Introducción a la Programación

Tema 1. Introducción a la Programación Orientada a Objetos

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.

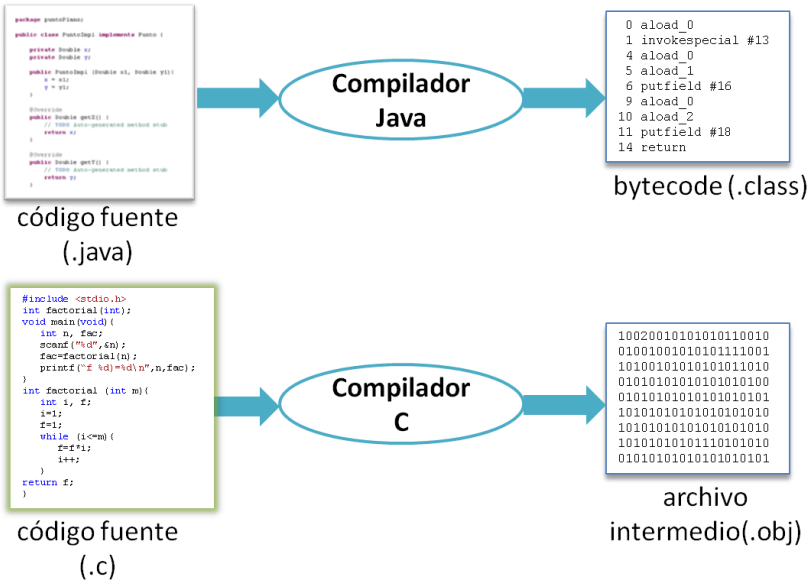


Figura 1. Proceso de compilación en Java y C

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.

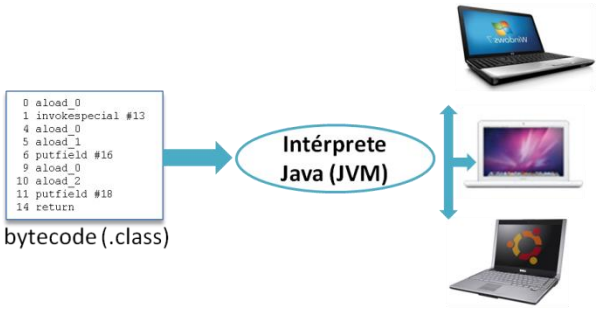


Figura 2. Proceso de interpretación en Java

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 capítulo vamos a dar una vista panorámica a la programación orientada a objeto y en particular a la programación usando el lenguaje Java. Muchos de los conceptos explicados en este capítulo los volveremos a ver más detallados más adelante

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. Un **tipo** es un conjunto de valores que una entidad puede tomar y las operaciones que podemos hacer con esos valores. Mediante **declaración** decimos que se va a usar una entidad que tiene como nombre el identificador y es del tipo indicado. Cada entidad declarada de un tipo tendrá, en un momento dado, un valor de ese tipo.

Asumimos que tenemos disponibles un conjunto de tipos que serán proporcionados por el lenguaje de programación que usemos. Entre ellos podemos destacar:

- **int, Integer, long, Long**: Valores de tipo entero y sus operaciones. Ejemplo: 3
- **float, Float, double, Double**: Valores de tipo real y sus operaciones: Ejemplo 4.5
- **boolean, Boolean**: Valores de tipo lógico y sus operaciones: Sus valores son: **true, false**
- **String**: Secuencias de caracteres y sus operaciones. Ejemplo: “Hola”.
- **char, Character**: Caracteres 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 representada por el operador **+.**

Junto a los anteriores se pueden definir tipos nuevos mediante la cláusula **enum.** Por ejemplo el tipo *Color* podemos definirlo como:

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

Los tipos definidos de esta manera tienen los valores declarados. Los valores son: Color.ROJO, Color.NARANJA, etc. Estos 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.

Ejemplo de declaración de variables.

```
Integer edad;
Double peso;
```

```
String s1, s2;
Color c;
```

Aquí declaramos una nueva entidad *edad* es de tipo *Integer* y *peso* de tipo *Double.* Las entidades *s1 y s2* están declaradas como *String* y *c* como *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 un ámbito está delimitado, en Java, por los símbolos {…}.

Las entidades que pueden cambiar el valor que almacenan las llamaremos **variables** y las que no **constantes**.

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

Ejemplo:

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

La primera es de tipo *boolean*. La segunda de tipo *Double.* La tercera de tipo *Color.*

Mediante una **asignación** (representada por **=**) podemos dar nuevos valores a una variable.

Por ejemplo:

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

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). **La expresión:** *precio = 4.5*2 + 34;* tiene como valor *43.* y como tipo *Double*.

La 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 pueden ser de dos tipos: **expresiones sin efectos laterales** y **con efectos laterales**. Una expresión se dice que tiene efectos laterales si alguna variable de la expresión cambia de valor al ser evaluada. Si ninguna variable cambia de valor al evaluarse la expresión decimos que es una expresión sin efectos laterales. Cuando una expresión contiene el operador de asignación tiene efectos laterales. La variable a la izquierda del operador de asignación cambia de valor. Más adelante veremos otros ejemplos de expresiones con y sin efectos laterales.

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**. Las expresiones bien formadas pueden mostrar errores cuando intentamos ejecutarlas. Son los 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 acabadas en punto y coma (;). Si las expresiones que forman la secuencia son con efectos laterales entonces el orden en que las escribamos es muy importante. El orden no importa si las expresiones que forman la secuencia son sin efectos laterales.

En una secuencia de expresiones hay **usos y definiciones de variables.** Si en una expresión con efectos laterales se cambia el valor de una variable decimos que hay una definición de la variable en esa expresión. Las variables que no cambian de valor decimos que han sido usadas en esa expresión. Cuando una variable es usada aporta en la expresión el valor de la última vez que se definió. Una variable puede ser usada y definida en una misma expresión. Usar una variable se denomina también **leer la variable**.

```
int a,b,c;
a = 3*5+24;
b = a+3;
a = a +29;
a = a*a+45;
```

En el ejemplo anterior después de la declaración la variable *a* es definida tres veces y usada cuatro, la variable *b* es definida pero no usada, la variable *c* no es usada ni definida. Cuando una variable no ha sido definida no tiene ningún valor establecido. Una inicialización es una

primera definición de una variable. Si sólo declaramos una variable (sin inicializarla) ésta no tiene ningún valor.

Una variable que nunca se use dentro del ámbito donde está declarada (nunca haya sido leída) es una **variable inútil** y así será indicado por el entorno que usemos. Una variable inútil puede ser eliminada del código. Nunca debemos usar una variable antes de haya sido definida. Es decir antes de que le hayamos dado un valor en una definición previa.

En una secuencia de expresiones puede haber **código inútil**. Una expresión de asignación, dentro de una secuencia, es inútil cuando el valor asignado a la variable a la izquierda de la asignación no es usado posteriormente. Una expresión de asignación inútil puede ser eliminada del código y al hacerlo puede que otras expresiones de asignación se hagan inútiles. Veamos el siguiente ejemplo:

```
1. int a,b,c;
2. a = 3*5+24;
3. b = a+3;
4. a = a+b;
5. a = a +29;
6. a = 3;
7. a = a*a+45;
8. // usos de las variables a y b
```

La expresión 6 hace inútil la 5 y al eliminar esta se hace inútil la 4. La expresión 2 es útil porque el valor se usa en la 3. Eliminando el código inútil y la variable inútil la secuencia anterior queda:

```
1. int a,b;
2. a = 3*5+24;
3. b = a+3;
4. a = 3;
5. a = a*a+45;
6. // usos de las variables a y b
```

3. Conceptos básicos de Programación Orientada a Objetos

Lenguajes Orientados a Objetos (LOO) son los que están basados en la modelización del mundo real o incluso virtual mediante objetos. Estos objetos tendrán la **información** y las **capacidades** necesarias para resolver los problemas en que participen. Ejemplo: Una persona tiene un nombre, una fecha de nacimiento, unos estudios, etc.

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: ¿cómo se encuentra el objeto? Es decir 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.

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)

Es un elemento de la POO que permite, a priori, establecer **cuáles** son sus propiedades, ***qué*** *se puede hacer con un objeto de un determinado tipo*, pero no se preocupa de saber **cómo** se hace. Un ejemplo puede ser una persona, sus propiedades podrían ser: nombre, apellidos, DNI, dirección, etc.

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 como 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 hacer el modelo de un objeto es conveniente seguir una metodología para 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. Indicar, en su caso, si la propiedad es compartida por todos los objetos del tipo.
5. 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.
6. 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.
7. 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.

Ejemplo:

Queremos modelar un objeto **p** que llamaremos punto del plano. Para ello definiremos una interfaz que representa un **contrato** con los posibles usuarios del objeto. Aunque para especificar completamente un contrato necesitaremos más elementos, además de una interfaz, por ahora hablaremos indistintamente de contrato o interface.

Punto:

- Propiedades:

- X: Double, consultable y modificable
- Y: Double, consultable y modificable

En el lenguaje Java un interface se declara como en el ejemplo siguiente para el caso del punto:

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

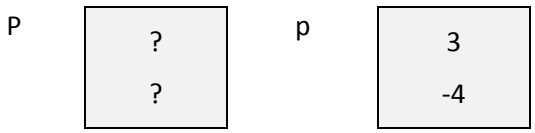
Para indicar que p representa un punto los indicaremos en Java por:

```
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.

```
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.



Si queremos consultar las propiedades de p:

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

Las dos anteriores son expresiones bien formadas. Ambas son de tipo *Double* y sus valores son los valores de las propiedades *X* y *Y*, respectivamente, guardadas en el estado el objeto en un momento dado. Otra forma de decirlo es: al invocar 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 que este el tipo de la variable *p*. La tercera tampoco está bien formada. En este caso setX si es un método de Punto pero el valor proporcionado no es de tipo *Double*.
Cuando definimos una interfaz (interface) 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. 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 el cuadrado de la distancia del punto al origen de coordenadas. 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 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 forma 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
    - Origen, Punto, Consultable, compartida

- DistanciaAlOrigen, Double, consultable, derivada
- DistanciaAOtroPunto(Punto p), Double, consultable, derivada

La interfaz asociada es:

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

Diseñemos ahora el tipo *Círculo.*

Circulo:

- Propiedades:
  - Centro: Punto, consultable y modificable
  - Radio: Double, consultable y modificable
  - Area: Double, consultable, derivada de Radio
  - Longitud: Double, consultable, derivada de Radio
- Operaciones
  - MoverElCentroAlOrigen, mueve el centro del círculo al origen de coordenadas

La interfaz asociada es:

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

La sintaxis para declarar una interfaz responde al siguiente patrón:

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

La cláusula **extends** la estudiaremos más adelante.

Como hemos dicho antes, llamamos **parámetros formales** a los parámetros que aparecen en el método en el momento de declararlo. 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 una interfaz

Como hemos visto arriba al declarar una interfaz estamos definiendo un tipo nuevo. Este tipo tiene el mismo identificador que la interfaz y está formado por todos los métodos declarados en la interfaz. 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 contenido en el estado del centro del círculo.

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

Como vemos cuando definimos una 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 ir detallando la clase nos guiamos por la interfaz que implementa.

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 la clase tendrá un atributo por cada propiedad no derivada de la interfaz que implementa y 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.

Ejemplo para definir los atributos de una clase que implemente la interfaz *Punto:*

```
private Double x = 0;
private Double y = 0;
private static Punto origen;
```

y para una que implemente el interface Circulo:

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

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.

Las propiedades compartidas son aquellas, que como indica su nombre, se comparten por todos los objetos de un mismo tipo. Aunque más adelante lo veremos con más detalle por ahora sólo indicaremos que los atributos que guardan valores de propiedades compartidas deben llevar el **modificador static** además del tipo y la visibilidad.

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]*;*

Donde **private** es un modificador que restringe la visibilidad de un atributo sólo al interior de la clase y **static** otro modificador que indica que el atributo y su correspondiente valor serán compartidos por todos los objetos del tipo. Más adelante aprenderemos otros modificadores posibles. 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. 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 cuyo ámbito es el cuerpo del método. El cuerpo está formado por declaraciones, expresiones y otro código necesario para indicar de forma concreta que 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. En una interfaz sólo aparecen las cabeceras de los métodos. En una clase deben aparecer la cabecera y el cuerpo. Puede haber métodos con el mismo nombre pero diferente cabecera. Pero como 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.

La sintaxis para los métodos corresponde al siguiente patrón:

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

Hay dos tipos de métodos: **observadores** y **modificadores**. Los métodos observadores no modifican el estado del objeto. Es decir no modifican los atributos. O lo que es lo mismo los atributos no pueden aparecer, dentro de su cuerpo, en la parte izquierda de una asignación. Normalmente serán métodos observadores aquellos cuyo nombre empiece por *get*. Los métodos modificadores modifican el estado del objeto. Los atributos aparecen, dentro de su cuerpo, en la parte izquierda de una asignación. Normalmente todos los métodos que empiecen por *set* son modificadores.

Ejemplo de un método modificador *(setX*) y otro consultor (*getX*).

```
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) {
            x = x1;
            y = y1;
      }
```

```
        public PuntoImpl(){
                x = 0.;
                y = 0.;
        }
```

Los constructores deben inicializar de manera general cada uno de los atributos salvo los que tienen el modificador static. Esto últimos se inicializan cuando se declaran. Puede haber otras posibilidades que aprenderemos más adelante. 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. La sintaxis responde al siguiente patrón:

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

Ejemplo de clase que implementa Punto. Como convención haremos que el nombre de la clase que implementa una interface será el mismo seguido de Impl.

```
public class PuntoImpl implements Punto {
        private Double x;
        private Double y;
        private static Punto origen = new PuntoImpl();
        public PuntoImpl (Double x1, Double y1) {
                this.x=x1;
                this.y=y1;
        }
        public PuntoImpl(){
                this.x=0.;
                this.y=0.;
        }
        public static Punto create(){
                return new PuntoImpl();
        }
        public static Punto create(Double x1, Double y1){
                return new PuntoImpl(x1,y1);
        }
        public Double getX() { return x; }
        public Double getY() { return y; }
        public void setX(Double x1)  { x = x1; }
        public void setY(Double y1)  { y = y1; }

        public Punto getOrigen(){ return origen;}

        public Double getDistanciaAlOrigen(){
                return this.getDistanciaAOtroPunto(origen){
        }
```

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

	public void aumentarCoordenadas(Double p){
		this.x = this.x*(1+p);
		this.y = this.y*(1+p);
	}
	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 diseñando y sin embargo *p.getX()* es la invocación del método *getX* sobre el objeto *p*. Dentro del cuerpo de una clase en la invocación de un método (o atributo) de la misma sobre el objeto *this* (el que estamos diseñando) podemos quitar, si no hay ambigüedad, la palabra *this*. Por lo tanto, si no hay ambigüedad, las expresiones que están en la misma línea son equivalentes dentro de una clase dada:

- *this.get(x), getX()*
- *this.distanciaAOtroPunto(origen), distanciaAOtroPunto(origen)*

Con la palabra *this* y el operador punto podemos formar expresiones correctas dentro de una clase. Tras el *this* y el punto (.) puede ir cualquier atributo o método de la clase que no sea *static.*

Como los métodos las clases tienen dos partes cabecera y cuerpo. La cabecera sigue el siguiente patrón:

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

Que en este caso es:

```
public class PuntoImpl implements Punto
```

Indica que *PuntoImpl* implementa el contrato de la interfaz *Punto* (por lo que, al menos, hay que programar todos los métodos definidos en la interfaz *Punto*). La clausula **extends** será estudiada más adelante.

Las clases, por ahora, llevarán el modificador de **public.** Es decir serán visibles desde cualquier parte. 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 misma cabecera y con un modificador **public.** Pero, como en este caso, la clase puede tener más métodos. En este caso el método *toString()*. La clase, además, siempre tiene los constructores que no aparecen en el interface.

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

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

Si una clase implementa una interfaz (*interface*) debe implementar (añadir el cuerpo al método) cada uno de los métodos de esa interfaz y posiblemente algunos más. En este caso la clase implementa también el método *toString*, *moverAOtroPunto*, y dos métodos con el nombre *create* modificador *static*. El primero convierte el objeto en una cadena de caracteres y el segundo cambia la posición del punto para colocarlo en la posición dada por otro punto que se proporciona como parámetro. Los métodos create crean objetos llamando a los respectivos constructores.

Se observa cómo los **modificadores de visibilidad** de los atributos son privados (ningún método de otra clase puede acceder a ellos), mientras que los métodos tienen visibilidad pública y pueden ser invocados desde cualquier otra clase. 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 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 conficto (una 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 fundamentalmente para construir nuevos objetos. El operador **new** delante de un constructor forma una expresión del tipo declarado con la clase. Pero una clase también define un tipo nuevo con el que podemos declara variables y construir expresiones con ellas. Este tipo definido por una clase está formado por todos los métodos públicos de la clase. Siguiendo las recomendaciones de los tutoriales de Java aquí y en lo que sigue restringiremos el tipo definido por una clase al conjunto de métodos públicos de la misma y que no tengan el modificador *static*. Más adelante veremos esto con más detalle. Como podemos comprobar el tipo *PuntoImpl* incluye un método *aumentarCoordenadas(Double)* que no está incluido en el tipo *Punto*.

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.

Vemos, por lo tanto, que cuando diseñamos una clase nueva definimos un tipo nuevo con el mismo nombre de la clase. Con variables de ese tipo y el operador punto (.) podemos formar expresiones. Estas expresiones estarán bien formadas si tras el operador punto usamos cualquier método público de la clase aunque no se recomienda usar aquellos métodos públicos etiquetados con *static*. Con los métodos públicos etiquetados *static* podemos construir expresiones. Estas expresiones se forman con el nombre la de la clase, el operador punto (.) y un método público de la clase etiquetado con *static*.

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.  Punto p4 = PuntoImpl.create();
5.  Punto p5 = p3.create();                  // incorrecta
6.  p2.aumentarCoordenadas(0.1);
7.  p3.aumentarCoordenadas(0.1);             // incorrecta
8.  private static Punto origen = new PuntoImpl();
9.  Double x = Math.sqrt(2.0);
10. Double y = Math.sqrt("2");               // incorrecta
11. p1.getDistanciaAlOrigen();
12. p2.getZ();                               // incorrecta
```

En 1, 2, 3, 4 declaramos puntos y los inicializamos. Veremos más adelante porque las asignaciones 1, 3, y 4 son expresiones válidas. Las expresiones 5, 7, 10 y 12 son incorrectas. La 5 porque el método *create()* no pertenece al tipo *Punto*. La 7 porque el método *aumentarCoordenadas(Double)* no pertenece al tipo *Punto* aunque sí al tipo *PuntoImpl* por lo que la 6 si es correcta. La 10 porque el parámetro real es de tipo *String* y el formal es de tipo *Double*. La 12 porque el tipo *Punto* no incluye el método *getZ().*

La expresión 4 es válida y la variable *p4* se inicializa al punto *(0.,0.).* La expresión 8 declara un atributo (debe ser realizada dentro del cuerpo de una clase) privado con el modificador *static*, de tipo *Punto* e inicializado al punto *(0.,0.).* La expresión 11 calcula un valor que no es usado.

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 sentencia son correctas además de la segunda.

En la línea siguiente declaramos la variable origen (en este caso la variable es un atributo tal como hemos visto antes) de tipo Punto y la inicializamos construyendo un punto nuevo con el constructor que no toma parámetros. Al ser un atributo lleva algunos modificadores como *private* y *static*.

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(String p) {
		System.out.println(p);
	}
	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;
interface …{
        …
}
```

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

Por lo tanto para indicar que una interface 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 interface). 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 clase con nombre *NombreClase* entonces su nombre cualificado será *paquete2.paquete1.NombreClase*

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.

En general es deseable detectar todos los errores posibles en tiempo de compilación. Para ello el sistema de tipos de un lenguaje dado (Java en particular), es decir las diferentes posibilidades de declarar tipos, es una herramienta clave.

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;
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)
```

Igualmente podemos diseñar la clase *TestPuntoPixel* que también reutiliza la clase Test y nos va a servir para evaluar si el tipo *Pixel* funciona como esperamos.

```
public class TestPuntoPixel extends Test {

  public static void main(String[ ] args) {
       Punto p = new PuntoImpl(2.0,3.0);
       mostrar(“Punto: ”, p);
       mostrar(“Distancia al origen: ”,p.getDistanciaAlOrigen());
       p.setX(3.);
       p.setY(2.);
       mostrar("Punto: ”, p);
       mostrar("Distancia al origen: “, p.getDistanciaAlOrigen());
       Pixel p2 = new PixelImpl(1.0,1.0,Color.ROJO);
       mostrar("Distancia al (1,1): ”, p.getDistancia(p2));
       mostrar(“Pixel: “, p2);  }
}
```

Los resultados esperados en la consola son:

```
Punto: (2.0,3.0)
Distancia al origen: 3.605551
Punto: (3.0,2.0)
Distancia al origen: 3.605551
Distancia al (1,1): 2.236067
Pixel: (1.0,1.0),ROJO
```

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 de un objeto interactúan con él a través del contrato ofrecido (interfaz). El estado y 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. Conceptos aprendidos en el tema

- Tipo, expresión, expresión bien formada
- Variable, declaración, inicialización, ámbito de una declaración
- Expresión con efectos laterales y sin efectos laterales.
- Secuencias de expresiones: uso y definición de una variable, código inútil, variable inútil.
- Objeto, estado, propiedades, operaciones
- Interfaz (interface).
- Método, signatura, sobrecarga de métodos, parámetros formales y parámetros reales
- Clase, atributos, constructores, métodos
- Tipos definidos por clases, interfaces y *enum*. Relaciones entre los tipos: herencia, implementación, uso, subtipo.
- Paquete.
- Modos de un programa: Modo de compilación, Modo de Ejecución.

9. Problemas propuestos

1. Cree el paquete ejemplos.
2. Cree el interface *Punto*
3. Cree el interface *Circulo*
4. Cree el tipo enum *Color*
5. Cree el interface *Pixel* teniendo en cuenta que debe heredar de Punto.
6. Cree el interface *Vector2D* teniendo en cuenta que debe heredar de Punto
7. Implemente la clase *PuntoImpl,* con el código visto en el capítulo, teniendo en cuenta que debe implementar la interfaz *Punto*.
8. Implemente la clase *PixelImpl* teniendo en cuenta que debe implementar la interfaz Pixel y heredar de *PuntoImpl*.
9. Implemente la clase *Test* vista anteriormente.
10. Implemente la clase *TestPuntoPixel* que herede de *Test* y nos sirva para comprobar el funcionamiento adecuado de las clases Punto y Pixel.

11. Implementar la clase *CirculoImpl* teniendo en cuenta que debe implentar el interface *Circulo* visto en el capítulo.
12. Implemente una clase *TestCirculo* que herede de la clase *Test* y nos sirva para comprobar el funcionamiento de la clase implementada
13. Implemente la clase *Vector2DImpl* teniendo que cuenta que debe implentar la interfaz *Vector2D* y heredar de la clase *PuntoImpl*.
14. Implemente la clase *TestVector2D* que herede de la clase *Test* y nos sirva para comprobar el adecuado funcionamiento de la clase *Vector2DImpl.*