

TEMA 5: DESARROLLO DE CLASES

1.	CONCEPTO DE CLASE	3
1.1	REPASO DEL CONCEPTO DE OBJETO	3
1.2	EL CONCEPTO DE CLASE	5
2.	EMPAQUETADO DE CLASES.....	6
2.1	JERARQUÍA DE PAQUETES	6
2.2	UTILIZACIÓN DE LOS PAQUETES.....	7
2.3	INCLUSIÓN DE UNA CLASE EN UN PAQUETE.....	8
2.4	PROCESO DE CREACIÓN DE UN PAQUETE.....	9
3.	ESTRUCTURA Y MIEMBROS DE UNA CLASE.....	10
3.1	DECLARACIÓN DE UNA CLASE	10
3.2	CABECERA DE UNA CLASE	12
3.3	CUERPO DE UNA CLASE.....	13
4.	ATRIBUTOS	14
4.1	DECLARACIÓN DE ATRIBUTOS.....	15
4.2	MODIFICADORES DE ATRIBUTO	19
4.3	MODIFICADORES DE ACCESO	19
4.4	MODIFICADORES DE CONTENIDO: ATRIBUTOS CONSTANTES (I).....	21
4.5	MODIFICADORES DE CONTENIDO: ATRIBUTOS CONSTANTES (II).....	23
4.6	COMBINANDO MODIFICADORES	27
4.7	OBJETOS INMUTABLES.....	34
5.	MÉTODOS	36
5.1	DECLARACIÓN DE UN MÉTODO	37
5.2	CABECERA DE UN MÉTODO.....	38
5.3	MODIFICADORES EN LA DECLARACIÓN DE UN MÉTODO	39
5.4	PARÁMETROS DE UN MÉTODO	41
5.4.1.	LISTA DE PARÁMETROS VARIABLES.....	43
5.4.2.	MODIFICADOR FINAL EN LOS PARÁMETROS.....	45
5.5	CUERPO DE UN MÉTODO	46
5.5.1.	VALOR DEVUELTO POR UN MÉTODO: RETURN.....	47
5.5.2.	VARIABLES LOCALES	48
5.5.3.	EL OPERADOR DE AUTORREFERENCIA THIS	49

TEMA 5: DESARROLLO DE CLASES

5.5.4. Lanzando excepciones desde un método	54
5.6 ENCAPSULACIÓN, CONTROL DE ACCESO Y VISIBILIDAD.....	58
5.6.1. MÉTODOS DE ACCESO(I): GETTERS	59
5.6.2. MÉTODOS DE ACCESO (II): SETTERS	63
5.6.3. MÉTODOS PRIVADOS.....	65
5.7 SOBRECARGA DE MÉTODOS.....	67
5.7.1. SOBRECARGA DE OPERADORES.....	71
5.8 MÉTODOS ESTÁTICOS.....	72
5.8.1. MÉTODO MAIN EN JAVA.	74
5.9 MÉTODO TOSTRING EN JAVA.....	76
6. CONSTRUCTORES.....	78
6.1 CONCEPTO DE CONSTRUCTOR.	78
6.2 IMPLEMENTACIÓN DE CONSTRUCTORES.....	79
6.3 LANZANDO EXCEPCIONES DESDE LOS CONSTRUCTORES.	81
6.4 SOBRECARGA DE CONSTRUCTORES.	84
6.4.1. USO DE LA LLAMADA THIS() EN LOS CONSTRUCTORES.	85
6.5 CONSTRUCTORES DE COPIA.	88
6.6 DESTRUCCIÓN DE OBJETOS.....	90
6.7 MÉTODOS “FABRICA” O PSEUDOCONSTRUCTORES.....	91
6.8 BLOQUES DE INICIALIZACIÓN EN JAVA.....	94
7. DOCUMENTACIÓN DE UNA CLASE.	96
7.1 ETIQUETAS Y POSICIÓN.	97
7.2 USO DE LAS ETIQUETAS.....	97
7.3 ORDEN DE LAS ETIQUETAS.	98
7.4 EJEMPLO PRÁCTICO.....	99
8. CREACIÓN Y UTILIZACIÓN DE OBJETOS.....	105
8.1 DECLARACIÓN DE UN OBJETO.....	105
8.2 CREACIÓN DE UN OBJETO.	106
8.3 REFERENCIAS A UN OBJETO.	109
8.4 MANIPULACIÓN DE UN OBJETO: UTILIZACIÓN DE MÉTODOS Y ATRIBUTOS.	111
8.5 RECOGIENDO LAS EXCEPCIONES LANZADAS POR UN MÉTODO.	112

1. CONCEPTO DE CLASE.

Nada más comenzar la unidad se nos propone un reto muy interesante: intentar desarrollar un modelo simple de un vehículo terrestre, por ejemplo, un automóvil. A lo largo de toda la unidad vamos a utilizar este proyecto como eje central para estudiar cada uno de los aspectos importantes a la hora de desarrollar una clase en Java y, por extensión, en cualquier lenguaje de programación orientado a objetos.

Como ya has visto en anteriores unidades, las **clases** están compuestas por atributos y métodos especificando de esa manera las características comunes de un conjunto de **objetos**. De esta forma, los programas que escribas estarán formados por un conjunto de **clases**, a partir de las cuales irás creando **objetos** que se interrelacionarán unos con otros. Nuestro primer objetivo será entonces implementar una clase **Vehículo**.

Recomendación

En esta unidad se va a utilizar el concepto de **objeto**, así como algunas de las diversas **estructuras de control** básicas que ofrece cualquier lenguaje de programación. Todos esos conceptos han sido explicados y utilizados en las unidades anteriores. Si consideras que es necesario hacer un repaso del concepto de objeto o del uso de las estructuras de control elementales, éste es el momento de hacerlo.

1.1 REPASO DEL CONCEPTO DE OBJETO.

Desde el comienzo del módulo llevas utilizando el concepto de **objeto** para desarrollar tus programas de ejemplo. En las unidades anteriores se ha descrito un objeto como una entidad que contiene **información** (atributos) y que es capaz de llevar a cabo ciertas **operaciones** (métodos) con esa información más algunos estímulos externos (parámetros de entrada de los métodos). Según la información que contengan esos atributos el objeto tendrá un **estado** determinado y según las operaciones que se puedan llevar a cabo con esos datos será responsable de un **comportamiento** concreto.

Recuerda que entre las características fundamentales de un objeto se encontraban la *identidad* (los objetos son únicos y por tanto distinguibles entre sí, aunque pueda haber objetos exactamente iguales), un *estado* (los atributos que describen al objeto y que tendrán cierto valor en cada momento) y un determinado *comportamiento* (acciones que se pueden realizar sobre el objeto).

Algunos ejemplos de objetos que podríamos imaginar podrían ser:

- Un vehículo de tipo turismo, de color rojo, marca SEAT, modelo Toledo, del año 2020. En este ejemplo tenemos una serie de atributos, como el color (en este caso rojo), la marca, el modelo, el año, etc. Así mismo también podríamos imaginar determinadas características como la cantidad de combustible que le queda, o el número de kilómetros recorridos hasta el momento.
- Otro vehículo de color amarillo, marca Opel, modelo Astra, del año 2018.
- Otro vehículo más de color amarillo, marca Opel, modelo Astra y también del año 2018. Se trataría de otro objeto (otro vehículo) con las mismas propiedades que el anterior (es idéntico, pero no es el mismo). Sería otro objeto diferente con características idénticas, casi como un clon.

TEMA 5: DESARROLLO DE CLASES

- Un cocodrilo de cuatro metros de longitud y de veinte años de edad.
- Un círculo de radio 2 centímetros, con centro en las coordenadas (0,0) y relleno de color amarillo.
- Otro círculo de radio 3 centímetros, con centro en las coordenadas (1,2) y relleno de color verde.
- Una persona con nombre "Juan", apellidos "Torres Waxman" y fecha de nacimiento 01/01/1990.
- Otra persona con nombre "Ana", apellidos "Castillo Gil" y fecha de nacimiento 04/07/1992.

Si observas los ejemplos anteriores podrás distinguir sin demasiada dificultad al menos cuatro familias de objetos diferentes, que no tienen nada que ver una con otra: Los vehículos, los círculos, los cocodrilos, las personas.

Es de suponer entonces que cada objeto tendrá determinadas posibilidades de **comportamiento (acciones)** dependiendo de la familia a la que pertenezcan. Por ejemplo, en el caso de los **vehículos** podríamos imaginar acciones como: **arrancar, apagar, repostar, frenar, acelerar, cambiar de marcha**, etc. En el caso de los **cocodrilos** podrías imaginar otras acciones como: **desplazarse, comer, dormir, cazar**, etc. Para el caso del **círculo** se podrían plantear acciones como: **cálculo de la superficie del círculo, cálculo de la longitud de la circunferencia que lo rodea**, etc.

Por otro lado, también podrías imaginar algunos **atributos** cuyos valores podrían ir cambiando en función de las acciones que se realizaran sobre el objeto: ubicación del vehículo (coordenadas), velocidad instantánea, kilómetros recorridos, velocidad media, cantidad de combustible en el depósito, etc. En el caso de los cocodrilos podrías imaginar otros atributos como: peso actual, el número de dientes actuales (irá perdiendo algunos a lo largo de su vida), el número de presas que ha cazado hasta el momento, etc.

Como puedes ver, un **objeto** puede ser cualquier cosa que puedas describir en términos de **atributos y acciones**. Dependerá de la aplicación que tengas que desarrollar para que tu objeto contenga unos u otros atributos y/o acciones. El límite estará en las necesidades de la aplicación y tu imaginación a la hora de "inventar" objetos.

Un objeto no es más que la representación de cualquier entidad concreta o abstracta que puedes percibir o imaginar y que pueda resultar de utilidad para modelar los elementos del entorno del problema que deseas resolver.

Reflexiona

Si has trabajado antes con el **modelo Entidad/Relación** para representar información a la hora de modelar un sistema de información o una base de datos, es posible que hayas reparado en la similitud que existe entre el concepto de **objetos** y el de **instancias o casos individuales de una entidad**.

1.2 EL CONCEPTO DE CLASE.

Está claro que dentro de un mismo programa tendrás la oportunidad de encontrar decenas, cientos o incluso miles de objetos. En algunos casos no se parecerán en nada unos a otros, pero también podrás observar que habrá muchos que tengan un gran parecido, compartiendo un mismo comportamiento y unos mismos atributos. Habrá muchos objetos que sólo se diferenciarán por los valores que toman algunos de esos atributos.

Es aquí donde entra en escena el concepto de **clase**. Está claro que no podemos definir la estructura y el comportamiento de cada objeto cada vez que va a ser utilizado dentro de un programa, pues la escritura del código sería una tarea interminable y redundante. La idea es poder disponer de una **plantilla** o **modelo** para cada conjunto de objetos que sean del mismo tipo, es decir, que tengan los mismos atributos y un comportamiento similar.

Una clase consiste en la definición de un tipo de objeto. Se trata de una descripción detallada de cómo van a ser los objetos que pertenezcan a esa clase, indicando qué tipo de información contendrán (atributos) y cómo se podrá interactuar con ellos (comportamiento).

Como ya has visto anteriormente, una clase consiste en una plantilla en la que se especifican:

- Los **atributos** que van a ser comunes a todos los objetos que pertenezcan a esa clase (información).
- Los **métodos** que permiten interactuar con esos objetos (comportamiento).

A partir de este momento podrás hablar ya sin confusión de objetos y de clases, sabiendo que los primeros son instancias concretas de las segundas, que no son más que una abstracción o definición.

Si nos volvemos a fijar en los ejemplos de objetos del apartado anterior podríamos observar que las clases serían lo que clasificamos como "familias" de objetos (**Vehiculo**, **Cocodrilo**, **Circulo**, **Persona**). Por ejemplo, podríamos hablar de la clase **Vehiculo** y de varios objetos de esa clase: el Seat Toledo rojo de 2020, el primer Opel Astra amarillo de 2018 y el segundo Opel Astra amarillo de 2018.

En el lenguaje cotidiano de muchos programadores puede ser habitual la confusión entre los términos clase y objeto. Aunque normalmente el contexto nos permite distinguir si nos estamos refiriendo realmente a una clase (definición abstracta) o a un objeto (instancia concreta), hay que tener cuidado con su uso para no dar lugar a interpretaciones erróneas, especialmente durante el proceso de aprendizaje.

Reflexiona

Nuevamente, si has trabajado antes con el **modelo Entidad/Relación**, también es muy probable que te hayas fijado en el paralelismo que existe entre el concepto de **clase** y el de **entidad**.

Para saber más

Aquí tienes un par de vídeos introductorios a la **programación orientada a objetos** usando el lenguaje Java:
<https://www.youtube.com/watch?v=XmUz5WJmJVU&list=PLU8oAIHdN5BktAXdEVCLUYzvDyqRQJ2Ik>

<https://www.youtube.com/watch?v=ZY5pwm92cWQ&list=PLU8oAIHdN5BktAXdEVCLUYzvDyqRQJ2Ik>

2. EMPAQUETADO DE CLASES.

La **encapsulación** de la información dentro de las clases permite llevar a cabo el proceso de ocultación, que es fundamental para el trabajo con clases y objetos. Es posible que conforme vaya aumentando la complejidad de tus aplicaciones necesites que algunas de tus clases puedan tener acceso a parte de la implementación de otras debido a las relaciones que se establezcan entre ellas a la hora de diseñar tu modelo de datos. En estos casos se puede hablar de un nivel superior de encapsulamiento y ocultación conocido como **empaquetado**.

Un **paquete** consiste en un conjunto de clases relacionadas entre sí y agrupadas bajo un mismo nombre. Normalmente se encuentran en un mismo paquete todas aquellas clases que forman una biblioteca o que reúnen algún tipo de característica en común. Esto facilita la organización de las clases para luego poder localizar fácilmente aquellas que vayas necesitando.

2.1 JERARQUÍA DE PAQUETES

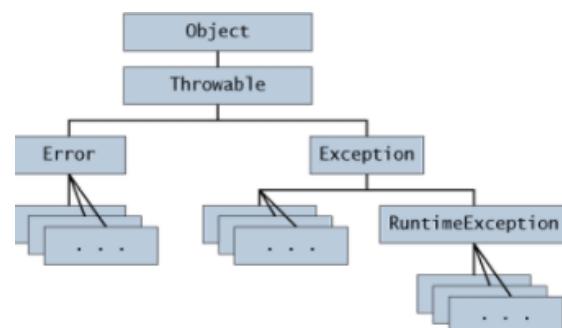
Los **paquetes en Java** pueden organizarse **jerárquicamente** de manera similar a lo que puedes encontrar en la estructura de carpetas en un dispositivo de almacenamiento, donde:

- las clases serían como los archivos,
- cada paquete sería como una carpeta que contiene archivos (clases),,
- cada paquete puede además contener otros paquetes (como las carpetas que pueden contener otras carpetas),
- para poder hacer referencia a una clase dentro de una estructura de paquetes, habrá que indicar la **trayectoria completa** desde el paquete raíz de la jerarquía hasta el paquete en el que se encuentra la clase, indicando por último el nombre de la clase (como el path absoluto de un archivo).

La estructura de paquetes en Java permite organizar y clasificar las clases, evitando conflictos de nombres y facilitando la ubicación de una clase dentro de una estructura jerárquica.

Por otro lado, la organización en paquetes permite también el **control de acceso** a miembros de las clases desde otras clases que estén en el mismo paquete gracias a los **modificadores de acceso**. Los modificadores de acceso se van a ver en detalle dentro de un par de apartados en esta misma unidad, aunque ya han aparecido en algunos ejemplos anteriores. ¿Recuerdas el modificador **public**?

Las clases que forman parte de la jerarquía de clases de Java se encuentran organizadas en diversos paquetes.



Todas las clases proporcionadas por Java en sus bibliotecas son miembros de distintos paquetes y se encuentran organizadas jerárquicamente. Dentro de cada paquete habrá un conjunto de clases con algún tipo de relación entre ellas. Se dice que todo ese conjunto de paquetes forman la API Java. Por ejemplo, las clases básicas del lenguaje se encuentran en el paquete **java.lang**, las clases de entrada/salida las podrás encontrar en el paquete **java.io** y en el paquete **java.math** podrás observar algunas clases para trabajar con números grandes y de gran precisión.

Para saber más

Puedes echar un vistazo a la jerarquía de paquetes de la API básica de Java en la documentación oficial de Oracle: [Java API Documentation. Java SE. Module java.base.](#)

2.2 UTILIZACIÓN DE LOS PAQUETES.

Es posible acceder a cualquier clase de cualquier paquete (siempre que ese paquete esté disponible en nuestro sistema, obviamente) mediante la **calificación completa** de la clase dentro de la estructura jerárquica de paquete. Es decir, indicando la trayectoria completa de paquetes desde el paquete raíz hasta la propia clase. Eso se puede hacer utilizando el operador **punto**: ".") para especificar cada subpaquete:

```
paquete_raiz.subpaquete1.subpaquete2. .... .subpaquete_n.NombreClase
```

```
java.lang.String
```

En este caso se está haciendo referencia a la clase **String** que se encuentra dentro del paquete **java.lang**. Este paquete contiene las clases elementales para poder desarrollar una aplicación Java. Otro ejemplo podría ser:

```
java.util.regex.Pattern
```

En este otro caso se hace referencia a la clase **Pattern** ubicada en el paquete **java.util.regex**, que contiene clases para trabajar con expresiones regulares.

Dado que puede resultar bastante tedioso tener que escribir la trayectoria completa de una clase cada vez que se quiera utilizar, existe la posibilidad de indicar que se desea trabajar con las clases de uno o varios paquetes. De esa manera cuando se vaya a utilizar una clase que pertenezca a uno de esos paquetes no será necesario indicar toda su trayectoria. Para ello se utiliza la sentencia **import** (importar):

```
import paquete_raiz.subpaquete1.subpaquete2. .... .subpaquete_n.NombreClase;
```

De esta manera a partir de ese momento podrá utilizarse directamente **NombreClase** en lugar de toda su trayectoria completa. Los ejemplos anteriores quedarían entonces:

```
import java.lang.String;
import java.util.regex.Pattern;
```

Si suponemos que vamos a utilizar varias clases de un mismo paquete, en lugar de hacer un **import** de cada una de ellas, podemos utilizar el **comodín** (símbolo **asterisco**: "*") para indicar que queremos importar todas las clases de ese paquete y no sólo una determinada:

```
import java.lang.*;
import java.util.regex.*;
```

Si un paquete contiene subpaquetes, el comodín no importará las clases de los subpaquetes, tan solo las que haya en el paquete. La importación de las clases contenidas en los subpaquetes habrá que indicarla explícitamente. Por ejemplo:

```
import java.util.*;
import java.util.regex.*;
```

En este caso se importarán todas las clases del paquete **java.util** (clases **Date**, **Calendar**, **Timer**, etc.) y de su subpaquete **java.util.regex** (**Matcher** y **Pattern**), pero NO las de otros subpaquetes como **java.util.concurrent** o **java.util.jar**.

Por último tan solo indicar que **en el caso del paquete java.lang, no es necesario realizar importación**. El compilador, dada la importancia de este paquete, permite el uso de sus clases sin necesidad de indicar su trayectoria (es como si todo archivo Java incluyera en su primera línea la sentencia **import java.lang.*** aunque no se escriba). El motivo, recordamos, es que este paquete contiene toda una serie de clases fundamentales para el buen funcionamiento del lenguaje, incluyendo la propia clase **Object**, de la que heredan todas las demás, por lo que no sería posible hacer nada sin disponer de ese paquete.

2.3 INCLUSIÓN DE UNA CLASE EN UN PAQUETE.

Al principio de cada archivo **.java** se puede indicar a qué paquete pertenece mediante la palabra reservada **package** seguida del nombre del paquete:

```
package nombrepaquete;

package packejemplo;
class ClaseEjemplo {
    ...
}
```

La sentencia **package** debe ser incluida en cada archivo fuente de cada clase que quieras incluir ese paquete. Si en un archivo fuente hay definidas más de una clase, todas esas clases formarán parte del paquete indicado en la sentencia **package**.

Si al comienzo de un archivo Java no se incluye ninguna sentencia **package**, el compilador considerará que las clases de ese archivo formarán parte del paquete por omisión (un paquete sin nombre asociado al proyecto), aunque se recomienda siempre usar algún paquete para el proyecto, distinto de paquete por defecto o por omisión.

Para evitar la ambigüedad, **dentro de un mismo paquete no puede haber dos clases con el mismo nombre**, aunque sí pueden existir clases con el mismo nombre si están en paquetes diferentes. El compilador será capaz de distinguir una clase de otra gracias a que pertenecen a paquetes distintos.

Como ya has visto en unidades anteriores, el nombre de un archivo fuente en Java se construye utilizando el nombre de la clase pública que contiene junto con la extensión **.java**, pudiendo haber únicamente una clase pública por cada archivo fuente. El nombre de la clase debía coincidir (en mayúsculas y minúsculas) exactamente con el nombre del archivo en el que se encontraba definida. Así, si por ejemplo tenías una clase **Punto** dentro de un archivo **Punto.java**, la compilación daría lugar a un archivo **Punto.class**.

En el caso de los paquetes, la correspondencia es a nivel de directorios o carpetas. Es decir, si la clase **Punto** se encuentra dentro del paquete **prog.figuras**, el archivo **Punto.java** debería encontrarse en la carpeta **prog\figuras**. Para que esto funcione correctamente el compilador ha de ser capaz de localizar todos los paquetes (tanto los estándares de Java como los definidos por otros programadores). Es decir, que el compilador debe tener conocimiento de dónde comienza la estructura de carpetas definida por los paquetes y en la cual se encuentran las clases. Para ello se utiliza el **ClassPath** cuyo funcionamiento hemos estudiado en las primeras unidades de este módulo. Se trata de una variable de entorno que contiene todas las rutas en las que comienzan las estructuras de directorios (distintas jerarquías posibles de paquetes) en las que están contenidas las clases.

Para saber más

En el siguiente enlace puedes aprender más sobre paquetes en Java. Concretamente se ve cómo compilar y ejecutar desde la consola (sin usar ningún IDE como NetBeans, etc.), con la estructura de paquetes. [Hola mundo con paquetes en Java](#)

2.4 PROCESO DE CREACIÓN DE UN PAQUETE.

Para crear un paquete en Java te recomendamos seguir los siguientes pasos:

1. **Poner un nombre al paquete.** Suele ser habitual utilizar el dominio de Internet de la empresa que ha creado el paquete. Por ejemplo, para el caso de **miempresa.com**, podría utilizarse un nombre de paquete **com.miemuestra**.
2. **Crear una estructura jerárquica de carpetas equivalente a la estructura de subpaquetes.** La ruta de la raíz de esa estructura jerárquica deberá estar especificada en el **ClassPath** de Java.
3. **Especificar a qué paquete pertenece la clase** (o clases) del archivo **.java** mediante el uso de la sentencia **package** tal y como has visto en el apartado anterior.

Este proceso ya lo has debido llevar a cabo en unidades anteriores al compilar y ejecutar clases con paquetes. Estos pasos simplemente son para que te sirvan como recordatorio del procedimiento que debes seguir a la hora de clasificar, jerarquizar y utilizar tus propias clases.

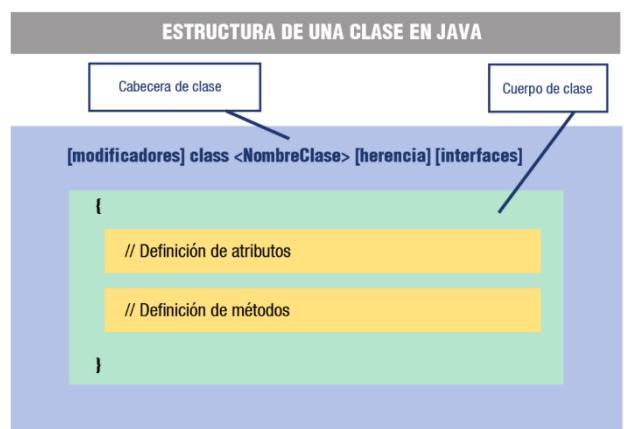
En Netbeans el proceso de creación de paquetes es muy sencillo, casi automático, pues basta con indicar que deseas crear un paquete y ya el IDE creará la carpeta por ti. A partir de ese momento aparecerá en tu proyecto un nuevo elemento representando el paquete que acabas de crear y dentro del cual podrás crear nuevas clases y subpaquetes.

3. ESTRUCTURA Y MIEMBROS DE UNA CLASE.

A estas alturas es muy probable que intuyas que para declarar una clase en Java se usa la palabra reservada **class**.

En la declaración de una clase vas a encontrar:

- **Cabecera de la clase.** Compuesta por una serie de **modificadores de acceso**, la palabra reservada **class** y el **nombre** de la clase.
- **Cuerpo de la clase.** En él se especifican los distintos **miembros** de la clase: **atributos** y **métodos**. Es decir, el contenido de la clase.



Como puedes observar, el **cuerpo de la clase** es donde se declaran los **atributos** que caracterizan a los objetos de la clase y donde se define e implementa el comportamiento de dichos objetos; es decir, donde se declaran e implementan los **métodos**.

3.1 DECLARACIÓN DE UNA CLASE.

La declaración de una clase en Java está basada en la palabra reservada **class** y tiene la siguiente estructura general:

```
// Cabecera de la clase
[modificadores] class <NombreClase> [herencia][interfaces] {

// Cuerpo de la clase

// Atributos
[Declaración de los atributos]

// Métodos
[Declaración de los métodos]

}
```

Todo lo que está entre corchetes es opcional. Por tanto, lo mínimo e imprescindible que necesitamos para declarar una clase en Java es la **palabra reservada class** más en el nombre que le queremos dar a la clase junto con las llaves que se abren y se cierran para codificar el cuerpo de la clase. Por el momento no necesitaríamos nada más.

¿Cómo podríamos hacerlo para nuestra clase **Vehiculo**?

Lo primero que tendríamos que hacer sería pensar en cómo queríamos llamar la clase (lo tenemos claro: **Vehiculo**). Eso junto con la palabra reservada **class** y las llaves que engloban al cuerpo podrían darnos una primera aproximación a nuestra clase:

```
class Vehiculo {
```

TEMA 5: DESARROLLO DE CLASES

En este caso se trata de una clase muy sencilla en la que **el cuerpo de la clase (el área entre las llaves) no contiene absolutamente nada**. Conforme vayamos avanzando habrá que ir rellenándola con el código y las declaraciones necesarias para que los objetos que se construyan (basándose en esta clase) puedan funcionar apropiadamente en un programa (declaración de atributos para contener el estado del objeto y declaración de métodos que implementen el comportamiento de los objetos creados a partir de ella). Ahora mismo no es más que un "envoltorio" vacío. Si queremos añadirle algunos comentarios para que quede más documentada, podríamos incluirlos. Aun así, seguiría siendo algo vacío y de poca utilidad. Simplemente hemos declarado una clase **Vehiculo** que no tiene ningún atributo ni ningún método:

```
// Declaración de la clase Vehiculo

// Cabecera de la clase:
// modificadores de acceso (ninguno),
// palabra reservada "class"
// nombre de la clase ("Vehiculo"),
// herencia (nada),
// implementación de interfaces (ninguna).
class Vehiculo {

    // Cuerpo de la clase: declaración de atributos e implementación de métodos -> Por ahora vacía

    // 1. Declaración de atributos: por ahora no hay ninguno

    // 2. Implementación de métodos: por ahora no hay ninguno
}
```

Si te fijas en los distintos programas que se han desarrollado en los ejemplos y las tareas de las unidades anteriores, podrás observar que cada uno de esos programas era en sí mismo una clase Java: se declaraban con la palabra reservada **class** y contenían algunos atributos (variables) así como algunos métodos (como mínimo el método **main**).

En el ejemplo anterior hemos visto lo mínimo que se tiene que indicar en la **cabecera de una clase** (el nombre de la clase y la palabra reservada **class**). Se puede proporcionar bastante más información mediante modificadores y otros indicadores como por ejemplo el nombre de su **superclase** (si es que esa clase hereda de otra), si implementa alguna interfaz (en inglés **interface**) y algunas cosas más que irás aprendiendo poco a poco.

A la hora de implementar **una clase Java** (escribirla en un archivo con un editor de textos o con alguna herramienta integrada como por ejemplo **Netbeans** o **Eclipse**) debes tener en cuenta:

- Como debes recordar, por convenio, se ha decidido que en lenguaje Java los nombres de las clases deben de **empezar por una letra mayúscula**. Así, cada vez que observes en el código una palabra con la primera letra en mayúscula sabrás que se trata de una clase sin necesidad de tener que buscar su declaración. Además, **si el nombre de la clase está formado por varias palabras, cada una de ellas también tendrá su primera letra en mayúscula**. No se usarán ni el carácter de subrayado también conocido como guión bajo o underscore: '_'), ni separadores como espacios en blanco. Tampoco recomendamos utilizar caracteres acentuados, la eñe, ni ningún otro carácter especial extraño por facilitar el trabajo a los teclados internacionales (imagina un código plagado de eñes y acentos que tenga que ser retocado por alguien que esté usando un teclado aglosajón en EEUU, podría ser una auténtica pesadilla). Siguiendo esta recomendación, algunos ejemplos de nombres de clases podrían ser **Circulo**, **Rectangulo**, **Jugador**, **Deposito**, **Ticket**, **JugadorDeFutbol**, **AnimalMarino**, **Vehiculo**, **VehiculoTerrestre**, **Persona**, **CuentaBancaria**, etc.
- **El archivo en el que se encuentra una clase Java debe tener el mismo nombre que esa clase** si queremos poder utilizarla desde otras clases que se encuentren fuera de ese archivo (**clase principal del archivo**). Por esa razón, mientras no se diga lo contrario, siempre tendremos **un archivo por cada clase** que desarrollemos.

TEMA 5: DESARROLLO DE CLASES

- Tanto la definición como la implementación de una clase se incluye en el mismo archivo (archivo ".java"). En otros lenguajes como por ejemplo C++, definición e implementación podrían ir en archivos separados (por ejemplo, en C++, serían sendos archivos con extensiones ".h" y ".cpp").

Para saber más

Si quieres ampliar un poco más sobre este tema puedes echar un vistazo a los tutoriales de iniciación de Java en el sitio web de Oracle: [Java Classes](#).

3.2 CABECERA DE UNA CLASE.

En general, la declaración de una clase puede incluir los siguientes elementos y en el siguiente orden:

1. **Modificadores** tales como **public**, **abstract** o **final**. No es obligatorio indicar ninguno.
2. El **nombre** de la clase (con la primera letra de cada palabra en mayúsculas, por convenio).
3. El nombre de su **clase padre (superclase)**, si es que se especifica, precedido por la palabra reservada **extends** ("extiende" o "hereda de"). Eso lo veremos en la siguiente unidad cuando aprendamos a usar la herencia. No es obligatorio indicar ningún tipo de herencia.
4. Una lista separada por comas de **interfaces** que son implementadas por la clase, precedida por la palabra reservada **implements** ("implementa"). También lo aprenderemos a utilizar en la próxima unidad. No es obligatorio implementar ninguna interfaz.
5. El **cuerpo** de la clase, encerrado entre llaves {}.

La sintaxis completa de una cabecera (los cuatro primeros puntos) queda de la forma:

```
// Cabecera de la clase
[modificadores] class <NombreClase> [extends <NombreSuperClase>]
[implements <NombreInterfaz1>] [, [<NombreInterfaz2>], ..., [<NombreInterfazN>]] {
```

En nuestro ejemplo de la clase **Vehículo** teníamos la siguiente cabecera:

```
class Vehiculo {
```

En este caso no hay **modificadores**, ni indicadores de herencia, ni implementación de **interfaces**. Tan solo la palabra reservada **class** y el nombre de la clase. Es lo mínimo que puede haber en la cabecera de una clase.

La **herencia** y las **interfaces** las verás más adelante. Vamos a ver ahora cuáles son los **modificadores** que se pueden indicar al crear la clase y qué efectos tienen. Los **modificadores de clase** son:

```
[public] [final | abstract]
```

Veamos qué significado tiene cada uno de ellos:

- Modificador **public**. Indica que la clase es visible (se pueden crear objetos de esa clase) desde cualquier otra clase. Es decir, desde cualquier otra parte del programa. Si no se especifica este modificador, la clase sólo podrá ser utilizada desde clases que estén en el mismo **paquete**. Sólo puede haber una clase **public** (clase principal) en un archivo .java. El resto de clases que se definen en ese archivo no serán públicas.
- Modificador **abstract**. Indica que la clase es **abstracta**. Una clase abstracta **no es instanciable**. Es decir, **no es posible crear objetos de una clase abstracta** (habrá que utilizar clases que hereden de ella). En este momento es posible que te parezca que no tenga sentido que esto pueda suceder (si no

TEMA 5: DESARROLLO DE CLASES

puedes crear objetos de esa clase, ¿para qué la quieres?), pero puede resultar muy útil a la hora de crear una jerarquía de clases. Esto lo verás también más adelante al estudiar el concepto de **herencia**.

- Modificador **final**. Indica que no podrás crear clases que hereden de ella. También volverás a este modificador cuando estudies el concepto de **herencia**. Los modificadores **final** y **abstract** son excluyentes (sólo se puede utilizar uno de ellos al mismo tiempo).

Todos estos modificadores y palabras reservadas las iremos viendo poco a poco, así que no te preocupes demasiado por intentar entender todas ellas en este momento.

En el ejemplo anterior de la clase **Vehiculo** tendríamos una clase que sería sólo visible (utilizable) desde el mismo paquete en el que se encuentra la clase (modificador de acceso por omisión o de paquete, o **package**). Desde fuera de ese paquete no sería visible o accesible. Para poder utilizarla desde cualquier otra clase desde cualquier paquete bastaría con añadir el atributo **public**:

```
public class Vehiculo {
```

Normalmente declararemos las clases como **public**.

3.3 CUERPO DE UNA CLASE

Como ya has visto anteriormente, el cuerpo de una clase se encuentra encerrado entre llaves y contiene la declaración e implementación de sus miembros. Los miembros de una clase pueden ser:

- **atributos**, que especifican los datos que podrá contener un objeto de la clase,
- **métodos**, que implementan las acciones que se podrán realizar con un objeto de la clase (el comportamiento).

Una clase puede no contener en su declaración atributos o métodos, pero debería contener al menos uno de los dos, pues una clase totalmente vacía no tiene ninguna utilidad.

Ahora ha llegado el momento de empezar a pensar en qué atributos podría tener nuestra clase **Vehiculo** que estamos construyendo poco a poco. Algunos ejemplos de posibles atributos para nuestra clase podrían ser:

- Nivel del depósito de combustible (un número real que exprese una cantidad en litros).
- Año de matriculación (un número entero).
- Estado del motor (si está apagado o arrancado).

Para cada uno de esos posibles atributos habrá que decidir qué **nombre** le asignamos (identificador) y qué **tipo** de Java le vamos a asociar (un tipo primitivo o bien algún tipo más complejo, como por ejemplo un **String**). Por ejemplo, podríamos plantearlo así:

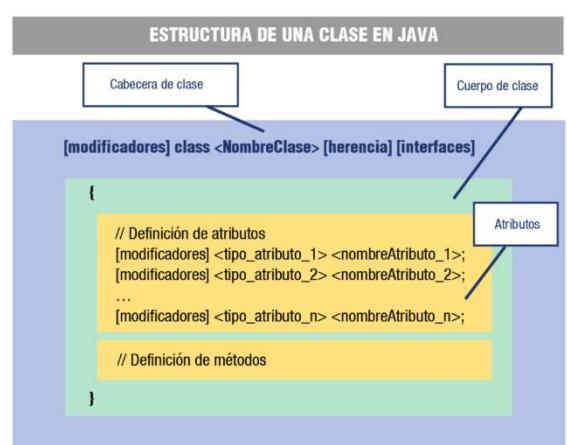
- Nivel del depósito de combustible: nombre **nivelDeposito** y tipo **double**.
- Año de matriculación: nombre **matriculacion** y tipo **short**. Podríamos haber usado **int** y también habría sido correcto, pero teniendo en cuenta va a ser siempre un número de cuatro cifras, es suficiente con un entero más pequeño (se utilizan menos bytes y por tanto ocupa menos memoria).
- Estado del motor: nombre **estadoMotor** y tipo **boolean**.

¿Y cómo expresaríamos eso en lenguaje Java? Pues simplemente declarando esos atributos dentro del cuerpo de la clase como si fueran variables:

```
public class Vehiculo {  
    double nivelDeposito;  
    short añoMatriculacion;  
    boolean estadoMotor;  
}
```

4. ATRIBUTOS.

Los **atributos** constituyen la **estructura interna de los objetos de una clase**. Se trata del conjunto de datos que los objetos de una determinada clase almacenan cuando son creados. Posteriormente esos datos pueden ir modificando su valor a lo largo de la vida del programa. Es decir, es como si fueran variables cuyo ámbito de existencia es el objeto dentro del cual han sido creadas. Fuera del objeto esas variables no tienen sentido y si el objeto deja de existir, esas variables también deberían desaparecer (proceso de destrucción del objeto).



Los atributos a veces también son conocidos con el nombre de propiedades (properties), campos (fields), variables miembro o variables de objeto.

Los atributos pueden ser de cualquier tipo del que pueda ser cualquier otra variable en un programa en Java: desde **tipos primitivos** como **int**, **boolean** o **float** hasta **tipos referenciados** como **arrays** u otros objetos con los que ya has trabajado (**String**, **LocalDate**, **LocalTime**, **StringBuilder**, etc.).

Además del tipo y del nombre, la declaración de un atributo puede contener también algunos **modificadores** (por ejemplo, **public**, **private**, **protected** o **static**). En el caso de la clase **Vehiculo** con la que hemos empezado a trabajar en apartados anteriores hemos declarado algunos de sus atributos como:

```
public class Vehiculo {
    double nivelDeposito;
    short añoMatriculacion;
    boolean estadoMotor;
}
```

Hasta este momento no nos habíamos planteado si los atributos debían incorporar también modificadores. Pues así es: también pueden llevar modificadores y de hecho lo recomendable es que los atributos lleven el atributo **private**. Por otro lado, nuestra sugerencia como norma general es **evitar el uso de caracteres no estándar en los identificadores** como por ejemplo la eñe (**añoMatriculacion**). Por tanto, nuestra clase **Vehiculo** debemos modificarla en este aspecto, y podría quedar así:

```
public class Vehiculo {
    private double nivelDeposito;
    private short añoMatriculacion;
    private boolean estadoMotor;
}
```

Como ya verás más adelante al estudiar el concepto de **encapsulación**, lo normal es declarar todos los atributos (o al menos la mayoría) como privados (**private**) de manera que si se desea acceder o manipular algún atributo se tenga que hacer a través de los métodos proporcionados por la clase.

Lo habitual: atributos privados y métodos de acceso públicos.

4.1 DECLARACIÓN DE ATRIBUTOS.

La sintaxis general para la declaración de un atributo en el interior de una clase es:

```
[modificadores] <tipo> <nombreAtributo>;
```

Ejemplos:

```
int x;                                // Un atributo de tipo int (entero)
static double descuentoGeneral;        // Un atributo de tipo double (real)
short numeroRuedas;                   // Un atributo de tipo short (entero corto)
float porcentajeDescuento;            // Un atributo de tipo double (real)
final boolean casado;                 // Un atributo de tipo boolean (lógico)
public static final int VALOR_MAXIMO; // Un atributo de tipo int (entero) y constante
public String nombre;                 // Un atributo de tipo String (cadena de caracteres)
private LocalTime horaDeLlegada;       // Un atributo de tipo LocalTime (hora)
```

Te suena bastante, ¿verdad?

La declaración de los atributos en una clase **es exactamente igual a la declaración de cualquier variable** tal y como has estudiado en las unidades anteriores y similar a como se hace en cualquier lenguaje de programación no necesariamente orientado a objetos. Es decir, mediante la indicación del tipo y a continuación el nombre del atributo.

También puedes declarar a la vez varios atributos del mismo tipo mediante una lista de nombres separada por comas (exactamente como ya has estudiado al declarar variables). Por ejemplo:

```
private StringBuilder nombre, apellido1, apellido2; // Tres atributos de tipo StringBuilder (cadena de caracteres)
private double peso, altura;                      // Dos atributos de tipo double (real)
private final LocalDate fechaInicio, fechaFin;    // Dos atributos de tipo LocalDate (fecha)
```

La declaración de un **atributo** (o **propiedad**, o **campo**, o **variable miembro** o **variable de objeto**) podría considerarse como la declaración de una variable que únicamente existe en el interior del objeto y por tanto su vida comenzará cuando el objeto comience a existir (el objeto sea creado por un **constructor**). Esto significa que cada vez que se cree o instancie un objeto se crearán tantas variables como atributos contenga ese objeto en su interior (definidas en la clase, que es la plantilla o "molde" del objeto). Todas esas variables estarán encapsuladas dentro del objeto y sólo tendrán sentido dentro de él.

En el ejemplo de clase que estamos diseñando con objetos de tipo **Vehiculo** (instancias de la clase **Vehiculo**), cada vez que se cree un nuevo vehículo **v1**, se crearán atributos **nivelDeposito**, **matriculacion** y **estadoMotor** de tipo **double**, **short** y **boolean** que estarán en el interior de ese vehículo **v1** (algo así como variables "internas" que sólo existen en el contexto de ese objeto vehículo **v1**). Si a continuación se crea un nuevo objeto **Vehiculo v2**, se crearán otros tres nuevos atributos **nivelDeposito**, **matriculacion** y **estadoMotor** de tipo **double**, **short** y **boolean** que estarán esta vez alojados en el interior de **v2**. Y así con todos los objetos de tipo **Vehiculo** que instanciemos o creemos. Cada **v1**, **v2**, **v3**, etc., tendrá sus propios valores para esos atributos (son vehículos independientes cada uno con sus características específicas).

Dentro de la declaración de un atributo puedes encontrar tres partes:

- **Modificadores.** Son palabras reservadas que permiten modificar la utilización del atributo (indicar el control de acceso, si el atributo es constante, si se trata de un atributo de clase, etc.). Los iremos viendo uno a uno. En los ejemplos anteriores ya has tenido la oportunidad de observar algunos de ellos como **public**, **private**, **static**, etc.

TEMA 5: DESARROLLO DE CLASES

- **Tipo.** Indica el tipo del atributo. Puede tratarse de un tipo primitivo (**int**, **char**, **boolean**, **double**, etc.) o bien de uno referenciado (objetos como por ejemplo **String**, **LocalDate** o **StringBuilder**, un array, otra clase implementada previamente por ti, etc.).
- **Nombre.** Identificador único para el nombre del atributo. Por convenio se **suelen utilizar las minúsculas**. En caso de que se trate de un identificador que contenga varias palabras, **a partir de la segunda palabra se suele poner la letra de cada palabra en mayúsculas**, tal y como ya hacíamos para declarar las variables (notación "camel case"). Por ejemplo: **primerValor**, **valor**, **puertalzquierda**, **cuartoTrasero**, **equipoVecendor**, **sumaTotal**, **nombreCandidatoFinal**, etc. Cualquier identificador válido de Java será admitido como nombre de atributo válido, pero **es muy importante seguir este convenio** para facilitar la legibilidad del código (todos los programadores de Java lo utilizan y nosotros también lo haremos). Si te fijas, es el **mismo convenio que se utiliza para las variables**. De este modo, cuando veamos un identificador que sigue ese convenio en cualquier parte del código sabremos rápidamente que se trata de un atributo o de una variable local.

Y ahora, antes de seguir avanzando, ha llegado el momento de que nos tomemos unos minutos para reflexionar sobre este tema de la declaración de atributos y realicemos un pequeño ejercicio para consolidar lo que llevamos aprendido.

Reflexiona

Casi siempre nos va a suceder que tendremos la opción de escoger entre varias posibilidades a la hora de decidir el **tipo de un atributo**. De hecho, ya te habrás encontrado con esa disyuntiva desde el comienzo del curso cada vez que vas a declarar **una variable**. Por ejemplo, para **enteros** hay cuatro posibilidades en Java (**byte**, **short**, **int**, **long**), para **reales** tienes dos (**float**, **double**), para **cadenas de caracteres** tienes también varias opciones (**String**, **StringBuilder**, **StringBuffer**).

Los **tipos primitivos** de Java ya los has estudiado y conoces las diferencias entre unos y otros: cuantos más bytes se utilicen para representar el valor, más preciso se podrá ser a la hora de almacenarlo y representarlo, aunque obviamente también se ocupará más memoria. En el caso de las **clases**, tendrás que decidir cuál es la que mejor se ajusta a la información que pretendes representar (un texto, una fecha, un intervalo entre fechas, una hora, una fecha y hora a la vez, etc.).

Una vez que tenemos claro el tipo de un atributo en "abstracto", de manera independiente al lenguaje de programación (entero, real, cadena de caracteres, etc.), ¿qué criterio deberíamos utilizar a la hora de decidirnos por uno u otro tipo en Java si disponemos de varias alternativas?

En el caso de los **enteros** en Java, normalmente se suele escoger **int** por simplicidad, ya que los literales de tipo entero son por omisión **int** en Java (recuerda que si quieres que un literal sea **long** debes añadirle el sufijo **L**). Pero si estás seguro de que la cantidad que vas a almacenar siempre va a estar por debajo de un determinado límite, puedes decidir utilizar algún tipo más "pequeño" como por ejemplo un **short** o un **byte**. Por ejemplo, si vas a almacenar la edad de una persona en número de años, probablemente tengas de sobra con un **byte** (no vas a superar los 127, que sería el mayor valor positivo representable por un **byte**), pero si vas a almacenar la edad de una persona en número de días, necesitarías un **short** (necesitas almacenar números de hasta decenas de miles, lo cual "cabe" en un **short** de sobra). En cualquier caso, siempre debes ser un poco precavido. Imagina por ejemplo que vas a almacenar la edad en años de personajes mitológicos, legendarios, imaginarios, etc. Entonces quizás sí podrían tener varios cientos de años y te vendría bien un **short**.

En cualquier caso, tampoco te obsesiones demasiado con esta elección, pues la diferencia entre usar uno u otro tipo son unos pocos bytes. Realizar este análisis seriamente sólo vale la pena cuando se van a utilizar estructuras con millones de estos valores (por ejemplo, en arrays muy grandes) y entonces sí puede ser importante la elección de uno u otro por la cantidad de bytes que se pueden ahorrar tanto desde el punto de

TEMA 5: DESARROLLO DE CLASES

vista del almacenamiento (ocupación de memoria) como del tiempo de transmisión (cuantos más bytes haya que transmitir, más tiempo se tardará). Lo normal es que en aplicaciones normales se use el tipo **int** para representar enteros y ni se recuerde que existen el **byte** y el **short**. Tan solo en algunos casos se recurre al **long** porque se prevea que se vaya a trabajar con números enteros muy grandes.

En cuanto a los **reales** en Java, se suele elegir **double** por la misma razón que en el caso de los **int**: porque los literales de tipo real en Java son por omisión de tipo **double** (si queremos que sean **float** deben tener el sufijo **f**). Ahora bien, si tenemos claro que para nuestros fines no vamos a necesitar más de dos o tres cifras significativas, podríamos declarar ese atributo como **float**.

Para el caso de **cadenas de caracteres** en Java, se suele optar por la clase **String** nuevamente por simplicidad y porque es lo más habitual. Pero también podría usarse **StringBuilder** si se trata de un atributo que puede cambiar de valor frecuentemente. En tal caso sería más eficiente trabajar con objetos **StringBuilder**, que pueden ser modificados, que con objetos **String**, que no puedes modificar (los objetos de la clase **String** son inmutables). Se crea un nuevo **String** cada vez que se lleva a cabo alguna operación de modificación sobre un objeto **String**.



Ejercicio Resuelto

Tras varias reuniones del equipo de diseño en las que se ha estado discutiendo el modelo de una clase para representar vehículos, se ha decidido que por el momento va a tener las siguientes características (atributos):

- ✓ **Capacidad del depósito de combustible.** En litros.
- ✓ **Matrícula.**
- ✓ **Fecha de matriculación.** Se ha decidido que en lugar de almacenar sólo el año, vamos a almacenar la fecha completa (día, mes y año).
- ✓ **Estado del motor** (encendido o apagado).
- ✓ **Distancia recorrida desde que se fabricó el vehículo** (memoria de ruta global). En kilómetros.
- ✓ **Distancia recorrida desde que se arrancó por última vez** (memoria de ruta parcial). En kilómetros.



MikesPhotos (Pixabay License)

Ahora nos toca a nosotros, como programadores, realizar un análisis en el que tendremos que decidir, para cada una de esas características:

- ✓ Un **nombre** o **identificador** para cada una de ellas (nombre del atributo).
- ✓ Un **tipo** genérico para poder representar apropiadamente esa característica (**número entero**, **número real**, **texto**, **fecha**, etc.). No es necesario que "afinemos" con un tipo específico del lenguaje Java (por ejemplo, si se trata de un número entero basta con que indiquemos "número entero", todavía no hay que indicar si va a ser **byte**, **short**, **int** o **long**).

Una vez que tenemos claro qué atributos va a tener nuestra clase (eso ya nos lo acaban de proporcionar en el enunciado), habrá que escoger un **nombre** o **identificador** para cada uno de ellos. Se trata de un proceso importante, porque de ello dependerá mucho la legibilidad de nuestro código. Cuanto más claro dejemos lo que se almacena o representa un atributo, mucho más fácil será entender el código cuando lo leamos.

Además del nombre, indicaremos también qué tipo de información se va a almacenar en ese atributo (números enteros, números reales, textos, fechas, valores lógicos, etc.).

Y éstas son las conclusiones a las que podríamos llegar:

- ✓ **capacidad del depósito de combustible.** En este atributo se va a almacenar un valor que representa **litros de combustible**, por tanto un nombre razonable podría ser **capacidadDeposito**. Por supuesto no es la única posibilidad. Se trata de escoger un nombre que identifique claramente qué es lo que se guarda dentro de ese atributo. Otras opciones podrían ser: **capacidadDelDeposito**, **litrosDeposito**, etc. En cuanto al tipo, el más apropiado para representar litros y sus posibles fracciones es un **tipo real**;
- ✓ **matrícula.** Se trata de un **texto** que representa la matrícula. Por tanto el nombre **matrícula** parece más que apropiado. Necesitaremos un **tipo texto** (cadena de caracteres);
- ✓ **fecha de matriculación.** Un nombre que no deja lugar a dudas podría ser **fechaMatriculacion**, aunque no es el único posible. Otras opciones podrían ser: **fechaDeMatriculacion**, **fechaMatricula**, etc. Necesitaremos algún tipo que permita representar una **fecha**;
- ✓ **estado del motor** (encendido o apagado). Aquí podría elegirse por ejemplo el nombre **estadoMotor**, aunque hay otras posibilidades que también podrían ser razonables. Por ejemplo: **encendido**, **estado**, **arrancado**, etc. Para este atributo vendría bien algún tipo que permita representar dos estados posibles: un **tipo lógico** o **booleano**;
- ✓ **distancia recorrida desde que se fabricó el vehículo.** En este caso podría optarse por el nombre **distanciaGlobal**, aunque se nos pueden ocurrir muchas otras alternativas: **kilometrosTotales**, **kilometrosTotalesRecorridos**, **kilometrosDesdeInicio**, **distanciaTotal**, etc. Dado que hay que representar kilómetros y sus posibles fracciones, lo apropiado en este caso sería un **tipo real**;
- ✓ **distancia recorrida desde que se arrancó por última vez.** En este caso habría que elegir un nombre que permitiera distinguirlo claramente del atributo anterior, para no dar lugar a confusión. Algunos ejemplos podrían ser: **distanciaParcial**, **distanciaDesdeUltimoArranque**, **distanciaLocal**, **kilometrosLocales**, **kilometrosParciales**, etc. Lo importante será que quede clara la diferencia entre ambos atributos simplemente con mirar el nombre. Respecto al tipo, nuevamente necesitaremos un **tipo real**.

TEMA 5: DESARROLLO DE CLASES

Una vez realizado ese análisis y decidido el **nombre** o **identificador** que se va a utilizar para cada atributo así como el tipo "genérico" que va a tener cada uno, habrá que escoger **tipos primitivos** de Java que representen apropiadamente las magnitudes o valores que queremos almacenar en ellos. Si fuéramos a trabajar con otro lenguaje diferente a Java (C++, PHP, Python, Ruby, etc.) habría que amoldarse a los tipos que ofrezca ese lenguaje.

Si la información que va a contener un atributo es demasiado compleja para poder ser representada por un tipo primitivo de Java, también puede recurrirse a **clases** ya disponibles en la API de Java, o incluso a otras clases que hayan sido implementadas por otros programadores.

Lleva a cabo este nuevo análisis y rellena una tabla en la que indiques, para cada atributo, qué tipo primitivo (o clase) de Java se le va a asignar.

Tendremos ahora que analizar para cada atributo cuál sería el tipo de Java más apropiado:

- ✓ **capacidad del depósito de combustible**. Para este atributo se ha decidido que es necesario un tipo **real**, así que podría usarse un **double** o bien **un float**. Nosotros usaremos **double** por ser el tipo por omisión que se usa para los literales de tipo real en Java.
- ✓ **matrícula**. Se trata de un **texto** para representar matrículas al estilo de, por ejemplo, "4264KPY". Por tanto podría usarse alguna clase que sirva para almacenar **cadenas de caracteres** como **String** o **StringBuilder**. Usaremos un **String**.
- ✓ **fecha de matriculación**. La API de Java proporciona clases para representar **fechas**, como por ejemplo **LocalDate**, así que usaremos esa clase. Es mucho más cómodo que tener que almacenar tres números enteros (día, mes año);
- ✓ **estado del motor** (encendido o apagado). Necesitamos un tipo que represente **dos estados posibles**. Java nos proporciona el tipo **boolean**;
- ✓ **distancia recorrida desde que se fabricó el vehículo**. Se trata nuevamente de otro **número real**. Por tanto podría usarse **double** o **float**. Escogeremos **double**;
- ✓ **distancia recorrida desde que se arrancó por última vez**. Lo mismo que en el caso anterior: algún tipo de Java que represente **números reales** como por ejemplo **double**.

Si no tienes claro cuándo usar uno u otro tipo en el caso de que dispongas de varias opciones, te recomendamos que vuelvas a leer la reflexión anterior sobre qué tipo elegir cuando tienes varias posibilidades.

El resultado del análisis anterior en forma de tabla quedaría de la siguiente manera:

Atributos de la clase **Vehículo**

Nombre	Tipo
capacidadDeposito	double
matricula	String
fechaMatriculacion	LocalDate
estadoMotor	boolean
kilometrosTotales	double
kilometrosParciales	double

A partir de la **tabla** que has confeccionado con el **nombre** y el **tipo** para cada atributo, escribe la **clase Java que representa a un vehículo**.

Ten en cuenta que se desea que sea una **clase visible desde cualquier otra clase**.

Además del nombre y el tipo de cada atributo, debemos también tener en cuenta las cuestiones de visibilidad que nos han indicado en el enunciado: "...que sea una **clase visible desde cualquier otra clase**". Por tanto tendrá que ser una clase con el modificador **public**, que en cualquier caso es lo que haremos normalmente.

Nuestra clase **Vehículo** en Java quedaría así, de momento:

```
public class Vehiculo {  
    double capacidadDeposito;           // Capacidad del depósito de combustible del vehículo (en litros)  
    String matricula;                   // Matrícula del vehículo  
    LocalDate fechaMatriculacion;       // Fecha de matriculación del vehículo  
    boolean estadoMotor;                // Estado del motor del vehículo (arrancado o apagado)  
    double kilometrosTotales;          // Distancia total recorrida desde que fabricó el vehículo (en kilómetros)  
    double kilometrosParciales;         // Distancia recorrida desde que el vehículo se arrancó por última vez (en kilómetros)  
}
```

Si te fijas, hemos añadido **un comentario a cada atributo** indicando qué es lo que representa o qué se va a almacenar en su interior. **No es algo estrictamente necesario, pero es muy recomendable para facilitar la legibilidad del código**. En cualquier caso, si hemos elegido bien el nombre de cada atributo, es posible que no sea necesario acudir a ese comentario para intuir qué es lo que se está almacenando en él. Cuanto más transparente, legible y natural se vea el código, más fácil nos resultará nuestro trabajo como programadores (y el de nuestros compañeros si alguna vez tienen que revisar nuestro código).

4.2 MODIFICADORES DE ATRIBUTO.

Una vez realizada la reflexión anterior y habiendo resuelto un ejercicio que nos ha ayudado a asimilar las cuestiones más básicas de la declaración de un atributo (**nombre** y **tipo**), podemos seguir avanzando y hablar de los **modificadores para atributos**. Como ya has visto, los atributos de una clase también pueden contener modificadores en su declaración (como sucedía al declarar la propia clase). Estos modificadores permiten indicar cierto comportamiento de un atributo a la hora de utilizarlo. Entre los modificadores de un atributo podemos distinguir:

- **modificadores de acceso (public, private y protected)**. Indican la forma de acceso al atributo desde otra clase. Son modificadores excluyentes entre sí. Sólo se puede poner uno. De hecho ya estás utilizando el modificador **private** como norma general para la declaración de atributos tal y como te hemos recomendado;
- **modificadores de contenido (static y final)**. No son excluyentes. Pueden aparecer varios a la vez;
- **otros modificadores: transient y volatile**. El primero se utiliza para indicar que un atributo es transitorio (no persistente) y el segundo es para indicar al compilador que no debe realizar optimizaciones sobre esa variable. Lo más probable es que no necesites utilizarlos en este curso de introducción a la programación. Puedes olvidarte de ellos por ahora.

Aquí tienes la sintaxis completa de la declaración de un atributo teniendo en cuenta la lista de todos los modificadores e indicando cuáles son compatibles o incompatibles entre ellos:

```
[private | protected | public] [static] [final] [transient] [volatile] <tipo> <nombreAtributo>;
```

4.3 MODIFICADORES DE ACCESO.

Los **modificadores de acceso disponibles** en Java para un atributo son:

- modificador de acceso **por omisión** (o **de paquete** o **package**). Si **no se indica ningún modificador de acceso** en la declaración del atributo, se utilizará este tipo de acceso. Se permitirá el acceso a este atributo desde todas las clases que estén dentro del **mismo paquete (package)** que esta clase (la que contiene el atributo que se está declarando). No es necesario escribir ninguna palabra reservada. Si no se pone nada se supone que se desea indicar este modo de acceso;
- modificador de acceso **public**. Indica que **cualquier clase** (por muy ajena o lejana que esté) tiene acceso a ese atributo. No es muy habitual declarar atributos públicos (**public**);
- modificador de acceso **private**. Indica que sólo se puede acceder al atributo desde **dentro de la propia clase**. El atributo estará "oculto" para cualquier otra zona de código fuera de la clase en la que está declarado. Es lo opuesto a lo que permite **public**. Será para nosotros el **modificador más habitual**;
- modificador de acceso **protected**. En este caso se permitirá acceder al atributo desde cualquier **subclase** (lo verás más adelante al estudiar la **herencia**) de la clase en la que se encuentre declarado el atributo, y también desde las clases del **mismo paquete**. Es decir, cualquier clase del mismo paquete, pero también las subclases, aunque estén en distinto paquete. En esta unidad no lo vamos a utilizar.

TEMA 5: DESARROLLO DE CLASES

Cuadro de niveles accesibilidad a los atributos de una clase.

Modificador de acceso (De menos restrictivo a más restrictivo)	Misma clase	Cualquier clase del mismo paquete	Cualquier subclase de otro paquete	Cualquier clase de otro paquete
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
Sin modificador (package)	Sí	Sí	No	No
private	Sí	No	No	No

¡Recuerda que los modificadores de acceso son excluyentes! Sólo se puede utilizar uno de ellos en la declaración de un atributo.



Ejercicio Resuelto

Seguimos con el modelo de nuestra clase **Vehículo** con los siguientes atributos:

- ✓ Capacidad del depósito de combustible.
- ✓ Matrícula.
- ✓ Fecha de matriculación.
- ✓ Estado del motor (encendido o apagado).
- ✓ Distancia recorrida desde que se fabricó el vehículo.
- ✓ Distancia recorrida desde que se arrancó por última vez.



Clicker-Free-Vector-Images (Pixabay License)

A partir de la **tabla** que has confeccionado con el **nombre** y el **tipo** para cada atributo en un ejercicio anterior, añade una **tercera columna** en la que indiques la visibilidad de cada atributo teniendo en cuenta que deseamos que **todos los atributos de la clase sólo sean accesibles desde la propia clase**.

Dado que todos los atributos de la clase van a ser visibles sólo desde la propia clase (privados) bastaría con añadir esa tercera columna indicando que todos son atributos **privados**:

Atributos de la clase Vehículo

Nombre	Tipo	Visibilidad
capacidadDeposito	double	privada (private)
matricula	String	privada (private)
fechaMatriculacion	LocalDate	privada (private)
estadoMotor	boolean	privada (private)
kilometrosTotales	double	privada (private)
kilometrosParciales	double	privada (private)

A partir de la tabla anterior escribe cómo quedaría la clase en Java.

A partir de la tabla anterior podemos escribir la clase Java sin dificultad. Se trata simplemente de añadir el modificador **private** a todos los atributos que ya teníamos en nuestra clase **Vehículo**.

```
public class Vehiculo {
    private double capacidadDeposito;           // Capacidad del depósito de combustible del vehículo (en litros)
    private String matricula;                   // Matrícula del vehículo
    private LocalDate fechaMatriculacion;        // Fecha de matriculación del vehículo
    private boolean estadoMotor;                // Estado del motor del vehículo (arrancado o apagado)
    private double kilometrosTotales;          // Distancia total recorrida desde que fabricó el vehículo (en kilómetros)
    private double kilometrosParciales;         // Distancia recorrida desde que el vehículo se arrancó por última vez (en kilómetros)
}
```

4.4 MODIFICADORES DE CONTENIDO: ATRIBUTOS CONSTANTES (I)

Continuando con ejemplo de la clase **vehículo**, podríamos llegar a la siguiente conclusión respecto a los atributos:

1. algunos atributos no van a cambiar nunca de valor una vez se fabrique el vehículo;
2. otros, sin embargo, podrán ir modificando su valor a lo largo del tiempo. En cada momento que se consulten, su valor puede ser diferente.

En el primer caso estaríamos hablando de datos que una vez creado el objeto ya no tiene sentido que se modifique su valor. Es decir, se trata de características del objeto que forman parte de su "naturaleza" (podemos considerarlos como información "estructural" del objeto). Es algo que no va a cambiar a lo largo del tiempo y que va a permanecer inalterable. Algunos ejemplos podrían ser:

- la **capacidad del depósito de combustible** del vehículo;
- la **matrícula** del vehículo;
- el **año de matriculación** o, si se prefiere, la **fecha completa de matriculación** del vehículo;
- la **máxima velocidad** que puede alcanzar el vehículo;
- el **número de marchas** del vehículo (si es que es de cambio manual);

Se trata claramente de valores que jamás van a cambiar para un vehículo concreto una vez creado.

En el segundo caso, se trata de características que sí pueden cambiar con el tiempo y que podríamos decir que representarían su "estado". Algunos ejemplos podrían ser:

- la **velocidad actual** a la que va el vehículo,
- el **nivel actual de combustible** en el depósito,
- la **distancia recorrida** hasta el momento,
- el **estado del motor** (si está apagado o encendido),

Cada uno de esos datos podrá ser diferente cada vez que se consulten. Tendrán un valor inicial al crearse el objeto e irán cambiando a lo largo del tiempo.

Para implementar el primer grupo de atributos (atributos de "naturaleza" del objeto, inmutables), podemos utilizar un modificador de contenido que obligue a que ese valor sea constante, es decir, que una vez que se le asigne un valor, éste no pueda ser modificado. Para poder lograr esto en Java contamos con el **modificador de contenido final**.

Para implementar el segundo grupo de atributos (atributos de "estado" del objeto, variables), simplemente podemos declararlos como lo hemos hecho hasta ahora. De este modo serán valores variables y no constantes.

Los modificadores de contenido no son excluyentes (pueden aparecer varios para un mismo atributo). Son los siguientes:

- modificador **final**. Indica que el atributo es una **constante**. Su valor no podrá ser modificado a lo largo de la vida del objeto. Por convenio, el nombre de los **atributos constantes (final)** se escribe con **todas las letras en mayúsculas, separando las palabras con el carácter '_'**. Aunque ahora vamos a ver una excepción;
- modificador **static**. Hace que el atributo sea común para todos los objetos de una misma clase. Es decir, todos los objetos de la clase compartirán ese mismo atributo con el mismo valor. Es un caso de miembro estático o miembro de clase: un **atributo estático** o **atributo de clase** o **variable de clase**.

En el siguiente apartado sobre atributos estáticos verás ejemplos de atributos estáticos (**static**). Veamos ahora un ejemplo de atributos constantes (**final**).



Ejercicio Resuelto

Continuamos con nuestra clase de ejemplo con la que pretendemos representar vehículos. Hasta el momento habíamos decidido incluir los siguientes atributos:

- ✓ **capacidad del depósito de combustible.** En litros;
- ✓ **matrícula;**
- ✓ **fecha de matriculación;**
- ✓ **estado del motor** (encendido o apagado);
- ✓ **distancia recorrida desde que se fabricó el vehículo** (memoria de ruta global). En kilómetros;
- ✓ **distancia recorrida desde que se arrancó por última vez** (memoria de ruta parcial). En kilómetros.



Izvar Muis (Pixabay License)

Debemos ahora realizar, para cada atributo, un análisis para decidir:

1. **cuáles deberían ser constantes o inmutables;**
2. **cuáles deberían poder cambiar su valor** con el tiempo (mutables o variables).

En actividades anteriores ya hemos decidido qué **identificadores** (nombres) asignar a cada uno de estos atributos, así como su **tipo** y su **visibilidad**.

Ahora habrá que analizar cuáles tiene sentido que sean constantes o inmutables (representan una característica que nunca va a cambiar) y cuáles no (representan el estado del objeto en cada momento del tiempo). Podríamos llegar a las siguientes conclusiones:

- ✓ **capacidad del depósito de combustible.** Se trata de una propiedad invariable para cada vehículo en particular. Una vez que se fabrique un determinado vehículo, siempre tendrá la misma capacidad de combustible pues el tamaño físico del depósito no va a variar. Se trata por tanto de un atributo **constante**: modificador `final`;
- ✓ **matrícula.** Una vez matriculado el vehículo, no podemos cambiar su matrícula, salvo casos excepcionales como importación/exportación o cosas así. Pero no vamos a tener en cuenta ese caso. Por tanto lo trataremos como un atributo inmutable o **constante**: modificador `final`;
- ✓ **fecha de matriculación.** Una vez que el vehículo es matriculado, esa es su fecha de matriculación. No debería cambiar, salvo que se quisiera tener en cuenta la misma excepción que en el caso anterior. Es por tanto también **constante** e irá acompañado de un modificador `final`;
- ✓ **estado del motor** (encendido o apagado). El vehículo podrá estar apagado en un determinado momento y pasar a estado encendido en cuanto se arranque. Se trata entonces de un atributo **variable** y no necesita de ningún modificador adicional;
- ✓ **distancia recorrida desde que se fabricó el vehículo.** Comenzará valiendo cero e irá aumentando según se vaya utilizando el vehículo. Nunca disminuirá. Será **variable**,
- ✓ **distancia recorrida desde que se arrancó por última vez.** Comenzará valiendo cero e irá aumentando según se vaya utilizando el vehículo, pero cada vez que se apague y se vuelva a arrancar comenzará de nuevo desde cero. Por tanto también será **variable**;

Una vez realizado ese análisis, podemos incluir una **nueva columna en nuestra tabla de atributos** para indicar, además del nombre, el tipo y la visibilidad, la posible **mutabilidad** o inmutabilidad del atributo (es decir, si va a ser un atributo constante o variable).

Aquí tenemos una tabla resumen de nuestros atributos una vez realizado el anterior análisis en el que hemos valorado, para cada uno, si debería ser constante o variable (condición de mutabilidad o inmutabilidad):

Atributos de la clase `Vehiculo`

Nombre	Tipo	Visibilidad	Mutabilidad (constante/variable)
<code>capacidadDeposito</code>	<code>double</code>	privada (<code>private</code>)	constante (<code>final</code>)
<code>matricula</code>	<code>String</code>	privada (<code>private</code>)	constante (<code>final</code>)
<code>fechaMatriculacion</code>	<code>LocalDate</code>	privada (<code>private</code>)	constante (<code>final</code>)
<code>estadoMotor</code>	<code>boolean</code>	privada (<code>private</code>)	variable
<code>kilometrosTotales</code>	<code>double</code>	privada (<code>private</code>)	variable
<code>kilometrosParciales</code>	<code>double</code>	privada (<code>private</code>)	variable

A partir de la **tabla** que has confeccionado con el **nombre**, el **tipo**, la **visibilidad** y la **mutabilidad** para cada atributo, escribe la **clase Java que representa a un vehículo**.

TEMA 5: DESARROLLO DE CLASES

A partir de la tabla de atributos es fácil escribir el código de la clase Java:

```
public class Vehiculo {  
  
    // Atributos de objeto constantes (representan características "inmutables" del vehículo)  
    private final double capacidadDeposito;          // Capacidad del depósito de combustible del vehículo (en litros)  
    private final String matricula;                  // Matrícula del vehículo  
    private final LocalDate fechaMatriculacion;      // Fecha de matriculación del vehículo  
  
    // Atributos de objeto variables (representan el estado del vehículo en un instante dado)  
    private boolean estadoMotor;                    // Estado del motor del vehículo (arrancado o apagado)  
    private double kilometrosTotales;              // Distancia total recorrida desde que fabricó el vehículo (en kilómetros)  
    private double kilometrosParciales;             // Distancia recorrida desde que el vehículo se arrancó por última vez (en kilómetros)  
}
```

Dado que cada vez hay más variedad de atributos, es bueno que los vayas clasificando por grupos para facilitar la legibilidad del código.



Reflexiona

Desde el comienzo de este curso, siempre hemos afirmado que el convenio más estándar de nombrado de identificadores en Java indica que aquellos elementos que sean de tipo constante (`final`) deberían seguir la nomenclatura de usar siempre mayúsculas y guión bajo para separar cada palabra.

Sin embargo, en el ejemplo anterior te has encontrado con atributos constantes que hemos declarado usando el formato "[lower camel case](#)" en lugar de las mayúsculas. Es habitual que **cuando se trate de atributos de objeto** (no de variables locales, sino de atributos de objeto) se use este modelo para distinguirlo de las constantes que sean atributos de clase, como verás más adelante.



[Emoji One \(CC BY-SA\)](#)

4.5 MODIFICADORES DE CONTENIDO: ATRIBUTOS CONSTANTES (II)

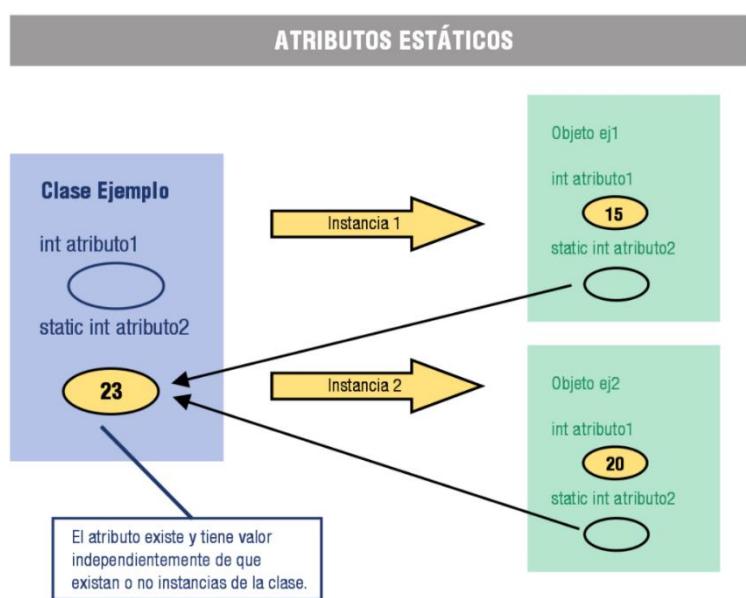
Cada vez que se produce una instancia de una clase (es decir, cada vez que se crea un objeto de esa clase), se desencadenan una serie de procesos (construcción del objeto) que dan lugar a la reserva en memoria de un espacio físico que alojará al objeto creado (los bytes que ocupan sus atributos) así como a la asignación de valores iniciales a esos atributos. De esta manera **cada objeto tendrá sus propios miembros** a imagen y semejanza de la plantilla propuesta por la clase.

En el caso de nuestra clase **Vehiculo**, cada vez que se instancie un objeto vehículo se ejecutará un mecanismo automático (construcción del objeto) que reservará espacio para cada uno de sus atributos (ocho bytes para el la capacidad del depósito, que es de tipo **double**; un byte para el estado del motor, que es de tipo **boolean**, etc.).

Supongamos ahora que nos ha pedido añadir un nuevo atributo a la clase que representa el número de vehículos que han sido creados hasta el momento. Podríamos llamarlo, por ejemplo, **vehiculosCreados** y podría ser de tipo entero (por ejemplo, **short**). Ese atributo se incrementará en uno cada vez que se construya un nuevo vehículo. Si te fijas, ese valor será siempre el mismo para cualquier vehículo, pues en realidad no es una característica específica de cada vehículo en particular, sino una característica de todos los vehículos en general. Es decir, podríamos hablar de una característica de la propia clase más que de una característica de cada objeto. Este tipo de atributos son conocidos con el nombre de **atributos estáticos** o **atributos de clase**, o **variables de clase**.

TEMA 5: DESARROLLO DE CLASES

Un atributo estático o de clase es común (el mismo) para todos los objetos de una misma clase. Podría decirse que la existencia del atributo no depende de la existencia de ningún objeto en particular, sino de la propia clase, y por tanto sólo habrá uno, independientemente del número de objetos que se creen, incluso aunque no exista aún ningún objeto. El atributo será siempre el mismo para todos los objetos y tendrá un valor único independientemente de cada objeto. Aunque no exista ningún objeto de esa clase, ese atributo sí existirá y contendrá un valor (pues se trata de un **atributo de la clase** más que del objeto). Por ejemplo, podríamos estar al inicio de un programa que utiliza vehículos, pero sin haber creado todavía ninguno y aun así el atributo **vehiculosCreados** ya tendría un valor (valor cero). Puedo querer preguntar por el número de vehículos que se han creado, y asegurar que obtendré una respuesta al consultar ese atributo, aunque el número sea cero porque todavía no se ha creado ninguno.



Para indicar que un atributo es estático o de clase Java proporciona el modificador de contenido **static**.

En el caso de nuestro atributo **vehiculosCreados** para la clase **Vehiculo** simplemente habrá que colocar delante el modificador **static**:

```
public class Vehiculo {  
    private static short vehiculosCreados; // Cantidad de vehículos creados hasta el momento  
    ...  
}
```

De esta manera acabamos de añadir un atributo común (el mismo) para todos los objetos de una misma clase. Gracias al modificador **static** no se creará un atributo **vehiculosCreados** de tipo **short** cada vez que se instancie un objeto **Vehiculo**, sino que ese atributo existirá desde antes de la creación de ningún objeto (será



Ejercicio resuelto

Continuando con nuestro proyecto de clase para modelar vehículos, se ha considerado añadir los siguientes nuevos atributos:

- ✓ número de vehículos que se han creado hasta el momento;
- ✓ consumo medio del vehículo (en litros por cada 100 kilómetros);
- ✓ cantidad de kilómetros totales realizados por todos los vehículos;
- ✓ número de vehículos con el motor encendido en el momento actual;
- ✓ nivel del depósito de combustible en el vehículo (en litros).



Gerd Altmann (Pixabay License)

Lleva a cabo un análisis para cada uno de esos nuevos atributos en el que tendrás que decidir:

- 1.- un **identificador** (**nombre** del atributo). Habrá que elegir un nombre representativo de la información que almacena;
- 2.- un **tipo**: entero, real, texto, etc., y la elección de un tipo Java apropiado;
- 3.- una **visibilidad**. Si va a ser visible desde cualquier clase, sólo desde el paquete o sólo desde la clase (modificador de acceso);
- 4.- **mutabilidad**. Si va a ser constante durante toda la vida del objeto (modificador **final**) o podría cambiar con el tiempo.

Esto ya lo has hecho para otros atributos en anteriores ejercicios, pero ahora además tendrás que decidir para cada uno si es más conveniente que sea un **atributo de objeto** (cada vehículo tendrá esa característica con un valor concreto) o un **atributo de clase** (esa característica será común y con el mismo valor para cualquier objeto, es decir, será una característica de la clase y no de cada objeto en particular). Por tanto tendrás que incluir en tu análisis un quinto criterio a considerar: **¿atributo de objeto o atributo de clase?**

TEMA 5: DESARROLLO DE CLASES

un atributo de clase y no de instancia) y será compartido por todos los objetos **Vehículo** que se vayan creando en tu programa.

Realizamos ese análisis para cada uno de los nuevos atributos:

- ✓ **número de vehículos que se han creado** hasta el momento:
 - ↳ nombre: `vehiculosCreados`;
 - ↳ tipo: **número entero** (podríamos usar un `short`);
 - ↳ visibilidad: **privada**, sólo desde dentro de la clase (se usará el modificador de acceso `private`). Todos estos los atributos serán privados según se nos indica en el enunciado;
 - ↳ mutabilidad: no constante, sino **variable** (cada vez que se cree un vehículo irá incrementando);
 - ↳ de **clase**, pues no tiene sentido que esta característica se repita en cada vehículo. Es una característica general de la clase, independientemente de la cantidad de objetos **Vehículo** que existan. De hecho podría no existir ningún vehículo y esta característica seguiría teniendo sentido: tendría valor cero.
- ✓ **Consumo medio** del vehículo:
 - ↳ nombre: `consumoMedio`.
 - ↳ tipo: **número real** (podríamos usar un `double`);
 - ↳ visibilidad: **privada**;
 - ↳ mutabilidad: **constante**. Se trata de una característica específica de cada vehículo que no va a cambiar con el tiempo;
 - ↳ de **objeto**, pues cada vehículo tendrá su propio consumo. No se trata de un valor igual para todos los vehículos. Podrá haber vehículos que tengan el mismo valor para este atributo del mismo modo que pueda haber vehículos con el mismo número de puertas, pero habrá otros vehículos que podrían tener un valor diferente. Se trata de una característica individual de cada objeto.
- ✓ **Cantidad de kilómetros totales realizados por todos los vehículos**:
 - ↳ nombre: `kilometrosTotalesFlota`;
 - ↳ tipo: **número real** (podríamos usar un `double`);
 - ↳ visibilidad: **privada**;
 - ↳ **variable** (cada vez que algún vehículo recorra un trayecto irá incrementando);
 - ↳ de **clase**, pues no tiene sentido que esta característica se repita en cada vehículo. Es una característica general de la clase, independientemente de la cantidad de objetos **Vehículo** que existan. Antes de que se cree ningún vehículo este atributo tendrá el valor 0.0.
- ✓ **Número de vehículos con el motor encendido** en el momento actual:
 - ↳ nombre: `vehiculosArrancados`;
 - ↳ tipo: **número entero** (podríamos usar un `short`);
 - ↳ visibilidad: **privada**;
 - ↳ **variable** (cada vez que se arranque el motor de algún vehículo se incrementará en uno y cada vez que se apague el motor se decrementará en uno);
 - ↳ de **clase**. Es una característica general de la clase, independientemente de la cantidad de objetos **Vehículo** que existan.
- ✓ **Nivel del depósito de combustible** en el vehículo:
 - ↳ nombre: `nivelDeposito`;
 - ↳ tipo: **número real**, pues habrá que almacenar un valor en litros, con posibles fracciones (podríamos usar un `double`);
 - ↳ visibilidad: **privada**;
 - ↳ **variable**. Cada vez que se use el vehículo se producirá un consumo que hará que ese nivel descienda hasta que el depósito quede vacío. Además, cada vez que se reposte combustible en una estación de servicio, esta cantidad podrá incrementarse sin poder superar nunca el tamaño del depósito;
 - ↳ de **objeto**, pues cada vehículo tendrá en cada momento un nivel de combustible. Es una característica de estado de cada vehículo en particular según el uso que se le esté dando.

Una vez realizado ese análisis, confecciona un **tabla para representar estos nuevos atributos** en la que incluyas una **columna adicional** que indique, además de todo lo que hemos visto anteriormente (nombre, tipo, visibilidad, mutabilidad), si el atributo va a ser **de objeto o de clase**.

Mostrar retroalimentación

Aquí tenemos una tabla resumen con los nuevos atributos una vez realizado el anterior análisis en el que hemos valorado, para cada uno, su **identificador**, **tipo**, **visibilidad**, **mutabilidad** (constante o variable) y si debe ser **de objeto o de clase**:

Nuevos atributos de la clase `Vehículo`

Nombre	Tipo	Visibilidad	Mutabilidad	De objeto/de clase
<code>vehiculosCreados</code>	<code>short</code>	privado (<code>private</code>)	variable	de clase (<code>static</code>)
<code>consumoMedio</code>	<code>double</code>	privado (<code>private</code>)	variable	de objeto
<code>kilometrosTotalesFlota</code>	<code>double</code>	privado (<code>private</code>)	variable	de clase (<code>static</code>)
<code>vehiculosArrancados</code>	<code>short</code>	privado (<code>private</code>)	variable	de clase (<code>static</code>)
<code>nivelDeposito</code>	<code>double</code>	privado (<code>private</code>)	variable	de objeto

A partir de la **tabla** anterior escribe la **clase Java que representa a un vehículo** incluyendo todos los atributos que hemos visto hasta el momento (los del ejercicio anterior más los de este ejercicio).

TEMA 5: DESARROLLO DE CLASES

Una vez que tenemos claro cómo van a ser estos nuevos atributos, los añadiríamos a nuestra clase `Vehiculo`:

```
private static short vehiculosCreados;           // Cantidad de vehículos creados hasta el momento
private static double KilometrosTotalesFlota;    // Cantidad de kilómetros recorridos por todos los vehículos creados hasta el momento
private static short vehiculosArrancados;         // Cantidad de vehículos arrancados en este momento

private final double consumoMedio;                // Consumo medio del vehículo (en litros/100km)

private double nivelDeposito;                    // Nivel del depósito de combustible del vehículo (en litros)
```

Estos son los nuevos atributos que debemos integrar junto a todos los demás que ya habíamos definido.

Aunque no es obligatorio, sería recomendable **separar los tipos de atributos indicando cuáles son de clase (estáticos) y cuáles son de objeto** con el fin de facilitar la legibilidad del código. Cuantas más marcas visuales incluyamos y más limpios y organizados seamos a la hora de declarar los atributos de la clase, más fácil lo tendremos en el futuro si tenemos que realizar alguna modificación para encontrar un posible error o para añadir algo nuevo.

Nuestra clase `Vehiculo` podría quedar finalmente con este aspecto:

```
public class Vehiculo {

    // ATRIBUTOS DE CLASE: sólo habrá un atributo (común para todos los objetos) para representar estas características
    // ----

    private static short vehiculosCreados;           // Cantidad de vehículos creados hasta el momento
    private static double KilometrosTotalesFlota;    // Cantidad de kilómetros recorridos por todos los vehículos creados hasta el momento
    private static short vehiculosArrancados;         // Cantidad de vehículos arrancados en este momento

    // ATRIBUTOS DE OBJETO: cada vehículo tendrá su propio valor para representar estas características
    // ----

    // Atributos de objeto constantes (representan características "inmutables" del vehículo)
    private final double capacidadDeposito;          // Capacidad del depósito de combustible del vehículo (en litros)
    private final String matricula;                   // Matrícula del vehículo
    private final LocalDate fechaMatriculacion;       // Fecha de matriculación del vehículo
    private final double consumoMedio;                // Consumo medio del vehículo (en litros/100km)

    // Atributos de objeto variables (representan el estado del vehículo en un instante dado)
    private boolean estadoMotor;                     // Estado del motor del vehículo (arrancado o apagado)
    private double nivelDeposito;                   // Nivel del depósito de combustible del vehículo (en litros)
    private double kilometrosTotales;               // Distancia total recorrida desde que se fabricó el vehículo (en kilómetros)
    private double kilometrosParciales;             // Distancia recorrida desde que el vehículo se arrancó por última vez (en kilómetros)
}
```

4.6 COMBINANDO MODIFICADORES.

Una vez que hemos estudiado los modificadores más importantes, y dado que no todos son excluyentes entre sí, vamos a ver qué sentido o utilidad práctica podrían tener algunas de sus combinaciones. Recordemos la sintaxis completa de la declaración de un atributo teniendo en cuenta la lista de todos los modificadores:

```
[private | protected | public] [static] [final] [transient] [volatile];
```

Quitando **transient** y **volatile**, que en este curso no los vamos a utilizar, podríamos tener a la vez hasta tres modificadores:

- un **modificador de acceso** (según queramos que la visibilidad del atributo sea **de paquete**, **privada**, "protegida" o **pública**),
- un **modificador de contenido final** (si queremos que el atributo no pueda cambiar de valor, es decir, que sea **constante** y no variable),
- un **modificador de contenido static** (si queremos que el atributo sea **de clase** y no de objeto).

Las combinaciones más habituales de modificadores que nos vamos a encontrar son las siguientes:

Combinación de modificadores en los atributos de una clase

Acceso	Mutabilidad	De objeto/de clase	Interpretación
private	final		Atributo de objeto constante y privado.
private			Atributo de objeto variable y privado.
private		static	Atributo de clase variable y privado.
public	final	static	Atributo de clase constante y público.

Salvo el último caso, nosotros ya hemos visto algunos ejemplos de esas combinaciones para nuestra clase **vehículo**:

1. **atributos de objeto constantes y privados**: `capacidadDeposito`, `consumoMedio`, `fechaMatriculacion`, `matricula`;
2. **atributos de objeto variables y privados**: `nivelDeposito`, `estadoMotor`, `kilometrosTotales`, `kilometrosParciales`;
3. **atributos de clase variables y privados**: `vehiculosCreados`, `kilometrosTotalesFlota`, `vehiculosArrancados`.

Si te fijas, la visibilidad de todos esos atributos es **privada**. No nos interesa que desde fuera de la clase se pueda observar o manipular el valor de esos atributos que conforman las características de cada objeto vehículo particular (o, en el caso de los estáticos, características de la clase en general). Lo habitual, como ya hemos reiterado en varias ocasiones, es que los atributos sean privados.

El cuarto ejemplo de combinación de modificadores que se propone es el de **atributos de clase constantes**. ¿Qué utilidad podría tener disponer de atributos de clase constantes? Una vez más, volvemos a nuestro modelo de la clase vehículo: por ejemplo, podría interesarnos tener almacenada cuál es la **máxima y mínima capacidad del depósito de combustible** para un vehículo. Está claro que la capacidad no puede ser negativa, ni tampoco cero. Nos tocaría a nosotros como diseñadores o como programadores decidir cuál va a ser la mínima capacidad que voy a permitir para un depósito de combustible. Y si no se cumple esa condición, no deberíamos dejar que construyera ese objeto de tipo depósito (eso veremos cómo hacerlo en el apartado de los constructores). Y lo mismo podríamos hacer respecto a la máxima capacidad que pueda tener el depósito de combustible de un vehículo. Tendremos que establecer un máximo y no es razonable que ese máximo esté

TEMA 5: DESARROLLO DE CLASES

definido por el máximo número representable por el tipo **double**, que podrían ser trillones de litros (no tendría mucho sentido).

Por tanto, podríamos declarar un par de **atributos constantes** que definan la **máxima y mínima capacidad que puede tener el depósito de combustible para cualquier objeto de la clase Vehículo** que pretendamos instanciar. Dado que es un valor independiente de cualquier objeto (son unos valores límite ya establecidos) parece más que justificado que se trate de atributos de clase y no de atributos de objeto. En consecuencia, ya tenemos nuestros dos primeros ejemplos de atributos constantes y de clase para nuestra clase **Vehículo**:

```
final double MINIMA_CAPACIDAD_DEPOSITO; // Minima capacidad posible de un depósito de combustible (en litros)
final double MAXIMA_CAPACIDAD_DEPOSITO; // Máxima capacidad posible de un depósito de combustible (en litros)
```

Dado que se trata de constantes cuyo valor se conoce ya en tiempo de compilación (no van a depender de ningún objeto), podemos asignarles valor a la vez que se realiza la declaración. Por ejemplo, podríamos considerar que un vehículo no puede tener un depósito con una capacidad inferior a 10.0 litros ni superior a 150.0 litros. En tal caso tendríamos:

```
final double MINIMA_CAPACIDAD_DEPOSITO = 10.0; // Minima capacidad posible de un depósito de combustible (en litros)
final double MAXIMA_CAPACIDAD_DEPOSITO = 150.0; // Máxima capacidad posible de un depósito de combustible (en litros)
```

Respecto a la **visibilidad** de estos atributos, se ha indicado que podría ser **pública**, algo que normalmente evitamos. ¿Por qué en este caso sí podría interesarnos que fueran atributos públicos? Pues porque se trata de **valores límite que son de interés para otro programador que vaya a utilizar esta clase**. De esta manera se podrá tener acceso a esos valores para evitar posibles errores a la hora de definir vehículos que no cumplan con esas restricciones. Por otro lado, al tratarse de atributos constantes, aunque se pueda tener acceso a su valor, no podrán ser modificados, por lo que no se corre ningún riesgo de integridad respecto al estado de los objetos. Es uno de los pocos casos en los que es positivo evitar la privacidad, pues estamos dando información útil al usuario de esta clase y además no pueden ser modificados.

Por tanto, la declaración de estos atributos quedará finalmente como **pública**:

```
public final double MINIMA_CAPACIDAD_DEPOSITO = 10.0; // Minima capacidad posible de un depósito de combustible (litros)
public final double MAXIMA_CAPACIDAD_DEPOSITO = 150.0; // Máxima capacidad posible de un depósito de combustible (litros)
```



Ejercicio Resuelto

Continuando con nuestro proyecto de clase para modelar vehículos, se ha considerado añadir los siguientes nuevos atributos:

- ✓ **mínima capacidad del depósito de combustible para cualquier vehículo** (10.0 litros),
- ✓ **máxima capacidad del depósito de combustible para cualquier vehículo** (150.0 litros),
- ✓ **mínimo consumo medio de combustible para cualquier vehículo** (1.0 litros/100km),
- ✓ **máximo consumo medio de combustible para cualquier vehículo** (18.0 litros/100km).



Tumisu (Pixabay License)

Lleva a cabo un análisis para cada uno de esos nuevos atributos en el que tendrás que decidir:

- 1.- un **identificador** (**nombre** del atributo). Habrá que elegir un nombre representativo de la información que almacena;
- 2.- un **tipo**: entero, real, texto, etc. y la elección de un tipo Java apropiado;
- 3.- una **visibilidad**. Si va a ser visible desde cualquier clase, sólo desde el paquete, sólo desde la clase (modificador de acceso);
- 4.- **mutabilidad**. Si va a ser constante durante toda la vida del objeto (modificador **final**) o podría cambiar con el tiempo;
- 5.- si se trata de un **atributo de objeto o de clase**.

TEMA 5: DESARROLLO DE CLASES

Realizamos ese análisis para cada uno de los nuevos atributos:

- ✓ **minima capacidad del depósito de combustible para cualquier vehículo:**
 - ↳ nombre: MINIMA_CAPACIDAD_DEPOSITO;
 - ↳ tipo: representa una magnitud de litros o fracciones de litro. Lo apropiado es un **número real** (`double`);
 - ↳ visibilidad: **pública**, accesible desde cualquier otra clase (se usará el modificador de acceso `public`). Es una constante de interés para ser observada desde fuera por el programador que utilice la clase, pues define restricciones que no deben saltarse;
 - ↳ mutabilidad: **constante**, de hecho ya se nos proporciona el valor que va a tener siempre, **10.0**;
 - ↳ de clase, pues no tiene sentido que esta característica se repita en cada vehículo. Es una característica general de la clase, independientemente de la cantidad de objetos `Vehiculo` que existan. De hecho podría no existir ningún vehículo y esta característica seguiría teniendo sentido: tendría el valor **10.0**, pues define un límite inferior que no se puede sobrepasar;
- ✓ **máxima capacidad del depósito de combustible para cualquier vehículo:**
 - ↳ nombre: MAXIMA_CAPACIDAD_DEPOSITO;
 - ↳ tipo: **número real** (`double`);
 - ↳ visibilidad: **pública**;
 - ↳ mutabilidad: **constante**, con valor **150.0**;
 - ↳ de clase;
- ✓ **minimo consumo medio de combustible para cualquier vehículo:**
 - ↳ nombre: MINIMO_CONSUMO_MEDIO;
 - ↳ tipo: representa una magnitud de litros por cada 100 kilómetros. Lo apropiado es un **número real** (`double`);
 - ↳ visibilidad: **pública**;
 - ↳ mutabilidad: **constante**, con valor **1.0**;
 - ↳ de clase;
- ✓ **máximo consumo medio de combustible para cualquier vehículo:**
 - ↳ nombre: MAXIMO_CONSUMO_MEDIO;
 - ↳ tipo: **número real** (podríamos usar un `double`);
 - ↳ visibilidad: **pública**;
 - ↳ mutabilidad: **constante**, con valor **18.0**;
 - ↳ de clase.

Una vez realizado ese análisis, confecciona una **tabla para representar estos nuevos atributos** con las cinco columnas descriptivas que ya hemos realizado en otras ocasiones (nombre, tipo, visibilidad, mutabilidad, clase/objeto)

Nuevos atributos de la clase `Vehiculo`

Nombre	Tipo	Visibilidad	Mutabilidad	De objeto/de clase
MINIMA_CAPACIDAD_DEPOSITO	double	pública (<code>public</code>)	constante (<code>final</code>)	de clase (<code>static</code>)
MAXIMA_CAPACIDAD_DEPOSITO	double	pública (<code>public</code>)	constante (<code>final</code>)	de clase (<code>static</code>)
MINIMO_CONSUMO_MEDIO	double	pública (<code>public</code>)	constante (<code>final</code>)	de clase (<code>static</code>)
MAXIMO_CONSUMO_MEDIO	double	pública (<code>public</code>)	constante (<code>final</code>)	de clase (<code>static</code>)

A partir de la tabla anterior, escribe cómo quedaría que la clase Java `Vehiculo` con sus nuevos atributos.

Una vez que tenemos claro cómo van a ser estos nuevos atributos, los añadiríamos a nuestra clase `Vehiculo`:

```
public final static double MINIMA_CAPACIDAD_DEPOSITO = 10.0; // Minima capacidad del depósito de combustible para cualquier vehículo (en litros)
public final static double MAXIMA_CAPACIDAD_DEPOSITO = 150.0; // Máxima capacidad del depósito de combustible para cualquier vehículo (en litros)
public final static double MINIMO_CONSUMO_MEDIO = 1.0; // Minimo consumo medio de combustible para cualquier vehículo (en litros/100km)
public final static double MAXIMO_CONSUMO_MEDIO = 18.0; // Máximo consumo medio de combustible para cualquier vehículo (en litros/100km)
```

Estos son los nuevos atributos que debemos integrar junto a todos los demás que ya habíamos definido en ejercicios anteriores.

La clase `Vehiculo` con los nuevos atributos de clase constantes quedaría así:

TEMA 5: DESARROLLO DE CLASES

```
public class Vehiculo {  
  
    // ATRIBUTOS DE CLASE: sólo habrá un atributo (común para todos los objetos) para representar estas características  
    // -----  
  
    // Atributos de clase constantes (representan características "inmutables" de la clase como por ejemplo restricciones o valores informativos)  
    public final static double MINIMA_CAPACIDAD_DEPOSITO = 10.0; // Minima capacidad del depósito de combustible para cualquier vehículo (en litros)  
    public final static double MAXIMA_CAPACIDAD_DEPOSITO = 150.0; // Máxima capacidad del depósito de combustible para cualquier vehículo (en litros)  
    public final static double MINIMO_CONSUMO_MEDIO = 1.0; // Mínimo consumo medio de combustible para cualquier vehículo (en litros/100km)  
    public final static double MAXIMO_CONSUMO_MEDIO = 18.0; // Máximo consumo medio de combustible para cualquier vehículo (en litros/100km)  
  
    // Atributos de clase variables (representan información de la clase que es común e independiente de cualquier objeto Vehiculo en particular)  
    private static short vehiculosCreados; // Cantidad de vehículos creados hasta el momento  
    private static double kilometrosTotalesFlota; // Cantidad de kilómetros recorridos por todos los vehículos creados hasta el momento  
    private static short vehiculosArrancados; // Cantidad de vehículos arrancados en este momento  
  
    // ATRIBUTOS DE OBJETO: cada vehículo tendrá su propio valor para representar estas características  
    // -----  
  
    // Atributos de objeto constantes (representan características "inmutables" del vehículo)  
    private final double capacidadDeposito; // Capacidad del depósito de combustible del vehículo (en litros)  
    private final String matricula; // Matrícula del vehículo  
    private final LocalDate fechaMatriculacion; // Fecha de matriculación del vehículo  
    private final double consumoMedio; // Consumo medio del vehículo (en litros/100km)  
  
    // Atributos de objeto variables (representan el estado del vehículo en un instante dado)  
    private boolean estadoMotor; // Estado del motor del vehículo (arrancado o apagado)  
    private double nivelDeposito; // Nivel del depósito de combustible del vehículo (en litros)  
    private double kilometrosTotales; // Distancia total recorrida desde que fabricó el vehículo (en kilómetros)  
    private double kilometrosParciales; // Distancia recorrida desde que el vehículo se arrancó por última vez (en kilómetros)  
}
```



Reflexiona

En el ejemplo anterior acabas de ver que para algunos atributos `final` se ha usado el formato de mayúsculas y que para otros se ha optado por formato "*lower camel case*".

Como ya comentamos en un ejercicio anterior, es habitual que cuando se trate de atributos constantes (`final`) se utilice para su nombrado o identificación:

- ✓ el formato "*lower camel case*" cuando se trate de **atributos de objeto** (tanto si son **variables** como si son **constants**). Es decir, que sería una excepción a la regla habitual de que todas las constantes deben identificarse usando mayúsculas;
- ✓ el formato de **mayúsculas con guiones**, cuando se trate de **atributos de clase constantes**, que además normalmente serán atributos públicos, pues contendrán información sobre configuraciones, opciones, restricciones, etc., que es interesante que sea visible desde fuera de la clase.

En el caso de los **atributos de clase variables** está claro que se seguiría el formato "*lower camel case*" habitual, pues se trata de atributos variables.

Una vez planteada esta observación es posible que te plantees: ¿debo seguir esta excepción en el nombrado o debo ser estricto y usar las mayúsculas para cualquier atributo que sea final? **Nosotros recomendamos que para los atributos `final` que sean de objeto sigas la excepción (uso "*lower camel case*")**, pues es la decisión que han tomado la mayoría de los programadores de Java y como están implementadas las clases de la API de Java.

FGHFGHFHFG



Para saber más

Acabamos de afirmar que las clases de la **API** usan la notación "*lower camel case*" para el caso de **atributos de objeto constantes**. Aún es pronto para navegar por el código fuente de clases profesionales escritas en Java, pues contienen muchísimo código que todavía no entiendes. Pero por si tienes curiosidad, hemos escogido un par de ejemplos y los hemos aislado y simplificado para que puedas observar cómo en efecto no se usan las mayúsculas para los atributos de objeto (sean constantes o no).

Primer ejemplo: implementación de la clase `LocalTime` (`java.time.LocalTime`), entre las líneas 217 y 229:

```
private final byte hour;
private final byte minute;
private final byte second;
private final int nano;
```

Aquí puedes ver cómo los desarrolladores de esta clase han optado por no usar las mayúsculas para estos **atributos de objeto constantes** (`final`) privados (nadie fuera de la clase va a tener acceso a ellos) con los cuales se representa una hora (hora, minuto, segundo, nanosegundo).

Puedes observar el código fuente completo en este enlace: [Código fuente de la clase LocalTime](#).

Segundo ejemplo: implementación de la clase `Integer` (`java.lang.Integer`), en la línea 840:

```
private final int value;
```

Aquí también puedes ver cómo no se están utilizando las mayúsculas para un atributo `final` que es privado (ningún programador que use la clase lo verá) y representa parte del estado del objeto.

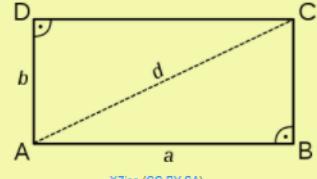
Puedes observar el código fuente completo en este enlace: [Código fuente de la clase Integer](#).



Ejercicio Resuelto

Deseamos implementar una clase que represente a un rectángulo en el plano (clase `Rectangulo`). Se ha pensado en almacenar la siguiente información para cada objeto de tipo rectángulo:

- ✓ la **ubicación** del rectángulo en el plano,
- ✓ el **color de relleno** del rectángulo,
- ✓ el **nombre** del rectángulo,
- ✓ la **cantidad de rectángulos** que han sido creados hasta el momento.



XZise (CC BY-SA)

Teniendo en cuenta esas decisiones de diseño, lleva a cabo un análisis para decidir cuántos atributos necesitarías, qué identificadores les asignarías, de qué tipo serían, y qué otras características tendrían (visibilidad, mutabilidad, de objeto o de clase) confeccionando finalmente una **tabla de atributos** con las cinco columnas descriptivas (nombre, tipo, visibilidad, mutabilidad, clase/objeto).

TEMA 5: DESARROLLO DE CLASES

Llevamos a cabo el análisis para decidir cuáles serían los atributos más apropiados en función de la información que deseamos representar para cada rectángulo:

- ✓ **ubicación:** con **dos puntos, vértice inferior izquierdo (x_1, y_1) y vértice superior derecho (x_2, y_2)**, se puede definir perfectamente la ubicación de un rectángulo en el plano. Por tanto sería suficiente con **cuatro atributos de tipo real** para poder almacenar la ubicación. Podríamos definir entonces cuatro atributos de tipo `double` llamados `x1, y1, x2, y2`,
- ✓ para el **color** podría usarse una **cadena de caracteres (String)** con el nombre del color,
- ✓ para el **nombre** podría usarse también una **cadena de caracteres (string)**,
- ✓ respecto a la **cantidad de rectángulos construidos** hasta el momento, nos vendría bien un **número entero (int)**. Por otro lado se trata de un atributo candidato a ser **estático** más que de objeto, pues con que se disponga de ese valor una sola vez para todos los posibles objetos creados es más que suficiente.

En cuanto la **mutabilidad** o inmutabilidad de estos atributos, en el caso de la cantidad de rectángulos construidos es obvio que debe ser variable (cada vez que se construya un nuevo rectángulo este atributo deberá incrementar su valor en uno). Respecto al resto de atributos, dado que en el enunciado no se ha indicado nada concreto, podríamos suponer que todos podrían cambiar de valor, por lo que no serán constantes sino **variables**. Por ejemplo, en el caso de la ubicación podríamos asumir que es posible que en el futuro pudiéramos querer desplazar el rectángulo a otra posición. En tal caso está claro que se trataría de atributos mutables o variables. Sin embargo, si se hubiera decidido que una vez que se ha creado un rectángulo no se puede modificar su posición, entonces deberían ser declarados como inmutables o constantes. Será una decisión de diseño dependiendo de la utilidad que vayan a tener esos objetos rectángulo en nuestro programa.

Sobre la **visibilidad** de estos atributos, seguiremos la norma habitual: **privados** salvo que se indique lo contrario. Tan solo en el caso de los atributos de clase constantes nos plantearíamos hacerlos públicos, como ya hemos visto en ejemplos anteriores en los que se almacenan datos sobre restricciones o informativos.

Respecto al debate de cuáles deberían ser **de objeto** y cuáles **de clase** (estáticos), parece claro que el único que debería ser de clase es el que contiene la cantidad de rectángulos construidos hasta el momento, que no tiene sentido que sea un valor que acompañe a cada rectángulo, sino que se trata de un valor general de la clase `Rectangulo`.

A partir de este análisis, ya podemos construir nuestra tabla de atributos:

Atributos de la clase Rectangulo

Nombre	Tipo	Visibilidad	Mutabilidad	De objeto/de clase
x1	double	privada (private)	variable	de objeto
y1	double	privada (private)	variable	de objeto
x2	double	privada (private)	variable	de objeto
y2	double	privada (private)	variable	de objeto
color	String	privada (private)	variable	de objeto
nombre	String	privada (private)	variable	de objeto
cantidadRectangulos	int	privada (private)	variable	de clase (static)

A partir de la tabla de atributos que has confeccionado, escribe cómo podría quedar la clase `Rectangulo` en lenguaje Java.

```
public class Rectangulo {  
    // ATRIBUTOS DE CLASE:  
    // Sólo habrá un atributo (común para todos los objetos) para representar estas características  
    // -----  
  
    // Atributos de clase variables  
    // (representan información de la clase que es común e independiente de cualquier objeto Rectangulo en particular)  
    private static int cantidadRectangulos; // Cantidad de rectángulos creados hasta el momento  
  
    // ATRIBUTOS DE OBJETO:  
    // Cada rectángulo tendrá su propio valor para representar estas características  
    // -----  
  
    // Atributos de objeto variables (representan el estado del rectángulo en un instante dado)  
    private double x1; // Coordenada x del vértice inferior izquierdo  
    private double y1; // Coordenada y del vértice inferior izquierdo  
    private double x2; // Coordenada x del vértice superior derecho  
    private double y2; // Coordenada y del vértice superior derecho  
    private String color; // Color de relleno  
    private String nombre; // Nombre  
}
```

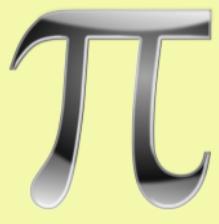
Una vez más, fíjate en la importancia de documentar al máximo todos los detalles acerca de los atributos de la clase así como de asignarles identificadores que nos lo pongan fácil a la hora de entender lo que representan o se almacena en ellos. Será la mejor forma de facilitar la legibilidad del código.



Ejercicio Resuelto

Del mismo modo que en el ejercicio anterior se ha implementado una clase que representara a un rectángulo en el plano, se nos ha solicitado ahora representar una nueva figura geométrica plana: un **círculo**. En este caso los diseñadores nos han indicado que necesitan almacenar la siguiente información para representar objetos de este tipo:

- ✓ al **ubicación del centro** del círculo;
- ✓ el **radio** del círculo;
- ✓ el **color de relleno** del círculo, que podrá ir cambiando a lo largo de la vida del objeto;
- ✓ el **nombre** del círculo, que una vez asignado al construirse, no podrá cambiar;
- ✓ la **cantidad de círculos** que han sido creados hasta el momento;
- ✓ el valor de la **constante pi** (π), con una buena cantidad de cifras significativas para evitar escribir el literal 3,14159265... una y otra vez cada ocasión en la que haya que realizar cálculos de áreas y longitudes.



[OpenClipart-Vectors \(Pixabay License\)](#)

También se nos ha indicado el que una vez creado un círculo, aunque podría desplazarse a otra ubicación, **su tamaño no podrá ser modificado**.

A partir de esas decisiones de diseño:

- 1.- lleva a cabo el **análisis** para decidir cuántos **atributos** necesitarías, qué identificadores les asignarías, de qué tipo serían, y qué otras características tendrían (visibilidad, mutabilidad, de objeto o de clase);
- 2.- construye una **tabla de atributos** con las cinco columnas descriptivas (nombre, tipo, visibilidad, mutabilidad, clase/objeto);
- 3.- **implementa en Java** la clase **Círculo** con los atributos que se hayan decidido.

El análisis para decidir los atributos necesarios es muy similar al que se ha tenido que realizar para la clase **Rectángulo**. Los tipos y los identificadores son muy parecidos. Las principales diferencias serían:

- ✓ para la **ubicación** tan solo serán necesarias las coordenadas de un punto (el **centro del círculo**): **dos atributos** (**x, y**) de tipo **real** (**double**). Si deseamos que esta ubicación pueda cambiar a lo largo de la vida del círculo, tal y como se nos ha indicado en el enunciado, entonces esos atributos serán mutables o **variables**. Si no fuera así, entonces serían constantes o inmutables;
- ✓ dado que nos han indicado que el **tamaño del círculo no puede cambiar**, está claro que su **radio** sí ha de ser inmutable o **constante** (**final**);
- ✓ el **nombre** del círculo no podrá cambiar una vez que se le asigne un valor. Será por tanto **constante** (**final**);
- ✓ el **valor de π** es obviamente una **constante** (**final**), y además no tiene sentido que se almacene su valor en cada objeto de tipo **Círculo**, pues siempre será el mismo y se trata más bien de una herramienta para cualquier objeto de la clase que de una información específica de objeto. Será por tanto un atributo **estático o de clase** (**static**). Además, dado que se trata de cierta información que puede resultar útil fuera de la clase y que no se corre peligro al hacerla pública (es constante), podría declararse como un atributo **público** (**public**).

A partir de esas conclusiones podemos construir nuestra tabla de atributos:

Atributos de la clase Círculo

Nombre	Tipo	Visibilidad	Mutabilidad	De objeto/de clase
x	double	privada (private)	variable	de objeto
y	double	privada (private)	variable	de objeto
radio	double	privada (private)	constante (final)	de objeto
color	String	privada (private)	variable	de objeto
nombre	String	privada (private)	constante (final)	de objeto
cantidadCírculos	int	privada (private)	variable	de clase (static)
PI	double	pública (public)	constante (final)	de clase (static)

TEMA 5: DESARROLLO DE CLASES

```
public class Circulo {  
    // ATRIBUTOS DE CLASE: sólo habrá un atributo (común para todos los objetos) para representar estas características  
    // -----  
  
    // Atributos de clase constantes  
    // (representan características "inmutables" de la clase como por ejemplo restricciones o valores informativos)  
    public final static double PI = 3.141592654 // Valor de la constante pi  
  
    // Atributos de clase variables  
    // (representan información de la clase que es común e independiente de cualquier objeto Circulo en particular)  
    private static int cantidadCirculos; // Cantidad de círculos creados hasta el momento  
  
    // ATRIBUTOS DE OBJETO: cada círculo tendrá su propio valor para representar estas características  
    // -----  
  
    // Atributos de objeto constantes (representan características "inmutables" del círculo)  
    private final double radio; // Radio del círculo  
    private final String nombre; // Nombre del círculo  
  
    // Atributos de objeto variables (representan el estado del círculo en un instante dado)  
    private double x; // Coordenada x del centro del círculo  
    private double y; // Coordenada y del centro del círculo  
    private String color; // Color de relleno del círculo  
}
```

4.7 OBJETOS INMUTABLES.

En algunas ocasiones te has encontrado con clases de la API de Java que se dice que son "inmutables", es decir, que los objetos instancia de esas clases no podían ser modificados. En realidad, sería más apropiado hablar de **objetos inmutables**. Algunos ejemplos podrían ser:

- la clase **String**, que almacena cadenas de caracteres,
- todas las clases **envoltorio ("wrapper")** de los tipos primitivos (**Byte, Short, Integer, Long, String, Float, Double, Boolean, Character**),
- las clases de manejo del tiempo **LocalDate, LocalTime y LocalDateTime**.

En todos esos casos cada vez que se crea un objeto instancia de alguna de esas clases, el objeto no puede ser modificado. Si se lleva a cabo alguna operación con ellos que da lugar a alguna modificación se obtiene un nuevo objeto con esa modificación, pero el objeto original permanece inalterado.

¿Cómo podemos nosotros implementar una clase que sea inmutable?

Para ello es necesario que absolutamente **todos sus atributos sean inmutables**, es decir, **constantes**. En el caso del lenguaje Java, tendrás que declarar todos sus atributos como **final**.

Además, la propia clase deberá ser declarada como **final**. El significado de que una clase sea **final** lo veremos en la próxima unidad cuando estudiemos la herencia. Baste decir por el momento que si una clase es **final**, no se puede implementar otra clase que sea descendiente de ella (no permite la herencia).

Por tanto, la estructura básica de una clase para que sea inmutable será:

```
public final class <NombreClase> {  
    private final <tipoAtributo1> nombreAtributo1;  
    private final <tipoAtributo2> nombreAtributo2;  
    private final <tipoAtributo3> nombreAtributo3;  
    ...
```



Ejercicio Resuelto

Además de los tradicionales dados de seis caras, existen otros dados con un número diferente de caras (cuatro, ocho, doce, etc.).



[Clicker-Free-Vector-Images \(Pixabay License\)](#)

Implementa una clase `Dado` que pueda representar a dados cuyo número de caras no sea necesariamente seis. ¿Sería razonable plantearse los objetos de esta clase como **objetos inmutables**?

[Mostrar retroalimentación](#)

Puesto que no se nos ha dado más información, en principio el único dato que tenemos sobre un dado sería el número de caras que tiene. Podríamos entonces representar un objeto dado con un único **atributo de tipo entero** que contuviera el **número de caras** del dado. Es razonable pensar que una vez creado un dado no va a cambiar su número de caras y como no va a tener más atributos (al menos por el momento) parece razonable pensar que **los objetos de tipo dado sean objetos inmutables**. En tal caso declararemos la clase como todos sus atributos de objeto **como final**.

```
public final class Dado {
    private final byte numeroCaras;
}
```



Reflexiona

Hemos definido un **objeto inmutable** como aquél cuyos **atributos no pueden cambiar de valor una vez que el objeto es instanciado**.

¿Eso significa que si no incluyes el modificador `final` en la declaración de cada atributo los objetos no serán inmutables?

Bueno, en teoría sus atributos son variables, pero si esos atributos son privados y ningún método los modifica, en la práctica también se trataría de una clase cuyos objetos son inmutables, pues sus atributos no van a cambiar nunca de valor.

Ahora bien, **si tienes claro que un atributo nunca va a cambiar una vez asignado un valor inicial, lo recomendable es que lo declares como final**. No es obligatorio pero sería lo más correcto y evitará posibles errores en el futuro que podrían traerte más de un dolor de cabeza. Si una vez que has añadido el modificador `final` a los atributos que van a ser constantes observas que todos los atributos son `final`, entonces podrás afirmar sin lugar a dudas que estás implementando una clase cuyos objetos serán inmutables.

Recuerda que hemos decidido adoptar la "excepción" de nombrado "*lower camel case*" para los **atributos final de objeto** aunque estrictamente deberían nombrarse usando las mayúsculas.

5. MÉTODOS.

Como ya has visto anteriormente, los **métodos** son las herramientas que nos sirven para definir el comportamiento de un objeto en sus interacciones con otros objetos. Forman parte de la estructura interna del objeto junto con los atributos.

En el proceso de declaración de una clase ya has visto cómo escribir la cabecera de la misma y cómo especificar sus atributos dentro del cuerpo de la clase. Tan solo falta ya declarar los métodos, que estarán también en el interior del cuerpo de la clase junto con los atributos. ¿En qué parte de la clase escribimos los métodos? **Los métodos suelen declararse después de los atributos.**

Aunque atributos y métodos pueden aparecer mezclados por todo el interior del cuerpo de la clase es aconsejable no hacerlo para mejorar la **claridad** y la **legibilidad** del código. De ese modo, cuando echemos un vistazo rápido al contenido de una clase, podremos ver rápidamente los atributos al principio (normalmente ocuparán menos líneas de código y serán fáciles de reconocer) y cada uno de los métodos inmediatamente después. Cada método puede ocupar un número de líneas de código más o menos grande en función de la complejidad del proceso que pretenda implementar.

Los métodos representan la **interfaz** de una clase. Son la forma que tienen otros objetos de comunicarse con un objeto determinado solicitándole cierta información o pidiéndole que lleve a cabo una determinada acción. Este modo de programar, como ya has visto en unidades anteriores, facilita mucho la tarea al desarrollador de aplicaciones, pues nos permite al programar abstraernos del contenido de las clases haciendo uso únicamente del interfaz (métodos). Veamos un ejemplo al respecto con nuestra clase **Vehiculo**:

- sabemos que los objetos instancia de la la clase **Vehiculo** contienen un atributo privado **boolean** que representa si el motor del vehículo está o no arrancado (**atributo estadoMotor**);
- supongamos que un programa externo a esa clase que utilice objetos **Vehiculo** desea arrancar el motor de un vehículo. Ese programa no tiene información alguna de que existe ese atributo **estadoMotor**. Tan solo quiere arrancar el motor de un objeto vehículo, llamado por ejemplo **v1**, que está apagado. Este tipo de programas diremos que son programas "cliente" o código "cliente" de la clase **Vehiculo**, pues se sirven de ella, la utilizan, pero no saben cómo está implementada por dentro;
- para poder **arrancar el vehículo**, desde el programa habrá que informar a ese vehículo **v1** (objeto) de que queremos arrancar el motor. Para hacerlo tendremos que **invocar a un método** que permita arrancar el motor (**método arrancar**). Esa comunicación es también conocida como "paso de mensajes", pues se está enviando un mensaje desde el código cliente del objeto hacia el propio objeto mandándole en este caso el siguiente mensaje: *"Oye vehículo v1, quiero que arranques el motor"*;
- ese método **arrancar** se encargará de llevar a cabo todas las acciones necesarias para **cambiar el estado del vehículo** y hacer que pase a modo "arrancado" con todas las consecuencias internas que ello tenga para sus atributos. Entre otras cosas tendrá que modificar el valor del atributo **estadoMotor** para pasarlo al valor **true**, y quizá consumir algo de combustible, pero eso no lo sabrá el programa (código cliente) que usa el objeto de tipo vehículo;
- para el código cliente el objeto vehículo es una "caja negra" que contiene atributos privados dentro, cuyos nombres y valores desconoce y sólo se puede comunicar con el objeto a través de métodos como por ejemplo **arrancar**. Ya se encargará ese método de llevar a cabo internamente los procesos oportunos para que el motor del vehículo pase a estar arrancado y el valor de sus atributos internos sea coherente con ese nuevo estado del vehículo. Desde fuera de la clase sólo sabremos que hemos arrancado el vehículo **v1** porque le hemos dado la orden de arrancar y no se ha producido ningún fallo.

El código cliente de un objeto no necesita conocer cómo está implementado un método por dentro, tan solo debe saber que existe y cómo debe usarlo.

5.1 DECLARACIÓN DE UN MÉTODO.

La definición de un método se compone de dos partes:

- **cabecera del método**, que contiene el **nombre** del método junto con el **tipo devuelto**, (**void** en caso de que no devuelva nada), un conjunto de posibles modificadores y una lista de parámetros o argumentos entre paréntesis (si no hay argumentos, se pondrá un paréntesis vacío);
- **cuerpo del método**, que contiene las sentencias que implementan el comportamiento del método.

Los **elementos mínimos** que deben aparecer en la declaración de un método son:

- el tipo devuelto por el método;
- el nombre del método;
- los paréntesis;
- el cuerpo del método entre llaves: {}.

Veamos un par de ejemplos de posibles métodos para nuestra clase **Vehículo**, que venimos modelando desde el comienzo de la unidad.

Primer ejemplo: método `getNivelDepósito`

Este método podría servirnos para **obtener la cantidad de combustible que queda en el depósito del vehículo**. En este caso la cabecera debería contener como mínimo:

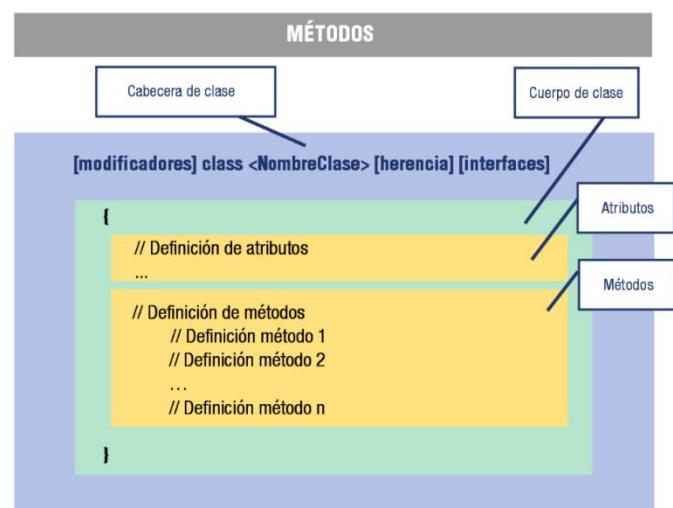
- el **tipo devuelto** por el método: **double**;
- el **nombre** del método: **getNivelDepósito**;
- no recibe **ningún parámetro**: aparece una lista vacía entre paréntesis: () .

Esos tres elementos conformarían la cabecera del método, que quedaría más o menos así:

```
double getNivelDepósito () {
    // Cuerpo del método
    ...
}
```

El **cuerpo** del método sería el código que habría encerrado entre llaves, que aquí representamos por un comentario y unos puntos suspensivos dentro de {}. Ya lo "rellenaremos" cuando veamos el cuerpo del método.

Dentro del cuerpo del método podrás encontrar declaraciones de variables, sentencias y todo tipo de estructuras de control (bucles, condiciones, etc.) que has estudiado en los apartados anteriores. Podemos imaginar el cuerpo de un método como un "mini programa".



TEMA 5: DESARROLLO DE CLASES

Segundo ejemplo: método arrancar

Un método con ese nombre podría servirnos para arrancar el motor del vehículo. En este caso la cabecera estaría formada por:

- el **tipo devuelto** por el método: **void**. (no devolvería nada, simplemente llevaría a cabo la acción de arrancar el motor del vehículo);
- el **nombre** del método: **arrancar**;
- **ningún parámetro**: aparece una lista vacía entre paréntesis: **()**.

La cabecera de este segundo método quedaría entonces más o menos así:

```
void arrancar () {  
    // Cuerpo del método  
    ...  
}
```

Ahora bien, sabemos que la declaración de un método puede incluir algunos elementos más. Vamos a estudiar con detalle cada uno de ellos.

5.2 CABECERA DE UN MÉTODO.

La declaración de un método puede incluir los siguientes elementos:

1. **modificadores** (como por ejemplo los modificadores de acceso ya vistos: **public**, **protected** o **private**). No es obligatorio incluir modificadores en la declaración;
2. el **tipo devuelto** (o tipo de retorno), que consiste en el tipo de dato (primitivo o referencia) que el método devuelve tras ser ejecutado. Si eliges **void** como tipo devuelto, el método no devolverá ningún valor;
3. el **nombre del método**, aplicándose el mismo convenio que para los atributos y las variables (primera palabra con todas las letras en minúscula y el resto de palabras con la primera letra en mayúscula);
4. una **lista de parámetros** separados por comas y entre paréntesis, donde cada parámetro debe ir precedido por su tipo. Si el método no tiene parámetros la lista estará vacía y únicamente aparecerán los paréntesis;
5. una **lista de excepciones** que el método puede lanzar. Se utiliza la palabra reservada **throws** seguida de una lista de nombres de excepciones separadas por comas. No es obligatorio que un método incluya una lista de excepciones, aunque muchas veces será conveniente. En unidades anteriores ya has trabajado con el concepto de excepción y más adelante volverás a hacer uso de ellas;
6. el **cuerpo del método**, encerrado entre llaves. El cuerpo contendrá el código del método (una lista sentencias y estructuras de control en lenguaje Java) así como la posible declaración de variables locales.

La sintaxis general de la cabecera de un método podría entonces quedar así:

```
[private | protected | public] [static] [abstract] [final] [native] [synchronized]  
<tipo> <nombreMetodo> ( [<lista_parametros>] ) [throws <lista_excepciones>]
```

TEMA 5: DESARROLLO DE CLASES

Como sucede con todos los identificadores en Java (variables, clases, objetos, métodos, etc.), puede usarse cualquier identificador que cumpla las normas. Ahora bien, para mejorar la legibilidad del código, se ha establecido el siguiente convenio para nombrar los métodos: **utilizar un verbo en minúscula o bien varias palabras comenzando por un verbo en minúscula, seguido por adjetivos, nombres, etc., los cuales sí aparecerán comenzando en mayúsculas.**

Algunos ejemplos de nombres de métodos que siguen este convenio podrían ser: **ejecutar, romper, mover, subir, responder, obtenerX, getX, establecerValor, setValor, estaVacio, isVacio, estaLleno, moverFicha, subirPalanca, responderRapido, girarRuedaIzquierda, abrirPuertaDelantera, cambiarMarcha, getNivelDeposito, repostar, arrancar, parar, recorrerTrayecto, reset**, etc.

En nuestro proyecto de la clase **Vehiculo**, podríamos definir algunos métodos como, por ejemplo:

Ejemplos de nombres de métodos para la clase **Vehiculo**

Possible nombre del método	Descripción
obtenerNivelDeposito O getNivelDeposito	Devuelve la cantidad de combustible que queda en el depósito del vehículo.
arrancar	Arranca el motor del vehículo.
parar	Para el motor del vehículo.
repostar	Introduce una cantidad determinada de combustible en el depósito del vehículo.
estaArrancado O isArrancado	Indica si el vehículo está arrancado o no.
recorrerTrayecto	Recorre un determinado trayecto con el vehículo.

5.3 MODIFICADORES EN LA DECLARACIÓN DE UN MÉTODO.

En la declaración de un método también pueden aparecer modificadores (como en la declaración de la clase o de los atributos). Un método puede tener los siguientes tipos de modificadores:

- **modificadores de acceso.** Son los mismos que en el caso de los atributos (por omisión o de paquete, **public, private y protected**) y tienen el mismo cometido (acceso al método sólo por parte de clases del mismo paquete, o por cualquier parte del programa, o sólo para la propia clase, o también para las subclases, aunque estén en otro paquete);
- **modificadores de contenido.** Son también los mismos que en el caso de los atributos (**static y final**), aunque su significado no es exactamente el mismo;
- **otros modificadores** (no son aplicables a los atributos, sólo a los métodos): **abstract, native, synchronized.**

Un método **static** (método **estático o de clase**) es un método desde cuya implementación no se accede a atributos de objeto. Desde este tipo de métodos sólo se puede acceder a atributos de la clase (estáticos). **Estos métodos pueden ser invocados sin necesidad de tener un objeto de la clase instanciado.**

En Java puedes encontrar una gran cantidad de métodos estáticos en la clase **Math**. De hecho, todos sus métodos son todos estáticos (**Math.abs, Math.sin, Math.cos**, etc.). La llamada a métodos estáticos se hace normalmente usando el nombre de la propia clase y no el de una instancia (objeto), pues se trata realmente de un método de clase.

Un método **final** es un método que no permite ser redefinido o sobreescrito por las clases descendientes de la clase a la que pertenece el método. Volverás a ver este modificador, así como el concepto de redefinición o sobreescritura de métodos cuando estudies en detalle la **herencia**. **En esta unidad no lo vamos a usar.**

TEMA 5: DESARROLLO DE CLASES

El modificador **native** es utilizado para señalar que un método ha sido implementado en código nativo (en un lenguaje que ha sido compilado a lenguaje máquina, como por ejemplo **C** o **C++**). En estos casos simplemente se indica la cabecera del método, pues no tiene cuerpo escrito en Java. **No lo utilizaremos en este curso.**

El modificador **abstract** sirve para indicar que un método es abstracto, esto es, un método sin implementación (no hay cuerpo, solo la declaración terminada en punto y coma, en lugar de las llaves con el cuerpo). Es importante el detalle del punto y coma: un método con un cuerpo vacío, es decir, con llaves de apertura y cierre, pero ninguna sentencia entre ellas **no sería un método abstracto**, simplemente sería un método inútil que no hace nada. La implementación será realizada en las clases descendientes. Un método sólo puede ser declarado como **abstract** si se encuentra dentro de una clase **abstract**. Usarás este modificador en unidades posteriores cuando trabajes con la **herencia**. **En esta unidad haremos uso de él.**

Por último, si un método ha sido declarado como **synchronized**, el entorno de ejecución obligará a que cuando un proceso esté ejecutando ese método, el resto de procesos que tengan que llamar a ese mismo método deberán esperar a que el otro proceso termine. Puede resultar útil si sabes que un determinado método va a poder ser llamado concurrentemente por varios procesos a la vez. **Tampoco lo vamos a utilizar en este curso.**

Dada la cantidad de modificadores que has visto hasta el momento y su posible aplicación en la declaración de clases, atributos o métodos, veamos un resumen de todos y en qué casos pueden aplicarse:

Cuadro de aplicabilidad de los modificadores.

Modificador	Clase	Atributo	Método
Sin modificador (package)	Sí	Sí	Sí
public	Sí	Sí	Sí
private	Sí (*)	Sí	Sí
protected	Sí (*)	Sí	Sí
static		Sí	Sí
final	Sí	Sí	Sí
synchronized			Sí
native			Sí
abstract	Sí		Sí

(*) Sólo aplicable para el caso de clases anidadas o internas.

A modo de resumen, por ahora los únicos modificadores con los que vamos a trabajar serán:

- los de **visibilidad** o **accesibilidad** (sólo **public** y **private**). El modificador **protected** se verá cuando estudiemos la herencia;
- el de **clase**, para definir **métodos estáticos (static)**.

El resto de modificadores los iremos viendo cuando sean necesarios.

5.4 PARÁMETROS DE UN MÉTODO.

La lista de parámetros de un método se coloca tras el nombre del método. Esta lista estará constituida por pares de la forma "**<tipoParametro> <nombreParametro>**". Cada uno de esos pares estará separado por comas y la lista completa estará encerrada entre paréntesis:

```
<tipo> nombreMetodo ( <tipo_1> <nombreParametro_1>, <tipo_2> <nombreParametro_2>, ..., <tipo_n> <nombreParametro_n> )
```

Si se trata de un método que no recibe parámetros, es decir, la lista de parámetros está vacía, tan solo aparecerán los paréntesis, pero éstos deben aparecer siempre:

```
<tipo> nombreMetodo ( )
```

A la hora de declarar un método, debes tener en cuenta lo siguiente:

- puedes incluir cualquier cantidad de parámetros. Se trata de una decisión del programador, pudiendo ser incluso una lista vacía;
- los parámetros podrán ser de cualquier tipo (tipos primitivos, referencias, objetos, arrays, etc.);
- no está permitido que el nombre de una variable local del método coincida con el nombre de un parámetro;
- no puede haber dos parámetros con el mismo nombre. Se produciría ambigüedad;
- si el nombre de algún parámetro coincide con el nombre de un atributo de la clase, éste será "ocultado" por el parámetro. Es decir, al indicar ese nombre en el código del método estarás haciendo referencia al parámetro y no al atributo. Para poder acceder al atributo tendrás que hacer uso del operador de autorreferencia **this**, que verás un poco más adelante;
- en Java el **paso de parámetros es siempre por valor**, excepto en el caso de los tipos referenciados (por ejemplo, los objetos) en cuyo caso se está pasando efectivamente una referencia. La referencia (el objeto en sí mismo) no podrá ser cambiada pero sí elementos de su interior (atributos) a través de sus métodos o por acceso directo si se trata de un miembro público.



Para saber más

Puedes echar un vistazo al artículo general sobre paso de parámetros a métodos en los manuales de Oracle sobre Java:[Passing Information to a Method or a Constructor](#).

Para saber más

Si consultas otras fuentes podrás observar que cuando se habla del paso de parámetros se alude a los **conceptos de paso por valor y paso por referencia**. Debes saber que **en Java el paso de cualquier parámetro se hace siempre por valor**, por tanto no vamos a entrar a explicar esa distinción. Ahora bien, cuando el parámetro no es de tipo primitivo (es una referencia), aunque la referencia no pueda ser cambiada, sí podría cambiarse el contenido (atributos de un objeto o elementos de un array) al que apunta la referencia. En ese sentido se podría intentar simular un paso por referencia. Aquí tienes algunos enlaces que pueden resultarte de interés por si estás interesado en profundizar en el tema:

- [Artículo en la Wikipedia sobre el paso de argumentos](#), donde se explica la diferencia entre el **paso por valor** y el **paso por referencia**.
- [Artículo en Stackoverflow "What's the difference between passing by reference vs. passing by value?"](#)
- [Artículo en la web codingornot.com "¿Cuándo se utiliza «paso por referencia» y cuándo «paso por valor»?"](#)



Ejercicio Resuelto

En nuestro proyecto de diseño e implementación de la clase `Vehículo`, hemos estado pensando en algunos métodos que podrían resultar útiles y por ahora tenemos los siguientes:

Ejemplos de nombres de métodos para la clase `Vehículo`

Possible nombre del método	Descripción
<code>getNivelDeposito</code>	Devuelve la cantidad de combustible que queda en el depósito del vehículo.
<code>arrancar</code>	Arranca el motor del vehículo.
<code>parar</code>	Para el motor del vehículo.
<code>repostar</code>	Introduce una cantidad determinada de combustible en el depósito del vehículo.
<code>isArrancado</code>	Indica si el vehículo está arrancado o no.
<code>recorrerTrayecto</code>	Recorre un determinado trayecto con el vehículo.

Ahora nos falta decidir para cada uno de estos métodos, si tiene parámetros o no y, en caso afirmativo, cuántos y de qué tipo sería cada uno.

Realiza un análisis en el que decidas, para cada uno de los métodos de la tabla anterior y teniendo en cuenta la función que van a realizar, qué parámetros serían los apropiados.

Tendremos que ir método por método e ir reflexionando acerca de su función y qué parámetros serían necesarios recibir desde fuera (al ser invocados) para que puedan llevar a cabo su tarea apropiadamente:

- ✓ método `getNivelDeposito`. En este caso se trata de un método que nos proporciona información del objeto y que no lleva a cabo realmente ninguna acción, simplemente devuelve un valor, pero no recibe información alguna. Por tanto no tendrá parámetros, aunque sí devolverá un valor de tipo `double` (el nivel actual de combustible);
- ✓ método `arrancar`. Éste sí es un método que va a realizar una acción: encender el motor del vehículo. Lo más probable es que no devuelva ningún valor (cuando pides "arrancar" el vehículo estás solicitando que se lleve a cabo una acción, no estás pidiendo información). Sin embargo, al igual que en el caso anterior, tampoco requiere que se le pase ningún valor a través de un parámetro, pues simplemente quieres arrancar el vehículo. Por tanto, tampoco tendrá parámetros y tampoco devolverá nada (`void`);
- ✓ método `parar`. Este método es similar al anterior. Realiza una acción, no devuelve ningún valor y no requiere que se le pase ninguna información a través de parámetros pues su acción es clara y no necesita matizaciones ni parametrizaciones. Tampoco devolverá ningún valor (`void`);
- ✓ método `repostar`. En este caso sí es necesario como mínimo un dato: la **cantidad de combustible** (litros) que se desea repostar. Es decir, necesita un parámetro de entrada para poder llevar a cabo la acción apropiadamente. Necesitará **un parámetro** de tipo `real` (`double`) y no devolverá ningún valor (`void`);
- ✓ método `isArrancado`. Aquí tenemos nuevamente un método que lo que hace es devolver información y no realizar una acción. En este caso no es necesario proporcionar ningún valor al método a través de parámetros, pues está clara la información que se va a dar: si el motor del vehículo está arrancado o no, es decir, el estado del motor. Podría devolver en este caso un tipo `boolean`;
- ✓ método `recorrerTrayecto`. En este caso tenemos un método que va a llevar a cabo una acción (recorrer una determinada distancia). Por tanto, necesitará **un parámetro**: la **distancia** que se desea recorrer (kilómetros), la cual será de tipo `real` (`double`). Respecto a lo que devuelve el método, en este caso no sería necesario que devolviera ningún valor (`void`), pues simplemente se lleva a cabo una acción.

Una vez realizado este análisis, que es bastante razonable, se puede observar que existen otras formas de plantear la situación y que resultarían igual de razonables. Por ejemplo:

- ✓ en el método `arrancar`, podría darse el caso de que por las especificaciones que nos hubieran dado, nos interesaría saber si la acción de arrancar se ha podido llevar a cabo o no, en cuyo caso podría ser útil que se devolviera un `boolean` (`true` si se ha arrancado, `false` si ha habido cualquier problema que lo haya impedido, como pueda ser que el depósito de combustible esté vacío, o que haya un fallo en el motor). En nuestro análisis inicial hemos supuesto que esa información no nos interesa de momento, así que lo hemos dejado como `void`. Pero como puedes observar, no es la única posibilidad ni tiene por qué ser la mejor. Tan solo se trata de que sea la que más nos interese dependiendo de cómo queramos que se utilice ese método, lo que se nos haya especificado en el equipo de desarrollo o cualquier otra consideración que se nos pueda ocurrir;
- ✓ para el método `parar` podríamos plantearnos exactamente lo mismo que para el método `arrancar`;
- ✓ para el método `recorrerTrayecto` podríamos hacer que devolviera un `double` indicando la distancia real que se ha podido recorrer, por si no coincide con la que se ha intentado recorrer. Imagina, por ejemplo, que se ha agotado el combustible antes de recorrer el trayecto completamente.

La cuestión es hacerte ver que no hay una manera "única" ni "sagrada" de plantearse la interacción con un método (qué parámetros recibe y qué valores devuelve) sino que se trata de algo que tendremos que analizar y sopesar dependiendo del uso que queramos darle a cada método. Nosotros vamos a escoger la primera opción, pero los ejemplos la segunda no tendrían por qué ser incorrectos.

A partir de ese análisis, escribe cómo quedaría la cabecera de cada uno de esos métodos teniendo en cuenta que deseamos que sean todos ellos **públicos** (accesibles desde cualquier objeto de cualquier paquete).

TEMA 5: DESARROLLO DE CLASES

Una vez que ya tenemos claro qué parámetros y de qué tipo va a tener cada método, tan solo hay que declararlos usando la sintaxis de Java:

```
[private | protected | public] [static] [abstract] [final] [native] [synchronized]  
<tipo> <nombreMetodo> ( [<lista_parametros> ] ) [throws <lista_excepciones>]
```

Por tanto la cabecera de cada uno de los métodos quedaría más o menos así:

```
public double getNivelDeposito()           // Sin parámetros, devuelve un número real  
public void arrancar()                     // Sin parámetros, no devuelve nada  
public void parar()                        // Sin parámetros, no devuelve nada  
public void repostar (double cantidad)     // Un parámetro: cantidad de litros a repostar (real). No devuelve nada  
public boolean isArrancado()               // Sin parámetros, devuelve un boolean (verdadero o falso)  
public void recorrerTrayecto (double distancia) // Un parámetro: distancia a recorrer (real). No devuelve nada
```

5.4.1. LISTA DE PARÁMETROS VARIABLES.

En el lenguaje Java es posible utilizar una construcción especial llamada **varargs (argumentos variables)** que permite que un método pueda tener un número variable de parámetros. Para utilizar este mecanismo se coloca un tipo, unos **puntos suspensivos** (tres puntos: "..."), un espacio en blanco y a continuación el nombre del parámetro que aglutinará la lista de argumentos variables.

```
<tipo> <nombreMetodo> (<tipo>... <nombre>)
```

Es posible además mezclar el uso de **varargs** con parámetros fijos. En tal caso, **la lista de parámetros variables debe aparecer al final (y sólo puede aparecer una)**.

En realidad se trata una manera transparente para el usuario del método de pasar un array con un número variable de elementos para no tener que hacerlo manualmente. Dentro del método habrá que ir recorriendo el array para ir obteniendo cada uno de los elementos de la lista de argumentos variables. Ya veremos algún caso de uso cuando tratemos el cuerpo de los métodos.

Veamos ahora un ejemplo. Supongamos que tenemos una clase que contiene un método llamado **sumar** que necesita recibir como parámetros varios elementos de tipo real para ser sumados. Inicialmente podríamos pensar en escribir varias versiones de métodos de suma con cada posibilidad (un parámetro, dos parámetros, tres parámetros, etc.):

```
public void sumarUnElemento (double elemento)  
public void sumarDosElementos (double elemento1, double elemento2)  
public void sumarTresElementos (double elemento1, double elemento2, double elemento3)  
...
```

Lo cual podría ser bastante tedioso y redundante. Para evitar esa situación, podríamos utilizar esta construcción, que nos permitiría tener la posibilidad de disponer de un número indeterminado de parámetros, todos del mismo tipo:

```
public void sumarElementos (double... elementos)
```

TEMA 5: DESARROLLO DE CLASES

Si quisieramos combinar esa construcción con algún otro parámetro "fijo", tendríamos que colocar ese parámetro o parámetros al principio. Por ejemplo si además de los elementos que se desean sumar se requieren otros dos parámetros (**sección** de tipo **String** y **unidad** de tipo **int**), habría que ubicarlos delante de la especificación de parámetros variables:

```
public void sumarElementos (String seccion, int unidad, double... elementos)
```

De esta manera tendríamos dos parámetros "fijos" (de tipo **String** e **int** respectivamente) junto con un número variable de parámetros de tipo **double**.

Para saber más

Puedes echar un vistazo a la sección "*Arbitrary Number of Arguments*" del artículo general sobre paso de parámetros al que hicimos referencia en el apartado anterior: [Passing Information to a Method or a Constructor.](#)



Ejercicio Resuelto

En apartados anteriores hemos considerado la posibilidad de incluir en la clase **Vehículo** un método llamado **recorrerTrayecto** con la idea de hacer que el objeto vehículo sobre el que se aplique ese método recorra una determinada distancia. Para ello habíamos declarado el método con un parámetro de tipo real:

```
public void recorrerTrayecto (double distancia) // Un parámetro: distancia que se desea recorrer
```

Si quisieramos que existiera la posibilidad de que ese método pudiera recibir varios parámetros para de esa manera recorrer varios trayectos y no solo uno, ¿cómo quedaría la cabecera del método?

[Mostrar retroalimentación](#)

Si aplicamos la construcción **varargs**, que nos permite indicar una lista de una cantidad indeterminada de parámetros, todos ellos del mismo tipo, la cabecera del método podría quedar así:

```
public void recorrerTrayecto (double... distancia) // Una lista variable de trayectos
```



Ejercicio Resuelto

Imagina que estamos desarrollando una clase llamada **Mates**, que va a incluir métodos para efectuar operaciones matemáticas.

Una de esas operaciones va a ser la de **sumar listas de números enteros** con una **cantidad de sumandos indeterminada**. Para ello se va a disponer de un método llamado **sumar**, de tal manera que la cantidad de parámetros pudiera ser variable, ¿cómo quedaría la cabecera de ese método?

[Mostrar retroalimentación](#)

Aplicando nuevamente la construcción **varargs**, que nos permite indicar una lista de una cantidad indeterminada de parámetros, todos ellos del mismo tipo, la cabecera del método podría quedar así:

```
public int sumar (int... sumando) // Una lista variable de sumandos
```

5.4.2. MODIFICADOR FINAL EN LOS PARÁMETROS.

Ya hemos visto que el modificador **final** aplicado a variables y atributos significa que su valor no puede ser modificado una vez que se realiza una primera asignación.

En Java también se permite añadir el modificador **final** a los parámetros de un método y tiene un significado similar: el valor del parámetro no podrá ser modificado dentro del método. Si el parámetro no incluye ese modificador, el valor en el interior del parámetro podría ser modificado dentro del método si así se decidiera hacer.

Es decir, que sería lícito realizar modificaciones de parámetros como, por ejemplo:

```
private void metodo (int v1, int v2, int v3) {
    v1 += 10;
    v2 += 20;
    v3 += 30;
}
```

Ahora bien, eso no suele tener mucho sentido. Si queremos hacer alguna modificación de un parámetro de entrada, lo más razonable sería disponer de variables locales que almacenaran el resultado de esa modificación y dejar a los parámetros con su valor original por si más adelante, dentro del mismo método, volviéramos a necesitar ese valor. Por ejemplo, haciendo algo así:

```
private void metodo (int v1, int v2, int v3) {
    int x1, x2, x3
    x1= v1 + 10;
    x2= v1 + 20;
    x3= v3 + 30;
}
```

Pero, si quisiéramos evitar que todos los parámetros (o bien sólo algunos) pudieran ser modificados desde dentro del método para evitar, por ejemplo, una posible confusión por parte del programador, Java proporciona la posibilidad de declarar esos parámetros como **final**:

```
private void metodo (final int v1, final int v2, final int v3) {
    int x1, x2, x3
    x1= v1 + 10;
    x2= v1 + 20;
    x3= v3 + 30;
}
```

De ese modo, si se intentara hacer una modificación de algunos de los parámetros como por ejemplo una asignación del tipo **v1 +=10**, Java no lo permitiría y se produciría un error de compilación.

En cualquier caso, dado que en Java el paso de parámetros es siempre por valor (también conocido como "por copia") ese cambio sólo afectaría al interior del método y no al valor que se pasa como parámetro en la invocación. En ese sentido no hay peligro alguno.

¡No debes cambiar el valor de un parámetro de entrada en el interior de un método! Para llevar a cabo cálculos auxiliares siempre puedes declarar variables locales.

Para saber más

Para el nivel de nuestro curso de iniciación a la programación, con lo que se ha dicho al respecto es más que suficiente, pero si estás interesado en este tema puedes profundizar algo más a través de los siguientes enlaces de la web Stackoverflow:[Artículo "Making java method arguments as final"](#) Y [Artículo "Why should I use the keyword "final" on a method parameter in Java?"](#)

5.5 CUERPO DE UN MÉTODO.

Por fin ha llegado el momento de que empecemos a escribir las líneas de código que conforman un método, esto es, el **cuerpo del método**.

El interior de un método (cuerpo) está compuesto por una serie de sentencias en lenguaje Java:

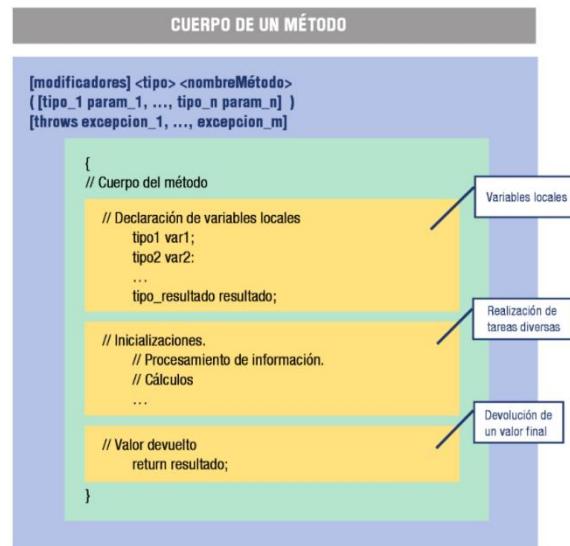
- sentencias de **declaración de variables locales** al método;
- sentencias que implementan la **lógica del método** (estructuras de control como bucles o condiciones; utilización de métodos del propio objeto o de otros objetos; cálculo de expresiones matemáticas, lógicas o de cadenas; creación de nuevos objetos, etc.). Es decir, todo lo que has visto en las unidades anteriores. Se trata de algo así como un "miniprograma" con su entrada, su procesamiento y su salida;
- sentencia de **devolución del valor de retorno (return)**. Aparecerá al final del método y es la que permite devolver la información que se le ha pedido al método. Es la última parte del proceso y la forma de comunicarse con la parte de código que llamó al método (paso de mensaje de vuelta). Esta sentencia de devolución siempre debe aparecer al final del método salvo que el tipo devuelto por el método sea **void** (vacío), pues en tal caso no hay que devolver nada al código que hizo la llamada.

Veamos cómo podrían quedar algunos de los métodos de la clase **Vehículo** si escribimos el código Java que deberían contener en su interior:

```
public double getNivelDeposito () {
    return nivelDeposito;
}

public LocalDate getFechaMatriculacion () {
    return fechaMatriculacion;
}
```

En ambos ejemplos lo único que hacen estos métodos es simplemente devolver un valor (utilización de la sentencia **return**). No recibe parámetros (información de entrada) ni hace cálculos, ni obtiene resultados intermedios o finales. Es un **método de consulta** que tan solo devuelve el contenido de un atributo. Se trata de uno de los métodos más sencillos que se pueden implementar: un método que devuelve el valor de un atributo. Se les suele llamar métodos de tipo **get** o **getters** (de **obtener**, en inglés) y ya los veremos detenidamente más adelante.



5.5.1. VALOR DEVUELTO POR UN MÉTODO: RETURN.

En el apartado anterior hemos visto que, desde el punto de vista del valor devuelto por un método, puede suceder:

1. que **no devuelva nada**, entonces en la cabecera se indica que devuelve el tipo **void** ("vacío");
2. que **devuelva un determinado valor**, en tal caso en la cabecera se indica el tipo que devuelve (bien un tipo primitivo o bien un tipo referenciado como podría ser un array o un objeto de alguna clase).

En Java, para devolver un valor desde un método se utiliza la sentencia return.

El tipo del valor que devuelva la sentencia return debería de ser del mismo tipo o de un tipo compatible con el tipo del valor de retorno definido en la cabecera del método, pudiendo ser desde un entero a un objeto creado por nosotros. Es decir, que si en la cabecera se indica **int**, se puede devolver un **int** (o un **short** o un **byte**), pero nunca un **long**, un **float**, un **double**, un **char** o un **boolean**.

Veamos algunos ejemplos sencillos:

1. Método que devuelve un valor entero aleatorio entre 1 y 6:
2. Método que recibe dos números reales y devuelve el producto de esos dos números:

```
byte valorAzar () {
    return (byte) (Math.random()*6) + 1;
}
```

```
double sumador (double numero1, double numero2) {
    return numero1 * numero2;
}
```

3. Método que devuelve la fecha actual incrementada en tres días:

```
LocalDate hoyMasTresDias () {
    return LocalDate.now().plusDays(3);
}
```

4. Método que incrementa la cantidad de combustible de un vehículo (no devuelve nada y por tanto no hay **return**):

```
public void repostar (double cantidad) {
    nivelDeposito= nivelDeposito + cantidad;
}
```

Como puedes observar, sólo en el caso de que el tipo devuelto por el método sea **void**, se puede evitar usar la sentencia **return**. Si no, es obligatorio introducirla y su ejecución haría que el método finalizara en esa línea devolviendo un determinado valor.

En algunos lenguajes, a los métodos que devuelven algún valor se les llama **funciones** y a los que no devuelven nada **procedimientos**.

Lo habitual es que la sentencia return esté siempre al final del método, de este modo se tendrá una entrada y una salida, cumpliendo con uno de los principios de la programación estructurada, de forma que la salida natural para cualquier método será ejecutar la última sentencia que contiene, y que debe ser un **return**. No tiene mucho sentido que exista código después de un **return**, pues el método finaliza su ejecución y se devuelve el control a las líneas de código que invocaron al método. Es algo así como una indicación del final del "mini programa", culminando con la devolución de algún resultado a la parte de código que invocó a ese método. No obstante, también es posible utilizar una sentencia **return** en cualquier punto de un método, con lo que éste finalizará en el lugar donde se encuentre dicho **return**. Pero de esta forma, estaríamos creando

TEMA 5: DESARROLLO DE CLASES

"puertas traseras" que complicarían la comprensión del código, y lo harían más difícil de entender y mantener, pudiendo generar efectos colaterales oscuros no previstos que deriven en un mal funcionamiento.

Aunque Java permite incluir más de una sentencia **return** dentro de un método, no es aconsejable, pues dificultaría la legibilidad del código al entenderse que la finalización de la ejecución del método puede suceder en distintas partes de éste. En conclusión:

Se recomienda NO incluir más de un return en un método, que deberá ir al final del mismo, como salida natural.

El valor de retorno es opcional, si lo hubiera debería de ser del mismo tipo o de un tipo compatible con el tipo del valor de retorno definido en la cabecera del método, pudiendo ser desde un entero a un objeto creado por nosotros. Si no lo tuviera, el tipo de retorno sería **void**, y **return** serviría para salir del método sin necesidad de añadirle ningún valor ni expresión detrás.

5.5.2. VARIABLES LOCALES.

Hemos realizado la comparación del cuerpo de un método con un pequeño programa que en ocasiones podrá tener cierta complejidad. En tales casos puede ser habitual que necesitemos algunas **variables internas o auxiliares** para realizar cálculos intermedios. Esas variables son conocidas con el nombre de **variables locales**.

La forma de declarar una variable local es exactamente la misma que vimos a principio de curso: tipo y nombre de la variable junto con aquellos modificadores que permita el lenguaje. De hecho, **en Java, las variables que has estado usando hasta ahora ya eran locales**, pues eran variables declaradas **dentro de un método llamado main**.

Por ejemplo, supongamos que vamos a implementar un método (llamado por ejemplo **sumaSerie**) que sume todos los números que hay entre dos enteros que se pasan como parámetros. Para llevar a cabo esta tarea necesitaríamos al menos **un par de variables**: un **contador** para ir desde un número hasta otro y un **acumulador** para ir sumando sucesivamente cada número.

```
int sumaSerie (int inicio, int fin) {  
    int contador; // Variable local contador  
    int suma= 0; // Variable local acumuladora de la suma de cada número recorrido  
    for (contador=inicio; contador<=fin; contador++) { // Recorremos todos los números desde inicio hasta fin  
        suma +=contador; // Vamos sumando en el acumulador suma cada uno de los números que vamos recorriendo  
    }  
    return suma; // Devolvemos la suma acumulada  
}
```

Debes evitar asignar a las variables locales nombres que coincidan con el de algún parámetro. El compilador no lo permitirá pues se trataría de un identificador duplicado.



Recomendación

Es muy importante que sepamos distinguir cuándo un dato determinado debe ser almacenado en un atributo y cuándo se trata simplemente de un cálculo intermedio o auxiliar y por tanto es suficiente con una variable local.

Ten en cuenta que la información almacenada en una variable local desaparece en cuanto finalice la ejecución de un método. En consecuencia, si se puede volver a calcular esa información en el futuro sin necesidad de tener que guardarla en un atributo, es que no es necesario un atributo para ella.

Ejemplo: imagina una clase que representa a un **rectángulo**, la cual contiene sendos atributos de tipo real para almacenar su **base** y su **altura**. Podríamos tener también un atributo para guardar su **superficie**, pero no tendría sentido, pues cada vez que quisieramos saber su superficie podríamos calcularla.

TEMA 5: DESARROLLO DE CLASES

Hay lenguajes en los que existe la posibilidad de disponer de variables globales. Se trata de variables a las cuales se tiene acceso desde cualquier parte del programa.

En Java, así como en otros lenguajes orientados a objetos, los programas consisten en un conjunto de clases, unas independientes de otras, dando lugar a que toda variable o atributo que se defina en cada clase sea independiente a las que se definan en las demás. Esto hace que el concepto de variable global no tenga mucho sentido en este contexto.

Lo más parecido a una variable global que podríamos encontrar en programas escritos en lenguaje Java serían los siguientes casos:

- ✓ Los **atributos estáticos** de una clase, que serían valores comunes y compartidos por todos los objetos instancia de una misma clase. Estos atributos podrían considerarse como variables "globales" a todos los objetos que se instancien de la clase.
- ✓ Los **atributos estáticos públicos**, que además de ser compartidos por todos los objetos de una misma clase, serían accesibles desde cualquier método de cualquier otra clase, no sólo desde los objetos instancia de la propia clase. Es decir, serían visibles desde cualquier parte del programa.
- ✓ Los **atributos estáticos contenidos por la clase principal de un programa Java**. Se considera la clase principal de un programa Java aquella que contiene un método `main`. El método `main` de una clase Java, que veremos más adelante, es un método estático especial a partir del cual comenzaría la ejecución de un programa Java.

Para saber más

Si tienes interés acerca del concepto de **variable global** frente al de **variable local** en los lenguajes de programación puedes consultar la siguiente referencia: [Variable global \(Wikipedia\)](#).

Si declaras una variable local con el mismo nombre (identificador) que un atributo, el compilador de Java lo permitirá, pero debes tener en cuenta que cuando hagas referencia a ese identificador te estarás refiriendo a la variable local y no al atributo. En estos casos se dice que has ocultado (en inglés "hidden", ocultado o "shadowed", "ensombrecido") el atributo.

Esto sucede tanto con las variables locales como con los parámetros del método: si se llaman igual que un atributo, ocultarán (*hide* o *shadow*, en inglés) al atributo. Pero entonces, ¿cómo podríamos acceder al atributo? Para resolver este problema dispondrás del operador de autorreferencia `this`, que verás en el próximo apartado.

Puede que entonces te preguntes ahora: ¿y qué pasa si un parámetro y un atributo tienen el mismo nombre? Esta situación no es posible y ya la hemos comentado anteriormente. *Dentro de un método, el compilador de Java no permite nombrar a una variable local con el mismo identificador que un parámetro del método.*

5.5.3. EL OPERADOR DE AUTORREFERENCIA THIS

Normalmente, en la mayoría de los lenguajes orientados a objetos, se dispone de algún **mecanismo de referencia al propio objeto dentro de un método**. En el caso específico de Java existe la palabra reservada `this`, conocida como el **operador de autorreferencia**. El uso de este operador puede resultar muy útil a la hora de evitar la ambigüedad que puede producirse entre el **nombre de un parámetro de un método y el nombre de un atributo cuando ambos tienen el mismo identificador (mismo nombre)**. En tales casos el parámetro "oculta" o se "superpone" al atributo y no tendríamos acceso directo a él (al escribir el identificador estaríamos haciendo referencia al parámetro y no al atributo). En estos casos la referencia `this` nos permite acceder a estos atributos ocultados por los parámetros.

Dado que `this` es una referencia al propio objeto con el que te encuentras trabajando en ese momento, puedes acceder a sus miembros (atributos y métodos) mediante el operador punto `(.)` como sucede con cualquier otro objeto. Por tanto, en lugar de poner el nombre del atributo (que en los casos de ambigüedad se haría referencia al parámetro), podrías escribir `this.nombreAtributo`, de manera que el compilador sabrá que te estás refiriendo al atributo y se eliminará la ambigüedad.

Por ejemplo, si tenemos una clase `Persona` con atributos `nombre`, `apellido1`, `apellido2`, etc. y queremos escribir un método para modificar el primer apellido, podríamos escribir un método llamado `cambiarApellido1` que hiciera lo siguiente:

TEMA 5: DESARROLLO DE CLASES

```
public class Persona {  
    private String nombre;  
    private String apellido1;  
    private String apellido2;  
    ...  
  
    public void cambiarApellido1 (String nuevoApellido1) {  
        apellido1= nuevoApellido1; // Se asigna al atributo apellido1 el valor contenido por el parámetro nuevoApellido1  
    }  
    ...  
}
```

Aquí no habría problema, dado que no se produce ambigüedad entre el identificador utilizado para el parámetro (**nuevoApellido1**) y el identificador usado para el atributo (**apellido1**). Sin embargo, si hubiéramos querido utilizar también **apellido1** como identificador para el parámetro habríamos tenido un problema de **ambigüedad**:

```
public void cambiarApellido1 (String apellido1) {  
    apellido1= apellido1; // Se asigna al atributo apellido1 el valor contenido por el parámetro apellido1 -> No funciona  
}
```

Esto no va a funcionar correctamente porque en la asignación **apellido1 = apellido1** nosotros deseamos que se asigne al atributo **apellido1** el valor contenido por el parámetro **apellido1**, pero eso no es posible porque el compilador de Java no sabe en qué caso nos estamos refiriendo al atributo y en qué caso nos estamos refiriendo al parámetro. Es imposible saberlo dado que tienen el mismo nombre o identificador. Se trata de una situación ambigua que el compilador resuelve haciendo que el parámetro "oculte" al atributo y siendo por tanto imposible acceder al atributo. Eso haría que esa sentencia simplemente hiciera la asignación al parámetro del propio valor del parámetro, lo cual no tiene mucho sentido, porque lo deja como estaba. De hecho, para evitar que el valor de un parámetro pueda ser modificado se le suele añadir el modificador **final** en la declaración del método.

Recuerda: en un método en Java, en caso de ambigüedad entre el identificador de un parámetro y el identificador de un atributo, el parámetro siempre oculta al atributo.

¿Cómo podemos solucionar esto? Podemos hacerlo de dos formas:

1. como se hizo inicialmente: asignando al parámetro un identificador (**nombre**) diferente al del atributo.
De este modo no se produce ambigüedad;
2. utilizando el operador de autorreferencia **this** para indicar cuándo nos estamos refiriendo al atributo.
De este modo, en lugar de escribir simplemente el nombre del atributo, que se podría confundir con el nombre del parámetro, escribimos **this.nombreAtributo**. En este caso **this.apellido1**.

Si utilizamos el operador de autorreferencia **this**, el método quedaría de la siguiente manera:

```
public void cambiarApellido1 (String apellido1) {  
    this.apellido1= apellido1; // Se asigna al atributo apellido1 el valor contenido por el parámetro apellido1  
}
```

TEMA 5: DESARROLLO DE CLASES

De esta manera hemos resuelto el caso de la ambigüedad. Ahora bien, ¿qué hacemos con el resto de atributos si no existe el peligro de ambigüedad? Cuando queramos referirnos a ellos, ¿escribimos simplemente el nombre del atributo o utilizamos siempre a partir de ahora la referencia **this**?

Nuestra recomendación es que que utilicéis siempre la referencia this.

De esa manera en el código de los métodos siempre quedará claro que nos estamos refiriendo a un atributo del objeto y no a un parámetro o a una variable local. Será una forma sencilla de mejorar la legibilidad del código y de que quede "mejor autodocumentado".

Quizá inicialmente nos pueda resultar tedioso tener que escribir **this** seguido de un punto delante de cada atributo cada vez que lo usemos en un método, pero a la larga hará que el cuerpo de nuestros métodos sea más legible dado que cuando lo veas sabrás rápidamente que se está haciendo referencia a un atributo sin tener que ir a consultar la lista de atributos de la clase.

En cualquier caso, siempre **puedes decidir usar la referencia this sólo cuando sea imprescindible** (casos de ambigüedad por coincidir con el nombre de un parámetro). Lo que sí te recomendamos en caso de que tomes esa decisión, es que seas coherente y lo hagas siempre de la misma manera: no utilices nunca **this** salvo que sea imprescindible y no de manera arbitraria para unos atributos sí y para otros no, o en unos métodos sí y en otros no, porque entonces podrías confundir a quien leyera el código. La idea es siempre facilitar al máximo la legibilidad del código para comprender qué es lo que se hace en cada método.

Recomendación: siempre resulta útil hacer uso de la referencia this aunque no sea necesario, pues ayuda a mejorar la legibilidad del código.

Para saber más

Puedes echar un vistazo al artículo general sobre la referencia **this** en los manuales de Oracle: [Using the this Keyword](#).



Reflexiona

Si has leído el enlace del para saber más anterior, puede que ya lo hayas descubierto al leer el artículo "*Using this with a Constructor*", pero conviene llamar la atención explícitamente sobre esto:

No debes confundir el operador de autorreferencia `this` con el método `this()`.

El método `this()` resulta útil para reutilizar código al invocar a un constructor ya existente desde la primera línea de otro constructor de la misma clase, para así no tener que volver a escribir las líneas de código que contenía el constructor invocado en el nuevo constructor. Se verá su funcionamiento y su utilidad un poco más adelante en esta unidad, en el apartado dedicado a constructores.

Por ahora sólo queremos advertirte acerca de la existencia de un método `this()`, para que sepas que es algo distinto al operador `this`.



Recomendación

En el caso de los **atributos de clase (estáticos)** no tiene mucho sentido utilizar el operador `this`, pues hace referencia al propio objeto y los atributos estáticos son atributos de clase y no de objeto.

En tal caso **se recomienda escribir el nombre de la clase y no `this`**.

Por ejemplo, en el caso de la clase `Vehiculo` se recomienda usar directamente el nombre de la clase (`Vehiculo`):

- ✓ `Vehiculo.vehiculosCreados` Y NO `this.vehiculosCreados`.
- ✓ `Vehiculo.vehiculosArrancados` Y NO `this.vehiculosArrancados`.
- ✓ `Vehiculo.kilometrosTotalesFlota` Y NO `this.kilometrosTotalesFlota`.
- ✓ `Vehiculo.MINIMA_CAPACIDAD_DEPOSITO` Y NO `this.MINIMA_CAPACIDAD_DEPOSITO`.
- ✓ etc.

Declarar un **atributo como estático** O `static` significa que en memoria **sólo habrá una única zona reservada para guardar el valor del mismo, ligada a la clase**. A la hora de acceder a ese valor, lo razonable será entonces **referenciarlo mediante el identificador de la clase** (por ejemplo, `Vehiculo.vehiculosCreados`), lo cual es lo más aconsejable.

Ahora bien, la referencia `this` (desde dentro del objeto) o el identificador de un objeto instancia de la clase (si el atributo es público) también pueden servir en Java para hacer referencia a un miembro estático (por ejemplo `this.vehiculosCreados`; o bien `miCoche.vehiculosCreados` suponiendo que `miCoche` sea de tipo `Vehiculo` y que el atributo `vehiculosCreados` fuera público) aunque se desaconseje hacerlo así.

El motivo de que Java permita la consulta referenciándolo mediante instancias es que el compilador siempre conoce cuál es la clase del objeto, así que no tiene problema en "averiguar" implícitamente cuál es la clase de ese objeto, y consultar el valor para esa clase. Sin embargo, para nosotros, si vemos `Vehiculo.vehiculosCreados`, es evidente, sin necesidad de ver la declaración, que `vehiculosCreados` es un atributo `static` mientras que si vemos `this.vehiculosCreados` O `miCoche.vehiculosCreados`, no podemos tener esa seguridad. De hecho, salvo que consultáramos la declaración de ese atributo, nos parecería que estamos accediendo a un atributo del objeto `miCoche`, cuando en realidad no es así (estamos accediendo a un atributo estático de la clase `Vehiculo`). ¡¡Siempre debemos intentar que el código sea claro a simple vista!!



Ejercicio resuelto

El método `arrancar` de la clase `Vehiculo` sirve para poner en marcha el motor de un vehículo.

Implementa el método `arrancar` de la clase `Vehiculo` considerando lo siguiente:

1. **no devuelve ningún valor (void);**
2. **no tiene parámetros;**
3. **si se intenta poner en marcha el motor de un vehículo que ya lo tiene arrancado**, no debe hacerse nada, pues el motor ya está arrancado;
4. **si se trata de un vehículo con el motor parado**, habrá que **cambiar su estado** y actualizar el atributo de clase que controla la **cantidad de vehículos arrancados** en cada momento.

Te recomendamos hacer referencia a los atributos usando `this`, pero ten en cuenta que no tiene mucho sentido hacerlo para los atributos de clase. En tal caso se recomienda utilizar como referencia el propio nombre de la clase (`Vehiculo`).



[linekslawek \(Pixabay License\)](#)

Teniendo en cuenta todo lo anterior, la implementación del método `arrancar` podría quedar así:

```
public void arrancar () {
    if (!this.estadoMotor) { // Si el motor no está arrancado
        this.estadoMotor= true; // Si el motor no estaba encendido, se arranca
        Vehiculo.vehiculosArrancados++; // Actualizamos la cantidad de vehículos arrancados (se incrementa en uno)
    }
}
```

Fíjate que hemos usado la referencia `this` (del propio objeto) para los atributos de objeto (`estadoMotor`) mientras que hemos usado la referencia de la clase (`Vehiculo`) para los atributos de clase (`vehiculosArrancados`).

¿Podríamos haber implementado el método sin haber usado las referencias a `this` y/o a `Vehiculo`?

Perfectamente. Dado que no hay posibilidades de ambigüedad con parámetros (de hecho ni siquiera hay parámetros), pueden usarse directamente los identificadores de los atributos sin ningún miedo a posibles coincidencias con los nombres de parámetros.

TEMA 5: DESARROLLO DE CLASES

```
public void arrancar () {
    if (!estadoMotor) { // Si el motor no está arrancado
        estadoMotor= true; // Si el motor no estaba encendido, se arranca
        vehiculosArrancados++; // Actualizamos la cantidad de vehículos arrancados (se incrementa en uno)
    }
}
```

¿Qué opción es la más recomendable? Nosotros, en principio, somos partidarios de **usar siempre las referencias explícitas al objeto (`this`) y a la clase (`Vehículo` en este caso) para identificar los atributos (de objeto y de clase)**. De ese modo siempre quedará claro si el elemento es:

- ✓ un atributo de objeto (va precedido de la referencia de objeto `this`),
- ✓ un atributo de clase (va precedido de la referencia de clase `Vehículo`),
- ✓ un parámetro o una variable local (no va precedido por ninguna referencia).

Te recomendamos hacerlo así al menos durante tu período de aprendizaje. Te será mucho más sencillo encontrar errores y entenderás mejor tu código. Más adelante, dependiendo de dónde desempeñes tu actividad profesional, es posible que tengas que amoldarte a algún convenio o estándar común para todos los programadores que trabajen en un mismo equipo.

Recomendación: usar la referencia `this` para los atributos de instancia y la referencia `NombreDeClase` para los atributos de clase, ayudará a mejorar la legibilidad de tu código.



Ejercicio Resuelto

El cálculo de la **superficie ocupada por un rectángulo** se realiza multiplicando su base por su altura ($\text{base} \times \text{altura}$). Para calcular la **longitud de su perímetro** se lleva a cabo mediante la suma de la longitud de cada uno de sus lados ($\text{base} + \text{altura} + \text{base} + \text{altura}$, o bien $2 \times \text{base} + 2 \times \text{altura}$).

Implementa los siguientes métodos para la clase `Rectangulo`:

- ✓ método para obtener la longitud de la **base**,
- ✓ método para obtener la longitud de la **altura**,
- ✓ método para obtener la longitud del **perímetro**,
- ✓ método para obtener la **superficie** ocupada.

Teniendo en cuenta que entre los atributos de objeto de un rectángulo tenemos los dos **vértices de ubicación inferior izquierdo (x_1, y_1) y superior derecho (x_2, y_2)**, es fácil calcular a partir de ahí su **base** y su **altura**:

- ✓ **base** = $x_2 - x_1$
- ✓ **altura** = $y_2 - y_1$

La implementación de los métodos que devuelven esa información quedaría entonces así:

```
public double calcularBase () {
    return this.x2 - this.x1;
}

public double calcularAltura () {
    return this.y2 - this.y1;
```

Respecto al cálculo del **perímetro** y de la **superficie**, podemos aprovechar esos métodos para hacer parte del trabajo:

```
public double calcularSuperficie () {
    return calcularBase() * calcularAltura();
}

public double calcularPerimetro () {
    return 2*calcularBase() + 2*calcularAltura();
```

TEMA 5: DESARROLLO DE CLASES

Como puedes observar, si es necesario puedes **invocar a métodos de la clase desde dentro de otros métodos**. De esa manera **evitas la redundancia de código**, algo que siempre debemos procurar.

Dado que esos métodos son **miembros del objeto** también podrías haber usado la referencia `this` con ellos:

```
public double calcularSuperficie () {  
    return this.calcularBase() * this.calcularAltura();  
}  
  
public double calcularPerimetro () {  
    return 2*this.calcularBase() + 2*this.calcularAltura();
```

Es menos habitual, pero también se utiliza en algunas ocasiones.

Siguiendo una línea de actuación similar a la anterior, ¿te atreverías a implementar los siguientes métodos para la clase `Círculo`?

- 1.- método para el cálculo de la **longitud del perímetro**,
- 2.- método para el cálculo de la **superficie**.

Para implementar esos dos métodos tan solo hemos de utilizar las expresiones matemáticas que nos permiten calcular esas dos medidas:

- 1.- longitud de la circunferencia que rodea a un círculo = $2 \times \pi \times \text{radio}$,
- 2.- superficie de un círculo = $\pi \times \text{radio}^2$.

La implementación por tanto quedaría:

```
public double calcularSuperficie () {  
    return Círculo.PI * this.radio*this.radio; // PI por radio al cuadrado  
}  
  
public double calcularPerimetro () {  
    return 2 * Círculo.PI * this.radio; // 2 por PI por el radio
```

5.5.4. LANZANDO EXCEPCIONES DESDE UN MÉTODO

Habrá ocasiones en las que el intento de ejecución de un método pueda dar lugar a una situación no deseable o un error. En tales casos, el método debería avisar de que se ha producido esa situación.

En algunos lenguajes eso se resuelve devolviendo algún tipo de código de error. Por ejemplo, si se espera que un método devuelva un número positivo o cero, puede devolverse un número negativo, por ejemplo -1, para indicar que se ha producido un error. Si existen diversas posibilidades de error, entonces podrían devolverse distintos números negativos en función del tipo de error que se haya producido.

Como ya hemos visto en unidades anteriores, Java dispone de un mecanismo más sofisticado de comunicación de errores o situaciones no deseables: la gestión de **excepciones**.

Hasta ahora habías visto cómo tratar posibles excepciones generadas por la ejecución de algún método de una clase o por algún operador. Para ello utilizabas la estructura **try - catch - finally**.



Recomendación

La gestión de excepciones ya se ha tratado en unidades anteriores. Si consideras que es necesario hacer un repaso del concepto de excepción o de la estructura `try - catch - finally`, éste es el momento de hacerlo.

TEMA 5: DESARROLLO DE CLASES

Hasta ahora tan solo habías trabajado con excepciones desde el punto de vista de su uso y captura, pero no habías generado ninguna en caso de que se produjera una situación de error. Esto es lo que vamos a aprender a hacer ahora.

Imagina el método **arrancar** de nuestra clase **Vehículo**. Se trata de un método muy simple que lo único que hace es modificar el valor del atributo **estadoMotor**. Sin embargo, podría ser razonable que no dejáramos arrancar un vehículo que ya se encuentra arrancado. Para llevar a cabo esa acción podríamos lanzar una excepción desde el método **arrancar** si el valor del atributo **estadoMotor** es ya **true**, o que no sea posible arrancarlo porque el depósito no tiene combustible, o porque haya algún fallo en el motor, y el intento "fracase".

¿Cómo se lanza una excepción en Java?

Para ello utilizamos la sentencia **throw**. La sintaxis de esta sentencia es la siguiente:

```
throw <objetoDeTipoExcepcion>;
```

Las excepciones son objetos de clases de tipo excepción. Nosotros no vamos a implementar ninguna clase de tipo excepción por el momento, pero sí podemos utilizar algunas que ya nos proporciona la API de Java como son, por ejemplo, **IllegalArgumentException** e **IllegalStateException**.

Para ello simplemente instanciaremos un objeto de una de esas dos clases usando alguno de sus constructores junto con el operador **new**. Las clases de tipo excepción incluyen al menos un constructor con un parámetro que permite proporcionar un mensaje de texto (parámetro de tipo **String**) indicando el tipo de error o situación no deseada que se ha producido.

Por tanto, el lanzamiento de una excepción quedaría más o menos así:

```
throw new ClaseDeTipoExcepcion (<listaParametros>);
```

Por ejemplo:

```
throw new IllegalStateException ("vehículo ya arrancado");
```

En el momento en que se ejecute esa sentencia, el resto del código del método no será ejecutado y se volverá a la línea desde donde el método fue invocado. Es decir, se produce una finalización "brusca" de la ejecución del método que habrá de ser gestionada por quien lo llamó.

En tal caso, el método **arrancar** podría quedar de la siguiente manera:

```
public void arrancar () {
    if (this.estadoMotor) { // Si el motor ya está arrancado
        throw new IllegalStateException ("vehículo ya arrancado"); // Lanzamos una excepción y finaliza la ejecución del método
    }
    this.estadoMotor= true; // Si no se ha lanzado la excepción, es que el motor no estaba encendido y puede arrancarse
    Vehiculo.vehiculosArrancados++; // Actualizamos la cantidad de vehículos arrancados (se incrementa en uno)
}
```

De este modo si se intenta ejecutar el método **arrancar** sobre un vehículo que ya está arrancado, se va a producir una excepción que hará que la ejecución del método finalice, debiendo esa excepción ser recogida y gestionada desde el código donde se realizó la invocación al método **arrancar**. Por ejemplo:

TEMA 5: DESARROLLO DE CLASES

```
try {
    v.arrancar(); // Se intenta arrancar el objeto v instancia de la clase Vehiculo
    System.out.println ("El vehículo ha sido arrancado sin problema.");
} catch (IllegalStateException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage()); // Se mostrará el mensaje "Error: vehículo ya arrancado."
}
```

Por último, nos falta indicar una cosa más: un método debe indicar en su cabecera cuáles son los tipos de excepción que puede lanzar. Ya lo hemos visto cuando estudiamos la cabecera de un método. Es el momento de ponerlo en práctica. Si un método puede lanzar más de un tipo de excepción se indicará con una lista de excepciones separadas por comas. Si sólo puede lanzar un tipo de excepción sólo se indicará esa excepción. En el caso de nuestro método **arrancar**, nos quedaría:

```
public void arrancar () throws IllegalStateException {
    if (this.estadoMotor) {
        throw new IllegalStateException ("vehículo ya arrancado");
    }
    this.estadoMotor= true;
    Vehiculo.vehiculosArrancados++;
}
```

Para saber más

Puedes obtener más información acerca del lanzamiento de excepciones en Java a través de la siguiente referencia: [How to Throw Exceptions](#).

Si quieres repasar aspectos previos sobre la gestión de excepciones vistos en unidades anteriores también puedes echar un vistazo a estas otras referencias: [What Is an Exception?](#) Y [Catching and Handling Exceptions](#).



Reflexiona

A partir de ahora, **cada vez que implementes un método deberás tener en cuenta si se dan las condiciones para que ese método pueda ejecutarse apropiadamente** o bien si te encuentras ante una situación no deseable o errónea y entonces hay que lanzar una excepción. Es decir, que no bastará simplemente con escribir el código que lleve a cabo ciertas acciones sino que en muchas ocasiones habrá que incluir también sentencias de comprobación (condicionales de tipo `if` o `switch`) para asegurarnos de que la acción puede llevarse a cabo.



Ejercicio Resuelto

El método `repostar` de la clase `Vehículo` sirve para añadir una cantidad determinada de carburante al depósito de combustible del vehículo.

Implementa el método `repostar` de la clase `Vehículo` teniendo en cuenta lo siguiente:

1. se trata de un método que **no devuelve ningún valor** (`void`);
2. tiene **un único parámetro** de tipo real que representa la **cantidad de combustible** que se desea introducir en el depósito;
3. si se intenta **repostar con el motor del vehículo encendido**, se debe **abortar la operación y lanzarse una excepción** de tipo `IllegalStateException` con el siguiente mensaje de error: "se ha intentado repostar con el motor encendido";
4. si la **cantidad que se intenta introducir es negativa**, debería también **lanzarse una excepción**, en este caso de tipo `IllegalArgumentException` con el siguiente mensaje de error "cantidad de combustible inválida: xx", donde xx sería la cantidad negativa que se ha pasado como parámetro, con una precisión de dos decimales;
5. si la **cantidad de combustible que se intenta introducir hace que el depósito rebose** porque no cabe, entonces debería **llenarse completamente del depósito** pero también **lanzar una excepción** de tipo `IllegalStateException` con el siguiente mensaje de error: "cantidad de combustible excesiva, se ha sobrepasado la capacidad del depósito en xx litros", donde xx sería la cantidad de combustible en exceso que se ha intentado introducir con una precisión de dos decimales.



Paul Brennan (Pixabay License)

Si te fijas, la complejidad del método ha aumentado sensiblemente. Ya no se trata simplemente de realizar una suma y asignar esa suma a un atributo. Influyen muchos otros factores que habrá que tener en cuenta.

TEMA 5: DESARROLLO DE CLASES

En este caso se trata de un método que puede lanzar **dos tipos de excepciones diferentes**, por lo que habrá que indicarlo en la cabecera mediante una lista de excepciones separadas por comas.

Además, habrá que tener en cuenta todas las condiciones y posibilidades que se han enumerado en el enunciado. Considerando todas esas posibilidades, el código del método `repostar` podría quedar así:

```
public void repostar (double cantidad) throws IllegalArgumentException, IllegalStateException {
    double capacidadLibre= this.capacidadDeposito - this.NivelDeposito; // Variable local

    if (this.estadoMotor) { // Si el motor está encendido no se puede repostar
        throw new IllegalStateException ("se ha intentado repostar el vehículo con el motor encendido");
    } else if (this.cantidad < 0) { // Si la cantidad de combustible es negativa es un error
        throw new IllegalArgumentException ( String.format ("cantidad de combustible inválida: %.2f", cantidad) );
    } else if (capacidadLibre < cantidad) { // Más combustible del que cabe en el depósito
        this.NivelDeposito= this.capacidadDeposito; // Llenamos el depósito completamente
        throw new IllegalStateException (
            String.format (
                "cantidad de combustible excesiva, se ha sobrepasado la capacidad del depósito en %.2f litros",
                cantidad - capacidadLibre)
        );
    }

    // Si todo ha ido bien (no se ha producido ninguna circunstancia excepcional)
    this.nivelDeposito += cantidad; // Incrementamos el nivel del depósito en la cantidad indicada
}
```

Si no terminas de entender cómo funciona el método `String.format` y los indicadores de formato como por ejemplo `%.2f` es un buen momento para que vuelvas a la **unidad 4** y hagas un repaso del formateo de cadenas con la clase `String`.



Ejercicio Resuelto

El método `parar` de la clase `Vehículo` sirve para detener el motor del vehículo.

Implementa el método `apagar` de la clase `Vehículo` teniendo en cuenta lo siguiente:

1. **no devuelve ningún valor (void);**
2. **no tiene parámetros;**
3. **si se intenta apagar el motor de un vehículo que ya lo tiene parado, se debe abortar la operación y lanzarse una excepción** de tipo `IllegalStateException` con el siguiente mensaje de error: "el motor ya estaba apagado";
4. **si el motor finalmente puede ser apagado, entonces sí debe cambiarse el estado del motor y además no debes olvidar actualizar el atributo de clase que controla la cantidad de vehículos arrancados.**



Pete Linforth (Pixabay License)

[Mostrar retroalimentación](#)

```
public void parar () throws IllegalStateException {
    if (!this.estadoMotor) {
        throw new IllegalStateException ("el motor ya estaba apagado"); // Si el motor está ya apagado, el método finaliza lanzando una excepción
    }
    this.estadoMotor= false; // Si el motor aún estaba encendido, cambiamos su estado a apagado (false)
    Vehiculo.vehiculosArrancados--; // y decrementamos la cantidad de vehículos arrancados en este momento
}
```

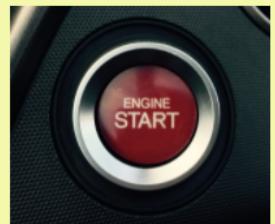


Ejercicio Resuelto

El método `arrancar` de la clase `Vehiculo` sirve para poner en marcha el motor de un vehículo.

Vuelve a implementar el método `arrancar` de la clase `Vehiculo` considerando lo siguiente:

1. **no devuelve ningún valor (void);**
2. **no tiene parámetros;**
3. **si se intenta poner en marcha el motor de un vehículo que ya está arrancado**, debe lanzarse una excepción de tipo `IllegalStateException` pues el motor ya está encendido;
4. **si el depósito de combustible está vacío**, también debe lanzarse una excepción de tipo `IllegalStateException` pues sin combustible no se puede arrancar el motor;
5. **si se trata de un vehículo con el motor parado y con suficiente combustible**, habrá que **cambiar su estado** y actualizar el atributo de clase que controla la **cantidad de vehículos arrancados** en cada momento.



linekslawek (Pixabay License)

```
public void arrancar () throws IllegalStateException {
    if (this.estadoMotor) {
        throw new IllegalStateException ("vehículo ya arrancado"); // El motor ya está arrancado
    }
    if (this.nivelDeposito == 0) {
        throw new IllegalStateException ("depósito de combustible vacío"); // Sin combustible para arrancar
    }
    // Se dan las circunstancias idóneas para arrancar
    this.estadoMotor= true;           // Cambiamos de estado el motor
    Vehiculo.vehiculosArrancados++; // Incrementamos la cantidad de vehículos activos
}
```

5.6 ENCAPSULACIÓN, CONTROL DE ACCESO Y VISIBILIDAD.

Dentro de la **Programación Orientada a Objetos** ya has visto que es muy importante el concepto de **ocultación**, que se logra gracias a la **encapsulación** de la información dentro de las clases. De esta manera una clase puede ocultar parte de su contenido o restringir el acceso a él para evitar que sea manipulado de manera inadecuada. Los **modificadores de acceso** en Java permiten especificar el **ámbito de visibilidad** de los miembros de una clase, proporcionando así un mecanismo de accesibilidad a varios niveles.

Los atributos de una clase suelen ser declarados como **private** (privados a la clase) o como mucho, **protected** (accesibles también por clases heredadas, algo que se verá en unidades posteriores), pero no como **public**. Sólo en algunos casos excepcionales (por ejemplo los atributos estáticos constantes) tendremos atributos públicos.

Podríamos imaginar los atributos de una clase como los distintos componentes por los que fluye la información que hace posible que los objetos de esa clase funcionen correctamente. Pongamos como ejemplo nuestra clase `Vehiculo`. Los objetos de esta clase disponen de una serie de atributos que describen:

- la **naturaleza del objeto** (características inmutables o atributos constantes): capacidad del depósito de combustible, consumo medio, etc.;
- el **estado del objeto** (características o atributos variables): estado del motor, cantidad de combustible, etc.

TEMA 5: DESARROLLO DE CLASES

También se dispone de algunos atributos que son directamente de clase (estáticos) y que representan el **estado de la clase en general** (cantidad total de vehículos creados, número de vehículos con el motor arrancado, etc.).

Toda esa **información almacenada en forma de atributos forma parte de la "maquinaria" interna (o las "tripas) de cada objeto** (o de la clase en general en el caso de atributos estáticos) y no debería ser accesible desde fuera sin cierto control y supervisión. Por eso **se recomienda que el acceso a los atributos se realice siempre a través de métodos**. No debemos dejar los atributos accesibles al exterior salvo casos muy específicos.

La mejor manera de evitar cualquier acceso no deseado a los componentes de un objeto es declarar los atributos como privados y sólo permitir el acceso a ellos a través de métodos.

Para saber más

Puedes echar un vistazo al artículo sobre el control de acceso a los miembros de una clase Java en los manuales de Oracle: [Controlling Access to Members of a Class](#).

5.6.1. MÉTODOS DE ACCESO(I): GETTERS

Como ya sabemos, **un método es un fragmento de código con una cabecera que puede ser invocado desde fuera de la clase y que puede o no llevar parámetros. Desde fuera de la clase lo único que se conoce de cada método es su cabecera**, que es la interfaz de comunicación entre el código externo a la clase y los objetos de esa clase: nombre del método, parámetros, tipo devuelto, etc.

Dado que los métodos son escritos por el programador de la clase, éste puede decidir qué información se va a proporcionar y cuál no. En principio se podría escribir un método para poder obtener el valor de cada uno de los atributos de un objeto, aunque no siempre tiene por qué ser así. Estos métodos son conocidos habitualmente con el nombre de métodos de tipo get (que en inglés significa **obtener**) o **getters**.

Por ejemplo, si tenemos una clase **Persona** con atributos **nombre, apellido1, apellido2, dni**, etc., podríamos definir **getters** para cada uno de esos atributos que son privados y por tanto no visibles desde fuera de la clase:

Si te fijas, lo habitual es que estos métodos no tengan parámetros de entrada, pues no necesitan ninguna información extra para poder devolver la información que se les está solicitando. En este caso no habría necesidad de utilizar la referencia **this**, pues al no haber parámetros no hay posibilidad ninguna de ambigüedad o confusión entre identificadores de atributos e identificadores de parámetros. Pero si decidimos optar siempre por el uso de **this**, podemos hacerlo en aras de una mayor claridad (saber que nos estamos refiriendo a un atributo y no a un parámetro o a una variable local).

En algunos casos se puede considerar interesante poder obtener el valor de varios atributos a la vez como por ejemplo disponer de un método que devuelva una cadena de caracteres con el nombre completo (nombre y apellidos separados por espacios; o bien primero los apellidos separados por espacios, a continuación una coma y luego el nombre). En tal caso podría tener sentido considerarlos como métodos get y llamarlos algo así como **getNombreApellidos** o **getApellidosNombre**:

```
public String getNombreApellidos () {
    return this.nombre + " " + this.apellido1 + " " + this.apellido2;
}

public String getApellidosNombre () {
    return String.format ("%s %s, %s", this.apellido1, this.apellido2, this.nombre);
}
```

```
public class Persona {
    private String nombre;
    private String apellido1;
    private String apellido2;
    private String dni;

    public String getNombre () {
        return this.nombre;
    }

    public String getApellido1 () {
        return this.apellido1;
    }

    public String getApellido2 () {
        return this.apellido2;
    }

    public String getDni () {
        return this.dni;
    }
}
```

TEMA 5: DESARROLLO DE CLASES

Fíjate que hemos elegido dos formas muy distintas de generar la cadena de resultado para que seas consciente de que existen muchas maneras diferentes de llevar a cabo los cálculos (el trabajo con cadenas de caracteres es muy importante en cualquier contexto y debes dominarlo bien).



Ejercicio Resuelto

En nuestro modelo de la clase `Vehículo`, podríamos tener por ejemplo un método llamado `getCapacidadDepósito`, que permitiría obtener la cantidad de combustible que cabe en el depósito del vehículo.

¿Qué otros `getters` crees que podríamos implementar en nuestra clase `Vehículo`?

[Mostrar retroalimentación](#)

Algunos métodos `get` para objetos de la clase `Vehículo` podrían ser:

- ✓ `getCapacidadDepósito`, para obtener la capacidad del depósito del vehículo;
- ✓ `getMatrícula`, para obtener la matrícula del vehículo;
- ✓ `getFechaMatriculacion`, para obtener la fecha de matriculación del vehículo;
- ✓ `getConsumoMedio`, para obtener el consumo medio del vehículo;
- ✓ `isArrancado`, para saber si el vehículo está arrancado (devolverá `true`) o no (devolverá `false`);
- ✓ `isParado`, para saber si el vehículo está parado (devolverá `true`) o no (devolverá `false`). En este caso no se devuelve exactamente el valor de un atributo, sino una transformación del valor de un atributo (`estadoMotor`);
- ✓ `getNivelDepósito`, para obtener la cantidad de combustible que queda en el depósito del vehículo;
- ✓ `getKilometrosTotales`, para obtener la distancia recorrida por el vehículo desde que se fabricó;
- ✓ `getKilometrosParciales`, para obtener la distancia recorrida por el vehículo desde que se arrancó por última vez.

Una vez que has decidido qué métodos `get` podría tener la clase `Vehículo`, ¿cómo los implementarías? Recuerda que para devolver un valor desde un método debes usar la sentencia `return`.

La implementación de estos `getters` quedaría más o menos así:

```
public double getCapacidadDepósito() {
    return this.capacidadDepósito;
}

public String getMatrícula() {
    return this.matrícula;
}

public LocalDate getFechaMatriculacion() {
    return this.fechaMatriculacion;
}

public double getConsumoMedio() {
    return this.consumoMedio;
}

public double isArrancado() {
    return this.estadoMotor;
}

public double isParado() {
    return !this.estadoMotor; // No se devuelve exactamente el valor del atributo sino una transformación
}

public double getNivelDepósito() {
    return this.nivelDepósito;
}

public double getKilometrosTotales() {
    return this.kilometrosTotales;
}

public double getKilometrosParciales() {
    return this.kilometrosParciales;
}
```

TEMA 5: DESARROLLO DE CLASES



Reflexiona:

¿Un getter para cada atributo?

Podría implementarse un getter por cada atributo de la clase, aunque no tiene por qué ser así. Será decisión de los desarrolladores si se desea que pueda accederse en cualquier momento al valor de cualquier atributo desde fuera del objeto o de la clase. Por ejemplo, quizás se disponga de atributos de uso interno cuyo valor sólo interesa al funcionamiento de los propios objetos, pero no son necesarios para interactuar con él desde fuera. En tal caso no sería necesario escribir un getter para esos atributos. Esto será decisión de quien diseñe o implemente la clase. Por eso, aunque algunos atributos sean constantes (y por tanto no habrá "peligro" en que fueran públicos), se sigue recomendando que sean siempre privados pues no todos los atributos tienen por qué ser "visibles".

¿Se te ocurre algún ejemplo de atributo en alguna clase para los que podría interesarnos no disponer de getter o bien que el getter no devolviera exactamente el valor de un atributo sino algún valor calculado a partir de éste?

Veamos algunos ejemplos de atributos para los cuales podría no interesarnos disponer de un getter "directo":

- ✓ imagina que tenemos una clase que representa **figuras geométricas** y que deseamos almacenar el **color** mediante **tres atributos** (intensidad de **rojo**, intensidad de **verde**, intensidad de **azul**). Podríamos entonces disponer de tres atributos de tipo `byte` llamados `red`, `green` y `blue`. Sin embargo quizás a la hora de implementar un método que permita obtener el color de la figura podríamos decidir tener un único método que devuelva un valor que combine esos tres atributos. De esta manera estaríamos devolviendo el valor de tres atributos a la vez mediante un único método. Hay quien llamaría a este valor devuelto un atributo "virtual" o "calculado", pues en realidad no existe en el objeto pero puede calcularse a través de uno o varios atributos de él;
- ✓ supongamos una clase `Bombo` que representa un bombo de un bingo de donde se van extrayendo números. Para ir almacenando los números que ya han salido podría usarse un array para así cada vez que se extraiga un nuevo número aleatorio podamos asegurarnos de que se trata de un número que aún no ha sido extraído. Ese array puede ser un atributo fundamental para el funcionamiento de los objetos `Bombo`, sin embargo es probable no nos interese disponer de un método `get` que devuelva ese array, pues se trata de un elemento de funcionamiento "interno";
- ✓ imagina una clase `Persona` cuyos objetos almacenan en uno de sus atributos un **DNI sin letra** (en forma de número entero) y sin embargo al construir el getter decidimos que también vamos a devolver el número de **DNI** junto con la letra (en forma de cadena de caracteres). Sería un caso en el que no devolvemos exactamente el valor del atributo (sólo los números) sino otro valor calculado a partir del atributo (los números y su letra);
- ✓ también podríamos ver el caso anterior del **DNI** desde otro punto de vista: podría disponerse de un único atributo (**DNI con letra**) y sin embargo de dos métodos diferentes para obtener información parcial de ese atributo: uno para obtener sólo el número y otro para obtener sólo la letra. Todo esto dependerá siempre de la decisión que tomen los diseñadores y programadores en función del uso que se le vaya a dar a la clase en el contexto concreto de una aplicación;
- ✓ un último ejemplo podría ser el de una clase que dispone de un **atributo que almacena un índice que va a ser utilizado como posición de un array**. Lo más habitual es que su valor esté entre 0 y un valor máximo (última posición del array, que será el tamaño del array menos uno). Sin embargo, quizás a la hora de implementar el `get` puede ser que nos interese devolver ese valor desplazado +1 para comenzar desde 1 (hasta el tamaño del array) y así facilitar su uso desde fuera de la clase donde se trabaje desde 1 hasta N y no desde 0 hasta N-1. Desde fuera de la clase no saben si se está utilizando un array o no.

También nos podemos encontrar en ocasiones casos en los que haya más de un getter para un mismo atributo. Ya hemos visto un ejemplo con el atributo `estadoMotor` de la clase `Vehículo`:

- ✓ método `isArrancado`,
- ✓ método `isParado`.

En ambos casos se hace referencia al mismo atributo con la diferencia de que en un caso se devuelve el valor del propio atributo y en otro caso se devuelve una transformación de ese mismo atributo.

Podríamos plantear otro ejemplo con el atributo `nivelDepósito`, para el cual ya disponemos el getter `getNivelDepósito`, pero también podríamos plantarnos un método `getCantidadDepósitoPorLlenar` que devolviera la cantidad del depósito que está libre (lo que ya se ha gastado o está vacío para poder llenar). En tal caso también se estaría trabajando con el atributo `nivelDepósito` (junto con el atributo `capacidadDepósito`):

```
public double getCantidadDepósitoPorLlenar ()  
    return this.capacidadDepósito - this.nivelDepósito;  
}
```

¿Son estos métodos de tipo `get`?

En principio podría considerarse que sí, aunque no devuelvan "exactamente" el valor de un atributo, sino alguna pequeña transformación (o veces grande). Pero también habrá gente que no considere que sea un `get` porque no devuelve estrictamente el valor de un atributo. Será una decisión de diseño y/o implementación el poner un nombre u otro al método.

Por tanto nos encontraremos ocasiones en las que haya quien llame al método `getCantidadDepósitoPorLlenar` (aunque no exista el atributo `cantidadDepósitoPorLlenar`) y quien lo llame por ejemplo `calcularCantidadDepósitoPorLlenar` (o simplemente `cantidadDepósitoPorLlenar`) porque no se está devolviendo el valor de ningún atributo específico sino un cálculo a partir de dos atributos (el nivel del depósito en ese momento y la capacidad del depósito).

¿Qué debemos hacer nosotros en estos casos?

Pues muchas veces dependerá de un criterio de nuestra organización o de nuestro jefe de proyecto y poco podremos decidir. En cualquier caso, si la decisión depende de nosotros lo que habrá que procurar es **ser coherentes y siempre adoptar el mismo criterio** para hacer que el código sea lo más sistemático y uniforme posible. De esa manera facilitaremos todo lo posible su legibilidad.

Como último ejemplo, otros posibles métodos `get` para ese atributo `nivelDepósito` podrían haber sido también `isDepósitoLleno` o `isDepósitoVacio`, que devolverían un `boolean` y cuya implementación consistiría simplemente en comprobar si el valor del depósito es igual a cero o igual a la capacidad máxima.

TEMA 5: DESARROLLO DE CLASES

```
public double isDepositoLleno ()  
    return this.nivelDeposito == this.capacidadDeposito;  
}  
  
public double isDepositoVacio ()  
    return this.nivelDeposito == 0.0;  
}
```

Nuevamente se trata de métodos que no devuelven estrictamente el contenido de un atributo sino un pequeño cálculo a partir de uno o varios atributos. Normalmente se les considera get (o "is" en este caso, dado que devuelven un valor boolean), pero nuevamente es una cuestión que tendréis que discutir en vuestro equipo de desarrollo para hacerlo siempre de la misma manera.

La cuestión fundamental es que no deseamos que se pueda acceder "alegremente" a los atributos de nuestros objetos, sino que se nos tenga que solicitar protocolariamente la información a través de métodos getter que habremos desarrollado nosotros y en los que habremos podido implementar todos los mecanismos de filtro y transformación que consideremos oportunos.

Lo normal es que no haya ningún proceso de transformación y se devuelva el valor original del atributo, pero siempre tendremos la opción de poder hacer el cambio o manipulación que juzguemos apropiado y de implementar sólo aquellos métodos get que se estimemos conveniente.



Recomendación: nomenclatura de los getters

Algunas cuestiones respecto a la **nomenclatura de los getters o métodos get**:

- ✓ **No es obligatorio utilizar este tipo de identificadores (getNombreAtributo)** si no se desea, aunque **es muy recomendable para mantener una manera estándar de llamar siempre del mismo modo a los métodos**. Cuanto más homogénea y estándar sea la nomenclatura más sencilla será la legibilidad del código.
- ✓ En ocasiones **puedes encontrar los nombres de los métodos get en español usando el verbo "obtener"** (por ejemplo obtenerColor en lugar de getColor) o en algún otro idioma si los programadores son nativos de ese otro idioma (por ejemplo obtenirCouleur en francés, o una mezcla como getCouleur).
- ✓ Aquellos casos en los cuales el valor devuelto sea un boolean, en lugar de usar el verbo **to get** ("obtener"), se suele utilizar el verbo **to be** ("ser" o "estar") en tercera persona "**is**" ("es" o "está") para mejorar la legibilidad del código y entender mejor qué es lo que devuelve ese método. Por ejemplo, para un método get del estado del motor de un vehículo podríamos tener **getEstado** pero también **isArrancado**, que indica si el motor está arrancado o no. También en ocasiones se utiliza la nomenclatura "**has**" ("tiene", del verbo **to have**, "tener") para estos casos en los que se devuelve un boolean. Por ejemplo, para una clase **Persona**, podríamos tener un método **hasSiblings**, O **hasHermanos**, O **tieneHermanos**, para indicar si tiene hermanos o no.

En conclusión, podemos tener métodos con nombres como **getNombre** O **getColor**, pero tampoco sería extraño que en alguna aplicación pendas encontrarlos como **obtenerNombre** U **obtenerColor**, aunque **en este curso hemos decidido usar la nomenclatura get, que es casi un convenio de uso generalizado, además de ser bastante abreviada y concisa**.

Lo que sí es muy importante es que una vez decidida una nomenclatura, debemos ser coherentes y usar siempre la misma para todos los métodos de esa clase, e incluso de todo el paquete para facilitar el trabajo a los programadores que vayan a utilizar esas clases.

5.6.2. MÉTODOS DE ACCESO (II): SETTERS

De la misma manera que hemos hablado de métodos **get** o **getter** para obtener el valor de un atributo también podemos hablar de métodos **set** o **setter** para establecer o modificar el valor de un atributo. Estos métodos normalmente no devolverán ningún valor, sino que asignarán un nuevo valor a un atributo (o a un conjunto de atributos).

Podremos escribir un método set por cada atributo que queramos que pueda ser modificado directamente. Por ejemplo, si tenemos una clase **Persona** con atributos **nombre**, **apellido1**, **apellido2**, etc. podríamos definir **setters** para cada uno de esos atributos si se trata de atributos mutables o variables:

TEMA 5: DESARROLLO DE CLASES

```
public class Persona {  
    private String nombre;  
    private String apellido1;  
    private String apellido2;  
    private String dni;  
  
    public void setNombre (String nombre) {  
        this.nombre= nombre;  
    }  
  
    public void setApellido1 (String apellido1) {  
        this.apellido1= apellido1;  
    }  
  
    public void setApellido2 (String apellido2) {  
        this.apellido2= apellido2;  
    }  
  
    public void setDni (String dni) {  
        this.dni= dni;  
    }  
}
```

Si te fijas, en este caso nos ha venido muy bien disponer de la referencia **this** para no tener que utilizar un identificador diferente al del atributo en el parámetro.

Lo habitual en estos métodos **set** es que no devuelvan ningún valor, por tanto serán de tipo **void**, aunque no es algo obligatorio. Por ejemplo en algunos casos hay quien devuelve el valor que ha sido asignado finalmente. En el caso del método **setNombre** podría haber quedado algo así:

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

También podría darse el caso de querer implementar un setter que permita establecer a la vez el nombre y los dos apellidos. Sería un caso de método set en el cual se establece el valor de varios atributos a la vez. Por ejemplo, podría implementarse un método **setNombreApellidos** de la siguiente manera:

```
public void setNombreApellidos (String nombre, String apellido1, String apellido2) {  
    this.nombre= nombre;  
    this.apellido1= apellido1;  
    this.apellido2= apellido2;  
}
```

Otra posibilidad que se nos podría ocurrir es que el método **setNombre** pudiera admitir varios nombres (para el caso de personas con nombre compuesto) y que internamente el método los uniera separándolos por espacios asignándolo finalmente a un único atributo **nombre**. Para ello podríamos modificar el método **setNombre** para que admitiera un número variable de parámetros (tantos nombres como se desea) utilizando el mecanismo **varargs** que ya se mencionó pero del que aún no hemos visto ningún ejemplo. También podríamos dejar el método **setNombre** sin modificar e implementar otro método llamado **setNombreCompuesto** que sería el que podría recibir un número indeterminado de parámetros:

TEMA 5: DESARROLLO DE CLASES

```
public void setNombreCompuesto (String ... nombre) { // Número indeterminado de parámetros String
    String acumuladorNombres= nombre[0]; // Variable local donde iremos concatenando los nombres, empezando por el primero

    for (int i=1; i<nombre.length; i++) {
        acumuladorNombres= acumuladorNombres + " " + nombre[i]; // Concatenamos nombres separados por espacios
    }
    this.nombre= acumuladorNombres; // Asignamos al atributo la cadena creada con todos los nombres concatenados
}
```

Para aquellos atributos que no sean variables, no tiene sentido implementar **setters**. Por ejemplo, para el caso de la clase **Vehiculo**, no tiene sentido dotarla de métodos **set** para los atributos **capacidadDeposito**, **matricula**, **consumoMedio**, o **fechaMatriculacion** pues esos atributos no van a poder ser modificados una vez que se construya el objeto y se les asigne un valor inicial (y definitivo).

Sí podríamos plantearnos en principio implementar setters para atributos como **nivelDeposito** o **kilometrosParciales**. Pero fíjate que, aunque se trata de atributos mutables o variables, no tiene mucho sentido hacerlo, pues si permitimos que desde fuera se pueda asignar un valor "arbitrario" a esos atributos podría desvirtuarse el funcionamiento interno de los objetos.

Por ejemplo, podríamos implementar un método **setNivelDeposito** con un parámetro que fuera la nueva cantidad de combustible del vehículo. Pero los vehículos en realidad no funcionan así, lo que se hace es verter en el depósito una determinada cantidad de combustible haciendo que el nivel del depósito aumente. Para ello tenemos un método **repostar** cuyo funcionamiento es más fiel al comportamiento real de un vehículo. Podríamos decir que **repostar** es una especie de método set dado que su ejecución da lugar a que algunos atributos del objeto cambien de valor, pero no se están cambiando directamente con un valor que se proporciona, sino de manera indirecta mediante cálculos que influyen en el valor final de uno o más atributos.



Recomendación

Si un método no realiza una actualización directa de un atributo con el valor que se pasa como parámetro sino que hay algún cálculo intermedio o se actualizan varios atributos en función de diversos criterios no se le suele poner el nombre de **set**.

Obviamente, esto no es algo obligatorio sino una recomendación y por supuesto será una decisión que deberán tomar quienes diseñen y desarrollen la clase.

5.6.3. MÉTODOS PRIVADOS

Normalmente los métodos de una clase pertenecen a su interfaz y por tanto parece lógico que sean declarados como públicos. Pero también es cierto que pueden darse casos en los que exista la necesidad de disponer de algunos métodos privados a la clase. Se trata de métodos que realizan operaciones intermedias o auxiliares y que son utilizados por los métodos que sí forman parte de la interfaz. Ese tipo de métodos (de comprobación, de adaptación de formatos, de cálculos intermedios, etc.) suelen declararse como privados pues no son de interés (o no es apropiado que sean visibles) fuera del contexto del interior del objeto.

Por ejemplo, imagina la clase **Persona** con la que ya hemos trabajado anteriormente a la cual le vamos a añadir un atributo que contenga el DNI (con letra). Podríamos disponer de un método "interno" que permitiera comprobar que ese DNI es válido. Para ello será necesario calcular la letra correspondiente a un determinado número de DNI y comprobar si una determinada combinación de número y letra forman un DNI válido. Este tipo de cálculos y comprobaciones podrían ser implementados en métodos privados o auxiliares de la clase.

TEMA 5: DESARROLLO DE CLASES

Por ejemplo, podríamos implementar un método llamado **isDniValido** que recibiera como parámetro un DNI y devolviera un **boolean** que indicara si ese DNI es válido en función de la letra que contenga. En tal caso se trataría de un método de comprobación "interno" que sería invocado desde el método **setDni** y no sería necesario que fuera visible desde fuera. Por tanto, el método **isDniValido** tendría sentido que fuera declarado como privado.



Ejercicio Resuelto

Dada la clase **Persona** con los atributos `nombre`, `apellido1`, `apellido2`, `dni`, se desea comprobar que cada vez que se intente asignar (método `set`) un nuevo DNI a un objeto **Persona**, sólo se haga si ese DNI es válido.

Modifica el método `setDni` de la clase **Persona** para que sólo se lleve a cabo la modificación si el DNI que se pasa como parámetro es válido.

Para ver cómo calcular la letra NIF correspondiente a un número de DNI puedes consultar el artículo sobre el NIF de la Wikipedia:

[Artículo en la Wikipedia sobre el Número de Identificación Fiscal \(NIF\).](#)



OpenIcons (Pixabay License)

Podríamos incluir todo el código de comprobación de validez de un DNI dentro del método `setDni`, pero quizás sea más inteligente encapsular esa funcionalidad dentro de un método aparte por si más adelante quisieramos poder utilizarla. A ese método lo podríamos llamar **isDniValido**.

Dado que para comprobar si un DNI es válido habrá que comprobar si la letra que le acompaña es la correcta, quizás también sería buena idea implementar otro método llamado `calcularLetraDni`.

Está claro que para comprobar si un DNI es válido **habrá que comprobar si la letra que le acompaña es la correcta**. Por tanto vamos a necesitar implementar el algoritmo de cálculo la letra de un número de DNI. Para ello podemos crear un método (que en principio también podría ser privado) que realice ese cálculo. Podríamos llamarlo `calcularLetraDni`. Para facilitar la implementación de ese método, crearemos una **cadena de caracteres (String) con las letras posibles que puede tener un DNI en el orden adecuado para la aplicación del algoritmo de cálculo de la letra** (algoritmo conocido como **módulo 23**). Esa cadena será un atributo estático (`static`) y constante (`final`) de la clase **Persona**.

```
private static final String LETRAS_DNI= "TRWAGMYFPDXBNJZSQVHLCKE";
```

A partir de esa cadena será muy sencillo construir un método que implemente el algoritmo módulo23:

```
private char calcularLetraDni (int dni) {
    // Cálculo y devolución de la letra NIF
    return LETRAS_DNI.charAt(numeroDni % 23);
}
```

Una vez que dispongamos del método `calcularLetraDni`, podemos implementar el método `isDniValido` con mayor facilidad:

```
private boolean isDniValido (String dni) {
    boolean valido= true;
    char letraCalculada;
    char letraObtenida;
    int dniNumero;

    if (dni == null) { // El dni debe ser un objeto String no vacío
        valido= false;
    } else if (!dni.matches ("[0-9]{8}[+LETRAS_DNI+]")) {
        valido= false; // No se cumple el patrón de 8 números + 1 letra
    } else {
        // Extraemos la letra (char)
        letraObtenida= dni.charAt(8);
        // Convertimos el DNI a entero
        dniNumero= Integer.parseInt(dni.substring(0,7));
        // Calculamos la letra de DNI (char) a partir del número (int)
        letraCalculada= calcularLetraDni (dniNumero);
        // Si las letras coinciden el DNI es válido.
        valido = letraObtenida == letraCalculada;
    }
    return valido;
}
```

TEMA 5: DESARROLLO DE CLASES

Una vez que disponemos del método `isDniValido`, se trata simplemente de utilizarlo desde el método `setDni`. Sólo se lleva a cabo la modificación si el `DNI` proporcionado es válido:

```
public void setDni (String dni) {  
    if (isDniValido(dni))  
        this.dni= dni;  
}
```

Si queremos proporcionar algo más de información, podemos devolver un `boolean` que indique si se ha podido realizar la modificación del atributo o no:

```
public boolean setDni (String dni) {  
    boolean valido= isDniValido(dni);  
    if (valido)  
        this.dni= dni;  
    return valido;  
}
```

Aunque en realidad lo apropiado en este caso sería hacer saltar una **excepción** indicando que el parámetro es erróneo (`IllegalArgumentException`):

```
public void setDni (String dni) throws IllegalStateException {  
    if (isDniValido(dni))  
        this.dni= dni;  
    else  
        throw new IllegalStateException ("DNI no válido");  
}
```

También veremos más adelante que estos dos nuevos métodos (`isDniValido` y `calcularLetraDni`) quizás deberían hacerse **públicos y de clase (estáticos)** para que pudieran ser utilizados como **herramientas disponibles desde fuera de la clase**. Volveremos a ello cuando empecemos a trabajar con **métodos estáticos**.

5.7 SOBRECARGA DE MÉTODOS.

En principio parece lógico pensar que el nombre de un método sólo puede ser utilizado una vez en la declaración de una clase. Es decir, que una clase no podría tener más de un método con el mismo nombre. Pero no tiene por qué ser siempre así. Es posible tener varias versiones de un mismo método (varios métodos con el mismo nombre) gracias a la **sobrecarga**.

El lenguaje Java soporta la característica conocida como **sobrecarga de métodos**. Ésta permite declarar en una misma clase varias versiones del mismo método con el mismo nombre. La forma que tendrá el compilador de distinguir entre varios métodos que tengan el mismo nombre será mediante la lista de parámetros del método: si el método tiene una lista de parámetros diferente, será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos nombres de método iguales en la misma clase.

Imagínate que estás desarrollando una clase para escribir sobre un lienzo que permite utilizar diferentes tipografías en función del tipo de información que se va a escribir. Es probable que necesitemos un método diferente según se vaya a pintar un número entero (`int`), un número real (`double`) o una cadena de caracteres (`String`). Una primera opción podría ser definir un nombre de método diferente dependiendo de lo que se vaya a escribir en el lienzo.

TEMA 5: DESARROLLO DE CLASES

Por ejemplo:

- método **pintarEntero (int entero)**,
- método **pintarReal (double real)**,
- método **pintarCadena (String cadena)**,
- método **pintarEnteroCadena (int entero, String cadena)**.

Y así sucesivamente para todos los casos que deseas contemplar...

La posibilidad que te ofrece la sobrecarga es utilizar un mismo nombre para todos esos métodos (dado que en el fondo hacen lo mismo: **pintar**). Pero para poder distinguir unos de otros será necesario que siempre exista alguna diferencia entre ellos en las listas de parámetros (bien en el número de parámetros, bien en el tipo de los parámetros, bien en el orden de los tipos de esos parámetros). Volviendo al ejemplo anterior, podríamos utilizar un mismo nombre, por ejemplo **pintar**, para todos los métodos anteriores:

- método **pintar(int entero)**,
- método **pintar(double real)**,
- método **pintar(String cadena)**,
- método **pintar(int entero, String cadena)**.

En este caso el compilador no va a generar ningún error, pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes.

Lo que sí habría producido un error de compilación habría sido por ejemplo incluir otro método **pintar (int otroEntero)**, pues es imposible distinguirlo del método que ya existía, **pintar(int entero)**, con el mismo nombre y con la misma lista de parámetros (ya existe un método **pintar** con un único parámetro de tipo **int**, y que el nombre del parámetro sea distinto no es algo que le permita diferenciarlo en una llamada como por ejemplo, **pintar(3)**, donde lo que tiene no es el nombre del parámetro, sino un valor de tipo **int** (**3** en este caso), que podría encajar en cualquiera de los dos métodos, así que nos deja sin saber a cuál de los dos hay que ligar esa llamada para ejecutar su código.

También debes tener en cuenta que **el tipo devuelto por el método no es considerado a la hora de identificar un método**, así que el tipo devuelto sea diferente no es suficiente para distinguir un método de otro. Es decir, no podrías definir dos métodos exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente. El compilador producirá un error de duplicidad en el nombre del método y no te lo permitirá.

La sobrecarga de métodos es muy útil para evitarnos tener que inventar nombres distintos para métodos que en realidad hacen lo mismo, aunque sea con distintos datos. Fuera de ahí, debes utilizarla con cierta moderación, dado que nombrar igual a métodos que en realidad tienen diferencias sustanciales en cuanto a lo que hacen, podría dificultar la legibilidad del código.



Ejercicio Resuelto

Desamos incluir en nuestra clase `Vehículo` un método que simule que el vehículo ha recorrido una determinada distancia para así hacer que se actualicen los atributos relacionados con el nivel de combustible y los kilómetros recorridos.

Implementa dentro de la clase `Vehículo` un método llamado `recorrerTrayecto` teniendo en cuenta que:

- ✓ Recibe como único parámetro un número real que representa la **cantidad de kilómetros que se van a recorrer**.
- ✓ Si el parámetro es **negativo**, debería lanzarse una excepción de tipo `IllegalArgumentException`.
- ✓ Si el **motor del vehículo está apagado**, el vehículo no debería poder moverse y se produciría una excepción de tipo `IllegalStateException`.
- ✓ Si **no hay combustible suficiente** para completar el trayecto, se debería **recorrer la parte de trayecto a la que se llegue hasta consumir todo el combustible** y a continuación **apagar el motor del vehículo**.
- ✓ El método devolverá el **trayecto efectivo** (en kilómetros) que ha podido realizarse, que no siempre coincidirá con el que se deseaba realizar.
- ✓ Todos los cuentakilómetros deben actualizarse apropiadamente.



Pexels (Pixabay License)

Teniendo en cuenta todos los condicionantes anteriores más cualquier otro efecto colateral que se nos pueda ocurrir, la implementación del método `recorrerTrayecto` podría quedar así:

```
// Método para recorrer un trayecto
double recorrerTrayecto (double distancia) throws IllegalArgumentException, IllegalStateException {
    double combustibleNecesario= distancia / this.consumoMedio; // Combustible necesario para realizar el trayecto
    double distanciaEfectiva= distancia; // Distancia efectiva recorrida por el vehículo

    if (distancia < 0) // Una distancia negativa es un error
        throw new IllegalArgumentException (String.format ("la distancia a recorrer es negativa: %.2f", distancia));
    if (!this.estadoMotor) // Si el motor está apagado no se puede recorrer ningún trayecto
        throw new IllegalStateException ("el motor está apagado");
    if (this.nivelDeposito < combustibleNecesario) { // Si no hay suficiente combustible para completar el trayecto
        distanciaEfectiva= this.nivelDeposito * this.consumoMedio; // Calculamos la distancia efectiva recorrida
        this.estadoMotor= false; // Apagamos el motor del vehículo al consumir todo el combustible
        Vehiculo.vehiculosArrancados--; // Actualizamos la cantidad de vehículos activos en ese momento (se decrementa en uno)
    }

    // Actualizamos los cuentakilómetros del objeto
    this.kilometrosTotales += distanciaEfectiva;
    this.kilometrosParciales += distanciaEfectiva;

    // Actualizamos el cuentakilómetros global de la clase
    Vehiculo.kilometrosTotalesFlota += distanciaEfectiva;

    // Devolvemos la distancia efectiva recorrida
    return distanciaEfectiva;
}
```

Si te fijas, la implementación de los métodos se complica cada vez más, pues hay acciones que tienen efectos colaterales en atributos en los que podríamos no reparar en un primer vistazo. Por ejemplo si el combustible se acaba sin poder completar el trayecto habrá que apagar el motor (atributo **estado del motor**) y si esto sucede habrá que actualizar también el atributo de clase de **número de vehículos arrancados**. Es decir, que no solamente se modifican los atributos de los **kilómetros recorridos** y el **nivel del depósito**. Cada vez que escribas el código de un método debes tener en cuenta todos los detalles de la "maquinaria" de la clase que has inventado para poder implementar apropiadamente su comportamiento.

Imagina que se nos encarga que desarrollemos una segunda versión del método para recorrer un trayecto, pero en esta ocasión con dos parámetros de entrada: el tiempo que se va a estar conduciendo (en horas) y la velocidad media a la que se va a conducir (en km/h).

Implementa esta nueva acción **sobrecargando** el método `recorrerTrayecto`.

TEMA 5: DESARROLLO DE CLASES

Dado que esta nueva versión del método para recorrer un trayecto tendría un número diferente de parámetros, no habría problema de ambigüedad y el compilador de Java no nos pondría ningún problema al implementar otro método que se llamara exactamente igual (`recorrerTrayecto`).

Realmente la única diferencia entre este método y el anterior podría ser únicamente una línea: un cálculo inicial en el que obtenemos la distancia que se desea recorrer a partir del tiempo y la velocidad media de conducción. A partir de ahí el método sería exactamente igual. Por tanto no tendría mucho sentido volver a escribir todo ese código. Lo más razonable en estos casos es aprovechar todo ese código haciendo una **llamada al método homónimo** (con el mismo nombre) desde dentro de nuestro nuevo método:

1. **Se calcula la distancia** a partir del tiempo y la velocidad media de conducción.
2. **Se invoca a la anterior versión del método** `recorrerTrayecto` con esa información (un único parámetro).
3. **Se recoge el valor devuelto por la llamada al método y se devuelve** desde el método mediante una sentencia `return`.

Respecto a los posibles errores que puedan darse, sólo debemos tener en cuenta los específicos de esta versión del método (parámetros negativos) porque de los otros (motor parado, insuficiente combustible, etc.) ya se encargará la versión anterior del método en hacer las comprobaciones necesarias. En tal caso se lanzarían las excepciones apropiadas que llegarían hasta este método. En el momento en que saltara una excepción, la ejecución de este método también finalizaría y se reenviaría esa excepción a quien lo hubiera invocado.

```
// Método para recorrer un trayecto basándose en el tiempo de conducción y la velocidad media
double recorrerTrayecto (double tiempo, double velocidadMedia) throws IllegalArgumentException, IllegalStateException {
    // NOTA: este método lanza las mismas excepciones que su método "sobrecargado"
    // Variables locales
    double distanciaEfectiva= 0.0; // Variable para almacenar la distancia realmente recorrida
    double distancia= tiempo * velocidadMedia; // Cálculo que se recorrería con ese tiempo y velocidad

    // Comprobación de que los parámetros no son negativos
    if (tiempo < 0)
        throw new IllegalArgumentException (String.format ("parámetro inválido: %.2f", tiempo));
    if (velocidadMedia < 0)
        throw new IllegalArgumentException (String.format ("parámetro inválido: %.2f", velocidadMedia));

    // Llamada a la versión anterior de recorrerTrayecto: (método "homónimo")
    distanciaEfectiva= recorrerTrayecto (distancia); // Si se lanza alguna excepción, este método la "relanzará"

    // Devolvemos la distancia efectiva calculada por el método (si es que no se produjo una excepción)
    return distanciaEfectiva;
}
```

De esta manera, dejamos que todo el "trabajo duro" (comprobaciones, actualizaciones, lanzamiento de excepciones, etc.) lo haga el método anterior (que ya habremos implementado y probado adecuadamente) y desde aquí simplemente lo utilizamos calculando previamente lo que necesita como entrada e invocándolo para que nos devuelva una respuesta.

Esta manera de trabajar también es conocida como "*escribelo una vez, úsalo muchas*" ("write it once, use multiple times") y la volveremos a ver cuando sobrecarguemos el constructor.

En realidad ni siquiera harían falta esas variables locales auxiliares. Lo hemos hecho para ilustrar el ejemplo con mayor claridad. El método podría haber quedado así de sencillo y compacto:

```
// Método para recorrer un trayecto basándose en el tiempo de conducción y la velocidad media
double recorrerTrayecto (double tiempo, double velocidadMedia) throws IllegalArgumentException, IllegalStateException {
    // Comprobación de que los parámetros no son negativos
    if (tiempo < 0)
        throw new IllegalArgumentException (String.format ("parámetro inválido: %.2f", tiempo));
    if (velocidadMedia < 0)
        throw new IllegalArgumentException (String.format ("parámetro inválido: %.2f", velocidadMedia));

    return recorrerTrayecto (tiempo * velocidadMedia);
}
```

5.7.1. SOBRECARGA DE OPERADORES.

Del mismo modo que hemos visto la posibilidad de sobrecargar métodos (disponer de varias versiones de un método con el mismo nombre cambiando su lista de parámetros), podría plantearse también la opción de sobrecargar operadores del lenguaje tales como `+`, `-`, `*`, `()`, `<`, `>`, etc. para darles otro significado dependiendo del tipo de operandos con los que vaya a operar.

En algunos casos puede resultar útil para ayudar a mejorar la legibilidad del código, pues esos operadores resultan muy intuitivos y pueden dar una idea rápida de cuál es su funcionamiento.

Un típico ejemplo podría ser el de la sobrecarga de operadores aritméticos como la suma (`+`) o el producto (`*`) para operar con fracciones. Si se definen objetos de una clase **Fraccion** (que contendrá los atributos numerador y denominador) podrían sobrecargarse los operadores aritméticos (habría que redefinir el operador suma (`+`) para la suma, el operador asterisco (`*`) para el producto, etc.) para esta clase y así podrían utilizarse para sumar o multiplicar objetos de tipo **Fraccion** mediante el algoritmo específico de suma o de producto del objeto **Fraccion** (pues esos operadores no están preparados en el lenguaje para operar con esos objetos).

Otro ejemplo podríamos verlo en las clases **LocalDate** y **LocalTime** que disponen de elementos como **isBefore** o **isAfter** para comparar si una fecha o una hora es anterior o posterior a otra (por ejemplo `t1.isBefore(t2)` devuelve `true` si `t1` es anterior a `t2`). En este caso habría estado bien poder sobrecargar los operadores `<` y `>` para poder comparar fechas u horas directamente con esos operadores en lugar de usando métodos. El ejemplo anterior podría entonces quedar como `t1 < t2`, que sería bastante más compacto e intuitivo que `t1.isBefore(t2)`.

En algunos lenguajes de programación como por ejemplo C++ o C# se permite la sobrecarga, pero no es algo soportado en todos los lenguajes.

¿Qué sucede en el caso concreto de Java?

El lenguaje Java NO soporta la sobrecarga de operadores.

En el ejemplo anterior de los objetos de tipo **Fraccion**, habrá que declarar métodos en la clase **Fraccion** que se encarguen de realizar esas operaciones, pero no lo podremos hacer sobrecargando los operadores del lenguaje (los símbolos de la suma, resta, producto, etc.). Por ejemplo:

```
public Fraccion sumar (Fraccion sumando)
public Fraccion restar (Fraccion sustraendo)
public Fraccion multiplicar (Fraccion multiplicador)
public Fraccion dividir (Fraccion divisor)
```

Y así sucesivamente...

Y lo mismo sucede con las clases **LocalDate** y **Localtime**. Por ello se han tenido que implementar métodos como **isAfter** o **isBefore**.

Dado que en este módulo se está utilizando el lenguaje Java para aprender a programar, no podremos hacer uso de esta funcionalidad. Más adelante, cuando aprendas a programar en otros lenguajes, es posible que sí tengas la posibilidad de utilizar este mecanismo.

5.8 MÉTODOS ESTÁTICOS.

Un **método estático** puede ser usado directamente desde la clase sin necesidad de hacer referencia a ninguna instancia. De hecho, son métodos que podrían invocarse sin necesidad de que existiera ningún objeto instancia de la clase.

Estos métodos también son conocidos como **métodos de clase** (como sucedía con los **atributos de clase**), frente a los **métodos de objeto** (es necesario un objeto para poder invocarlos).

Los métodos estáticos no pueden manipular atributos de instancias (objetos), sólo atributos estáticos (de clase) y suelen ser utilizados para realizar operaciones comunes a todos los objetos de la clase, más que para una instancia concreta.

Algunos ejemplos de operaciones que suelen realizarse desde métodos estáticos:

- acceso a atributos específicos de clase: incremento o decremento de contadores internos de la clase (no de instancias), métodos getter de acceso a atributos de clase, etc.;
- operaciones genéricas relacionadas con la clase pero que no utilizan atributos de instancia. Por ejemplo, para la clase **Persona** hemos definido anteriormente métodos para calcular y comprobar la letra de un DNI. Este método podría ser interesante hacerlo público para poder usarlo desde fuera de la clase;
- herramientas de cálculo auxiliar que reciben parámetros, llevan a cabo operaciones con esos parámetros y devuelven el resultado obtenido.

En la biblioteca de Java es muy habitual encontrarse con clases que proporcionan métodos estáticos que pueden resultar muy útiles para cálculos auxiliares, conversiones de tipos, etc. Por ejemplo, la mayoría de las clases del paquete **java.lang** que representan tipos (**Integer**, **String**, **Float**, **Double**, **Boolean**, etc.) ofrecen métodos estáticos para hacer conversiones. Aquí tienes algunos ejemplos:

- **static String valueOf(int i)**. Devuelve la representación en formato **String** (cadena) de un valor **int**. Se trata de un método que no tiene que ver nada en absoluto con instancias concretas de **String**, es más bien un método auxiliar que puede servir como herramienta para ser usada desde otras clases. Se utilizaría directamente con el nombre de la clase. Por ejemplo:

```
String enteroCadena= String.valueOf (23);
```

- **static String valueOf(float f)**. Algo similar para un valor de tipo **float**. Ejemplo de uso:

```
String floatCadena= String.valueOf (24.341f);
```

- **static int parseInt (String s)**. En este caso se trata de un método estático de la clase **Integer**. Analiza la cadena pasada como parámetro y la transforma en un **int**. Ejemplo de uso:

```
int cadenaEntero= Integer.parseInt ("-12");
```

TEMA 5: DESARROLLO DE CLASES

Todos los ejemplos anteriores son casos en los que se utiliza directamente la clase como una especie de **caja de herramientas** que contiene métodos que pueden ser utilizados desde cualquier parte, por eso suelen ser métodos públicos, además de estáticos.

Un ejemplo muy habitual en Java de **clase que sirve como caja de herramientas** es la clase **Math**, de la cual no tenemos objetos instancias de esa clase y sin embargo usamos muchos de sus métodos estáticos para realizar operaciones matemáticas como por ejemplo:

```
static double abs(double a) // Calcula y devuelve el valor absoluto del parámetro
static double cos(double a) // Calcula y devuelve el coseno trigonométrico del parámetro
static double log(double a) // Calcula y devuelve el logaritmo neperiano del parámetro
static double max(double a, double b) // Calcula y devuelve el máximo de los dos parámetros
static double pow(double a, double b) // Calcula y devuelve el valor del primer parámetro elevado al segundo
static double sqrt(double a) // Calcula y devuelve la raíz cuadrada del parámetro
static double random() // Devuelve un valor aleatorio mayor o igual que 0.0 y menor que 1.0
```

Para saber más

Puedes echar un vistazo a algunas clases del paquete **java.lang** (por ejemplo **Integer**, **String**, **Float**, **Double**, **Boolean** y **Math**) y observar la gran cantidad de métodos estáticos que ofrecen para ser utilizados sin necesidad de tener que crear objetos de esas clases: [Package java.lang](#).



Ejercicio Resuelto

La obtención del valor de atributos **estáticos** debería realizarse mediante **métodos getter estáticos**. No tiene mucho sentido que los métodos sean de objeto, pues no se va a acceder a ningún atributo de objeto.

Teniendo eso en cuenta, implementa los métodos de la clase **Vehiculo**:

- ✓ getVehiculosCreados
- ✓ getVehiculosArrancados
- ✓ getKilometrosTotalesFlota



Vigan Hajdar | Pixabay License

[Mostrar retroalimentación](#)

Dado que esos métodos únicamente acceden a atributos estáticos, lo razonable es que sus getter sean métodos **estáticos**:

```
public static short getVehiculosCreados() {
    return Vehiculo.vehiculosCreados;
}

public static short getVehiculosArrancados() {
    return Vehiculo.vehiculosArrancados;
}

public static double getKilometrosTotalesFlota() {
    return Vehiculo.kilometrosTotalesFlota;
}
```



Ejercicio Resuelto



OpenClipart-Vectors (Pixabay License)

Anteriormente se implementaron para la clase `Persona` sendos métodos `isDniValido` y `calcularLetraDni`, que se utilizaron como herramientas auxiliares a la hora de asignar un nuevo `DNI` a un objeto `Persona` (método `setDni`).

En su momento esos métodos se ocultaron en el interior de la clase pues se trataba de herramientas "internas" que resultaban de utilidad para otros métodos.

Una vez que hemos entendido el funcionamiento de los métodos estáticos podemos observar que esos dos métodos son claros candidatos a ser métodos estáticos de la clase `Persona`: **no utilizan ningún atributo de objeto en su interior**. Además, se ha llegado a la conclusión de que también podría ser interesante su uso desde fuera de la clase, pues proporcionan una herramienta muy útil para trabajar con números de `DNI`.

Como conclusión, podríamos decir que esos dos métodos podrían ser declarados como públicos y estáticos.

Modifica los métodos `isDniValido` y `calcularLetraDni` de la clase `Persona` para que sean **de clase (estáticos)** y además puedan ser utilizados desde cualquier elemento de un programa que use la clase `Persona` y no sólo desde dentro de la clase. Además, aprovecha para sobrecargar el método `calcularLetraDni` para que también pueda recibir como parámetro un `String` en lugar de un `int`.

Para que los métodos `isDniValido` y `calcularLetraDni` sean de clase y accesibles desde cualquier parte del programa habrá que declararlos como **estáticos (static)** y **públicos (public)**:

```
public static char calcularLetraDni (int dni)
public static boolean isDniValido (String dni)
```

El cuerpo de los métodos quedaría exactamente igual.

Respecto a sobrecargar el método `calcularLetraDni` recibiendo como parámetro un `String` en lugar de un `int`, podemos aprovechar el método homónimo (con el mismo nombre) ya implementado como hemos hecho en otras ocasiones:

```
public static char calcularLetraDni (String dni) {
    char resultado=0;
    if (dni == null) { // El dni debe ser un objeto String no vacío
        resultado= 0;
    }else{
        // Convertimos el String a entero y calculamos su letra
        try { // Llamamos al método "homólogo" pasándole un int
            resultado= calcularLetraDni (Integer.parseInt(dni));
        } catch (NumberFormatException ex) {
            resultado=0; // Si el String no contiene un número
        }
    }
    return resultado;
}
```

5.8.1. MÉTODO MAIN EN JAVA.

En el lenguaje Java, toda aplicación debe contener al menos una clase con un método estático `main` cuya cabecera es:

```
public static void main(String[] args)
```

Normalmente suele utilizarse como **punto de inicio de la ejecución del programa** a partir del cual se comenzarán a crear objetos instancias de clases y a ejecutar métodos. Suele incluirse en una clase que normalmente no se instancia y es habitualmente conocida como la clase de inicio del programa. Se trata de un método similar a la función `main` que existe en lenguajes como C o C++.

Este método es **estático** y puede ser invocado sin necesidad de que exista un objeto instancia de la clase. El único **parámetro de entrada** que posee (`args`, un **array de cadenas** de tipo `String`) es el mecanismo a través del cual el sistema pasa información desde la consola (línea de órdenes o "*command line*") hacia la aplicación

TEMA 5: DESARROLLO DE CLASES

(argumentos o parámetros que acompañan al escribir el nombre de la clase en la consola). Eso significa que si se lanza el programa desde consola de la siguiente manera:

```
java NombreClase uno dos tres 0 1 2
```

El array **args** contendría la siguiente información:

Contenido del array `String[] args`

Posición	0	1	2	3	4	5
Contenido	uno	dos	tres	0	1	2

Aquí tienes un ejemplo de uso el método **main** y su ejecución enviándole seis parámetros:

```
public class PruebaMain {  
  
    public static void main(String[] args) {  
  
        System.out.printf ("Prueba de ejecución del método main.\n");  
        for (int i=0; i< args.length; i++) {  
            System.out.printf ("Argumento %d: %s\n", i, args[i]);  
        }  
    }  
}
```

Si ejecutáramos esta clase (podemos hacerlo dado que contiene un método **main**) con los parámetros anteriores:

```
java PruebaMain uno dos tres 0 1 2
```

Se obtendría el siguiente resultado por pantalla:

Prueba de ejecución del método main.

Argumento 0: uno

Argumento 1: dos

Argumento 2: tres

Argumento 3: 0

Argumento 4: 1

Argumento 5: 2

En algunos casos podemos incluir un método **main** en nuestras clases (y no en una clase "principal" de comienzo de ejecución del programa) por si queremos probar el funcionamiento de los métodos de la clase sin tener que implementar una clase de prueba adicional que incluya ese método **main**. No es lo habitual, pero puedes encontrarlo con cierta frecuencia. Nosotros lo haremos en alguna ocasión.

5.9 MÉTODO `toString` EN JAVA.

En Java, toda clase, por el hecho de ser una clase Java, incluye el método `toString` (esto es porque toda clase Java hereda siempre de la clase **Object**, algo que veremos en la siguiente unidad cuando estudiemos la herencia).

El propósito de este método es **asociar a todo objeto un texto representativo de su contenido**. Es decir, que se trata de un método que devuelve una representación "textual" del contenido del objeto, de forma que en cualquier sentencia o expresión donde se espere un **String**, y se encuentre un objeto, intentará sustituirlo por su representación como **String**, siguiendo para ello las "instrucciones" del método `toString`. Ahora bien, ¿qué formato debe tener esa representación textual? Esa es una cuestión que deberá decidir el programador.

No es obligatorio implementar el método `toString`. Toda clase Java ya dispone de ese método implícitamente. Pero si nosotros no lo reescribimos, el resultado textual que devuelve es muy "pobre" y tendrá un aspecto parecido a algo como `<nombreClase>@<numero>` (por ejemplo `Persona@1af7ab3`). Esa información devuelta es conocida como "hash" y está relacionada con la posición de memoria en la que se almacena el objeto. Si deseamos que la representación textual de nuestro objeto sea más explícita y detallada tendremos que implementar nosotros el método.

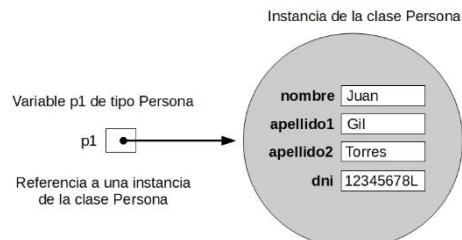
Veamos un ejemplo con la clase **Persona**. Si quisiéramos que la representación textual del objeto de tipo **Persona** en la que se obtuviera un texto con el siguiente formato:

```
{Nombre: xxx, Apellido1: yyy, Apellido2: zzz, Dni: vvv}
```

donde xxx, yyy, zzz, vvv son los valores contenidos en los atributos **nombre**, **apellido1**, **apellido2**, **dni** respectivamente, podríamos implementar un método `toString` de la siguiente manera, para que nos mostrara los datos de la persona en una línea, encerrados entre llaves, con una etiqueta indicando el nombre del atributo seguido de dos puntos, y separados por comas:

```
public String toString() {
    return String.format ("{Nombre: %s, Apellido1: %s, Apellido2: %s, Dni: %s}",
        this.nombre,
        this.apellido1,
        this.apellido2,
        this.dni);
}
```

Si, por ejemplo, el contenido de los atributos de un objeto instancia de la clase **Persona** al que apunta una variable **p1** fuera:



La ejecución de la siguiente línea `System.out.printf ("p1= %s", p1.toString());` mostraría por pantalla lo siguiente:

```
p1= {Nombre: Juan, Apellido1: Gil, Apellido2: Torres, Dni: 12345678L}
```

Para obtener la representación textual (salida del método `toString`) de un objeto no es necesario llamar explícitamente al método `toString`. Por ejemplo, en el caso anterior podrías haber escrito directamente `System.out.printf ("p1: %s", p1)`. Es decir, que **si vamos a interpretar una variable referencia (objeto) como un String, no es necesario invocar explícitamente al método `toString`** pues Java lo hará implícitamente por nosotros.

TEMA 5: DESARROLLO DE CLASES

Si deseamos que el formato de salida sea, por ejemplo, con una línea para cada atributo, podríamos hacer entonces:

```
public String toString() {  
    return String.format ("Nombre: %s\nApellido1: %s\nApellido2: %s\nDni: %s\n",  
        this.nombre,  
        this.apellido1,  
        this.apellido2,  
        this.dni);  
}
```

En este caso la salida por pantalla de `System.out.printf ("p1: %s", p1);` tendría el siguiente aspecto:

p1:

Nombre: Juan

Apellido1: Gil

Apellido2: Torres

Dni: 12345678L

Como puedes observar, el formato de salida del método `toString` puede ser tan elaborado como tú decidas a la hora de implementarlo. Por ejemplo, puedes decidir:

- qué atributos deseas que se muestren (no tienes por qué mostrar todos los atributos),
- número de decimales para los números reales,
- cómo interpretar el valor de los atributos `boolean`,
- longitud máxima para las cadenas de caracteres,
- si quieres que cada atributo aparezca en una línea diferente o todos en la misma línea,
- si quieres incluir cálculos adicionales.



Ejercicio Resuelto

Desde el equipo de diseño nos informan de que se ha decidido que el método `toString` de la clase `Vehiculo` muestre únicamente algunos atributos de estado del objeto (en concreto serían **tres atributos**: si el motor está apagado o encendido, la cantidad de combustible disponible en el depósito y la cantidad de kilómetros realizados desde que se ha arrancado el vehículo por última vez).

Se desea además que la salida tenga el siguiente formato:

{Motor: [encendido/apagado], Combustible: <nivel del depósito> litros, Kilómetros: <kilómetrosParciales>}

Además, en el caso de los números reales (combustible en litros y distancia en kilómetros) se desea que aparezcan únicamente **dos cifras decimales**.

Un ejemplo de salida para un vehículo con el motor encendido, con 20,55 litros de combustible en el depósito y 352 kilómetros recorridos hasta el momento podría ser algo así:

{Motor: encendido, Combustible: 20,55 litros, Kilómetros: 352,00}



[OpenClipart-Vectors \(Pixabay License\)](#)

Implementa un método `toString` para la clase `Vehiculo` teniendo en cuenta todo lo anterior.

Teniendo en cuenta que los atributos que deseamos representar textualmente mediante el método `toString` son:

1. `estadoMotor`
2. `nivelDeposito`
3. `kilometrosParciales`

la implementación podría quedar así:

```
public String toString() {  
    return String.format ("{Motor: %s, Combustible: %.2f litros, Kilómetros: %.2f}",  
        this.estadoMotor ? "encendido" : "apagado",  
        this.nivelDeposito,  
        this.kilometrosParciales);  
}
```

6.CONSTRUCTORES.

Como ya has estudiado en unidades anteriores, en el ciclo de vida de un objeto se pueden distinguir las fases de:

- construcción del objeto;
- manipulación y utilización del objeto accediendo a sus miembros;
- destrucción del objeto.

Durante la fase de construcción o instanciación de un objeto es cuando se reserva espacio en memoria para sus atributos y se inicializan algunos de ellos. Un **constructor** es un método especial con el **mismo nombre de la clase** y que se encarga de realizar este proceso.

El proceso de creación de un objeto se lleva a cabo mediante el **operador new**, que realiza una llamada a un constructor de la clase a la que pertenece el objeto y devuelve una referencia a una zona de memoria donde se encuentra el objeto que se acaba de crear (esto es, un conjunto de bytes con los valores de los atributos de ese objeto). Ha llegado el momento de aprender a implementar tus propios constructores.

Los métodos constructores se encargan de llevar a cabo el proceso de creación o construcción de un objeto.

6.1 CONCEPTO DE CONSTRUCTOR.

Un **constructor** es un método que tiene el mismo nombre que la clase a la que pertenece y que no devuelve ningún valor explícitamente tras su ejecución. Su función es la de proporcionar el mecanismo de creación de instancias (objetos) de la clase, siguiendo para ello el "molde" que define esa clase.

Cuando una variable referencia a objeto es declarada, en realidad aún no existe ningún objeto. La declaración sólo crea la referencia que más tarde apuntará al objeto. Tan solo se trata de un nombre simbólico (una variable) que en el futuro apuntará a una zona de memoria que contendrá la información que representa realmente a un objeto (el conjunto de sus atributos). Para que esa variable referencia aún "vacía" (se suele decir que es una referencia nula o vacía o que apunta a **null**) apunte a una zona de memoria que represente a una instancia de clase (objeto) existente, es necesario "**construir**" o "crear" el objeto. Ese proceso se llevará a cabo a través del método **constructor** de la clase.

Por tanto para instanciar un nuevo objeto es necesario realizar una llamada a un método constructor de la clase a la que pertenece ese objeto. Ese proceso se realiza mediante la utilización del operador **new**.

Hasta el momento ya has utilizado en numerosas ocasiones el operador **new** para instanciar o crear objetos. En realidad lo que estabas haciendo era una llamada al constructor de la clase para que reservara memoria para ese objeto y por tanto "crear" físicamente el objeto en la memoria (dotarlo de existencia física dentro de la memoria del ordenador). Dado que en esta unidad estás ya definiendo tus propias clases, parece que ha llegado el momento de que empieces a escribir también los constructores de tus clases.

Por otro lado, si un constructor es al fin y al cabo una especie de método (aunque algo especial) y Java soporta la sobrecarga de métodos, podrías plantearte la siguiente pregunta: ¿podrá una clase disponer de más de constructor? En otras palabras, ¿será posible la sobrecarga de constructores? La respuesta es afirmativa.

Una misma clase puede disponer de varios constructores. Los constructores soportan la sobrecarga.

Es necesario que **toda clase tenga al menos un constructor**. Si no se define ningún constructor en una clase, el compilador creará por nosotros un **constructor por omisión (o por defecto)** vacío y sin parámetros que se

TEMA 5: DESARROLLO DE CLASES

encarga de inicializar todos los atributos a sus valores por omisión (0 para los numéricos y el tipo **char**, **false** para los **boolean**, **null** para las referencias.). Que lo cree por nosotros no quiere decir que en nuestro código fuente aparezcan añadidas nuevas líneas de código escritas en Java para que podamos verlo, sino que lo genera en el código compilado. Por tanto, aunque no lo veamos, estará disponible. A este constructor se le suele llamar **default constructor** (o constructor por omisión, o constructor por defecto).

Algunas analogías que podrías imaginar para representar el constructor de una clase podrían ser:

- los moldes de cocina para flanes, galletas, pastas, etc.,
- un cubo de playa para crear castillos de arena,
- un molde de un lingote de oro,
- una bolsa para hacer cubitos de hielo.

Se trata del "fabricante" que rellena el molde de la clase para crear un objeto real con contenido.

6.2 IMPLEMENTACIÓN DE CONSTRUCTORES.

Cuando se escribe el código de una clase normalmente se pretende que los objetos de esa clase se instancien de una determinada manera. Para ello se definen uno o más constructores en la clase. En la declaración de un constructor se indican:

- el tipo de acceso,
- el nombre de la clase (el nombre de un método constructor es siempre el nombre de la propia clase),
- la lista de parámetros que puede aceptar,
- si lanza o no excepciones,
- el cuerpo del constructor (un bloque de código como el de cualquier método).

Como puedes observar, la estructura de los constructores es similar a la de cualquier método, con las excepciones de que **no tiene tipo de dato devuelto**, ni siquiera **void** (no devuelve ningún valor explícitamente con la sentencia **return**), y que **el nombre del método constructor debe ser obligatoriamente el nombre de la clase**.



Reflexiona

Si al implementar una clase en Java no se le proporciona a ésta ningún constructor, el compilador le añade un **constructor por omisión sin parámetros** (también conocido como **constructor por defecto** o **default constructor**) para poder utilizar el operador **new** con algún constructor a la hora de instanciar objetos de esa clase.

Ahora bien, si defines constructores personalizados para esa clase, el constructor por omisión dejará de ser generado automáticamente por el compilador y ya no dispondrás de él. Si entre los constructores implementados por ti, no hay ninguno que no tenga parámetros (como sucede con el constructor por omisión), ya no habrá disponible un constructor sin parámetros. Tendrás que implementarlo tú.

Por tanto, si se ha implementado un constructor (o varios) con parámetros y no se ha implementado un constructor sin ningún parámetro, el intento de utilización de un constructor sin parámetros producirá un error de compilación. El constructor por omisión con una lista de parámetros vacía sólo se añade cuando nosotros no proporcionamos ningún constructor. A partir del momento en que proporcionemos alguno ya no se añadirá.

Un ejemplo de constructor básico para la clase **Persona** podría ser:

```
public Persona (String nombre, String apellido1, String apellido2, String dni) {  
    // Asignación inicial de valores a los atributos  
    this.nombre= nombre;  
    this.apellido1;  
    this.apellido2;  
    this.dni= dni;  
}
```

TEMA 5: DESARROLLO DE CLASES

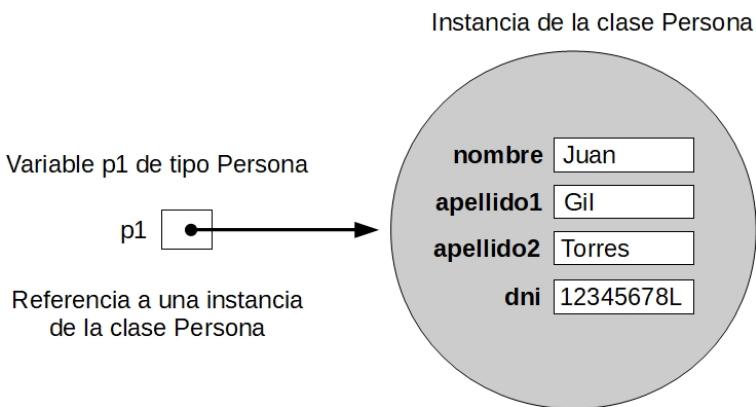
La diferencia fundamental con un método es que no devuelve ningún tipo (ni siquiera **void**) y que se llama exactamente igual que la clase (**Persona** en este caso). Por tanto tampoco tiene sentido que incluya en su interior ninguna sentencia **return**.

En este caso el constructor recibe cuatro parámetros. Además de reservar espacio para los atributos (de lo cual se encarga automáticamente Java), también asigna valores iniciales a los atributos **nombre**, **apellido1**, **apellido2**, y **dni**.

Desde un programa que usara esta clase podría invocarse al constructor de la siguiente manera:

```
Persona p1;  
p1 = new Persona ("Juan", "Gil", "Torres", "12345678L");
```

La invocación a ese constructor es la que dará lugar a la instanciación de un objeto **Persona**, devolviendo el operador **new** una referencia a una zona de memoria que es asignada a la variable **p1**.



A partir de ese momento podemos considerar la variable **p1** como una referencia a un objeto instancia de la clase **Persona** sobre la cual se podrá invocar a cualquiera de sus métodos de objeto. Pero observa bien que aunque a veces podamos llamar a la variable **p1** con el término "objeto", en realidad no contiene el objeto, sino una referencia (una posición de memoria) que indica dónde se encuentra el objeto o instancia en memoria.

El constructor por omisión (o por defecto) no se ve en el código de una clase, a pesar de lo cual podrías usarlo. Lo incluirá el compilador de Java al compilar la clase si descubre que no se ha creado ningún método constructor para esa clase. No lo creará si ya has añadido explícitamente cualquier otro constructor.

6.3 LANZANDO EXCEPCIONES DESDE LOS CONSTRUCTORES.

De la misma manera que hemos lanzado **excepciones** desde métodos cuando se producían **situaciones no deseadas o erróneas** interrumpiendo la ejecución del método, también puede hacerse desde un constructor. Esto haría que detuviera bruscamente la ejecución del constructor y evitaría que se finalizara el proceso de construcción. De esta forma se impediría que llegara a instanciarse el objeto devolviéndose el control de la ejecución a lugar desde donde se invocó al constructor.

¿En qué casos puede ser conveniente impedir que se intance un objeto? Imagina, por ejemplo que los parámetros que se han pasado al constructor no son apropiados por no cumplir alguna condición. En tal caso no debería poder crearse un objeto con atributos inválidos. Es una manera de garantizar que si el objeto ha sido creado es porque ha sido posible hacerlo con todas las garantías.

Si volvemos a nuestro ejemplo anterior de la clase **Persona**, podríamos por ejemplo evitar que se lleve a cabo la instancia si los valores que se pasan como parámetros no son adecuados:

- ninguno de los parámetros **nombre**, **apellido1**, **apellido2**, **dni** debería **null** o vacío (cadena vacía),
- el parámetro **dni**, además, debería contener un DNI válido (ocho números más una letra que corresponda a esos números).

Por tanto, el constructor no va a consistir simplemente en la asignación de valores a atributos, sino que en muchas ocasiones también contendrá comprobaciones de validez y coherencia como ya hemos hecho en muchos otros métodos.

Teniendo en cuenta todo eso, un **constructor con cuatro parámetros** para nuestra clase **Persona** podría quedar de la siguiente manera:

```
public Persona (String nombre, String apellido1, String apellido2, String dni) throws IllegalArgumentException {
    // Comprobación de que los valores de entrada son válidos
    if (nombre==null || apellido1==null || apellido2==null || dni==null) { // Comprobación de valores no nulos
        throw new IllegalArgumentException ("Alguno de los parámetros de entrada es null");
    } else if (nombre.isEmpty() || apellido1.isEmpty() || apellido2.isEmpty() || dni.isEmpty()) { // Comprobación valores no vacíos
        throw new IllegalArgumentException ("Alguno de los parámetros de entrada es una cadena vacía");
    } else if (!isDniValido(dni)) { // Comprobación de que el DNI es válido
        throw new IllegalArgumentException ("el DNI no es válido");
    }

    // Una vez que se ha garantizado que todos los valores de entrada son apropiados, puede continuar el proceso de instancia
    // del objeto

    // Asignación de valores iniciales a los atributos a partir de los parámetros de entrada recibidos
    this.nombre= nombre;
    this.apellido1;
    this.apellido2;
    this.dni= dni;
}
```

Si quisieramos ser más específicos a la hora de indicar el error que se ha producido, siempre podemos tratar cada comprobación por separado y generar un texto de error que proporcione información más concreta. El precio que tendremos que pagar será escribir más líneas de código para comprobar cada posible error por separado. Por ejemplo:

TEMA 5: DESARROLLO DE CLASES

```
public Persona (String nombre, String apellido1, String apellido2, String dni) throws IllegalArgumentException {  
    // Comprobación de que los valores de entrada son válidos  
    if (nombre==null || nombre.isEmpty())  
        throw new IllegalArgumentException ("el parámetro nombre es null o contiene la cadena vacía");  
    if (apellido1==null || apellido1.isEmpty())  
        throw new IllegalArgumentException ("el parámetro apellido1 es null o contiene la cadena vacía");  
    if (apellido2==null || apellido2.isEmpty())  
        throw new IllegalArgumentException ("el parámetro apellido2 es null o contiene la cadena vacía");  
    if (dni==null || dni.isEmpty())  
        throw new IllegalArgumentException ("el parámetro dni es null o contiene la cadena vacía");  
    if (!isDniValido(dni))  
        throw new IllegalArgumentException ("el DNI no es válido");  
    ...  
}
```

Por otro lado, también podría decidirse lo contrario: devolver siempre el mismo mensaje de error en caso de que algún parámetro no sea apropiado (un texto de error del estilo "*parámetros no válidos*"). Por ejemplo:

```
public Persona (String nombre, String apellido1, String apellido2, String dni) throws IllegalArgumentException {  
    // Comprobación de que los valores de entrada son válidos  
    if (nombre==null || nombre.isEmpty() || apellido1==null || apellido1.isEmpty() || apellido2==null || apellido2.isEmpty() ||  
        dni==null || dni.isEmpty() || !isDniValido(dni)) {  
        throw new IllegalArgumentException ("parámetros no válidos");  
    }  
}
```

Todo dependerá de lo "finos" que queramos ser a la hora de indicar por qué se ha producido un error. Cuanto más específicos seamos más información se proporcionará a quien use el constructor y mayor será la complejidad de las líneas de código del cuerpo del constructor. Cuanto menos específicos seamos, menos información proporcionaremos sobre el error y el código del constructor quedará más simple. Deberemos decidir en cada caso qué es lo más ventajoso para el uso de la clase por parte de otro programador.

En cualquier caso, a partir de ahora, **cada vez que intentemos instanciar un objeto de la clase Persona deberíamos hacerlo dentro de un bloque try - catch** para poder capturar los posibles errores que se pudieran producir durante el proceso de construcción o instantiación:

```
Persona p1;  
try (IllegalArgumentException ex) {  
    p1= new Persona ("Juan", "Gil", "Torres", "12345678L");  
    System.out.printf ("Objeto Persona creado y referenciado desde la variable p1.\n");  
    System.out.printf ("Contenido referenciado por p1: %s.\n", p1.toString());  
} catch () {  
    System.out.printf ("Error al intentar instanciar un objeto persona: %s.\n", ex.getMessage());  
}
```



Ejercicio Resuelto

En la clase `Vehiculo`, cada vez que se intenta crear un nuevo objeto instancia de esta clase, éste comienza con todos sus atributos de estado (atributos de objeto variables) a "cero" (nivel del depósito, kilómetros realizados, estado apagado, etc.). Sin embargo, aquellos atributos que permanecerán inmutables (constantes) durante toda la vida del objeto (capacidad del depósito, consumo medio, matrícula, fecha de matriculación) deberán ser asignados a un determinado valor. Esos valores deberán ser proporcionados al constructor de la clase a través de sus parámetros.



Se desea implementar un constructor para clase `Vehiculo` que reciba cuatro parámetros: **matrícula, fecha de matriculación, capacidad del depósito de combustible, consumo medio**.

Debes tener en cuenta que esos parámetros deben cumplir los siguientes requisitos:

[Clicker-Free-Vector-Images \(Pixabay License\)](#)

- ✓ la **matrícula** debe seguir el patrón "NNNNLLL", es decir cuatro números y tres letras mayúsculas. Además, debes tener en cuenta que no se permiten vocales y que las consonantes que se utilizan son: B, C, D, F, G, H, J, K, L, M, N, P, R, S, T, V, W, X, Y y Z. Se excluyen la Ñ y la Q por posible confusión con la N y la O;
- ✓ la **fecha de matriculación** debe estar entre el año 1920 y la fecha actual;
- ✓ la **capacidad del depósito** debe estar dentro del rango permitido por las constantes de clase que definen la máxima y la mínima capacidad;
- ✓ el **consumo medio** debe estar dentro del rango permitido por las constantes de clase que definen el máximo y el mínimo consumo.

Implementa un **constructor** para la clase `Vehiculo` a partir de las condiciones anteriores.

Teniendo en cuenta todo lo anterior, un constructor para la clase `Vehiculo` con esos parámetros podría quedar más o menos así:

```
public Vehiculo (String matricula, LocalDate fechaMatriculacion, float capacidadDeposito, float consumoMedio) throws IllegalArgumentException {

    // Comprobación de que los valores de entrada son válidos
    if ( matricula==null ) // Comprobación de que la matrícula no es null
        throw new IllegalArgumentException ("matrícula no válida (null)");
    if (!matricula.matches("[0-9]{4}[BCDFGHJKLMNPQRSTVWXYZ]{3}")) // Comprobación de que la matrícula cumple el patrón NNNNLLL
        throw new IllegalArgumentException (String.format ("matrícula no válida (%s)", matricula));
    if (fechaMatriculacion==null) // Comprobación de que la fecha de matriculación no es null
        throw new IllegalArgumentException ("fecha de matriculación no válida (null)");
    if (fechaMatriculacion.getYear() < 1920 || fechaMatriculacion.isAfter(LocalDate.now())) // Comprobación entre 1920 y fecha actual
        throw new IllegalArgumentException ("fecha de matriculación no válida (%s)", fechaMatriculacion);
    if (capacidadDeposito < Vehiculo.MINIMA_CAPACIDAD_DEPOSITO || capacidadDeposito > Vehiculo.MAXIMA_CAPACIDAD_DEPOSITO) // Comprobación rango capacidad
        throw new IllegalArgumentException ("Capacidad del depósito no válida (%.2f)", capacidadDeposito);
    if (consumoMedio < Vehiculo.MINIMO_CONSUMO_MEDIO || consumoMedio > Vehiculo.MAXIMO_CONSUMO_MEDIO) // Comprobación rango consumo
        throw new IllegalArgumentException ("Consumo medio no válido (%.2f)", consumoMedio);

    // Una vez que se ha garantizado que todos los valores de entrada son apropiados, puede continuar el proceso de instanciación
    // del objeto

    // Asignación de valores definitivos a los atributos inmutables a partir de los parámetros de entrada recibidos
    this.matricula= matricula;
    this.fechaMatriculacion= fechaMatriculacion;
    this.capacidadDeposito= capacidadDeposito;
    this.consumoMedio= consumoMedio;

    // Asignación de valores iniciales a los atributos de estado (variables)
    this.estadoMotor= false;      // Motor apagado
    this.nivelDeposito= 0;         // Depósito vacío
    this.kilometrosTotales=0 ;     // Vehículo sin recorrer aún ningún trayecto
    this.kilometrosParciales=0 ;

    // Actualización de atributos de clase por el hecho de crearse un nuevo vehículo
    Vehiculo.vehiculosCreados++; // Se incrementa en uno la cantidad de vehículos fabricados
}
```

Al llevar a cabo comprobaciones sobre parámetros que son referencias a objetos (por ejemplo `String` o `LocalDate`) debes tener la precaución de comprobar primero que no son `null`, porque si intentas acceder a un miembro (método o atributo) de una referencia `null`, se producirá un error saltando una excepción de tipo `NullPointerException`. Una vez que sepas que no es `null`, ya sí podrás intentar acceder al método que consideres oportuno.

6.4 SOBRECARGA DE CONSTRUCTORES.

Habrá ocasiones en las que puede que te interese disponer de más de un constructor porque no siempre necesitemos pasarle la misma cantidad de parámetros. Imagina por ejemplo el caso de la clase **Rectangulo** que ya hemos usado en ocasiones anteriores. Podríamos definir un primer constructor donde se pasarán todos los valores iniciales de los atributos de objeto:

```
public Rectangulo (double x1, double y1, double x2, double y2, String nombre, String color) throws IllegalArgumentException {
    // Comprobación de que los valores de entrada son válidos

    if (x1>=x2 || y1>y2) // Comprobación de que la ubicación no es inconsistente (x1,y1) debe estar a la izquierda y por debajo de (x2,y2)
        throw new IllegalArgumentException (
            String.format ("El vértice (x1,y1)=(%.2f,%.2f) debe estar a la izquierda y por debajo del (x2,y2)=(%.2f,%.2f)",
                           x1, y1, x2, y2)
        );
    if ( nombre == null || nombre.isEmpty() ) // Comprobación de que el nombre no es null ni vacío
        throw new IllegalArgumentException ("nombre null o vacío");
    if ( color== null || color.isEmpty() ) // Comprobación de que el color no es null ni vacío
        throw new IllegalArgumentException ("color null o vacío");

    // Una vez que se ha garantizado que todos los valores de entrada son apropiados, puede continuar el proceso de instanciación
    // del objeto

    // Asignación de valores iniciales a los atributos de estado
    this.x1= x1;
    this.y1= y1;
    this.x2= x2;
    this.y2= y2;
    this.nombre= nombre;
    this.color= color;

    // Actualización de atributos de clase por el hecho de crearse un nuevo rectángulo
    Rectangulo.cantidadRectangulos++; // Incrementamos la cantidad de rectángulos creados
}
```

Sin embargo, puede ser que también nos interesa disponer de un constructor al que sólo le pasáramos las coordenadas de ubicación. Pero entonces, ¿qué valor inicial podríamos asignarle al nombre y al color? Podría decidirse disponer de unos valores por omisión para esos casos. Si se toma esa decisión es recomendable incluir atributos de clase constantes que almacenen esos valores para evitar tener que escribir literales dentro de nuestro código. De este modo, si algún día se decide modificar esos valores por omisión, tan solo habrá que cambiarlos en la declaración de esos atributos y no habrá que estar realizando cambios el resto del código. Es la filosofía conocida como: "*escríbelo una vez, úsallo muchas*" ("*write it once, use multiple times*").

Para ello entonces podríamos definir esos dos nuevos atributos:

```
public class Rectangulo {
    // ATRIBUTOS DE CLASE: sólo habrá un atributo (común para todos los objetos) para representar estas características
    // -----
    // Atributos de clase constantes. Representan características "inmutables" de la clase,
    // como por ejemplo restricciones, valores informativos, valores por omisión (default), etc.
    public final static String NOMBRE_DEFAULT = "Anónimo"; // Nombre que se asignará por omisión si no se proporciona uno
    public final static double COLOR_DEFAULT = "negro"; // Color que se asignará por omisión si no se proporciona uno
    ...
}
```

TEMA 5: DESARROLLO DE CLASES

Y usaremos esos valores para asignarlos a los atributos **nombre** y **color** en la versión del constructor que no recibe esos dos parámetros:

```
public Rectangulo (double x1, double y1, double x2, double y2) throws IllegalArgumentException {  
    ...  
  
    this.nombre= NOMBRE_DEFAULT;  
    this.color= COLOR_DEFAULT;  
    ...  
}
```

Ahora bien, ¿tenemos que escribir otra vez todo el código del constructor cambiando únicamente esas dos líneas? ¿Tendremos que escribir otra vez todas las sentencias de comprobación de los parámetros y todas las instrucciones de asignación de valores inciales a los atributos? No parece muy razonable, ¿verdad? Sobre todo después de haber dicho que íbamos a intentar seguir una filosofía del tipo "*esríbelo una vez, úsalo muchas*". Repetir código en nuestras clases para supuestamente hacer lo mismo o algo muy parecido es muy peligroso, pues si más adelante hay que realizar algún cambio, tendrás que aplicar esa modificación a todos los sitios donde ese cambio afecte.

Por ejemplo, imagínate que has escrito tres o cuatro constructores en los cuales repites una y otra vez una buena parte de las comprobaciones y las asignaciones con ligeras variaciones y posteriormente descubres que te has confundido en alguna de esas comprobaciones. Tendrías que corregir ese error en todos los sitios donde aparezca. Si sólo apareciera en un sitio y el resto de constructores lo reutilizaran, sólo tendrías que arreglarlo una vez. Esto, además de ahorrarte mucho tiempo y dolores de cabeza, te evita el peligro de que no llegues a corregir el fallo en todos los lugares donde se ha repetido, quedando en algunas zonas del código arreglado y en otras sin arreglar. Imagínate el lío que podría producirse y cómo se incrementaría el tiempo que necesitarías para implementar tu clase cada vez que descubrieras un pequeño error.

Muy bien, una vez que tenemos claro que **debemos evitar siempre que sea posible el código redundante**, viene el momento de preguntarnos si es posible hacerlo en el caso de los constructores en Java. ¿Proporciona el lenguaje algún mecanismo para poder llamar a un constructor desde otro constructor?

Efectivamente sí se puede llamar a un constructor de una clase desde otro constructor de la misma clase. Ésa es la función del método `this()`, que dentro de un constructor sirve para hacer una llamada al constructor de la misma clase cuya lista de argumentos coincida con los que se le pasan a `this()`. La única restricción es que, de invocar a `this()` dentro de un constructor, ésta tiene que ser obligatoriamente la primera sentencia del nuevo constructor.

6.4.1. USO DE LA LLAMADA THIS() EN LOS CONSTRUCTORES.

Reutilizando el constructor.

¿Recuerdas que en un apartado anterior titulado como "*La referencia this*", te dijimos que no había que confundir el operador **this** con el método **this()**?

Había un bloque "*Para saber más*" donde se hablaba de usar **this** en constructores, y un bloque "*Reflexiona*" donde te indicábamos que se explicaría con más detalle más adelante. Pues bien, ha llegado el momento de desarrollar ese tema.

Volviendo al **constructor** de nuestra clase **Rectangulo**, en el apartado anterior afirmamos que no era necesario volver a tener que escribir de nuevo todo el código relativo a las comprobaciones y asignaciones de atributos a partir de los parámetros recibidos. Dijimos que para ello Java nos ofrecía un mecanismo conocido como llamada a **this()**.

TEMA 5: DESARROLLO DE CLASES

La llamada a **this()** consiste en la invocación a un constructor de la clase desde un constructor diferente. De esta manera podemos tener un constructor que se encargue de todo el "trabajo duro" de comprobaciones y asignaciones y el resto de constructores (que normalmente tendrán una menor cantidad de parámetros) harán uso de este constructor más "grande, complejo e inteligente" evitando tener que repetir el código que éste contiene.

En nuestro ejemplo de constructor de la clase **Rectangulo** que no recibe como parámetros el nombre y el color, podría quedar así:

```
// Constructor con cuatro parámetros: x1,y1,x2,y2
// Los atributos nombre y color tomarán los valores por omisión
public Rectangulo (double x1, double y1, double x2, double y2) throws IllegalArgumentException {
    // Llamada al constructor con seis parámetros
    this (x1, y1, x2, y2, Rectangulo.NOMBRE_DEFAULT, Rectangulo.COLOR_DEFAULT);
}
```

De esta manera el cuerpo de esta versión del constructor (con cuatro parámetros) queda reducida a su mínima expresión haciendo una invocación al constructor con seis parámetros que ya se ha implementado anteriormente. Así podemos reutilizar ese constructor y aprovechar todas las comprobaciones y asignaciones que se realizan en él sin tener que volver a escribir todo ese código. Qué fácil, ¿no?

Veamos otro ejemplo más. Supongamos que también nos interesa disponer de un constructor sin parámetros donde se establezcan, además del nombre y el color por omisión, también una ubicación por omisión definida por los puntos (0.0,0.0) y (1.0,1.0).

En tal caso podríamos hacer una llamada al anterior constructor de cuatro parámetros con esos valores por omisión:

```
// Constructor sin parámetros
// Todos los atributos tomarán sus valores por omisión
public Rectangulo () {
    // Llamada al constructor con cuatro parámetros con la ubicación (0.0,0.0), (1.0,1.0)
    this (0.0, 0.0, 1.0, 1.0);
}
```

Por otro lado, como hemos dicho antes, también es conveniente disponer de los **valores por omisión** (así como máximos, mínimos, etc.) en **constantes de clase** para no tener que escribirlos una y otra vez a lo largo de todo el código: **X1_DEFAULT**, **Y1_DEFAULT**, etc. En tal caso el constructor podría quedar así:

```
public Rectangulo () {
    // Llamada al constructor con cuatro parámetros usando la ubicación por omisión
    this (Rectangulo.X1_DEFAULT, Rectangulo.Y1_DEFAULT, Rectangulo.X2_DEFAULT, Rectangulo.Y2_DEFAULT);
}
```

Desde otro punto de vista, podrías haber decidido llamar directamente desde el constructor sin parámetros al constructor con seis parámetros y saltarte el "paso" del constructor con cuatro parámetros. Podrías haberlo hecho perfectamente:

```
public Rectangulo () {
    // Llamada al constructor con seis parámetros usando todos los valores por omisión
    this (X1_DEFAULT, Y1_DEFAULT, X2_DEFAULT, Y2_DEFAULT, NOMBRE_DEFAULT, COLOR_DEFAULT);
}
```

Ahora bien, tendrías que escribir algo más de código, que si algún día se descubre que contiene algún error, tendrías que cambiar. Cuanto menos código tengas que escribir, mejor.

Además de estos constructores, puedes implementar todos los que quieras y consideres que te pueden resultar útiles. Por ejemplo, imagina que te interesa un constructor con dos parámetros (nombre y color) que cree un rectángulo con ese nombre y ese color en la ubicación por omisión.

TEMA 5: DESARROLLO DE CLASES

No habría problema, podrías implementarlo también. La única limitación que vas a tener es la propia que impone la sobrecarga de métodos: **no puede haber constructores que tengan exactamente la misma lista de parámetros** (para evitar la ambigüedad).

Teniendo en cuenta esto, ¿podríamos entonces declarar ese constructor con dos parámetros de tipo **String**? No habría problema, pues aún no se ha declarado ningún constructor que tenga únicamente dos parámetros de tipo **String**. Nuestro constructor quedaría entonces así:

```
// Constructor con dos parámetros: sólo nombre y color
// Los atributos de ubicación (x1,y1,x2,y2) tomarán sus valores por omisión
public Rectangulo (String nombre, String color) throws IllegalArgumentException {
    // Llamada al constructor con seis parámetros usando la ubicación por omisión
    this (Rectangulo.X1_DEFAULT, Rectangulo.Y1_DEFAULT, Rectangulo.X2_DEFAULT, Rectangulo.Y2_DEFAULT, nombre, color);
}
```



Ejercicio Resuelto

Nos han pedido modelar una clase clase **Reloj** que almacena la hora, el minuto y el segundo. Las horas estarían en el rango 0-23, mientras que los minutos y segundos lo están en el rango 0-59.

Implementa la clase **Reloj** con **tres atributos** (hora, minuto, segundo) y **dos constructores**:

- ✓ **constructor con tres parámetros** (hora, minuto y segundo),
- ✓ **constructor sin parámetros** que inicia el reloj a su valor por omisión, que sería un reloj a las **00:00:00**.



```
/**
 * Clase para modelar un reloj. Las horas deben tener valores permitidos de
 * 0 hasta 23. Los minutos y segundos de 0 hasta 59.
 */
public class Reloj {

    // Atributos constantes de clase
    private static final byte HORAS_DEFAULT= 0;
    private static final byte MINUTOS_DEFAULT= 0;
    private static final byte SEGUNDOS_DEFAULT= 0;

    // Atributos de objeto
    private byte horas;
    private byte minutos;
    private byte segundos;

    // Constructor con tres parámetros (horas, minutos, segundos)
    public Reloj(byte horas, byte minutos, byte segundos) throws IllegalArgumentException {
        // Comprobamos si los parámetros están en el rango
        if (horas < 0 || horas > 23)
            throw new IllegalArgumentException ("horas inválidas");
        if (minutos < 0 || minutos > 59)
            throw new IllegalArgumentException ("minutos inválidos");
        if (segundos < 0 || segundos > 59) {
            throw new IllegalArgumentException ("segundos inválidos");

        // Si los parámetros son válidos, iniciamos los atributos con esos valores
        this.horas = horas ;
        this.minutos = minutos ;
        this.segundos = segundos ;
    }

    // Constructor sin parámetros que instancia el objeto con los valores por omisión
    public Reloj() {
        // Invocamos desde este constructor, más sencillo, el constructor más extenso con los valores por omisión
        this(Reloj.HORAS_DEFAULT, Reloj.MINUTOS_DEFAULT, Reloj.SEGUNDOS_DEFAULT);
    }
}
```



Ejercicio Resuelto

Nos han pedido que implementemos una segunda versión del **constructor para la clase Vehículo** en la que sólo se reciban dos parámetros: **matrícula** y **fecha de matriculación**. La **capacidad del depósito** y el **consumo medio** tomarían los valores por omisión de 50.0 litros y 5.0 litros/100km.

Implementa este nuevo constructor para la clase **Vehículo** aprovechando todo lo que ya se hizo en el anterior constructor. Usa **this** para realizar una llamada a ese constructor desde el nuevo constructor que implementes. Recuerda que si usas **this** en un constructor debe ser obligatoriamente en la primera línea de código. Si no, no se te permitirá compilar.



OpenClipart-Vectors (Pixabay License)

Mostrar retroalimentación

Antes de implementar el constructor, convendría definir dos nuevas estáticas en la clase **Vehículo** para almacenar los valores por omisión de **capacidad del depósito** y **consumo medio**. Así no habrá que utilizar literales desde el constructor y se hará directamente referencia a esos dos atributos constantes de clase:

```
// Atributos de clase constantes
// (representan características "inmutables" de la clase: restricciones, valores informativos, valores por omisión, etc.)
public final static double DEFAULT_CAPACIDAD_DEPOSITO = 50.0; // capacidad del depósito de combustible por omisión (en litros)
public final static double DEFAULT_CONSUMO_MEDIO = 5.0; // Consumo medio por omisión (en litros/100km)
```

Una vez que disponemos de esas constantes con los valores por omisión, podemos implementar el constructor de dos parámetros haciendo una llamada al constructor con cuatro parámetros usando esos valores por omisión:

```
public Vehiculo (String matricula, LocalDate fechaMatriculacion) throws IllegalArgumentException {
    this (matricula, fechaMatriculacion, Vehiculo.DEFAULT_CAPACIDAD_DEPOSITO, Vehiculo.DEFAULT_CONSUMO_MEDIO);
}
```

Para saber más

Aquí tienes un vídeo donde se explica el concepto de **sobrecarga de constructores** mediante un par de ejemplos:

<https://www.youtube.com/watch?v=ZWcobe9afw&list=PLU8oAIHdN5BktAXdEVCLUYzvDyqRQJ2lk>

6.5 CONSTRUCTORES DE COPIA.

Una forma de iniciar un objeto podría ser mediante la copia de los valores de los atributos de otro objeto ya existente. Imagina que necesitas varios objetos iguales (con los mismos valores en sus atributos) y que ya tienes uno de ellos perfectamente configurado (sus atributos contienen los valores que tú necesitas). Estaría bien disponer de un constructor que hiciera copias idénticas de ese objeto.

Durante el proceso de creación de un objeto puedes generar objetos exactamente iguales (basados en la misma clase) que se distinguirán posteriormente porque podrán tener estados distintos (valores diferentes en los atributos). La idea es poder decirle a la clase que además de generar un objeto nuevo, que lo haga con los mismos valores que tenga otro objeto ya existente. Es decir, algo así como si pudieras **clonar** el objeto tantas veces como te haga falta. A este tipo de mecanismo se le suele llamar **constructor copia** o **constructor de copia**.

Un constructor copia es un método constructor como los que ya has utilizado pero con la particularidad de que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar. Este método revisa cada uno de los atributos del objeto recibido como parámetro y se copian todos sus valores en los atributos del objeto que se está creando en ese momento en el método constructor.

TEMA 5: DESARROLLO DE CLASES

Un ejemplo de constructor copia para la clase **Rectangulo** podría ser:

```
public Rectangulo (Rectangulo r) {
    // Asignación de valores iniciales a los atributos de estado
    this.x1= r.x1;
    this.y1= r.y1;
    this.x2= r.x2;
    this.y2= r.y2;
    this.nombre= r.nombre;
    this.color= r.color;

    // Actualización de atributos de clase por el hecho de crearse un nuevo rectángulo
    Rectangulo.cantidadRectangulos++; // Incrementamos la cantidad de rectángulos creados
}
```

Además, si nos es posible, deberíamos aprovechar la llamada a **this()** para no tener que volver a escribir todas las líneas de código de asignación de atributos de objeto y de actualización de atributos de clase, tal y como ya hemos hecho en otros casos de **sobrecarga de constructores**:

```
public Rectangulo (Rectangulo r) {
    this (r.x1, r.y1, r.x2, r.y2, r.nombre, r.color); // Llamada al constructor con seis parámetros
}
```

Si te fijas, aquí ya no se puede dar el caso de un parámetro inválido para "rellenar" los atributos del objeto, pues estamos pasando un objeto que ya ha sido creado y que por tanto debe tener todos sus atributos correctos.

Ahora bien, debemos tener cuidado porque podría ser que el parámetro **r** fuera **null**. En tal caso se nos produciría una **NullPointerException** en cuanto intentaramos realizar cualquier acceso a un miembro de **r** (por ejemplo **r.x1**). Es decir, que este método, aunque no lance una **IllegalArgumentException**, sí habrá que añadirle que puede lanzar una **NullPointerException**. Por tanto el constructor nos quedaría finalmente:

```
public Rectangulo (Rectangulo r) throws NullPointerException { // Si r es null se lanzará una NullPointerException
    this (r.x1, r.y1, r.x2, r.y2, r.nombre, r.color); // Llamada al constructor con seis parámetros
}
```



Ejercicio resuelto

Podríamos diseñar un constructor copia para clase la **Vehiculo** generando un nuevo vehículo con las mismas características que el vehículo que se pasa como parámetro aunque recién fabricado (motor apagado, depósito de combustible vacío, sin haber recorrido ningún kilómetro, etc.).

Implementa un **constructor copia** para la clase **Vehiculo**.



[OpenClipart-Vectors \(Pixabay License\)](#)

Mostrar retroalimentación

Las características del nuevo vehículo serán las mismas que las del vehículo que se pasa como parámetro aunque el estado de este nuevo vehículo será el de un vehículo nuevo. Por tanto, podríamos utilizar directamente el constructor habitual invocándolo mediante una llamada a **this()**.

```
public Vehiculo (Vehiculo v) throws NullPointerException {
    this (v.matricula, v.fechaMatriculacion, v.capacidadDeposito, v.consumoMedio);
}
```

6.6 DESTRUCCIÓN DE OBJETOS.

Como ya has estudiado en unidades anteriores, cuando un objeto deja de ser utilizado, los recursos usados por él (memoria, acceso a archivos, conexiones con bases de datos, etc.) deberían de ser liberados para que puedan volver a ser utilizados por otros procesos (**mecanismo de destrucción del objeto**).

En Java, mientras que de la construcción de los objetos se encargan los métodos constructores, implementados por el programador, de la destrucción se encarga un proceso del entorno de ejecución conocido como **recolector de basura (garbage collector)**. Este proceso va buscando periódicamente objetos que ya no son referenciados (no hay ninguna variable que haga referencia a ellos y por tanto no hay manera de poder llegar a ellos para usarlos desde el programa) y los marca para ser eliminados. Posteriormente los irá eliminando de la memoria cuando lo considere oportuno (en función de la carga del sistema, los recursos disponibles, etc.).

Normalmente se suele decir que en Java no hay método destructor, mientras que en otros lenguajes orientados a objetos como C++, sí se implementa explícitamente el destructor de una clase de la misma manera que se define el constructor. En realidad en Java también es posible implementar el método destructor de una clase: se trata del método **finalize()**. Ahora bien, este proceso de destrucción no funciona exactamente de la misma forma que en otros lenguajes, donde el control total de la destrucción se deja en manos del programador. En Java más bien podemos decidir qué deberá hacerse al destruir un objeto, pero nunca cuándo se destruirá (de eso ya se encargará el recolector de basura).

Este método **finalize()** es llamado por el recolector de basura cuando va a destruir el objeto (lo cual nunca se sabe cuándo va a suceder exactamente, pues una cosa es que el objeto sea marcado para ser borrado y otra que sea borrado efectivamente). Si ese método no existe, se ejecutará un destructor por omisión (el método **finalize()** que contiene la clase **Object**, de la cual heredan todas las clases en Java) que liberará la memoria ocupada por el objeto. Se recomienda por tanto que si un objeto utiliza determinados recursos de los cuales no tienes garantía que el entorno de ejecución los vaya a liberar (cerrar archivos, cerrar conexiones de red, cerrar conexiones con bases de datos, etc.), implementes explícitamente un método **finalize()** en tus clases. Si el único recurso que utiliza tu clase es la memoria necesaria para albergar sus atributos, eso sí será liberado sin problemas. Pero si se trata de algo más complejo, será mejor que te encargues tú mismo de hacerlo implementando tu destructor personalizado (**finalize()**).

Por otro lado, esta forma de funcionar del entorno de ejecución de Java (destrucción de objetos no referenciados mediante el recolector de basura) implica que no puedes saber exactamente cuándo un objeto va a ser definitivamente destruido, pues si una variable deja de ser referenciada (se cierra el ámbito de ejecución donde fue creada) no implica necesariamente que sea inmediatamente borrada, sino que simplemente es marcada para que el recolector la borre cuando pueda hacerlo.

Si en un momento dado fuera necesario garantizar que el proceso de finalización (método **finalize()**) sea invocado, puedes recurrir al método estático **runFinalization()** de la clase **System** para forzarlo:

```
System.runFinalization();
```

Este método se encarga de llamar a todos los métodos **finalize()** de todos los objetos marcados por el recolector de basura para ser destruidos.

Si necesitas implementar un destructor (normalmente no será necesario), debes tener en cuenta que:

- el nombre del método destructor debe ser **finalize()**,
- no puede recibir parámetros,
- sólo puede haber un destructor en una clase. No es posible la sobrecarga dado que no tiene parámetros,
- no puede devolver ningún valor. Debe ser de tipo **void**.

TEMA 5: DESARROLLO DE CLASES

También existe la posibilidad de invocar al recolector de basura explícitamente, mediante el método **System.gc()** (Garbage Collector). Pero su invocación no es más que una sugerencia a la máquina virtual para que considere la conveniencia de lanzar el recolector de basura en ese momento que nosotros estimamos oportuno por algún motivo (haber borrado de muchos elementos, etc.), pero la máquina virtual podrá atender nuestra solicitud o no tenerla en cuenta en función de las necesidades del sistema.

6.7 MÉTODOS "FÁBRICA" O PSEUDOCONSTRUCTORES.

En algunos casos podrás encontrarte con métodos de una clase que devuelven objetos instancias de esa misma clase. En algunas ocasiones puede deberse a que se trata de un método que realiza algún tipo de operación entre el objeto cuyo método se invoca y el o los parámetros que recibe, devolviendo un nuevo objeto resultado de esa operación. Algunos ejemplos de esto podrían ser los métodos **plusDays**, **plusMonths** o **plusYears** de la clase **LocalDate** que devuelven una nueva fecha sumando una determinada cantidad de días, meses o años a la fecha objeto original. Son métodos que no modifican el objeto cambiando sus atributos de fecha sino que generan una nueva fecha con esos cambios (algo similar sucede con los atributos de la clase **String**).

En otros casos se trata de un **método estático** que genera un nuevo objeto de esa clase a partir de ciertos parámetros, como si fuera un constructor. Hay quien llama a estos métodos "pseudoconstructores" o métodos "fábrica" pues se trata de métodos que "fabrican" o instancian objetos de una manera similar a como lo hacen los constructores.

Ya has visto algunas clases de la API de Java que tienen métodos de este tipo. Por ejemplo la clase **LocalTime**, que no dispone de constructores públicos sino que proporciona tres métodos estáticos para poder obtener un objeto de tipo **LocalTime**:

- método **of**, que devuelve un nuevo objeto **LocalTime** a partir de varios números enteros (hora, minutos, segundo). Es muy similar a un constructor;
- método **parse**, que devuelve un nuevo objeto **LocalTime** a partir de una cadena de caracteres con formato de hora (como por ejemplo "09:15");
- método **now**, que devuelve un nuevo objeto **LocalTime** con la hora actual del sistema.

Esos tres métodos permiten instanciar un nuevo objeto de la clase **LocalTime** como si fueran constructores, aunque formalmente no lo son. ¿Cómo podríamos nosotros implementar métodos de ese tipo en nuestras clases? Debemos tener en cuenta lo siguiente:

1. el método debe ser **estático**, pues no tiene mucho sentido que sea invocado desde un objeto. Debe ser invocado desde la clase;
2. el método debe **devolver un objeto del mismo tipo que la clase** a la que pertenece;
3. hay que **llamar a algún constructor de la clase** con el operador **new** dentro del método;
4. hay que **devolver la referencia obtenida** con el operador **new** mediante una sentencia **return**.

Llevando a cabo esos pasos podremos crear nuestro propio "pseudoconstructor" o método "fábrica". Si te fijas, en el fondo vamos a necesitar hacer internamente una llamada a un constructor con el operador **new**, pero eso no será percibido desde fuera por quien use el método.

Veamos algunos ejemplos de implementación de métodos "pseudoconstructores" para la clase **Rectangulo**.

TEMA 5: DESARROLLO DE CLASES

Por ejemplo, podríamos implementar el método llamado **crear**, que genera un rectángulo de la misma manera que lo haría el constructor:

```
public static Rectangulo crear (double x1, double y1, double x2, double y2, String nombre, String color) throws IllegalArgumentException {  
    return new Rectangulo (x1, y1, x2, y2, nombre, color);  
}
```

En este caso no añadimos nada nuevo, pues se trata simplemente una llamada al constructor.

Veamos otro ejemplo en el que implementemos un método llamado **random**, que genere un rectángulo con la esquina inferior izquierda en (0,0) y con la **esquina superior derecha** en un **punto aleatorio** entre (1,1) y (10,10).

```
public static Rectangulo random (String nombre, String color) throws IllegalArgumentException {  
    Rectangulo r;  
    double x2 = 1.0 + (int)(Math.random() * 10);  
    double y2 = 1.0 + (int)(Math.random() * 10);  
    r = new Rectangulo(Rectangulo.X1_DEFAULT, Rectangulo.Y1_DEFAULT, x2, y2, nombre, color);  
    return r;  
}
```

Este ejemplo sí hace algo nuevo o diferente respecto a un constructor "habitual".

Podríamos seguir complicando los ejemplos todo lo que quisiéramos. Imaginemos un método **buildSampleRectangulos** que crea un array con varios rectángulos de ejemplo. En este caso en lugar de devolver un único rectángulo se crean y devuelven varios:

```
public static Rectangulo[] buildSampleRectangulos()  
{  
    return new Rectangulo[]{  
        new Rectangulo(0.0, 0.0, 1.0, 1.0, "black", "r1"),  
        new Rectangulo(0.0, 0.0, 2.0, 2.0, "white", "r2"),  
        new Rectangulo(2.0, 2.0, 5.0, 3.0, "red", "r3"),  
        new Rectangulo(6.0, 6.0, 8.0, 8.0, "black", "r4")  
    };  
}
```



Ejercicio Resuelto

Se nos ha pedido que la clase **Vehículo** incorpore un nuevo método que permita "replicar" o "clonar" un objeto vehículo completamente (incluyendo su estado: cantidad de combustible, kilómetros recorridos, estado del motor, etc.).

Implementa un método llamado **clonar** que replique exactamente el objeto vehículo que se pase como parámetro.

TEMA 5: DESARROLLO DE CLASES

En este caso no podremos hacer simplemente una llamada a `this` pues no se trata simplemente de generar un vehículo nuevo con las mismas características que otro, sino de generar un auténtico clon del vehículo original tanto en características como en estado (mismo estado del motor, mismos kilómetros recorridos, etc.). Esto dará lugar a la necesidad de actualizar también muchas variables de clase. Por ejemplo, si el vehículo original estaba con el motor arrancado, el clon también lo estará, lo que dará lugar que tengamos que incrementar en uno el número de vehículos arrancados.

Teniendo en cuenta todo eso, el método `clonar` podría quedar así:

```
public Vehiculo clonar (Vehiculo original) throws NullPointerException {
    Vehiculo copia;

    // Comprobamos que el vehículo pasado como parámetro no es null
    if ( original==null )
        throw new NullPointerException ("vehículo no válido (null)");

    // No es necesario realizar más comprobaciones
    // pues si el objeto es un vehículo creado, sus valores han de ser válidos

    // Creamos un vehículo "nuevo" replicando el original mediante el constructor copia
    copia= new Vehiculo (original);

    // Ya tenemos un vehículo "copia" pero "nuevo"
    // Ahora hay que "envejecerlo" con los mismos valores de estado que en el vehículo original

    // Asignación de valores iniciales a los atributos de estado (atributos variables de objeto)
    // Para ello usamos los atributos del vehículo recibido como parámetro (vehículo "original")
    copia.estadoMotor=          original.estadoMotor;           // Replicamos el estado del motor
    copia.nivelDeposito=        original.nivelDeposito;        // Replicamos el nivel del depósito de combustible
    copia.kilometrosTotales=   original.kilometrosTotales;   // Replicamos los kilómetros recorridos
    copia.kilometrosParciales= original.kilometrosParciales; // (este vehículo es igual de "viejo" que el original)

    // Actualización de atributos de clase por el hecho de crearse un vehículo con "historia"

    // Si el vehículo que estamos clonando tiene el motor arrancado
    if (original.estadoMotor)
        Vehiculo.vehiculosArrancados++; // Se incrementa en uno la cantidad de vehículos activos

    // Actualizamos los kilómetros totales de la flota con los del nuevo clon
    Vehiculo.kilometrosTotalesFlota += original.KilometrosTotales;

    // Devolvemos el vehículo recién creado y "envejecido"
    return copia;
}
```

En este caso podría plantearse un interesante debate: ¿habría que actualizar el atributo estático del total de kilómetros recorridos por todos los vehículos (`kilometrosTotalesFlota`) si clonamos un vehículo que ya ha recorrido cierta cantidad de kilómetros? ¿Habrá que añadir los kilómetros de ese nuevo vehículo clonado al total de vehículos recorridos por la flota? En nuestro ejemplo lo hemos hecho:

```
// Actualizamos los Kilómetros totales de la flota con los del nuevo clon
Vehiculo.kilometrosTotalesFlota += original.KilometrosTotales;
```

Pero es algo discutible y que podría hacerse o no dependiendo de la finalidad que busquemos a la hora de implementar este método. Será una decisión de diseño el optar por una cosa u otra.

Para saber más

La decisión de implementar/usuarios constructores o bien métodos fábrica es algo que se sale de los objetivos de este curso. Entraríamos en cuestiones como **patrones de diseño**, que serían más propias de un curso de **diseño orientado a objetos** que de un curso de **introducción a la programación**.

Para el que quiera profundizar algo más sobre este tema os dejamos algunos enlaces introductorios:

- Artículo "[Java Constructors vs Static Factory Methods](#)" en la web de Baeldung.
- Artículo "[What are static factory methods?](#)" en web Stackoveflow.
- Artículo "[Constructors or Static Factory Methods?](#)" en la web DZone.

6.8 BLOQUES DE INICIALIZACIÓN EN JAVA.

El lenguaje Java permite especificar bloques de inicialización en las clases. Pero, **¿qué es un bloque de inicialización?** Se trata de un fragmento de código encerrado entre llaves que se ejecuta al inicializar una clase o un objeto. Se puede hablar por tanto de dos tipos de bloques de inicialización:

- **bloque de inicialización estático.** Se ejecuta al cargar la clase en memoria (al inicio del programa) y no tendrá acceso a atributos o métodos de objeto. Es lógico pues aún no puede existir ninguna instancia de esa clase;
- **bloque de inicialización de instancia (o de objeto).** Se ejecuta al comienzo del constructor, justo después de la llamada al constructor de la superclase ("superconstructor") y antes de la ejecución de cualquier otra línea de código del constructor.

Estos bloques de código pueden resultar de utilidad a la hora realizar inicializaciones tanto de la clase (al comienzo de la ejecución del programa) como del objeto (a la hora de ejecutar los constructores).

El **bloque de inicialización estático**, puede ser adecuado para:

- cargar información de configuración inicial desde un archivo,
- abrir una conexión de red,
- establecer una conexión con una base de datos,
- inicializar atributos estáticos,
- ejecutar algún otro tipo de código de configuración o inicialización,

Ese bloque de código sólo se ejecutará una vez (al comenzar la ejecución de un programa en el que se haga uso de la clase).

Un bloque de inicialización estático se indica dentro del cuerpo de la clase mediante un par de **llaves precedidas del modificador static**:

```
static {
    // Líneas de código del bloque de inicialización estático
    ...
}
```

El **bloque de inicialización de instancia** puede resultar conveniente para llevar a cabo tareas similares a las anteriores pero cada vez que se construya un nuevo objeto instancia de la clase (cuando se invoque a un constructor).

Un bloque de inicialización de instancia se indica dentro del cuerpo de la clase mediante un par de **llaves sin ningún modificador**:

```
{
    // Líneas de código del bloque de inicialización de instancia
    ...
}
```

TEMA 5: DESARROLLO DE CLASES

Aquí tienes un ejemplo de uso de ambos bloques en una clase:

```
public class PruebaDeBloques {
    private static int atributoClase;
    private int atributoInstancia;

    static { // Bloque de inicialización estático. Sólo se ejecutará una vez
        System.out.println("Bloque de inicialización estático.");
        atributoClase= 1;
    }

    { // Bloque de inicialización estático. Se ejecutará tantas veces como objetos se creen
        System.out.println("Bloque de inicialización de instancia.");
        atributoInstancia= atributoClase*10;
        atributoClase *= 2;
    }

    public static void main(String[] args) {
        PruebaDeBloques b1, b2, b3, b4;
        System.out.printf ("Inicio de ejecución del programa...\n");
        System.out.printf ("atributoClase= %d\n", PruebaDeBloques.atributoClase);

        System.out.printf ("\nAsignamos a b1 la referencia de una nueva instancia de PruebaDeBloques.\n");
        b1 = new PruebaDeBloques();
        System.out.printf ("atributoClase= %d\n", PruebaDeBloques.atributoClase);
        System.out.printf ("b1.atributoInstancia= %d\n", b1.atributoInstancia);

        System.out.printf ("\nAsignamos a b2 la referencia de una nueva instancia de PruebaDeBloques.\n");
        b2 = new PruebaDeBloques();
        System.out.printf ("atributoClase= %d\n", PruebaDeBloques.atributoClase);
        System.out.printf ("b2.atributoInstancia= %d\n", b2.atributoInstancia);

        System.out.printf ("\nAsignamos a b3 la referencia almacenada en b1.\n");
        b3 = b1;
        System.out.printf ("atributoClase= %d\n", PruebaDeBloques.atributoClase);
        System.out.printf ("b3.atributoInstancia= %d\n", b3.atributoInstancia);

        System.out.printf ("\nAsignamos a b4 la referencia de una nueva instancia de PruebaDeBloques.\n");
        b4 = new PruebaDeBloques();
        System.out.printf ("atributoClase= %d\n", PruebaDeBloques.atributoClase);
        System.out.printf ("b4.atributoInstancia= %d\n", b4.atributoInstancia);
    }
}
```

Bloque de inicialización estático.
Inicio de ejecución del programa...
atributoClase= 1

Asignamos a b1 la referencia de una nueva instancia de PruebaDeBloques.
Bloque de inicialización de instancia.
atributoClase= 2
b1.atributoInstancia= 10

Asignamos a b2 la referencia de una nueva instancia de PruebaDeBloques.
Bloque de inicialización de instancia.
atributoClase= 4
b2.atributoInstancia= 20

Asignamos a b3 la referencia almacenada en b1.
atributoClase= 4
b3.atributoInstancia= 10

Asignamos a b4 la referencia de una nueva instancia de PruebaDeBloques.
Bloque de inicialización de instancia.
atributoClase= 8
b4.atributoInstancia= 40

Para saber más

Si quieres ampliar con algo más de información sobre los bloques de inicialización puedes echar un vistazo a los artículos sobre inicializaciones en Java en los manuales originales de Oracle: [Initializing Fields.](#)

7. DOCUMENTACIÓN DE UNA CLASE.



Reflexiona

Una vez que ya tenemos nuestra clase implementada con todos sus miembros, nos queda una cuestión de gran importancia: la **documentación de la clase**. Del mismo modo que nosotros hemos podido hacer uso de la documentación javadoc de todas las clases de la API de Java y de esa manera hemos podido consultar cómo funcionan determinados métodos, qué parámetros pueden recibir, qué valores devuelven, qué posibles excepciones pueden saltar, etc. ahora nos toca a nosotros proporcionar una documentación similar a quienes pudieran utilizar en el futuro nuestras clases.

Piensa en las siguientes cuestiones:

- ✓ ¿Quién crees que accederá a la documentación de la clase?
- ✓ ¿Qué debemos incluir en la documentación de una clase?

[Mostrar retroalimentación](#)

- ✓ Accederán a la documentación de una clase todas aquellas personas que desarrollen aplicaciones usando esa clase, incluidos los autores o autoras del propio código.
 - ✓ En la documentación de una clase deberíamos incluir, como mínimo, la **descripción general de la clase**, así como la **descripción de cualquier miembro que sea público (constructores, métodos, atributos)** y por tanto pueda ser utilizado por otras personas cuando utilicen nuestra clase como parte del desarrollo de sus aplicaciones.

La documentación de las clases que se proporciona junto con la API de Java en el JDK ha sido generada con una herramienta llamada **Javadoc**. Nosotros también podemos generar la documentación de nuestras clases a través de dicha herramienta.

Si desde el principio nos acostumbramos a documentar el funcionamiento de nuestras clases desde el propio código fuente, estaremos facilitando la generación de la futura documentación de nuestras aplicaciones. ¿Cómo lo logramos? A través de una serie de comentarios especiales, llamados **comentarios de documentación** que serán tomados por **Javadoc** para generar una serie de archivos HTML que permitirán posteriormente, navegar por nuestra documentación con cualquier navegador web.

Los comentarios de documentación o comentarios "javadoc" tienen una marca de comienzo (`/**`) y una marca de fin (`*/`). En su interior podremos encontrar dos partes diferenciadas: una para realizar una descripción y otra en la que encontraremos más etiquetas de documentación. Veamos un ejemplo:

```
/**  
 * Descripción principal (texto/HTML)  
 *  
 * Etiquetas (texto/HTML)  
 */
```

Este es el formato general de un comentario de documentación. Comenzamos con la marca de comienzo en una línea. Cada línea de comentario comenzará con un asterisco. El final del comentario de documentación deberá incorporar la marca de fin. Las dos partes diferenciadas de este comentario son:

- **Zona de descripción:** es aquella en la que el programador escribe un comentario sobre la clase, atributo, constructor o método que se vaya a codificar bajo el comentario. Se puede incluir la cantidad de texto que se necesite, pudiendo añadir etiquetas HTML que formateen el texto escrito y así ofrecer una visualización mejorada al generar la documentación mediante **Javadoc**.
- **Zona de etiquetas:** en esta parte se colocará un conjunto de etiquetas de documentación a las que se asocian textos. Cada etiqueta tendrá un significado especial y aparecerán en lugares determinados de la documentación, una vez haya sido generada.

Aquí tienes un ejemplo de comentario de la documentación de un método:

```
/**  
 * Obtiene el saldo actual de la cuenta  
 *  
 * @return saldo actual de la cuenta  
 */  
public double getSaldo() {
```

7.1 ETIQUETAS Y POSICIÓN.

Cuando hemos de incorporar determinadas **etiquetas** a nuestros **comentarios de documentación** es necesario conocer dónde y qué etiquetas colocar, según el tipo de código que estemos documentando en ese momento. Existirán dos tipos generales de etiquetas:

- Etiquetas de bloque:** son etiquetas que solo pueden ser incluidas en el bloque de documentación, después de la descripción principal y comienzan con el símbolo `@`.
- Etiquetas en texto:** son etiquetas que pueden ponerse en cualquier punto de la descripción o en cualquier punto de la documentación asociada a una etiqueta de bloque. Son etiquetas definidas entre llaves, de la siguiente forma `{@etiqueta}`

En la siguiente tabla podrás encontrar una referencia sobre las diferentes etiquetas y su ámbito de uso.

	<code>@autor</code>	<code>{@code}</code>	<code>{@docRoot}</code>	<code>@deprecated</code>	<code>@exception</code>	<code>{@inheritDoc}</code>	<code>{@link}</code>	<code>{@literal}</code>
Descripción	✓		✓	✓			✓	✓
Paquete	✓		✓	✓			✓	✓
Clases e Interfaces	✓		✓	✓			✓	✓
Atributos			✓	✓			✓	✓
Constructores y métodos			✓	✓	✓	✓	✓	

7.2 USO DE LAS ETIQUETAS.

¿Cuáles son las etiquetas típicas y su significado? Pasaremos seguidamente a enumerar y describir la función de las etiquetas más habituales a la hora de generar comentarios de documentación.

- @author texto con el nombre:** esta etiqueta solo se admite en **clases e interfaces** (las interfaces las veremos en la siguiente unidad). El texto después de la etiqueta no necesitará un formato especial. Podremos incluir tantas etiquetas de este tipo como necesitemos. Por ejemplo: `@author Profe`
- @version texto de la versión:** el texto de la versión no necesitará un formato especial. Es conveniente incluir el número de la versión y la fecha de esta. Podremos incluir varias etiquetas de este tipo una detrás de otra.
- @deprecated texto:** indica que no debería utilizarse, indicando en el texto las causas de ello. Se puede utilizar en todos los apartados de la documentación. Si se ha realizado una sustitución debería indicarse qué utilizar en su lugar. Por ejemplo:

```
@deprecated El método no funciona correctamente. Se recomienda el uso de {@link metodoCorrecto}
```

- @exception nombre-excepción texto:** esta etiqueta es equivalente a `@throws`.
- @param nombre-atributo texto:** esta etiqueta es aplicable a parámetros de constructores y métodos. Describe los parámetros del constructor o método. Nombre-atributo es idéntico al nombre del parámetro. Cada etiqueta `@param` irá seguida del nombre del parámetro y después de una descripción de este. Por ejemplo: `@param cantidad Cantidad que se desea ingresar (en euros)`

TEMA 5: DESARROLLO DE CLASES

- **@return texto:** esta etiqueta se puede omitir en los métodos que devuelven void. Deberá aparecer en todos los métodos que devuelvan algo distinto de void, haciendo explícito qué tipo o clase de valor devuelve y sus posibles rangos de valores. Veamos un ejemplo:

```
/**  
 * Chequea si un vector no contiene elementos.  
 * @return <code>verdadero</code>si y solo si este vector no contiene componentes, esto es, su tamaño es cero;  
 * <code>falso</code> en cualquier otro caso.  
 */  
public boolean VectorVacio() {  
    return elementCount == 0;  
}
```

- **@see referencia:** se aplica a clases, interfaces, constructores, métodos, atributos y paquetes. Añade enlaces de referencia a otras partes de la documentación. Podremos añadir a la etiqueta: cadenas de caracteres, enlaces HTML a páginas y a otras zonas del código. Por ejemplo:

```
* @see "Diseño de patrones: La reusabilidad de los elementos de la programación orientada a objetos"  
* @see <a href="http://www.w3.org/WAI/">Web Accessibility Initiative</a>  
* @see String#equals(Object) equals
```

- **@throws nombre-excepción texto:** en nombre-excepción tendremos que indicar el nombre completo de ésta. Podremos añadir una etiqueta por cada excepción que se lance explícitamente con una cláusula **throws**, pero siguiendo el orden alfabético. Esta etiqueta es aplicable a constructores y métodos, describiendo las posibles excepciones del constructor/método.

7.3 ORDEN DE LAS ETIQUETAS.

¿Es importante el orden a la hora de poner las etiquetas en los comentarios de documentación, o por el contrario es indiferente? Las etiquetas **deben disponerse en un orden determinado**, ese orden es el siguiente:

Etiqueta.	Descripción.
@author	En clases e interfaces. Se pueden poner varios. Es mejor ponerlas en orden cronológico.
@version	En clases e interfaces.
@param	En métodos y constructores. Se colocarán tantos como parámetros tenga el constructor o método. Mejor en el mismo orden en el que se encuentren declarados.
@return	En métodos.
@exception	En constructores y métodos. Mejor en el mismo orden en el que se han declarado, o en orden alfabético.
@throws	Es equivalente a @exception.
@see	Podemos poner varios. Comenzaremos por los más generales y después los más específicos.
@deprecated	En cualquier parte de la documentación. Marca código que ha pasado a estar obsoleto, para el que se recomienda dejar de usarlo y eventualmente, sustituirlo por algún otro método, clase, etc.

Para saber más

Si quieres conocer cómo obtener a través de **Javadoc** la documentación de tus aplicaciones, sigue los siguientes enlaces: [Documentación con Javadoc](#). Y [Documentación de clases y métodos con Javadoc](#).

7.4 EJEMPLO PRÁCTICO.

Una vez que hemos visto la mayoría de las etiquetas que se pueden colocar en la documentación javadoc de una clase Java, vamos a ver un ejemplo práctico de cómo llevar a cabo esta tarea.

Aquí tienes una **guía-resumen** de todo aquello que debes documentar:

- **documentación de la clase.** Debes incluir una descripción lo más completa y detallada posible de la clase, explicando todo aquello que consideres que pueda ser de interés para otros programadores que vayan a utilizarla. Ten en cuenta que no tienen por qué conocer cómo está implementada por dentro, sino únicamente cómo utilizarla (cómo usar sus miembros que actúan como interfaz con el código externo: métodos y atributos públicos). Esta descripción se puede subdividir en:
 - **descripción corta** o resumen de la clase. Consiste en todo el texto que incluyas hasta el primer punto (".") que contenga el texto. Es el texto que aparecerá en la tabla resumen de clases del paquete en el que se encuentre la clase;
 - **descripción larga** de la clase. Todo el texto que haya después del primer punto. Ambas descripciones aparecerán en la descripción general de la clase;
- **documentación de atributos públicos y protegidos** (los miembros protegidos los verás en la próxima unidad) indicando para cada uno de ellos una pequeña descripción;
 - si se trata de un **atributo constante**, incluye su valor usando la etiqueta **@value**. **Debes usar la etiqueta @value y no escribir un literal en el comentario javadoc**. De esta manera si se modificara el valor de la constante en el código, no habría que modificar el comentario.
- **documentación de los métodos públicos y protegidos**, incluyendo para cada método:
 - **descripción del método:**
 - **descripción corta** o resumen del método. Consiste en todo el texto que incluyas hasta el primer punto. Es el texto que aparecerá en la tabla resumen de métodos;
 - **descripción larga** del método. Todo el texto que haya después del primer punto. Ambas descripciones aparecerán en la descripción detallada del método;
 - **descripción del valor devuelto** (etiqueta **@return**), si es que devuelve algo. Esta información aparecerá en la descripción detallada del método;
 - **descripción de cada uno de los parámetros** (etiqueta **@param**);
 - **descripción de las posibles excepciones** que puede lanzar (etiqueta **@throws**);

Es más que recomendable que una vez que hayas terminado de redactar todos tus comentarios javadoc generes la documentación para observar en los HTML obtenidos si están todos tus comentarios javadoc y los textos son correctos, apropiados y completos como para poder ser utilizados como **manual de referencia de la biblioteca de clases sobre los elementos que puedes encontrar en un juego de tablero**.

TEMA 5: DESARROLLO DE CLASES



Ejercicio Resuelto

Estamos implementando una clase llamada `CuentaBancaria` que servirá para **gestionar una cuenta bancaria**. Los objetos de esa clase contarán con una serie de **características** como **identificador de cuenta**, fecha de **creación**, **saldo**, etc. y un repertorio de métodos que permitirán llevar a cabo ciertas acciones (ingresos, transferencias, extracciones, etc.).

Si queremos que esa clase pueda ser utilizada apropiadamente por otros desarrolladores que no hayan participado en su implementación, tendremos que ser muy rigurosos y detallados a la hora de documentar todos y cada uno de sus elementos. Si no, no sabrán como utilizarla. Debemos procurar elaborar un "manual de instrucciones" de la clase lo suficientemente completo como para que cualquier otra persona pueda utilizar esta clase en sus programas sin ayuda externa de los programadores originales.

Para ello disponemos de la herramienta **Javadoc**.

Lo primero que debemos hacer es proporcionar una descripción general de la clase explicando su utilidad.

Teniendo en cuenta que la cabecera de la clase es la siguiente:

```
public class CuentaBancaria {
```

Si quisieramos generar un javadoc que tuviera el siguiente aspecto:

The screenshot shows a Java IDE interface with the following details:

- PACKAGE:** CLASS
- DESCRIPTION:** USE TREE | DEPRECATED | INDEX | HELP
- PREV CLASS | NEXT CLASS | FRAMES | NO FRAMES**
- SUMMARY | NESTED | FIELD | CONSTR | METHOD | DETAIL | FIELD | CONSTR | METHOD**
- pkg201920progtareas05**
- Class CuentaBancaria**
- java.lang.Object**
pkg201920progtareas05.CuentaBancaria
- public class CuentaBancaria**
extends java.lang.Object
- Clase que representa una cuenta bancaria.**
- Los objetos de esta clase contienen atributos que permiten almacenar información sobre:**
 - Identificador de la cuenta. Este valor se establecerá al crear la cuenta y ya no podrá cambiar. Es un valor constante.
 - Fecha de creación de la cuenta. Es también un valor constante. Se establecerá al crear la cuenta y ya no podrá cambiar su valor.
 - Porcentaje de embargo de la cuenta.
 - Saldo actual de la cuenta.
 - Saldo máximo que ha tenido la cuenta a lo largo de su historia.
 - Ingresos totales que ha tenido la cuenta a lo largo de su historia.
- La clase también dispone de información general independiente de los objetos concretos que se hayan creado. Es el caso de:**
 - Saldo global entre todas las cuentas en el momento actual.
 - Número de cuentas embargadas en el momento actual.
 - Fecha de creación de la cuenta más moderna creada hasta el momento actual.

¿Qué comentario javadoc incluirías antes de la cabecera de la clase para poder obtener esa documentación?

En este caso se trata **un único comentario javadoc** con todo ese texto apropiadamente formateado utilizando el lenguaje de marcado **HTML**:

```
1  /**
2  * Clase que representa una <strong>cuenta bancaria</strong>.
3  * <p>
4  * Los objetos de esta clase contienen atributos que permiten almacenar
5  * información sobre:</p>
6  * <ul>
7  * <li><strong>Identificador</strong> de la cuenta. Este valor se establecerá
8  * al crear la cuenta y ya no podrá cambiar. Es un valor constante.</li>
9  * <li><strong>Fecha de creación</strong> de la cuenta. Es también un valor
10 * constante. Se establecerá al crear la cuenta y ya no podrá cambiar su valor.</li>
11 * <li><strong>Porcentaje de embargo</strong> de la cuenta.</li>
12 * <li><strong>Saldo actual</strong> de la cuenta.</li>
13 * <li><strong>Saldo máximo</strong> que ha tenido la cuenta a lo largo de su
14 * historia.</li>
15 * <li><strong>Ingresos totales</strong> que ha tenido la cuenta a lo largo de
16 * su historia.</li>
17 * </ul>
18 * <p>
19 * La clase también dispone de información general independiente de los objetos
20 * concretos que se hayan creado. Es el caso de:</p>
21 * <ul>
22 * <li><strong>Saldo global</strong> entre todas las cuentas en el momento
23 * actual.</li>
24 * <li><strong>Número de cuentas embargadas</strong> en el momento actual.</li>
25 * <li><strong>Fecha de creación</strong> de la <strong>cuenta más moderna</strong>
26 * creada hasta el momento actual.</li>
27 * </ul>
28 */
```

DATE	DESCRIPTION	NETDEP/WITHDRAWAL	DEPOSITS	BALANCE
09-10-15	ATM	**1.25		**74.11
09-10-16	ATM	**1.50		**72.61
09-10-20	DEEP	**2.99		**69.62
09-10-21	HEEP	**300.00		**169.62
09-10-22	ATM	**100.00		**69.62
09-10-23	DEEP	**29.08		**40.54
09-10-24	DEEP		**6.77	**40.53
09-10-25	ATM		**19.01	**59.76
09-10-26	PIRE			**701.57
09-10-29	GERT		**50.00	**701.57

Please refer to the back cover for the
list of common transaction codes.
Please verify your account activity regularly.
If there is a error, modify the bank within 45 days.

Sergio Ortega. (CC BY-SA)

TEMA 5: DESARROLLO DE CLASES

Como puedes observar, en este caso la descripción corta o resumen sería el texto "*Clase que representa una cuenta bancaria.*", que es todo lo que hay hasta el primer punto (carácter '.'). Ese es el texto que aparecerá en el catálogo resumen de clases del paquete en el que se encuentre integrada la clase, por ejemplo:

Class Summary	
Class	Description
CuentaBancaria	Clase que representa una cuenta bancaria .
PruebaCuentaBancaria	Programa de prueba de la clase CuentaBancaria.

Teniendo en cuenta que la clase dispone de una serie de atributos públicos como `DEFAULT_MAX_DESCUBIERTO`, `DEFAULT_SALDO`, `MAX_DESCUBIERTO` etc.:

```
public class CuentaBancaria {  
  
    // -----  
    //          ATRIBUTOS ESTÁTICOS (de clase)  
    // -----  
    // Atributos estáticos constantes públicos (rangos y requisitos de los atributos de objeto)  
    // Son públicos, para que cualquier código cliente pueda acceder a ellos  
  
    public static final double MAX_DESCUBIERTO =      -2_000.00; // euros  
    public static final double MIN_EMBARGO =           0.00;   // porcentaje  
    public static final double MAX_EMBARGO =           100.00; // porcentaje  
    public static final int   MIN_YEAR =                1900;   // año  
    public static final double MAX_SALDO =             50_000_000.00; // euros  
    public static final double DEFAULT_SALDO =         0.00;   // euros  
    public static final double DEFAULT_MAX_DESCUBIERTO = 0.00; // euros
```

Y que queremos que aparezcan documentados con el siguiente formato:

Field Summary	
Fields	Modifier and Type
static double	DEFAULT_MAX_DESCUBIERTO Límite de descubiertos por omisión para una cuenta: 0.0 euros.
static double	INITIAL_SALDO Saldo inicial por omisión para una cuenta: 0.0 euros.
static double	MAX_DESCUBIERTO Descubiertos máximos permitidos al crear una cuenta: 2000.0 euros.
static double	MAX_EMBARGO Embargo máximo de una cuenta: 100.0 %.
static double	MAX_SALDO Saldo máximo para una cuenta: 5.0E7 euros.
static double	MIN_EMBARGO Embargo mínimo de una cuenta: 0.0 %.
static int	MIN_YEAR Año mínimo para la creación de una cuenta: 1900.

¿Qué comentarios javadoc incluirías antes de cada uno de esos atributos para lograrlo?

Procura **no escribir el valor literal de los atributos en tus comentarios javadoc**. Puedes usar para ello la etiqueta `@value`.

El código documentado mediante comentarios javadoc podría quedar así:

```
/**  
 * Descubiertos máximos permitidos al crear una cuenta :  
 * {@value MAX_DESCUBIERTO} euros.  
 */  
public static final double MAX_DESCUBIERTO = -2_000.00; // euros  
/**  
 * Embargo mínimo de una cuenta : {@value MIN_EMBARGO}%.  
 */  
public static final double MIN_EMBARGO = 0.00; // porcentaje  
/**  
 * Embargo máximo de una cuenta: {@value MAX_EMBARGO} %.  
 */  
public static final double MAX_EMBARGO = 100.00; // porcentaje  
/**  
 * Año mínimo para la creación de una cuenta: {@value MIN_YEAR}.  
 */  
public static final int MIN_YEAR = 1900; // año  
/**  
 * Saldo máximo para una cuenta: {@value MAX_SALDO} euros.  
 */
```

TEMA 5: DESARROLLO DE CLASES

```
public static final double MAX_SALDO = 50_000_000.00; // euros
/**
 * Saldo inicial por omisión para una cuenta: {@value DEFAULT_SALDO} euros.
 */
public static final double DEFAULT_SALDO = 0.00; // euros
/**
 * Límite de descubierto por omisión para una cuenta:
 * {@value DEFAULT_MAX_DESCUBIERTO} euros.
 */
public static final double DEFAULT_MAX_DESCUBIERTO = 0.00; // euros
```

Al usar la etiqueta `@value` para documentar el valor de las constantes conseguimos escribir una sola vez el literal y así hacemos referencia directamente al valor de la constante. De este modo, si se modificara el valor de la constante en el código, el comentario javadoc no habría que retocarlo, pues hace referencia directamente al valor y no tiene el valor literal escrito "a mano". Recuerda que debemos evitar escribir cualquier cosa más de una vez en nuestro código. Así evitaremos potenciales fuentes de inconsistencias.

La clase `CuentaBancaria` dispone de cuatro constructores:

```
public CuentaBancaria(double saldoInicial, LocalDate fechaCreacion, double limiteDescubierto) throws IllegalArgumentException {
    ...
}

public CuentaBancaria(double saldoInicial, LocalDate fechaCreacion) throws IllegalArgumentException {
    this(saldoInicial, fechaCreacion, CuentaBancaria.DEFAULT_MAX_DESCUBIERTO);
}

public CuentaBancaria(double saldoInicial) throws IllegalArgumentException {
    this(saldoInicial, LocalDate.now());
}

public CuentaBancaria() {
    this(CuentaBancaria.DEFAULT_SALDO);
}
```

Y deseamos que aparezcan documentados de la siguiente manera en el resumen de constructores:

Constructor	Description
<code>CuentaBancaria()</code>	Constructor sin parámetros.
<code>CuentaBancaria(double saldoInicial)</code>	Constructor con un parámetro.
<code>CuentaBancaria(double saldoInicial, java.time.LocalDate fechaCreacion)</code>	Constructor con dos parámetros.
<code>CuentaBancaria(double saldoInicial, java.time.LocalDate fechaCreacion, double limiteDescubierto)</code>	Constructor con tres parámetros

Además, en la descripción detallada del primer constructor (el que tiene tres parámetros) nos gustaría que apareciera la siguiente información:

Constructor Detail

`CuentaBancaria`

```
public CuentaBancaria(double saldoInicial,
                      java.time.LocalDate fechaCreacion,
                      double limiteDescubierto)
                    throws java.lang.IllegalArgumentException
```

Constructor con tres parámetros

Parameters:

- `saldoInicial` - Saldo inicial de la cuenta (en euros)
- `fechaCreacion` - Fecha de creación de la cuenta
- `limiteDescubierto` - Límite de descubierto de la cuenta (en euros)

Throws:

- `java.lang.IllegalArgumentException` - Si alguno de los parámetros no es válido

¿Qué comentarios javadoc y qué etiquetas utilizarías para lograrlo?

TEMA 5: DESARROLLO DE CLASES

En el comentario javadoc de ese constructor necesitarás hacer uso de las etiquetas `@param` para documentar los parámetros y `@throws` para documentar las posibles excepciones:

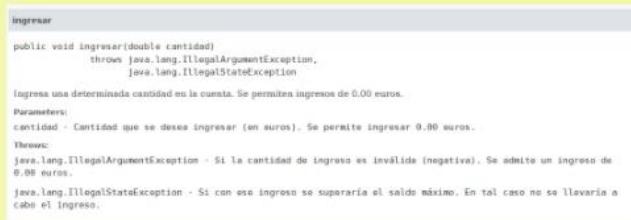
```
/**  
 * Constructor con tres parámetros  
 *  
 * @param saldoInicial Saldo inicial de la cuenta (en euros)  
 * @param fechaCreacion Fecha de creación de la cuenta  
 * @param limiteDescubierto Límite de descubierto de la cuenta (en euros)  
 * @throws IllegalArgumentException Si alguno de los parámetros no es válido  
 */  
public CuentaBancaria(double saldoInicial, LocalDate fechaCreacion, double limiteDescubierto) throws IllegalArgumentException {
```

Te dejamos a ti que intentes la documentación del resto de constructores.

El método `ingresar` tiene la siguiente cabecera:

```
public void ingresar(double cantidad) throws IllegalArgumentException, IllegalStateException {
```

Y nos gustaría que apareciera documentado de la siguiente manera:



¿Qué comentarios javadoc escribirías? ¿Qué etiquetas utilizarías? ¿Necesitarías la etiqueta `@return`?

En este caso también haremos uso de las etiquetas `@param` y `@throws`:

```
/**  
 * Ingrera una determinada cantidad en la cuenta. Se permiten ingresos de  
 * 0.00 euros.  
 *  
 * @param cantidad Cantidad que se desea ingresar (en euros). Se permite  
 * ingresar 0.00 euros.  
 * @throws IllegalArgumentException Si la cantidad de ingreso es inválida  
 * (negativa).  
 * @throws IllegalStateException Si con ese ingreso se superaría el saldo  
 * máximo. En tal caso no se llevaría a cabo el ingreso.  
 */  
public void ingresar(double cantidad) throws IllegalArgumentException, IllegalStateException {
```

No hay que utilizar la etiqueta `@return` dado que este método es `void`, es decir, que no devuelve (`return`) nada, y por tanto no hay que documentarlo.

El método `ingresar` tiene la siguiente cabecera:

```
@Override  
public String toString() {
```

TEMA 5: DESARROLLO DE CLASES

Y nos gustaría que apareciera documentado de la siguiente manera:

toString

```
public java.lang.String toString()
Devuelve una cadena que representa el estado actual de la cuenta. Esta cadena proporcionará la siguiente información:
1. Identificador de la cuenta.
2. Saldo actual de la cuenta.
3. Si la cuenta está o no embargada.

El formato de salida será del siguiente tipo:
Id: XXX - Saldo: YYYYYYYY.YY - Embargada: sí|no [ZZZ.Z%]

donde XXX será el identificador de la cuenta(número entero); YYYYYYYY.YY será el saldo actual en euros, con dos decimales, con una anchura para poder representar el máximo saldo de 50000000.00 euros; ZZZ.Z será el porcentaje actual de embargo de la cuenta con una anchura de 5 caracteres, y con un decimal, si es que está embargada. Si no lo está, no aparecerá información de porcentaje. Simplemente aparecerá "no".
```

Algunos ejemplos de este String de salida podrían ser:

```
Id: 0 - Saldo: 50000000,00 - Embargada: no
Id: 0 - Saldo: 0,00 - Embargada: no
Id: 0 - Saldo: -2000,00 - Embargada: no
Id: 1 - Saldo: 5000,00 - Embargada: sí 25,0%
Id: 1 - Saldo: 3750,00 - Embargada: sí 50,0%
Id: 3 - Saldo: 0,00 - Embargada: sí 100,0%
```

Overrides:
toString in class java.lang.Object
Returns:
Cadena que representa el estado actual de la cuenta.

¿Qué etiquetas necesitarías para ello? Escribe el javadoc apropiado para generar esa documentación. Fíjate que en este caso hay bastante texto y está muy formateado. Tendrás que hacer uso de tus habilidades HTML para intentar que se parezca lo máximo posible...

En este caso necesitaremos únicamente la etiqueta `@return`. El resto es texto, eso sí, con muchas etiquetas HTML.

```
/**
 * Devuelve una cadena que representa el estado actual de la cuenta. Esta
 * cadena proporcionará la siguiente información:
 * <ol>
 * <li><strong>Identificador</strong> de la cuenta.</li>
 * <li><strong>Saldo actual</strong> de la cuenta.</li>
 * <li>Si la cuenta está o no <strong>embargada</strong>.</li>
 * </ol>
 * <p>
 * <strong>El formato de salida</strong> será del siguiente tipo: </p>
 * <pre>Id: XXX - Saldo: YYYYYYYY.YY - Embargada: sí|no [ZZZ.Z%]</pre>
 * <p>
 * donde XXX será el identificador de la cuenta(número entero); YYYYYYYY.YY
 * será el saldo actual en euros, con dos decimales, con una anchura para
 * poder representar el máximo saldo de 50000000.00 euros; ZZZ.Z será el
 * porcentaje actual de embargo de la cuenta con una anchura de 5
 * caracteres, y con un decimal, si es que está embargada. Si no lo está, no
 * aparecerá información de porcentaje. Simplemente aparecerá "no".</p>
 * <p>
 * Algunos ejemplos de este <code>String</code> de salida podrían ser:</p>
 * <pre>Id: 0 - Saldo: 50000000,00 - Embargada: no</pre>
 * <pre>Id: 0 - Saldo: 0,00 - Embargada: no</pre>
 * <pre>Id: 0 - Saldo: -2000,00 - Embargada: no</pre>
 * <pre>Id: 1 - Saldo: 5000,00 - Embargada: sí 25,0%</pre>
 * <pre>Id: 1 - Saldo: 3750,00 - Embargada: sí 50,0%</pre>
 * <pre>Id: 3 - Saldo: 0,00 - Embargada: sí 100,0%</pre>
 *
 * @return Cadena que representa el estado actual de la cuenta.
 */
@Override
public String toString() {
```

Como puedes observar, el texto que hay antes del primer punto es "Devuelve una cadena que representa el estado actual de la cuenta". Eso será lo único que aparecerá en la parte de resumen de métodos:

java.lang.String **toString()**
Devuelve una cadena que representa el estado actual de la cuenta.

8. CREACIÓN Y UTILIZACIÓN DE OBJETOS.

Una vez que ya tienes implementada una clase con todos sus atributos y métodos, ha llegado el momento de utilizarla, es decir, de instanciar objetos de esa clase e interaccionar con ellos. En realidad, en unidades anteriores ya has visto cómo declarar un objeto de una clase determinada (de la API de Java o bien implementada por otros), instanciarlo con el operador **new** y utilizar sus miembros (métodos y atributos).

Para saber más

Puedes echar un vistazo a los artículos sobre la creación y uso de objetos en Java en los manuales de Oracle: [Creating Objects](#). Y [Using Objects](#).

8.1 DECLARACIÓN DE UN OBJETO.

Como ya has visto en unidades anteriores, la declaración de un objeto se realiza exactamente igual que la declaración de una variable de cualquier tipo:

```
<tipo> nombreVariable;
```

En este caso el tipo será alguna clase que ya hayas implementado (**Vehiculo**, **Persona**, **Rectangulo**, **Circulo**, **Reloj**, etc.) o bien alguna de las proporcionadas por la API Java (**String**, **StringBuilder**, **LocalDate**, etc.) o por alguna otra biblioteca escrita por terceros.

Por ejemplo, podríamos declarar algunas variables que apuntarían a objetos de diferente tipo (clases):

```
Vehiculo v1, v2, v3;
Persona p1, p2, p3;
Reloj r1, r2;
StringBuilder sb1, sb2
LocalDate f1, f2;
```

Esas variables (**v1**, **v2**, **v3**, **p1**, **p2**, **p3**, **r1**, **r2**, **sb1**, **sb2**, **f1**, **f2**) en realidad son referencias (también conocidas como punteros o direcciones de memoria) que apuntan (hacen "referencia") a un objeto (una zona de memoria) de la clase indicada en la declaración.

Como ya estudiaste en la unidad dedicada a los objetos, una variable de tipo referencia (objeto) recién declarada no apunta a nada. Se dice que la referencia está vacía o que es una referencia nula (la variable no contiene ninguna dirección a ninguna zona de memoria). Es decir, la variable existe y está preparada para guardar una dirección de memoria (que será la zona donde se encuentre el objeto al que hará referencia), pero el objeto aún no existe (no ha sido creado o instanciado). Por tanto se dice que apunta a un objeto nulo o inexistente.

Para que esa variable (referencia) apunte realmente a un objeto (contenga una referencia o dirección de memoria que apunte a una zona de memoria en la que se ha reservado espacio para un objeto) es necesario crear o instanciar el objeto. En Java se utiliza para ello el operador **new**.



Ejercicio resuelto

Utilizando la clase `Rectangulo` implementada en ejercicios anteriores, indica cómo declararías tres objetos (variables) de esa clase llamados `rectangulo1`, `rectangulo2`, `rectangulo3`.

[Mostrar retroalimentación](#)

Se trata simplemente de realizar una declaración de esas tres variables:

```
Rectangulo rectangulo1;
Rectangulo rectangulo2;
Rectangulo rectangulo3;
```

También podrías haber declarado los tres objetos en la misma sentencia de declaración:

```
Rectangulo rectangulo1, rectangulo2, rectangulo3;
```

Ahora bien la declaración de esas tres variables no significa que existan ya tres objetos instancia de la clase `Rectangulo`. Lo único que se ha hecho es declarar tres variables que en el futuro podrán contener referencias (direcciones de memoria, también conocidas como punteros) a zonas de la memoria donde se alojarán objetos instancia de la clase `Rectangulo`. Pero esas instancias aún no existen. Tendrán que ser creadas mediante la invocación al constructor por medio del operador `new`.

8.2 CREACIÓN DE UN OBJETO.

Para poder crear un objeto (instancia de una clase) es necesario utilizar el operador `new`, con la siguiente sintaxis:

```
nombreObjeto = new <ConstructorClase> ([listaParametros]);
```

El constructor de una clase (`<ConstructorClase>`) es un método especial que tiene toda clase y **cuyo nombre coincide con el de la clase**. Es el encargado de inicializar el objeto, llevando a cabo comprobaciones y asignando valores iniciales a los atributos de objeto si fuera necesario. Dado que el constructor es un método más de la clase, podrá tener también una lista de parámetros como tienen todos los métodos (ya lo hemos visto en el apartado sobre constructores).

De la tarea de reservar memoria para la estructura del objeto (los bytes ocupados por sus atributos más alguna otra información de carácter interno para el entorno de ejecución) se encarga el propio entorno de ejecución de Java al usar el operador `new`. Es decir, que por el hecho de invocar a un método constructor, el entorno sabrá que tiene que realizar una serie de tareas (solicitud de una zona de memoria disponible, reserva de memoria para los atributos, obtención de una referencia a la dirección de memoria de esa zona, etc.) y se pondrá inmediatamente a desempeñarlas.

Algunos ejemplos de instantiación o creación de objetos podrían ser:

```
v1 = new Vehiculo ("2827KPY", LocalDate.of(2018, 12, 12), 60.0, 5.0);
rectangulo1= new Rectangulo (0.0, 0.0, 5.0, 5.0, "r1", "black");
rectangulo2= new Rectangulo (rectangulo1);
r1= new Reloj ();
p1= new Persona ("Juan", "Gil", "Torres", "12345678L");
sb1= new StringBuilder ("cadena");
```

TEMA 5: DESARROLLO DE CLASES

Se puede declarar a la vez una variable referencia e invocar al constructor en la misma línea para así llevar a cabo el proceso de declaración y asignación inicial simultáneamente, de manera similar a como se hace con cualquier variable en Java. Por ejemplo:

```
Rectangulo rec1= new Rectangulo (0.0, 0.0, 5.0, 5.0, "r1", "black");
Rectangulo rec2= new Rectangulo (rec1);
```

También, como ya hemos visto, podemos obtener nuevas instancias de objetos con:

- métodos que generen nuevos objetos como resultado de aplicar una operación;
- métodos "pseudoconstructores", de los que ya hemos hablado.

Por ejemplo:

```
LocalDate f1= LocalDate.of (1944, 6 ,6); // Genera un nuevo LocalDate con la fecha 6 de junio de 1944
LocalDate f2= LocalDate.now(); // Genera un nuevo LocalDate con la fecha actual del sistema
LocalDate f3= f1.plusDays (3); // Obtiene un nuevo objeto LocalDate a partir de f1 (f1 + tres días)
Rectangulo recAleatorio= Rectangulo.random ("recRandom", "red"); // Genera un objeto Rectangulo con ubicación aleatoria
Vehiculo v1 = new Vehiculo ("2827KPY", LocalDate.of(2018, 12, 12), 60.0, 5.0);
Vehiculo v2= Vehiculo.clonar (v1); // Genera un nuevo vehículo "clon" exacto del objeto v1
Rectangulo[] arrayRecMuestra= Rectangulo.buildSampleRectangulos();
```

Si te fijas, en alguno de estos casos ha llegado a crearse no sólo una instancia sino todo un array de instancias de rectángulos de muestra.



Ejercicio Resuelto

Declara e instancia sendos objetos `per1` y `per2`, de la clase `Persona`, cuyos datos personales sean:

- ✓ para la primera persona: Luis Ruiz Ruz, 12345678Z,
- ✓ para la segunda persona: Ana Torres Sanz, 21212121A.

[Mostrar retroalimentación](#)

```
Persona per1= new Persona ("Luis", "Ruiz", "Ruz", "12345678Z");
Persona per2= new Persona ("Ana", "Torres", "Sanz", "21212121A");
```

Declara un **array** de tres elementos de la clase `Persona` y rellénalo con tres objetos de ese tipo con los siguientes datos:

- ✓ para la primera persona: Luis Ruiz Ruz, 12345678Z;
- ✓ para la segunda persona: Ana Torres Sanz, 21212121A;
- ✓ para la tercera persona: Tania Soto Leal, 11111111H.

Podríamos realizar todo ese proceso paso a paso:

- 1.- declararemos el array (de elementos de tipo `Persona`) y reservamos espacio para tres ítems;
- 2.- declararemos tres variables de tipo `Persona` e instanciamos los objetos correspondientes;
- 3.- asignamos a cada posición del array cada una de las referencias (variables) que acabamos de obtener al llamar a los constructores.

```
// Declaramos un array de elementos de tipo Persona
Persona[] arrayPersonas= new Persona[3]; // y reservamos espacio para 3 ítems

// Declaramos tres variables de tipo Persona (referencia a instancias de objetos Persona)
// instanciamos tres objetos Persona (usando new y el constructor)
// y asignamos la referencia (posición de memoria) que devuelve el operador new a cada una de esas variables referencia
Persona p1= new Persona ("Luis", "Ruiz", "Ruz", "12345678Z"); // Creación del primer objeto persona
Persona p2= new Persona ("Ana", "Torres", "Sanz", "21212121A"); // Creación del segundo objeto persona
Persona p3= new Persona ("Tania", "Soto", "Real", "11111111H"); // Creación del tercer objeto persona
```

TEMA 5: DESARROLLO DE CLASES

```
// Asignamos a cada una de las posiciones del array la referencia almacenada en cada una de las variables
// De esa manera cada posición del array también apuntará a cada una de las tres instancias recién creadas
arrayPersonas[0]= p1; // Asignación a la primera posición del array de la referencia al primer objeto persona
arrayPersonas[1]= p2; // Asignación a la segunda posición del array de la referencia al segundo objeto persona
arrayPersonas[2]= p3; // Asignación a la tercera posición del array de la referencia al tercer objeto persona
```

Aunque también podríamos haber compactado el código instanciando inicialmente los objetos `Persona` y en la declaración del array se podrían haber "metido" directamente usando las **llaves** de inicialización de un array (`{ ... }`):

```
Persona p1= new Persona ("Luis", "Ruiz", "Ruz", "12345678Z");
Persona p2= new Persona ("Ana", "Torres", "Sanz", "21212121A");
Persona p3= new Persona ("Tania", "Soto", "Real", "11111111H");
Persona[] arrayPersonas= {p1, p2, p3};
```

Una opción más directa y compacta aún es ahorrar el paso de asignar a las variables `p1`, `p2`, `p3` la referencia a los objetos recién instanciados con los constructores (mediante el operador `new`) y hacer la invocación a los constructores directamente dentro de las llaves de inicialización del array:

```
Persona[] arrayPersonas= {new Persona ("Luis", "Ruiz", "Ruz", "12345678Z"),
                           new Persona ("Ana", "Torres", "Sanz", "21212121A"),
                           new Persona ("Tania", "Soto", "Real", "11111111H")
};
```

Si no necesitamos las variables `p1`, `p2`, `p3` y tenemos claro en tiempo de compilación (al escribir el programa) todos esos datos, esta última forma sería la más sencilla y legible de hacerlo.

¿Se te ocurre alguna manera de encapsular el código anterior para implementar un método llamado `BuildSamplePersonas` que genere automáticamente todo ese array y lo devuelva para poder utilizarlo en programas de prueba?

Pista: podríamos hacer algo similar a lo que se hizo en el método `BuildSampleRectangulos` de la clase `Rectangulo`.

Lo único que habría que hacer sería realmente encapsular dentro de un método ese fragmento de código donde se realizan las reservas de memoria para los objetos `Persona` y para el array, devolviéndose ese array mediante una sentencia `return`.

Lo razonable sería que este método fuera **de clase (estático)** pues no se interacciona con los atributos de ningún objeto en particular. De hecho, no hace falta que existan objetos de tipo `Persona` para poder invocar a este método, que es algo así como un "superconstructor", pues `instancia` no ya un objeto `Persona` sino varios.

El código del método podría quedar así:

```
public static Persona[] buildSamplePersonas() {
    // Creamos el array de personas con todos los objetos Persona de ejemplo dentro
    Persona[] arrayPersonas= {new Persona ("Luis", "Ruiz", "Ruz", "12345678Z"),
                               new Persona ("Ana", "Torres", "Sanz", "21212121A"),
                               new Persona ("Tania", "Soto", "Real", "11111111H")
    };
    return arrayPersonas; // Devolvemos ese array
}
```

El método podría usarse desde un programa de pruebas de la siguiente manera:

```
Persona[] personasPrueba= Persona.buildSamplePersonas();
```

8.3 REFERENCIAS A UN OBJETO.

Como ya hemos indicado en anteriores ocasiones, lo que contiene una variable de tipo referencia (variable que no es de tipo primitivo) es una referencia (algunos lo llaman puntero) a una zona de memoria donde se encuentran alojados los atributos de un objeto instancia de una clase.

Eso significa que en la variable de tipo "referencia" u "objeto" no está almacenado el objeto en sí (la sucesión de todos los bytes que representan la información que contienen los atributos), sino un número (que nosotros llamamos referencia, dirección de memoria o puntero) que indica dónde están todos esos bytes con los datos de los atributos. Una analogía podría ser la de que la variable referencia es un casillero donde se guarda la llave de una habitación. La habitación con todo su contenido sería el objeto con sus atributos dentro y el casillero sería la variable referencia que contiene la dirección de memoria o referencia (llave de la habitación) a donde está el objeto (habitación).

Por tanto, cuando se realiza una asignación de un objeto de tipo referencia a otro objeto de tipo referencia, lo único que estamos haciendo es copiar y asignar referencias (direcciones de memoria) pero no haciendo copias de los objetos. Por ejemplo, cuando ejecutábamos la sentencia

```
Persona p1= new Persona ("Juan", "Gil", "Torres", "12345678L");
```

Podíamos considerar que se llevaban a cabo tres pasos:

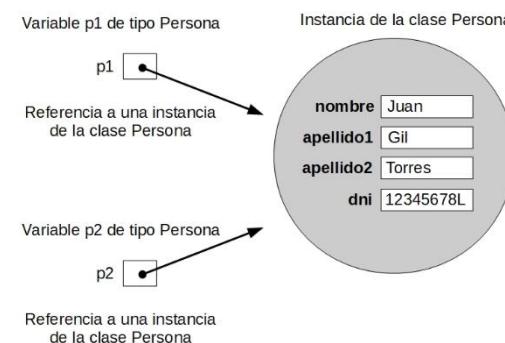
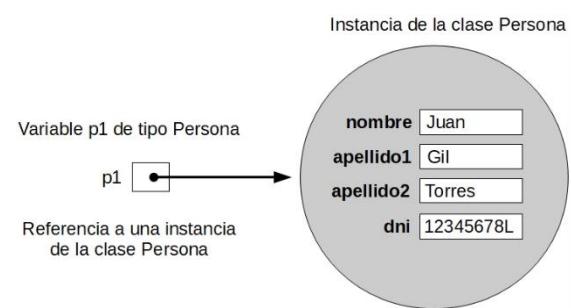
1. se declaraba la variable **p1** de tipo referencia a objetos instancia de la clase **Persona**;
2. se instanciaba un objeto de la clase **Persona** con las características "Juan", "Gil", "Torres", "12345678L" (invocación al constructor a través del operador **new**);
3. se asignaba a la variable **p1** la referencia que devuelve el operador **new** tras ejecutar el código del constructor y reservarse una zona de memoria para almacenar los atributos de un nuevo objeto instancia de la clase **Persona**.

Es decir, que hemos hecho un **new** y por tanto tenemos un nuevo objeto **Persona**. Y como además hemos hecho una asignación a **p1**, podemos decir que **p1** "apunta" a ese objeto recién creado. Eso es más correcto que decir que **p1** "contiene" al objeto.

```
Persona p2= p1;
```

Si a continuación hacemos:

No estaremos creando un nuevo objeto Persona, sino que estaremos asignando a **p2** la misma referencia que contiene **p1**. De este modo, aunque **p1** y **p2** son dos variables distintas (podemos imaginarlas como dos casilleros distintos) están apuntando al mismo objeto (podríamos decir que ambos contienen una llave para una misma habitación). Eso significa que cualquier acción que se lleve a cabo sobre el objeto apuntado por **p1** también se estará llevando a cabo sobre el objeto apuntado por **p2**, pues no son más que dos referencias al mismo objeto. No se trata de objetos diferentes.

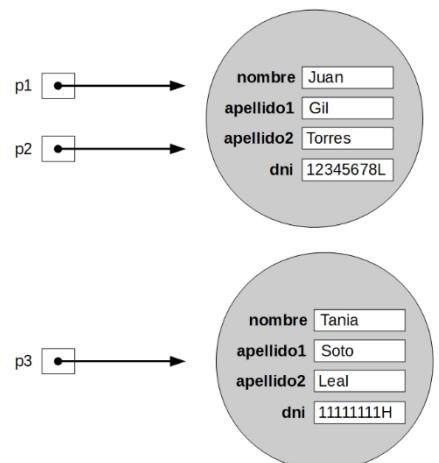


TEMA 5: DESARROLLO DE CLASES

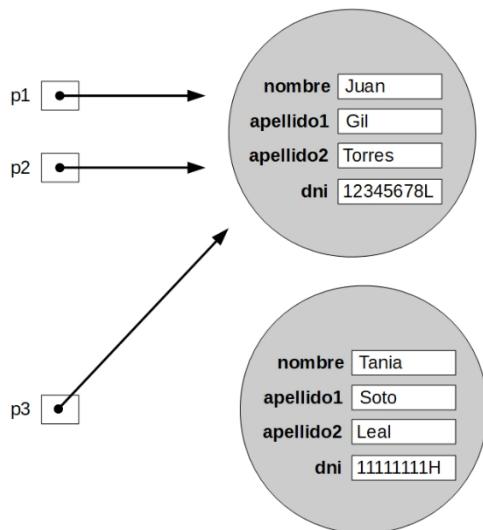
Si queremos tener un segundo objeto de tipo **Persona**, tendríamos que hacer un segundo **new** (o bien una invocación a algún método que devuelva un nuevo objeto instancia de la clase **Persona** como por ejemplo un "pseudoconstructor") y asignarlo a alguna variable de tipo referencia a la clase **Persona**. Por ejemplo:

```
Persona p3= new Persona ("Tania", "Soto", "Leal", "11111111H");
```

En este caso la variable **p3** apuntaría a un nuevo objeto de tipo **Persona** diferente al que apuntan **p1** y **p2**.



¿Qué sucede si en un momento dado **p3** deja de apuntar a su objeto original y se le asigna por ejemplo el valor de **p2**? **p3= p2;**



No habría problema, **p3** cambiará de valor y a partir de ese momento las variables **p1**, **p2** y **p3** apuntarían al mismo objeto. Por otro lado, el objeto al que apuntaba originalmente **p3** se habrá quedado "perdido", no existe ninguna variable referencia que apunte a él, y por tanto no hay manera de "alcanzarlo". El recolector de basura de Java (garbage collector) acabará borrándolo de la memoria, pues se trata de un elemento irrecuperable desde el programa y por tanto inútil.

8.4 MANIPULACIÓN DE UN OBJETO: UTILIZACIÓN DE MÉTODOS Y ATRIBUTOS.

Una vez que un objeto ha sido declarado y creado (clase instanciada) ya sí se puede decir que el objeto existe en el entorno de ejecución, y por tanto que puede ser manipulado como un objeto más en el programa, haciéndose uso de sus atributos y sus métodos públicos.

Para acceder a un miembro de un objeto se utiliza el operador **punto (.)** del siguiente modo:

Donde **<nombreMiembro>** será el nombre de algún miembro del objeto (atributo o método) al cual se tenga acceso.

Por ejemplo, en el caso de los objetos de tipo **Círculo**, podríamos acceder a sus miembros de la siguiente manera:

```
Circulo c1, c2, c3;
System.out.printf ("Cantidad de círculos creados hasta el momento: %d.\n", Circulo.numeroCirculos);
c1= new Circulo (0.0, 0.0, 0.25, "c1", "black");
System.out.printf ("Propiedades del círculo c1: %s.\n", c1);
System.out.printf ("Superficie del círculo c1: %.2f.\n", c1.calculaSuperficie());
System.out.printf ("Longitud de la circunferencia que rodea al círculo c1: %.2f.\n", c1.calculaArea());
System.out.printf ("Cantidad de círculos creados hasta el momento: %d.\n", Circulo.numeroCirculos);
c2= new Circulo();
c2.setX(0.50);
c2.setY(0.25);
System.out.printf ("El centro del círculo c2 está en (%.2f,%2f).\n", c2.getX(), c2.getY());
System.out.printf ("El nombre y el color del círculo c2 son: \"%s\" y \"%s\".\n", c2.getNombre(), c2.getColor());
System.out.printf ("Cantidad de círculos creados hasta el momento: %d.\n", Circulo.numeroCirculos);
```

Es decir, colocando el operador **punto (.)** a continuación del nombre del objeto y seguido del nombre del miembro público al que se desea acceder. Obviamente, si se trata de un miembro estático (atributo de clase o método de clase como por ejemplo **numeroCirculos**) entonces habrá que escribir el nombre de la clase (**Círculo.numeroCirculos**) y no del objeto (**c1.numeroCirculos**).

Escribe el código Java necesario para:

- 1.- mostrar por pantalla cuántos objetos instancias de la clase **Rectangulo** hay creados hasta el momento;
- 2.- crear un objeto instancia de la clase **Rectangulo** ubicado en los coordenadas (0.0, 0.0), (5.0, 5.0), con nombre "Bloque Uno" y color "black";
- 3.- volver a mostrar por pantalla cuántos objetos instancias de la clase **Rectangulo** hay creados hasta el momento;
- 4.- mostrar una representación textual del objeto por pantalla;
- 5.- calcular y mostrar por pantalla su área y la longitud de su perímetro;
- 6.- modificar las coordenadas del vértice superior derecho a (2.0, 2.0);
- 7.- volver a mostrar una representación textual del objeto por pantalla;
- 8.- volver a calcular y mostrar por pantalla su área y la longitud de su perímetro;
- 9.- mostrar una vez más por pantalla cuántos objetos instancias de la clase **Rectangulo** hay creados hasta el momento;
- 10.- crear otro objeto **Rectangulo** exactamente igual que el primer rectángulo creado;
- 11.- cambiar el atributo nombre de ese segundo objeto a "Bloque Dos" y el color a "red";
- 12.- mostrar por pantalla (representación textual) los datos de ese nuevo objeto;
- 13.- mostrar por pantalla por última vez cuántos objetos instancias de la clase **Rectangulo** hay creados hasta el momento.

```
System.out.printf ("Cantidad de rectángulos creados hasta el momento: %d.\n", Rectangulo.cantidadRectangulos);
Rectangulo r1= new (0.0, 0.0, 5.0, 5.0, "Bloque Uno", "black");
System.out.printf ("Cantidad de rectángulos creados hasta el momento: %d.\n", Rectangulo.cantidadRectangulos);
System.out.printf ("Atributos del rectángulo r1: %s.\n", r1);
System.out.printf ("Superficie del rectángulo r1: %.2f.\n", r1.calcularSuperficie());
System.out.printf ("Longitud del perímetro del rectángulo r1: %.2f.\n", r1);
r1.setX2 (2.0);
r1.setY2 (2.0);
System.out.printf ("Atributos del rectángulo r1: %s.\n", r1);
System.out.printf ("Superficie del rectángulo r1: %.2f.\n", r1.calcularSuperficie());
System.out.printf ("Longitud del perímetro del rectángulo r1: %.2f.\n", r1);
System.out.printf ("Cantidad de rectángulos creados hasta el momento: %d.\n", Rectangulo.cantidadRectangulos);
Rectangulo r2= new (r1);
r2.setNombre ("Bloque Dos");
r2.setColor ("red");
System.out.printf ("Atributos del rectángulo r1: %s.\n", r1);
System.out.printf ("Cantidad de rectángulos creados hasta el momento: %d.\n", Rectangulo.cantidadRectangulos);
```

8.5 RECOGIENDO LAS EXCEPCIONES LANZADAS POR UN MÉTODO.

Cada vez que utilicemos un método de una clase (incluido el constructor) debemos revisar si existe la posibilidad de que puedan lanzarse excepciones desde ese método en caso de que se produzca alguna situación anómala.

Si estamos ante ese caso, habrá que encerrar la llamada a ese método en un bloque **try - catch** (o **try - catch - finally**).

Veamos un ejemplo en el que se intentan crear y manipular algunos objetos instancias de la clase **Vehiculo**:

```
Vehiculo v1, v2, v3;
try {
    v1= new Vehiculo ("4650CKE", LocalDate.now().plusDays(2), 60.0, 5.0); // Fallará por la fecha
    System.out.printf ("Vehículo v1: %s\n", v1);
} catch (IllegalArgumentException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage()); // Fallo por fecha inválida (fecha futura)
}
try {
    v2= new Vehiculo ("7829KPY", LocalDate.of (2018, 10, 10), 60.0, 5.0);
    v2.arrancar(); // Fallará al intentar arrancar sin combustible
    System.out.printf ("Vehículo v1: %s\n", v1);
} catch (IllegalArgumentException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage());
} catch (IllegalStateException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage()); // Fallo por falta de combustible
}
try {
    v3= new Vehiculo ("1221CGW", LocalDate.of (2003, 1, 1), 60.0, 5.0);
    v3.repostar(10.0);
    System.out.printf ("Vehículo v1: %s\n", v1); //No se producirán fallos
} catch (IllegalArgumentException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage());
} catch (IllegalStateException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage());
}
try {
    v4= new Vehiculo ("2128JZP", LocalDate.of (2017, 1, 6), 60.0, 5.0);
    v4.repostar(30.0);
    v4.recorrerTrayecto (100.0);
    v4.parar();
    System.out.printf ("Vehículo v1: %s\n", v1); // No se producirán fallos
} catch (IllegalArgumentException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage());
} catch (IllegalStateException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage());
}
```

En general, cada vez que utilices algún mecanismo del vehículo que pueda dar lugar a alguna situación "delicada", tendrás que proteger esa sección de código con un bloque **try - catch** para asegurarte de poder tomar el control ante el fallo y decidir qué hacer a partir de ese momento. Si tu programa no toma el control (mediante un **catch**), esa excepción será enviada a quien llamó a tu programa (la máquina virtual de Java) y el programa abortará su ejecución interrumpiéndose bruscamente. Eso es una circunstancia nada deseable y que debemos evitar por todos los medios. **Un programa que interrumpe bruscamente su ejecución ante un fallo es un programa inestable en el que no podemos confiar.**

Nuestros programas deben ser lo suficientemente robustos como para no "romperse" cuando se produce un error. Deben prever cualquier posible error y tener prevista su gestión.