Programación

Bloque 02 - Introducción a la programación orientada a objetos

Índice

1.- Introducción....2
2.- La programación orientada a objetos....2
  2.1.- La realidad y los modelos....2
  2.2.- Programación orientada a objetos....3
    2.2.1.- Estado de un objeto....3
    2.2.2.- Comportamiento de un objeto....4
  2.3.- Clases de objetos....4
  2.4.- Interfaz de un objeto....5
  2.5.- Implementación....6
  2.6.- Objetos como proveedores de servicio....6
3.- Objetos en Java....7
  3.1.- Creación de objetos en Java....7
  3.2.- Referencias a objetos en Java....8
    3.2.1.- El valor especial null....9
    3.2.2.- Acceso directo a los atributos de un objeto....10
    3.2.3.- Copia de referencias....10
  3.3.- Uso de servicios de un objeto....11
    3.3.1.- Uso de un mensaje....12
    3.3.2.- Paso de parámetros....12
    3.3.3.- Devolución de resultados....15
    3.3.4.- Acceso indirecto a los atributos....16
  3.4.- Elementos estáticos....17
  3.5.- Constructores....18
4.- Uso de objetos de las librerías estándar: String....20
  4.1.- Creación de objetos String....20
  4.2.- Lectura de cadenas desde el teclado....21
  4.3.- Concatenación de cadenas....21
  4.4.- Mensajes de información....21
  4.5.- Mensajes de comparación....21
  4.6.- Mensajes de extracción....22
5.- Librerías de objetos....23
  5.1.- Inclusión de librerías....23
  5.2.- Uso de librerías....24
    5.2.1.- Paquetes de clases....24
    5.2.2.- Uso de clases. Nombre cualificado....25
    5.2.3.- Uso de clases. Importación....25
6.- Destrucción de objetos y liberación de memoria....26
7.- Resumen....27
8.- Referencias....27

1.- Introducción

La programación es una labor que se realiza sobre una máquina (el ordenador). Por lo tanto, en los inicios de la programación, los lenguajes de programación y los programas que se escribían con ellos estaban orientados a la forma de funcional de la máquina.

Durante estos años de evolución de la programación han aparecido (y se han dejado de usar) varios paradigmas o modos de enfocar la tarea completa de la programación, desde los inicios, en que no había paradigma alguno y la programación era vista como un arte esotérico, pasando por la programación estructurada, la programación lógica, programación funcional, etc.

El paradigma que actualmente es el más extendido y reconocido es el de la programación orientada a objetos. En esta unidad presentaremos los conceptos básicos de la programación orientada a objetos, asi como la forma de usar objetos en Java.

2.- La programación orientada a objetos

La programación es una tarea compleja. Requiere mucho tiempo el crear una aplicación de una complejidad media y la dificultad crece bastante con la complejidad de aquello que se pretende programar.

Asimismo, la mayor parte del ciclo de vida de una aplicación transcurre *modificando* la aplicación, más que en la creación inicial de la misma, especialmente en caso de que la aplicación tenga éxito.

Por lo tanto se debe intentar enfocar la programación de forma que no sólo se construyan aplicaciones que funcionen, sino que se debe intentar también que sean lo más fáciles posibles de construir y que asimismo sean lo más fáciles de modificar una vez construidas.

Para ello han surgido a lo largo de la corta historia de la programación varias metodología o paradigmas que describen la forma en que los programadores deben enfocar el proceso de construcción y mantenimiento de las aplicaciones.

De todas estas metodologías, la más ampliamente utilizada en los últimos tiempos es la metodología orientada a objetos.

2.1.- La realidad y los modelos

La programación se puede ver como un proceso de *modelado de la realidad*. Un modelo es una representación simplificada de algo real que se utiliza para un propósito o propósitos. Por ejemplo, para probar la eficiencia aerodinámica de un coche (lo bien que corta el aire) se emplean unos aparatos llamados túneles de viento. En estos túneles, potentes ventiladores hacen el papel de viento y se mide como se comporta el coche bajo esas condiciones. Esta claro que para probar el coche se podría usar un vehiculo recien salido de fábrica pero si lo piensas un poco llegarás a la misma conclusión que los ingenieros de coches: ¿Para qué? ¿Va a influir en algo el motor, las luces, si tiene o no gasolina el coche en el desarrollo de la prueba? Aparte de en el peso parece que de poco. Es por eso que los ingenieros hacen pruebas con un *modelo* del coche del cual se eliminan todas las partes innecesarias (y costosas) que no influyen en la prueba. El peso se puede simular llenando el coche de sacos de arena. Lo que sí es importante es que la *forma* del coche sea exactamente igual al original de forma que se puedan extrapolar los datos obtenidos.

Como puedes ver, el proceso de modelar es tomar algo que existe y crear una versión simplificada con un uso concreto.

La programación se puede ver como un proceso de modelado puesto que se toma una tarea manual ya existente y se crea un modelo simplificado que funciona sobre el ordenador.

Por lo tanto la programación exige observar de forma sistemática la realidad que se quiere modelar y obtener un modelo que funcione sobre el ordenador y que se comporte de forma lo más parecida posible a la realidad, siempre teniendo en cuenta que una aplicación se hace para unos propósitos concretos y que, por tanto, no debe imitar al 100% a la realidad.

Por ejemplo, para una aplicación que gestiona una agencia de alquiler de coches, se tendrán **clientes**, que son personas que alquilan coches. De estos clientes necesitaremos conocer cierta información (nombre, DNI, método de pago, etc.) pero otros datos serán completamente irrelevantes, como el color de ojos, número de hijos o estado civil. ¿Quiere decir esto que las personas que estamos modelando en nuestro programa no tienen esta información? No. Lo que queremos decir es que esta información no es relevante para nuestro modelo de persona. Otro ejemplo es de los *dummies*. Estos son muñecos que se colocan dentro de un coche sobre el que se van a realizar pruebas de accidentes (choques, volcados, etc.) El objetivo es determinar como afectarían estos accidentes a los pasajeros humanos reales. Obviamente no se pueden hacer pruebas con personas así que se usan estos dummies que son modelos de personas de verdad pero que no tienen todas sus características, sólo las importantes para las pruebas (peso, movilidad, etc.)

2.2.- Programación orientada a objetos

En la programación orientada a objetos se intenta crear un modelo informático de la realidad (una aplicación) como una colección de *objetos* que interactúan entre si, a semejanza de la realidad

Un objeto es una *cosa* que existe y que es relevante para nuestra aplicación. Cuando analizamos el problema a resolver, examinamos la realidad en busca de cosas que aparezcan en el ámbito del problema y creamos una versión simplificada en nuestro programa que sea lo más parecida posible a la cosa real, siempre teniendo en cuenta las necesidades de la aplicación (recuerda los ejemplos de los dummies o los clientes: no es necesario implementar al 100% la cosa del mundo real, sólo lo que nos es necesario para nuestra aplicación).

En la programación orientada a objetos, a estas *cosas* las llamaremos *objetos*.

En programación, todo objeto tiene tres características fundamentales: Identidad, Estado y Comportamiento.

La propiedad de la Identidad especifica que **cada objeto de un sistema puede ser distinguido de forma única de cualquier otro objeto del mismo sistema**. Esto significa que dos objetos que aparentemente son iguales siempre se podrán distinguir uno de otro. Por ejemplo, imaginemos que tenemos en nuestra aplicación dos coches de la misma marca, modelo, color, combustible, motor, etc. Aunque los dos coches parezcan el mismo no lo son: son dos coches idénticos.

2.2.1.- Estado de un objeto

Un objeto, como modelo de algo real, puede tener datos que son relevantes para la aplicación. Por ejemplo, un vehículo puede tener marca, modelo, tipo de motor (eléctrico, híbrido de gasolina, etc.) color, número de plazas, etc. A estos datos se les denomina **atributos** del objeto. Son el conjunto de

datos que son relevantes para el objeto en cuestión en el contexto de nuestra aplicación. Otra aplicación que trabaje con coches tendrá objetos *coche* que tengan otros atributos, adaptados al contexto de la nueva aplicación. Es posible que algunos aparezcan en ambas aplicaciones pero eso es pura coincidencia.

Los valores contenidos en los atributos de un objeto pueden cambiar a lo largo del tiempo con el objetivo de reflejar de la forma más fiel posible los cambios en la realidad. Por ejemplo, un cliente puede tener un atributo llamado *dirección*, que contenga la dirección postal del mismo. Si el cliente se muda, es obvio que el contenido de este atributo se deberá modificar para reflejar la nueva realidad.

Al conjunto de los atributos de un objeto, junto con los valores concretos contenidos en los mismos se le denomina **estado** de un objeto. Como ya hemos visto, el estado de un objeto puede cambiar a lo largo del tiempo.

Usualmente los nombres de los atributos son sustantivos (o sea nombres) ya que indican propiedades de los objetos.

2.2.2.- Comportamiento de un objeto

El comportamiento de un objeto define que es lo que éste puede hacer. Esto se define mediante operaciones, que en programación orientada a objetos se denominan **mensajes**. Un objeto especifica una serie de mensajes que es capaz de recibir desde otros objetos del sistema. Cuando un objeto recibe un mensaje determinado, realiza la operación que se le solicita. Un objeto NO puede recibir mensajes que no esté preparado para recibir o que no estén definidos. Esto tiene logica pues un objeto sólo puede hacer cosas para las que esté diseñado.

Por ejemplo, un objeto *coche* puede recibir los mensajes arrancar (que arranca el motor, si no estuviera ya arrancado), parar (que para el motor, si estuviera en marcha), mover (que mueve el coche hacia delante) y frenar (que detiene el coche).

Usualmente los nombres de los mensajes son verbos ya que se refieren a *acciones* a realizar.

2.3.- Clases de objetos

Cuando examinamos cuidadosamente la realidad nos daremos cuenta de que aunque todos los objetos (las cosas) son únicos, también es posible agruparlos en grupos de objetos que tienen características comunes. Por ejemplo, mientras que dos clientes son objetos distintos es cierto que tienen características y comportamientos comunes. Por ejemplo, todos los clientes tienen un dato (atributo) nombre, otro llamado DNI, otro llamado edad, etc. Asimismo, todos los clientes pueden realizar las mismas acciones con idéntico resultado (por ejemplo, un cliente puede *alquilar* un coche).

La idea es que todos los objetos que forman parte de un sistema, aunque únicos, también forman parte de grupos de objetos que tienen las mismas características (atributos) y comportamiento. En programación orientada a objetos, a estos grupos se les denomina **clases**.

Una clase describe un conjunto de objetos que tienen características identicas (atributos y comportamiento). A cada objeto que pertenece a una clase determinada se le llama **instancia** de dicha clase. Por ejemplo, el coche de Pepe es una instancia de la clase coche. Mi coche también es otra instancia de esa misma clase y todos los coches que existen en nuestro sistema.

El concepto de clase es muy potente porque nos permite especificar las características de un grupo de objetos más o menos grande especificando únicamente una sóla vez las características comunes a todos los objetos de la clase. Por ejemplo, ya que todos los clientes tienen un atributo llamado `nombre`, la definición de este se realizará al definir la clase `cliente` y a partir de ese momento ya se sabe que TODOS los clientes tienen ese atributo puesto que todos pertenecen a la clase. No es necesario indicarlo de uno en uno para cada cliente distinto.

El concepto no está libre de problemas, sin embargo, puesto que también limita la "libertad" de los objetos de la clase al obligar a todas las instancias de la clase a tener la forma indicada en la misma. Para permitir esta flexibilidad aparecen conceptos mas avanzados como la herencia, que discutiremos en unidades posteriores.

Cuando creamos una nueva clase estamos creando un nuevo tipo de datos similar a los tipos predefinidos (int, long, double, etc.) pero con la información y comportamiento que nosotros queremos.

Una vez que hemos definido una clase podremos crear tantos objetos pertenecientes a la misma como queramos y manipularlos de la misma forma que se manipularía la cosa que se representa en el mundo real, sólo que de forma más simple ya que son modelos simplificados de la realidad.

Uno de los retos del desarrollo orientado a objetos es el observar la realidad y obtener objetos en el ordenador que se comporten, a efectos de nuestra aplicación, como los objetos reales.

2.4.- Interfaz de un objeto

Para que un objeto sea util para el resto de objetos del sistema es necesario que aquel pueda realizar algún trabajo útil para el que lo usa u ofrecer información útil a otros objetos.

Esto implica que es necesario que exista una forma en que un objeto pueda realizar peticiones a otro de forma que éste haga algo tal y como alquilar un coche o encender una bombilla. Por la naturaleza misma de un objeto, éste sólo puede responder a determinadas peticiones. Por ejemplo, a una bombilla se le puede decir que se encienda o se apague pero no se le puede pedir que se mueva (por ahora) o que alquile un coche.

Las peticiones que se le pueden realizar a un objeto determinado constituyen lo que se denomina el **interfaz** del objeto.

Asimismo, el interfaz también podría contener atributos que se ofrecen a otros objetos de forma que estos puedan leer o escribir en ellos. Esto último, aunque posible, está fuertemente desaconsejado, como se intentará explicar en siguientes secciones.

Por lo tanto, el interfaz de un objeto, definido en la clase a la que pertenece, indica de forma precisa las peticiones que se pueden hacer a un objeto perteneciente a esa clase asi como los atributos que se exponen a otros objetos. Sólo las operaciones y atributos que aparecen en el interfaz pueden ser utilizadas desde otros objetos y es un error realizar una petición a un objeto de una operación o atributo que no aparece en su interfaz.

Cuando diseñemos una aplicación y creemos nuestras propias clases que usen otras ya existentes, necesitaremos conocer los interfaces de las clases que vamos a usar a fin de saber cómo se usan los objetos de esas clases y que cosas pueden hacer por nosotros.

2.5.- Implementación

El interface de un objeto determina *qué* puede hacer un objeto. Sin embargo, ¿cómo hace un objeto lo que se le pide? O dicho de otra forma: cuando se envía un mensaje a un objeto, ¿cómo hace este la tarea que se le pide? Las instrucciones que realizan estas tareas, junto con algún posible dato necesario para poder llevarlas a cabo constituyen lo que se denomina la **implementación** del objeto.

La implementación, que forma las "tripas" de un objeto, debe permanecer lo más oculta posible al resto de objetos que pertenecen a otras clases. Lo único que debe conocer un objeto de una clase de otros objetos de otras clases es su interface. De esta forma, mientras que se mantenga el interface sin cambios, se puede cambiar la implementación, a veces completamente, sin afectar al resto de objetos del sistema.

Un ejemplo. Supongamos que tenemos una empresa de transportes que dispone de varias furgonetas iguales para el reparto de mercancías. Todas las furgonetas disponen de un motor de gasolina y tienen cambio de velocidad automático. Cualquier empleado de la empresa es capaz de conducir cualquier furgoneta, de forma que cuando un empleado necesita una, toma una cualquiera de las que queden libres.

Un buen día la empresa decide ampliar el parque y compra más furgonetas, pero, queriendo ser más respetuosos con el medio ambiente, decide que las nuevas furgonetas serán eléctricas, en lugar de tener un motor de gasolina.

¿Qué cambios son necesarios realizar en la empresa, aparte del hecho de deber disponer de enchufes para cargar las nuevas furgonetas? ¿Deberán los empleados cambiar de alguna forma para adaptarse a las nuevas circunstancias?

La respuesta, obviamente, es no. Los empleados que sabían conducir las furgonetas antiguas serán capaces de conducir inmediatamente las nuevas. ¿Por qué es esto? Esto es debido a que tanto las antiguas furgonetas como las nuevas disponen del mismo interfaz (volante a un lado u otro para girar, pedal de freno y pedal de aceleración). Sin embargo, de forma interna, las furgonetas funcionan de forma completamente distinta (quemando combustible unas, usando energía de baterías y motores eléctricos las otras). La *implementación* de los dos grupos de furgonetas es distinta pero su interfaz es el mismo lo que permite que los empleados puedan usar unas u otras sin problemas.

Si cambiara el interfaz, sin embargo, la cosa sería distinta. Imagínate que compran nuevas furgonetas con cambio de marchas manual. Puede que algunos (o muchos) de los empleados tengan problemas para manejar las nuevas furgonetas debido a que su interfaz ha cambiado (ahora tienen una palanca para seleccionar la velocidad y un pedal, el embrague, para ordenar el cambio).

2.6.- Objetos como proveedores de servicio

Otro punto de vista sobre lo discutido en los apartados anteriores es el de considerar a los objetos como proveedores de servicio. Una aplicación ofrece (provee) servicios a sus usuarios y lo hace utilizando los servicios que proporcionan los objetos que lo componen. Estos, a su vez, usan los servicios de otros objetos y asi sucesivamente, formando una red de objetos que colaboran entre si para realizar las funciones para las que se ha diseñado la aplicación.

El objetivo a la hora de crear una aplicación es crear objetos que proporcionen los servicios que se necesitan para crear la misma. Una opción aún mejor es buscar y localizar objetos ya creados que ya proporcionen algunos (o todos) los servicios que se necesitan, de forma que tengamos que trabajar lo menos posible.

Una forma de plantear esta forma de trabajar es preguntarse lo siguiente: "Si pudiera sacarlos de una chistera mágica, ¿que objetos resolverían mi problema ahora mismo?" Imagina que estamos creando una aplicación de facturación de un comercio. Puedes imaginarte que ya existen objetos que producen y gestionan pantallas de interacción con el usuario para registrar ventas y compras de artículos, otros objetos que realizan los cálculos sobre estas ventas y compras e incluso otros que imprimen tickets, facturas o albaranes en cualquier impresora. Puede que, si investigas lo suficiente, encuentres que algunos de estos objetos ya existen y los puedes tomar y utilizar. ¿Qué pasa con los objetos que NO puedes encontrar? Esos serían los que te tocaría desarrollar. Para estos objetos debes preguntarte: ¿Qué servicios deberían ofrecer estos objetos? ¿Qué otros objetos se necesitarían para ayudar a aquellos a proporcionar los servicios? Si haces esto de forma iterativa llegará un punto en que todos los objetos detectados serán, o bien lo bastante simples para escribirlos tu mismo, o bien existen ya y los puedes reutilizar. Esta es una manera de descomponer una aplicación en un conjunto de objetos.

El ver los objetos como proveedores de servicios proporciona otra ventaja adicional. Los objetos que se obtengan usando este método tendrán una gran cohesión interna. Un objeto tiene cohesión interna cuando sólo hace una cosa y la hace bien. Un objeto con poca cohesión interna hace varias cosas y las distintas cosas que hace no casan bien entre si. Los objetos con mucha cohesión interna son más fáciles de reutilizar en el futuro y más fáciles de mantener, por lo que es algo que merece la pena buscar a la hora de desarrollar aplicaciones.

3.- Objetos en Java

En esta sección entraremos al detalle de cómo crear y utilizar objetos de clases ya existentes, usando el lenguaje Java. En próximos bloques de contenido aprenderemos a crear nuestras propias clases y objetos.

A fin de poder realizar una mejor exposición de los siguientes apartados vamos a suponer que tenemos una clase `Persona` ya existente con el siguiente interfaz:

- Atributos:
  - `edad`. `int`
- Mensajes: Ninguno (por ahora)

3.1.- Creación de objetos en Java

En Java la creación de objetos se realiza con la instrucción `new`. Esta instrucción recibe un nombre de clase y devuelve un nuevo objeto (una instancia) perteneciente a dicha clase.

Por ejemplo, la instrucción:

```
new Persona();
```

crea un nuevo objeto perteneciente a la clase `Persona` y lo devuelve.

La instrucción anterior, por si misma es bastante inútil porque aunque crea un nuevo objeto no se hace nada con él. Para resultar de utilidad, el nuevo objeto se debe almacenar en una variable, tal y como hacíamos con los valores de tipo primitivo (`int`, `long`, `double`, etc.).

A fin de poder crear una variable que contenga un objeto de una clase hay que declarar la variable usando el nombre de la clase en cuestión, de la misma forma que para declarar una variable entera usábamos el tipo `int`, por ejemplo.

Por lo tanto el ejemplo anterior lo ampliaríamos de la siguiente forma:

```
Persona persona = new Persona();
```

Esta nueva instrucción consiste realmente de dos instrucciones en una. Por un lado la parte:

```
.... = new Persona()
```

crea el nuevo objeto de la clase Persona y por otro la parte

```
Persona persona = ......
```

toma el objeto recién creado y lo almacena en la nueva variable llamada `persona`. A partir de ese momento ya tenemos en la variable `persona` un objeto de la clase `Persona` listo para usar enviándole mensajes o accediendo a sus atributos.

Seguro que te estarás preguntando. Si ha dicho que para crear un objeto se usa `new` y el nombre de la clase, ¿qué pintan los paréntesis `()` después del nombre de la clase?

Los paréntesis forman parte del mecanismo que permite proporcionar a Java información adicional sobre cómo crear el nuevo objeto. Por ahora simplemente los colocamos tal y como se ve en el ejemplo y más tarde ampliaremos la información sobre ellos.

Es importante indicar que podemos crear todos los objetos que necesitemos en nuestro programa. El único límite es el espacio en memoria principal disponible que, para aplicaciones del tamaño que vamos a hacer en este curso, es suficiente.

3.2.- Referencias a objetos en Java

Llegados a este punto es importante que ampliemos el concepto de variable que vimos en bloques de contenido anteriores para acomodar los objetos y otros tipos que veremos más adelante como los arrays.

Cuando vimos las variables anteriormente, todas ellas eran de *tipos primitivos* (`int`, `long`, `double`, `boolean`, etc). Esto significa que la variable es un espacio en memoria que contiene el `valor real` de lo que contiene. Por ejemplo, si tenemos el código Java:

```
int numero = 1;
```

Java crea un espacio en memoria suficiente para almacenar un entero y almacena el valor 1 en él.

Si ahora hacemos lo siguiente:

```
int otraVariable = 1;
```

Java crea otro espacio en memoria suficiente para almacenar un entero y almacena el valor 1 en él.

Es importante recalcar que los valores de ambas variables son independientes. Es decir, aunque en `numero` esté almacenado el valor 1 y en `otraVariable` esté almacenado el valor 1, si cambiamos el valor de `numero` o de `otraVariable` por otro (por ejemplo 2), el otro no se verá afectado en absoluto y seguirá valiendo 1.

Para el caso de los objetos, sin embargo esto no funciona asi. Cuando en Java se crea un objeto, de la clase que sea, Java reserva espacio suficiente en memoria para contener a información del objeto (este espacio dependerá del número y tipo de los atributos que tenga el objeto, principalmente, aunque contendrá información adicional para uso interno de Java) y nos devuelve una **referencia** al objeto. Una referencia es un número identificador único que se asigna a cada objeto automáticamente en el momento de su creación y sirve para identificarlo entre todos los objetos existentes en la aplicación. Lo podrías ver, si quieres, como el "DNI" del objeto.

Cuando creamos una variable para un objeto, por ejemplo:

```
Persona persona
```

Lo que hace Java es crear una variable, llamada persona, y reservar espacio en memoria para almacenar una referencia a un objeto de la clase `Persona`. Java proporciona un mecanismo de seguridad que impide que se asigne una referencia a un objeto de una clase a una variable declarada como de otra clase, por lo que este tipo de asignaciones está prohibida y hará que el programa no compile. Esta regla se puede relajar un poco (o bastante) usando herencia, como veremos en otros bloques de contenido, pero por ahora vamos a suponer que es absoluta.

En resumen, la instrucción

```
Persona persona = new Persona();
```

lo que hace es crear un nuevo objeto de la clase `Persona` y almacena su referencia en la variable `persona`.

Cuando usemos la variable, Java usará la referencia almacenada en la misma de forma automática para acceder al objeto correspondiente.

3.2.1.- El valor especial null

Para las variables que almacenan referencias existe un valor especial de referencia, el valor `null` (que se escribe asi). Cuando una variable contiene el valor `null` significa que esa variable no está conectada a ningún objeto. El valor null se puede asignar a cualquier variable que admita referencias, sin importar su tipo.

Por ejemplo:

```
Persona persona = null;
```

asigna a la variable `persona` la referencia `null`.

Cuando una variable contiene el valor de referencia `null` es ilegal el intentar acceder a un objeto a través de ella. Esto provoca un error que hace que el programa se detenga inmediatamente. Esto tiene su lógica ya que si la variable no referencia a nada, no se puede hacer nada con ello.

3.2.2.- Acceso directo a los atributos de un objeto

Para acceder directamente a los atributos de un objeto a partir de una referencia al mismo se usa el operador punto (.).

La sintaxis es:

```
variableObjeto.atributo
```

donde `variableObjeto` es una variable que contenga una referencia a un objeto de un tipo determinado y `atributo` es el nombre del atributo que queremos acceder.

Por ejemplo

```
Persona persona = new Persona();

persona.nombre;
```

crea un objeto de la clase Persona y accede a su atributo nombre. Es ilegal y proporciona un error de compilación el querer acceder a un atributo que el objeto no posee. Por ejemplo, en nuestro caso la expresión:

```
persona.dni;
```

fallaría puesto que la clase `Persona` no ofrece un atributo llamado `dni`.

Una expresión con . se porta exactamente como una variable normal, esto es, podemos hacer con ella lo mismo que podemos hacer con una variable normal.

Si para modificar una variable entera llamada `numero` y guardar en ella el valor 20 hacemos:

```
int numero;

numero = 20;
```

Con los atributos se hace lo mismo:

```
Persona persona = new Persona();

persona.edad = 25;
```

Esto crea un nuevo objeto de la clase Persona y guarda en el atributo edad de ese objeto el valor 25.

Lo mismo aplica si queremos acceder al contenido del atributo. Si usamos una expresión . en un cálculo se sustituirá por el valor.

Por ejemplo, si después de lo anterior hacemos:

```
System.out.println(persona.edad * 2);
```

mostraría el valor 50 por pantalla, ya que el atributo edad contenía 25.

3.2.3.- Copia de referencias

Dado que las variables almacenan referencias a objetos en lugar de los objetos en si, se pueden producir algunas situaciones que en principio pueden parecer contraintuitivas. Por ejemplo, supongamos el siguiente código Java:

```
Persona persona = new Persona();
```

```
persona.edad = 25;
```

```
Persona persona2 = persona;
```

Si accedemos al atributo edad de `persona` o de `persona2` el contenido es el mismo, como cabría esperar ya que se ha hecho una "copia", pero, ¿qué ocurre si hicieramos lo siguiente:

```
persona.edad = 35;
```

¿Cuanto vale el atributo `edad` de `cadena2` ahora? ¿25 ó 35?

La respuesta es 35. La razón es bien sencilla

En la instrucción:

```
Persona persona2 = persona;
```

lo que estamos haciendo es copiar *la referencia* contenida en la variable `persona` a la variable `persona2`. El efecto obtenido es que ambas variables acceden o apuntan hacia el mismo objeto. Por lo tanto los cambios que se realicen sobre el objeto usando una de las variables afecta al objeto que se accede usando la otra, ya que de hecho son el mismo objeto.

Otra cuestión sobre las referencias es la que implica los operadores de comparación (`==`, `!=`, `>`, etc.). Estos operadores comparan el contenido de las variables, esto es, las referencias. Por lo tanto, los operadores `>`, `<`, etc. no tienen mucho sentido ya que el que el "DNI" de un objeto sea mayor o menor que el "DNI" de otro no significa nada. Por otra parte, los operadores `==` y `!=` si son significativos.

Por ejemplo, la comparación:

```
Persona persona1;
```

```
Persona persona2;
```

```
persona1 == persona2;
```

La ultima expresión valdrá `true` si las dos variables apuntan al mismo objeto y `false` si cada variable apunta a un objeto distinto (en el ejemplo devolverá false puesto que se trata de dos objetos distintos). De la misma forma la expresión:

```
persona1 != persona2
```

valdrá `true` si las dos variables apuntan a objetos distintos o `false` si apuntan al mismo (en el ejemplo devolvería true puesto que se trata de distintos objetos).

Se puede utilizar `null` en estas comparaciones, por ejemplo

```
persona1 == null
```

valdrá `true` si la `variable1` no apunta a ningún objeto o `false` si apunta a alguno.

3.3.- Uso de servicios de un objeto

Una vez creado un objeto, el siguiente paso lógico es utilizarlo. Para ello hay que acceder a los servicios que ofrece. Ya hemos visto como se accede a los atributos de un objeto. En esta sección describiremos como se envía un mensaje a un objeto.

Para ilustrar la exposición, ampliaremos la clase Persona definida más arriba para añadirle un mensaje al que responde. El mensaje que añadiremos se llama `envejece` y su efecto neto es que la edad de la persona que lo recibe avanza un año. Por ejemplo, si una persona tiene una edad de 25 y recibe el mensaje `envejece`, su edad pasará a ser 26. Si se envía otra vez, 27 y así sucesivamente.

3.3.1.- Uso de un mensaje

En Java, el paso de mensajes se realiza mediante un método similar al de acceso a atributos.

Para enviar el mensaje `mensaje` al objeto `objeto` se utiliza la sintaxis:

```
objeto.mensaje()
```

El efecto es que se envía el mensaje al objeto, se ejecuta la implementación del mismo y la ejecución continuará por el mismo punto cuando se termine de procesar la implementación. Nótese los paréntesis al final del nombre del mensaje. Estos son obligatorios y son una forma muy efectiva de distinguir qué es lo que se está haciendo en la instrucción ya que tanto el acceso a atributos como el paso de mensajes usan el operador punto (.). Si lo que va después del punto **no** lleva paréntesis estamos accediendo a un atributo del objeto. Si los lleva se trata en cambio de un envío de mensaje.

En Java la implementación de un mensaje se denomina **método**. Un método en una clase es el código que implementa el procesamiento de un mensaje

Por ejemplo, el siguiente código:

```
Persona persona = new Persona();

persona.edad = 25;

persona.envejece();

System.out.println(persona.edad);
```

debe imprimir 26 por pantalla ya que, a pesar de que se cambia la edad a 15 en la segunda línea, el paso del mensaje `envejece` en la tercera hace que la edad pase a valer finalmente 26.

3.3.2.- Paso de parámetros

Cuando se envía un mensaje a un objeto, en muchas ocasiones sería conveniente el poder proporcionar información adicional con el mensaje. Por ejemplo, nuestro mensaje envejece está un poco limitado porque sólo puede añadir los años de uno en uno con lo que si, por ejemplo, queremos envejecer a una persona 3 años deberíamos hacer:

```
persona.envejece();

persona.envejece();

persona.envejece();
```

Con lo pesado y repetitivo que es.

Lo óptimo, en este caso, sería el poder indicar de alguna forma junto con el mensaje el número de años que queremos que envejezca la persona. Una solución podría ser hacer otro método, llamado `envejece2`, que envejezca a la persona por 2 años en lugar de 1. Si lo piensas un poco, esta

"solución" es una chapuza porque si queremos envejecer por otra cantidad debemos crear (y usar) otro mensaje nuevo, con lo que terminamos teniendo una gran cantidad de mensajes (`envejece3`, `envejece4`, etc.) que hacen casi lo mismo pero varían en un pequeño detalle (la cantidad de años que cambian, en este caso).

Otra solución sería poder adjuntar al mensaje información adicional que sirva para personalizar o adaptar el significado exacto de lo que se desea hacer. En el caso de nuestro `envejece`, una solución definitive sería el poder añadir al mensaje la cantidad de años que se quiere envejecer.

A esta información adicional que se adjunta a un mensaje se le denomina de forma genérica **parámetro**. Un parámetro es un dato que se proporciona junto a un mensaje y que sirve para especificar información adicional necesaria para realizar el mismo.

Mas concretamente, un mensaje lleva adjunta una **lista de parámetros**, esto es, uno, dos, tres, etc. parámetros, incluyendo una lista vacía cuando un mensaje no necesita ningún parámetro.

Es necesario añadir en el interfaz del objeto los parámetros que aceptan los distintos mensajes.

Para cada parámetro es necesario indicar:

- **Nombre**. Todo parámetro debe tener un nombre. Este nombre debe ser único dentro de la implementación del mensaje (dentro del método). Los nombres de parámetros siguen las mismas reglas de nombrado que los nombres de variables.
- **Tipo**. Todo parámetro debe tener un tipo. Por lo tanto sólo aceptará ciertos valores y no otros.
- **Significado**. Es necesario informar al usuario (mediante comentarios, por ejemplo), de qué forma afecta el valor del parámetro al funcionamiento del mensaje, incluyendo los posibles valores válidos y qué se hace en caso de que los valores no lo sean.

Un mensaje puede que no necesite parámetros. En este caso seguirá teniendo una lista de parámetros pero ésta estará vacía. Dicho de otra forma, todos los mensajes tienen una lista de parámetros, aunque esta pueda tener cero elementos.

A la hora de pasar un mensaje, los parámetros se indican entre los paréntesis, separados por comas y en el mismo orden en que se especifican en el interface. No existe el concepto de parámetro opcional (en realidad si existe pero nosotros no vamos a tratarlo, con lo cual haremos como que no existe), por lo que hay que proporcionar un valor para cada parámetro del mensaje.

Los valores se pueden proporcionar usando cualquier expresión que proporcione un valor del tipo apropiado, incluyendo desde valores literales, constantes, variables o expresiones aritméticas, lógicas o de comparación. Java calculará el resultado de la expresión y éste valor será el que se envíe en el parámetro.

Por ejemplo, supongamos que ahora tenemos una nueva versión del mensaje envejece, la cual recibe ahora un parámetro. En su interfaz se indica que el primer parámetro (y único) se llama cantidad, es de tipo entero y significa la cantidad de años que se va a sumar a la edad actual. Además se especifica que el valor debe ser mayor que cero pues no tiene sentido "envejecer" hacia menos años que los que se tenía (esto sería más bien rejuvenecer).

Por lo tanto podríamos hacer:

```
Persona persona = new Persona();

persona.edad = 25;

persona.envejece(10);

System.out.println(persona.edad);
```

Esto mostraría el valor 35 (25 años que se asignan en la línea 2 y 10 años que se envejecen en la línea 3.
Asimismo también podríamos hacer:

```
Persona persona = new Persona();

persona.edad = 25;

int anyos = 10;

persona.envejece(anyos);

System.out.println(persona.edad);
```

Que también mostraría 35. En este caso como parámetro se ha usado el valor de la variable `anyos`.
Otra opción podría ser:

```
Persona persona = new Persona();

persona.edad = 25;

int anyos = 10;

persona.envejece(anyos + 6);

System.out.println(persona.edad);
```

Que mostraría ahora el valor 41 ya que a los 25 años iniciales les hemos añadido `anyos + 6` = 16 años (25 + 16 = 41).
Por completitud vamos a añadir dos mensajes más a la clase `Persona`:

- `recienNacido`. Este mensaje reinicia la edad de la persona como si acabara de nacer (a 0 años). No es muy realista para un objeto "real" pero como ejemplo vale. Este mensaje no recibe parámetros.
- `acota`. Este mensaje se asegura que la edad de la persona está en un rango de edades concreto y que no está fuera de éste. Si la edad actual es menor que el límite inferior, la edad se cambiará a dicho límite inferior y si es superior al límite inferior, se cambiará a dicho límite superior. Por último, en caso de que la edad ya estuviera en el rango indicado, se mantendría sin cambios. Este mensaje necesita dos parámetros, uno para indicar el límite inferior de la edad y otro para indicar el límite superior. El primero se llamará `inferior` y el segundo se llamará `superior`. Se indicarán en este orden y son ambos enteros. Por simplicidad vamos a suponer que el cliente no hace trampas que los valores de `inferior` y `superior` que se proporcionen son coherentes (`inferior` debe ser menor o igual a `superior`).

Con este nuevo interfaz en cuenta, podemos hacer ahora:

```
Persona persona = new Persona();
persona.edad = 25;
System.out.println("Edad actual = " + persona.edad);
persona.recienNacido();
System.out.println("Edad después de recienNacido = " +
persona.edad);
```

Este ejemplo debe mostrar como primer valor de la edad el número 25 y como segundo 0 ya que `recienNacido` "reinicia" la edad a cero.
Por último, este ejemplo:

```
Persona persona = new Persona();
persona.edad = 25;
persona.acota(10,90);
persona.acota(30,90);
persona.acota(10,18);
```

La línea 3 no tiene efecto visible alguno puesto que la edad de la persona (25) está comprendida entre 10 y 90.
La línea 4 modifica la edad a 30 años ya que la actual (25) no está comprendida entre 30 y 90. De hecho es menor que la edad inferior o mínima. Por lo tanto se cambia por esta (30).
La línea 5 modifica la edad a 18 años ya que la actual (30, cambiada en la línea 4) no está comprendida entre 10 y 18. Ya que es mayor que la edad máxima o superior, se cambia por esta (18).
Un concepto muy importante a tener en cuenta es que cuando se envía el mensaje lo que se está enviando son **valores** para los parámetros. No se están enviando variables ni nada por el estilo. Por lo tanto es indiferente la forma en que se obtengan los valores a pasar siempre y cuando sean del tipo requerido para el parámetro.

3.3.3.- Devolución de resultados

Como ya hemos visto, los mensajes pueden ser acompañados por uno o más valores (los parámetros) que sirven para parametrizar el funcionamiento del mensaje en cuestión, con lo cual existe una comunicación de información desde el objeto que envía el mensaje hacia el que lo recibe (el mensaje en si y los valores de los parámetros que lo acompañan). ¿No sería útil el que también se pudiera enviar información de vuelta desde el objeto que recibe el mensaje hacia el objeto que lo envía? La respuesta, obviamente, es SI y esto se realiza mediante el mecanismo de devolución de resultados.

Todo mensaje puede devolver un (y sólo un) valor al término del procesamiento del mismo por parte del objeto. Este resultado es opcional y no es necesario que todos los mensajes devuelvan uno. Un mensaje puede elegir no devolver resultado alguno.

Por lo tanto ahora el interfaz del objeto debe indicar, para cada mensaje, además de su nombre y lista de parámetros, el posible valor de resultado que devuelve (si es que lo hace), su tipo y su significado. Dado que sólo se devuelve un resultado no es necesario que éste tenga nombre ya que no hay confusión posible.

¿Y cómo funciona esta devolución? El funcionamiento es muy sencillo. El envío del mensaje es para Java *una expresión* y por lo tanto tiene un valor. Dicho de otra manera, podemos utilizar un envío de mensaje dentro de una expresión y el valor que devuelve es el que se utilizará para calcular la expresión.

En muchas ocasiones este valor de retorno o devolución no nos interesa o no se usa y por lo tanto se descarta.

Prosigamos con nuestro ejemplo. Ahora que sabemos que nuestros mensajes pueden producir un valor de respuesta, vamos a modificar el interfaz de la clase `Persona` de la siguiente forma:

- Modificamos `envejece` para que devuelva un entero que será la nueva edad de la persona a la que se ha enviado el mensaje.

- Asimismo también vamos a modificar `acota` para que nos devuelva un valor booleano: true si la edad estaba en el rango indicado (y por lo tanto no se ha modificado) o false si la edad no estaba en el rango (y se ha debido de ajustar de forma acorde).

Una vez hecha esta modificación podemos hacer lo siguiente:

```
Persona persona = new Persona();

persona.edad = 25;

System.out.println(persona.envejece());
```

¿Qué se imprimirá por pantalla? Analicemos el código. Las dos primeras líneas ya son conocidas. Crea un objeto de clase persona y cambia su edad a 25.

La tercera línea imprime por pantalla. ¿Qué es lo que se imprime? Pues el valor de la expresión `persona.envejece()`. Para poder calcular lo que vale esta expresión Java envía el mensaje envejece al objeto referenciado por la variable `persona`. Este mensaje aumenta en uno la edad almacenada en el objeto (que pasa de 25 a 26) y devuelve este último valor (26). Por lo tanto la expresión vale 26 (el valor devuelto) y esto es lo que se imprimirá por pantalla.

3.3.4.- Acceso indirecto a los atributos

Ahora que conocemos el paso de mensajes, podemos dar un nuevo giro de tuerca al acceso a los atributos.

En secciones anteriores comentamos la forma de acceder de forma directa a los atributos de un objeto. Esto puede presentar problemas de mantenimiento del código ya que ofrece demasiada información acerca de como funciona internamente la clase y limita que podamos hacer cambios en la misma. Imaginemos que se nos proporciona una clase círculo con el siguiente interfaz:

- Clase: Circulo
  - Atributos:
    - double radio. El radio del círculo
  - Mensajes:
    - calculaArea. Sin parámetros. Devuelve un valor double que es el área del círculo.
    - calculaPerimetro. Sin parámetros. Devuelve un valor double que es el perímetro (lo que mide la circunferencia) del círculo.

A continuación, desde nuestro código hacemos:

```
Circulo circulo = new Circulo();

circulo.radio = 10;

System.out.println("El área del círculo es " +
circulo.calculaArea());
```

Esto muestra por pantalla el área del círculo (62,8318530718).

Al cabo de un tiempo decidimos que nos interesa modificar la implementación de nuestra clase `Circulo` y para ello vamos a almacenar el diámetro en lugar del radio.

Para ello tendríamos dos opciones, pero ambas implicarían cambios en el código que **usa** la clase `Circulo`, lo cual no es deseable. Lo deseable es que los cambios que impliquen modificaciones de la implementación de una clase sin modificar el interfaz no tengan consecuencias para el código del resto de clases de nuestro sistema.

La primera opción consiste en cambiar el interfaz de la clase `Circulo`, eliminando el atributo radio y añadiendo el atributo diámetro. Esta opción implica:

- Modificar la implementación de los mensajes `calculaArea` y `calculaDiametro` que deben ajustarse para usar ahora el diámetro para hacer los cálculos en lugar del radio, del que ya no se dispone. Esto no supone problema ya que sólo cambia la implementación y no afecta para nada a la interfaz (y al resto de objetos que la usan).
- Modificar todos los objetos que usan el atributo radio de `Circulo`. Este código dejará de funcionar porque radio no existe y se deberán modificar para que use diametro en su lugar. Dependiendo de lo "popular" que sea `Circulo` esto puede implicar muchos cambios en el programa, cada uno de ellos una potencial fuente de errores y gazapos.

La otra opción consistiría en mantener `radio` y *añadir* `diametro`. Esta opción es como la anterior pero peor ya que aunque parezca que el código de los objetos "cliente" no se debe modificar esto no es cierto. La razón es bien sencilla. La nueva implementación de `Circulo` emplea el valor del diámetro para hacer los cálculos en `calculaArea` y `calculaPerimetro`. El código cliente sólo modifica el radio con lo cual a la hora de realizar los cálculos en la clase `Circulo` se devolverán valores incorrectos porque el diámetro *no se modifica de forma acorde en el codigo cliente*. Por lo tanto, a la hora de hacer los cálculos no estarán alineados los valores del radio y el diámetro y los resultados devueltos no serán los esperados.

Para evitar este tipo de problemas una técnica muy empleada, aunque no es la única, es **eliminar de los interfaces de una clase los atributos**. Estos siguen estando ahi pero no son accesibles directamente. El problema con esta solución es que se pierde acceso a la información contenida en los atributos. Para mantener este acceso se utilizan los denominados mensajes o métodos de acceso (getters y setters).

Los métodos de acceso permiten acceder a los valores de los atributos de forma indirecta, usando un mensaje en lugar del acceso directo. Esto permite que los mensajes puedan realizar tareas adicionales, cálculos, etc, haciendo el acceso a los atributos más seguro.

En nuestro caso, cambiar de diámetro a radio no sería un problema porque el atributo que se pierda se podría calcular a partir del otro en el mensaje adecuado.

Se denominan estos mensajes getters y setters porque suelen tener un nombre de la forma `getXXXX()` para acceder al contenido del atributo y de la forma `setXXXX(tipo valor)` para modificarlos. Por tradición, los mensajes que devuelve un atributo o valor booleano se suelen llamar `isXXXX()` en lugar de `getXXXX()`.

3.4.- Elementos estáticos

Como hemos descrito en anteriores secciones, la programación orientada a objetos trata a *todo* como un objeto con unos atributos y unos mensajes. Asi hemos visto clases `Persona`, `Circulo`, etc. Si seguimos esta línea de razonamiento, ¿Qué pasa con las clases? ¿No son las clases objetos

tambien? La respuesta, obviamente, es si. Por lo tanto y continuando el hilo, si una clase es un objeto también podría tener atributos y mensajes. A estos atributos y mensajes de una clase se les denomina **elementos estáticos**.

Un atributo estático de una clase es un atributo que no pertenece a ninguna instancia de la clase sino a la clase en si. Los objetos que pertenecen a esa clase pueden acceder al atributo pero sólo existe uno, que es compartido por todos los objetos de la clase, no existe un atributo para cada objeto.

Un atributo estático se accede de forma similar a un atributo no estático, usando el operador punto (.). La diferencia es que se usa la clase directamente en lugar de un objeto.

Imaginemos que ampliamos nuestra clase `Circulo`, añadiendo un nuevo atributo estático llamado creados, de tipo entero, que significa el número de círculos que se han creado desde que se inició el sistema. Supongamos que de forma mágica se inicia al valor cero al arrancar la aplicación. Este atributo se incrementará en 1 cada vez que se invoque la instrucción new sobre Circulo. De esta forma podríamos hacer lo siguiente:

System.out.println("Circulos creados " + Circulo.creados);

Circulo circulo1 = new Circulo();

System.out.println("Circulos creados " + Circulo.creados);

Circulo circulo2 = new Circulo();

System.out.println("Circulos creados " + Circulo.creados);

Suponiendo que aún no se ha creado ningún círculo desde otras partes del programa, esto mostrará por pantalla los valores 0, 1 y 2. Observa como se accede al atributo usando el nombre de la clase en lugar de una referencia a una instancia. La primera línea accede al valor de creados. Como el sistema se acaba de arrancar su valor es cero. Cuando se ejecuta la línea 2 se crea una nueva instancia y adicionalmente se añade 1 al valor de creados, pasando éste de 0 a 1. La línea 3 muestra por pantalla este nuevo valor.

La siguiente línea (4) crea otro objeto diferente, lo que ocasiona que en la 5 se muestre el valor 2.

De la misma forma que existen atributos estáticos también existen mensajes estáticos. Son mensajes a los que responde la misma clase, no ninguna instancia en concreto de la misma. La forma de acceso es similar a lo que veíamos anteriormente en el caso de los atributos estáticos. Un mensaje estático se envía usando el nombre de la clase, punto y el mensaje. El resto funciona todo igual: paso de parámetros y devolución de resultados.

3.5.- Constructores

Seguro, amigo lector, que te habrás dado cuenta de que hasta el momento no hemos hablado nada sobre el estado inicial de un objeto.

Hemos hablado de que si un objeto tiene tales o cuales atributos, que su contenido forma el estado del objeto y de como acceder a los mismos pero no hemos respondido de ninguna manera a la pregunta: ¿Qué valores tienen los atributos en los nuevos objetos?

En esta sección vamos a intentar responder a esta pregunta.

En el caso más simple no se especifica de ninguna manera qué valores van a tener inicialmente los atributos. En dicho caso Java asigna a cada uno el valor por defecto correspondiente al tipo del atributo (`0` para los datos numéricos y `false` para los booleanos).

Otra forma es que la clase define internamente los valores iniciales y los indica como nota adicional a la declaración del interfaz de la clase. El problema es que **todos** los objetos se crean inicialmente con el mismo estado.

Por último existe una forma mejor (la cual es la recomendada) para crear objetos indicándoles un estado inicial: los constructores.

Los constructores son mensajes especiales que recibe una clase durante la creación de una instancia de la misma. Sólo se pueden usar durante la creación de un objeto y no se pueden utilizar de forma general como el resto de mensajes, los cuales se pueden emplear en cualquier momento de la vida de un objeto.

Los constructores se envían cuando se usa `new` para crear un nuevo objeto. Cuando vimos inicialmente el funcionamiento de new, se comentó que había que utilizar el nombre de la clase y a continuación poner dos paréntesis `()`. En su momento puede que te pareciera raro, pero ahora que hemos visto el envío de mensajes quizás comiences a sospechar a que se deben estos paréntesis.

Si has pensado que se parece al paso de parámetros de un envío de mensajes has acertado.

Una clase define en su interfaz los constructores que soporta. Todos hacen lo mismo (ayudar a iniciar el objeto) pero cada uno admite distintos parámetros para personalizar la inicialización del nuevo objeto. Al constructor que no necesita ningún parámetro se le denomina **constructor por defecto**. Algunas clases pueden definir que éste constructor por defecto no existe y hay que usar alguno de los que se proporcionen con parámetros. Es incluso posible que se defina una clase que no ofrece ningún constructor. El efecto de éste último caso es que **no se pueden crear objetos de esta clase usando new**. La creación de objetos de estas clases se realizaría por otros métodos que comentaremos en siguientes bloques. Por ahora vamos a suponer que todas las clases disponen de uno o más constructores.

Los constructores se distinguen unos de otros por el número y tipo de sus parámetros (los nombres de los parámetros no se tienen en cuenta). Por lo tanto sólo puede existir un constructor por defecto o uno que admita un sólo parámetro entero, por ejemplo. No es posible que haya dos constructores que admitan un sólo parámetro entero, aunque tengan distintos nombres. Esto es asi por diseño del lenguaje.

Siguiendo con nuestro ejemplo de Persona, podríamos indicar en el interfaz que el constructor por defecto asigna un valor inicial al atributo edad de cero (lo esperado en una nueva persona). Asimismo podríamos añadir un nuevo constructor que admita un parámetro entero. El valor que se especifique en este parámetro será el valor que tenga inicialmente la edad de la persona. Por lo tanto nuestros ejemplos anteriores podrían quedar como:

```
Persona bebe = new Persona();

Persona bebe2 = new Persona(0);

Persona adulto = new Persona(35);

System.out.println(bebe);
```

```
System.out.println(bebe2);
System.out.println(adulto);
```

La línea 4 imprimiría 0 (al usar el constructor por defecto del valor inicial de la edad es cero).

La línea 5 también imprimiría 0. En este caso se ha usado el constructor con parámetro pero casualmente este vale 0 también.

La línea 6 imprimiría 35, que es el valor que hemos indicado en el constructor (y que no hemos modificado después).

4.- Uso de objetos de las librerías estándar: String

Ahora que conocemos como crear y usar objetos vamos a ver como usaríamos una clase que ya existe.

Para ello vamos a emplear la clase `String`, que viene incluida en la librería estándar de Java. La librería estándar de Java son un conjunto de clases ya hechas que están disponibles para todas las aplicaciones Java, ya que se incluyen por defecto en todas las instalaciones. Otro tipo de librerías, que veremos más adelante, contienen clases que definen otros programadores y hay que especificar su uso de forma directa.

La clase `String` representa una cadena de caracteres. Los mensajes que ofrece están orientados, por tanto, a la manipulación de cadenas de caracteres y texto.

La clase está diseñada de tal forma que todos los objetos son *inmutables*, esto es, que una vez que un objeto de clase `String` contiene una cadena, ésta no se puede modificar. Una de las consecuencias de este diseño es que la mayoría de mensajes que vamos a ver no operan sobre el objeto que recibe el mensaje sino que se crea un nuevo objeto de clase String para almacenar el resultado, manteniendo el objeto original que recibió el mensaje sin cambios. Existe una clase alternativa para trabajar con cadenas de caracteres, la clase StringBuilder, que *si* permite la modificación de su contenido.

La documentación de la clase String la puedes encontrar en:
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html

4.1.- Creación de objetos String

Los objetos String se crean como el resto de objetos, usando new y los constructores. Hay una gran variedad de ellos pero los únicos que vamos a comentar porque son los más usados son:

- `String()`. Constructor por defecto. Crea una cadena vacía
- `String(String original)`. Crea una cadena con la misma secuencia de caracteres que otra ya existente (`original`).

Asimismo, String se distingue del resto de clases del sistema en que permite un tercer método de construcción. En nuestro código podemos hacer:

```
String cadena = "Esto es un literal de cadena";
```

Esto crea un nuevo objeto de la clase String con el valor especificado en el literal. Nótese que la instrucción `new` no aparece por ningún lado.

4.2.- Lectura de cadenas desde el teclado

En bloques anteriores aprendimos la forma de leer datos de distintos tipos (int, long, double, etc.) desde teclado.

Para leer un dato de tipo String podemos utilizar la forma:

```
Scanner sc = new Scanner(System.in);

String dato = sc.nextLine();
```

Esto lee una cadena desde teclado, hasta que se pulsa la tecla `Enter` y la almacena en la cadena `dato`.

4.3.- Concatenación de cadenas

La concatenación de cadenas es una operación que toma 2 cadenas y devuelve otra formada por el contenido de la primera seguido por el contenido de la segunda. Por ejemplo, si concatenamos la cadena `"Hola"` con la cadena `"Adios"` obtendremos una nueva cadena con el contenido `"HolaAdios"`.

Para concatenar cadenas en Java no se emplea un mensaje sino el operador `+`, que ya hemos conocido como operador aritmético. Cuando uno de los operandos es una cadena se convierte en el operador concatenación de cadenas. Si el otro operando no es ya una cadena lo convierte a cadena. Veremos más sobre esto cuando veamos la herencia.

Por ejemplo, el código:

```
System.out.println("Hola " + "Caracola");
```

imprimiría por pantalla el texto `Hola Caracola`

4.4.- Mensajes de información

Los siguientes mensajes proporcionan información sobre la cadena:

- `int length()`. Devuelve el número de caracteres que componen la cadena. Puede valer cero si la cadena está vacía.
- `boolean isEmpty()`. Devuelve `true` si la cadena está vacía o `false` si no lo está. Es lo mismo que hacer `cadena.length() == 0` pero más sencillo y claro.
- `boolean isBlank()`. devuelve `true` si la cadena está vacía o sólo contiene caracteres blancos y `false` en caso contrario. Se consideran caracteres blancos los espacios, tabuladores o saltos de línea.

4.5.- Mensajes de comparación

Como hemos visto en secciones anteriores, para comparar dos objetos no basta con comparar las referencias, ya que esto sólo nos indica si las dos apuntan al mismo objeto, no si los valores

contenidos en los dos objetos son "iguales" (el concepto de igualdad entre dos objetos lo veremos en bloques siguientes pero no es tan sencillo como comparar los valores de los atributos).

Por lo tanto, String ofrece varios métodos para comparar cadenas. Los métodos son, entre otros:

- `boolean equals(String otraCadena)`. Devuelve `true` si el contenido de las dos cadenas es igual o `false` en caso contrario. Es sensible a mayúsculas y minúsculas por lo que la comparación de `"Hola"` y `"hola"` devolverá false.
- `boolean equalsIgnoreCase(String otraCadena)`. Devuelve `true` si el contenido de las dos cadenas es igual o `false` en caso contrario. No es sensible a mayúsculas / minúsculas, por lo que la comparación de `"Hola"` y `"hola"` devolverá true.
- `int compareTo(String otraCadena)`. Devuelve un entero negativo (el valor no importa, sólo el signo) si esta cadena es anterior alfabéticamente a `otraCadena`, el valor 0 si las dos son iguales o un valor positivo (tampoco importa el valor) si esta cadena es alfabéticamente posterior a `otraCadena`. Versión sensible a mayúsculas / minúsculas.
- `int compareToIgnoreCase(String otraCadena)`. Igual que el anterior pero no es sensible a mayúsculas / minúsculas.
- `boolean contains(String otraCadena)`. Devuelve `true` si `otraCadena` está contenida dentro de la cadena que recibe el mensaje, incluyendo el caso en que ambas sean iguales. Por ejemplo (simulado), `"Hola".contains("la")` devuelve `true` ya que `"la"` forma parte de `"Hola"`.
- `boolean startsWith(String otraCadena)`. Devuelve `true` si esta cadena comienza por `otraCadena`. `false` en cualquier otro caso.
- `boolean endsWith(String otraCadena)`. Devuelve `true` si esta cadena termina por `otraCadena`. `false` en cualquier otro caso.
- `int indexOf(String otraCadena)`. Devuelve la posición, dentro de esta cadena, en la que se encuentra `otraCadena`. Si no se encuentra devuelve el valor `-1`.
- `int indexOf(String otraCadena, int posicion)`. Otra versión del mensaje que admite también un entero. Este indica la posición a partir de la que se va a buscar dentro de esta cadena.
- `int lastIndexOf(String otraCadena)`. Igual que los anteriores pero se comienza a buscar desde el final de la cadena hacia el inicio.
- `int lastIndexOf(String otraCadena, int posicion)`. Igual pero en lugar desde el final se comienza desde la posición dada.

4.6.- Mensajes de extracción

Estos mensajes se emplean para extraer partes de una cadena. Entre otros podemos encontrar:

- `char charAt(int indice)`. Devuelve el carácter situado en la posición `indice` de la cadena. indice puede valer desde 0 (para el primer carácter) hasta la longitud de la cadena - 1 para el último. Si se da un indice incorrecto se produce un error.

- `String trim()`. Devuelve esta misma cadena eliminando los blancos al inicio y al final. Para ver qué caracteres son blancos consulta la descripción de `isBlank()`.
- `String substring(int posicion)`. Devuelve la sub-cadena de esta que comienza en la posición dada y sigue hasta el final de la cadena. La posicion debe seguir las mismas reglas que el indice de `charAt`.
- `String substring(int posicionInicial, int posicionFinal)`. Igual que el anterior pero en lugar de devolver hasta el final de la cadena devuelve sólo hasta la posición final dada, que debe ser mayor o igual que la inicial.
- `String toUpperCase()`. Convierte la cadena que recibe el mensaje de forma que todas las letras son mayúsculas. **OJO: Este método modifica la cadena que lo recibe. Devuelve referencia a la misma cadena.**
- `String toLowerCase()`. Exactamente igual que el anterior pero pasa todas las letras a minúsculas.

5.- Librerías de objetos

Si las clases se crean de forma inteligente, es posible su reutilización. Por reutilización de una clase entendemos el usar dicha clase en otra aplicación distinta a aquella para la que se escribió inicialmente. Ya hablaremos más adelante a qué nos referimos por creación de forma inteligente.

Algunos programadores crean clases directamente con la intención de que sean reutilizadas, esto es, no crean clases para una aplicación en concreto sino que las crean con el objetivo de distribuirlas sueltas, sin integrar en ninguna aplicación, con el objetivo de que las usen otros programadores para otras aplicaciones, compensación económica mediante o no.

A estos paquetes de clases sueltos orientados a ser usados para hacer aplicaciones se les denominan **librerías de clases** o simplemente **librerías**.

Por supuesto nada nos impide a nosotros, como desarrolladores que vamos a ser, el desarrollar nuestras propias librerías pero por ahora vamos a limitarnos a usarlas.

5.1.- Inclusión de librerías

Para incluir una librería en nuestra aplicación hay que indicar al sistema Java donde puede encontrarla a fin de que puede incorporarla a nuestra aplicación.

En Java usualmente las librerías se distribuyen mediante archivos JAR. Un archivo JAR es un archivo especial de Java que contiene en su interior una o más clases ya compiladas. Cuando se le dice a Java que deseamos usar un archivo JAR, hay que indicarle al sistema la localización y nombre del archivo y Java lo cargará durante el inicio del programa y hará las clases contenidas en el mismo disponibles al resto de clases de la aplicación. La forma exacta de indicar esto depende de la forma en que se ejecute la aplicación. Nosotros lo veremos en nuestro IDE favorito.

5.2.- Uso de librerías

Una vez incluida la librería o librerías que queremos usar, nos toca el emplearlas desde nuestra aplicación.

Para ello es necesario el introducir antes el concepto de paquete de clases o simplemente paquete en Java.

5.2.1.- Paquetes de clases

En Java, un paquete de clases es un contenedor que agrupa a un conjunto de clases. Un paquete siempre debe tener un nombre. Asimismo un paquete también puede contener, a su vez, a otros paquetes. En este sentido podríamos hacer un símil entre los paquetes de Java y las carpetas de ficheros en un ordenador. Un paquete (carpeta) puede contener clases (ficheros) u otros paquetes (otras carpetas). De la misma forma no pueden existir dos clases con el mismo nombre dentro del mismo paquete, aunque si está permitido que dos clases pertenecientes a paquetes distintos tengan el mismo nombre. Asimismo, dentro de un paquete determinado no pueden existir otros paquetes con el mismo nombre aunque si pueden existir paquetes con el mismo nombre pero que estén contenidos en paquetes distintos.

El objetivo de los paquetes de clases es el de agrupar clases que lógicamente hacen una tarea común o colaboran estrechamente para hacer tareas comunes. Asimismo proporcionan libertad al desarrollador para poder llamar a sus clases con los nombres que crean más convenientes, siempre y cuando procure que sus clases estén colocadas en paquetes cuyos nombres sólo puede usar él. Es por esto una práctica normal el utilizar el nombre de dominio de la empresa como nombre del paquete principal a fin de que sea seguro que nadie pueda hacer clases cuyos nombres choquen con los de la empresa (exceptuando, claro está, otros empleados dentro de la misma empresa).

Se parte de un paquete por defecto o sin nombre y dentro de éste se van creando paquetes y subpaquetes dentro de éstos. formando una estructura en árbol, en la cual el paquete por defecto sería la raiz, los distintos paquetes y subpaquetes las ramas y las clases las hojas.

Para localizar a una clase determinada hay que utilizar su **nombre cualificado** que se escribe o construye partiendo desde el paquete por defectoy se va descendiendo por las distintas ramas del árbol (los nombres de los paquetes) hasta llegar al nombre de la clase. Las distintas etapas se van separando por un punto (.).

Asi, por ejemplo, el nombre cualificado de clase:

`es.iespablopicasso.programacion.unidad01.ejercicio01.Persona`

se referiría a la clase `Persona`, que está colocada dentro del paquete `ejercicio01`, que a su vez está contenido dentro del paquete `unidad01`, que a su vez está dentro del paquete `programacion`, que a su vez está dentro del paquete `iespablopicasso` que a su vez está dentro del paquete `es`, que depende del paquete por defecto.

Tanta es la similaridad entre los paquetes y las carpetas que en la práctica son lo mismo ya que cada paquete se representa como una carpeta dentro de la carpeta que contiene los fuentes del programa (que sería la que representa al paquete por defecto), cada subpaquete por una subcarpeta y cada clase por un archivo con la extensión .java que corresponderá con el código de la clase en cuestión. Nuestro IDE se ocupará normalmente de la gestión de estas carpetas y archivos dejándonos el

trabajar simplemente con paquetes y clases pero es conveniente saberlo si algún día se necesita hacer algo de forma manual, sin asistencia del IDE.

Asimismo, cada archivo .java, que contiene el código de una clase, debe llamarse exactamente igual que la clase.

O sea, la clase `Persona` debe estar contenida en un archivo llamado `Persona.java`. Si no se produce esta coincidencia el compilador rehusará compilar la clase y proporcionará un error.

Una vez explicado el funcionamiento de los paquetes, vamos a ver cómo usamos las clases de una librería desde nuestros programas.

5.2.2.- Uso de clases. Nombre cualificado

En cualquier momento podemos utilizar una clase en nuestro programa usando su nombre cualificado.

Por ejemplo, para usar la clase Persona definida en el apartado anterior desde otra clase **perteneciente a otro paquete** (para las del mismo paquete esto no es necesario) se podría usar el nombre cualificado de la clase. Por tanto el código quedaría:

```
es.iespablopicasso.programacion.unidad01.ejercicio01.Persona
persona = new
es.iespablopicasso.programacion.unidad01.ejercicio01.Persona();

persona.edad = 25;
```

(aunque parezcan tres líneas, en realidad son dos. Todo lo que hay antes de persona.edad... es una sola línea).

Como puedes ver, este método es muy trabajoso para el uso diario ya que, además de requerir una gran cantidad de texto, es fácil equivocarse con nombres tan largos.

Afortunadamente, Java proporciona una forma más sencilla de hacer uso de una clase desde clases contenidas en otros paquetes.

Aún así, este método, aunque pesado, es imprescindible en el caso, poco corriente, de que dentro de una misma clase se vayan a utilizar objetos de dos (o más) clases con exactamente el mismo nombre de clase aunque pertenezcan paquetes distintos. En ese caso es necesario usar esta forma para que Java pueda determinar, sin ambigüedades, a qué clase nos estamos refiriendo exactamente en cada instrucción.

5.2.3.- Uso de clases. Importación

La forma más sencilla de usar desde un paquete clases contenidas en otros es mediante el mecanismo de **importación**.

Mediante este mecanismo se hace saber a Java que vamos a usar una clase de otro paquete dentro del nuestro y a partir de ese momento sólo es necesario indicar el nombre de la clase (sin los paquetes) para usarla.

Esta importación hay que realizarla al inicio del fichero .java que contiene la clase que hace el uso de otra. Si se usa más de una clase habría que hacer una importación por cada una.

La importación se realiza mediante la instrucción (más bien declaración) `import`, con la forma:

```
import nombre_cualificado_clase;
```

Esto hace que dentro de la clase se pueda utilizar solamente el nombre de la clase, sin necesidad de usar el nombre cualificado completo.

Por ejemplo, en una clase nuestra podríamos hacer:

```
import es.iespablopicasso.programacion.unidad01.Persona;

.... mas codigo....

Persona persona = new Persona();

persona.edad = 25;
```

y funcionaría perfectamente. Como ves, usamos el nombre cualificado una sola vez (en el `import`) y a partir de ahi se puede usar únicamente el nombre de la clase, sin incluir los paquetes.

Existe otra forma de `import`, que se comentará aqui, aunque su uso está totalmente desaconsejado en general y totalmente prohibido en este curso.

La otra forma de import es:

```
import ruta.a.un.paquete.*;
```

Como puedes ver, en lugar de un nombre cualificado de clase se usa un asterisco. Esta forma de `import` indica que se van a usar *todas* las clases contenidas en el paquete especificado, por lo que todas se podrán usar sin necesidad de emplear el nombre cualificado. Es equivalente a usar una instrucción `import` por cada clase contenida en el paquete.

Esta tarea de nombrado e importaciones, evidentemente pesada, se ve aliviada cuando usamos un IDE lo suficiéntemente inteligente, por ejemplo, nuestro amigo Eclipse.

Cuando creamos una clase, Eclipse nos solicita tanto el nombre del paquete en el que va a estar contenida como el nombre de la clase. Con estos mimbres, Eclipse crea de forma automáticamente, y si no existe ya previamente, las carpetas necesarias para los paquetes correspondientes y crea el fichero .java correspondiente a la clase.

Cuando necesitamos usar una clase, Eclipse nos permite, mediante autocompletado, buscar la clase que necesitamos a partir del inicio de su nombre. Si elegimos la clase de la lista, Eclipse se encarga de incluir la instrucción `import` en su sitio de forma que nos ahorremos su escritura.

Asimismo, si necesitamos tanto cambiar el nombre de una clase como moverla de un paquete a otro, Eclipse realiza esta tarea de forma automática encargándose de las, tediosas y tendentes a errores, tareas relacionadas con los ficheros e incluso cambiando el nombre de la clase o los paquetes en aquellas otras partes del programa en que estén siendo utilizadas.

6.- Destrucción de objetos y liberación de memoria

Cuando un objeto deja de ser necesario, ya sea porque su representación en el mundo real ha dejado de existir, bien porque lo necesitábamos temporalmente para realizar una tarea que ya ha terminado, o por cualquier otra razón es necesario el destruirlo.

Destruir un objeto significa reclamar el espacio que éste ocupaba en memoria de forma que queda más memoria disponible para nuevos objetos. Si no se destruyeran objetos, el consumo de memoria de todas las aplicaciones sería siempre creciente y terminarían por colapsar al equipo donde se ejecutan. La destrucción de objetos hace que una aplicación sólo use la memoria que necesita en un momento determinado, pero no más.

En muchos lenguajes, por ejemplo C++, la destrucción de objetos se realiza a petición del programador. Existe una instrucción específica, antagónica de `new`, que realiza la destrucción del objeto que se le indique. Siguiendo el ejemplo de C++, esta instrucción es `delete` y se le proporciona una referencia al objeto a destruir.

Esta forma de trabajar empodera al programador, ya que éste tiene control absoluto acerca de cuando se crean o destruyen los objetos que forman la aplicación pero, como es habitual, también tiene sus desventajas ya que proporciona oportunidades para errores de programación tales como:

- Consumo excesivo de memoria, si un programador se "olvida" de destruir objetos que ya no son necesarios y no se van a volver a usar.
- Uso incorrecto de un objeto ya destruido, si un programador intenta usar una referencia a un objeto que ya no existe puesto que ha sido destruido, normalmente con consecuencias catastróficas.

Para evitar este tipo de errores, otros lenguajes, como es el caso de Java, adoptan otra estrategia para solucionar el problema de la destrucción de objetos: La destrucción automática.

En el caso de Java (y de otros lenguajes que hacen algo similar), existe un "programa especial", conocido como **recolector de basura**, que se ejecuta en segundo plano, o sea, al mismo tiempo que el programa de verdad, y va monitorizando continuamente los objetos que se han creado en éste último, intentando detectar objetos que ya no se pueden usar porque no hay variables que contengan referencias a ellos. Cuando se detectan estos objetos, se eliminan automáticamente. En caso de duda, no se eliminan. Este mecanismo presenta la ventaja de que es muy sencillo para el programador, que sólo debe ocuparse de crear objetos cuando lo necesite que ya se destruirán automáticamente cuando sea conveniente, despreocupándose completamente del asunto. Por supuesto, este enfoque no es la panacea ya que el recolector de basura consume recursos en su incesante tarea de búsqueda y destrucción de objetos abandonados.

7.- Resumen

En este bloque hemos aprendido a usar objetos realizados por otros programadores a partir de una descripción de su interfaz. En bloques subsecuentes aprenderemos a crear nuestros propios objetos.

8.- Referencias

- Programación Orientada a Objetos (sin los dos últimos pilares que veremos más adelante): https://www.campusmvp.es/recursos/post/los-conceptos-fundamentales-sobre-programacion-orientada-objetos-explicados-de-manera-simple.aspx
- Paquetes en Java: https://www.arkaitzgarro.com/java/capitulo-19.html