Tema 1. Introducción a la Programación Orientada a Objetos

Autor: Miguel Toro Bonilla
Revisión: José C. Riquelme, Antonia M. Reina

Contenido

En este tema vamos a definir una serie de conceptos básicos, para situar el contexto en el que nos vamos a mover en la asignatura, y daremos una vista panorámica a la programación orientada a objetos y, en particular, a la programación usando el lenguaje Java. Algunos de los conceptos explicados en este capítulo los volveremos a ver más detallados o refinados en los temas siguientes.

1. Conceptos básicos

Programa informático. Es una lista de instrucciones con una finalidad concreta que se ejecutan habitualmente de modo secuencial (una detrás de otra) aunque en ciertos ordenadores es posible también en paralelo. Estas instrucciones se escriben en un **fichero fuente** siguiendo unas reglas que vienen dadas por un lenguaje de programación concreto. Normalmente un programa procesa unos datos a partir de una entrada de información y presenta al acabar unos resultados de salida.

Sistema Operativo. Es un programa o conjunto de programas que en un sistema informático gestionan los recursos de hardware y facilitan el uso de programas de aplicación como el explorador de ficheros o el navegador web.

Lenguajes de programación. Las instrucciones de un programa deben estar escritas en un lenguaje comprensible por el ordenador y, dependiendo de la cercanía de ese lenguaje a la máquina concreta, se

hablan de lenguajes de bajo o alto nivel. El lenguaje de más bajo nivel es el lenguaje máquina binario constituido por un conjunto de unos y ceros, que evidentemente es incomprensible para el ser humano. A partir de ahí, los lenguajes ensambladores constituyen el siguiente nivel, comprensibles pero difíciles de programar. Finalmente, los lenguajes de alto nivel son capaces de escribir instrucciones con una estructura sintáctica comprensible por el programador y que convenientemente "traducidas" son capaces de ejecutarse en un ordenador. Lenguajes de alto nivel primitivos como Fortran o Cobol de mediados del siglo XX han ido evolucionando hasta lenguajes como Java o Python. Todos los programas excepto los escritos en código máquina binario deben ser "traducidos" para que puedan ser ejecutados.

Compilador. Un compilador es un programa que "traduce" un conjunto de instrucciones escritas en un lenguaje de alto nivel a un lenguaje comprensible por el ordenador. Normalmente el compilador está integrado con otras funcionalidades constituyendo un **Entorno de Desarrollo Integrado** (IDE en inglés). Un IDE además del compilador, suele disponer de un editor, un depurador y otras herramientas que facilitan la construcción y prueba de programas. A lo largo del curso utilizaremos dos IDE's, Eclipse, para trabajar con Java, y Visual Studio, para trabajar con C.

Una de las principales tareas de un buen compilador es ayudar al programador a descubrir los **errores sintácticos** del programa. Como se ha dicho anteriormente, los lenguajes de alto nivel tienen una sintaxis bastante estricta, es decir, la estructura de cada instrucción y sus relaciones con las demás están fuertemente condicionadas por un conjunto de reglas sintácticas. Estas reglas obligan al programador a ser muy cuidadoso en la escritura de un programa para que pueda ser traducido por el compilador. Un buen IDE debe proporcionar información adecuada sobre por qué una instrucción no está bien escrita para que el programador pueda corregirla. Otra tarea básica del IDE es el depurador o facilidad que ofrece la posibilidad de ejecutar paso a paso un programa controlando si el orden de las sentencias y los datos que procesa son los que se esperaba o no. En los dos lenguajes de programación que vamos a estudiar en la asignatura (Java y C), tendremos que utilizar el compilador para traducir los programas que escribamos en estos lenguajes de alto nivel a código intermedio. La Figura 1 muestra cómo se realiza el proceso de compilación el Java y C, respectivamente.

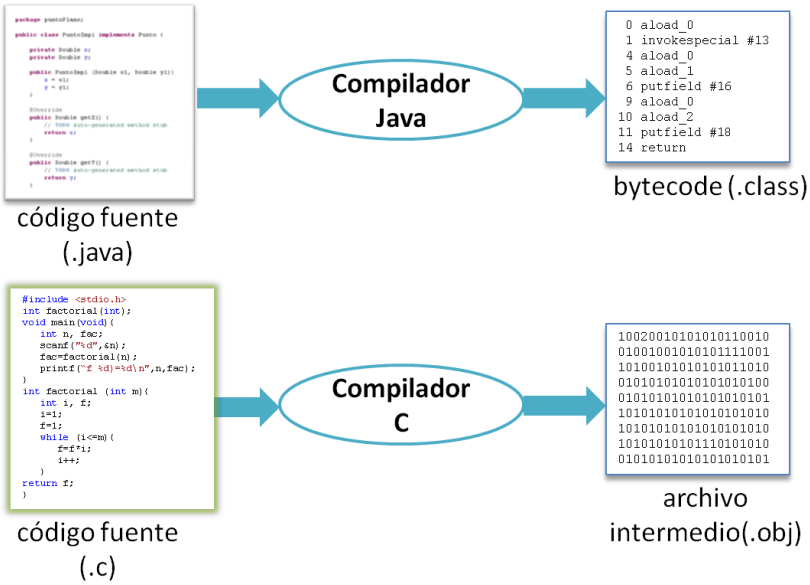


Figura 1. Proceso de compilación en Java y C

Intérprete. Un intérprete, según Wikipedia, es un programa capaz de analizar y ejecutar otros programas, escritos en un lenguaje de alto nivel. Los intérpretes se diferencian de los compiladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción. La Figura 2 muestra el proceso de interpretación en Java. Note que no hay un proceso equivalente en C. Por eso se dice que C es un lenguaje compilado, mientras que Java es compilado e interpretado.

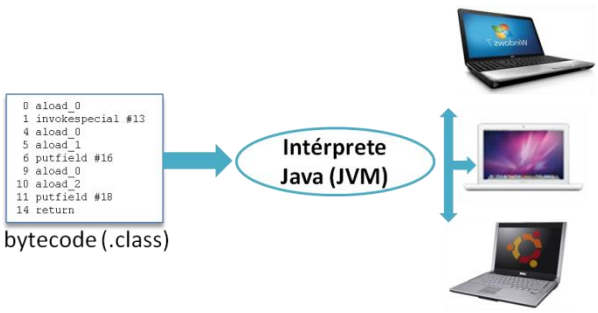


Figura 2. Proceso de interpretación en Java

Ingeniería del Software. El desarrollo de aplicaciones software con las restricciones habituales de todo proceso productivo, esto es, con el menor esfuerzo y coste y la mejor calidad posible, requiere de una metodología propia. La ingeniería del software es la disciplina que trata de dar un enfoque sistemático y disciplinado al diseño, desarrollo y mantenimiento del software. Desde este punto de vista, en la actualidad podemos afirmar que la programación de ordenadores no es un arte, a pesar de que durante años la mala interpretación del título de la obra de Knuth "The Art of Computer Programming" parecía decir lo contrario. La programación de ordenadores debe ser entonces contemplada como un proceso ingenieril, base de la ingeniería del software y pilar fundamental de numerosos problemas que resuelven las distintas ingenierías.

2. Lenguajes de programación

En este apartado vamos a definir algunos conceptos que se manejan en cualquier lenguaje de programación. Aunque los ejemplos que vamos a usar son del lenguaje Java, se pueden extrapolar a cualquier otro lenguaje de programación.

2.1. Conceptos básicos de los Lenguajes de Programación

Para construir programas necesitamos varios elementos de partida: **identificadores, tipos y declaraciones.** Los **identificadores** son secuencias de caracteres que sirven para dar nombre a una determinada entidad que el programa va a usar (variables, constantes, objetos, clases, interfaces, métodos, etc). Un **tipo** representa el conjunto de valores que una entidad puede tomar y las operaciones que podemos hacer con esos valores. Mediante una **declaración** decimos que se va a usar una entidad que tiene como nombre un identificador dado y es del tipo indicado. Cada entidad declarada de un tipo tendrá, en un momento dado, un valor de ese tipo. Por ejemplo, dada una variable de tipo entero, tomará valores enteros. Si una constante se define cadena de caracteres su valor será una cadena de caracteres, etc.

Los lenguajes de programación proporcionan un conjunto de tipos predefinidos. Así, por ejemplo, en Java existen los tipos:

- **int, Integer, long, Long**, que representan valores de tipo entero y sus operaciones. Ejemplo: 3
- **float, Float, double, Double**, que representan valores de tipo real y sus operaciones. Ejemplo: 4.5
- **boolean, Boolean**, que representan valores de tipo lógico y sus operaciones. Los valores de los tipos lógicos son dos: **true, false**
- **String**: Representan secuencias de caracteres y sus operaciones. Ejemplo: “Hola”.
- **char, Character**: Representan un caracter de un alfabeto determinado. Ejemplo: ´d´;
- **void, Void**: Es un tipo que no tiene ningún valor.

Los tipos numéricos tienen las operaciones usuales. El tipo cadena tiene, entre otras, la operación de concatenación, que en Java se representa mediante el operador **+.**

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
}
```

Además, los tipos enumerados tipos tienen operaciones de igualdad y de orden. Los detalles completos de los tipos anteriores (operadores disponibles, conjunto de valores, representación de las constantes…) serán vistos más adelante.

Una **variable** es una entidad que puede cambiar el valor que almacena. Antes de usar una variable en Java, se debe declarar su tipo y su nombre. Algunos ejemplos de declaración de variables son los siguientes:

```
Integer edad;
Double peso;
String s1, s2;
Color c;
```

En la primera línea declaramos una nueva entidad (variable en este caso) con nombre *edad* y de tipo *Integer.* Luego, otra de nombre *peso* y de tipo *Double.* En la tercera línea se declaran las entidades *s1 y s2* de tipo *String* y, en la última línea, se declara *c* de tipo *Color.*

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).

A las entidades que no pueden cambiar el valor que almacenan las se les llama **constantes**.

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. El operador **=** tiene siempre un sentido de derecha a izquierda y no debemos confundirlo con el operador de igualdad que veremos más adelante. La asignación funciona como si pusiéramos **variable ← expresión** aunque se representa por el símbolo **=.** 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**.

Las expresiones que escribamos deben estar bien formadas. Si no lo están aparecerán errores que serán detectados cuando las escribimos. Son los denominados **errores en modo de compilación**. Aunque las expresiones estén bien formadas, éstas pueden mostrar errores cuando intentamos ejecutarlas. En este caso, los errores son denominados **errores en tiempo de ejecución**. Por ejemplo, la expresión siguiente está bien formada, no produce errores en tiempo de compilación, pero cuando se intenta ejecutar se produce un error en tiempo de ejecución. Esto es debido a que no se puede hacer una división por cero.

```
int a;
a = 3*4/0;
```

Las expresiones se combinan en **secuencias de expresiones.** Como su nombre indica son secuencias de expresiones y, en Java, están acabadas en punto y coma (;).

3. Conceptos básicos de Programación Orientada a Objetos

Los **Lenguajes Orientados a Objetos** (LOO) 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….

En el aprendizaje de la Programación Orientada a Objetos (POO) usaremos el lenguaje Java para concretar las ideas explicadas.

3.1. 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. Éstas 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**

3.2. Objeto

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

- 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.**

3.3. Interfaz (interface)

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.

Cuando empezamos a pensar en un tipo nuevo de objeto debemos modelarlo. Para ello estableceremos sus propiedades y cuáles se pueden modificar y cuáles no, debemos indicar qué cosas puede hacer el objeto. Para ello definiremos un **contrato** que indique cómo es posible interaccionar con el objeto. Ese **contrato** establece cómo se puede relacionar un objeto con los demás que le rodean. Una parte del contrato se recoge en una interfaz.

Formalmente una interfaz (**interface**) contiene, principalmente, las **signaturas** de los métodos que nos permiten consultar y/o cambiar las propiedades de los objetos. 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 la signatura de un método las llamaremos también **cabecera del método.** Una tercera forma de denominar la signatura es **firma del método.**

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

Para modelar un objeto es conveniente seguir una metodología, con objeto de obtener una **interfaz** adecuada para el tipo de objeto. Una metodología apropiada puede ser:

1. Establecer las propiedades relevantes que tiene el objeto. Las propiedades pueden depender de parámetros.
2. Para cada propiedad indicar su tipo, si es consultable o no, posibles parámetros de los que pueda depender.
3. Indicar las relaciones o ligaduras entre propiedades si las hubiera e indicar las propiedades derivadas y no derivadas.
4. Por cada propiedad consultable escribir un método que comience por *get* y continúe con el nombre de la propiedad. Si la propiedad depende de parámetros, este método tomará los parámetros de los que dependa y devolverá valores del tipo de la propiedad. Estos métodos no modificarán el estado del objeto.
5. Por cada propiedad modificable escribir un método que comience por *set* y continúe con el nombre de la propiedad. Este método tomará al menos un parámetro con valores del tipo de la propiedad y no devolverá nada. Las propiedades derivadas no son modificables.
6. Junto a las propiedades el tipo puede definir operaciones sobre el objeto. Una operación (que puede tener parámetros) es una forma de cambiar, desde fuera las propiedades de un objeto. Por cada operación escribiremos un método que dependerá de los mismos parámetros que la operación.

Esta metodología la refinaremos en los siguientes temas, cuando veamos el concepto de propiedad compartida.

Un ejemplo de aplicación de esta metodología es el siguiente: Supongamos que queremos modelar un objeto que llamaremos Punto (y que representará un punto del plano). Para ello definiremos una interfaz que representa un **contrato** con los posibles usuarios del objeto. 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 contrato de Punto obtenida con la metodología 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

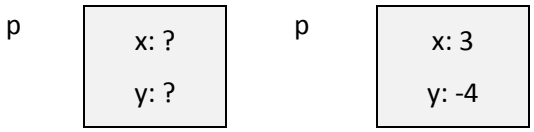
general, a las variables cuyos tipos vengan definidos por una interface o una clase. En el siguiente ejemplo se declara la variable *p,* que representa un objeto de tipo punto:

```
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** sobre el objeto *p*. Por ejemplo:

```
p.setX(3);
p.setY(-4);
```

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,-4). 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 cambia el estado del objeto.

Hay expresiones que no están bien formadas. Cuando usamos el operador punto (.) la expresión correspondiente sólo está bien formada si el método que está a la derecha del punto es un método del tipo con el que hemos declarado la variable. Hay un segundo requisito para que la expresión esté bien formada: los valores proporcionados como parámetros (que denominaremos **parámetros reales**) deben ser del mismo tipo que los parámetros que tiene declarados el método correspondiente. Los parámetros declarados en un método los denominaremos **parámetros formales**.

Las expresiones siguientes no están bien formadas.

```
p.getZ();
p.setU(4.1);
p.setX("4");
```

La primera y la segunda no están bien formadas porque los métodos *getZ*, *getU* no están declarados en el tipo Punto (tipo de la variable *p*). La tercera tampoco está bien formada. En este caso, *setX* sí es un método de Punto, pero el valor proporcionado como parámetro real no es de tipo *Double*.

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:
Punto

- Propiedades
  - X, Real, Consultable, modificable
  - Y, Real, Consultable, modificable
- Operaciones:
  - DistanciaAOtroPunto(Punto p), Real, consultable, derivada

La interfaz asociada es:

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

Diseñemos ahora el tipo *Círculo.*
Circulo:

- Propiedades:
  - Centro: De tipo Punto, consultable y modificable
  - Radio: De tipo Real, consultable y modificable
  - Area: De tipo Real, consultable, derivada de Radio
  - Longitud: De tipo Real, consultable, derivada de Radio

La interfaz asociada es:

```
public interface Circulo {
	Punto getCentro();
	Double getRadio();
	Double getArea();
	Double getLongitud();
	void setCentro(Punto p);
	void setRadio(Double r);
}
```

La sintaxis para declarar un interfaz responde al siguiente patrón:

[Modificadores] **interface** *NombreInterfazHija* [**extends** *NombreInterfacesPadres [,…]*] {
	[….]
	[cabeceras de métodos]
}

La cláusula **extends** la estudiaremos más adelante, y, como puede verse al situarse entre corchetes, es opcional, es decir, podemos declarar interfaces en las que no aparezca la cláusula **extends**.

Puede haber métodos con el mismo nombre pero diferente cabecera. Es lo que se denomina **sobrecarga de métodos**. En Java en particular, es posible la sobrecarga de métodos, pero con una restricción: no puede haber dos métodos con el mismo nombre, los mismos parámetros formales y distintos tipos devueltos. Esto es debido a que en Java, en particular, dos métodos son iguales si tienen el mismo nombre y los mismos parámetros formales. Por lo tanto, en Java, para **identificar un método de forma única** debemos indicar su nombre y sus parámetros formales.

Tipo definido por un interfaz

Como hemos visto arriba al declarar un interfaz estamos definiendo un tipo nuevo. Este tipo tiene el mismo identificador (o nombre) que la interfaz y está formado por todos los métodos declarados en ella. Con ese nuevo tipo se pueden declarar objetos, construir nuevas expresiones y llevar a cabo las asignaciones correspondientes. Por ejemplo, si *c* es de tipo *Circulo* y *x* de tipo *Double.* La expresión siguiente está bien formada y el valor de la variable x es actualizado al valor de la propiedad X, contenida en el estado del centro del círculo.

```
x = c.getCentro().getX();
```

Como vemos, cuando definimos un interfaz definimos un tipo nuevo. En general al definir un tipo nuevo usamos otros tipos ya definidos. En este caso, el tipo *Punto* usa el tipo *Double* ya definido en Java. El tipo *Circulo* usa el tipo *Double* y el tipo *Punto*.

3.4. 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).

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 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, dentro de nuestra asignatura, haremos que el nombre de la clase que implementa una interfaz sea el mismo del interfaz, seguido de Impl. Por ejemplo, una clase que implemente la interfaz *Punto* se llamará *PuntoImpl.*

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;
```

Y para una clase que implemente la interfaz Circulo tendríamos:

```
private Double radio;
private Punto centro;
```

Note en este caso que no definimos ningún atributo para las propiedades derivadas área y longitud del círculo.

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 (compartidas o no). Las propiedades derivadas, es decir, aquellas que se pueden calcular a partir de otras, no tienen un atributo asociado en general, aunque más adelante aprenderemos otras posibilidades.

Existen otros tipos de modificadores, como ***static***, que se usan cuando se trabaja con propiedades compartidas, aunque estos los veremos con más detalles más adelante.

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.

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 aquéllos 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.

Un ejemplo de clase que implementa Punto es el siguiente:

```
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);
	}
	//Representación como cadena
	public String toString() {
		String s = "(" + this.getX() + "," + this.getY() + ")";
		return s;
	}
}
```

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) 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.get(x)* es equivalente a *getX()*
- *this.distanciaAOtroPunto(origen)* es equivalente a *distanciaAOtroPunto(origen)*

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

Como los métodos, las clases también tienen dos partes: cabecera y cuerpo. Como ya se vio anteriormente, La cabecera sigue el siguiente patrón:

[Modificadores] **class** *NombreClase* [**extends** ...] [**implements** ...]

En el caso de la clase *PuntoImpl*, la cabecera es:

```
public class PuntoImpl implements Punto
```

En nuestra metodología, las clases, por ahora, llevarán el modificador **public.** Es decir, serán visibles desde cualquier parte. La cláusula **extends** será estudiada más adelant. Finalmente, la cláusula **implements** indica que la clase implementa la interfaz *Punto.* Esto quiere decir que la clase deben debe tener todos los métodos especificados en la interfaz con las mismas cabeceras y con un modificador **public.** Pero, como en este caso, la clase puede tener más métodos. En este caso el método *toString(),* que convierte el objeto en una cadena de caracteres. La clase, además, siempre tiene los constructores que no aparecen en la interfaz.

El cuerpo de la clase está formado por los atributos y los métodos. Sigue el esquema:

```
{
        [atributos]
        [métodos]
}
```

En cuanto a los **modificadores de visibilidad,** resumiendo lo que hemos visto hasta ahora en nuestra metodología tenemos que, de momento, los atributos son privados (ningún método de otra clase puede acceder a ellos); los métodos tienen visibilidad pública y pueden ser invocados desde cualquier otra clase; y, finalmente, las clases e interfaces también las declaramos con visibilidad pública. Más adelante aprenderemos más detalles sobre los modificadores y diseñaremos métodos y clases con otras visibilidades.

En una clase hay, según hemos visto arriba, tres tipos de declaraciones: de atributos, de parámetros formales y de variables locales. Cada una de ellas tiene un ámbito y unas restricciones.

Las declaraciones de atributos tienen como ámbito desde su declaración hasta el final de la clase. No puede haber dos atributos con el mismo identificador.

Las declaraciones de parámetros formales o variables locales (las que se declaran dentro de los cuerpos de los métodos) tienen como ámbito desde la declaración hasta el final del cuerpo del método. No puede haber dos declaraciones de este tipo (parámetro formal o variable local) con el mismo identificador (o nombre) en un método dado. A su vez, una declaración de parámetro formal o variable local oculta a otra posible declaración de un atributo con el mismo identificador. Ocultar quiere decir que si hay un conflicto (un atributo y una variable local o un parámetro formal con el mismo identificador), entonces nos estamos refiriendo a la variable local o al parámetro formal.

Tipo definido por una clase

Como en el caso de la declaración de una interface, cada vez que declaramos una nueva clase declaramos un nuevo tipo que tiene el mismo nombre de la clase. Con ese nuevo tipo podemos declarar nuevas entidades. A estas entidades declaradas de esta forma, como en el caso de los interfaces, las llamaremos **objetos**. Con los objetos declarados podemos formar expresiones. Las consultas a objetos son un caso particular de expresiones.
Los constructores de las clases sirven para construir nuevos objetos. El operador **new** delante de un constructor forma una expresión del tipo declarado con la clase. Una clase entonces define un tipo nuevo con el que podemos declarar objetos y construir expresiones con ellos. Este tipo definido por una clase está formado por todos los métodos públicos de la clase.

El nombre de la clase es un objeto especial que puede formar parte de expresiones. Mediante el operador punto (.) podemos combinar el nombre de una clase con los métodos públicos de la misma que estén etiquetados *static* para formar expresiones correctas. Un caso concreto es la clase *Math* proporcionada en el API de Java. Tiene un método etiquetado con *static* con nombre *sqrt(…)* que calcula la raíz cuadrada de su argumento. Por lo tanto de *Math.sqrt(a)* es una expresión bien formada que devuelve un valor de tipo *double* que es la raíz cuadrada de parámetro que se le haya pasado.

En el ejemplo siguiente cada línea representa una declaración de una variable (atributo o variable local según los caso) posiblemente con su inicialización o una expresión. Unas son correctas y otras no.

```
1. Punto p1 = new PuntoImpl()
2. PuntoImpl p2 = new PuntoImpl (2.1, 3.7);
3. Punto p3 = new PuntoImpl(-2.3, 7.8);
4. Double x = Math.sqrt(2.0);
5. Double y = Math.sqrt("2");      // incorrecta
6. p2.getZ()                       // incorrecta
```

En 1, 2, 3 declaramos puntos y los inicializamos. Las expresiones 5 y 6 son incorrectas. La 5 porque el parámetro real es de tipo *String* y el formal es de tipo *Double*. La 6 porque el tipo *Punto* no incluye el método *getZ().*

Aunque explicaremos esto con detalle más adelante, un objeto creado con uno de los constructores de una clase puede ser asignado a una variable cuyo tipo puede ser el nombre de la clase o alguno de los interfaces que implementa. Por eso la primera y tercera sentencias son correctas además de la segunda.

Clases de utilidad

En general es útil agrupar métodos reutilizables en clases para facilitar su uso. Aquellas clases que agrupan métodos reutilizables etiquetados con static las llamamos clases de utilidad. Como ejemplo diseñamos una clase de utilidad con un conjunto de métodos reutilizables para visualizar objetos en la consola. Java nos ofrece otras de este tipo como Math.

```
package test;

public class Test {
	public static void mostrar(Object p) {
		System.out.println("El objeto es: " + p);
	}
	public static void mostrar(String s, Object p) {
		System.out.println(s + p);
	}
}
```

Algunos detalles no vistos todavía:

- *System.out.println(String s):* Es un método que muestra en la pantalla el contenido de la cadena s.
- *System.out.println(s + p):* Muestra en la consola la cadena `s concatenada con p.toString()`.

4. Paquete

Un paquete es una **agrupación** de clases e interfaces que por las funciones que desempeñan conviene mantener juntos. Un paquete es similar a una carpeta que contiene diferentes ficheros. La sintaxis responde al siguiente patrón:

```
package nombre_del_paquete;
public interface … {
	…
}
```

```
package nombre_del_paquete;
public class … {
	…
}
```

Por lo tanto, para indicar que una interfaz o una clase están en un determinado paquete, su declaración debe ir precedida por **package** nombre_del_paquete. En numerosas ocasiones diferentes paquetes pueden contener clases (o interfaces) con el mismo nombre. Para distinguir unas de otras se usa el **nombre cualificado de la clase** (o interfaz). Este nombre se forma con el nombre del paquete seguido de un punto y el nombre de la clase: *nombreDePaquete.NombreDeLaClase*. Un paquete (*paquete1*) puede estar dentro de otro paquete (*paquete2*). Si dentro del *paquete1* está la clase con nombre *NombreClase* entonces su nombre

cualificado será *paquete2.paquete1.NombreClase.* Cuando en un paquete queremos usar las clases o interfaces de otro paquete se pone una clausula **import** después de la línea **package** nombre_del_paquete.

Por ejemplo, para usar el tipo Punto en una clase Test del paquete test, se pondrá una clausula import punto.Punto; detrás de la línea package test;

Si se van a usar varios tipos de un paquete en una clase se puede poner una sentencia:

```
import punto.*;
```

que importa todas las clases e interfaces del paquete punto.

5. Estructura y funcionamiento de un Programa en Java

Un programa en Java está formado por un conjunto de declaraciones de tipos enumerados, 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 éste debe haber compilado con éxito previamente. 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. Reutiliza la clase previamente diseñada *Test*.

```
package test;
import punto.*;

public class TestPunto extends Test {
	public static void main(String[ ] args) {
		Punto p= new PuntoImpl(2.0,3.0);
		mostrar("Punto:", p);
		p.setX(3.0);
		mostrar("Punto:", p);
		p.setY(2.0);
		mostrar("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)`.

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)
```

6. Convenciones Java. Reglas de estilo

Razones para mantener convenciones:

- El 80% del código de un programa necesita de un posterior **mantenimiento** y/o adaptación a nuevas necesidades.
- Casi ningún software es mantenido durante toda su vida por el autor original.
- Las convenciones de código mejoran la **lectura** del software, permitiendo entender el código nuevo mucho más rápidamente y más a fondo.
- Si distribuyes tu código fuente como un producto, necesitas asegurarte de que está bien hecho y **presentado** como cualquier otro producto.

Reglas para nombrar interfaces y clases:

- Los nombres de interfaces deben ser **sustantivos**.
- Cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas.
- Se deben mantener los nombres de interfaces **simples** y **descriptivos**.
- Usar palabras **completas**, evitar acrónimos y abreviaturas (salvo que ésta sea mucho más conocida que el nombre completo, como dni, url, html…)
- Las clases tendrán el nombre de la interfaz principal que implementan terminada en **Impl** (**Implementación**)
- Si hay más implementaciones se añadirá Impl1, Impl2… o algún sufijo explicativo, p.e. MatrizImplEnteros, MatrizImplPersona.

Reglas para nombrar métodos, variables, constantes:

- Cualquier nombre compuesto tendrá la primera letra en minúscula, y la primera letra de las siguientes palabras en mayúscula.
- Los nombres no deben empezar por los caracteres "_" o "$", aunque ambos estén permitidos por el lenguaje.
- Los nombres deben ser **cortos pero con significado**, de forma que sean un mnemónico, designado para indicar la función que realiza en el programa.
- Deben evitarse nombres de variables de un solo carácter salvo para índices temporales, como son i, j, k, m, y n para enteros; c, d, y e para caracteres.
- Las constantes se escriben con todas las letras en mayúsculas separando las palabras con un guión bajo ("_")

7. 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. 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. Los clientes (programadores que usan) de un objeto interactúan con él a través del contrato ofrecido (interfaz). El código de los métodos quedan ocultos 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 mediante la herencia de clases como se ha visto arriba.
- **Facilidad de Depuración**: Es la facilidad para encontrar los errores en un programa. Un programa bien estructurado en módulos, es decir con un buen diseño de clases, interfaces y paquetes, se dice que es fácilmente depurable y por tanto la corrección de errores será más fácil.
- **Comprensibilidad:** Es la facilidad para entender un programa. La comprensibilidad de un programa aumenta si está bien estructurado en módulos y se ha usado la encapsulación adecuadamente.

8. Problemas propuestos

1. Cree la interfaz *Punto*
2. Cree la interfaz *Circulo*
3. Cree el tipo enum *Color*

4. Implemente la clase *PuntoImpl,* con el código visto en el tema, teniendo en cuenta que debe implementar el interfaz *Punto*.
5. Implementar la clase *CirculoImpl* teniendo en cuenta que debe implementar la interfaz *Circulo* vista en el tema.
6. Especifique el tipo Persona, con las propiedades: DNI, nombre, apellidos, todas de tipo cadena. Escriba una interfaz, y la clase que la implementa.
7. Especifique el tipo Libro, con las propiedades:
    - ISBN, de tipo cadena, consultable.
    - titulo, de tipo cadena, consultable y modificable;
    - autor, de tipo Persona, consultable y modificable;
    - número de Páginas, de tipo entero, consultable y modificable;
    - precio, de tipo real, consultable y modificable;
    - best-seller, de tipo booleano, consultable y modificable
8. Especifique el tipo Televisor, con las siguientes propiedades:
    - Programa, de tipo entero, consultable y modificable.
    - Volumen, de tipo entero, consultable.

    Otras operaciones:

    - incrementaPrograma, suma uno al programa actual.
    - decrementaPrograma, resta uno al programa actual.
    - incrementaVolumen, suma uno al volumen actual.
    - decrementaVolumen, resta uno al volumen actual.