

Fundamentos de Programación

Tema 4: Programación básica de clases

Contenidos

1.- Introducción	2
2.- Estructura interna de una clase.....	4
3.- Programación de una clase	6
4.- Programación de métodos constructores.....	12
5.- Test de las clases	20
6.- Test Driven Development (TDD)	28
5.- Programación de métodos de instancia.....	32
6.- Programación de métodos que lanzan excepciones.....	44
7.- Programación de métodos recursivos.....	51
8.- Programación de métodos estáticos.....	56
9.- Propiedades estáticas.....	59
10.- Interfaces.....	67
11.- Records.....	74
12.- Enumeraciones.....	79

Tema

4

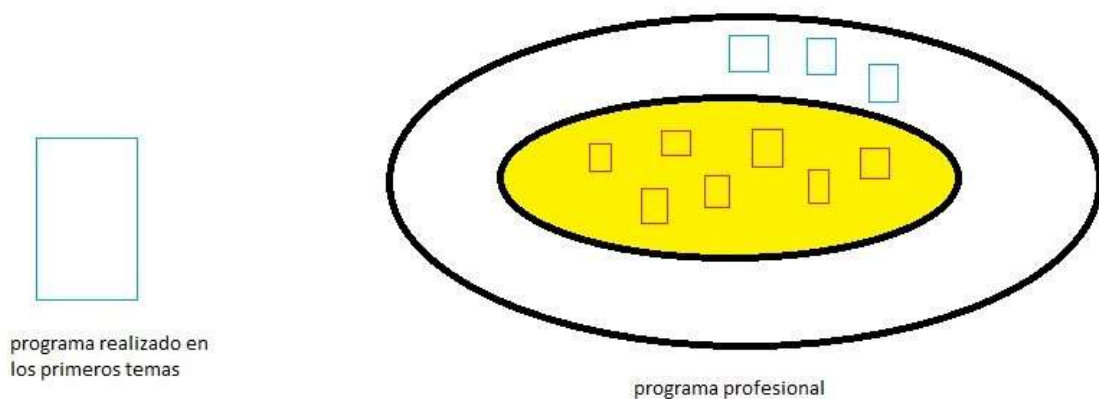
PROGRAMACIÓN BÁSICA DE CLASES

Las clases son el elemento fundamental de la programación orientada a objetos y del desarrollo profesional de aplicaciones. Gracias a las clases, podemos crear nuestros propios tipos de datos añadiendo a cada uno sus propios métodos. Las clases tienen la ventaja de que se pueden reutilizar en cualquier proyecto y además permiten dividir el trabajo entre varias personas.

1.- Introducción

Como ya hemos estudiado en los primeros temas, la programación orientada a objetos fomenta la utilización de objetos informáticos para realizar programas de ordenador. Los objetos son capaces de realizar acciones, y obedecen en todo momento al programador de aplicaciones.

Las clases son fundamentales en la programación. Si pudiéramos hacer un dibujo de cómo es un programa profesional por dentro, y compararlo con los programas que hemos hecho en los primeros temas, podríamos hacerlo así:



Los programas (más bien ejercicios) que hemos hecho en los primeros temas están formados por un solo archivo de código fuente más o menos grande, en el que se concentra todo lo que hace el programa.

En cambio, un programa profesional está formado por clases que representan los tipos de datos que aparecen en la situación del mundo real que tiene que ver con el programa (por ejemplo, en un programa para gestionar un banco, tendríamos la clase CuentaCorriente, la clase Cliente, etc). Esos tipos de datos serían las clases que están en la parte de color amarillo¹.

¹ Estas clases se llaman el **modelo**, o el **core** de la aplicación

El programa profesional también tiene otras clases (en la zona blanca), que se encargan de la interacción con el usuario y la presentación de la interfaz²

Dividir el programa en clases es bueno porque:

- Facilita el mantenimiento. Un programa gigante no puede modificarse cómodamente, pero si el programa está dividido en clases, es posible ampliarlo y corregirle los fallos con más facilidad.
- Facilita el trabajo en equipo. Las personas asignadas a un proyecto pueden trabajar a la vez en clases diferentes, que luego se juntan para formar el programa en conjunto.
- Permite reutilizar código. Las clases se pueden empaquetar en librerías, lo que facilita que se transporten fácilmente de un proyecto a otro y no haya que volver a programar cosas que ya se han hecho y probado.

Los programadores que hacen las clases se denominan **programadores de clases**. Como sabemos, esto es un rol que una persona puede adoptar en un momento determinado. Este perfil de programador se encarga de diseñar y programar las clases que luego el programador de aplicaciones se encontrará empaquetadas en librerías.

El programador de clases se encarga de programar la clase entera, lo que incluye programar constructores, métodos de instancia, métodos estáticos, constantes, interfaces, etc. Normalmente también escribe comentarios que luego se convertirán en el javadoc de la clase y durante todo el proceso hace **test** para comprobar que lo que va programando funciona correctamente.

Para programar una clase lo primero que hay que hacer es cambiar la mentalidad y comprender que vamos a hacer algo diferente a los programas a los que estamos acostumbrados.

Una dificultad importante que encuentran muchas personas que empiezan a programar clases es que no son conscientes del cambio de perfil que supone pasar de ser “programador de aplicaciones” a “programador de clases”. Como veremos durante el tema, cuando se programa una clase hay que pensar de manera diferente, porque hacer una clase no es hacer “un programa que se ejecuta de arriba abajo”. Hacer una clase es crear un tipo de dato que será utilizado dentro de un programa.

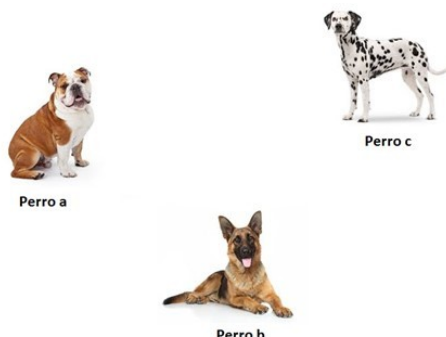
Por último, como veremos durante el tema, para programar una clase deberemos disponer del **diagrama de clases** que vamos a programar. Como ya sabemos, en ese diagrama aparece la forma de los métodos que hay en la clase, y será muy útil para programar la clase.

En la vida real hay un puesto de trabajo denominado analista, que entre otras funciones, se encarga de diseñar los diagramas de las clases del proyecto. En una empresa tradicional, los diagramas de clases los elaboran los analistas y los proporcionan a los programadores, si bien, en muchas empresas son los programadores quienes tienen que diseñar las clases.

² En la zona blanca hay dos tipos de clases: las que se encargan de la interacción con el usuario se denominan **controladores**, y las que se encargan del diseño de la interfaz de usuario, **vistas**

2.- Estructura interna de una clase

Vamos a pensar por un momento en los objetos del mundo real. Por ejemplo, en la siguiente imagen podemos ver tres objetos (o instancias) de la clase **Perro**.



Como ya sabemos, cada uno de estos objetos ha sido creado con su correspondiente método constructor, lo cual da lugar a tres objetos diferentes en la memoria RAM.

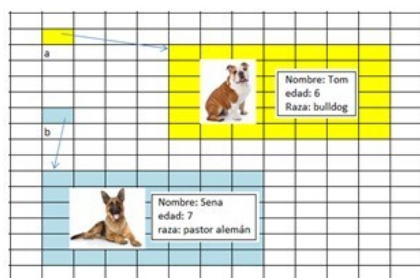
Sin embargo, cada uno de estos objetos lleva consigo **variables internas** que definen como es y cómo se encuentra el perro en ese momento. En la siguiente imagen podemos ver cuáles son estas variables para los tres objetos de la clase Perro de imagen anterior:



Estas variables se denominan **variables de instancia** (también llamadas **propiedades** o **atributos**), y el programador de aplicaciones no las puede ver, ya que son internas para cada uno de los objetos. Estas variables guardan características fijas del objeto (su nombre y su raza) y también características que pueden cambiar con el tiempo y que representan el estado del objeto (su edad, su posición y si está hambriento).

Las propiedades de cada objeto son variables independientes que lleva dentro. Por ejemplo, las propiedades del perro “Tom” son diferentes de las del perro “Jim”. Sin embargo, todos los objetos de la clase Perro tienen el mismo juego de propiedades. Es decir, todo objetoPerro tiene las propiedades nombre, edad, raza, posición y hambriento.

Las propiedades se crean en la memoria RAM cuando se llama al método constructor para crear el objeto, y se guardan con él. Podemos verlo en esta imagen:



```
Perro a = new Perro("Tom",6,"bulldog");
```

```
Perro b = new Perro("Sena",7,"pastor alemán");
```

Las propiedades de un objeto aparecen en la **zona central** del diagrama de clases, y por ese motivo nunca las hemos visto cuando teníamos el perfil de programador de aplicaciones, ya que a este perfil le aparecen ocultas.

Perro
<ul style="list-style-type: none"> - String nombre - int edad - String raza - Point posición - boolean hambriento
<ul style="list-style-type: none"> + Perro(String n, int e, String r) + Perro(String n, int e, String r, Position p, boolean h) + void saltar(int a) + void correr(int velocidad) + String ladrar() + void comer()

Los signos + y – que acompañan a las propiedades y métodos se denominan **modificadores de acceso**, y nos indican qué perfil del programador puede tener acceso a ellos. En el lenguaje Java tenemos los siguientes modificadores de acceso:

- El signo + se llama **public** y significa que cualquier perfil de programador puede verlos. Siempre hemos visto este modificador en los métodos. También se podría poner en las propiedades, pero entonces el programador de aplicaciones podría verlas, y más adelante veremos que esto es **malísimo** y está totalmente desaconsejado.
- El signo – se llama **private** y significa que solo la persona que programa la clase puede ver esa propiedad. Este modificador se pone en las propiedades (para que estén ocultas al programador de aplicaciones) y también en algunos métodos que son auxiliares, y que solo hace falta que los conozca quien programa la clase.

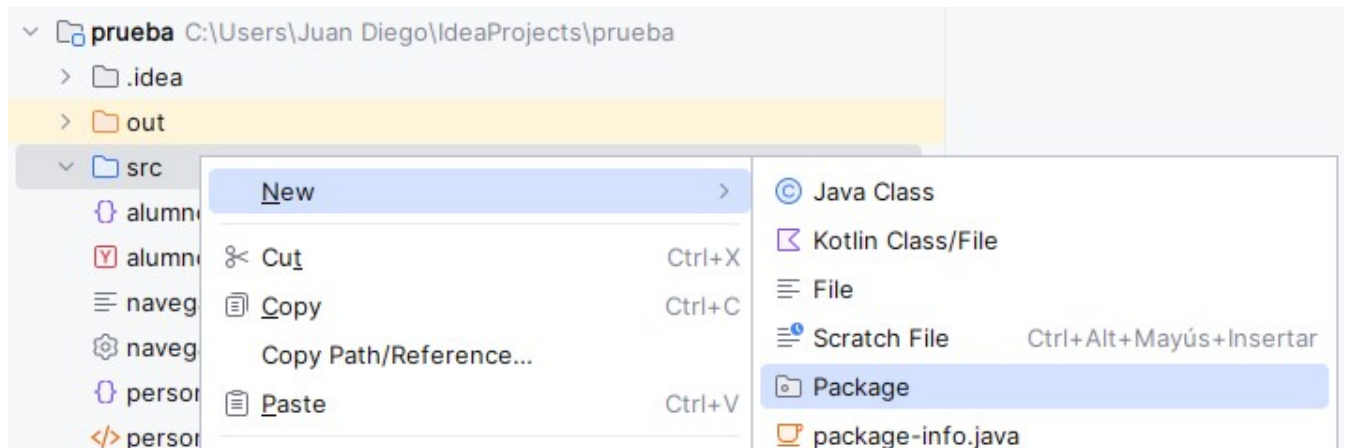
También existen otros dos modificadores de acceso, que se encuentran a medio camino entre el public y el private. Es decir, los puede ver quien programa la clase, pero no quien programa la aplicación. Sin embargo, el efecto que producen y sus diferencias con private los estudiaremos en el próximo tema. Estos nuevos modificadores son:

- El signo #, llamado **protected**
- No poner modificador a la propiedad. Esto se llama **modificador por defecto**³. También admite en el diagrama de clases (pero no en el código fuente) el signo ~

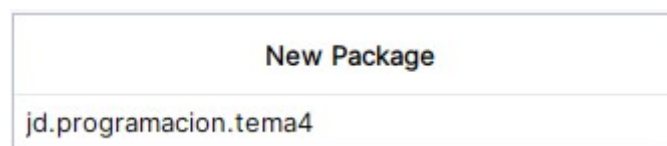
³ El modificador por defecto también se denomina **modificador package** o **modificador package-private**

3.- Programación de una clase

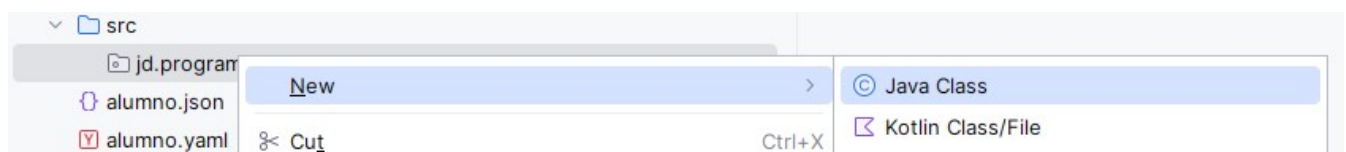
Podemos añadir clases a cualquier proyecto que hagamos. Lo primero que haremos será crear un paquete⁴ dentro del proyecto pulsando el botón derecho del ratón sobre **src** y eligiendo la opción **new → Package**



En la ventana que aparece escribiremos el nombre del paquete. Se puede poner el nombre que quieras. Aunque el nombre que se pone a un paquete no tiene importancia, el nombre que se suele poner a los paquetes tiene aspecto de “dirección web” escrita al revés⁵.



Una vez que tenemos hecho el paquete, le podemos añadir una clase a ese paquete pulsando en **new → Java class**



En la ventana que aparece escribiremos el nombre de la clase (por ejemplo, Perro) y luego pulsaremos el botón de aceptar. En ese momento, el IDE nos creará un archivo con el código fuente de la clase que vamos a programar.

Nuestra clase, al principio estará vacía, más o menos con un código fuente como se ve a continuación:

⁴ Es posible crear clases que no estén en ningún paquete (se crearían en el **default package**), pero esto está desaconsejado, porque después no se podrían importar en ningún programa (salvo que el programa también estuviera en el default package)

⁵ Este tipo de nombre se denomina **notación dns inversa**

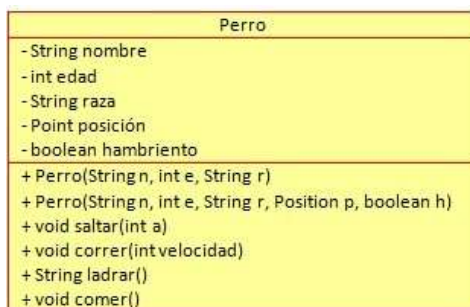
```

1. package jd.programacion.tema4;
2.
3. public class Perro {
4.
5. }

```

3.1.- Programación de las propiedades

Lo primero que hay que hacer para programar una clase es escribir las propiedades tal y como aparecen en la zona central del diagrama de clases, sustituyendo cada modificador de acceso por su nombre, así:



```

01. package com.animales;
02. import java.awt.Point;
03.
04. public class Perro {
05.     // PROPIEDADES
06.     private String nombre;
07.     private int edad;
08.     private String raza;
09.     private Point posicion;
10.     private boolean hambriento;
11. }

```

Es importante darse cuenta de estos detalles:

- Las propiedades están escritas en un color diferente⁶. Esto es muy importante porque así podremos diferenciarlas de las variables normales que usemos más adelante.
- En principio, las propiedades no se inicializan, sino que simplemente, se dejan declaradas al principio de la clase.
- Las propiedades pueden tener cualquier tipo de dato, ya sea básico (edad, hambriento) o referencia (nombre, raza, posición). En este caso, si se utiliza alguna clase externa (en este ejemplo, se usa Point), deberá importarse el paquete donde esté esa clase (en este caso, Point está en java.awt).

En la vida real cuando se trabaja en empresas, es **vital seguir al pie de la letra** los diagramas de clases que nos dan, sin cambiar el nombre de las propiedades o parámetros que aparecen en él. Esto es para que la documentación que comparten todas las personas que trabajan en el proyecto coincida con el código fuente que realmente se está programando.

3.2.- El constructor por defecto

La clase Perro que acabamos de programar solo tiene propiedades y no tiene ningún método. De hecho, no le hemos puesto ningún método constructor. En esa situación, Java le añade automáticamente a la clase un método constructor que no recibe parámetros, así:

⁶ Como es lógico, el color de las propiedades depende del IDE y tema visual que estamos utilizando

Perro
<ul style="list-style-type: none"> - String nombre - int edad - String raza - Point posición - boolean hambriento
+ Perro () → Constructor por defecto añadido automáticamente

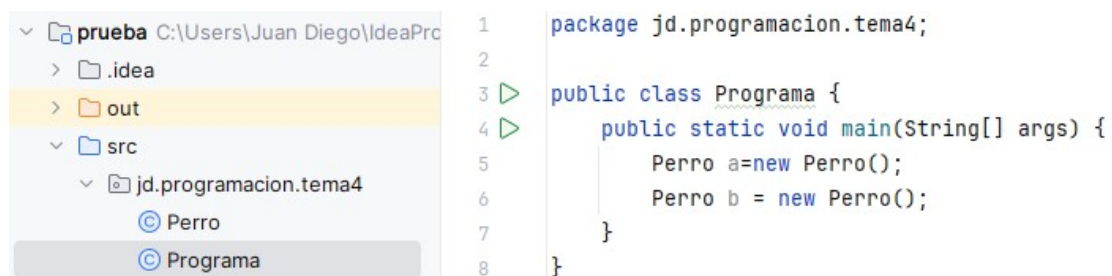
Ese método constructor añadido de forma automática se llama **constructor por defecto**, y no tiene parámetros. Java siempre añade este constructor a las clases cuyo programador no le ha puesto ningún método constructor. El constructor por defecto no hace nada, y cuando se crea el objeto, sus propiedades toman el valor por defecto de esta tabla:

Tipo de dato de la propiedad	Valor que toma la propiedad
números (int, byte, short, long, double, float)	0
boolean	false
char	carácter con código ASCII 0
objetos	null

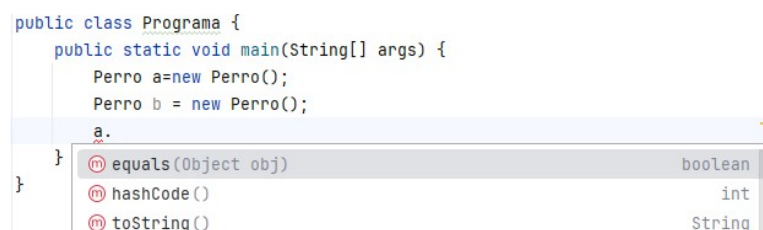
Ejemplo: Hacemos un programa que cree dos objetos de la clase Perro que hemos hecho.

Para hacer un programa que use la clase Perro, bastará con añadir a nuestro proyecto un archivo con el código fuente del programa, como ya sabemos. Si ponemos dicho archivo dentro del paquete de la clase, no será necesario importarlo (por defecto, un archivo tiene acceso a todas las clases de su mismo paquete), pero si no, tendremos que poner un **import**.

Una vez creado el archivo, dentro del main simplemente crearemos dos objetos usando el constructor por defecto que le ha puesto Java.



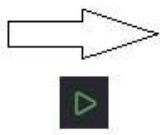
Como hemos visto en el apartado anterior, los perros que hemos creado aún no tienen métodos y por tanto, si queremos “darle órdenes o hacerles preguntas” usando el punto, veremos que solo aparecerán los métodos heredados de la clase Object⁷:



⁷ Recordemos que la clase Object es la clase padre de todas las clases de Java, incluida nuestra clase Perro. Por tanto, nuestra clase Perro recibe automáticamente todos los métodos de la clase Object, como equals, toString y hashCode

De hecho, podemos llamar, por ejemplo al método `toString` y comprobar que produce una salida, que viene dada por cómo está programado el método `toString` en la clase `Object`. Dicha salida no produce nada significativo, y más adelante veremos que podremos reprogramar dicho método `toString` para que produzca una salida más acorde a lo que nos haga falta para nuestro programa.

```
public class Programa {
    public static void main(String[] args) {
        Perro a=new Perro();
        Perro b = new Perro();
        System.out.println(a.toString());
        System.out.println(b.toString());
    }
}
```



 jd.programacion.tema4.Perro@6d03e736
 jd.programacion.tema4.Perro@568db2f2

En este ejemplo hemos visto cómo podemos usar la clase `Perro` para luego hacer dos objetos `Perro` diferentes. Por este motivo, el programador de clases ve una clase como una **plantilla o molde** que nos permite posteriormente crear los objetos de la clase.



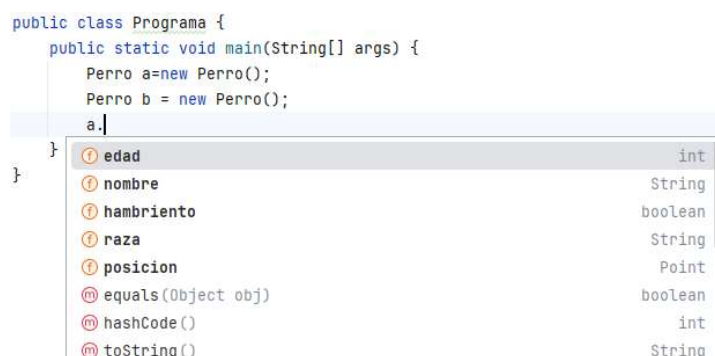
3.3.- Acceso público a las propiedades (**mala práctica!!**)

La clase `Perro` que hemos hecho compila correctamente y con el programa de pruebas que hemos hecho hemos visto que funciona correctamente. Se crean objetos de la clase `Perro` y podemos llamar a sus métodos heredados como `toString` y `hashCode`.

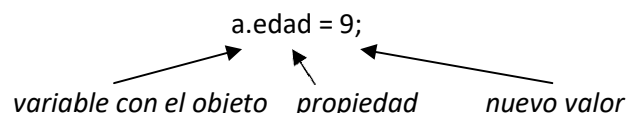
Como hemos dicho en el apartado anterior, las propiedades llevan el modificador `private`. Sin embargo, se lo podemos cambiar y ponerle **public**. Con ello lo que conseguimos es que el programador de aplicaciones pueda verlas y usarlas en los programas. Primero cambiamos el modificador en la clase así:

```
1. package com.animales;
2. import java.awt.Point;
3.
4. public class Perro {
5.     // PROPIEDADES
6.     public String nombre;
7.     public int edad;
8.     public String raza;
9.     public Point posicion;
10.    public boolean hambriento;
11. }
```

Ahora, podremos volver al programa de pruebas y veremos que al darle al punto sobre un objeto Perro, nos aparece la lista de propiedades del perro. Esto sucede porque ahora son públicas y cualquier perfil de programador puede acceder a ellas.



Como las propiedades en realidad son variables ligadas al objeto, podremos modificarlas libremente, utilizando una asignación sobre ellas, así:



Por ejemplo, el siguiente programa crea un Perro, le pone un nombre, una edad y después muestra algunas de sus propiedades por pantalla:

```

1. public class Programa {
2.     public static void main(String[] args){
3.         Perro a = new Perro();
4.         a.nombre="Tim";
5.         a.edad=6;
6.         System.out.println("Nombre: "+a.nombre);
7.         System.out.println("Edad: "+a.edad);
8.         System.out.println("Raza: "+a.raza);
9.         System.out.println("Hambriento: "+a.hambriento);
10.    }
11. }

```

Si lo ejecutamos, veremos que las propiedades nombre y edad del perro creado se han cambiado correctamente, mientras que las propiedades raza y hambriento tienen el valor por defecto de su tipo de dato (como String es un objeto, su valor por defecto según la tabla anterior es null, y como hambriento es boolean, su valor por defecto es false).

```

Nombre: Tim
Edad: 6
Raza: null
Hambriento: false

```

Cuando tenemos más de un objeto, podemos acceder sin ningún problema a las propiedades de cada uno sin más que usar el nombre correcto de la variable. Por ejemplo, este programa crea dos perros y le cambia el nombre a cada uno.

```

1. public class Programa {
2.     public static void main(String[] args){
3.         Perro a = new Perro();
4.         Perro b = new Perro();
5.         a.nombre="Tim";
6.         b.nombre="Junior";
7.         System.out.println("Nombre: "+a.nombre);
8.         System.out.println("Nombre: "+b.nombre);
9.     }
10. }

```

La salida del programa muestra correctamente los nombres de cada perro:

```

Nombre: Tim
Nombre: Junior

```

Dejar que un programador de aplicaciones acceda libremente a las propiedades es muy mala costumbre. En general los programadores de aplicaciones pueden ser personas diferentes a quien han programado la clase y por desconocimiento (o malicia), pueden hacer cosas como esta:

```

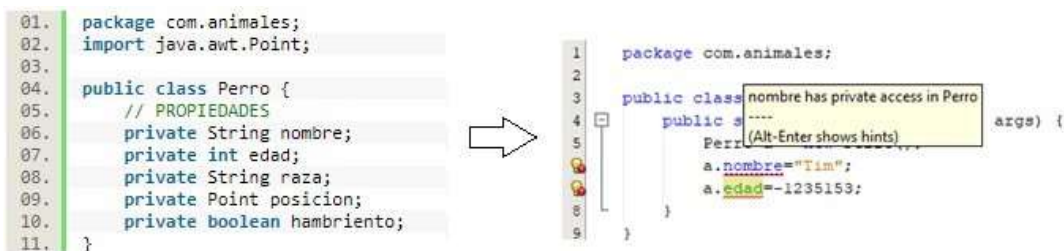
1. public class Programa{
2.     public static void main(String[] args){
3.         Perro a = new Perro();
4.         a.nombre="Tim";
5.         a.edad=-1235153;
6.     }
7. }

```

Como vemos, el programador ha asignado un valor incoherente a la edad de un perro, que no se corresponde con los valores posibles que puede tomar esa propiedad en el mundo real. Hacer que las propiedades admitan valores irreales puede causar serios problemas en el funcionamiento de las clases, propagándose luego a las aplicaciones y favoreciendo la obtención de resultados erróneos difíciles de arreglar.

Por ese motivo, las propiedades nunca se dejan con modificador public, sino que se usa en su lugar, preferentemente, el modificador private (en algunos casos veremos que se admitirá también protected y package). Para trabajar correctamente con las propiedades, la clase deberá ofrecer métodos, que cambien o consulten las propiedades de forma controlada.

Cuando una propiedad tiene modificador que no sea public, intentar acceder a ella en un programa dará un error de compilación como el que se ve en la imagen:

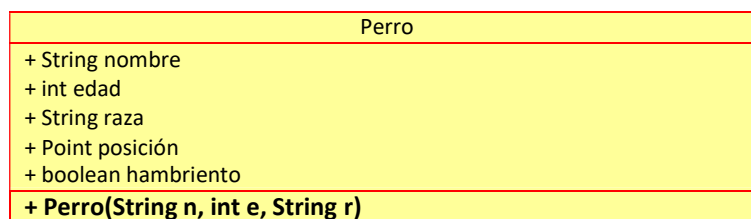


En este momento, deberemos mantener **public** las propiedades para poder comenzar a programar clases, pero más adelante durante el tema, las cambiaremos a **private** y ya siempre las tendremos así.

4.- Programación de métodos constructores

El método constructor permite a un programador de aplicaciones crear un objeto nuevo de la clase. Los parámetros que el constructor recibe entre paréntesis van a servir para dar los **valores iniciales de las propiedades del objeto recién creado**, sustituyendo así a los valores por defecto que tienen asignadas las propiedades.

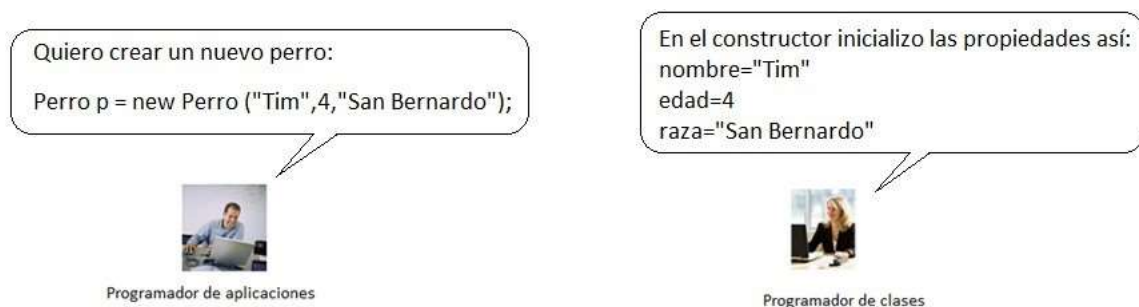
Por ejemplo, supongamos que queremos añadir a la clase un método constructor que permita a los programas crear un Perro indicando su nombre, edad y raza. El diagrama de la clase se quedaría así (observa que al añadir nuestro constructor propio, desaparece el constructor por defecto que ponía Java automáticamente):



Gracias a este método constructor, un programador podrá crear un objeto de la clase Perro escribiendo un programa como este:

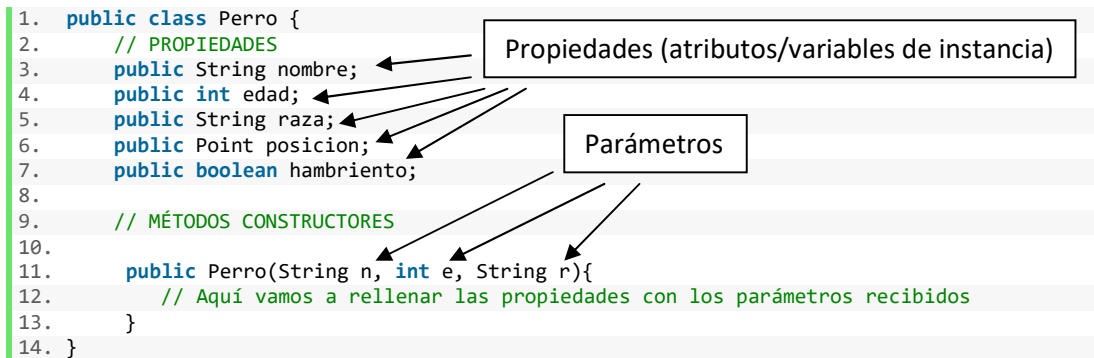
```
1. public class Programa{  
2.     public static void main(String[] args){  
3.         Perro a = new Perro("Tim",4,"San Bernardo");  
4.     }  
5. }
```

Internamente, el método constructor va a coger los valores recibidos entre paréntesis y los va a guardar en las propiedades correspondientes:

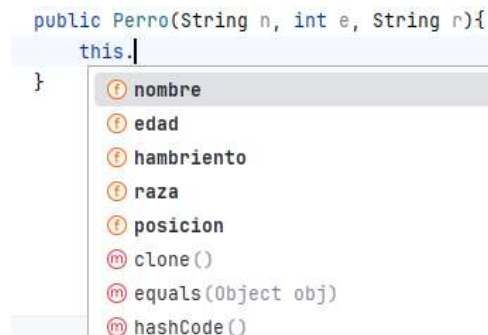


Para programar el método constructor el programador de clases guarda los parámetros recibidos dentro de las propiedades adecuadas. En el ejemplo que estamos viendo, el programador de aplicaciones quiere crear un Perro y usa el constructor pasando "Tim", 4 y "San Bernardo". Estos serían los valores que habría que guardar en las propiedades "nombre", "edad" y "raza".

Vamos a ver como se programaría este constructor en la clase Perro. Simplemente escribimos su cabecera tal y como aparece en el diagrama de la clase y abrimos un bloque de llaves para escribir su código, donde realizaremos la asignación que guardará el valor de los parámetros "n", "e" y "r" dentro de las propiedades "nombre", "edad" y "raza" respectivamente:



Para acceder a una propiedad de la clase, usaremos la palabra **this**⁸, que representa el objeto que está siendo creado por la clase. Por tanto, ese objeto “this” tiene todas las propiedades definidas que hemos definido: nombre, edad, raza, posición, hambriento, y así lo podemos ver si le damos al punto:



Por tanto, para programar el constructor, bastará con asignar en las propiedades correspondientes el valor de los parámetros recibidos, usando **this** para hacer referencia al objeto de la clase Perro que está siendo creado:

```

1. public class Perro {
2.     // PROPIEDADES
3.     public String nombre;
4.     public int edad;
5.     public String raza;
6.     public Point posicion;
7.     public boolean hambriento;
8.
9.     // MÉTODOS CONSTRUCTORES
10.    public Perro(String n, int e, String r){
11.        this.nombre=n;
12.        this.edad=e;
13.        this.raza=r;
14.    }
15. }
16. }

```

A partir de este momento, el programador de aplicaciones ya puede crear un objeto Perro indicando en el constructor su nombre, edad y raza. ¿Qué sucede con las propiedades posición y hambriento? Ambas se quedan con el valor por defecto asociado a su tipo de dato: null para posición y false para hambriento.

⁸ En realidad, la palabra **this** es opcional en Java. Sin embargo, conviene ponerla porque facilita la comprensión de la clase. Además, en otros lenguajes como Python, si es obligatorio su uso.

Podemos comprobarlo con un programa que cree un perro y muestre sus propiedades:

```
1. public class Programa {
2.     public static void main(String[] args) {
3.         Perro p=new Perro("Tim",4,"San Bernardo");
4.         System.out.println("Nombre del perro: "+p.nombre);
5.         System.out.println("Edad del perro: "+p.edad);
6.         System.out.println("Raza del perro: "+p.raza);
7.         System.out.println("¿Hambriento?: "+p.hambriento);
8.         System.out.println("Posición: "+p.posicion);
9.     }
```

Si lo ejecutamos, veremos que funciona perfectamente:

```
Nombre del perro: Tim
Edad del perro: 4
Raza del perro: San Bernardo
¿Hambriento?: false
Posición: null
```

Si queremos, en el constructor podemos cambiar el valor de las propiedades posición que no se han rellenado con un parámetro (posición y y hambriento en este caso) y poner otro diferente. Dicho valor deberá venir indicarlo en la documentación del método.

Por ejemplo, vamos a modificar el constructor de antes, pero ahora, la localización queremos que sea el punto de coordenadas (0,0). Siempre se recomienda que aparezcan en el constructor todas las propiedades (aunque se les den el valor que tomaría por defecto, como le pasa a hambriento).

```
1. public class Perro {
2.     // PROPIEDADES
3.     public String nombre;
4.     public int edad;
5.     public String raza;
6.     public Point posicion;
7.     public boolean hambriento;
8.     // MÉTODOS CONSTRUCTORES
9.     public Perro(String n, int e, String r){
10.        this.nombre=n;
11.        this.edad=e;
12.        this.raza=r;
13.        this.posicion = new Point(0,0);
14.        this.hambriendo = false;
15.    }
16. }
```

Si lo probamos, veremos que el programa nos muestra ahora los valores correspondientes:

```
Nombre del perro: Tim
Edad del perro: 4
Raza del perro: San Bernardo
¿Hambriento?: true
Posición: java.awt.Point[x=0,y=0]
```

A menudo puede ocurrir que los parámetros que recibe el constructor se llamen igual que las propiedades. Por ejemplo, el parámetro “n” podría llamarse “nombre”, igual que la propiedad de la clase. Esto no es ningún problema, puesto que para acceder a la propiedad siempre se utiliza la palabra **this**. De esta forma, si ponemos **this.nombre** nos referimos a la propiedad de la clase, mientras que si solo ponemos **nombre** nos referiremos al parámetro del constructor. De esa forma, **this.nombre=nombre** rellenaría la propiedad “nombre” de la clase con el parámetro “nombre” recibido en el constructor. El IDE nos ayuda a resolver la confusión coloreando de distinta forma dichas variables para así ver que son diferentes, aunque se llamen igual.

4.1.- Protección de las propiedades

Los métodos constructores también pueden comprobar que los valores de las propiedades que introduzca el programador de aplicaciones sean correctos. Si no lo hacen, no habría diferencia entre poner las propiedades públicas o privadas. Una de las utilidades de las clases es **evitar** que el valor de las propiedades puedan tomar un valor incoherente con el mundo real, y recordamos que las propiedades se ponen privadas precisamente para ello.

En el ejemplo de antes, la edad del perro debe ser positiva. Al programar el constructor podemos detectar si la variable “e” es negativa, y en ese caso, poner una edad por defecto (por ejemplo, 1 año). Como veremos más adelante, todo este tipo de decisiones deben aparecer en la documentación que acompaña a la clase.

```
1. public class Perro {
2.     // PROPIEDADES
3.     public String nombre;
4.     public int edad;
5.     public String raza;
6.     public Point posicion;
7.     public boolean hambriento;
8.     // MÉTODOS CONSTRUCTORES
9.     public Perro(String n, int e, String r){
10.        this.nombre=n;
11.        if(e>0){
12.            this.edad=e;
13.        }else{
14.            this.edad=1;
15.        }
16.        this.raza=r;
17.        this.posicion = new Point(0,0);
18.        this.hambriento = false;
19.    }
20. }
```

Para hacer este tipo de comprobaciones y asignaciones es muy útil usar el if corto:

```
1. public class Perro {
2.     // PROPIEDADES
3.     public String nombre;
4.     public int edad;
5.     public String raza;
6.     public Point posicion;
7.     public boolean hambriento;
8.     // MÉTODOS CONSTRUCTORES
9.     public Perro(String n, int e, String r){
10.        this.nombre=n;
11.        this.edad = e>0 ? e:1;
12.        this.raza=r;
13.        this.posicion = new Point(0,0);
14.        this.hambriento = false;
15.    }
16. }
```

Si hacemos un nuevo programa de pruebas y creamos dos perros, por ejemplo uno con edad 10 y otro con edad -4, veremos que el de edad 10 conserva dicho valor, pero el de edad -4 tendrá edad 1, tal y como ha sido programado en el constructor.

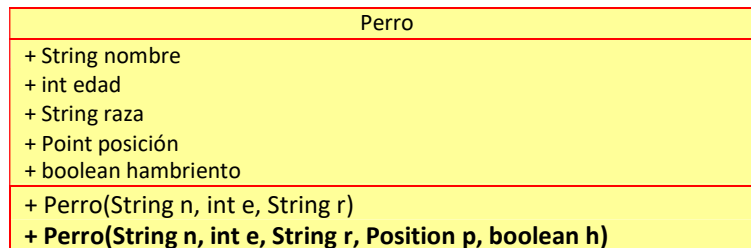
```
public class Programa {
    public static void main(String[] args) {
        Perro a = new Perro("Tim", 10, "San Bernardo");
        Perro b = new Perro("Tom", -1, "Bull dog");
        System.out.println("Edad de Tim: "+a.edad);
        System.out.println("Edad de Tom: "+b.edad);
    }
}
```



```
Edad de Tim: 10
Edad de Tom: 1
```

4.2.- Programación de varios constructores

Vamos ahora a programar el segundo método constructor de la clase Perro. Como ya se sabe, una clase puede tener todos los métodos constructores que decida su programador de clases. Vamos a añadir un constructor que recibe parámetros para todas las propiedades. O sea, ampliamos el diagrama de la clase Perro con el nuevo constructor así:



Para programarlo, simplemente lo escribimos a continuación del que ya tenemos.

```
1. public class Perro {
2.     // PROPIEDADES
3.     public String nombre;
4.     public int edad;
5.     public String raza;
6.     public Point posicion;
7.     public boolean hambriento;
8.     // MÉTODOS CONSTRUCTORES
9.     public Perro(String n, int e, String r){
10.         this.nombre=n;
11.         this.edad=e>0?e:1;
12.         this.raza=r;
13.         this.posicion=new Point(0,0);
14.         this.hambriento=false;
15.     }
16.     public Perro(String n, int e, String r, Position p, boolean h){
17.         this.nombre=n;
18.         this.edad=e>0?e:1;
19.         this.raza=r;
20.         this.posicion=p;
21.         this.hambriento=h;
22.     }
23. }
24. }
```

Como podemos ver, el segundo constructor recibe un valor para asignarlo directamente sobre cada propiedad. Aquí es importante destacar que el parámetro “Position p” es un objeto que ya ha sido creado previamente por el programador de aplicaciones. Es decir, para crear un perro con este constructor, la aplicación lo haría así:

```
1. public class Programa{
2.     public static void main(String[] args){
3.         Position punto = new Position(120,90);
4.         Perro a = new Perro("Tim",4,"San Bernardo",punto,false);
5.     }
6. }
```

Al programador de aplicaciones le puede resultar un poco “rollo” usar este constructor porque debe crear antes un objeto Position. Vamos a facilitarle un poco las cosas (aunque complicándolas al programador de clase) añadiendo un tercer constructor para la clase Perro.

Este nuevo constructor va a recibir dos números enteros, de forma que sea la clase Perro

quien cree el objeto Position para inicializar la propiedad posición.

Perro
+ String nombre + int edad + String raza + Point posición + boolean hambriento
+ Perro(String n, int e, String r) + Perro(String n, int e, String r, Position p, boolean h) + Perro(String n, int e, String r, int x, int y, boolean h)

La clase, con ese nuevo constructor, quedaría así:

```
1. public class Perro {
2.     // PROPIEDADES
3.     public String nombre;
4.     public int edad;
5.     public String raza;
6.     public Point posicion;
7.     public boolean hambriento;
8.     // MÉTODOS CONSTRUCTORES
9.     public Perro(String n, int e, String r){
10.         this.nombre=n;
11.         this.edad=e>0?e:1;
12.         this.raza=r;
13.         this.posicion=new Point(0,0);
14.         this.hambriento=false;
15.     }
16.     public Perro(String n, int e, String r, Position p, boolean h){
17.         this.nombre=n;
18.         this.edad=e>0?e:1;
19.         this.raza=r;
20.         this.posicion=p;
21.         this.hambriento=h;
22.     }
23.     public Perro(String n, int e, String r, int x, int y, boolean h){
24.         this.nombre=n;
25.         this.edad=e>0?e:1;
26.         this.raza=r;
27.         this.posicion=new Position(x,y);
28.         this.hambriento=h;
29.     }
30. }
31. }
```

Es importante saber la diferencia entre el segundo y el tercer constructor. En el segundo, el programador de aplicaciones tenía que crear el objeto Position para la posición del perro, mientras que con el tercero solo tiene que indicar números con las dos coordenadas:

```
1. public class Programa{
2.     public static void main(String[] args){
3.         // Creo un perro con el segundo constructor
4.         Position punto = new Position(120,90);
5.         Perro a = new Perro("Tim",4,"San Bernardo",punto,false);
6.         // Creo un perro con el tercer constructor
7.         Perro a = new Perro("Tim",4,"San Bernardo",120,90,false);
8.     }
9. }
```

4.3.- Constructor principal y constructores secundarios

En todos los constructores que hemos añadido a la clase Perro vemos que hay partes comunes en todos ellos que se repiten constantemente. Por ejemplo la asignación del nombre, la raza, y la protección y asignación de la edad. Nunca es bueno tener código repetido, puesto que nos dificulta el mantenimiento cuando luego tengamos que modificar la clase.

¿Hay alguna forma de escribir el código común a los constructores solo una vez y que no se repita?

La solución consiste en definir un **constructor principal** y que los demás sean **constructores secundarios**⁹.

El constructor principal puede ser el que nosotros queramos, pero lo más cómodo (y lo que se hará el 99.9% de las veces) será coger aquel que recibe un parámetro por cada propiedad y realiza una asignación directa sobre ellas (previa comprobación de las protecciones).

En nuestro ejemplo, el constructor que consideraremos principal será el segundo, porque la clase Perro tiene las propiedades "nombre", "edad", "raza", "posición", "hambriento" y el segundo constructor recibe los parámetros "n", "e", "r", "p", "h" que sirven para darles un valor a todas ellas. Además, dicho constructor ya incluye una protección para comprobar que la edad es positiva.

Por tanto, los demás constructores se considerarán secundarios, y eso significa que **pasarán sus parámetros**, junto con el resto de objetos necesarios para la inicialización, a otro constructor (ya sea principal o secundario)

Para hacer que un constructor secundario pase los parámetros que recibe a otro constructor, se escribe la palabra **this** y entre paréntesis se ponen los parámetros que necesita el constructor utilizado, junto al resto de objetos que necesite.

☞ **IMPORTANTÍSIMO:** En este contexto la palabra **this** tiene un significado diferente del que hemos visto en los apartados anteriores. Cuando se usa la palabra **this** de esta forma, debe ser la **primera línea del constructor**, y no puede haber nada escrito antes de ella.

Por ejemplo, vamos a volver a programar el primer constructor, de forma que llame al principal (que es el segundo). El constructor principal necesita 5 parámetros (uno para cada propiedad), mientras que el primero recibe solo 3 (porque los otros 2 son fijos).

Propiedad	Constructor principal (el segundo)	Constructor secundario (el primero)
nombre	Parámetro n	Parámetro n
edad	Parámetro e	Parámetro e
raza	Parámetro r	Parámetro r
posición	Parámetro p	Objeto Point(0,0)
hambriento	Parámetro h	Valor false

⁹ La terminología "constructor principal" y "constructor secundario" aparece en el lenguaje Kotlin, donde es necesario distinguir explícitamente entre ellos, y además es obligatorio que siempre haya un constructor principal y los demás constructores sean secundarios.

La idea consiste en que el constructor secundario pase al principal los parámetros “n”, “e”, “r” y los dos valores Point(0,0) y false.

Propiedad	Constructor principal (el segundo)	Constructor secundario (el primero)
nombre	Parámetro n	Parámetro n
edad	Parámetro e	Parámetro e
raza	Parámetro r	Parámetro r
posición	Parámetro p	Objeto Point(0,0)
hambriento	Parámetro h	Valor false

Para hacerlo nos vamos al primer constructor y escribimos **this** encerrando entre paréntesis los 5 datos que nos pide entre paréntesis el constructor principal, así:

```

1. public class Perro {
2.     // PROPIEDADES
3.     public String nombre;
4.     public int edad;
5.     public String raza;
6.     public Point posicion;
7.     public boolean hambriento;
8.
9.     // MÉTODOS CONSTRUCTORES
10.    public Perro(String n, int e, String r){
11.        // llama al segundo constructor, pasando lo que necesita entre paréntesis
12.        this(n,e,r,new Point(0,0),false);
13.    }
14.
15.    public Perro(String n, int e, String r, Position p, boolean h){
16.        this.nombre=n;
17.        this.edad=e>0?e:1;
18.        this.raza=r;
19.        this.posicion=p;
20.        this.hambriento=h;
21.    }
22. }
```

Lo que hace la línea marcada es activar el segundo constructor pasándole como parámetros las variables “n”, “e”, “r” que recibe el primer constructor y completar la llamada al segundo constructor pasando un objeto Point(0,0) que creamos ahí mismo y también un boolean false. Cuando hacemos esto se dice que el primer constructor llama al segundo constructor, y como vemos, es un poco más difícil que programar el primer constructor entero (como se hacía antes), pero mejora el mantenimiento ya que se reduce el código y además, se eliminan las redundancias.

Igualmente, podemos programar el tercer constructor para que llame al constructor principal, así:

```

1. public Perro(String n, int e, String r, int x, int y, boolean h){
2.     this(n,e,r,new Point(x,y),false);
3. }
```

De hecho, si lo preferimos, también podemos programar el primer constructor para que utilice el tercero. Aquí tenemos una versión alternativa del primer constructor:

```

1. public Perro(String n, int e, String r){
2.     this(n,e,r,0,0,false);
3. }
```

5.- Test de las clases

5.1.- Test unitarios

Un aspecto de vital importancia para todo programador de clases es comprobar que una clase funciona correctamente. Una posibilidad es hacer un programa que pruebe la clase, tal y como hemos hecho en los apartados anteriores con la clase Perro. Este enfoque está muy bien para programas pequeños, pero en programas grandes es poco práctico porque cuando se hace una modificación en una clase, lo ideal es volver a testear todas las clases para comprobar que ninguna se ha visto afectada, y eso nos llevaría a lanzar un montón de programas y comprobar, de manera “visual” con nuestros ojos que el resultado producido es correcto. Además, haciendo programas para probar las clases tenemos el problema de que podemos formar un lío mezclando unas clases con otras, dejar métodos sin probar o tener tantos programas que nos sea muy difícil localizar luego alguna prueba concreta que queramos hacer.

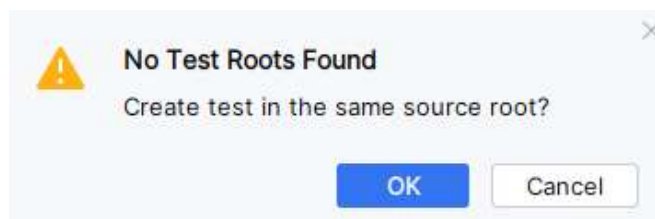
Para resolver estos problemas aparecieron los **frameworks de pruebas**, que son unas herramientas que se integran en los IDE y que nos permiten definir y lanzar test automatizados, de forma que podemos probar en cualquier momento la clase (o clases) que queramos y saber en cuestión de segundos si alguna de ellas falla, sin necesidad de ninguna intervención por nuestra parte. Bastará con mirar un informe para saber qué test han fallado, sin necesidad de tener que poner un programa en marcha.

☞ **Un poco de historia:** Al principio las clases no se probaban y se hacían enteras del tirón. Como puedes imaginar, esto era un disparate gigante porque luego todo fallaba y se hacía prácticamente imposible encontrar los errores entre tanto código. Debido a esto, los programadores comenzaron a hacer un programa de prueba para cada clase. Es decir, por cada clase, creábamos en el mismo proyecto un programa con un main y dentro de él se hacían las pruebas de las clases. Esto es lo que hemos hecho en los apartados anteriores.

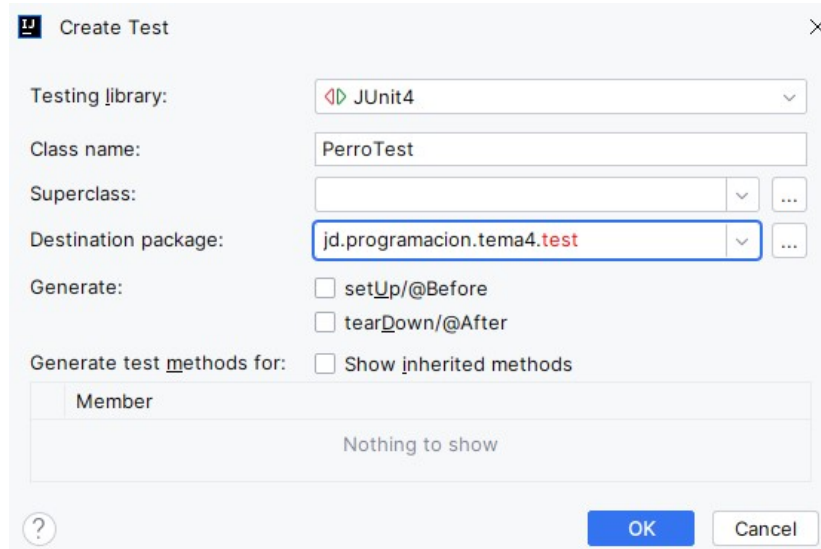
Actualmente se ha dado un paso más y se utilizan los **frameworks de pruebas**, que se integran en el IDE y sirven para probar las clases conforme las vamos programando o incluso para guiar el desarrollo de la clase (metodología TDD, como veremos posteriormente).

Los tests destinados a comprobar que una clase funciona correctamente se denominan **tests unitarios**. Cuando tenemos varias clases y queremos comprobar que funcionan bien en conjunto, los tests destinados a ello se denominan **tests de integración**.

Para crear el test unitario de la clase Perro usando IntelliJ abrimos el código fuente de la clase y pulsamos **ALT + Insert** y elegimos **Test**. Si nos sale la siguiente ventana, pulsaremos OK.



Una vez aceptada la ventana anterior, aparecerá la siguiente ventana, donde podremos elegir el framework de pruebas que usaremos. Usaremos **JUnit4**, que es más compatible y de momento dejaremos todas las opciones con sus valores por defecto. Solamente cambiaremos el “destination package” de los tests, para que se guarden en un paquete separado de las clases del proyecto. No pasa nada porque el paquete de tests no exista y salga marcado en rojo, porque al pulsar OK se creará.



Como vemos en la imagen, es posible que nos salga una advertencia diciendo que JUnit4 no se encuentra en el proyecto. Para agregarlo, pulsaremos **Fix** y **OK** y el problema se solucionará.

Se generará, dentro del paquete que hayamos elegido para guardar los tests una clase llamada **PerroTest** donde programaremos, uno por uno todos los tests de la clase.

El test es un pequeño programa de prueba que se añade a esa clase y tiene el siguiente formato:

```
1. public class PerroTest {
2.     @Test
3.     public void test1(){
4.         // aquí va el código fuente del test
5.     }
6. }
```

Ahora mismo, el test1 no hace nada, porque su bloque de llaves está vacío. Pronto añadiremos en ese bloque una especie de “mini programa de pruebas” que será un test concreto que queramos realizar.

Podemos añadir a la clase PerroTest todos los tests que queramos¹⁰. Por ejemplo, aquí hemos añadido 3 tests diferentes (aunque ninguno hace nada todavía porque sus bloques de llaves están vacíos).

```
1. public class PerroTest {
2.     @Test
3.     public void test1(){
4.         // aquí va el código fuente del test 1
5.     }
6.     @Test
7.     public void test2(){
8.         // aquí va el código fuente del test 2
9.     }
10.    @Test
11.    public void test3(){
12.        // aquí va el código fuente del test 3
13.    }
14. }
```

¹⁰ En realidad el nombre del test es irrelevante. Lo que importa es que aparezca la anotación @Test

5.2.- Asertos

Vamos a ver cómo se programa un test. Si observamos los primeros programas de pruebas que hicimos al comenzar el tema, se utilizaba `System.out.println` para poder “ver” con nuestros ojos la salida del programa y comprobar “en persona” que esta es correcta. Esto nos exige tener que estar delante de dicho programa para ejecutarlo y examinar la salida, lo cual es inviable cuando nuestro programa tiene cientos de clases, hemos hecho un cambio en una de ellas y queremos relanzar todos los programas de prueba para ver si se ha introducido un error colateral en otra clase.


La principal utilidad de los tests es que permiten automatizar todo el proceso de pruebas para que no tengamos que ir mirando con nuestros ojos los resultados de los tests uno por uno.

La idea es que en cada test vamos a poner unas condiciones llamadas **asertos**, que asocian el resultado de un valor que produce nuestra clase con el resultado correcto que debería salir, de manera que si se produce una discordancia, JUnit considera que el test ha fallado y nos avisará con un mensaje de error.

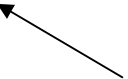
Aunque hay muchos asertos posibles, el principal (con el que es posible hacer cualquier cosa que queramos) es **`assertEquals`**, que permite comprobar si un valor producido por una clase coincide con el resultado que sabemos que debe salir.

Por ejemplo, supongamos que hemos creado un perro llamado “Tim” y queremos comprobar si su propiedad **`p.nombre`** vale, efectivamente, Tim. Dentro del test, escribiremos la siguiente línea para examinar si la propiedad `p.nombre` vale Tim:

`assertEquals("Tim", p.nombre);`



el dato correcto que queremos ver



lo que ha hecho/calculado nuestra clase

En caso de que `p.nombre` valga “Tim”, el test se considera superado, pero si `p.nombre` vale cualquier otra cosa, el test falla y aparecerá un informe con el fallo. De esta forma, usando asertos, podemos lanzar todos los tests que queramos a una clase de forma automatizada.

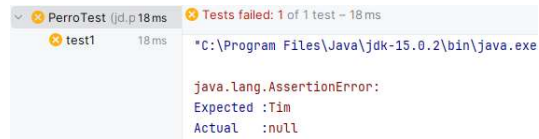
El test que crea un perro llamado Tim y comprueba si el nombre está bien rellenado sería este:

```
1. @Test
2. public void test1(){
3.     Perro p = new Perro("Tim",10,"San Bernardo");
4.     assertEquals("Tim",p.nombre);
5. }
```

Para lanzar el test podemos hacer click en el icono de “play” que hay al lado de cada test, o bien en el que hay en la clase. Dándole a este último lanzaremos todos los tests de la clase. Si todo va bien, veremos un mensaje como este:



En cambio, si se produce un error (por ejemplo, si dejamos el constructor vacío, sin nada, la propiedad p.nombre se quedará con el valor null) saldrá un mensaje como este:



En ese mensaje podemos ver que se esperaba como valor correcto “Tim” y que p.nombre ha devuelto null, produciéndose así el error en el test.

Dentro del test podemos poner todos los asertos que queramos, como por ejemplo, uno para comprobar cada una de las propiedades tras usar el constructor:

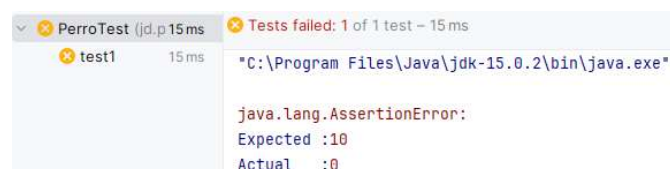
```
1. @Test
2. public void test1() {
3.     Perro p=new Perro("Tim",4,"San Bernardo");
4.     assertEquals("Tim",p.nombre);
5.     assertEquals("San Bernardo",p.raza);
6.     assertEquals(4,p.edad);
7.     assertEquals(null,p.posicion);
8.     assertEquals(false,p.hambriento);
9. }
```

IMPORTANTE: Cuando la propiedad que tenemos que comprobar es de tipo **double**, es necesario incluir un tercer parámetro de **margen de error**, para que, debido a errores de redondeo no se considere el valor exacto, sino que tenga en cuenta una pequeña variación.

Vamos a comprobar que el test nos avisa en caso de fallo. Supongamos que durante el mantenimiento del programa un compañero de nuestro equipo modifica la clase Perro y sin querer le introduce un error típico de principiante, como asignar al revés el parámetro a la propiedad:

```
1. public class Perro {
2.     public String nombre;
3.     public int edad;
4.     public String raza;
5.     public Point posicion;
6.     public boolean hambriento;
7.     public Perro(String n, int e, String r) {
8.         this.nombre = n;
9.         e=this.edad; // ERROR MUY FRECUENTE
10.        raza = r;
11.    }
12. }
```

Si lanzamos el test, veremos que es el ordenador quien se da cuenta de que el valor de la edad del perro no coincide con el valor correcto que esperaríamos encontrar y nos avisa así:



El tema de los test ha evolucionado muchísimo¹¹ y además de **assertEquals** hay otros tipos de asertos, como estos:

- **assertTrue** → Entre paréntesis se pone una condición boolean y el test falla si esa condición no se cumple.
- **assertNotNull** → Se pone dentro de sus paréntesis un objeto y el test falla si el objeto es null.
- **fail** → Directamente hace que el test falle (es útil para indicar que el test se mete en un if o else en el que no debería si todo fuera bien)

Pese a todo, hay que indicar que normalmente haciendo un buen uso de **assertEquals** nos permite cubrir un rango casi total de las necesidades habituales, por lo que durante el curso este será el único tipo de aserto que usaremos.

Ejercicio 1 : Haz un proyecto, crea en él un paquete llamado **tema4.persona** y programa la siguiente clase DNI, que está formada por un DNI compuesto por un número y una letra.

DNI
+ int número
+ char letra
+ DNI(int n, char l)
+ DNI (String dni)

- El primer constructor crea un DNI a partir del número y letra recibidos como parámetros. No se realizará ninguna comprobación de que el DNI sea correcto.
- El segundo constructor crea un DNI a partir de un String que guarda el número con la letra. El método deberá separarlos (se supone que el String proporcionado es correcto y que está formado por 8 dígitos y una letra).

Test: (hacerlos en un paquete llamado **tema4.persona.test**)

- Crea un DNI con número 12345678 y letra Z. Comprueba que ambos valores se han asignado correctamente a las propiedades
- Crea un DNI a partir del String "12345678X". Comprueba que en el número se guarda 12345678 y en la letra X.

Ejercicio 2 : Programa la siguiente clase, que representa una caja que puede estar abierta o cerrada y que tiene un mensaje dentro.

Caja
+ boolean abierto
+ String mensaje
+ Caja(String m)
+ Caja(boolean a, String m)
+ Caja()

¹¹ De hecho, hay librerías como **Hamcrest** que añaden muchos más tipos de asertos, permitiendo incluso hacer razonamientos lógicos con ellos

- El primer constructor crea una caja que tiene el mensaje indicado y está cerrada
- El segundo constructor crea una caja que está abierta según lo indique el primer parámetro y tiene el mensaje del segundo parámetro.
- El tercer constructor crea una caja cerrada que guarda el mensaje “Viva el tema 4”

Test: (hacerlos en el paquete **tema4.test**)

- Crea una caja con el mensaje “Hola amigos” y comprueba que la propiedad “abierto” vale false y la propiedad “mensaje” guarda dicho texto.
- Crea una caja abierta con el mensaje “Adios amigos” y comprueba que las propiedades guardan dichos valores
- Crea una caja con el tercer constructor y comprueba que el mensaje es correcto y la caja está cerrada.

Ejercicio 3 : Programa la siguiente clase, que representa un marcador de baloncesto en el que hay un equipo local un equipo visitante, ambos con una puntuación.

MarcadorBaloncesto
+ String nombreLocal + String nombreVisitante + int puntosLocal + int puntosVisitante + LocalDate fecha
+ MarcadorBaloncesto(String nL, String nV) + MarcadorBaloncesto(String nL, String nV, LocalDate f) - MarcadorBaloncesto(String nL, int pL, String nVl, int pV, LocalDate fecha)

- El primer constructor crea un marcador que recibe los nombres del equipo local y visitante, ambos tienen 0 puntos y se juega en la fecha actual.
- El segundo constructor crea un marcador que recibe los nombres del equipo local y visitante, ambos tienen 0 puntos y se juega en la fecha recibida como parámetro.
- El tercer constructor crea un marcador que recibe los nombres del equipo local y visitante, tienen los puntos que se reciben como parámetros y se juega en la fecha recibida como parámetro.

Test: (hacerlos en el paquete **tema4.test**)

- Crea un marcador con el tercer constructor para el partido Granada, Estudiantes con 100 y 90 puntos respectivamente, para la fecha 10/10/2010 y comprueba que todas las propiedades tienen el valor correcto.
- Crea un marcador con el primer constructor para el partido Granada, Estudiantes y comprueba que los nombres son correctos, la fecha es la de hoy y los puntos de los equipos son 0.
- Crea un marcador con el segundo constructor para el partido Granada, Estudiantes para la fecha 28/2/2000 y comprueba que la fecha es correcta y los puntos son 0

Ejercicio 4 : Añade al paquete tema4.persona del ejercicio 1 la siguiente clase, que representa una Persona, que tiene un DNI:

Persona

```

+ String nombre
+ DNI dni
+ double sueldo
+ LocalDate fechaNacimiento
+ Persona(String n, DNI d, double s, LocalDate fn)
+ Persona(String n, int numDNI, char letraDNI, double s, LocalDate fn)
+ Persona(String n, DNI d)
+ Persona(String n, int numDNI, char letraDNI)

```

- El primer constructor crea una persona a partir de su nombre, DNI, sueldo y fecha de nacimiento pasadas como parámetros.
- El segundo constructor crea una persona a partir de su nombre, número, letra de DNI, sueldo y fecha de nacimiento pasada como parámetro
- El tercer constructor crea una persona cuyo nombre y DNI se pasan como parámetro, gana 900 euros y ha nacido hace 20 años.
- El cuarto constructor crea una persona cuyo nombre, número y letra de DNI se pasan como parámetro, gana 900 euros y ha nacido hace 20 años.

Test: (hacerlos en el paquete **tema4.test**)

- Crea una persona llamada Pepe, con dni 12345678B, sueldo 1500 y fecha de nacimiento 8/3/2007 usando el primer y luego el segundo constructor.
- Crea la persona Antonio con DNI 88776655X con el tercer y luego, con el cuarto constructor y comprueba que el sueldo es 900 y nació hace 20 años.

Ejercicio 5 : Programa la siguiente clase, que es una oficina en la que trabajan varias personas:

```

Oficina
+ String nombre
+ ArrayList<Persona> trabajadores
+ Oficina(String nombre)
+ Oficina(String nombre, int tipo)
+ Oficina()

```

- El primer constructor crea una oficina con el nombre recibido, sin trabajadores.
- El primer constructor crea una oficina que se llama como indica el nombre recibido, y un tipo, que hace lo siguiente:
 - Si el tipo es 0 o mayor de 3, la lista de trabajadores estará vacía.
 - Si el tipo es 1, la lista de trabajadores solo tiene este trabajador:

Antonio Pérez Pérez	11111111H	900	28/2/2000
---------------------	-----------	-----	-----------

- Si el tipo es 2, la lista de trabajadores tendrá al de tipo 1, y también a:

Luis López López	22222222J	1000	10/9/1995
------------------	-----------	------	-----------

- Si el tipo es 3, la lista de trabajadores tendrá a los trabajadores de tipo 2 y a:

Ana Díaz Díaz	33333333P	1200	21/5/1985
---------------	-----------	------	-----------

- El tercer constructor crea una oficina llamada “Industrias DAW”, que es de tipo 3.

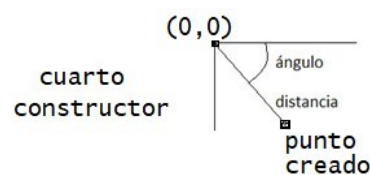
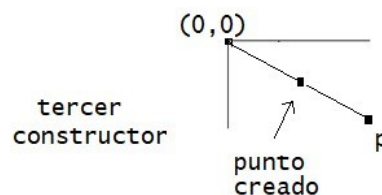
Test: (hacerlos en el paquete **tema4.test**)

- Crea una oficina llamada HP con el primer constructor y comprueba que el nombre es correcto y su lista de empleados tiene 0 elementos.
- Crea cuatro oficinas (todas llamadas HP), cada una de un tipo diferente y comprueba que su lista de trabajadores tiene 0, 1, 2 y 3 elementos respectivamente.
- Crea la oficina HP de tipo 5 y comprueba que tiene 0 empleados.
- Crea la oficina HP de tipo 3 y comprueba que contiene un empleado con dni 22222222J, sueldo 1000 y a otro con DNI 33333333P, sueldo 1200

Ejercicio 6 : Programa la siguiente clase, que representa un punto de la pantalla que está en unas coordenadas (x,y) comprendidas entre 0 y el tamaño de la pantalla.

Punto
+ int x + int y
+ Punto() + Punto(int x, int y) + Punto(Punto p) + Punto(double angulo, double distancia)

- El primer constructor crea un punto situado en las coordenadas (0,0)
- El segundo constructor crea un punto situado en las coordenadas (x,y). Si las coordenadas caen fuera de la pantalla (utiliza la clase **Toolkit** de la librería Java 2D para obtener el tamaño de la pantalla), el punto se pondrá en la esquina inferior derecha de la pantalla.
- El tercer constructor crea un punto situado en el punto medio del origen (0,0) y el punto pasado como parámetro. Observa la imagen adjunta para ver lo que se pide.
- El cuarto constructor crea un punto situado a un ángulo (en grados) y distancia del origen, según se indica la imagen adjunta, redondeando los decimales. (*Sugerencia: usa la definición de seno y coseno para obtener el valor de las coordenadas x e y*)



Test: (hacerlos en el paquete **tema4.test**)

- Crea un punto con el segundo constructor en las coordenadas (120,90) y comprueba que son correctas
- Crea un punto con el segundo constructor situado en las coordenadas (-200,10000) y comprueba que sus coordenadas son la esquina inferior derecha (usa la clase Toolkit en el test).
- Crea un punto situado en medio del origen y el punto (120,90). Comprueba que ha salido el punto de coordenadas (60,45)
- Crea un punto situado a 100 píxeles del origen, formando un ángulo de 45 grados con él. Comprueba que sale el punto de coordenadas (70,70)

6.- Test Driven Development (TDD)

JUnit fue el primer framework de pruebas que apareció. Tal fue su éxito, que inmediatamente fue portado a otros lenguajes (Python, PHP, etc también disponen de sus versiones) y su creador Ken Beck, desarrolló una nueva forma de programar clases que se basa en el uso constante de JUnit. Esta metodología de programación se denomina **TDD (Test-Driven Development, o desarrollo guiado por pruebas)** y se suele pedir mucho hoy día en las ofertas de empleo.

Requisitos deseados:

- Graduado en Ingeniería informática (o equivalente).
- Graduado en Formación Profesional Superior o Medio en Informática.
- Buen nivel de inglés a nivel conversación.
- Experiencia con **Java 8+**.
- Experiencia en arquitectura de **Microservicios y Spring Boot**.
- Experiencia en proyectos reales con **TDD**.
- Valorable experiencia con **JUnit**.
- Deseable experiencia con **Kafka, CI/CD, Cloud**, etc.

La idea del TDD es programar primero los tests y después ir programando la clase para que los tests vayan siendo superados uno por uno

Por ejemplo, queremos programar la siguiente clase Coche usando TDD:

Coche
+ String matricula
+ String marca
+ int velocidad
+ Coche(String matricula, String marca, int velocidad)
+ Coche(String marca)

- Primer constructor: crea un coche cuya matrícula, marca y velocidad se pasan como parámetros. Si se pasa una velocidad negativa, esta se pondrá a 0.
- Segundo constructor: crea un coche con matrícula aleatoria entre 1111 y 9999, matrícula XYZ y velocidad 0.

Cuando se usa TDD es imprescindible disponer de los tests que tiene que cumplir la clase:

1. Crea un coche Ferrari con matrícula 1234ABC, velocidad 15 y comprueba que la marca es Ferrari, la matrícula es 1234ABC y la velocidad es 15
2. Crea un coche Ferrari con matrícula 2153ABX, velocidad -32 y comprueba que la velocidad es 0
3. Crea un coche Ferrari con el segundo constructor y comprueba que la matrícula tiene 7 caracteres, está entre 1111 y 9999, la letra es XYZ y la velocidad es 0.

En la metodología tradicional de clases primero se programaría la clase Coche y después haríamos sus tests. Si queremos usar la metodología TDD, vamos a comenzar realizando un **stub** de la clase, que es una versión rápida de la clase en la que todos los métodos (incluidos los constructores) lanzan una **UnsupportedOperationException**, que es señal de que el método es provisional y está sin programar. Por tanto, comenzamos haciendo el **stub** de la clase Coche, que sería este:

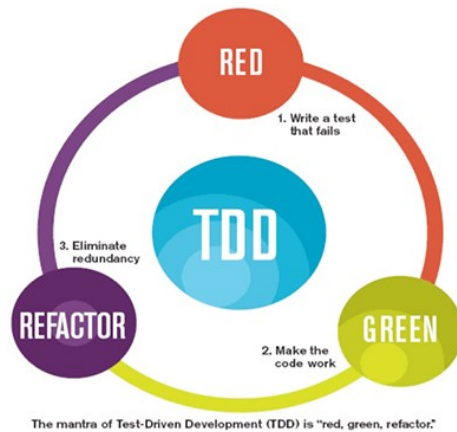
```
1. public class Coche {
2.     public String matricula;
3.     public String marca;
4.     public int velocidad;
5.     public Coche(String matricula, String marca, int velocidad){
6.         throw new UnsupportedOperationException("Sin programar");
7.     }
8.     public Coche(String marca){
9.         throw new UnsupportedOperationException("Sin programar");
10.    }
11. }
```

La idea del stub es que sea muy rápido de programar para que esté disponible inmediatamente para los demás compañeros del equipo y así puedan compilar las clases que dependan de la clase Coche. Cuando las propiedades son privadas, mucha gente no las pone en el stub para así ahorrar más tiempo y hacer que el stub esté disponible antes.

A continuación, una vez que tenemos el stub, ya podemos hacer la clase **CocheTest** y añadiremos el código fuente de todos los tests. Observa que podemos hacerlos porque la clase Coche, aunque no es funcional, compila.

```
1. public class CocheTest {
2.     @Test
3.     public void test1(){
4.         Coche c = new Coche("1234ABC", "Ferrari", 15);
5.         assertEquals("Ferrari", c.marca);
6.         assertEquals("1234ABC", c.matricula);
7.         assertEquals(15, c.velocidad);
8.     }
9.     @Test
10.    public void test2(){
11.        Coche c = new Coche("1234ABC", "Ferrari", -32);
12.        assertEquals(0, c.velocidad);
13.    }
14.    @Test
15.    public void test3(){
16.        Coche c = new Coche("Ferrari");
17.        String matricula = c.matricula;
18.        assertEquals(7, matricula.length());
19.        int numero = Integer.parseInt(matricula.substring(0, 4));
20.        String letras = matricula.substring(4);
21.        if(numero < 1111 || numero > 9999){
22.            fail("El número no está en el rango correcto");
23.        }
24.        assertEquals("XYZ", letras);
25.        assertEquals(0, c.velocidad);
26.    }
27. }
```

Como es lógico, en este momento ninguno de los test funciona, pero ese es el primer paso de la metodología TDD: El **estado red**, que significa que la clase posee al menos un método que falla.



El siguiente paso consiste en elegir uno de los tests y **programar lo mínimo** para que dicho test funcione. Por ejemplo, si cogemos el primer test, nos damos cuenta de que solo con asignar directamente (sin comprobar nada) los parámetros a las propiedades ya lo tenemos:

```
1. public class Coche {
2.     public String matricula;
3.     public String marca;
4.     public int velocidad;
5.     public Coche(String matricula, String marca, int velocidad){
6.         this.matricula=matricula;
7.         this.marca=marca;
8.         this.velocidad=velocidad;
9.     }
10.    public Coche(String marca){
11.        throw new UnsupportedOperationException("Sin programar");
12.    }
13. }
```

Si lanzamos los tests nuevamente veremos que el primero ya funciona, con lo cual vamos bien.

CocheTest (jd. 14 ms)	
✓ test1	3 ms
✗ test2	9 ms
! test3	2 ms

La clase aún sigue en el estado red, por lo que debemos coger otro test que falle y repetir el proceso. Cogemos el segundo test y vemos que para superarlo necesitamos implementar la protección de la velocidad del coche en el primer constructor. Lo programamos:

```
1. public class Coche {
2.     public String matricula;
3.     public String marca;
4.     public int velocidad;
5.     public Coche(String matricula, String marca, int velocidad){
6.         this.matricula=matricula;
7.         this.marca=marca;
8.         if(velocidad>=0){
9.             this.velocidad=velocidad;
10.        }else{
11.            this.velocidad=0;
12.        }
13.    }
14.    public Coche(String marca){
15.        throw new UnsupportedOperationException("Sin programar");
16.    }
17. }
```

Ahora relanzamos los tests y ya tenemos dos tests superados, pero la clase sigue en estado red porque aún hay un test que no va:

✓	test1	7 ms
✓	test2	0 ms
✗	test3	9 ms

Tomamos el último test y lo programamos de forma que se cumplan sus requisitos:

```

1. public class Coche {
2.     public String matricula;
3.     public String marca;
4.     public int velocidad;
5.     public Coche(String matricula, String marca, int velocidad){
6.         this.matricula=matricula;
7.         this.marca=marca;
8.         if(velocidad>=0){
9.             this.velocidad=velocidad >=0;
10.        }else{
11.            this.velocidad=0;
12.        }
13.    }
14.    public Coche(String marca){
15.        int numero = new Random().nextInt(1111,10000);
16.        this.matricula=numero+"XYZ";
17.        this.velocidad=0;
18.    }
19. }
```

Ahora ya todos los tests funcionan, lo que significa que la clase ha pasado al **estado green**, que significa que ha superado todos los tests disponibles hasta ese momento. El siguiente paso es **refactorizar** la clase, que significa mejorar su diseño interno para mejorar la eficiencia o el mantenimiento. En este caso, podemos cambiar el if por una asignación condicional así:

```

1. public class Coche {
2.     public String matricula;
3.     public String marca;
4.     public int velocidad;
5.     public Coche(String matricula, String marca, int velocidad){
6.         this.matricula=matricula;
7.         this.marca=marca;
8.         this.velocidad=velocidad >=0 ? velocidad : 0;
9.     }
10.    public Coche(String marca){
11.        int numero = new Random().nextInt(1111,10000);
12.        this.matricula=numero+"XYZ";
13.        this.velocidad=0;
14.    }
15. }
```

La clase estaría lista hasta que se decidan añadir nuevos requisitos a la clase. Dichos requerimientos deberán ir acompañados de nuevos tests que fallen, de forma que la clase volverá al estado red y se repetirá el proceso.

En general, los pasos del TDD son:

- 1) Se realiza una fase de análisis y diseño de la que se obtiene el diagrama de las clases que se van a desarrollar.
- 2) Se programan los esqueletos (**stub**) de las clases
- 3) Se programan los test de las clases (aunque no compilen).
- 4) Se van programando los métodos, hasta que se superan todas las pruebas (**estado green**). Durante el proceso se pueden añadir y modificar los test ya realizados conforme vaya cambiando el aspecto de la clase.

- 5) Cuando se superan todas las pruebas se realiza una **refactorización**, que consiste en optimizar el código programado, de manera que se eliminen redundancias y se mejoren los algoritmos utilizados, para proporcionar una solución de calidad que supere las pruebas.
- 6) El proceso vuelve a repetirse hasta que el software es desarrollado por completo

El objetivo final del TDD es que no se programe nada sin que exista un test previo que respalde aquello que va a ser programado.

7.- Programación de métodos de instancia

Por nuestra experiencia como programadores de aplicaciones sabemos que los métodos son el elemento fundamental de las clases, ya que son los que permiten que el objeto nos obedezca y realice las cosas que necesitamos para poder hacer nuestro programa.

Como sabemos, los **métodos de instancia** son métodos que están asociados a un objeto, de forma que siempre necesitamos un objeto concreto para poder llamarlos. A su vez, se dividen en:

- **Métodos procedimiento:** Realizan una acción y no devuelven ningún resultado. Van acompañados de palabra **void** en la documentación.
- **Métodos función:** Son aquellos cuya llamada produce un resultado, que es necesario recoger en una variable. Su documentación nos dice el tipo de dato de dicho resultado.

7.1.- Programación de métodos procedimiento

Para programar un método procedimiento bastará con abrir un bloque de código con la misma descripción que encontremos en el diagrama de su clase y a continuación, programar dentro de él todas las tareas que realiza el método teniendo en cuenta los siguientes puntos:

- **Podemos utilizar las propiedades:** Esto es lo más importante cuando programamos un método. Siempre que programemos un método podremos utilizar las propiedades conforme nos sea necesario, cambiándoles su valor, consultándolas, etc. Normalmente los métodos realizan acciones en las que intervienen las propiedades.

Por ejemplo, supongamos que añadimos un método a la clase Perro para cambiar el nombre del perro. Vamos a llamar¹² a dicho método **setNombre**.

¹² Es costumbre en Java comenzar por la palabra **set** el nombre de los métodos que cambian una propiedad. Este tipo de métodos se denominan **setters**.

Perro
+ String nombre + int edad + String raza + Point posición + boolean hambriento
+ Perro(String nombre, int edad, String raza) + void setNombre(String nuevoNombre)

Este método es de tipo procedimiento puesto que tiene la palabra void. Para programarlo lo único que tenemos que hacer es escribirlo en el código fuente de la clase y añadir el código de lo que hace.

Ahora es cuando tenemos que cambiar el chip y pensar que estamos programando lo que sucederá cuando un programador llame a setNombre. ¿Qué le pasa al perro cuando se llama a setNombre? Su nombre debe cambiar. Como el nombre del perro está guardado en la propiedad “nombre”, lo que hay que hacer para que el cambio de nombre sea efectivo es asignar el nombre nuevo (que es el parámetro “nuevoNombre”) a la propiedad, así:

```
1. public void setNombre(String nuevoNombre){
2.     this.nombre = nuevoNombre;
3. }
```

Ahora, desde un test, podríamos crear un perro y cambiarle su nombre:

```
1. @Test
2. public void test(){
3.     Perro a = new Perro("Tim",4,"San Bernardo");
4.     a.setNombre("Junior");
5.     a.assertEquals("Junior",a.nombre);
6. }
```

- **Podemos crear variables auxiliares:** Cuando programamos un método podemos crear todas las variables auxiliares que sean necesarias, sean del tipo que sea. Dichas variables “morirán” al finalizar el método, y por tanto, solo existen dentro de él. La única precaución es que si hacemos una variable que se llame como una propiedad, se producirá ocultación y necesitaremos usar this si queremos acceder a la propiedad.

Por ejemplo, vamos a añadir a la clase perro un método que muestra por pantalla el año de nacimiento del perro.

Perro
+ String nombre + int edad + String raza + Point posición + boolean hambriento
+ Perro(String nombre, int edad, String raza) + void setNombre(String nuevoNombre) + void mostrarAñoNacimiento()

Para programarlo, añadimos a la clase un bloque de código con la cabecera que vemos en el diagrama de la clase. Dentro del bloque de llaves escribimos el código que necesitamos para hacer lo que se nos pide. En este caso, para obtener el año de nacimiento, obtendremos un objeto con el año actual y le restamos la propiedad edad.

```

1.     public void mostrarAñoNacimiento(){
2.         LocalDate hoy = LocalDate.now();
3.         int año = hoy.getYear();
4.         int añoNacimiento = año - edad;
5.         System.out.println("El perro ha nacido en "+añoNacimiento);
6.     }

```

Como vemos, se han usado dos variables auxiliares: una para guardar el objeto `LocalDate` con la fecha actual y otra para el año. Estas variables desaparecen al terminar el método, por lo que en otro método distinto podemos reusar sus nombres.

Ahora, si queremos hacer un test para probarlo no podemos usar asertos y tenemos que hacer un programa de pruebas que muestre salida por pantalla:

```

7.     public class Probador{
8.     public static void main(String[] args){
9.         Perro a = new Perro("Tim",4,"San Bernardo");
10.        a.mostrarAñoNacimiento();

```

Esta situación desagradable lo que nos indica es que el método no ha sido bien diseñado por el analista que ha dado forma a la clase. En general, una clase bien diseñada **nunca** deberá mostrar salida por pantalla, porque:

- Hace que no podamos usar asertos en los test y tengamos que mirar a mano si los valores son correctos
- Fastidia la presentación al programador de aplicaciones, ya que la salida por pantalla puede estar en un idioma o formato diferente al que necesita el programa.

Lo ideal es que el método no muestre la edad por pantalla, sino que la **devuelva** al programador de aplicaciones, para que este haga lo que quiera con ese valor (veremos cómo hacer esto en el próximo apartado).

- **Podemos llamar a métodos de nuestra clase:** Para evitar escribir código repetido, podemos llamar a métodos de la clase que estamos programando. Para hacer referencia al objeto que estamos programando usaremos la palabra **this**, como cuando accedíamos a las propiedades de la clase (esto tiene sentido puesto que **this** es el objeto que está siendo programado con la clase).

Por ejemplo, supongamos que queremos añadir a la clase `Perro` un método llamado **setEdad** que va a hacer dos cosas: por un lado, cambiará la edad del `Perro`, y una vez cambiada, nos mostrará su año de nacimiento. Una primera versión sería así:

```

1.     public void setEdad(int edad){
2.         this.edad = edad;
3.         LocalDate hoy = LocalDate.now();
4.         int año = hoy.getYear();
5.         int añoNacimiento = año - edad;
6.         System.out.println("El perro ha nacido en "+añoNacimiento);
7.     }

```

En esta primera versión estamos repitiendo el código fuente del método `mostrarAñoNacimiento`, por lo que podemos evitar dicha redundancia llamando a ese método. Para hacerlo, usamos la palabra **this**

```

1. public void setEdad(int edad){
2.     this.edad = edad;
3.     this.mostrarAñoNacimiento();
4. }

```

- **Podemos usar programación estructurada:** El código de un método es como si fuese un mini-programa con las acciones que se realizan cuando se llama a dicho método. Por tanto, podemos usar cualquier for, while, if, switch, etc, según sea necesario.

Vamos a añadir un método para que el perro ladre. Si el perro tiene hambre, solo emitirá un ladrido con letras minúsculas. Si el perro no tiene hambre, se emitirán varios ladridos en mayúsculas:

```

1. public void ladrar(){
2.     if(this.hambriento){
3.         System.out.println("guau");
4.     }else{
5.         for(int i=0;i<5;i++){
6.             System.out.println("GUAU");
7.         }
8.     }
9. }

```

Nuevamente, igual que pasaba con el método mostrarAñoNacimiento, este método muestra datos en pantalla, por lo que su diseño no es correcto por parte del analista.

Ejercicio 7 : Añade al paquete **tema4.persona** esta clase, que representa una cuenta corriente:

CuentaCorriente
+ int número + double saldo
+ CuentaCorriente() + CuentaCorriente(int número) + CuentaCorriente(int número, double saldo) + void añadirDinero(int cantidad) + void retirarDinero(int cantidad)

- El primer constructor crea una cuenta con número aleatorio entre 0 y 1000, y saldo 0.
- El segundo constructor crea una cuenta corriente cuyo número de cuenta se pasa como parámetro y tiene saldo 0 euros.
- El tercer constructor crea una cuenta corriente cuyo número de cuenta se pasa como parámetro y tiene el saldo indicado como parámetro.
- Los métodos de añadir y retirar cantidad, añaden o retiran del saldo la cantidad indicada. En caso de que no haya saldo suficiente, el método no hará nada.

Test

- Crea una cuenta con número 1253 y saldo 850. Comprueba que las propiedades se rellenan correctamente
- Crea la misma cuenta de antes y añádele 100€. Comprueba que el saldo es 950. A continuación, retira 200€ y comprueba que el saldo es 750€. Por último, retira 1000€ y comprueba que el saldo sigue siendo 750€.

Ejercicio 8 : Añade a la clase Caja estos métodos:

Caja
+ boolean abierto
+ String mensaje
+ Caja(String m)
+ Caja(boolean a, String m)
+ Caja()
+ void setMensaje(String m)
+ void pasarMayusculas()

- setMensaje: Si la caja está abierta, cambia el mensaje de la caja, y si no, no hace nada.
- pasarMayusculas: Pasa a mayúsculas el mensaje de la caja

Test: (se añaden a CajaTest)

- Crea una caja con mensaje "Hola" y usa setMensaje para cambiarlo por "Adios". Comprueba que la caja está cerrada y el mensaje que contiene es "Hola".
- Crea una caja con mensaje "Hola", abre la caja y usa setMensaje para cambiarlo por "Adios". Comprueba que la caja está abierta y el mensaje que contiene es "Adios"
- Crea una caja con mensaje "Hola", pásalo a mayúsculas y comprueba que es "HOLA".

Ejercicio 9 : Añade a la clase MarcadorBaloncesto los métodos indicados:

MarcadorBaloncesto
+ String nombreLocal
+ String nombreVisitante
+ int puntosLocal
+ int puntosVisitante
+ LocalDate fecha
+ MarcadorBaloncesto(String nL, String nV)
+ MarcadorBaloncesto(String nL, String nV, LocalDate f)
- MarcadorBaloncesto(String nL, int pL, String nV, int pV, LocalDate fecha)
+ void añadirCanasta(char equipo, int puntos)
+ void reset()

- añadirCanasta: Si recibe una L, añade los puntos indicados al local. Si recibe una V, los añade al visitante. Solo se añadirán los puntos que sean 1,2 o 3.
- reset: Hace que el marcador se vuelva a poner a 0.

Test:

- Crea un partido Granada-Estudiantes con marcador 0,0 y comprueba que el marcador queda 33 a 26 tras añadir estas canastas:

	Canasta de 1 punto	Canasta de 2 puntos	Canasta de 3 puntos
Granada	5	8	4
Estudiantes	3	10	1

- Crea el partido Granada-Estudiantes con 100-80. Resetealo y comprueba que queda 0 a 0.
- Crea un partido Granada-Estudiantes con marcador 10-12 y haz que un equipo marque una canasta de 4 puntos. Comprueba que el marcador sigue siendo 10-12.

Ejercicio 10 : Modifica la clase Persona para que quede de esta forma:

Persona
+ String nombre + DNI dni + double sueldo + LocalDate fechaNacimiento + CuentaCorriente cuenta
+ Persona(String n, DNI d, double s, LocalDate fn) + Persona(String n, int numDNI, char letraDNI, double s, LocalDate fn) + Persona(String n, DNI d) + Persona(String n, int numDNI, char letraDNI) + void aumentarSueldo(int porcentaje) + void cobrarSueldo()

- En todos los constructores deberá crearse una cuenta corriente para el empleado. Dicha cuenta tendrá saldo 0 y el número de cuenta será aleatorio.
- El método aumentarSueldo aumenta el sueldo del empleado el porcentaje indicado como parámetro. Por ejemplo, si el sueldo es 1000 y se llama al método pasando un 50, el sueldo final del empleado será 1500.
- El método cobrarSueldo añadirá al empleado su sueldo en su cuenta corriente.

Test:

- Crea una persona llamada Juan, con dni 11111111H y sueldo 1500. Comprueba que el saldo de su cuenta corriente es 0. A continuación, haz que cobre su sueldo 3 veces y comprueba que el saldo de su cuenta es 4500. Súbele el sueldo un 25% y haz que cobre. Comprueba que el saldo es 6375.

Ejercicio 11 : Modifica la clase Oficina para que quede de esta forma:

Oficina
+ String nombre + ArrayList<Persona> trabajadores
+ Oficina(String nombre) + Oficina(String nombre, int tipo) + Oficina() + void añadirEmpleado(Persona p) + void añadirEmpleado(String nombre, String DNI, double sueldo, LocalDate fechaNac)

- **Primer añadirEmpleado:** Añade a la oficina el trabajador pasado como parámetro.
- **Segundo añadirEmpleado:** Crea y añade una persona con los parámetros recibidos.

Test:

- Crea una oficina vacía y añade un empleado llamado Antonio con dni 44444444M y sueldo 2000 (no importa la fecha de nacimiento). Comprueba que la oficina tiene un solo empleado, cuyo dni es 44444444M.

En este ejercicio vemos que el método añadirEmpleado está disponible en dos versiones, que se diferencian por su lista de parámetros. Estas versiones diferentes del mismo método se denominan “sobrecargas”, o lo que es lo mismo, “el método añadirEmpleado está sobrecargado”

7.2.- Programación de métodos función

La programación de un método función es muy parecida a la programación de un método procedimiento. De hecho, las cuatro normas que hemos visto en el punto anterior para los métodos procedimiento también sirven para programar métodos función.

La única diferencia es que al final del método tenemos que devolver un resultado al programa que ha llamado al método. Para ello, se utiliza la palabra reservada **return**. Cuando escribimos **return**, la ejecución del método termina y el valor que se escribe al lado de la palabra **return** es devuelto al programa.

Se recomienda utilizar una sola vez la palabra `return`, justo al final del método.

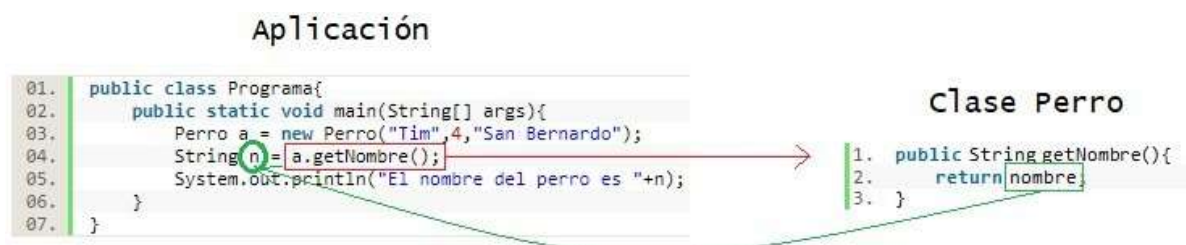
Como ejemplo, vamos a añadir a la clase `Perro` que estamos haciendo en los ejemplos un método llamado **`getNombre`**¹³ que devuelva al programador de aplicaciones el nombre del perro:

Perro
+ String nombre + int edad + String raza + Point posición + boolean hambriento
+ Perro(String nombre, int edad, String raza, Position posición, boolean hambriento) + String getNombre()

Como el objetivo de ese método es devolver el valor del nombre del perro, y este está almacenado en la propiedad “nombre”, lo único que hay que hacer para programar dicho método es usar la palabra **return** para devolver al programador de aplicaciones dicho dato:

```
1. public String getNombre(){  
2.     return this.nombre;  
3. }
```

De esta forma, lo que estamos diciendo es que cuando un programador de clases llame al método `getNombre`, la respuesta que da dicho método es el valor guardado en la propiedad “nombre”. Como ya sabemos, el programador de aplicaciones debe guardar dicho resultado en una variable de tipo `String`.



Vamos a hacer un test para comprobar que nuestro método `getNombre` funciona

¹³ Se denominan **getters** a los métodos **get** que nos devuelven el valor de las propiedades. En Java son muy habituales este tipo de métodos, aunque como veremos más adelante, pronto van a dejar de usarse estos nombres.

correctamente. Simplemente crearemos un perro y llamaremos al método `getNombre`, guardando su resultado en una variable. Por último, con un aserto comprobaremos que dicha variable guarda el nombre correcto del perro.

```
1. @Test
2. public void test() {
3.     Perro p=new Perro("Tim",4,"San Bernardo");
4.     String nombre=p.getNombre();
5.     assertEquals("Tim", nombre);
6. }
```

Gracias a métodos como `getNombre`, podemos modificar la clase `Perro` para que todas las propiedades dejen de ser `public` y ponerlas **`private`**. De esa forma, las propiedades estarán ocultas y a salvo de que un programador de aplicaciones les de valores indebidos. Los métodos `getNombre`, `getEdad`, etc, controlarán el acceso al valor de la propiedad:

Perro
- String nombre - int edad - String raza - Point posición - boolean hambriento
+ Perro(String nombre, int edad, String raza, Position posición, boolean hambriento) + String getNombre() + int getEdad() + String getRaza() + Point getPosicion() + boolean getHambriento()

```
1. public class Perro {
2.     private String nombre;
3.     private int edad;
4.     private String raza;
5.     private Point posicion;
6.     private boolean hambriento;
7.     public Perro(String n, int e, String r) {
8.         this.nombre = n;
9.         this.edad=e;
10.        this.raza = r;
11.    }
12.    public String getNombre() {
13.        return this.nombre;
14.    }
15.    public int getEdad() {
16.        return this.edad;
17.    }
18.    public String getRaza() {
19.        return this.raza;
20.    }
21.    public Point getPosicion() {
22.        return this.posicion;
23.    }
24.    public boolean getHambriento() {
25.        return this.hambriento;
26.    }
27. }
```

Es importante indicar que al hacer un cambio en la estructura de la clase, los test que tengamos dejarán de compilar. Eso no es ningún problema, salvo que deberemos irnos a los test y volver a reprogramarlos adaptándolos a la nueva situación. Por ejemplo, el cambio de propiedades nos obliga a reprogramar el test que hicimos en uno de los ejemplos:

```

1. @Test
2. public void test() {
3.     Perro p=new Perro("Tim",4,"San Bernardo");
4.     assertEquals("Tim",p.nombre);
5.     assertEquals("San Bernardo",p.raza);
6.     assertEquals(4,p.edad);
7.     assertEquals(null,p.posicion);
8.     assertEquals(false,p.hambriento);
9. }

```



```

01. @Test
02. public void test() {
03.     Perro p=new Perro("Tim",4,"San Bernardo");
04.     assertEquals("Tim",p.getNombre());
05.     assertEquals("San Bernardo",p.getRaza());
06.     assertEquals(4,p.getEdad());
07.     assertEquals(null,p.getPosicion());
08.     assertEquals(false,p.getHambriento());
09. }

```

test cuando las propiedades
eran public (ya no compila)

test actualizado a propiedades
privadas y getters

Los métodos más fáciles de programar son los getters, porque lo único que hacen es devolver el valor de una propiedad con la palabra return. Sin embargo, hay métodos que pueden ser más complicados y tener más líneas de código fuente, con bucles, condicionales, variables auxiliares, llamadas a otros métodos, etc. En este tipo de métodos es donde se recomienda que la palabra return aparezca una sola vez al final.

Ejemplo: Vamos a añadir un método llamado **getEsperanzaVida** que nos devuelva la esperanza de vida del perro, según su raza. Para ello nos basaremos en esta tabla:

Raza	Esperanza de vida (en años)
Maltés, Chihuahua, Yorkshire	18
Gran Danés, San Bernardo, Mastín	8
Grifón, Bull Terrier	14
Cualquier otra raza	-1

La programación de este método es algo más complicada y se puede hacer de varias formas. La más simple consiste en hacer un if para comprobar si la raza coincide con alguna de las que salen en la tabla. En dicho caso se rellena una variable “esperanza” con el valor de la tabla, y el método termina con un return de dicho valor. Lo haríamos así:

```

1. public int getEsperanzaVida(){
2.     int esperanza=-1;
3.     if(this.raza.equals("Maltés")||this.raza.equals("Chihuahua")||this.raza.equals("Yorkshire")){
4.         esperanza =18;
5.     }else if(this.raza.equals("Gran Danés")||this.raza.equals("San Bernardo")||this.raza.equals("Mastín")){
6.         esperanza =8;
7.     }else if(this.raza.equals("Grifón")||this.raza.equals("Bull Terrier")){
8.         esperanza =14;
9.     }
10.    return esperanza;
11. }

```

En lugar del código anterior, podemos usar un switch mejorado para devolver la esperanza de vida según la raza. Recordemos que el switch mejorado podía devolver un valor que podíamos guardar en una variable. En este caso, no guardaremos el resultado del switch en ninguna variable, sino que lo devolveremos directamente:

```

1. public int getEsperanzaVida(){
2.     return switch(this.raza){
3.         case "Maltés", "Chihuahua", "Yorkshire" -> 18;
4.         case "Gran Danés", "San Bernardo", "Mastín" -> 8;
5.         case "Grifón", "Bull Terrier" -> 14;
6.         default -> -1;
7.     }
8. }

```


El problema del switch mejorado es que no está disponible en versiones antiguas de Java, y si estamos manteniendo un proyecto viejo no lo podremos usar. La primera forma que hemos visto no tiene buen mantenimiento, ya que si queremos añadir nuevas razas, tenemos que buscar el lugar adecuado y puede ser algo lioso. Una alternativa con mejor mantenimiento usa listas. Haremos tres listas con las razas y miraremos si la raza está en alguna de ella, adaptando la esperanza de vida a la lista en la que se encuentre:

```

1. public int getEsperanzaVida(){
2.     int esperanza=-1;
3.     List<String> lista18 = List.of("Maltés", "Chihuahua", "Yorkshire");
4.     List<String> lista8 = List.of("Gran Danés", "San Bernardo", "Mastín");
5.     List<String> lista14 = List.of("Grifón", "Bull Terrier");
6.     if(lista18.contains(raza)){
7.         esperanza =18;
8.     }else if(lista8.contains(this.raza)){
9.         esperanza =8;
10.    }else if(lista14.contains(this.raza){
11.        esperanza =14;
12.    }
13.    return esperanza;
14. }

```

Ejercicio 12 : Pon privadas las propiedades de la clase Persona y añade estos métodos:

Persona
+ String getNombre() + DNI getDNI() + double getSueldo() + LocalDate getFechaNacimiento() + CuentaCorriente getCuentaCorriente() + boolean esMayorEdad() + boolean tieneDinero() + boolean esMileurista()

- Los getters devuelven el valor de la correspondiente propiedad.
- esMayorEdad: Devuelve true si la persona es mayor de 18 años.
- tieneDinero: Devuelve true si la cuenta corriente del empleado tiene saldo positivo.
- esMileurista: Devuelve true si la persona gana menos de 1200 euros

Test:

- Modifica todos los test que tengas para que compilen y sigan funcionando
- Usa cualquier constructor para crear una persona con 15 años (dan igual sus datos) y comprueba que esMayorEdad devuelve false.
- Usa cualquier constructor para crear una persona con 40 años (dan igual sus datos) y comprueba que esMayorEdad devuelve true
- Crea una persona y comprueba que su método tieneDinero devuelve 0.
- Crea una persona con sueldo 900 y comprueba que su método esMileurista devuelve true.
- Crea una persona con sueldo 1800 y comprueba que su método esMileurista devuelve false.

Ejercicio 13 : Pon privadas las propiedades de la clase Caja y añade estos métodos:

Caja
+ void abrir() + void cerrar() + boolean esAbierta() + String getMensaje()

- abrir: Abre la caja
- cerrar: Cierra la caja
- esAbierta: Devuelve true si la caja está abierta y false si está cerrada
- getMensaje: Si la caja está cerrada, devuelve null, y si la caja está abierta devuelve el mensaje que hay dentro de la caja.

Test:

- Modifica todos los test que tengas para que compilen y sigan funcionando. Usa el método getMensaje en todos los lugares donde antes accedías de forma directa a la propiedad “mensaje” de la clase (ahora no puedes acceder a ella porque es private y el test no la puede ver).
- Haz un test que cree una caja con el mensaje “Hola”. Abre la caja, obtén el mensaje que contiene y comprueba que sale null

Ejercicio 14 : Pon privadas las propiedades de la clase Oficina y añade estos métodos:

Oficina
+ int getTotalEmpleados() + int getTotalEmpleadosMileuristas() + List<Persona> getMileuristas() + boolean trabaja(Persona p) + void pagarEmpleados() + void mostrarInformeEmpleados()

- getTotalEmpleados: Devuelve el número de trabajadores que hay en la oficina
- getTotalEmpleadosMileuristas: Devuelve el número de trabajadores mileuristas.
- getMileuristas: Devuelve una lista formada por los empleados que son mileuristas.
- trabaja: Devuelve true si la persona pasada como parámetro trabaja en la empresa.
- pagarEmpleados: Hace que todos los empleados cobren su sueldo.
- mostrarInformeEmpleados: Muestra por pantalla un listado con todos los empleados de la empresa. Por cada empleado se mostrará su nombre, su sueldo y si es mileurista.

Test:

- Modifica todos los test que tengas para que compilen y sigan funcionando.
- Crea una oficina vacía y añade dos empleados con sueldos 1000 y 2000. Comprueba que el método trabaja nos devuelve true si le pasamos cualquier de ellos. Por último, paga a los empleados y comprueba que los saldos de esos empleados son 1000 y 2000 € respectivamente.

Ejercicio 15 : Pon privadas las propiedades de MarcadorBaloncesto y añade estos métodos:

MarcadorBaloncesto
+ int getPuntosLocal() + int getPuntosVisitante() + boolean ganaLocal() + boolean ganaVisitante() + boolean hayEmpate()

- getPuntosLocal y getPuntosVisitante devuelven los puntos de los respectivos equipos
- ganaLocal, ganaVisitante y hayEmpate devuelven true según esté ganando el equipo local, el visitante, o ambos equipos tengan los mismos puntos.

Test:

- Modifica todos los test que tengas para que compilen y sigan funcionando.
- Crea el partido Granada – Estudiantes con marcador 80-80 y comprueba que hayEmpate devuelve true y ganaLocal y ganaVisitante devuelven false.
- Crea el partido Granada – Estudiantes con marcador 90-80 y comprueba que el método ganaLocal devuelve true y ganaVisitante y hayEmpate devuelven false.

Ejercicio 16 : Crea la siguiente clase DNIMejorado, que representa un DNI cuyo número de letra se calcula automáticamente según la tabla y el procedimiento del ejercicio 19 del tema 4.

DNIMejorado
- int número - char letra
+ DNI(int n) + int getNumero() + char getLetra() - char calcularLetra(int n)

- El constructor crea un DNI cuyo número se pasa como parámetro y cuya letra se obtiene llamando al método privado calcularLetra.
- Los métodos getNúmero y getLetra devuelven el número y la letra del DNI
- El método privado calcularLetra recibe un número y nos devuelve la letra que tendría el DNI correspondiente a ese número teniendo en cuenta el procedimiento del ejercicio 19 del tema 4.

El método privado calcularLetra es un ejemplo de un método auxiliar que solo le interesa al programador de clases, y por tanto, lo hace privado para que nadie más lo pueda ver.

Test:

- Crea un DNIMejorado con número 11111111 y comprueba que getNumero devuelve 11111111 y getLetra devuelve H

8.- Programación de métodos que lanzan excepciones

Sabemos que los métodos pueden lanzar excepciones cuando fallan por algún motivo. También sabemos que hay dos tipos de excepciones:

- **CheckedExceptions:** Son excepciones que se corresponden con un fallo que no es culpa del programador (por ejemplo, se cae la conexión a Internet). El programador de aplicaciones está obligado a capturar en un try-catch estas excepciones.
- **RuntimeExceptions:** Son excepciones que se producen por una cosa que el programador ha hecho mal. Por ejemplo, acceder a una posición de una lista que no existe.

Cuando programamos una clase, podemos elegir si queremos que nuestro método lance CheckedExceptions o RuntimeExceptions. Como veremos en el próximo tema, incluso podemos crear nuestros propios tipos de excepciones. No obstante, en este tema vamos a hacer que nuestros métodos lancen excepciones de las predefinidas en Java.

8.1.- Métodos que lanzan RuntimeExceptions

Las RuntimeExceptions son excepciones que suelen lanzarse cuando un programador de aplicaciones usa mal un objeto, como por ejemplo, cuando llama a métodos pasando valores incorrectos para los parámetros. Las RuntimeException más frecuentes son:

IllegalArgumentException	Se lanza cuando el programador de aplicaciones ha llamado a un método y alguno de los parámetros es incorrecto (ejemplo, una edad negativa)
IllegalStateException	Se lanza cuando el programador de aplicaciones da una orden a un objeto que no está listo para ello (ejemplo, un perro duerme y se llama al método comer)

Por ejemplo, en el método **setEdad** de la clase Perro un programador de aplicaciones puede pasar un número negativo. Al comienzo del tema lo que hicimos en esa situación fue asignar una edad por defecto, pero es mucho más apropiado hacer que el método falle lanzando una excepción (siempre es mejor que un programador reciba una excepción a que se encuentre con un valor por defecto inesperado porque no se ha leído la documentación de la clase).

Un método puede lanzar una RuntimeException en algún punto de su código fuente, creando un objeto de la clase de la excepción que quiere lanzar, y usando la palabra **throw**.

Ejemplo: Vamos a modificar el método setEdad de la clase Perro para que lance una IllegalArgumentException cuando se llame al método setEdad con un número negativo.

```
1. public void setEdad(int e){
2.     if(e>0){
3.         this.edad = e;
4.     }else {
5.         IllegalArgumentException ex=new IllegalArgumentException("Edad negativa");
6.         throw ex;
7.     }
8. }
```

Como vemos, se crea un objeto de la clase `IllegalArgumentException` (normalmente los tipos de excepción reciben en su constructor un `String` con un mensaje que indica por qué se está lanzando la excepción) y usando la palabra **throw** se lanza al programador de aplicaciones. Ambos pasos se pueden hacer en una sola línea (y es lo recomendado), así:

```
1. public void setEdad(int e){
2.     if(e>0){
3.         this.edad = e;
4.     }else {
5.         throw new IllegalArgumentException("Edad negativa");
6.     }
7. }
```

Es importante señalar que cuando se lanza la excepción el método revienta y ya no prosigue su ejecución. Tras lanzarse la excepción, el programador de aplicaciones recibe el error y, si no ha puesto un `try-catch` en su programa, este finalizará abruptamente su ejecución.

Vamos ahora a ver cómo se hace un test que espera que se lance la excepción. Nuestro test creará un perro y le pondrá la edad a -8. En este caso, el lanzamiento de la excepción nos indica que el test ha sido superado y sería malo si esta no se lanzase. Para realizar esta situación, incluimos la palabra **expected**, junto con el tipo de error que se espera, dentro de la anotación `@Test`, así:

```
1. @Test(expected=java.lang.IllegalArgumentException.class)
2. public void testHambre2(){
3.     Perro p = new Perro("Tim",4,"San Bernardo");
4.     p.setEdad(-8);
5.     fail("No se ha lanzado la IllegalArgumentException que se esperaba");
6. }
```

Si ejecutamos el test, vemos que se da por superado, lo cual es lógico porque `p.setEdad(-8)` hace que se lance una `IllegalArgumentException`, que es lo que “espera” (expected) el test. Si por algún motivo no se lanza la excepción, en la línea 5 vemos la palabra **fail**, que hará que el test falle inmediatamente.

Ejercicio 17 : Modifica el segundo método constructor de la clase `Oficina` para que se lance una `IllegalArgumentException` si se pasa un tipo mayor que 3 y haz un test para comprobarlo.

Ejercicio 18 : Programa el método `añadirCanasta` del marcador de baloncesto para que lance una `IllegalArgumentException` si se pasa un char que no sea L,V o unos puntos no validos y haz un test.

8.2.- Métodos que lanzan CheckedExceptions

Las checked exceptions son errores que se producen cuando se da alguna causa ajena al programador de aplicaciones. Por tanto, no son errores suyos y no se le puede echar la culpa de que aparezcan. Java obliga a que todas las checked exceptions sean capturadas con un `try-catch`¹⁴.

El lanzamiento de las checked exception es igual al que hemos visto para las runtime exception, pero además deberemos incluir la palabra **throws** (cuidado con la s final) con el tipo de excepción lanzada en la línea que abre el método. Además, este tipo de excepciones deberán aparecer explícitamente en el diagrama de clases y la documentación correspondiente.

¹⁴ Aunque la idea de Java tenía buena intención, el tiempo ha demostrado que trabajar con checked exceptions es muy engorroso. Por este motivo, ningún otro lenguaje obliga la captura de excepciones.

Ejemplo: Vamos a programar un método **comer()** en la clase Perro que lo alimente. Lo haremos de esta forma: si el perro está hambriento, se muestra un mensaje por pantalla y el perro ya no tiene hambre. En cambio, si el perro no tiene hambre, haremos que se lance una excepción. En la documentación, este método aparecería así:

Perro
- String nombre - int edad - String raza - Point posición - boolean hambriento
+ Perro(String nombre, int edad, String raza) + void comer() throws IOException

En lugar de IOException, en el diagrama de la clase podemos encontrar cualquier tipo de checked exception, como por ejemplo SQLException. La documentación siempre nos dirá cuál es el tipo de excepción que tenemos que poner junto a la palabra “throws”.

A la hora de programar este método, lo que haremos será usar un if para comprobar el valor de la propiedad “hambriento”. Si es true, simplemente la pondremos a false porque el perro ya no tendrá hambre. En cambio, ¿qué haremos si “hambriento” vale true? Ese es justo el momento donde debemos **crear y lanzar la excepción**.

- Para crear la excepción, creamos un objeto de la clase **IOException** (o la que nos diga la documentación) y en su constructor pondremos un String con el mensaje explicando por qué se lanza la excepción.
- Para lanzar la excepción usaremos la palabra **throw** y pondremos a su lado el objeto Exception que hemos creado. Esto hará que se interrumpa la ejecución del método y que automáticamente el programa se meta en el catch.

Todo esto que hemos dicho se hará así:

```

1. public void comer() throws IOException {
2.     if(hambriento){
3.         System.out.println("El perro ha comido");
4.         this.hambriento=false;
5.     }else {
6.         // creamos la excepción
7.         IOException e=new IOException("El perro porque no tiene hambre");
8.         // lanzamos la excepción
9.         throw e;
10.    }
11. }
```

En la práctica los pasos de crear la excepción (línea 7) y lanzarla (línea 9) se unifican en uno, así:

```

1. public void comer() throws IOException {
2.     if(hambriento){
3.         System.out.println("El perro ha comido");
4.         this.hambriento=false;
5.     }else {
6.         // creamos y lanzamos la excepción en la misma línea
7.         throw new IOException("El perro porque no tiene hambre");
8.     }
9. }
```

Para comprobar que todo funciona correctamente vamos a hacer dos test: uno para probar que si el perro tiene hambre no se lanza la excepción y otro que verifique que se lanza la excepción si el perro no tiene hambre.

El primer test es simple. Basta con crear un perro que tenga hambre y llamar al método comer. Tras ello, comprobamos que el perro deja de tener hambre usando un aserto. En el caso de que indebidamente se lance una excepción, usaremos la palabra **fail** para indicar que el test ha fallado.

```
1. @Test
2. public void testHambre1() {
3.     Perro p=new Perro("Tim",4,"San Bernardo");
4.     // me aseguro de que el perro tenga hambre
5.     p.setHambriento(true);
6.     try {
7.         // doy de comer al perro
8.         p.comer();
9.         // compruebo que después de comer el perro ya no tiene hambre
10.        assertEquals(false,p.getHambriento());
11.    }catch(IOException ex){
12.        // si entra aquí es porque el método comer no ha funcionado bien
13.        fail("El método comer lanza una excepción y no debía");
14.    }
15. }
```

Si lo probamos veremos que funciona correctamente.

El segundo test es igual que los que vimos para las RuntimeExceptions y se hace con la palabra **expected**, poniendo ahora IOException. En el test crearemos un perro sin hambre y le daremos de comer. En este caso, esperamos que se lance una excepción y para ello hay que incluir el parámetro **expected** dentro de la anotación **@Test** esperando una IOException:

```
1. @Test(expected=java.lang.IOException.class)
2. public void testHambre(){
3.     Perro p=new Perro("Tim",4,"San Bernardo");
4.     // me aseguro de que el perro tenga hambre
5.     p.setHambriento(false);
6.     // doy de comer al perro
7.     p.comer();
8.     // compruebo que después de comer el perro ya no tiene hambre
9.     fail("No se ha lanzado la excepción");
10. }
```

Ejercicio 19 : Añade a la clase Oficina el siguiente método:

Oficina
+ double getSueldoMedio() throws IOException

- getSueldoMedio: Devuelve la media de los sueldos de todos los empleados de la oficina. Si la oficina no tiene empleados el método lanza una IOException.

Test:

- Crea una oficina sin empleados y añade 4 empleados de sueldos 1000,2000,1500 y 900 € respectivamente. Comprueba que getSueldoMedio devuelve 1350. Usa un margen de error de 0.1
- Crea una oficina sin empleados y comprueba que getSueldoMedio lanza IOException

Ejercicio 20 : Modifica el método **retirarDinero** de la clase **CuentaCorriente** para que lance una **IOException** si se intenta retirar una cantidad de dinero mayor que el saldo de la cuenta.

CuentaCorriente
+ void retirarDinero(int cantidad) throws IOException

Test:

- Crea una **CuentaCorriente** con 2000€ e intenta retirar de ella 6000€. Comprueba que se lanza una **IOException**.

8.3.- Programar un método sin usar try-catch en su interior

Cuando se programa un método que lanza una checked exception, Java nos perdona el uso del try-catch para dicha excepción dentro del método si necesitamos llamar a alguna librería cuyo método lance dicha excepción. O sea, si nuestro método pone *throws IOException* y durante su programación necesitamos llamar a un método de alguna librería que lance **IOException**, **no necesitaremos** encerrar las líneas que usan la librería en try-catch. Java entenderá que si la librería lanza **IOException**, nuestro método lanzará automáticamente la misma **IOException**.

Ejemplo: Vamos a programar este método llamado **guardar** en la clase **Perro**. Guardará en un archivo cuya ruta se pasa como parámetro, los valores de las propiedades del **Perro**. En caso de producirse un error al guardar el archivo (falta de espacio en disco, falta de permisos, etc), se lanzará una **IOException**.

Perro
+ void guardar(String ruta) throws IOException

Una primera versión de este método podría ser esta:

```
1. public void guardar(String ruta) throws IOException{
2.     try{
3.         PrintWriter p=new PrintWriter(ruta);
4.         p.println(this.nombre);
5.         p.println(this.edad);
6.         p.println(this.raza);
7.         p.println(this.posicion.toString());
8.         p.println(this.hambriento);
9.         p.close();
10.    }catch(IOException ex){
11.        throw ex;
12.    }
13. }
```

Lo que estamos haciendo con este código es escribir en un archivo los valores de las propiedades, y si se produce una excepción, la capturamos y la lanzamos. Si hacemos tests de la clase, podremos ver que todo funciona correctamente.

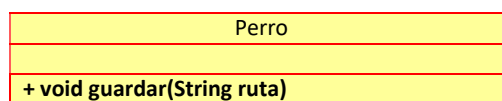
Sin embargo, puesto que el método pone *throws IOException* en su declaración, podemos

quitarnos ese paso de “capturar y lanzar” de las líneas 10 y 11 debido a que Java nos permite quitar el try-catch que lanza IOException, quedando todo así:

```
1. public void guardar(String ruta) throws IOException{
2.     PrintWriter p=new PrintWriter(ruta);
3.     p.println(nombre);
4.     p.println(edad);
5.     p.println(raza);
6.     p.println(posicion.toString());
7.     p.println(hambriento);
8.     p.close();
9. }
```

Poner throws al comienzo de los métodos parece algo muy bueno y cómodo porque nos quita escribir los try-catch en el interior del método, pero hay que tener **muchísimo cuidado** con eso.

Supongamos que nos dan este diagrama para que lo programemos:



- Guarda los datos del perro en un archivo. Si se produce un error, se mostrará en pantalla un mensaje con el error.

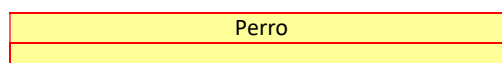
Si nos dan este diagrama, lo que nos están pidiendo es que capturemos en el interior del método la posible excepción y mostremos un mensaje, así:

```
1. public void guardar(String ruta) {
2.     try{
3.         PrintWriter p=new PrintWriter(ruta);
4.         p.println(this.nombre);
5.         p.println(this.edad);
6.         p.println(this.raza);
7.         p.println(this.posicion.toString());
8.         p.println(this.hambriento);
9.         p.close();
10.    }catch(IOException ex){
11.        System.out.println("Se ha producido este error: "+ex.getMessage());
12.    }
13. }
```

Si unilateralmente el programador decide que no tiene ganas de poner try-catch en el interior del método, o le da a una bombilla del IDE, puede programar esto:

```
1. public void guardar(String ruta) throws IOException{
2.     PrintWriter p=new PrintWriter(ruta);
3.     p.println(this.nombre);
4.     p.println(this.edad);
5.     p.println(this.raza);
6.     p.println(this.posicion.toString());
7.     p.println(this.hambriento);
8.     p.close();
9. }
```

Pero esto, ahora se correspondería con este otro diagrama de clases, que, aunque no lo parece, es **completamente diferente** de lo que nos habían pedido:



+ void guardar(String ruta) throws IOException

- Guarda los datos del perro en un archivo. Si se produce un error, se lanzará al programa una IOException

Teniendo en cuenta que en las empresas todos los compañeros del equipo de desarrollo comparten los mismos diagramas de clases, es muy importante programarlos tal y como nos los dan, porque si introducimos una modificación en ellos, el resto de lo que hagan los compañeros ya no compilará y nuestra conducta habrá sido fuente de problemas.

Por ejemplo, con la versión que nos han pedido, los compañeros usarán la clase Perro así:

```
1. Perro p = new Perro("Tim",4,"San Bernardo");
2. p.guardar("c:/tim.txt");
```

Pero si guiados por la comodidad programamos la segunda versión, el código anterior ya no compilará debido a que guardar ahora lanzaría una IOException, que los compañeros deberían capturar así:

```
1. try{
2.     Perro p = new Perro("Tim",4,"San Bernardo");
3.     p.guardar("c:/tim.txt");
4. }catch(IOException e){
5.     System.out.println("Error al guardar tim.txt");
6. }
```

Pero esto no es lo que esperan escribir los compañeros cuando leen del diagrama de clases del proyecto, porque ellos tendrán la versión original, en la que guardar no lanza excepción y la usarán de acuerdo a dicha documentación.

Normalmente, los **analistas** son los que diseñan las clases y deciden si un método debe llevar “throws” o no en el diagrama de la clase. El programador **nunca puede decidir esto por su cuenta**, porque si lo hace y va en contra del diseño realizado por los analistas, estará incumpliendo la documentación que estos han diseñado y hará que el código que estén programando otros compañeros no compile por culpa de alguien que no ha seguido al pie de la letra la documentación del proyecto.

Por tanto, como siempre, habrá que leer la documentación de las clases que elaboran los analistas y cumplirla escrupulosamente cuando programamos para que todos los compañeros del equipo de desarrollo tengan la misma visión del software que se está construyendo.

Ejercicio 21 : Añade el siguiente método guardar a la clase MarcadorBaloncesto:

MarcadorBaloncesto
+ void guardar(String ruta) throws IOException

- guardar: Escribe en el archivo de texto cuya ruta se pasa como parámetro el nombre del equipo local, sus puntos, el nombre del equipo visitante y sus puntos con el siguiente formato:

nombre local: puntos local – nombre visitante: puntos visitantes

Test:

- Crea un marcador para el partido Granada-Estudiantes con resultado 12-8. Llama al método guardar pasando como parámetro test.txt. A continuación, el test deberá comprobar que existe un archivo llamado text.txt y abrirlo. Se leerá una línea del archivo y se comprobará que dicha línea es

Granada:12 – Estudiantes:8

9.- Programación de métodos recursivos

Ya hemos visto que cuando programamos un método podemos utilizar cualquier otro método de la clase. Sin embargo, aunque pueda sonar un poco raro, también podemos llamar **al mismo método** que estamos programando.

Un método es **recursivo** cuando se llama a sí mismo

¿Por qué puede ser necesario esto? El motivo está en que muchas veces la tarea que tenemos que resolver se puede expresar en función de la misma tarea que estamos realizando, pero pasándole un dato de entrada más pequeño.

Ejemplo: Estamos haciendo una clase y queremos ponerle un método que calcule la potencia de un número, o sea, a^n , siendo a y n números enteros.

+ void potencia(int a, int n)

Como sabemos de matemáticas, la potencia se calcula de esta forma:

$$a^n = a * a * a * a \dots * a \rightarrow \text{hay } n \text{ multiplicaciones}$$

Esta definición nos permite programar el método **potencia** usando un bucle for. Simplemente iniciamos un acumulador con 1 y hacemos un bucle en el que en cada iteración se multiplica dicho acumulador por a . De esta forma, al repetir ese bucle n veces, habremos multiplicado a consigo mismo n veces y como resultado tendremos la potencia a^n .

```
1. public int potencia(int a, int n){
2.     int acumulador = 1;
3.     for(int i=0; i<n; i++){
4.         acumulador *= a;    // *= es como +=, pero multiplicando
5.     }
6.     return acumulador;
7. }
```

Este código fuente es la **versión iterativa** del método potencia, porque utiliza bucles tal y como hemos estudiado en el tema 3. Sin embargo, vamos a ver otra forma diferente de programar este método, usando no la definición de potencia, sino una forma alternativa que nos permite calcular a^n a partir del resultado de la potencia anterior, a^{n-1} con esta fórmula:

$$a^n = a * a^{n-1}$$

Si reescribimos la fórmula anterior usando nuestro método potencia, tendremos esto:

$$\text{potencia}(a,n) = a * \text{potencia}(a,n-1)$$

Como podemos ver, la potencia de a elevado a n la podemos conseguir a partir de la multiplicación de “ a ” por la potencia “ a elevado a $n-1$ ” (por ejemplo, 5^3 se puede calcular como $5*5^2$). Vamos a volver a escribir el código fuente del método potencia, pero usando esto último que hemos puesto: simplemente potencia(a,n) retorna el producto de a con la llamada a potencia($a,n-1$)

```
1. public int potencia(int a, int n){
2.     return a * this.potencia(a,n-1);
3. }
```

Como vemos, en la línea señalada estamos viendo cómo el método que estamos programando (potencia) se llama a sí mismo, pero con un dato más pequeño (observa que el método recibe el parámetro “ n ” y la llamada interna llama al mismo método (potencia) pero pasando “ $n-1$ ”, que es un número más pequeño). Este tipo de llamada, donde el método se llama a si mismo, se llama **llamada recursiva**.

El código anterior aún no funciona, puesto que al lanzarlo se forma la siguiente cadena infinita de llamadas recursivas, que nos produciría un **StackOverflowError**:¹⁵

potencia(a,n) → potencia ($a,n-1$) → potencia ($a,n-2$) → ... → potencia ($a,n-10000$) → ...

Para que no pase lo anterior, es muy importante asegurarnos de que en algún momento se termina la serie de llamadas recursivas. Es decir, se necesita un momento, llamado **caso base**, donde el método es capaz de devolver un valor directamente, sin necesidad de hacer una llamada recursiva. Normalmente, el caso base es una situación donde el resultado del cálculo es muy evidente, siendo un valor conocido, que no requiere ningún cálculo.

En nuestro ejemplo el caso base se encuentra cuando el número “ n ”, que es el que va disminuyendo en cada llamada recursiva, se hace 0. Como sabemos de matemáticas, elevar cualquier número a 0 siempre da como resultado 1. Si nos fijamos bien, en cada llamada recursiva se disminuye en 1 el exponente, de forma que habrá un momento donde se tendrá que calcular potencia($a,0$). Si en dicho momento aprovechamos para devolver 1 sin hacer más llamadas recursivas, ya lo tendremos todo solucionado.

```
1. public int potencia(int a, int n){
2.     // variable para guardar el resultado
3.     int resultado=0;
4.     if(n==0){
5.         // si el exponente es 0, el resultado siempre sale 1 (caso base)
6.         resultado=1;
7.     }else{
8.         // si el exponente no es 0, calculamos la potencia haciendo una llamada recursiva
9.         resultado=a*potencia(a,n-1);
10.    }
11.    return resultado;
12. }
```

Ahora, al llamar al método potencia(a,n), este delega en potencia($a,n-1$) que a su vez lo hará en potencia($a,n-2$) y así sucesivamente. Como en algún momento el segundo parámetro llegará a 0 y ese caso produce un resultado directo, la serie de llamadas recursivas no es infinita, y se obtiene una cadena de resultados que “sube” hasta obtener el resultado de potencia(a,n).

Como ejemplo, aquí vemos gráficamente como la llamada a potencia($5,4$) daría lugar primero a una serie de llamadas recursivas que al llegar al caso base se convierte en una cadena de resultados

¹⁵ La capacidad de hacer llamadas recursivas está limitada y este error significa que se ha desbordado dicha capacidad.

que “sube” hasta obtener el resultado final de potencia(5,4)

potencia(5,4) → potencia (5,3) → potencia (5,2) → potencia (5,1) → potencia (5,0) = 1
20 ← 15 ← 10 ← 5 ← 1 (caso base)

Podemos comprobar mediante un test que el método funciona. Por ejemplo, podemos comprobar que el método va bien para algunas potencias:

```
1. @Test
2. public void testPotencia1(){
3.     assertEquals(32,potencia(2,5));
4.     assertEquals(121,potencia(11,2));
5.     assertEquals(1,potencia(113,0));
6.     assertEquals(2401,potencia(7,4));
7. }
```

Como acabamos de comprobar, nuestro método recursivo para multiplicar funciona perfectamente. Sin embargo, la cantidad de llamadas recursivas que pueden realizarse tiene un límite máximo, que cuando se supera se produce un `StackOverflowError`. Por otra parte, si comparamos la eficiencia, la versión recursiva tiene **menor eficiencia** que la correspondiente versión iterativa debido a que con cada llamada recursiva aumenta la saturación del procesador.

Java no lo admite, pero lenguajes como Kotlin incorporan una optimización denominada **tailrecursion** que permite que programando de forma especial los métodos recursivos estos se puedan compilar de forma indistinguible a como lo harían sus versiones iterativas, lo cual elimina los problemas de eficiencia y limitación del número de llamadas recursivas.

Por último, vamos a comentar qué pasaría si en nuestro método potencia elegimos el caso base cuando el exponente llega a 1. Esto es natural planteárselo, porque elevar cualquier número a 1 es algo muy sencillo, porque sale siempre el propio número. No hay ningún problema y lo podemos programar también. Se haría así (esta vez usamos la asignación condicional en lugar del if):

```
1. public int potencia(int a, int n){
2.     return n==1? a : a * potencia(a,n-1);
3. }
```

Si lanzamos los tests, veremos que todos funcionan bien excepto el que calcula 113^0 , donde se espera 1 como resultado correcto, pero se produce un **StackOverflowError**. La presencia de ese error nos indica que el caso base nunca se alcanza y se desborda la capacidad del procesador para realizar más llamadas recursivas.

En nuestro ejemplo esto sucede porque hemos programado el método para que el exponente descienda hasta llegar al 1. Por tanto, si el exponente empieza en 0, irá disminuyendo a -1, -2, -3, etc y nunca llegará a 1, produciéndose el error. Por tanto, cuando se trabaja con recursividad, es muy importante elegir bien el caso base, de forma que nos aseguremos de que siempre se alcance y el método recursivo termine correctamente.

La recursividad es una técnica que se utiliza mucho para simplificar problemas que pueden llegar a ser muy complicados, como los que aparecen cuando se trabajan con datos dispuestos en estructura de **árbol**. Sin embargo, no es una técnica imprescindible, ya que se ha demostrado que los problemas que se resuelven con recursividad también se pueden resolver por los procedimientos iterativos habituales, que además, terminan siendo más eficientes. No obstante, en muchas ocasiones se mantiene el enfoque recursivo debido a que el código es mucho más corto y fácil de comprender y mantener. Al final, la idea de la recursividad es:

{ ¿la tarea es evidente? (caso base) → devolvemos el resultado

método(tarea) →

En caso contrario llamamos a **método(tarea más pequeña)** y usamos su resultado para obtener **método(tarea)**

Ejercicio 22: En matemáticas el “factorial de un número” es una operación que consiste en multiplicar dicho número por todos los anteriores a él, hasta llegar al 1. Por ejemplo:

$$\text{factorial}(5) = 5 * 4 * 3 * 2 * 1$$

Realiza una clase que contenga un método **+ int factorial(int n)** que permita calcular el factorial de un número positivo. Se considerará como caso base el 0, cuyo factorial es igual a 1. Realiza los siguientes tests a dicha clase:

factorial(0)	1
factorial(1)	1
factorial(5)	120
factorial(7)	5040
factorial(20)	-2102132736 → ¿tiene sentido este resultado?
factorial(2000000)	StackOverflowError
factorial(-2)	IllegalArgumentException

Ejercicio 23: Realiza una clase que contenga un método llamado **+ boolean esPar(int n)**. Este método devolverá true si el número pasado como parámetro es par, sin usar los operadores de división (/) o resto (%). Para hacerlo, hay que tener en cuenta que un número es par si su anterior es impar. El caso base se da en el número 0, que es par. Aquí tienes los tests:

esPar(0)	true
esPar(1)	false
esPar(8)	true
esPar(17)	false

Ejercicio 24: *Versión recursiva del algoritmo de selección para ordenar un array*

Realiza la siguiente clase (y un test para probarla), cuyo objetivo es ordenar los números de un array de menor a mayor usando la versión recursiva del algoritmo de selección (ver tema 3).

OrdenacionSeleccion
+ void ordenar(int[] array) - void ordenar(int[] array, int pos)

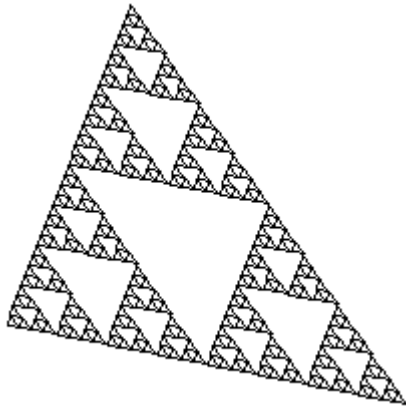
- El primer método ordenar simplemente llama al segundo pasando un 0 como parámetro.
- El segundo método ordenar hace esto:
 - Busca el menor elemento del array, comenzando en la posición “pos”
 - Una vez encontrado, se intercambia el elemento del array de la posición “pos” por el mínimo que se ha encontrado en el paso anterior
 - Llamamos nuevamente al mismo método para ordenar el array a partir de la posición “pos+1”
 - El caso base se produce cuando “pos” supera el tamaño de la lista

Test:

- Realiza un test que cree este array: 4, 1, 3, 9, 2, 6, 14, 10, 11, 19, 20, 8, 5, 12 y usa el método ordenar para ordenarlo. Comprueba que el array ordenado es: 1,2,3,4,5,6,8,9,10,11,12,14,19,20

Ejercicio 25: El triángulo de Sierpinski

Programa la siguiente clase, que sirve para dibujar el famoso triángulo de Sierpinski en la Consola DAW.



El triángulo de Sierpinski es un dibujo creado por el matemático polaco Waclav Sierpinski en 1919, aunque los romanos ya lo conocían. Se trata de un triángulo cuyo interior se descompone en tres triángulos, que a su vez, están dibujados de la misma manera. De esta forma, podemos ver cómo un triángulo de Sierpinski contiene infinitos triángulos de Sierpinski.

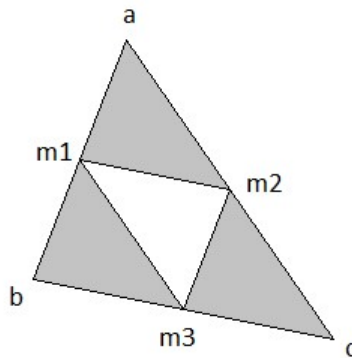
*Las figuras geométricas que se autocontienen de manera infinita (como este dibujo) se denominan **fractales**, y tienen la propiedad de que no son objetos lineales (como las rectas) ni tampoco planos (como el interior de una circunferencia). Su dimensión está entre 1 y 2. La dimensión del triángulo de Sierpinski es $\log(3)/\log(2) \approx 1.5849$*

Triangulo
- Point a - Point b - Point c
+ Triangulo(Point a, Point b, Point c) - Point getPuntoMedio(Point p1, Point p2) + void dibujar(Graphics g) - void dibujar(Graphics g, int profundidad)

- Las propiedades son los vértices "a","b" y "c" del triángulo.
- El método getPuntoMedio devuelve las coordenadas del punto medio del segmento que une p1 con p2. Dicho punto medio está definido por estas coordenadas:

$$\left(\frac{p1.x + p2.x}{2}, \frac{p1.y + p2.y}{2} \right)$$

- El primer método dibujar simplemente llama al segundo pasando 0 como parámetro
- El segundo método dibujar hace esto:
 - Si el nivel de profundidad es igual a 10, el método termina sin hacer nada más (este sería el caso base). En caso contrario, hacemos los siguientes apartados:
 - Dibuja los lados del triángulo "a","b","c"
 - Calcula las coordenadas de los puntos medios de los lados del triángulo y los guarda en variables "m1","m2" y "m3" tal y como se ve en el siguiente dibujo:



- A continuación tres objetos Triangulo para formar los triángulos que tienen color gris en el dibujo
- Llama al método dibujar de dichos triángulos, pero sumando 1 al nivel de profundidad de partida.

Test: Debido a que la clase muestra continuamente salida gráfica, no es apropiado hacer tests de JUnit (salvo para el método getPuntoMedio). En esta ocasión el mejor test es hacer este programa de pruebas:

- Haz un programa que cree una Consola DAW y en su capa canvas dibuje el triángulo de Sierpinski con estos vértices: a(520,90), b(1000,880), c(50,880). Prueba a cambiar las coordenadas y observa que el triángulo se dibuja siempre.

10.- Programación de métodos estáticos

Además de métodos de instancia, las clases también tienen métodos estáticos, que son aquellos que pueden ser llamados directamente sobre la clase, sin necesidad de usar un objeto. Por tanto, los métodos estáticos son métodos que tiene la clase, pero que no están vinculados a ningún objeto en particular.

La programación de métodos estáticos es exactamente igual a los métodos de instancia, pero tenemos que tener esta precaución: **no es posible usar propiedades ni métodos de instancia cuando programamos un método estático**. El motivo es sencillo: las propiedades y métodos de instancia son variables que tiene dentro el objeto que estamos programando. Cuando hacemos un método estático, no estamos programando la respuesta de ningún objeto, sino la de la propia clase, y por tanto, no podemos tener acceso a las propiedades y métodos de instancia.

Como ejemplo de método estático, vamos a crear la siguiente clase **Calculadora**, que va a tener métodos estáticos para sumar, restar y dividir.

Calculadora
<pre>+ static int sumar(int a, int b) + static int restar(int a, int b) + static int dividir(int a, int b) throws Exception + static int inverso(int a) throws Exception</pre>

Los métodos de la clase son estáticos, y por tanto pueden ser llamados en cualquier momento por un programador de aplicaciones. Para programarlos simplemente copiamos la declaración del método tal y como la encontramos en el diagrama de la clase y escribimos en su interior el código que realiza lo que hay que hacer, tal y como ya hemos visto.

Vamos a ver los tres primeros métodos:

```

1. public class Calculadora{
2.     public static int sumar(int a, int b){
3.         return a+b;
4.     }
5.     public static int restar(int a, int b){
6.         return a-b;
7.     }
8.     public static int dividir(int a, int b) throws Exception{
9.         int resultado=0;
10.        if(b!=0){
11.            resultado=a/b;
12.        }else {
13.            IllegalArgumentException ex=new IllegalArgumentException("No se puede dividir entre 0");
14.            throw ex;
15.        }
16.        return resultado;
17.    }
18. }

```

Para programar el método de invertir necesitamos calcular $1/a$, lo cual podemos hacer reutilizando el método dividir. Como el método es estático, **no podemos usar this** para acceder a él, ya que es un método de la clase y no de un objeto concreto. En lugar de this, usaremos el nombre de la clase, de forma que dicho método quedará así:

```

1. public static int dividir(int a) throws Exception{
2.     return Calculadora.dividir(1,a);
3. }

```

Ejercicio 26: Crea la siguiente clase **Fracción**, que representa una Fracción que tiene un numerador y un denominador. Para que sea más fácil, no entraremos en simplificar la fracción.

Fracción
- int numerador - int denominador
+ Fracción(int numerador, int denominador) + int getNumerador() + int getDenominador() + double getValorReal() + Fracción getInversa() + static Fraccion sumar(Fraccion a, Fraccion b) + static Fraccion multiplicar(Fraccion a, Fraccion b) + static Fraccion dividir(Fraccion a, Fraccion b)

- Los getters y el constructor actúan como se espera de ellos.
- El método getValorReal divide el numerador entre denominador y nos da un double con el resultado (con decimales) de dicha división
- El método getInversa devuelve una nueva fracción que se obtiene invirtiendo el numerador y el denominador de la fracción que estamos programando.
- El método sumar suma dos fracciones, teniendo en cuenta que la suma se calcula así:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

- El método multiplicar realiza la multiplicación de dos fracciones, teniendo en cuenta que la multiplicación se calcula así:

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$$

- El método dividir realiza la división de dos fracciones, teniendo en cuenta que la división se calcula multiplicando la primera por la inversa de la segunda

Test:

- Usa la clase Fracción para hacer las siguientes operaciones y comprueba que los resultados son los indicados:

$\frac{1}{2} + \frac{3}{4}$	$\frac{10}{8}$
$\frac{2}{5} + \frac{3}{7}$	$\frac{29}{35}$
$\frac{1}{2} : \frac{4}{3}$	$\frac{3}{8}$
$\frac{1}{2} \left(\frac{3}{7} + \frac{8}{5} \right)$	$\frac{71}{70}$

Ejercicio 27: Modifica la clase Oficina, de esta forma: pon privados sus métodos constructores, y haz varios métodos estáticos que devuelvan un objeto Oficina, según su tipo:

Oficina
- String nombre
- ArrayList<Persona> trabajadores
- Oficina(String nombre)
- Oficina(String nombre, int tipo)
- Oficina()
+ static Oficina getOficinaVacia()
+ static Oficina getOficinaPequeña()
+ static Oficina getOficinaMediana()
+ static Oficina getOficinaGrande()

- getOficinaVacia: Devuelve una oficina de tipo 0
- getOficinaPequeña: Devuelve una oficina de tipo 1
- getOficinaMediana: Devuelve una oficina de tipo 2
- getOficinaGrande: Devuelve una oficina de tipo 3

Es muy frecuente poner los constructores privados, para forzar a que los programadores de aplicaciones tengan que usar un método estático para obtener los objetos

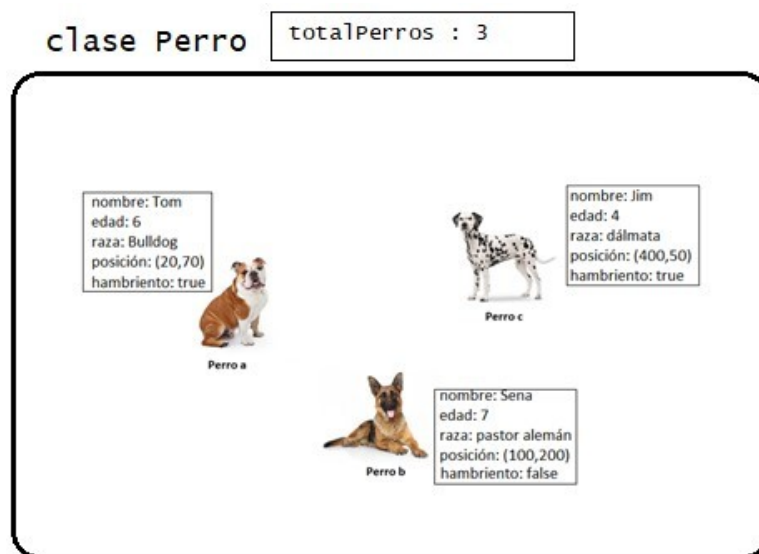
Test:

- Crea un Oficina[] y rellénalo con una oficina de cada tipo usando los métodos estáticos. Comprueba, mediante un bucle, que cada oficina tiene la cantidad de empleados que indica el índice del bucle.

11.- Propiedades estáticas

Igual que los objetos incluyen propiedades y métodos de instancia, las clases también tienen propiedades y métodos estáticos. Las propiedades estáticas simplemente son variables que incorpora la clase, y que no están asociadas a ningún objeto.

Por ejemplo, la clase Perro podría tener un contador llamado “totalPerros” que nos lleve la cuenta de cuántos objetos Perro se han creado. Ese contador sería una propiedad de la clase, y es diferente de las propiedades que tienen dentro los objetos.



En la documentación de la clase nos encontraríamos algo así:

Perro
<ul style="list-style-type: none">- String nombre- int edad- String raza- Point posición- boolean hambriento- static int totalPerros
<ul style="list-style-type: none">+ Perro(String n, int e, String r)+ Perro(String n, int e, String r, Position p, boolean h)+ static int getTotalPerros()

Para programar la propiedad estática, simplemente la escribimos tal y como se describen el resto de las propiedades, solo que incluiremos la palabra “static” al principio:

```
1. class Perro{
2.     private String nombre;
3.     private int edad;
4.     private String raza;
5.     private Point posicion;
6.     private boolean hambriento;
7.     private static int totalPerros;
8.     // resto de la clase
9. }
```

A la hora de dar el valor inicial de la propiedad estática, tenemos dos opciones:

1. Asignar el valor a la propiedad directamente en su declaración: Esta es la opción más habitual, y se usa siempre que queramos que la propiedad estática tenga un valor inicial que no implique usar código fuente para obtenerlo. Por ejemplo, podemos dar un 0 al valor inicial de la propiedad `totalPerros`:

```
1. class Perro{
2.     private String nombre;
3.     private int edad;
4.     private String raza;
5.     private Point posicion;
6.     private boolean hambriento;
7.     private static int totalPerros = 0;
8.     // resto de la clase
9. }
```

2. Usar un bloque inicializador estático: Es un bloque de código que comienza con la palabra **static**, y en él se escriben las líneas que den el valor inicial a dichas propiedades. Este bloque solo es ejecutado la primera vez que se utilice la clase (crear un objeto, llamar a un método, etc). Esta opción, menos habitual, se usa cuando el valor inicial de la propiedad estática requiere código fuente elaborado para obtenerlo.

Un detalle muy importante es que para acceder a las propiedades estáticas no se utiliza `this`, sino el nombre de la clase. Esto es así porque las propiedades estáticas pertenecen a la clase y no a los objetos.

Por ejemplo, si queremos usar un bloque inicializador estático para dar el valor 0 a la propiedad `totalPerros`, haríamos esto:

```
1. class Perro{
2.     private String nombre;
3.     private int edad;
4.     private String raza;
5.     private Point posicion;
6.     private boolean hambriento;
7.     private static int totalPerros;
8.     static{
9.         Perro.totalPerros = 0;
10.    }
11.    // resto de la clase
12. }
```

Observa como para acceder a la propiedad estática no se ha usado `this.totalPerros`, sino **`Perro.totalPerros`**. Como hemos dicho, se debe a que dicha propiedad no pertenece a ningún objeto concreto sino a la clase.

Debido a que las propiedades estáticas pertenecen a la clase y no a los objetos, muchos autores las consideran “valores comunes compartidos por todos los objetos de la clase” y las ven como si fuesen propiedades de los objetos, cuyo valor es compartido por todos a la vez. Por tanto, si cambia el valor de una propiedad estática, todos los objetos verán que dicho valor ha cambiado. Esto no ocurre con las propiedades de instancia, que son variables ligadas a cada objeto, e independientes unos de otros.

Por último, faltaría programar el método estático `getTotalPerros`, que simplemente es un getter que devuelve el valor de la propiedad estática `totalPerros`. No hay ningún problema a la hora de programar dicho método, ya que al ser estático, puede acceder a `Perro.totalPerros`.

Ejercicio 28: Crea la siguiente clase, que representa la matrícula de un alumno en una asignatura:

Matricula
+ static int siguienteNúmeroMatricula - int númeroMatricula - String nombreAlumno - String nombreAsignatura
+ Matricula(String nombreAlumno, String nombreAsignatura) + String getNombreAlumno() + String getNombreAsignatura() + int getNúmeroMatricula()

- La propiedad estática guarda el siguiente número que se usará para matricular a un alumno. El primer número de matrícula disponible será el 1
- El constructor crea una matrícula para un alumno en la asignatura indicada. Su número de matrícula será el que indique la propiedad estática “siguienteNúmeroMatricula”, que deberá incrementarse una vez asignado dicho número a una matrícula concreta.
- Los getters devuelven los valores de las propiedades.

Test:

- Crea tres objetos matrícula y comprueba que la propiedad siguienteNumeroMatricula es 4.

Mucho cuidado al hacer tests en los que intervienen propiedades estáticas, ya que cada test modifica el valor de la propiedad estática y eso influye en los siguientes tests. Para evitar el problema, se mete en la clase de los test un método anotado con **@Before**, donde se pone código para reiniciar el valor de las propiedades estáticas.

```
1. @Before // este método se llama antes de realizar cada test
2. public void reestablecerNumeroMatricula(){
3.     Matricula.siguienteNumeroMatricula = 0;
4. }
5. @Test
6. public void testMatricula(){
7.     Matricula m = new Matricula("Antonio","Programación");
8.     assertEquals(2,m.siguienteNumeroMatricula);
9. }
```

Ejercicio 29: Crea la siguiente clase, que representa una Bola de Dragón. La clase está diseñada para que solo se puedan crear en la memoria RAM un máximo de 7 bolas. Al intentar crear la octava, se lanzará una excepción.

BolaDragón
- static final int MAXIMO_BOLAS - static int siguienteBola - int número
- BolaDragón(int número) + int getNúmero() + static BolaDragón crearBolaDragón() throws Exception



- La propiedad estática MAXIMO_BOLAS indica la cantidad máxima de bolas que se pueden crear. Se inicializará a 7. La palabra “final” que la acompaña nos indica que esa propiedad se convierte en constante, y por tanto, no se podrá modificar.
- La propiedad estática siguienteBola es el número de la siguiente bola de dragón que se

- generará. Por defecto, valdrá 1.
- La propiedad de instancia número es el número que tiene la bola de dragón
- getNúmero: devuelve el número de la bola de dragón.
- El método estático crearBolaDragón hace lo siguiente:
 - Si se ha alcanzado la cantidad máxima de bolas creadas, el método lanzará una excepción con el mensaje “Ya se han creado 7 bolas de dragón”.
 - En caso contrario, se creará una nueva bola de dragón con el número que indique la propiedad “siguienteBola” y se devolverá. Además, la propiedad “siguienteBola” se incrementará.

Observa cómo ocultando el constructor y usando propiedades y métodos estáticos podemos controlar la cantidad de objetos que se pueden crear en la memoria RAM.

Test:

- Usa un bucle para crear 7 bolas de dragón. Una vez creadas, comprueba con otro bucle que el número de cada bola es igual al contador del bucle.
- Usa un bucle para crear 8 bolas de dragón y comprueba que se lanza una excepción.

Ejercicio 30: Programa la siguiente clase, que es un altavoz:

Altavoz
+ static final int VOL_MAX + static final int VOL_MIN - int volumen
+ Altavoz() + void ponerVolumenMaximo() + void setVolumen(int v) + int getVolumen() + String toString()

- Las constantes VOL_MIN y VOL_MAX indican los valores mínimo y máximo que puede tener cualquier altavoz. Estos valores se deberán inicializar a 0 y 255 respectivamente.
- La propiedad de instancia volumen indica el volumen del altavoz.
- El constructor crea un altavoz apagado (su volumen es 0)
- El método ponerVolumenMaximo pone el altavoz al máximo de su volumen
- El método setVolumen pone el altavoz al nivel pasado como parámetro. Si dicho volumen es incorrecto, lanzará una IllegalArgumentException.
- El método getVolumen devuelve el volumen del altavoz.
- El método toString devuelve un String con este formato: entre corchetes se ve el volumen, y luego se ve una barra formada por un total de 10 asteriscos y guiones que indican de forma gráfica el tanto por ciento del volumen del altavoz sobre su máximo.
Ejemplo: [127] *****-----

Test:

- Crea un Altavoz y comprueba que su volumen es la constante VOL_MIN. Llama a su método ponerVolumenMaximo y comprueba que el volumen coincide con la constante VOL_MAX
- Crea un Altavoz y ponlo a volumen -28. Comprueba que se lanza una IllegalArgumentException

- Crea un Altavoz y ponle volumen 127. Comprueba que el método toString nos devuelve este String:

[127] *****-----

Pon el volumen máximo y comprueba que toString tiene 10 asteriscos.

Ejercicio 31: Programa la siguiente clase, que es un equipo de música que tiene varios altavoces:

EquipoMusica
- Altavoz[] altavoces
+ EquipoMusica(int numeroAltavoces)
+ Altavoz getAltavoz(int posición)
+ void setVolumen(int numeroAltavoz, int volumen)

- La propiedad “altavoces” es un array en el que se guardarán los altavoces del equipo
- El constructor inicializa la propiedad “altavoces”, creando un array con la cantidad de altavoces pasada como parámetro, y rellenándola con objetos altavoz.
- getAltavoz: devuelve el altavoz que se encuentra en la posición indicada, dentro del array de altavoces.
- setVolumen: cambia el volumen del altavoz cuya posición se pasa como parámetro.

Test:

- Crea un equipo de música con 5 altavoces. Comprueba que volumen de todos ellos es 0.
- Crea un equipo de música con 10 altavoces y pon a los pares su volumen máximo y a los impares volumen 50. Comprueba que dichos volúmenes se han asignado correctamente.

Ejercicio 32: Programa la siguiente clase Bola, que representa una bola que tiene un número:

Bola
- int número
+ Bola(int número)
+ int getNúmero()

A continuación, programa la siguiente clase llamada Bombo, que tiene una cola (consultar el pdf de Java Collection Framework para recordar lo que es una cola y la documentación de la interfaz Queue<T>) con todas las bolas que irán saliendo del bombo:

Bombo
- Queue<Bola> bolas
+ Bombo(int totalBolas)
+ int getNúmeroBolas()
+ Bola sacarBola()

- El constructor creará una ArrayDeque<Bola> para inicializar la propiedad “bolas”. A continuación, le añadirá la cantidad de bolas indicada como parámetro. Cada bola tendrá un número aleatorio entre 1 y 100. No importa que se repitan los números de las bolas.
- El método getNúmeroBolas devolverá la cantidad de bolas que hay en el bombo.
- El método sacarBola devuelve un objeto Bola con la siguiente bola de la cola, o **null** si no queda

ninguna bola en el bombo.

Test:

- Crea un bombo con 100 bolas y comprueba que su método `getNumeroBolas` devuelve 100. A continuación, saca las 100 bolas y comprueba que al terminar el número de bolas del bombo es 0.

Ejercicio 33: Programa la siguiente clase, que es un objeto que sirve para registrar las notas que vas sacando en tus exámenes y poder hacer algunas estadísticas:

Notas
- List<Double> notas
+ Notas() + void añadirNota(double n) + int getTotalNotas() + double calcularNotaMedia() + double calcularNotaMaxima()

- El constructor inicializa la propiedad “notas” con un `ArrayList<Double>`
- `añadirNota`: Añade a la lista de notas la nota que se pasa como parámetro.
- `getTotalNotas`: Devuelve la cantidad de notas que se pasa como parámetro.
- `calcularNotaMedia`: Devuelve un `double` relleno con la nota media de todas las notas que hay en la lista. En caso de que la lista esté vacía se lanzará una **`IllegalStateException`** con el mensaje “no hay notas para calcular la media”
- `calcularNotaMáxima`: Devuelve un `double` relleno con la nota máxima de todas las notas que hay en la lista. En caso de que la lista esté vacía se lanzará una **`IllegalStateException`** con el mensaje “no hay notas para calcular la nota máxima”

Test:

- Crea un objeto `notas` y rellénalo con las notas 8,5,9,1,6. Comprueba que la nota máxima es 9 y la media es 5.8, con un margen de error de 0.01
- Crea un objeto `notas` vacío y llama a `calcularNotaMáxima`. Comprueba que se lanza una **`IllegalStateException`**

Ejercicio 34: Programa la siguiente clase, que es un reloj:

Reloj
- LocalTime hora
+ Reloj(LocalTime h) + Reloj() + LocalTime getHora() + void añadir(int segundos) + boolean esNoche() + void esperar(int segundos) + String toString()

- El primer constructor crea un reloj que guarda la hora pasada como parámetro
- El segundo constructor crea un reloj que guarda la hora del momento actual.

- El método getHora nos devuelve la hora del reloj.
- El método añadir sirve para añadir la cantidad de segundos indicada a la hora del reloj.
- El método esNoche devuelve true si la hora está entre las 20:00 y las 8:00
- El método esperar hace una pausa de la cantidad de segundos pasada como parámetro y después actualizará la propiedad "hora"
- toString: devuelve la hora escrita en el siguiente formato: hora:minutos:segundos

Test:

- Crea un reloj que refleje las 16:00 y añádele 1 hora. Comprueba que la hora del reloj son las 17:00
- Crea un reloj que refleje las 23:59 y añádele 60 segundos. Comprueba que la hora del reloj son las 00:00
- Crea un reloj que refleje las 19:30 y comprueba que no es de noche. Añádele una hora y comprueba que ya es de noche.
- Crea un reloj que refleje las 7:30 y comprueba que es de noche. Añádele una hora y comprueba que ya no es de noche
- Crea un reloj que refleje las 15:28 y comprueba que toString devuelve 15:28:00

Ejercicio 35 : Programa la siguiente clase, que es un bolígrafo que puede escribir en laCapaTexto de la Consola DAW:

Bolígrafo
- static final Color[] COLORES
- int color
- CapaTexto ct
+ Bolígrafo(CapaTexto ct)
+ void setColor(int n)
+ void escribir(String texto)
+ void escribirGuay(String texto)

- La lista COLORES guarda los colores con los que puede escribir el bolígrafo. Estos colores son (en este orden): rojo, verde, azul, blanco, amarillo.
- El constructor crea un bolígrafo para escribir en la capa de texto que se pasa como parámetro. La propiedad "color" es la posición del color con el que se va a dibujar, dentro la lista COLORES. Inicialmente estará activado el color rojo.
- El método setColor recibe como parámetro el número de posición del color con el que se va a escribir el texto.
- El método escribir recibe un String y lo escribe en la capa de texto con el color activo
- El método escribirGuay recibe un String y lo escribe en la capa de texto, usando un color diferente de la lista de colores para cada letra. Los colores deberán rotarse.

Test: *Debido a que la clase muestra continuamente salida gráfica, no es apropiado hacer tests de JUnit. En esta ocasión el mejor test es hacer este programa de pruebas:*

- Haz un programa que cree una Consola DAW y pregunte al usuario un texto y el número de posición de un color (entre 1 y 5). El programa mostrará el texto usando el color cuya posición ha indicado el usuario. A continuación, lo mostrará con el método escribirGuay.

Ejercicio 36 : Programa la siguiente clase, que representa un examen realizado por un alumno

Examen
- String nombre
- Map<String,String> respuestas
+ Examen(String nombre)
+ Examen(Examen c)
+ void responder(String pregunta, String respuesta)
+ String getRespuesta(String pregunta)
+ double calificar(Map<String,String> respuestasCorrectas)

- nombre: Guarda el nombre del alumno
- respuestas: Es un map que asocia las preguntas del examen a las respuestas del alumno
- primer constructor: Crea un examen sin respuestas, para el alumno cuyo nombre se pasa como parámetro.
- Segundo constructor: Crea un examen cuyo nombre y respuestas son iguales a las del examen pasado como parámetro.

Este segundo constructor, cuya finalidad es producir un duplicado exacto de un objeto, se denomina constructor de copia, y tiene su origen en el lenguaje C++

- responder: Asocia la pregunta a la respuesta pasada como parámetro
- getRespuesta: Devuelve la respuesta de una pregunta
- calificar: Recibe un map con las preguntas y respuestas correctas, y mira cuántas de ellas coinciden con las del map de la propiedad "nombre". La nota del alumno se calcula dividiendo la cantidad de preguntas acertadas entre el total, y multiplicando por 10.

Test:

- Crea un examen para el alumno "Jose" y haz que responda a estas preguntas con las respuestas indicadas:

¿cuánto es 2+2?	24
¿cuál es el planeta más cercano al Sol?	Mercurio
¿cuál es la capital de España?	Madrid
¿cuántos kilos son 2500 gramos	25000

Por último, corrige el examen usando esta plantilla y comprueba que su nota es un 5

¿cuánto es 2+2?	4
¿cuál es el planeta más cercano al Sol?	Mercurio
¿cuál es la capital de España?	Madrid
¿cuántos kilos son 2500 gramos	2.5

12.- Interfaces

En el tema 2 vimos que los tipos referencia eran las **clases** y las **interfaces**. Hasta ahora hemos aprendido a programar clases y para terminar el tema vamos a ver qué son en realidad cómo se programan las interfaces.

Para un programador de aplicaciones una interfaz era algo muy parecido a una clase, ya que era un tipo de dato referencia que admitía la herencia múltiple. Sin embargo, para un programador de clases una interfaz va a ser algo diferente:

Para un programador de clases una interfaz es un conjunto de métodos que representan una “habilidad” que puede tener una clase.

Por ejemplo, una habilidad puede ser “Nadador”. Para que una clase sea considerada un “nadador” bastará con que tenga estos métodos:

<<interface>> Nadador
+ void nadar() + void sumergirse(int profundidad)

La interfaz solamente describe cuáles son los métodos que tiene que tener una clase para que tenga dicha habilidad. Las clases deberán programar todos los métodos indicados en la interfaz para adquirir la habilidad. Por ejemplo, las siguientes clases son nadadores, porque tienen los métodos que aparecen en la interfaz Nadador:

Persona
-String nombre -DNI dni # double sueldo -LocalDate fechaNacimiento + Persona(String n, DNI d, double s, LocalDate fn) + int getSuelo() + void nadar() + void sumergirse(int profundidad) + String getNombre() + void saltar()

Delfín
-Aletas aletas -int longitud + Delfin() + void nadar() + void sumergirse(int profundidad) + void saltar()

Pulpo
-Tentaculos[] tentáculos + Pulpo() + void nadar() + void sumergirse(int profundidad) + Tinta lanzarTinta()

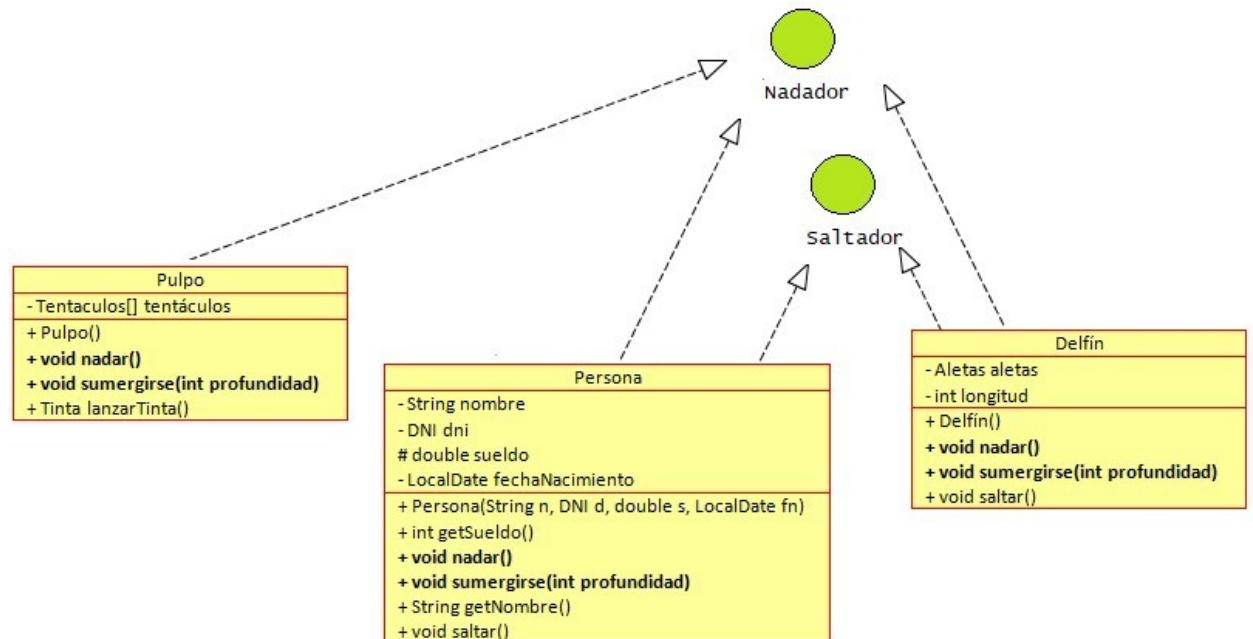
Una clase puede implementar todas las interfaces que necesite. Por ejemplo, supongamos que definimos esta interfaz llamada “Saltador”:

<<interface>> Saltador
+ void saltar()

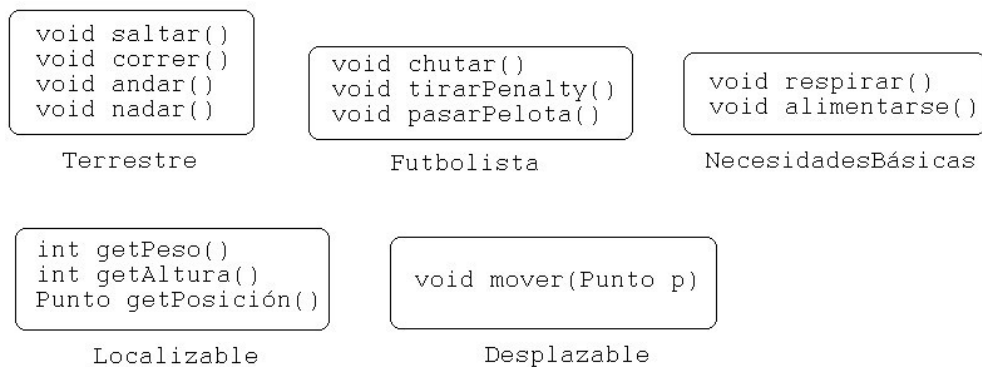
Entonces las clases Persona y Delfín también implementan la interfaz “Saltador”, porque ambas tienen el método saltar, tal y como lo describe la interfaz “Saltador”. En cambio, Pulpo implementa Nadador, pero no implementa Saltador.

Cuando una clase incorpora los métodos que aparecen en una interfaz, se dice que “la clase implementa la interfaz”

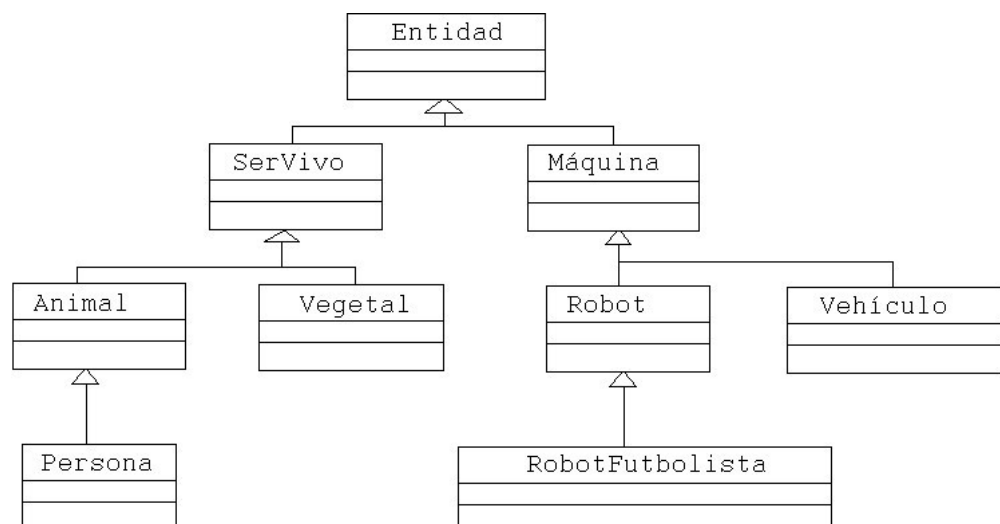
Hay varias formas de representar en un diagrama de clases que una clase implementa interfaces. Una de ellas consiste en representar cada interfaz con una bolita y dibujar una flecha discontinua desde la clase hasta la interfaz:



Ejercicio 37: Supongamos que tenemos las siguientes interfaces:



¿Qué interfaces implementarían las clases del siguiente diagrama?



12.1.- Programación de una interfaz

Programar una interfaz es muy sencillo. Para crear una en IntelliJ (en los demás IDE el proceso es muy similar), pulsamos **new** → **Java class** como si fuésemos a crear una clase, pero en el desplegable, elegimos **interface** (si nos equivocamos, no pasa nada porque luego en el código fuente podemos cambiar la palabra **class** por **interface** y estará solucionado) y escribimos el nombre de nuestra interfaz:



Una vez que el IDE nos abre el código fuente con la cabecera de la interfaz, nos vamos a su interior y lo único que hay que hacer es escribir los métodos de la interfaz, **sin programarlos**.

Por ejemplo, la interfaz Nadador que hemos visto en el ejemplo se programaría así:

```
1. public interface Nadador{
2.     public void nadar();
3.     public void sumergirse(int profundidad);
4. }
```

Como vemos, no hay que hacer nada más con los métodos de la interfaz, salvo ponerlos. Este tipo de métodos, que se escriben pero no se programan, se denominan **métodos abstractos**. De hecho, se les puede marcar con la palabra **abstract**, aunque no es obligatorio.

```
1. public interface Nadador{
2.     public abstract void nadar();
3.     public abstract void sumergirse(int profundidad);
4. }
```

Por otra parte, en las interfaces todo es public, por lo que la palabra **public** también puede quitarse. O sea, los siguientes métodos, son “public” y “abstract” aunque no lo ponga:

```
1. public interface Nadador{
2.     void nadar();
3.     abstract void sumergirse(int profundidad);
4. }
```

12.2.- Implementación de una interfaz

Las clases que implementan una interfaz son clases que tienen los métodos que define la interfaz, y por tanto, los llevan programados en su interior. Sin embargo, para que una clase implemente “de forma oficial” una interfaz, no basta con escribir en la clase los métodos de la interfaz y programarlos.

Para que una clase implemente una interfaz, y Java sepa reconocerlo, hay que utilizar la palabra **implements** en la línea que da comienzo a la clase.

Por ejemplo, para indicar que la clase Pulpo implementa la interfaz Nadador, escribiremos:

```
1. public class Pulpo implements Nadador{
2.     @Override
3.     public void nadar(){
4.         System.out.println("El pulpo está nadando");
5.     }
6.     @Override
7.     public void sumergirse(int profundidad){
8.         System.out.println("El pulpo se sumerge "+profundidad+" metros");
9.     }
10. }
```

Como vemos, la clase Pulpo tiene programados en su interior los métodos que aparecen como “abstractos” en la interfaz. Por ese motivo, el IDE nos sugiere¹⁶ que pongamos encima de esos métodos la palabra **@Override**, que en este momento¹⁷ nos indica que ese método viene de la implementación de una interfaz.

Cuando una clase implementa varias interfaces, las pondremos separadas por comas, así:

```
1. public class Persona implements Nadador, Saltador{
2.     private String nombre;
3.     private int edad;
4.     ...
5.     @Override
6.     public void nadar(){
7.         System.out.println("La persona está nadando");
8.     }
9.     @Override
10.    public void sumergirse(int profundidad){
11.        System.out.println("La persona se sumerge "+profundidad+" metros");
12.    }
13.    @Override
14.    public void saltar(){
15.        System.out.println("La persona ha saltado");
16.    }
17. }
```

Como vemos, las interfaces no son más que tipos de datos que definen los métodos que luego, las clases que las implementan, deberán programar. Por ese motivo, muchos autores consideran las interfaces como un “contrato”, que incluye los métodos que las clases se comprometen a programar.

Las interfaces, al igual que las clases, son tipos de datos referencia. Como veremos en el próximo tema (y en el super ejercicio del final), las interfaces servirán **para almacenar, mediante abstracción, objetos que las implementen**, y así hacer a los programas independientes del tipo de los objetos utilizados, facilitando el mantenimiento y como también veremos, el trabajo en equipo.

Ejercicio 38: Programa la siguiente interfaz, que representa algo que tiene dinero:

<<interface>> Adinerado
+ double getDineroTotal() + boolean añadirDinero(int cantidad) + boolean retirarDinero(int cantidad)

¹⁶ En Java el uso de @Override no es obligatorio (en Kotlin sí lo es), pero se recomienda ponerlo porque así, si nos equivocamos al escribirlo, el IDE nos avisará y nos dirá que ese método no coincide con ninguno de la interfaz.

¹⁷ En el tema 5 veremos otro lugar diferente donde aparecerá la anotación @Override

- getDineroTotal devuelve la cantidad de dinero que tenga el objeto
- añadirDinero recibe una cantidad e incrementa la cantidad de dinero del objeto.
Devuelve false si no es posible añadir la cantidad de dinero al objeto.
- retirarDinero recibe una cantidad y decrementa la cantidad de dinero del objeto.
Devuelve false si el objeto no tiene dinero suficiente para retirar dicha cantidad.

Programa las siguientes clases, y haz que implementen la interfaz Adinerado.

Banco implements Adinerado
- double dinero
+ Banco()

- El constructor crea un Banco cuyo dinero almacenado es 0. Es posible ingresar cualquier cantidad de dinero en el banco.

Test:

- Crea un banco y añade 1000€. Comprueba que dicho método ha devuelto true y que hay 1000€ en el banco. A continuación, retira 200€ y comprueba que dicho método devuelve true y hay 800€ en el banco. Por último, retira 4000€ , comprueba que devuelve false y hay 800€.

Monedero implements Adinerado
- double dinero
+ Monedero()

- El constructor crea un Monedero cuyo dinero almacenado es 0, y solo admite almacenar hasta 1000 €.

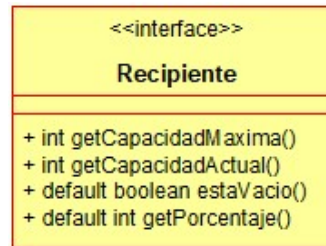
Test:

- Crea un monedero y añade 50€. Comprueba que dicho método ha devuelto true y que hay 50€ en el monedero. A continuación, retira 30€ y comprueba que dicho método devuelve true y hay 20€ en el monedero. Por último, retira 400€ y comprueba que dicho método devuelve false y que hay 20€ en el monedero.
- Crea un monedero y añade 50€. Comprueba que hay 50€ en el monedero. A continuación añade 2000€ y comprueba que el método devuelve false y que en el monedero hay 50€. Añade 950€ y comprueba que el método devuelve true y que en el monedero hay 1000€. Por último, añade 1€ y comprueba que el método devuelve false y que en el monedero hay 1000€.

12.3.- Métodos default

Desde el punto de vista tradicional de la programación orientada a objetos, las interfaces solo tienen métodos que deben ser programados en las clases hijas. Sin embargo, desde que en la industria del software se ha impuesto el uso de interfaces, se ha considerado interesante la posibilidad de que estas puedan incluir algunos métodos **ya programados**.

Un método default es un método de una interfaz que **puede programarse en ella, usando los demás métodos de la interfaz**. De esta forma las clases que implementen dicha interfaz lo recibirán ya programado. Aquí tenemos un ejemplo:



```

1. public interface Recipiente{
2.     int getCapacidadMaxima();
3.     int getCapacidadActual();
4.     default boolean estaVacio(){
5.         return this.getCapacidadActual()==0;
6.     }
7.     default int getPorcentaje(){
8.         int p=0;
9.         if(!this.estaVacio()){
10.            p= 100*this.getCapacidadActual()/this.getCapacidadMaxima();
11.        }
12.        return p;
13.    }
14. }

```

En este ejemplo podemos ver que la interfaz Recipiente define como abstractos los métodos getCapacidadMaxima y getCapacidadActual, que a la fuerza deberán programarse en las clases que implementen Recipiente. Sin embargo, la interfaz incorpora dos métodos default, que sirven para comprobar si el recipiente está vacío y calcular su porcentaje. De esta forma, las clases que implementen la interfaz los recibirán tal cual. Por supuesto, si lo desean pueden volver a programarlos, sobrescribiendo así el comportamiento por defecto definido en la interfaz.

En el caso de que una clase implemente dos interfaces y en las dos esté programado el mismo método default, se producirá un conflicto que se resuelve sobrescribiendo en la clase dicho método compartido.

Por último, indicamos que además de métodos default, las interfaces pueden tener propiedades y métodos **estáticos**, que funcionan igual que como hemos visto para las clases. Aquí tenemos un ejemplo sencillo de una interfaz con propiedades estáticas y un método estático.

```

1. public interface Recipiente{
2.     public static final int CERO = 0;
3.     public static final int CIEN = 100;
4.     int getCapacidadMaxima();
5.     int getCapacidadActual();
6.     default boolean estaVacio(){
7.         return this.getCapacidadActual()==Recipiente.CERO;
8.     }
9.     default int getPorcentaje(){
10.        int p=Recipiente.CERO;
11.        if(!this.estaVacio()){
12.            p= Recipiente.CIEN*this.getCapacidadActual()/this.getCapacidadMaxima();
13.        }
14.        return p;
15.    }
16.     public static void mostrarInformacion(Recipiente r){
17.         System.out.println("Capacidad actual: "+r.getCapacidadActual());
18.         System.out.println("Capacidad máxima: "+r.getCapacidadMaxima());
19.         System.out.println("Porcentaje de ocupación: "+r.getPorcentaje()+"%");
20.         System.out.println("Vacío: "+r.estaVacio()+"si":"no");
21.     }
22. }

```

Al igual que en las clases, en los métodos estáticos de las interfaces no podemos usar **this**.

12.4.- Métodos privados en interfaces

Los métodos default fueron introducidos en **Java 8** para poder hacer que la interfaz tuviese métodos de instancia programados en su interior, pero no son la única opción para esto. En **Java 9** apareció la posibilidad de incluir métodos privados en las interfaces. Estos métodos son similares a los métodos default, pero no pueden ser usados fuera de la interfaz. Además, no necesitan de la palabra “default”.

Ejemplo: Vamos a repetir el método `getPorcentaje` del ejemplo anterior, pero haciéndolo privado:

```
1. public interface Recipiente{
2.     int getCapacidadMaxima();
3.     int getCapacidadActual();
4.     default boolean estaVacio(){
5.         return this.getPorcentaje()==0;
6.     }
7.     private int getPorcentaje(){
8.         int p=0;
9.         if(!this.estaVacio()){
10.            p= 100*this.getCapacidadActual()/this.getCapacidadMaxima();
11.        }
12.        return p;
13.    }
14. }
```

En esta situación, el método `getPorcentaje` solo podría ser usado dentro de la interfaz `Recipiente`, como método auxiliar para programar otros métodos (como hacemos en la línea 5 para calcular, de una forma muy rebuscada, si un depósito está vacío).

Ejercicio 39 : Añade los siguientes miembros a la interfaz **Adinerado**:

<<interface>> Adinerado
+ static int TRANSFERENCIA_MINIMA=1526
+ double getDineroTotal() + boolean añadirDinero(int cantidad) + boolean retirarDinero(int cantidad) + default boolean tieneDinero() + default boolean transferirHacia(Adinerado receptor, double cantidad) + default boolean transferirDesde(Adinerado emisor, double cantidad) + static boolean transferir(Adinerado emisor, Adinerado receptor, double cantidad)

- `tieneDinero` devuelve true si el adinerado tiene dinero.
- `transferirHacia` ingresa en el “receptor” la cantidad de dinero que se retira del objeto `Adinerado` que se está programando. El método devuelve false si no hay dinero suficiente para la transferencia o dicho dinero no llega a la transferencia mínima
- `transferirDesde` ingresa en el objeto `Adinerado` que se está programando la cantidad de dinero que se extrae del “emisor”. El método devuelve false igual que el anterior.
- `transferir` ingresa en el objeto “emisor” la cantidad de dinero que se extrae del objeto “receptor”. El método devuelve false igual que el anterior.

Test:

- Crea un Banco y comprueba que `tieneDinero` devuelve false
- Crea dos Bancos, añádeles 2000€ y 5000€ respectivamente y transfiere con 2500€ del primero al segundo. Comprueba que los saldos son 4500€ y 7500€ respectivamente
- Repite el test anterior usando otro método diferente para hacer la transferencia.

- Crea dos Bancos, añádeles 2000€ y 5000€ respectivamente y transfiere 500€ del primero al segundo. Comprueba que el método que realiza la transferencia devuelve false y que los saldos siguen siendo 1500€ y 5500€ respectivamente

13.- Records

En los primeros 20 años del siglo XXI Java ha sido el lenguaje predominante en el mundo empresarial y no ha tenido ninguna competencia. Esto ha hecho que algunas de las cosas que surgieron desde sus primeros años se hayan mantenido hasta la actualidad. Por ejemplo, el uso de getters y setters tal y como los hemos estudiado aquí¹⁸:

```

1. public class Alumno{
2.     private int numeroMatricula;
3.     private String nombre;
4.     private String direccion;
5.     private boolean repetidor;
6.     public Alumno(int nm, String n, String d, boolean r){
7.         this.numeroMatricula=nm;
8.         this.nombre=n;
9.         this.direccion=d;
10.        this.repetidor=r;
11.    }
12.    public int getNumeroMatricula(){
13.        return this.numeroMatricula;
14.    }
15.    public String getNombre(){
16.        return this.nombre;
17.    }
18.    public String getDireccion(){
19.        return this.direccion;
20.    }
21.    public boolean esRepetidor(){
22.        return this.repetidor;
23.    }
24. }
```

Una de las críticas que se han hecho a Java durante todos estos años está precisamente en la necesidad de escribir tantas líneas para proporcionar métodos que accedan o cambien el valor de una propiedad. Aunque es cierto que el IDE nos facilita la labor, muchos programadores odian tener que escribir constantemente getters y setters.

En los últimos años, donde se ha establecido una feroz competencia entre los lenguajes de programación, han aparecido las **data classes**, que son clases simples cuyo único objetivo es ser portadores de datos y poco más. En Java, dicho concepto se ha adaptado con el nombre de **record**.

Un **record** es una forma abreviada de escribir una clase cuyo único objetivo es portar una serie de datos de solo lectura (no hay setters). Cuando hacemos un record, Java automáticamente proporciona:

¹⁸ El nombre de “getters y setters” tal y como los conocemos surgió con una especificación llamada **JavaBeans**, que eran unas normas de nomenclatura para los nombres de las clases (entre ellas, los getters/setters) que cuando se seguían al pie de la letra, ayudaban al IDE y a otras herramientas a facilitar el trabajo. La especificación JavaBeans no tuvo mucho éxito, pero contribuyó a popularizar el uso de los getters y setters en todo tipo de clases (aunque no siguieran la especificación JavaBeans). En la actualidad, Java está renegando de su uso y poco a poco intenta desmarcarse de esos nombres. Las nuevas mejoras que introducen prescinden de la terminología getter/setter, como puede apreciarse en los records.

- Una implementación **private final** de las propiedades. Por tanto, las propiedades no se pueden modificar una vez asignado su valor en el constructor.
- Un **constructor principal** (en la terminología oficial de Java se denomina **compact constructor**) que asigna directamente los valores en las propiedades (más adelante veremos cómo hacer para lanzar excepciones si los valores son incorrectos)
- Un getter para cada propiedad. Es importante destacar que el nombre de los getters no va a empezar por get, sino que se llamará exactamente igual que la propiedad¹⁹.

Por ejemplo, si una propiedad se denomina **edad**, lo que hasta ahora hemos llamado **getEdad()** en los records se llamará **edad()**

- Un método `toString` que muestra en pantalla el valor de las propiedades (y no la cadena `clase@hashCode` habitual)
- Implementaciones de **equals** y **hashCode** que nos permiten detectar si dos objetos son iguales basándonos en el valor de sus propiedades (en el tema 5 profundizaremos en esto).

La forma de programar un record es muy sencilla. Por ejemplo, la clase `Alumno` se haría así usando un record:

```
1. public record Alumno(int numeroMatricula, String nombre,int edad, String direccion, boolean repetidor){
2. }
```

Como podemos ver, las propiedades aparecen definidas en el mismo lugar de la clase, imitando así a lenguajes como Python y Kotlin. En un programa, el uso de un record es exactamente igual al de una clase:

```
1. Alumno a = new Alumno(35,"Antonio Pérez",15, "C/Marcianitos 43 2ºZ",true);
2. System.out.println(a.nombre()); // muestra Antonio Pérez
3. System.out.println(a.repetidor()); // muestra true
```

Cuando usamos el constructor principal, por defecto los datos que le pasamos son asignados a las propiedades sin realizar ningún tipo de comprobación. Como sabemos, esto es un problema porque entonces las propiedades podrían tomar valores incorrectos. Por ejemplo, podríamos crear un alumno con -20 años.

Para evitar este problema, Java nos permite añadir un bloque de código adicional, llamado **extensión del constructor principal**, que se ejecuta justo después del constructor principal. Allí podremos validar que las propiedades tomen un valor correcto y en caso contrario, lanzar alguna excepción:

¹⁹ Java vuelve aquí a sus orígenes, ya que en la primera versión de Java se hacía así. Recordemos que la terminología getter/setter surgió más adelante, con la especificación JavaBean. En el futuro la palabra “getter” se va a sustituir por **accesor** y “setter” por **mutator**.

```

1. public record Alumno(int numeroMatricula, String nombre, int edad, String direccion, boolean repetidor){
2.     public Alumno{
3.         if(edad<0) {
4.             throw new IllegalArgumentException("La edad no puede ser negativa");
5.         }
6.     }
7. }

```

En el ejemplo anterior, el bloque **public Alumno{ ... }** sirve solo para comprobar que la edad es positiva y no podemos modificar las propiedades dentro de él. Esto se debe a que Java ya ha llamado al constructor principal y las propiedades (que son de solo lectura) ya están rellenas cuando se ejecuta dicho bloque de código.

En un record podemos añadir más constructores (llamados **constructores secundarios**), que **obligatoriamente** deberán usar **this** para llamar al constructor principal.

```

1. public record Alumno(int numeroMatricula, String nombre, int edad, String direccion, boolean repetidor){
2.     // crea un alumno no repetidor con los parámetros recibidos
3.     public Alumno(int nm, String n, int e, String d){
4.         this(nm,n,e,d,false);
5.     }
6. }
7.

```

Una vez estudiado el constructor de los records, el resto de posibilidades que nos ofrecen son similares a las clases, teniendo siempre en cuenta que las propiedades son de solo lectura. Por ejemplo, en un record podemos añadir métodos, de la forma habitual:

```

1. public record Alumno(int numeroMatricula, String nombre, int edad, String direccion, boolean repetidor){
2.     public boolean mayorEdad(){
3.         return edad>=18;
4.     }
5.     public boolean menorEdad(){
6.         return !this.mayorEdad();
7.     }
8. }

```

Por último, señalamos que un Record se comporta igual que las clases en muchas cosas. Por ejemplo, los records pueden implementar interfaces y también pueden tener propiedades y métodos estáticos.

13.1.- Inmutabilidad y records

Como curiosidad, podríamos preguntarnos por qué los records no tienen setters. La explicación está en las nuevas tendencias de programación, que apuntan a la **programación funcional**. Para que este estilo de programación sea óptimo deben utilizarse **objetos inmutables**, que son aquellos cuyas propiedades son de solo lectura, y no cambian una vez que el objeto se ha creado. O sea, si queremos modificar alguna característica de ellos, tenemos que obtener un objeto diferente con la modificación que queramos, pero no podemos cambiar (o mutar) el objeto original.

Durante el curso hemos trabajado con algunas clases inmutables. Por ejemplo **String** es inmutable porque una vez creado un String no podemos cambiar de ninguna forma su contenido. Si queremos, por ejemplo, pasarlo a mayúsculas, su método **toUpperCase** nos dará otro String diferente con la versión en mayúsculas del original. Lo mismo sucede por ejemplo, con **LocalDate** o **LocalTime**.

En cambio, el **DepositoAgua** es mutable, porque cuando le añadimos o retiramos un litro,

cambia el valor de sus propiedades para adaptarse a la nueva situación. El **ArrayList** es otro ejemplo de clase mutable porque cuando le añadimos o retiramos objetos, cambia su estado interno.

Por tanto, pensando en la futura proliferación de los métodos funcionales en Java, los records han sido implementados para favorecer la programación de objetos inmutables. Por ejemplo, el Alumno definido en el ejemplo anterior es inmutable, porque una vez establecidas sus propiedades, ya no se pueden modificar.

Sin embargo, podemos hacer algunos objetos mutables con records, como ocurre en el siguiente ejemplo (podemos añadir o quitar notas a la lista de notas del alumno, y por tanto, un objeto Alumno sería mutable).

```
1. public record Alumno(String nombre, ArrayList<Integer> notas){
2. }
```

Ejercicio 40: Programa el siguiente record, que es la versión inmutable de la clase DepositoAgua:

<<record>> DepositoAguaInmutable	
+ DepositoAguaInmutable(int capacidadActual, int capacidadMaxima) + DepositoAguaInmutable(int capacidadMaxima) + DepositoAguaInmutable añadirLitro() + DepositoAguaInmutable retirarLitro() + int getPorcentaje()	

- El constructor principal declara las propiedades y les asigna el valor solamente si son positivas. En caso contrario, lanza una `IllegalArgumentException`. También se lanzará esa excepción si la capacidad actual es mayor que la máxima.
- El segundo constructor crea un depósito vacío con la capacidad máxima pasada como parámetro. Si esta es negativa se lanza una `IllegalArgumentException`
- `añadirLitro` devuelve un depósito de agua que tiene las mismas características del que estamos programando, pero con un litro más de agua
- `retirarLitro` devuelve un depósito de agua que tiene las mismas características del que estamos programando, pero con un litro más de agua
- `getPorcentaje` devuelve el porcentaje del depósito que está lleno

Tests:

- Crea un depósito de agua con -100 litros de capacidad máxima y 50 actuales, y comprueba que se lanza una `IllegalArgumentException`
- Crea un depósito de agua con 200 litros de capacidad máxima y 300 actuales, y comprueba que se lanza una `IllegalArgumentException`
- Crea un depósito de agua de 500 litros de capacidad máxima y 20 actuales, y comprueba que su porcentaje es 4
- Crea un depósito de agua de 500 litros de capacidad máxima y 400 actuales. Añade un litro y comprueba que el objeto original sigue teniendo 400 litros de capacidad actual, y el que ha devuelto el método de añadir tiene 401.

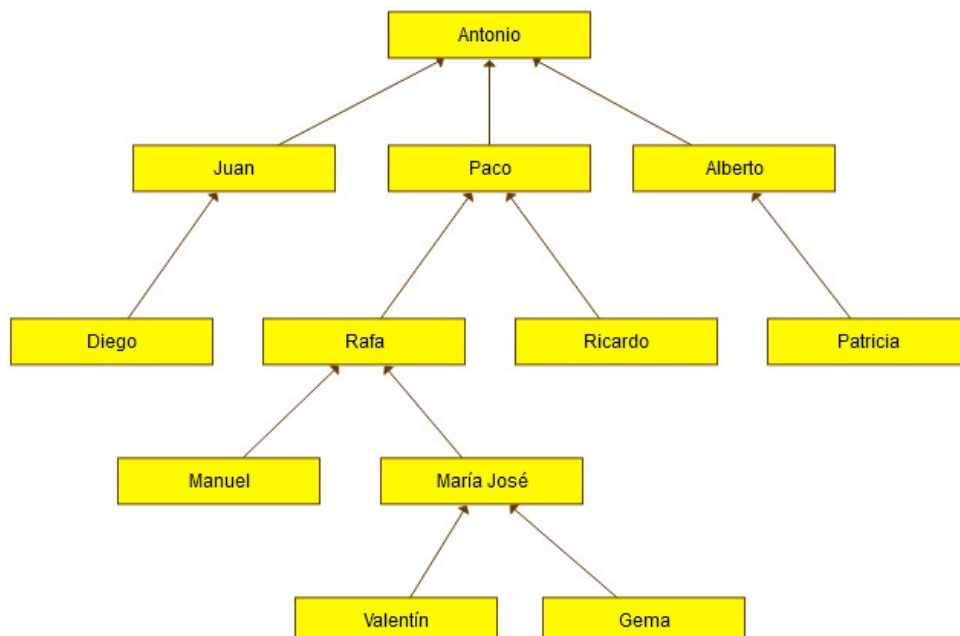
Ejercicio 41: Programa el siguiente record, que representa un empleado de una empresa que tiene

varios empleados a su cargo:

<<record>> Empleado
+ Empleado(String nombre, List<Empleado> subordinados) + Empleado(String nombre) + void mostrarSubordinados() - void mostrarSubordinados(int tabs)

- El constructor principal declara e inicializa las propiedades sin más.
- El segundo constructor crea un empleado que no tiene subordinados (su lista de subordinados es un ArrayList vacío).
- El primer método mostrarSubordinados muestra en pantalla el nombre del empleado y llama al segundo método mostrarSubordinados pasando un 1 como parámetro
- El segundo método mostrarSubordinados hace esto:
 - Recorre la lista de subordinados
 - Muestra el nombre de cada empleado subordinado, pero antes imprime una secuencia de tantos guiones consecutivos (-) como indica el parámetro "tabs"
 - A continuación llama al método mostrarSubordinados del empleado que está siendo recorrido, pero añadiendo 1 a la cantidad de tabs (eso se hace para que al mostrar en pantalla la salida parezca un árbol)

Test: Puesto que el método mostrarSubordinados muestra salida por pantalla, en esta ocasión el test será un programa de prueba que cree la siguiente estructura de objetos Empleado y posteriormente llame a mostrarSubordinados:



Si lo has hecho bien, el programa mostrará en pantalla esto:

```
Antonio
-Juan
--Diego
-Paco
--Rafa
---Manuel
---María José
----Valentín
----Gema
--Ricardo
-Alberto
--Patricia
```

14.- Enumeraciones

Una enumeración (**enum**) es un tipo de dato referencia²⁰ que solo puede tomar una serie de valores predefinidos.

Por ejemplo, supongamos que estamos haciendo un programa y necesitamos un tipo de dato que represente un día de la semana. Podríamos hacer una clase, pero como solo hay 7 días de la semana, una enumeración es la solución ideal. La programaríamos así:

```
1. public enum DiaSemana{
2.     LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;
3. }
```

Como puede verse en el ejemplo, para hacer una enum:

- En lugar de **public class** escribimos **public enum** y el nombre de la enumeración.
- Ponemos con letras mayúsculas (no es obligatorio que sean mayúsculas, pero es la costumbre que sigue toda la comunidad de Java) el nombre de los valores que puede tomar.
- La lista de valores termina en punto y coma. En realidad, cuando la enum no tiene nada más que la lista de valores, el punto y coma es opcional y se puede quitar. No obstante, si luego añadimos cosas a la enum entonces el punto y coma ya si es obligatorio.

Ahora, en un programa podemos crear un objeto de la enum **DiaSemana** y asignarle cualquiera de los valores que incluye. Por ejemplo, para almacenar en una variable el sábado haríamos esto:

```
1. DiaSemana d = DiaSemana.SABADO;
```

Las enum pueden tener métodos. Por ejemplo, podemos añadir a nuestra enum del ejemplo un método que nos diga si un día es fin de semana.

```
1. public enum DiaSemana{
2.     LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;
3.     public boolean esFinde(){
4.         return this.equals(SABADO) || this.equals(DOMINGO);
5.     }
6. }
```

²⁰ Internamente, al compilar el proyecto las enumeraciones se convierten en clases

Otra forma de hacer ese método es comprobando si la posición que ocupa nuestra constante en la lista de valores de la enum es 5 o 6. Podemos acceder a dicho dato con el método de instancia **ordinal()**, que nos devuelve el número de orden de la constante de la enum representada por **this**.

```
1. public enum DiaSemana{
2.     LUNES,MARTES,MIERCOLES,JUEVES,VIERNES,SABADO,DOMINGO;
3.     public boolean esFinde(){
4.         return this.ordinal()>=5;
5.     }
6. }
```

Cuando necesitamos una lista con todas las constantes de la enum (por ejemplo, para recorrerlas o para encontrar alguna constante concreta), podemos obtenerla con el método estático **values()** de la enum. Por ejemplo, si queremos mostrar el nombre de todos los días de la semana podríamos hacer esto:

```
1. public static void main(String[] args) {
2.     DiaSemana[] dias = DiaSemana.values();
3.     for(DiaSemana i:dias){
4.         System.out.println(i);
5.     }
6. }
```

Por último, a modo de ejemplo citaremos que las enum tienen características avanzadas, como:

- Es posible añadir propiedades (variables de instancia) a las constantes de la enum, de forma que puedan ser inicializadas en el constructor.

```
1. public enum DiaSemana{
2.     // constantes de la enum
3.     LUNES(false),MARTES(false),MIERCOLES(false),
4.     JUEVES(false),VIERNES(false),SABADO(true),DOMINGO(true);
5.     // variables de instancia (propiedades) de la enum
6.     private boolean esFinde;
7.     // constructor de la enum (no puede ser público)
8.     private DiaSemana(boolean f){
9.         esFinde=f;
10.    }
11.    // otros métodos de la enum
12.    public boolean esFinde(){
13.        return esFinde;
14.    }
15. }
```

- Es posible añadir métodos abstractos a la enum, y dichos métodos deben ser programados en un bloque de código dentro de cada constante de la enum.

```
1. enum Operacion{
2.     SUMAR{
3.         @Override
4.         public int calcularResultado(int a, int b) {
5.             return a+b;
6.         }
7.     },RESTAR{
8.         @Override
9.         public int calcularResultado(int a, int b) {
10.            return a-b;
11.        }
12.    };
13.    // método abstracto (en la enum la palabra abstract si es obligatoria)
14.    public abstract int calcularResultado(int a, int b);
15. }
```

En un programa podríamos escribir: **int resultado = Operación.SUMAR.calcularResultado(3,4);**

Ejercicio 42: Programa la siguiente enum, que representa las posibles canastas que pueden anotarse en un partido de baloncesto:

<<enum>> TipoCanasta
+ TIRO_LIBRE
+ CANASTA_NORMAL
+ TRIPLE
+ getValor()

- El método getValor devuelve 1, 2 o 3 según sea el tipo de la canasta

A continuación, programa la enum TipoEquipo, que sirve para identificar local y visitante:

<<enum>> TipoEquipo
+ LOCAL
+ VISITANTE

Por último, programa la interfaz MarcadorBaloncesto, que representa el comportamiento que debe tener un objeto para ser considerado un marcador de baloncesto:

<<interface>> MarcadorBaloncesto
+ void añadirCanasta(TipoEquipo e, TipoCanasta t)
+ String getNombreEquipo(TipoEquipo e)
+ int getPuntos(TipoEquipo e)
+ default String getMarcador()

- El método getMarcador devuelve el nombre del equipo local, sus puntos, y luego el nombre del equipo visitante y sus puntos, con este formato:

NombreLocal puntosLocal – NombreVisitante puntosVisitante

Ejercicio 43: Usa las enum y la interfaz del ejercicio anterior para programar la clase MarcadorFacil:

MarcadorFacil implements MarcadorBaloncesto
- String nombreEquipoLocal
- String nombreEquipoVisitante
- int puntosLocal
- int puntosVisitante
+MarcadorFacil(String local, String visitante)

- El constructor simplemente rellena las propiedades con los valores recibidos y les pone 0 a las puntuaciones de ambos equipos
- añadirCanasta añade una canasta del tipo recibido como parámetro al equipo que también se pasa como parámetro
- getNombreEquipo devuelve el nombre del equipo local o del visitante, según sea el parámetro recibido
- getPuntos devuelve los puntos del equipo local o del visitante según sea el parámetro recibido

Test:

- Crea un partido Granada-Estudiantes con marcador 0,0 y comprueba que el marcador queda 33 a 26 tras añadir estas canastas:

	Canasta de 1 punto	Canasta de 2 puntos	Canasta de 3 puntos
Granada	5	8	4
Estudiantes	3	10	1

- Crea un partido Granada-Estudiantes con marcador 100-80 y comprueba que el método getMarcador devuelve “Granada 100 – Estudiantes 83”

Ejercicio 44: Programa el siguiente record, que representa un equipo que juega un partido de baloncesto:

<<record>> Equipo
+ Equipo(String nombre, int puntos) + Equipo(String nombre) + Equipo añadirCanasta(TipoCanasta t)

- El constructor principal crea e inicializa las propiedades. Si los puntos son negativos, lanzará una IllegalArgumentException.
- El constructor secundario crea un equipo que lleva 0 puntos
- El método añadirCanasta devuelve un Equipo con las mismas características del que estamos programando, pero añadimos a su puntuación los puntos que nos indica el tipo de canasta pasada como parámetro.

Test:

- Obtén, con el método estático **TipoCanasta.values** un array TipoCanasta[] que almacena las constantes de la enum. Recorre (con un for del lenguaje C) dicho array, y en cada iteración crea un equipo llamado HLanz Basket y añade la canasta recorrida. Comprueba que se obtiene un equipo cuya puntuación es correcta.

Ejercicio 45: Usa el record del ejercicio anterior para programar la clase MarcadorDefinitivo:

MarcadorDefinitivo implements MarcadorBaloncesto
- Equipo local - Equipo visitante
+ MarcadorDefinitivo(String local, String visitante)

- El constructor inicializa las propiedades creando objetos Equipo a partir de los nombres de los equipos recibidos.
- añadirCanasta añade una canasta del tipo recibido como parámetro al equipo que también se pasa como parámetro
- getNombreEquipo devuelve el nombre del equipo local o del visitante, según sea el parámetro recibido
- getPuntos devuelve los puntos del local o del visitante según indica el parámetro

Test:

- Crea un partido Granada-Estudiantes con marcador 0,0 y comprueba que el marcador queda 33 a 26 tras añadir estas canastas:

	Canasta de 1 punto	Canasta de 2 puntos	Canasta de 3 puntos
Granada	5	8	4
Estudiantes	3	10	1

Ejercicio 46: Programa la siguiente clase (se recomienda usar TDD), que representa un examen que un profesor realiza a su clase.

Examen
- List<String> alumnos
- List<Double> notas
+ Examen()
+ void añadirNota(String alumno, double nota)
+ double getNota(String alumno)
+ double getNotaMedia()
+ boolean mediaAprobado()
+ List<String> getAprobados()

- alumnos: Es una lista con los nombres de los alumnos
- notas: Es una lista con las notas de los alumnos, de forma que la nota de la posición P es la nota del alumno de la posición P que está en la lista de alumnos.
- Constructor: Crea un examen sin alumnos y sin notas
- añadirNota: Añade al examen al alumno y nota recibidos como parámetros. Si la nota no está entre 0 y 10 lanzará una `IllegalArgumentException`
- getNota: Devuelve la nota de un alumno, o lanza una `IllegalArgumentException` si no existe el alumno.
- getNotaMedia: Devuelve la nota media del examen. Si no hay alumnos, se lanza una `IllegalStateException`
- mediaAprobado: Devuelve true si la media de la clase sale mayor de 5
- getAprobados: Devuelve una lista con los nombres de los alumnos aprobados

Test:

- Crea un examen, añade que Antonio ha sacado un 6 y comprueba que la nota de Antonio es un 6
- Crea un examen y añade estos alumnos con sus notas

Alicia	10
Alejandro	0
Juan Antonio	5

Comprueba que su nota media es un 5, con un margen de error de 0.1

- Comprueba que en el test anterior, el método `mediaAprobado` sale true
- Crea un examen, añade que Antonio ha sacado un -20 y comprueba que se lanza una `IllegalArgumentException`
- Crea un examen y añade estos alumnos:

Armando	3
Matias	4.9
Rigoberto	2

Comprueba que el método `mediaAprobado` sale false

- Crea un examen y añade estos alumnos:

Juan Antonio	6
Jose Antonio	4
Oscar	7

Comprueba que `getAprobados` devuelve una lista con Juan Antonio y Oscar

- Crea un examen sin nada y calcula la media. Comprueba que sale `IllegalStateException`

Ejercicio 47: Programa la misma clase de antes (se recomienda usar TDD), cambiando las propiedades por un Map, de esta forma:

Examen
- Map<String,Double> notas
+ Examen() + void añadirNota(String alumno, double nota) + double getNota(String alumno) + double getNotaMedia() + boolean mediaAprobado() + List<String> getAprobados()

El Map de notas asocia a cada nombre su nota. Los métodos siguen haciendo lo mismo y puedes reutilizar los mismos tests de antes.

En este ejercicio estamos viendo cómo los tests nos ayudan a refactorizar una clase: hemos cambiado la estructura interna sin modificar su comportamiento al exterior (los métodos públicos)

Ejercicio 48: Programa siguiente clase (se recomienda usar TDD), que representa un restaurante:

Restaurante
- String nombre - String[] platos - int[] precios
+ Restaurante(String n, String[] pl, String[] pr) + Restaurante() + String getNombre() + String getPrecio() + boolean estaDisponible() + List<String> getPlatosBaratos() + int getCuenta(String[] platosConsumidos)

- nombre: Es el nombre del restaurante
- platos: Es una lista con todos los platos que se ofrecen en el restaurante
- precios: Es una lista con los precios de los platos del restaurante, de forma que el precio de la posición P de esta lista es el precio del plato de la posición P en la lista de platos

- primer constructor: Crea un restaurante con el nombre, platos y precios recibidos como parámetros.
- Segundo constructor: Crea un restaurante con el nombre recibido como parámetro, estos platos y precios:

bocadillo	5
filete	15
sopa	8
ensalada	9
caviar	30

- getNombre: Devuelve el nombre del restaurante
- getPrecio: Devuelve el precio del plato pasado como parámetro, o lanza una IllegalArgumentException si no existe ese plato
- estaDisponible: Devuelve true si el plato recibido como parámetro se ofrece
- getPlatosBaratos: Devuelve una lista con los platos cuyo precio es menor de 12 euros.
- getCuenta: Devuelve el precio de una consumición formada por todos los platos que se pasan como parámetro.

Test

- Crea un restaurante con los valores por defecto y comprueba que su nombre se ha inicializado correctamente y que los precios de los platos por defecto son correctos
- Crea un restaurante con los valores por defecto y comprueba que el bocadillo, la sopa y el filete están disponibles.
- Crea un restaurante con los valores por defecto y comprueba que al consultar el precio de los calamares se lanza una IllegalArgumentException
- Crea un restaurante con los valores por defecto y comprueba que no tienen calamares.
- Crea un restaurante con los valores por defecto y comprueba que la lista de platos caros solo tiene dos elementos, que son filete y caviar
- Crea un restaurante con los valores por defecto y comprueba que el precio de consumir bocadillo, filete y caviar es 50

Ejercicio 49: Programa la misma clase Restaurante del ejercicio anterior, cambiando los arrays de las propiedades por un Map, de esta forma:

Restaurante
- String nombre
- Map<String,Integer> platos
+ Restaurante(String n, String[] pl, String[] pr)
+ Restaurante()
+ String getNombre()
+ String getPrecio()
+ boolean estaDisponible()
+ List<String> getPlatosBaratos()
+ int getCuenta(String[] platosConsumidos)

El Map de notas asocia a cada plato su precio. Los métodos siguen haciendo lo mismo y puedes reutilizar los mismos tests de antes.

SUPER EJERCICIO

Al instituto H Lanz llegan los títulos de los alumnos que han terminado sus estudios y es necesario llevar un control de los títulos que están pendientes de llegar, los que están en el centro pero no han sido recogidos, y los que ya han sido retirados por sus dueños.

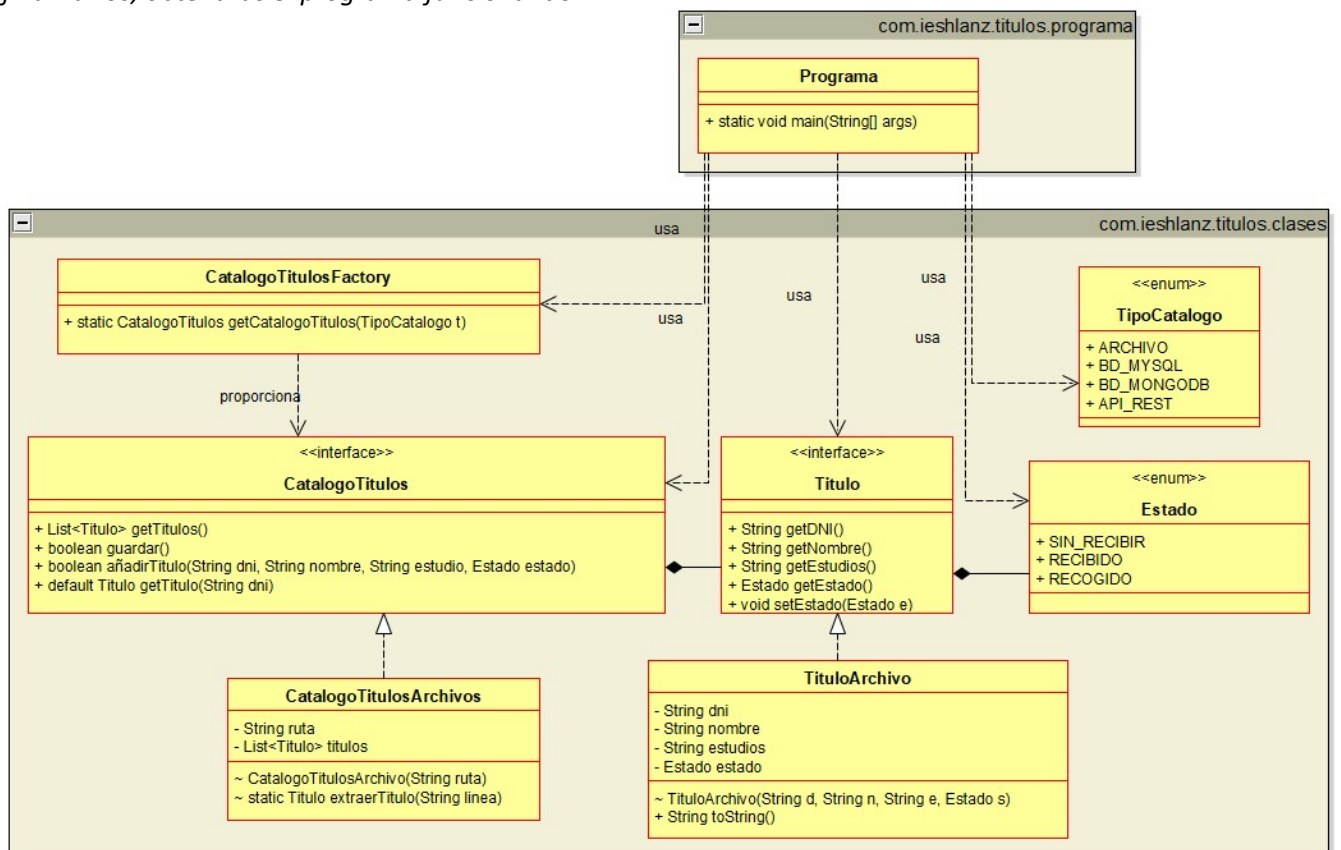
Se va a realizar un programa que guarda en un archivo la información sobre los títulos del instituto, y nos va a permitir gestionar dicha información.

En nuestro caso el programa estará organizado en las clases que se muestran en este diagrama, de manera que el paquete `com.ieshlanz.titulos.clases` contendrá las clases del programa, mientras que el paquete `com.ieshlanz.titulos.programa` tendrá todo lo relacionado con la interfaz, que en este caso, es de texto.

Los programas profesionales no constan de un único archivo en el que está todo mezclado (interfaz, acceso a archivos, etc) sino que están separados en clases para facilitar el trabajo en equipo y sobre todo, el mantenimiento.

Este programa está estructurado siguiendo el principio “Program to Interface”, que significa que el programa no interactúa directamente con clases, sino con interfaces (observa cómo las flechas que unen el programa con los elementos que necesita son básicamente interfaces). Posteriormente las interfaces son implementadas por clases que “trabajan con títulos guardados en archivos”. De esta forma, si alguna vez se decide cambiar de archivos a una base de datos, bastará con añadir nuevas implementaciones de las interfaces, mediante clases que “trabajan con títulos guardados en bases de datos” y no será necesario tocar casi nada el programa principal, facilitando así el mantenimiento.

En los siguientes ejercicios se explica la misión de cada elemento del diagrama y cómo programarlo. Al finalizarlos, obtendrás el programa funcionando.



Ejercicio 50: Programa la enum **Estado**, que recoge los posibles estados que puede tener un título:

- SIN_RECIBIR → El título aún no ha llegado al instituto
- RECIBIDO → El título ha llegado al instituto pero no ha sido recogido
- RECOGIDO → El título ha sido recogido por el alumno

Ejercicio 51: Programa la interfaz **Título**, que define los métodos que debe tener una clase que se pueda considerar un título.

- getters → Devuelve el valor del dni, nombre, estudios
- getEstado → Devuelve el estado del título
- setEstado → Cambia el estado al título

Ejercicio 52: Programa la interfaz **CatalogoTitulos**, que representa un lugar genérico (archivo, base de datos, etc) de donde podemos obtener títulos.

- getTitulos → Devuelve una lista inmutable con todos los títulos disponibles
- guardar → hace que la clase que implementa la interfaz guarde los títulos actualizados
- añadirTitulo → Las clases que implementen la interfaz usarán este método para añadir un título
- getTitulo → recorre todos los títulos en busca de aquel cuyo dni se pasa como parámetro. Si no se encuentra, lanzará una NoSuchElementException (es una RuntimeException)

Ejercicio 53: Programa la clase **TituloArchivo**, que es una implementación de Título que representa un título que está guardado dentro de un archivo que contiene muchos títulos

- La clase implementará los métodos de Título tal y como están explicadas en los ejercicios anteriores
- toString → Devuelve un String con este formato: **dni;nombre;estudios;ordinal del estado**

Test:

- Crea un título para el alumno Pepe López, con dni 12345678M, que ha estudiado “Explotación de Sistemas Informáticos” y estado RECIBIDO. Comprueba que todas las propiedades se han inicializado correctamente y que el método toString devuelve:

12345678M;Pepe López;Explotación de Sistemas Informáticos;1

Ejercicio 54: Programa la clase **CatalogoTitulosArchivo**, que es una implementación de la interfaz CatalogoTitulos, en la cual los títulos están guardados en un archivo.

- Constructor: Inicializa la propiedad ruta con el parámetro recibido. Si el archivo no existe, lo crea vacío. En caso de que el archivo exista, lo abre y lee una por una todas sus líneas. Cada línea leída será pasada al método **extraerAlumno**, que nos devolverá un alumno que será almacenado en la lista **alumnos**.

- añadirTitulo: Crea un objeto **TituloArchivo** con los parámetros recibidos y lo añade a la lista de títulos
- getTitulos: Devuelve una versión inmutable de la lista de títulos. Para ello se usará el método **Collections.unmodifiableList**

☞ *¿y esto por qué? Si getTitulos devuelve directamente el ArrayList<Titulo> de las propiedades, un programador podría añadir o borrar títulos directamente sobre él, ignorando cualquier comprobación que se pudiera poner en los métodos de la clase. Por ese motivo, a los programadores se les da una versión de solo lectura, de forma que las modificaciones del ArrayList se hagan solo con los métodos proporcionados por la clase*

- guardar: Crea un archivo en la ruta indicada por la propiedad **ruta**. A continuación, recorre la lista **alumnos** y por cada uno, llama a su método **toString** y se guardará en el archivo el String que nos ha devuelto ese método. No olvides cerrar el archivo al terminar. Si se hace bien, deberá crearse un archivo con una línea de texto por cada alumno
- extraerTitulo: Este método recibe una línea en la que aparecen los datos del título separados por punto y coma (igual que la que devuelve el método toString de TituloArchivo), y creará un objeto **TituloArchivo** a partir de ellos.

Test: (hazlos en el mismo paquete de las clases)

- Crea un CatalogoTitulosArchivo, llama al método **extraerTitulo** pasándole el siguiente String, y comprueba que el Titulo que devuelve es correcto

12345678M;Pepe López;Explotación de Sistemas Informáticos;1

- Crea un CatalogoTitulosArchivo, añade los títulos de la tabla y comprueba que el método de añadir devuelve true. A continuación, guarda el archivo en la ruta que tú quieras y comprueba que dicho método devuelve true.

DNI	Nombre	Estudios	Estado
11111111H	Luis López	Carrocería	Recogido
22222222W	María Pérez	Desarrollo de aplicaciones web	Sin recoger
33333333M	John López	Electricidad	Recibido

- Crea un CatalogoTitulosArchivo para abrir el archivo generado en el test anterior. Comprueba que hay exactamente 3 títulos en el catálogo, y que cada uno tiene los datos indicados en la tabla

Sugerencia: En este momento del curso hay dos formas de hacerlo. Elige la que prefieras:

- Saca el primer título de la lista y comprueba que sus datos son correctos mirando la tabla. Repite lo mismo con el segundo y tercer título.
- Crea un String[] con la representación textual (toString) de cada título de la tabla. Recorre la lista de títulos del catálogo y comprueba que su método toString coincide con la representación textual del String[]

Ejercicio 55: Programa la enum **TipoCatalogo**, que recoge los posibles lugares de donde podemos obtener títulos (archivos, bases de datos). En este momento solo es posible obtener los títulos de archivos.

- ARCHIVO → Los datos se obtienen de archivos
- BD_MYSQL → Los datos se obtienen de una base de datos MySQL
- BD_MONGODB → Los datos se obtienen de una base de datos MongoDB
- API_REST → Los datos se obtienen consultando a un servidor mediante un API

Ejercicio 56: Programa la enum **CatalogoTitulosFactory**, que permite al programa obtener un catálogo de títulos del tipo que necesite.

- getCatalogoTitulos: Recibe un tipo de catálogo, y si es ARCHIVO, devuelve un objeto de la clase CatalogoTitulosArchivo. En cualquier otro caso, lanzará una IllegalArgumentException con el mensaje “Tipo de catálogo no soportado por el programa”.

Test:

- Usa la clase CatalogoTitulosFactory para obtener una implementación de CatalogoTitulos que trabaje sobre archivos. Comprueba que no ha salido null.
- Repite lo anterior para obtener una implementación de CatalogoTitulos que trabaje sobre una base de datos MySQL. Comprueba que se lanza una IllegalArgumentException.

Ejercicio 57: Programa la clase **Programa**, que es el programa en si. Funciona de esta manera:

- Obtiene un CatalogoTitulos que trabaje sobre un archivo llamado **títulos.csv**
- Mostrará este menú en bucle, hasta pulsar la opción de salir.
 1. Añadir título
 2. Consultar todos los títulos
 3. Cambiar estado del título
 4. Salir
- La opción 1 preguntará al usuario por teclado un DNI, nombre, estudios y estado y lo añadirá al catálogo de títulos
- La opción 2 recorrerá todos los títulos del catálogo y nos mostrará la salida del método toString de cada título en pantalla.
- La opción 3 preguntará un dni por teclado y recuperará del catálogo de títulos el título con ese dni (si no se encuentra, se informará al usuario). Nos preguntará el nuevo estado que le queremos poner al título y lo actualizará.
- La opción 4 llama al método guardar del catálogo de títulos y el programa finalizará.