

```
Punto: (2.0,3.0)
Punto: (3.0,3.0)
Punto: (3.0,2.0)
```

2. Elementos básicos del lenguaje

2.1 Identificadores

Son palabras que permiten referenciar los diversos elementos que constituyen el código. Es decir, sirven para nombrar a clases, interfaces, métodos, atributos, variables, parámetros y paquetes. Para nombrar a estos elementos, hay que seguir unas determinadas reglas para que puedan ser entendidos por el compilador. Los identificadores se construyen mediante una secuencia de letras, dígitos, o los símbolos _ y $. En cualquier caso, se debe observar que:

- No pueden coincidir con **palabras reservadas** de Java (ver más adelante)
- Deben comenzar por una letra, _ o $, aunque estos dos últimos no son aconsejables.
- Pueden tener cualquier longitud.
- Son sensibles a las mayúsculas, por ejemplo, el identificador min es distinto de MIN o de Min.

Ejemplos de identificadores válidos son los siguientes:

```
tiempo, distancia1, caso_A, PI, velocidad_de_la_luz
```

Por el contrario, los siguientes nombres no son válidos (¿por qué?)

```
1_valor, tiempo-total, dolares%, final
```

En general, es muy aconsejable elegir los nombres de los identificadores de forma que permitan conocer a simple vista qué representan, utilizando para ello tantos caracteres como sean necesarios. Esto simplifica enormemente la tarea de programación y –sobre todo– de corrección y mantenimiento de los programas. Es cierto que los nombres largos son más laboriosos de teclear, pero, en general, resulta rentable tomarse esa pequeña molestia. Unas reglas aconsejables para los identificadores son las siguientes:

- Las variables normalmente tendrán nombres de sustantivos y se escribirán con minúsculas, salvo cuando estén formadas por dos o más palabras, en cuyo caso, el primer carácter de cada palabra se escribirá en mayúscula. Por ejemplo: `salario`, `salarioBase`, `edadJubilacion`.
- Los identificadores de constantes (datos que no van a cambiar durante la ejecución del programa) se deben escribir con todos los caracteres con mayúsculas; si el identificador está compuesto por varias palabras es aconsejable separarlas con el guion bajo (_). Por ejemplo: `PI`, `PRIMER_VALOR`, `EDAD_MINIMA`.
- Los identificadores de métodos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras en mayúscula: `getX`, `setX`, `incrementaVolumen`, etc.
- Los identificadores de clases e interfaces se deben escribir con el primer carácter de cada palabra en mayúsculas y el resto en minúsculas: `Punto`, `PuntoImpl`, etc.

2.2 Palabras reservadas de Java

Una palabra reservada es una palabra que tiene un significado especial para el compilador de un lenguaje, y, por lo tanto, no puede ser utilizada como identificador. Algunos ejemplos de palabras reservadas son `main`, `int`, `return`, `if` o `for`.

2.3 Literales

Son elementos del lenguaje que permiten representar valores constantes de los distintos tipos del lenguaje. Por ejemplo, 23 es un literal de tipo int, 3.14 es un literal de tipo double, “Guadalquivir” es un literal de tipo String y true y false son los dos literales que corresponden al tipo boolean. Para los objetos existe un literal null que representa un objeto sin valor.

2.4 Comentarios

Los comentarios son un tipo especial de separadores que sirven para explicar o aclarar algunas sentencias del código, por parte del programador, y ayudar a su prueba y mantenimiento. De esta forma, se intenta que el código pueda ser entendido por una persona diferente o por el propio programador algún tiempo después. Los comentarios son ignorados por el compilador.

En Java existen comentarios de línea, que se marcan con //, y bloques de comentarios, que comienzan con /* y terminan con */.

```
// Este es un comentario de una línea
/* Este es un bloque de comentario
   que ocupa varias líneas
*/
```

3. Tipos de datos

Los **tipos de datos básicos, nativos o primitivos** de Java son:

- **int, long**: representan valores de tipo entero y sus operaciones. Ejemplo: 3
- **float, double:** representan valores de tipo real y sus operaciones. Ejemplo: 4.5
- **boolean**: representa valores de tipo lógico y sus operaciones. Los valores de los tipos lógicos son dos, **true** y **false**
- **char**: representa un caracter de un alfabeto determinado. Ejemplo: ´d´;
- **void**: es un tipo que no tiene ningún valor.

Junto a los tipos anteriores en Java también se pueden definir tipos nuevos mediante la cláusula **enum.** Un tipo enumerado puede tomar un conjunto determinado de valores, que se enumeran de forma explícita en su declaración. Por ejemplo, en Java el tipo *Color* podemos definirlo como un enumerado con seis valores: rojo, naranja, amarillo, verde, azul y violeta.

```
public enum Color {
      ROJO, NARANJA, AMARILLO, VERDE, AZUL, VIOLETA
}
```

El tipo **String** representa una secuencia de caracteres, como por ejemplo la cadena “Hola”. El tipo cadena tiene, entre otras, la operación de concatenación, que en Java se representa mediante el operador **+**. Más adelante veremos otras operaciones de este tipo.

3.1 Tipos envoltura (wrappers)

Los tipos básicos son herencia de lenguajes de programación anteriores a Java. Por cada tipo básico se tiene disponible un tipo envoltura (*wrapper*):

- **Byte** para byte
- **Short** para short
- **Integer** para int
- **Long** para long
- **Boolean** para boolean
- **Float** para float
- **Double** para double
- **Character** para char
- **Void** para void

Los tipos envoltura añaden funcionalidad a los tipos primitivos. Esta funcionalidad añadida y los detalles que diferencian a uno y otros los iremos viendo más adelante.

3.2 Envolturas y concepto de inmutabilidad

Los objetos de tipos envoltura son inmutables. Un objeto inmutable se caracteriza porque:

- Las propiedades son fijadas por el constructor cuando el objeto se crea. Estas propiedades, como su valor, no pueden variar. Los métodos *set* no existen o son innecesarios.
- Si existen métodos que realicen operaciones sobre las propiedades del objeto, el resultado es otra instancia del tipo que contiene los datos modificados. Esto tiene consecuencias importantes, sobre todo, cuando los tipos envolturas se pasan como parámetros de funciones como veremos más adelante.

Además de las envolturas existen otros tipos que también son inmutables, como por ejemplo las cadenas de caracteres (*String*).

4. Variables y Constantes

4.1 Variables

Las **variables** son elementos del lenguaje que permiten guardar y acceder a los datos que se manejan. En Java es necesario declararlas antes de usarlas en cualquier parte del código y, por convenio, se escriben en minúsculas. Mediante la declaración indicamos que la variable guardará un valor del tipo declarado. Mediante una asignación podemos dar un nuevo valor a la variable.

Algunos ejemplos de declaraciones de variables son los siguientes:

```
int valor;
Double a1 = 2.25, a2 = 7.0;
char c = 'T';
String cadena= "Curso Java";
Color relleno = Color.AZUL;
```

En la primera línea del ejemplo se declara una variable, *valor*, de tipo entero; en la segunda línea se declaran e inicializan dos variables, *a1* y *a2*, de tipo *Double*; en la tercera, se declara una variable, *c*, de tipo *char* y se inicializa; en la cuarta, se declara e inicializa la variable *cadena* de tipo *String*. Finalmente, en la última línea se declara la variable *relleno* del tipo enumerado *Color* y se inicializa con el valor AZUL.

Cada declaración tiene un **ámbito**. Por **ámbito de una declaración** entendemos el segmento de programa donde esta declaración tiene validez. Más adelante iremos viendo los ámbitos asociados a cada declaración. En la mayoría de los casos, en Java un ámbito está delimitado por los símbolos {…} (llaves).

4.2 Constantes

Las **constantes** son elementos del lenguaje que permiten guardar y referenciar datos que van a permanecer invariables durante la ejecución del código. La declaración de una constante comienza por la palabra reservada *final*.

Ejemplos de declaraciones de constantes:

```
final int DIAS_SEMANA = 7;
final Double PI = 3.1415926;
final String TITULO = "E.T.S. de Ingeniería Informática";
```

En este ejemplo se declaran tres constantes, `DIAS_SEMANA`, `PI`, y `TITULO`, de tipo *int*, *Double* y *String*, respectivamente. Note que, por convención, los nombres de constantes se escriben en mayúsculas.

5. Expresiones y operadores

5.1 Expresiones

Una **expresión** se forma con identificadores, valores constantes y operadores. Toda **expresión bien formada** tiene asociado un valor y un tipo. Por ejemplo:

```
edad >= 30
(2 + peso) / 3
Color.ROJO
```

La primera línea muestra una expresión de tipo *boolean*. La segunda, una expresión de tipo de tipo *Double.* Y la tercera, una de tipo *Color*.

Mediante una **asignación** (representada por **=**) podemos dar nuevos valores a una variable. La asignación es un operador que da el valor de la expresión de la derecha a la variable que tiene a la izquierda. Una expresión formada con un operador de asignación tiene como tipo resultante el de la variable de la izquierda y como valor resultante el valor asignado a esa variable (la de la izquierda). Por ejemplo:

```
Double precio;
precio = 4.5 * peso + 34;
```

Si la variable peso, almacena el valor 2. La expresión `precio = 4.5 * peso + 34;` tiene como valor 43, y como tipo `Double`.

Una declaración puede ser combinada con una asignación. Como por ejemplo:

```
Integer edad = 30;
Double peso = 46.5;
String s1 = "Hola ", s2 = "Anterior";
Color c = Color.VERDE;
```

En este caso decimos que hemos declarado la variable y hemos hecho la **inicialización** de la misma. Es decir, le hemos dado un **valor inicial**.

5.2 Operadores

Los operadores más habituales en Java son los siguientes:

- Operadores aritméticos: + (suma), - (resta), * (producto), / (división) y % (módulo)
- Operadores lógicos: && (and), || (or) y ! (not)
- Operadores relacionales: > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que), == (igual que), != (distinto de)
- Operadores de asignación: = y los abreviados: +=, -=, etc; ++ (incremento) y -- (decremento)

5.3 Precedencia y asociatividad de los operadores

El resultado de una expresión depende del orden en que se ejecutan las operaciones. Por ejemplo, si queremos calcular el valor de la expresión `3 + 4 * 2`, podemos tener distintos resultados dependiendo de qué operación se ejecuta primero. Así, si se realiza primero la suma (3 + 4) y después el producto (7 * 2), el resultado es 14; mientras que si se realiza primero el producto (4 * 2) y luego la suma (3 + 8), el resultado es 11.

Para saber en qué orden se realizan las operaciones es necesario definir unas reglas de precedencia y asociatividad. La tabla siguiente resume las reglas de precedencia y asociatividad de los principales operadores en el lenguaje Java.

Operador	Asociatividad
. [] ()	
+ - ! ++ -- (tipo) new	derecha a izquierda
* / %	izquierda a derecha
+ –	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
? :	
= += –= *= /= %=	derecha a izquierda

Los operadores que están en la misma línea tienen la misma precedencia, y las filas están en orden de precedencia decreciente. También se debe tener en cuenta que el uso de paréntesis modifica el orden de aplicación de los operadores. Si no hay paréntesis el orden de aplicación de los operadores viene definido por la precedencia y la asociatividad de los mismos definida arriba.

6. El tipo String, tipos para el manejo de fechas

6.1 Tipo String

Como hemos visto antes, el tipo String representa una secuencia o cadena de caracteres. El tamaño de un String es inmutable y, por lo tanto, no cambia una vez creado el objeto. Se representa por el método *length*.

Cada carácter de la cadena ocupa una posición, comenzando por la posición 0 y terminando por la posición *length()-1*. Para acceder al carácter que ocupa una posición dada se utiliza el método *charAt(int i)*. Por ejemplo,

```
String nombre = "Amaia";
lon = nombre.length() // El valor de lon será 5
inicial = nombre.charAt(0) // El valor de inicial será 'A'
ultima = nombre.charAt(lon - 1) // El valor de ultima será 'a'
```

No es posible modificar un carácter de la cadena una vez que se ha creado. Tampoco se pueden añadir o eliminar caracteres. El tipo *String* es, pues, inmutable.

El tipo *String* ofrece otros métodos para, entre otras cosas, decidir si la cadena contiene una secuencia dada de caracteres, buscar la primera posición de un carácter u obtener la subcadena dada por dos posiciones. Los detalles de estos métodos pueden verse en la documentación de la clase *String* en la API de Java.

6.2 Tipos para el manejo de fechas y horas

Java dispone de un conjunto de tipos para trabajar con fechas, horas, instantes temporales y duraciones, incluidos en el paquete `java.time`. Algunos de estos tipos son los siguientes:

- `LocalDate`, tipo inmutable para representar fechas (sin zona horaria), es decir, solo con día, mes y año. Por ejemplo, 03-12-2015
- `LocalTime`, tipo inmutable para representar horas sin fecha (ni zona horaria), solo con hora, minutos, segundos y nanosegundos (si son necesarios). Por ejemplo, 10:10:12.99.
- `Duration`, tipo inmutable que representa una cantidad temporal que se mueve en el rango de nanosegundos, segundos, minutos, horas o días.
- `Period`, tipo inmutable que representa una cantidad temporal que se mueve en el rango de años, meses o días.

Veamos algunas operaciones habituales con estos tipos. Comencemos por la creación de objetos de tipo fecha y hora:

```
LocalDate fecha1 = LocalDate.of(2014, Month.MAY, 23);
LocalDate fecha2 = LocalDate.of(2014, 5, 23);
LocalTime hora1 = LocalTime.of(11, 00);
LocalTime hora2 = LocalTime.of(10, 59, 59);
```

Además del método of, se pueden crear objetos con la fecha y hora del sistema en el momento en que se invoca al método. Por ejemplo,

```
LocalDate fecha3 = LocalDate.now();
LocalTime hora3 = LocalTime.now();
```

Dada una fecha o una hora, podemos acceder a una parte de ella, como por ejemplo el día de una fecha o los minutos de una hora:

```
LocalDate hoy = LocalDateTime.now();
Integer dia = hoy.getDayOfMonth();
DayOfWeek diaSemana = hoy.getDayOfWeek();
LocalDate ahora = LocalDateTime.now();
Integer minutos = ahora.getMinute();
```

También podemos sumar o restar un periodo de tiempo a una fecha u hora:

```
LocalDate f1 = LocalDate.of(2008, Month.FEBRUARY, 29);
LocalDate f2 = f1.plusYears(1);
System.out.println("Un año después..." + f2);
```

Otra operación habitual es obtener el tiempo transcurrido entre dos fechas:

```
LocalDate fechaNacimiento = LocalDate.of(2013, 12, 3);
LocalDate hoy = LocalDate.now();
Period p = fechaNacimiento.until(hoy);
Period p = Period.between (fechaNacimiento, hoy); // También así
System.out.println ("El periodo entre las dos fechas es " + p);
System.out.println ("Los años del periodo son " + p.getYears());
System.out.println ("Los meses del periodo son " + p.getMonths());
System.out.println ("Los días de periodo son " + p.getDays());
```

De manera similar para las horas, pero usando el tipo `Duration`:

```
LocalTime hora1 = LocalTime.of(15, 30);
LocalTime hora2 = LocalTime.of(15, 45);
Duration d = Duration.between(hora1, hora2);
System.out.println("Duración - " + d);
System.out.println ("Los segundos de la duración son " +
        d.getSeconds());
```

Para comparar dos fechas u horas, utilizamos los métodos `isBefore`, `isAfter` e `isEqual`:

```
LocalDate f1 = LocalDate.of (2016, Month.SEPTEMBER, 19);
LocalDate f2 = LocalDate.of (2017, Month.JANUARY, 13);
System.out.println("¿Es f1 anterior a f2? " + f1.isBefore(f2));
System.out.println("¿Es f1 posterior a f2? " + f1.isAfter(f2));
System.out.println("¿Es f1 igual a f2? " + f1.isEqual(f2));
```

Finalmente, cuando mostramos fechas y horas en la pantalla, queremos que aparezcan en un formato concreto. Los siguientes ejemplos muestran varias formas de hacerlo:

```
LocalDate fecha = LocalDate.of(2016, 12,3);
```

```
String fechaFormateada =
      fecha.format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
System.out.println("Fecha: " + fechaFormateada);
LocalTime hora = LocalTime.now();
String horaFormateada =
      hora.format(DateTimeFormatter.ofPattern("HH:mm:ss:n"));
System.out.println("Hora: " + horaFormateada);
```

La salida que se produce al ejecutar el código anterior es la siguiente:

```
Fecha: 03-12-2016
Hora: 19:00:19:56000000
```

7. Escritura de datos en pantalla

Para escribir los datos por consola usaremos una llamada al método `println`. Así, por ejemplo, si escribimos la línea de código:

```
System.out.println("El valor de la variable n es " + n + ".");
```

Suponiendo que *n* se ha declarado como variable entera y tiene el valor 5, cuando se ejecuta la línea de código anterior se obtiene por consola el siguiente resultado:

```
El valor de la variable n es 5.
```

Igual que existe el método `println`, hay un método llamado `print`. La diferencia entre ambos es que el primero imprime el carácter fin de línea al final de la cadena que se pasa como argumento. Así, si escribimos las siguientes líneas de código:

```
System.out.print("El valor de la variable n es " + n + ".");
System.out.println("Fin");
```

La palabra Fin se escribirá tras el punto, siendo el resultado

```
El valor de la variable n es 5.Fin
```

Sin embargo, si la llamada a `print` fuera sustituida por una llamada a `println`, el resultado sería

```
El valor de la variable n es 5.
Fin
```

8. Sentencias de control selectivas

Hay dos tipos de sentencias de control: las bifurcaciones y los bucles. Las bifurcaciones permiten ejecutar un bloque de sentencias u otro, pero no ambos a la vez. En este tipo de sentencias de control se encuadran las sentencias *if* y *switch*.

8.1 Sentencia if-else

La sentencia if evalúa una expresión lógica (o condición) y, según sea cierta o falsa, ejecuta un bloque de sentencias u otro.

```
if (n % 2 == 0) {
      System.out.println("El número es par ");
} else {
      System.out.println("El número es impar");
}
```

Las sentencias *if-else* pueden encadenarse haciendo que se evalúe un conjunto de condiciones y se ejecute una sola de varias opciones.

```
Float impuesto = 0.0f;
if (salario >= 5000.0) {
      impuesto = 20.0f;
} else if (salario < 5000.0 && salario >= 2500.0) {
       impuesto = 15.0f;
} else if (salario < 2500.0 && salario >= 1500.0) {
       impuesto = 10.0f;
} else if (salario > 800.0) {
      impuesto = 5.0f;
}
```

8.2 Sentencia switch

Normalmente *switch* se utiliza cuando se requiere comparar una variable de un tipo discreto con una serie de valores diferentes. En la sentencia *switch*, se indican los posibles valores que puede tomar la variable y las sentencias que se tienen que ejecutar si la variable coincide con alguno de dichos valores. Es una sentencia muy indicada para comparar una variable de un tipo enumerado con cada uno de sus posibles valores.

La expresión tiene que ser de uno de los siguientes tipos: *byte*, *short, char*, e *int*; sus envolturas, *Byte*, *Short*, *Character* e *Integer,* y los tipos enumerados (*enum*).

Ejemplo:

```
String s;
switch (dia) {
     case 1:
          s = "Lunes";
          break;
     case 2:
          s = "Martes";
          break;
     case 3:
          s = "Miércoles";
          break;
     case 4:
          s = "Jueves";
          break;
     case 5:
          s = "Viernes";
          break;
     case 6:
          s = "Sábado";
          break;
```

```
            case 7:
                 s = "Domingo";
                 break;
            default:
                 s = "Error";
     }
```

Existe también posibilidad de ejecutar el mismo bloque de sentencias para varios valores del resultado de expresión. Se haría de la siguiente forma:

```
switch (dia) {
      case 1: case 2: case 3: case 4: case 5: case 10: case 12:
            System.out.println("Día laborable");
            break;
      case 6: case 7:
            System.out.println("Fin de semana");
            break;
}
```

9. Agregados de datos

9.1 Listas

Las listas representan colecciones de elementos de un mismo tipo en los que importa cuál es el primero, el segundo, etc. Cada elemento está referenciado mediante un índice; el índice del primer elemento es el 0. Las listas pueden contener elementos duplicados.

En Java, las listas se representan mediante el tipo *List*. Para crear una lista se invoca a un constructor del tipo *List*, indicando el tipo de los elementos que va a contener la lista. Por ejemplo, la instrucción

```
List<Double> temperaturas = new LinkedList<>();
```

crea una lista de números reales. Inicialmente la lista está vacía, y podemos añadir elementos de la siguiente forma:

```
temperaturas.add(27.5);
temperaturas.add(22.0);
temperaturas.add(25.3);
```

Al añadir elementos con el método *add*, se van colocando al final de la lista. Por tanto, en el caso anterior tendríamos la lista formada por los elementos {27.5, 22.0, 25.3}. Para acceder a un elemento de la lista hemos de utilizar el método *get*, indicando la posición del elemento al que queremos acceder. Por ejemplo, para obtener la primera temperatura de la lista haríamos lo siguiente:

```
t1 = temperaturas.get(0);
```

Es importante que el índice esté dentro del rango de valores válidos, que van desde 0 hasta uno menos que el tamaño de la lista. Podemos conocer este tamaño mediante el método *size*:

```
Integer numeroElementos = temperaturas.size();
```

que en el caso del ejemplo asignaría a la variable `numeroElementos` el valor 3.

9.2 Conjuntos

El tipo *Set* de Java se corresponde con el concepto matemático de conjunto: un agregado de elementos en el que no hay orden (no se puede decir cuál es el primero, el segundo, el tercero, etc.) y donde no puede haber elementos repetidos.

La siguiente instrucción

```
Set<Character> letras = new HashSet<>();
```

crea un conjunto formado por caracteres, inicialmente vacío. A continuación podemos añadir elementos (en este caso caracteres) mediante el método *add*:

```
letras.add('A');
letras.add('V');
letras.add('E');
```

Podemos obtener el número de elementos de un conjunto con *size*, como en las listas. También podemos saber si un determinado elemento se encuentra dentro de un conjunto. La expresión

```
letras.contains('V')
```

tomará valor *true* si el conjunto `letras` contiene el elemento 'V', y *false* en caso contrario.

En un conjunto no existe un orden entre sus elementos. No obstante, existe un tipo especial de conjuntos, el *SortedSet*, en el cual los elementos sí están ordenados. Se construye de la siguiente forma:

```
SortedSet<Character> letrasOrdenadas = new TreeSet<>();
```

Cuando se añaden elementos a este conjunto, se colocan de forma ordenada. Todas las operaciones aplicables a *Set* son también válidas para *SortedSet*, que además incluye otras operaciones exclusivas.

9.3 El tipo Map

El tipo de dato *Map* permite modelar el concepto de aplicación: una relación entre los elementos de dos conjuntos de modo que a cada elemento del conjunto inicial le corresponde uno y solo un elemento del conjunto final. Los elementos del conjunto inicial se denominan claves (*keys*) y los del conjunto final valores (*values*).

Para crear un *Map* hay que indicar el tipo de las claves y el tipo de los valores. Por ejemplo, para construir un *Map* cuyas claves son cadenas de caracteres y cuyos valores son números reales, haríamos lo siguiente:

```
Map<String, Double> temperaturasCiudad = new HashMap<>();
```

Con este *Map* podemos representar las temperaturas de distintas ciudades. Observa que dos ciudades pueden tener la misma temperatura, es decir, los valores del *Map* pueden repetirse, pero no así las claves, que son únicas.

Para almacenar en el *Map* la temperatura de una ciudad haríamos lo siguiente:

```
temperaturasCiudad.put("Córdoba", 19.1);
```

El método *put* crea en el *Map* una pareja con la cadena `"Córdoba"` como clave y el número `19.1` como valor.

Para obtener el valor asociado a una clave de un *Map*, utilizamos el método *get*. Por ejemplo, la instrucción

```
Double t = temperaturasCiudad.get("Córdoba");
```

almacena en la variable `t` el valor `19.1`.

Los valores de un *Map* pueden ser de un tipo simple, como en este caso, o también pueden ser agregados, como una lista o un conjunto.

Para calcular el número de parejas de un *Map* disponemos del método *size*, al igual que en las listas y conjuntos. Existen otros métodos para obtener las claves, los valores y las parejas, que veremos más adelante.

10. Sentencias de control iterativas

Los bucles son sentencias de control que ejecutan un bloque de sentencias un número determinado de veces. En Java los bucles se implementan con las sentencias *for* y *while*.

10.1 Sentencia while

La sentencia while ejecuta el bloque de sentencias mientras la condición evaluada sea cierta. Su sintaxis es la que se muestra a continuación.

Por ejemplo, para sumar los números comprendidos entre 1 y n, haríamos lo siguiente:

```
Integer suma = 0;
int i = 1;
while (i <= n) {
       suma = suma + i;
       i++;
}
```

10.2 Sentencia for clásico

La sentencia *for* tiene la siguiente sintaxis, donde *inicialización* y *actualización* son sentencias y *condición* es una expresión lógica.

Sintaxis:

```
for (inicialización; condición; actualización) {
    sentencia-1;
    sentencia-2;
       ...
    sentencia-n;
}
```

¿Cómo funciona el *for*? Cuando el control de un programa llega a una sentencia *for*, lo primero que se ejecuta es la *inicialización*. A continuación, se evalúa la *condición*, si es cierta (valor distinto de cero), se ejecuta el bloque de sentencias de 1 a n. Una vez terminado el bloque, se

ejecuta la sentencia de actualización antes de volver a evaluar la condición, si ésta fuera cierta de nuevo se ejecutaría el bloque y la actualización, y así repetidas veces hasta que la condición fuera falsa y abandonáramos el bucle.

El bucle *while* anterior, escrito con una sentencia *for*, sería así:

```
Integer suma = 0;
for (int i = 0; i <= n; i++) {
	suma = suma + i;
}
```

10.3 Sentencia for extendido

Existe una variante del *for* que se utiliza para recorrer agregados, como listas o conjuntos. Supongamos por ejemplo que queremos calcular el valor medio de las temperaturas de la lista `temperaturas`. Lo podríamos hacer mediante un *for* clásico:

```
Double suma = 0.0;
for (int i = 0; i < temperaturas.size(); i++) {
	suma = suma + temperaturas.get(i);
}
Double temperaturaMedia = suma / temperaturas.size();
```

Existe una forma alternativa, usando un *for* extendido, que resulta más simple. Se haría de la forma siguiente:

```
Double suma = 0.0;
for (Double t: temperaturas) {
	suma = suma + t;
}
Double temperaturaMedia = suma / temperaturas.size();
```

La variable `t` va recorriendo las temperaturas de la lista, desde la primera hasta la última.

10.4 Sentencia break

La sentencia **break** dentro de un bloque de sentencias que forma parte de una estructura iterativa hace que el flujo de programa salte fuera del bloque, dando por finalizada la ejecución de la sentencia iterativa. Imagina por ejemplo que queremos saber si existe una temperatura por debajo de cero en la lista. Si al recorrer la lista encontramos una, ya sabemos que la hay, así que no es necesario seguir recorriendo el resto de temperaturas. En ese caso usamos un break para salir del bucle:

```
Boolean temperaturaNegativa = false;
for (Double t: temperaturas) {
	if (t < 0) {
		temperaturaNegativa = true;
		break;
	}
}
```

Al terminar la ejecución de este código, la variable `temperaturaNegativa` tendrá valor *true* si la lista de números contiene al menos un valor negativo, y *false* si no contiene ninguno.

11. Streams

A partir de la versión 8, Java introduce una nueva y potente forma de realizar operaciones con agregados de datos. Para ello nos proporciona varios elementos que podemos combinar para realizar cualquier tratamiento que necesitemos sobre un agregado de datos.

El elemento fundamental es el **Stream**. Un *stream* es un agregado de elementos. Por ejemplo, las canciones de un artista, las películas estrenadas el año pasado o las temperaturas diarias en una ciudad. Los *streams* se obtienen habitualmente a partir de una lista o conjunto, y podemos realizar sobre ellos diversos tratamientos.

Supongamos por ejemplo nuestra lista de temperaturas. Si queremos conocer el número de ellas que están por debajo de cero, podríamos hacerlo mediante el siguiente bucle:

```
long bajoCero = 0;
for (Double t: temperaturas) {
	if (t < 0) {
		bajoCero++;
	}
}
System.out.println("Hay " + bajoCero + " temperaturas bajo cero.");
```

Los *streams* nos ofrecen una alternativa a este bucle. Solo tenemos que transformar la lista en un *stream* y aplicar las operaciones propias de los *streams*. El mismo tratamiento anterior se haría de esta otra forma:

```
long bajoCero = temperaturas.stream()
	.filter(x -> x < 0)
	.count();
System.out.println("Hay " + bajoCero + " temperaturas bajo cero.");
```

Veamos un segundo ejemplo. En este caso queremos saber si existe alguna temperatura por encima de los 40 grados. Sería así:

```
boolean algunaPorEncima = temperaturas.stream()
	.anyMatch(x -> x > 40);
if (algunaPorEncima) {
	System.out.println("Hay al menos una temperatura mayor de 40º");
}
```

El paso de usar bucles a usar streams supone el paso de un paradigma imperativo a un paradigma funcional, en el cual indicamos qué es lo que queremos hacer, y no cómo hacerlo; de eso se encarga ya el propio lenguaje, simplificando así nuestra tarea.