

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.....	13
5.- Programación de métodos de instancia.....	25
6.- Programación de métodos que lanzan excepciones.....	35
7.- Programación de métodos recursivos.....	43
8.- Programación de métodos estáticos.....	47
9.- Propiedades estáticas.....	49
10.- Interfaces	51
11.- Records.....	56
12.- Enumeraciones.....	60
13.- Test Driven Development (TDD)	63

Tema

4

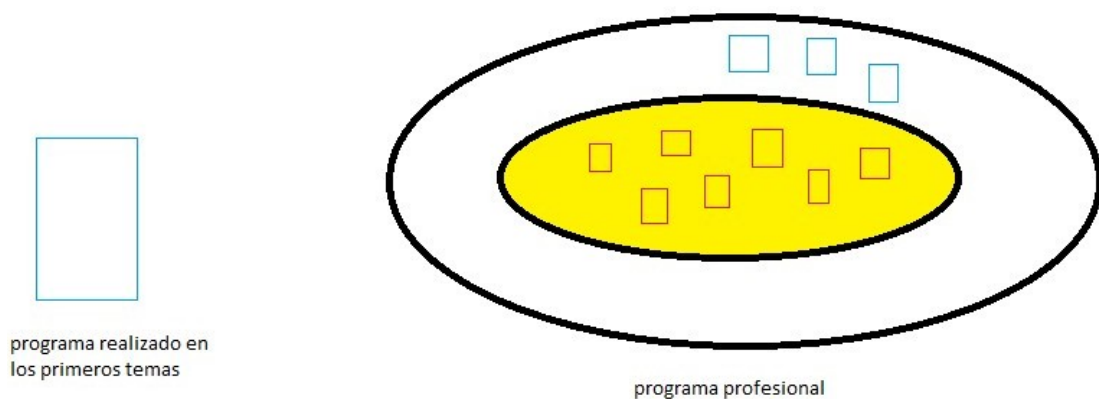
PROGRAMACIÓN BÁSICA DE CLASES

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.

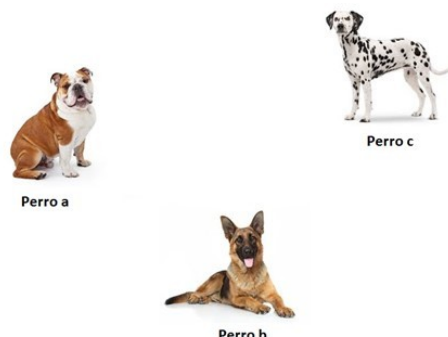
Por último, como veremos durante el tema, para programar una clase deberemos disponer del **diagrama de la clase** que vamos a programar. Como ya sabemos, en ese diagrama aparece la forma de los métodos que hay en la clase, y será muy útil para programar la clase.

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.

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.

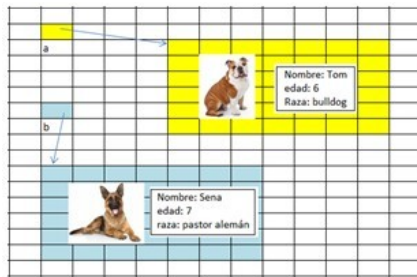
Sin embargo, cada uno de estos objetos lleva consigo **variables internas** que definen como es y cómo se encuentra el perro en ese momento. En la siguiente imagen podemos ver cuáles son estas variables para los tres objetos de la clase Perro de imagen anterior:



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
- 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) + 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 todo 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.

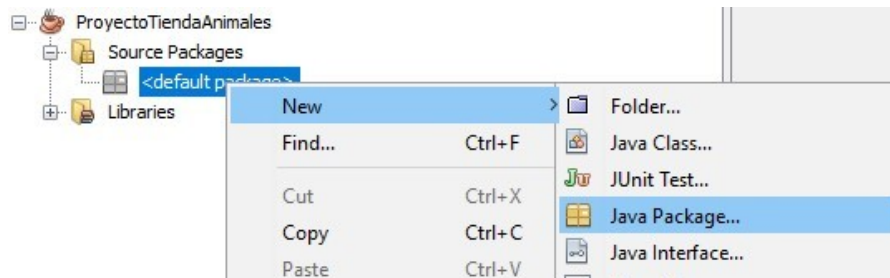
También existen otros dos modificadores de acceso, que se encuentran a medio camino entre el public y el private. Es decir, los puede ver quien programa la clase, pero no quien programa la aplicación. Sin embargo, el efecto que producen y sus diferencias con private los estudiaremos en el próximo tema.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

- El signo #, llamado **protected**
- No poner modificador a la propiedad. Esto se llama **modificador por defecto**³

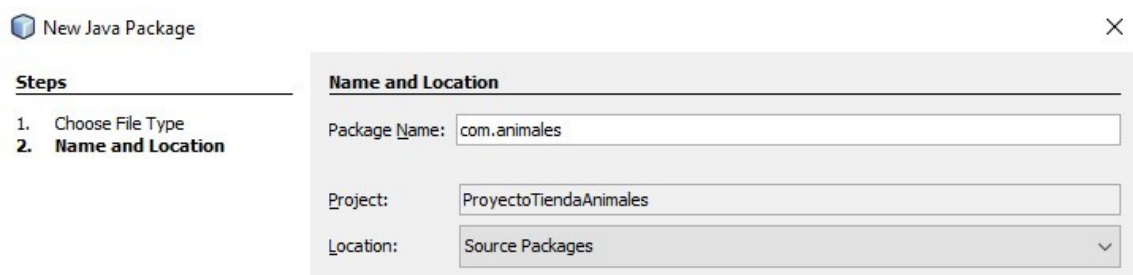
³ El modificador por defecto también se denomina **modificador package** o **modificador package-private**

3.- Programación de una clase

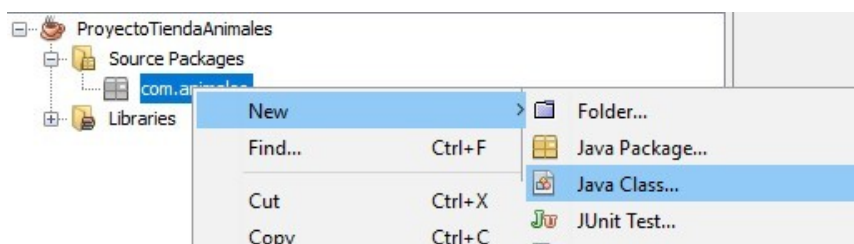
Podemos añadir clases a cualquier proyecto que hagamos. Lo primero que haremos será crear un paquete⁴ dentro del proyecto pulsando el botón derecho del ratón sobre “default package” y eligiendo **new → Java package**



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.

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)

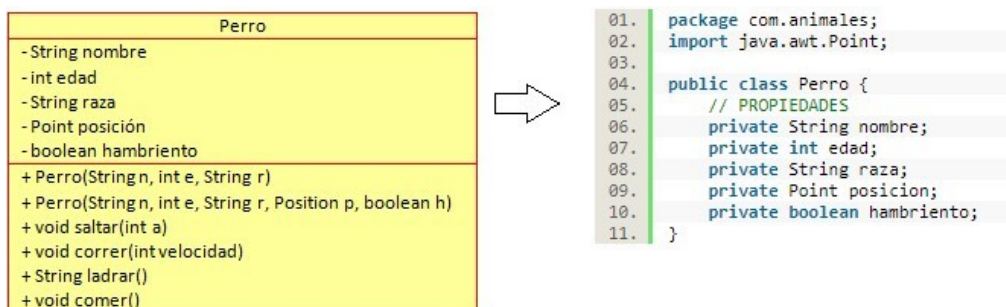
⁵ Este tipo de nombre se denomina **notación dns inversa**

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

```
1. package com.animales;  
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:

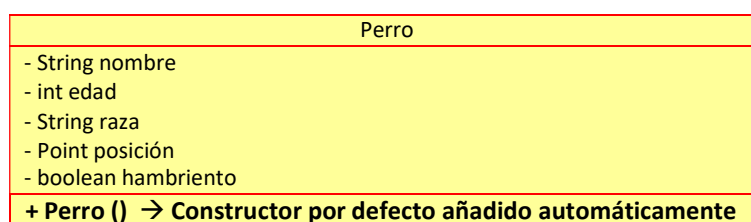


Es importante darse cuenta de estos detalles:

- Las propiedades están escritas en color **verde**⁶. Esto es muy importante porque así podremos diferenciarlas de las variables normales que usemos más adelante.
- Las propiedades no se inicializan, sino que simplemente, se dejan declaradas.
- 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).

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

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

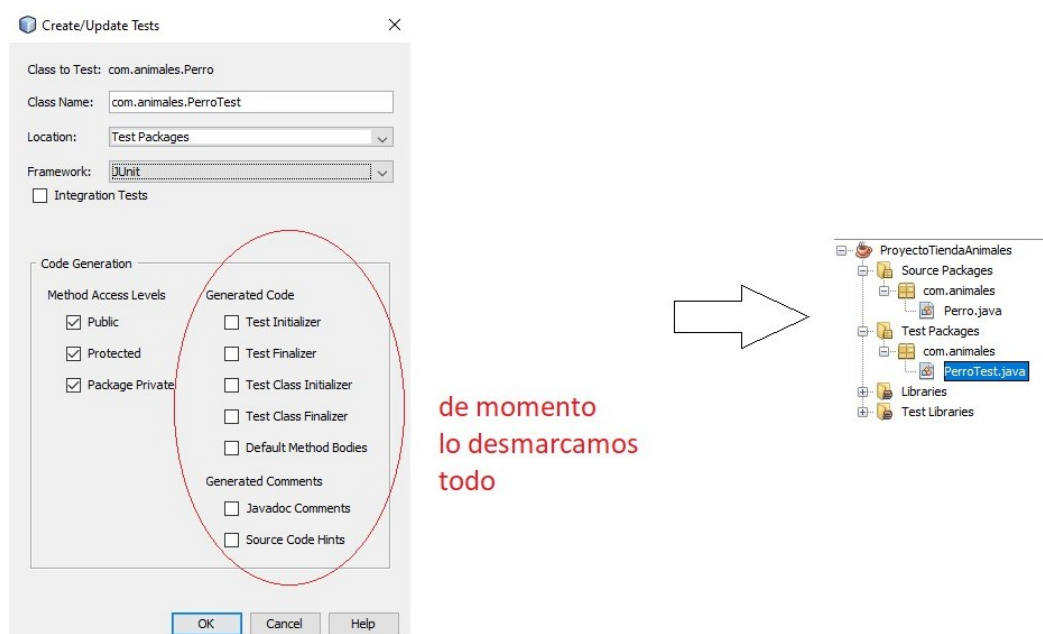
3.3.- Test de la clase

¿Cómo podemos probar que la clase que hemos hecho funciona correctamente? Recuerda que estamos haciendo una clase y no un programa.

☞ **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. El resultado era un montón de programas desorganizados.

Actualmente se ha dado un paso más y se utilizan los **test de JUnit**, que es una herramienta que viene con el IDE y sirve para probar las clases conforme las vamos programando. Durante el tema iremos viendo la enorme utilidad que tienen estos test.

Para hacer los test de la clase Perro pulsamos el botón derecho del ratón en ella, elegimos **tools** → **create/update test** y elegimos como framework **JUnit4**



Cuando pulsamos el botón de aceptar, veremos que NetBeans nos crea en el proyecto una carpeta llamada **Test Packages** y en ella un archivo llamado **PerroTest.java** con una apariencia más o menos así (la apariencia puede variar según el IDE utilizado):

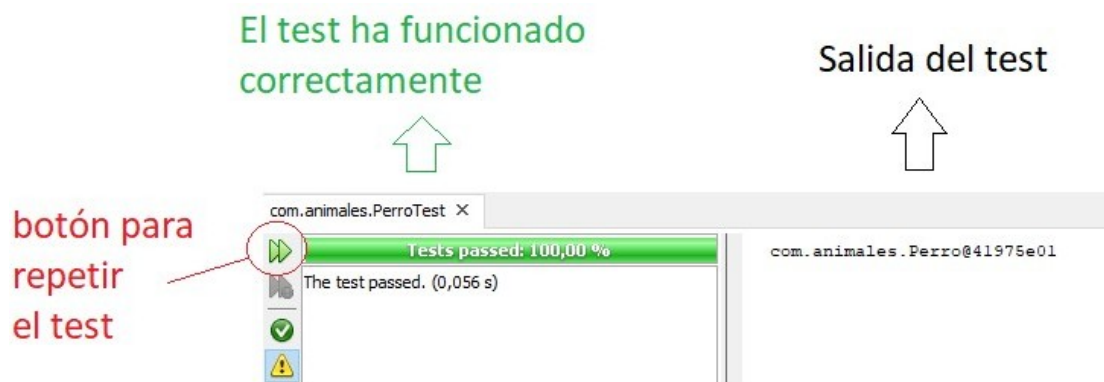
```
1. package com.animales;
2.
3. import org.junit.Test;
4. import static org.junit.Assert.*;
5.
6. public class PerroTest {
7.
8.     public PerroTest() {
9.     }
10.
11.     @Test
12.     public void testSomeMethod() {
13.     }
14.
15. }
```

En este archivo lo importante es el bloque que hay señalado en amarillo. Como vemos, ese bloque aparece anotado con **@Test** y eso significa que dentro de ese bloque podemos hacer un pequeño programa de prueba con cualquier cosa que se nos ocurra para comprobar que la clase funciona bien.

Por ejemplo, vamos a crear un objeto de la clase Perro y mostrar por pantalla el valor que nos devuelve su método toString.

```
1. package com.animales;
2.
3. import org.junit.Test;
4. import static org.junit.Assert.*;
5.
6. public class PerroTest {
7.
8.     public PerroTest() {
9.     }
10.
11.     @Test
12.     public void testSomeMethod() {
13.         Perro p=new Perro();
14.         System.out.println(p.toString());
15.     }
16.
17. }
```

Si pulsamos el botón derecho del ratón dentro del editor de texto y **run file** se ejecutará el test y NetBeans nos mostrará el resultado:




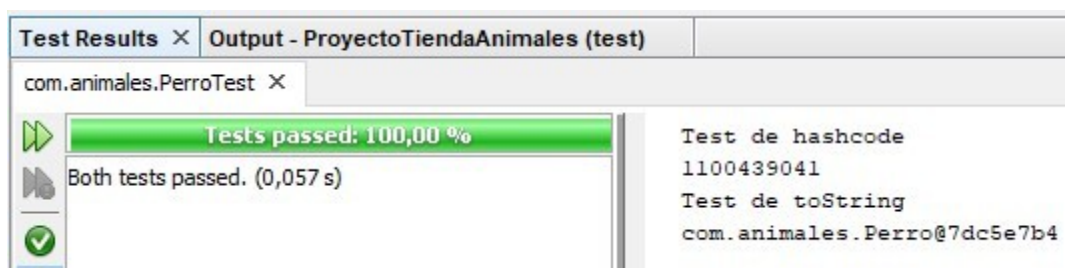
Lo bueno de JUnit es que podemos añadir todos los test que queramos a ese archivo. Cada test es como una especie de “miniprograma” que se añade al archivo y que tendrá este formato:

```
1. @Test
2. public void nombre(){
3.     // aquí viene el código fuente del test
4. }
```

Por ejemplo, vamos a añadir un nuevo test que muestre por pantalla el hashCode del perro:

```
1. package com.animales;
2.
3. import org.junit.Test;
4. import static org.junit.Assert.*;
5.
6. public class PerroTest {
7.
8.     public PerroTest() {
9.     }
10.
11.     // este test muestra el método toString del perro por pantalla
12.     @Test
13.     public void testSomeMethod() {
14.         System.out.println("Test de toString");
15.         Perro p=new Perro();
16.         System.out.println(p.toString());
17.     }
18.
19.     // este test muestra el hashCode del perro por pantalla
20.     @Test
21.     public void testHashCode(){
22.         System.out.println("Test de hashCode");
23.         Perro p=new Perro();
24.         System.out.println(p.hashCode());
25.     }
26. }
```

Al ejecutar pulsar el botón  se ejecutarán todos los test que haya en el archivo PerroTest.java y si falla algún test, la barra se mostrará en rojo.



Gracias a este tipo de pruebas, podemos tener una colección de test que pueden ser lanzados de forma automática en cualquier momento. Esto es muy importante sobre todo en la fase de **mantenimiento** de un programa, porque así, cuando hagamos un cambio, podemos lanzar nuevamente todos los test para ver si todo sigue funcionando bien o hemos introducido un error inesperado en algún lugar del código fuente.

Usar JUnit para programar una clase es como ponernos una red de seguridad

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

La clase Perro que hemos hecho compila correctamente y con los test hemos podido comprobar que funciona correctamente. Los test crean un objeto de la clase Perro y llaman a sus métodos heredados 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 a los test y podremos modificar libremente las propiedades de un objeto Perro, utilizando una línea con este formato:

a.edad = 9;

El diagrama muestra la línea de código `a.edad = 9;` con tres flechas que apuntan a sus componentes: la primera flecha apunta a `a` y está etiquetada como *variable con el objeto*; la segunda flecha apunta a `.edad` y está etiquetada como *propiedad*; la tercera flecha apunta a `9` y está etiquetada como *nuevo valor*.

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

```
1. @Test
2. public void test(){
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).

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 variable. Por ejemplo, este test crea dos perros y le cambia el nombre a cada uno.

```

1. @Test
2. public void test(){
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:



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:

```

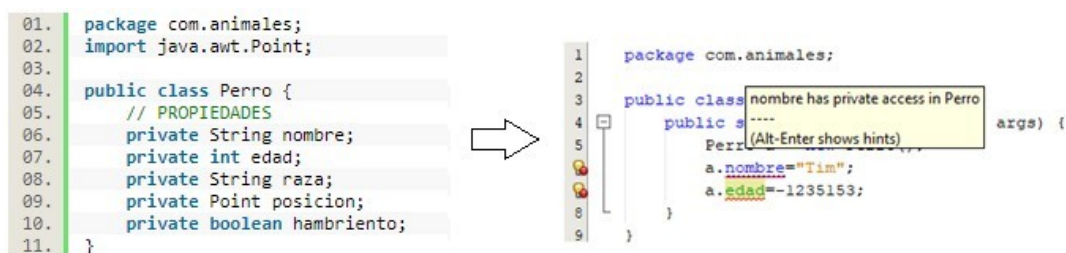
1. public class Programa{
2.     public static void main(String[] args){
3.         Perro a = new Perro();
4.         a.nombre="Tim";
5.         a.edad=-1235153;
6.     }
7. }

```

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 aplicaciones y 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 private (o protected/package), intentar acceder a ella en un programa dará un error de compilación como el que se ve en la imagen:



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**.

Por ejemplo, supongamos que queremos añadir un método constructor que permita crear un Perro indicando su nombre, edad y raza (observa que al añadir nuestro constructor propio, desaparece el constructor por defecto que ponía Java automáticamente):

Perro
+ String nombre + int edad + String raza + Point posición + boolean hambriento
+ Perro(String n, int e, String r)

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:

Quiero crear un nuevo perro:

Perro p = new Perro ("Tim",4,"San Bernardo");



Programador de aplicaciones

En el constructor inicializo las propiedades así:

nombre="Tim"
edad=4
raza="San Bernardo"



Programador de clases

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, que se limitaría a realizar la asignación que hemos dicho:

```

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.
11.     public Perro(String n, int e, String r){
12.         nombre=n;
13.         edad=e;
14.         raza=r;
15.     }
16. }

```

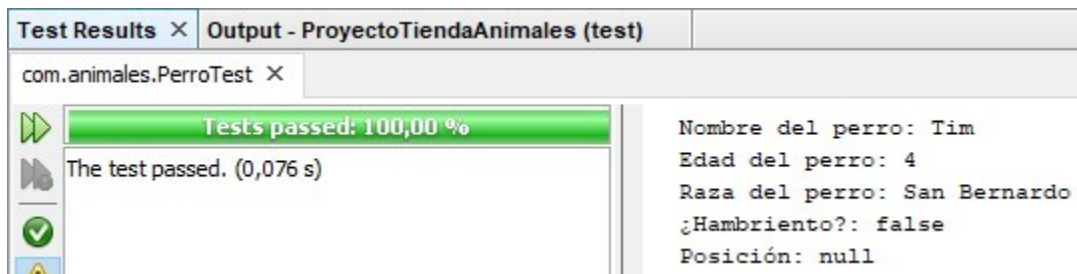
A partir de este momento, el programador de clases ya puede crear un objeto Perro indicando en el constructor su nombre, edad y raza. ¿Qué sucede con las propiedades posición y hambriento? Ambas se quedan con el valor por defecto asociado a su tipo de dato: null para posición y false para hambriento.

Podemos comprobarlo con un test que cree un perro y muestre en pantalla el valor de sus propiedades:

```

1. @Test
2. public void test() {
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 queremos, en el constructor podemos cambiar el valor por defecto de las propiedades posición y hambriento y poner otro diferente, pero deberemos indicarlo en la documentación del método. Por ejemplo, vamos a modificar el constructor de antes, pero ahora, la localización es 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 Perro(String n, int e, String r){
2.     nombre=n;
3.     edad=e;
4.     raza=r;
5.     posicion=new Point(0,0);
6.     hambriento=false;
7. }

```

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



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. 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, es muy importante escribir en la documentación de la clase todas estas decisiones que hacemos cuando programamos la clase.

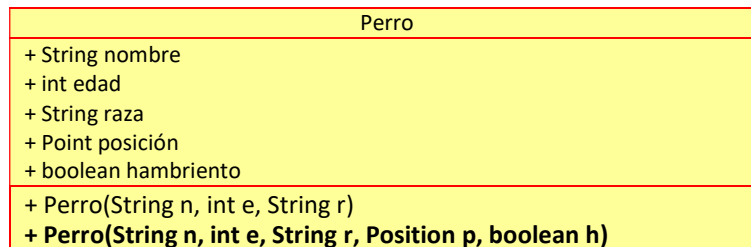
```
1. public class Perro {
2.     // PROPIEDADES
3.     public String nombre;
4.     public int edad;
5.     public String raza;
6.     public Point posicion;
7.     public boolean hambriento;
8.
9.     // MÉTODOS CONSTRUCTORES
10.    public Perro(String n, int e, String r){
11.        nombre=n;
12.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

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.
9.     // MÉTODOS CONSTRUCTORES
10.    public Perro(String n, int e, String r){
11.        nombre=n;
12.        edad=e>0?e:1;
13.        raza=r;
14.        posicion=new Point(0,0);
15.        hambriento=false;
16.    }
17. }
```

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.
9.     // MÉTODOS CONSTRUCTORES
10.    public Perro(String n, int e, String r){
11.        nombre=n;
12.        edad=e>0?e:1;
13.        raza=r;
14.        posicion=new Point(0,0);
15.        hambriento=false;
16.    }
17.
18.    public Perro(String n, int e, String r, Position p, boolean h){
19.        nombre=n;
20.        edad=e>0?e:1;
21.        raza=r;
22.        posicion=p;
23.        hambriento=h;
24.    }
25. }
```

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.
9.     // MÉTODOS CONSTRUCTORES
10.    public Perro(String n, int e, String r){
11.        nombre=n;
12.        edad=e>0?e:1;
13.        raza=r;
14.        posicion=new Point(0,0);
15.        hambriento=false;
16.    }
17.
18.    public Perro(String n, int e, String r, Position p, boolean h){
19.        nombre=n;
20.        edad=e>0?e:1;
21.        raza=r;
22.        posicion=p;
23.        hambriento=h;
24.    }
25.
26.    public Perro(String n, int e, String r, int x, int y, boolean h){
27.        nombre=n;
28.        edad=e>0?e:1;
29.        raza=r;
30.        posicion=new Position(x,y);
31.        hambriento=h;
32.    }
33. }

```

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.- Llamada de un constructor a otro

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

Si nos damos cuenta, el segundo constructor es el que recibe todas las propiedades, y el primero y el tercero no son más que casos particulares de este. Por tanto, cuando programemos el primer y tercer constructor, podremos **llamar** al segundo, de forma que siempre se active dicho constructor. Es decir, el primer y el tercer constructor van a redirigir al segundo constructor.

Para hacer que un constructor llame a otro, se escribe la palabra **this** y entre paréntesis se ponen los parámetros que necesita el constructor que estamos llamando.

Por ejemplo, vamos a volver a programar el primer constructor, de forma que llame al segundo. El segundo constructor necesita 5 parámetros (todas las propiedades), mientras que el primero recibe solo 3 (porque los otros 2 son fijos). Lo único que hay que hacer es irnos al primer constructor y escribir **this** encerrando entre paréntesis los 5 parámetros que nos pide entre paréntesis el segundo constructor, 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.        nombre=n;
17.        edad=e>0?e:1;
18.        raza=r;
19.        posicion=p;
20.        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. Hacer esto es un poco más difícil que “copiar y pegar” el constructor, 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 segundo, 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 podríamos haber programado el primer constructor para que utilizase 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. }

```

4.4.- Ocultación de propiedades

Hay muchas veces en que en el diagrama de clases los nombres de los parámetros del constructor vienen escritos igual que el nombre de las propiedades⁶. Por ejemplo, imaginemos que la clase Perro viene escrita así:

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

En este caso, a la hora de programar el método constructor, cuando escribimos “raza”, ¿nos referimos al parámetro “raza” o nos referimos a la propiedad “raza”?

```

01. public class Perro {
02.     // PROPIEDADES
03.     public String nombre;
04.     public int edad;
05.     public String raza;
06.     public Point posicion;
07.     public boolean hambriento;
08.
09.     // MÉTODOS CONSTRUCTORES
10.
11.     public Perro(String nombre, int edad, String raza, Position posicion, boolean hambriento){
12.
13.     }
14.
15. }

```

¿raza?

Siempre que pasa esto, el parámetro **oculta** a la propiedad. Esto significa que si escribimos “raza”, prevalece el parámetro recibido y por tanto, no es posible acceder a la propiedad “raza”, que queda oculta por el parámetro también llamado “raza”.

Afortunadamente, hay una forma de poder acceder a la propiedad, que es usando la palabra reservada **this**. La palabra “this” es una referencia al objeto que estamos programando. De esta forma, podemos programar el constructor así:

⁶ 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 ninguno de los nombres 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. A menudo, encontramos en los diagramas de clases nombres de parámetros que pueden ocultar a las propiedades.

```

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 nombre, int edad, String raza, Position posicion, boolean hambriento){
11.        this.nombre=nombre;
12.        this.edad=edad;
13.        this.raza=raza;
14.        this.posicion=posicion;
15.        this.hambriento=hambriento;
16.    }
17. }

```

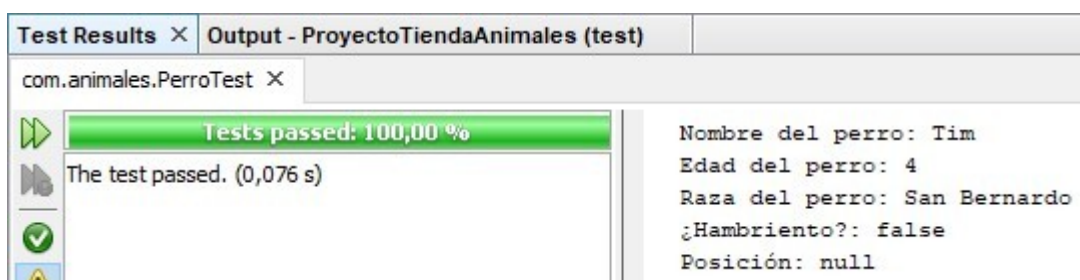
4.5.- Asertos

Volvamos al tema de los test. Si nos fijamos bien, los test que hemos hecho hasta ahora se limitan a mostrar por pantalla mensajes para que nosotros veamos si los datos son correctos. Por ejemplo, este test (hecho anteriormente) nos sirve para que miremos si las propiedades han sido asignadas correctamente a un objeto y por tanto, es útil para saber si un constructor ha sido bien programado:

```

10. @Test
11. public void testConstructorCorrecto() {
12.     Perro p=new Perro("Tim",4,"San Bernardo");
13.     System.out.println("Nombre del perro: "+p.nombre);
14.     System.out.println("Edad del perro: "+p.edad);
15.     System.out.println("Raza del perro: "+p.raza);
16.     System.out.println("¿Hambriento?: "+p.hambriento);
17.     System.out.println("Posición: "+p.posicion);
18. }

```



El problema de hacer esto es que al final tenemos muchísimos test y la salida por pantalla puede llegar a ser muy extensa y convertirse en un barullo muy grande en el que nos cueste trabajo encontrar lo que queremos.

Para arreglar este problema surgen los **asertos**, que son métodos que nos permiten **eliminar la salida por pantalla de los test**, de forma que sea el ordenador quien “mire” si un valor es correcto. El test fallará si el valor que debería mostrarse en pantalla no coincide con el valor que esperaríamos encontrar.

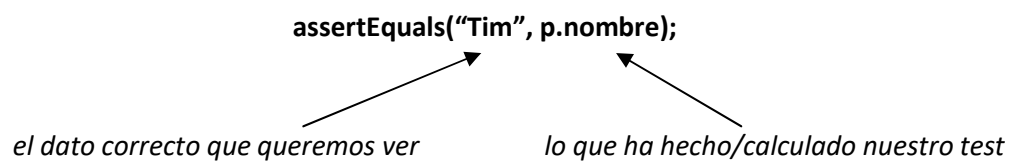
Vamos a cambiar el test del ejemplo anterior para que en lugar de mostrar por pantalla los datos de las propiedades (para que los vea un humano con sus ojos), use asertos para que los vea el ordenador y nos avise si los resultados son incorrectos.

La idea es la siguiente: cuando en un test escribimos esto:

```
1. Perro p=new Perro("Tim",4,"San Bernardo");
2. System.out.println("Nombre del perro: "+p.nombre);
```

es porque queremos ver la palabra "Tim" escrita en la pantalla.

Usando asertos, la línea 2 la reemplazaremos por:



O sea, el test completo reemplazando todas las salidas por pantalla por asertos se quedará así:

```
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. }
```

que, como podemos ver, sigue funcionando igual de bien que antes, con la diferencia de que ahora **no se muestra nada por pantalla** y el test es transparente al programador.



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:

```
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.         nombre = n;
9.         e=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:



Gracias a los asertos podemos añadir todos los test que queramos y lanzarlos de forma automatizada cada vez que hagamos algún cambio en el proyecto.

El tema de los test ha evolucionado muchísimo⁷ 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.

En los siguientes ejercicios, programa las clases y añade varios test que usen asertos para comprobar que los métodos constructores que has programado funcionan correctamente.

Ejercicio 1: Haz un proyecto, crea en él un paquete llamado **daw.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).

⁷ De hecho, hay librerías como **Hamcrest** que añaden muchos más tipos de asertos, permitiendo incluso hacer razonamientos lógicos con ellos

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()

- 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 7”

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 nVI, 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.

Ejercicio 4 : Añade al paquete daw.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.

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 que se llama como indica el nombre recibido, y la lista de trabajadores está vacía.
- 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.

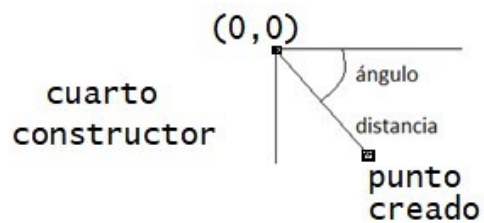
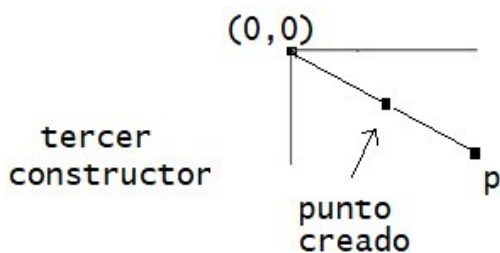
Ejercicio 6 : Programa la siguiente clase, que representa un punto de la pantalla que está en unas coordenadas (x,y) comprendidas entre 0 y el tamaño de la pantalla.

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 y distancia del origen, según se indica la imagen adjunta. (*Sugerencia: usa la definición de seno y coseno para obtener el valor de las coordenadas x e y*)



5.- 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.

5.1.- 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 llamar⁸ a dicho método **setNombre**.

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 una propiedad llamada “nombre”, lo que deberemos hacer para que el cambio de nombre sea efectivo es simplemente asignar el nombre nuevo (que es el parámetro “nuevoNombre”) a la propiedad, así:

```
1. public void setNombre(String nuevoNombre){  
2.     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.

⁸ 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) + 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.     }

```

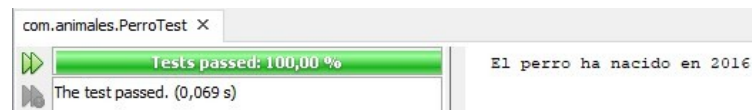
Como vemos, se han usado dos variables auxiliares: una para guardar el objeto LocalDate con la fecha actual y otra para el año. Estas variables desaparecen al terminar el método, por lo que en otro método distinto podemos reusar sus nombres.

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

```

7.     @Test
8.     public void test(){
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 apartado 5.2).

- **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 podemos usar la palabra **this**, aunque su uso es opcional y puede omitirse.

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.     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. }
```

En esta primera versión estamos repitiendo el código fuente del método mostrarAñoNacimiento, por lo que podemos evitar dicha redundancia llamando a ese método. Para llamar a un método del objeto que estamos programando, usamos la palabra **this** como si “this” fuese una variable con nuestro propio objeto:

```
1. public void setEdad(int e){
2.     nombre = nuevoNombre;
3.     this.mostrarAñoNacimiento();
4. }
```

El lenguaje Java nos permite dejar sin poner la palabra **this**, por lo que otra forma de hacer lo mismo sería esta:

```
1. public void setEdad(int e){
2.     nombre = nuevoNombre;
3.     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(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 **daw.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.

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. Si la caja está cerrada, no hace nada.
- pasarMayusculas: Reemplaza el mensaje de la caja por su versión en letras mayúsculas.

Ejercicio 9 : Añade a la clase Caja 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 nVl, 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.

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.

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

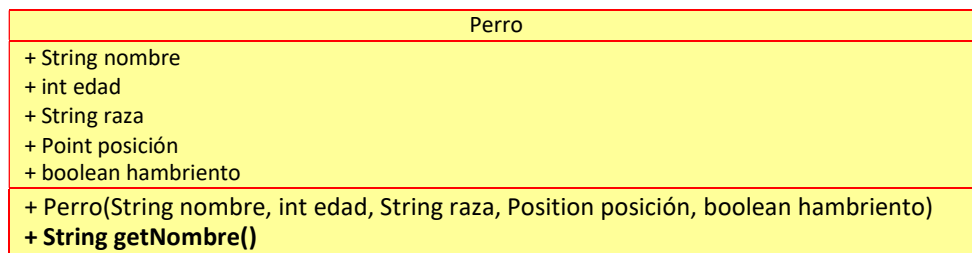
5.2.- Programación de métodos función

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**⁸ que devuelva al programador de aplicaciones el nombre del perro. O sea, el diagrama de clases será este:

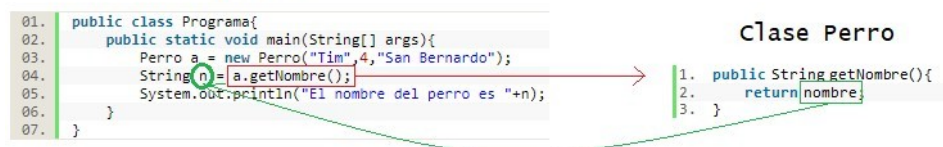


Como el objetivo de ese método es devolver el valor del nombre del perro, y este está almacenado en la propiedad “nombre”, lo único que hay que hacer para programar dicho método es usar la palabra **return** para devolver al programador de aplicaciones dicho dato:

```
1. public String getNombre(){
2.     return 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.

Aplicación



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

⁸ 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.

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

```

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.         nombre = n;
9.         edad=e;
10.        raza = r;
11.    }
12.    public String getNombre() {
13.        return nombre;
14.    }
15.    public int getEdad() {
16.        return edad;
17.    }
18.    public String getRaza() {
19.        return raza;
20.    }
21.    public Point getPosicion() {
22.        return posicion;
23.    }
24.    public boolean getHambriento() {
25.        return 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. }

```

test cuando las propiedades
eran public (ya no compila)



```

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 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.

Por ejemplo, vamos a añadir un método llamado **getEsperanzaVida** que nos devuelve 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(raza.equals("Maltés")||raza.equals("Chihuahua")||raza.equals("Yorkshire")){
4.         esperanza =18;
5.     }else if(raza.equals("Gran Danés")||raza.equals("San Bernardo")||raza.equals("Mastín")){
6.         esperanza =8;
7.     }else if(raza.equals("Grifón")||raza.equals("Bull Terrier")){
8.         esperanza =14;
9.     }
10.    return esperanza;
11. }
```

Esta forma no tiene buen mantenimiento, ya que si queremos añadir nuevas razas, tenemos que buscar el lugar adecuado y puede ser algo lioso. Vamos a hacer ese método de otra forma que va a usar 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 = Arrays.asList("Maltés","Chihuahua","Yorkshire");
4.     List<String> lista8 = Arrays.asList("Gran Danés","San Bernardo","Mastín");
5.     List<String> lista14 = Arrays.asList("Grifón","Bull Terrier");
6.     if(lista18.contains(raza)){
7.         esperanza =18;
8.     }else if(lista8.contains(raza)){
9.         esperanza =8;
10.    }else if(lista14.contains(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

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

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

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

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 que son mileuristas.
- getMileuristas: Devuelve una lista formada por todos 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.

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.

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.

6.- 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.

6.1.- Métodos que lanzan CheckedExceptions

Vamos a programar un método **comer()** en la clase Perro que lo alimente. Vamos a hacerlo 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
<ul style="list-style-type: none"> •String nombre •int edad •String raza •Point posición •boolean hambriento
+ Perro(String nombre, int edad, String raza) + void comer() throws Exception

En lugar de Exception, en el diagrama de la clase podemos encontrar cualquier tipo de checked exception, como por ejemplo IOException. 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 lanzar la excepción, creamos un objeto de la clase **Exception** (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 Exception {
2.     if(hambriento){
3.         System.out.println("El perro ha comido");
4.         hambriento=false;
5.     }else {
6.         // creamos la excepción
7.         Exception e=new Exception("El perro porque no tiene hambre");
8.         // lanzamos la excepción
9.         throw e;
10.    }
11. }
```

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 (Exception 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 diferente. En él crearemos un perro sin hambre y le daremos de comer. En este caso, esperamos que se lance una excepción y pondremos **fail** en caso de que no se lance, para indicar que es el caso incorrecto.

```

1. @Test
2. public void testHambre2() {
3.     Perro p=new Perro("Tim",4,"San Bernardo");
4.     // me aseguro de que el perro tenga hambre
5.     p.setHambriento(false);
6.     try {
7.         // doy de comer al perro
8.         p.comer();
9.         // compruebo que después de comer el perro ya no tiene hambre
10.        fail("No se ha lanzado la excepción");
11.    } catch (Exception ex) {
12.        // no se pone nada, porque es lo que esperamos
13.    }
14. }

```

En este test vemos que estamos dejando un catch vacío, lo cual sabemos que no es una buena costumbre. Para evitarlo, JUnit nos permite que en el caso en que esperamos una excepción, escribamos su tipo dentro de la anotación @Test y además le pongamos el “throws Exception” al test. Así nos evitamos usar el try-catch dentro del test:

```

1. @Test(expected=java.lang.Exception.class)
2. public void testHambre2() throws Exception{
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 17 : Añade a la clase Oficina el siguiente método:

Oficina
+ double getSueldoMedio() throws Exception

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

Ejercicio 18 : Modifica el método **retirarDinero** de la clase **CuentaCorriente** para que lance una Exception en caso de que se intente retirar una cantidad de dinero mayor que el saldo de la cuenta.

CuentaCorriente
+ void retirarDinero(int cantidad) throws Exception

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

Hay veces que para programar un método usamos otros métodos que pueden lanzar excepciones, y escribir todo el rato try-catch puede ser un poco pesado. Hay una forma de deshacernos del try-catch, poniendo en la línea donde comienza el método la palabra **throws** y una lista (separada por comas) con todas las excepciones que pueden salirnos en su interior.

Por ejemplo, supongamos que queremos programar un método en la clase Perro llamado **guardar** que guarde en un archivo los valores de las propiedades del Perro. Ese método recibiría un String con la ruta del archivo y puede ocurrir que se lance una IOException si se produce algún error al usar los métodos de Java IO.

Una primera versión de este método podría ser esta:

```
1. public void guardar(String ruta){
2.     try{
3.         PrintWriter p=new PrintWriter(ruta);
4.         p.println(nombre);
5.         p.println(edad);
6.         p.println(raza);
7.         p.println(posicion.toString());
8.         p.println(hambriento);
9.         p.close();
10.    }catch(IOException ex){
11.        System.out.println(ex.getMessage());
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 mostramos un mensaje en pantalla.

Sin embargo, podemos programar **sin el try-catch** si en el diagrama de la clase el método guardar nos lo encontramos definido así:

Perro
+ void guardar(String ruta) throws IOException

```
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. }
```

Al quitar el try-catch del interior de nuestro método todo parece más fácil y cómodo, pero hemos introducido un **importante cambio colateral** en nuestro programa.

Con la versión inicial, un programador de aplicaciones que use la clase Perro llamaría al método guardar de esta forma:

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

En caso de fallo, la clase Perro es la que se encargaría de capturar la excepción y mostrar un mensaje de error (lo normal en estos casos es que la librería no muestre ningún mensaje y devuelva false para que el programador de la aplicación sepa que ha fallado y muestre el mensaje que prefiera).

Sin embargo, al quitar el try-catch del interior del método y ponerlo como throws en el inicio del método, lo que hacemos es pasar la pelota al programador de aplicaciones para que sea él quien capture y gestione el try-catch. O sea, con la nueva versión la llamada al método guardar dentro de un programa se tiene que hacer 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. }
```

Como vemos, la diferencia es muy importante y cobra más importancia cuando se trabaja en equipo con otros compañeros. 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 19 : 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.

6.3.- Métodos que lanzan RuntimeExceptions

Las RuntimeExceptions son excepciones que se deben a cosas que un programador de aplicaciones ha hecho mal. Por ejemplo, una situación donde sería adecuada es llamar al método **setEdad** pasando un número negativo. En su momento lo que hicimos fue asignar una edad por defecto, pero ahora vamos a cambiar ese método para que se lance una excepción.

Las RuntimeException pueden ser lanzadas en cualquier momento y no es necesario escribir el “throws” al comenzar el método. Para lanzar una RuntimeException lo suyo es crear alguna de las excepciones que vienen con Java. Con estas suele ser suficiente:

IllegalArgumentException	Un programador de clases la puede lanzar si detecta que un parámetro recibido por el método es incorrecto.
IllegalStateException	Un programador de clases la puede lanzar si detecta que se llama a un método y el objeto no está preparado para hacer su acción.

En el ejemplo que vamos a hacer, vamos a lanzar una RuntimeException cuando detectamos que se llama al método setEdad con un número negativo. El tipo de RuntimeException que nos interesa es IllegalArgumentException:

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

Para comprobar que funciona correctamente, vamos a hacer un test donde creamos un perro y le intentamos poner una edad negativa:

```
1. @Test
2. public void testSetEdad() throws Exception{
3.     Perro p=new Perro("Tim",4,"San Bernardo");
4.     try{
5.         p.setEdad(-8);
6.         fail("Hemos puesto al perro una edad negativa. Error!");
7.     }catch(IllegalArgumentException e) {
8.         // responde bien al cambio de edad negativa
9.     }
10. }
```

Como vemos, esta forma nos obliga a dejar un catch vacío. La mejoramos así:

```
1. @Test(expected=java.lang.IllegalArgumentException.class)
2. public void testHambre2() throws Exception{
3.     Perro p=new Perro("Tim",4,"San Bernardo");
4.     p.setEdad(-8);
5. }
```

Cuando ejecutamos el test, vemos que todo funciona correctamente.

En general la decisión de si un método debe lanzar una CheckedException o una RuntimeException la toma el analista que diseña la clase.

- Se suelen poner como CheckedException las situaciones que puedan presentarse sin que el programador pueda evitarlas y en las que por tanto, deban programarse los casos en los que se lanzan y los que no.
- Se suelen poner como RuntimeException las situaciones que corresponden a fallos o malos usos de los métodos por parte del programador. Por tanto, el programador debe hacer lo posible para evitarlas y que no se produzcan.

Ejercicio 20 : 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.

Ejercicio 21 : Modifica el segundo método constructor de la clase Oficina para que se lance una `IllegalArgumentException` si se pasa un tipo mayor que 3.

7.- 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.

Por ejemplo, imaginemos que estamos haciendo una clase y queremos ponerle un método que multiplique dos números positivos “a” y “b” sin utilizar el operador *

+ void multiplicar(int a, int b)

Como sabemos de matemáticas, el producto de dos números positivos $a*b$ puede expresarse de esta forma:

$$a * b = a * (b-1) + a$$

Y esto, usando nuestro método multiplicar se puede expresar de esta forma:

$$\text{multiplicar}(a,b) = \text{multiplicar}(a,b-1) + a$$

Como podemos ver, la multiplicación de a y b la podemos conseguir a partir de la multiplicación de “a” por “b-1” y luego sumando “a” nuevamente, tal como nos dice la fórmula anterior (por ejemplo, $5*4$ se puede calcular como $5*3 + 5$). Esto, a la hora de programar nos conduciría a escribir este código fuente:

```
1. public class Calculadora{
2.     public int multiplicar(int a, int b){
3.         return multiplicar(a,b-1)+a;
4.     }
5. }
```

Como vemos, en la línea señalada se está produciendo la llamada del método que estamos programando (multiplicar) a sí mismo, pero con un dato más pequeño (observa que el método recibe el parámetro “b” y la llamada recursiva llama al mismo método pero pasando “b-1”, que es un número más pequeño).

El código anterior aún no funciona, puesto que se forma la siguiente cadena infinita de llamadas recursivas, que nos produciría un **StackOverflowError**:⁹

$\text{multiplicar}(a,b) \rightarrow \text{multiplicar}(a,b-1) \rightarrow \text{multiplicar}(a,b-2) \rightarrow \dots \rightarrow \text{multiplicar}(a,b-10000) \rightarrow \dots$

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 (normalmente, un valor fácil o evidente) y ya no es necesario realizar más llamadas recursivas.

En nuestro ejemplo el caso base lo vamos a poner cuando el número “b”, que es el que va disminuyendo en cada llamada recursiva, se hace 0. En ese momento directamente el resultado de la multiplicación por “b” es 0. El código fuente terminado quedaría así:

```
1. public class Calculadora {
2.     public int multiplicar(int a, int b){
3.         // variable para guardar el resultado
4.         int r=0;
5.         // si b es 0, entonces a * b es 0 (caso base que se calcula directamente sin recursividad)
6.         // si b no es 0, calculamos a * b como a * (b-1) + a llamando al mismo método que programamos
7.         if(b==0){
8.             r=0; // caso base
9.         }else{
10.            r=multiplicar(a,b-1) + a; // llamada recursiva
11.        }
12.        return r;
13.    }
14. }
```

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

Como ejemplo, aquí vemos gráficamente como la llamada a $\text{multiplicar}(5,4)$ daría lugar primero a una serie de llamadas recursivas que al llegar al caso base se convierte en una cadena de resultados que “sube” hasta obtener el resultado final de $\text{multiplicar}(5,4)$

$\text{multiplicar}(5,4) \rightarrow \text{multiplicar}(5,3) \rightarrow \text{multiplicar}(5,2) \rightarrow \text{multiplicar}(5,1) \rightarrow \text{multiplicar}(5,0) = 0$
20 ← 15 ← 10 ← 5 ← 0 (caso base)

Podemos comprobar mediante un test que el método funciona. Por ejemplo, podemos comprobar que el método va bien para todas las tablas de multiplicar del 1 al 10 así:

```
1. @Test
2. public void testSumar() {
3.     for(int i=0;i<10;i++){
4.         for(int j=0;j<10;j++){
5.             // compruebo que i*j coincide con multiplicar(i,j)
6.             assertEquals(i*j, new Calculadora().multiplicar(i, j));
7.         }
8.     }
9. }
```

Aquí hemos puesto el caso base cuando alguno de los dos números es 0 y podríamos preguntarnos qué pasaría si ponemos el caso base en otra situación, como por ejemplo, cuando b, que es el parámetro que va bajando en cada llamada recursiva, es igual a 1.

No hay ningún problema y lo podemos programar también. Se haría así:

⁹ Este error significa que se ha desbordado la capacidad del sistema para realizar llamadas recursivas.

```

1. public int multiplicar(int a, int b){
2.     // variable para guardar el resultado
3.     int r=0;
4.     // si b es 1, entonces a*b es "a", por lo que directamente devolvemos "a" (caso base)
5.     if(b==1){
6.         r=a; // caso base
7.     }else{
8.         r=multiplicar(a,b-1) + a; // llamada recursiva
9.     }
10.    return r;
11. }

```

El problema de este enfoque es que el método funciona bien excepto cuando “b” es 0. En ese caso, el if nunca llega al caso base y se vuelve a formar la secuencia infinita de llamadas recursivas que termina dando un **StackOverflowError**. 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 habituales¹⁰, 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:

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

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.

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).

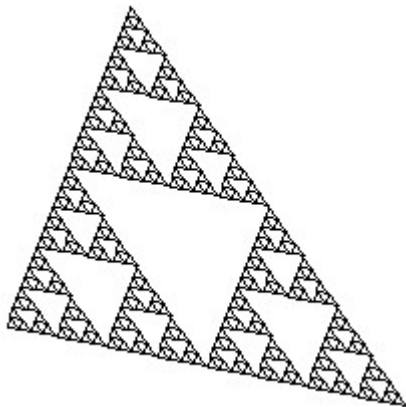
OrdenacionSeleccion
+ void ordenar(int[] array)
- void ordenar(int[] array, int pos)

¹⁰ Cualquier programa recursivo puede convertirse en uno iterativo usando una **pila**

- 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

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.

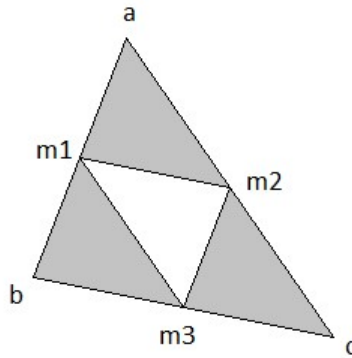
*Las figuras geométricas que se autocontienen de manera infinita (como este dibujo) se denominan **fractales**, y tienen la propiedad de que no son objetos lineales (como las rectas) ni tampoco planos (como el interior de una circunferencia). Su dimensión está entre 1 y 2. La dimensión del triángulo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

Una vez que lo tengas, 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.

8.- 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</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:

```

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. }

```

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

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 getOficinaVacía() + static Oficina getOficinaPequeña() + static Oficina getOficinaMediana() + static Oficina getOficinaGrande()

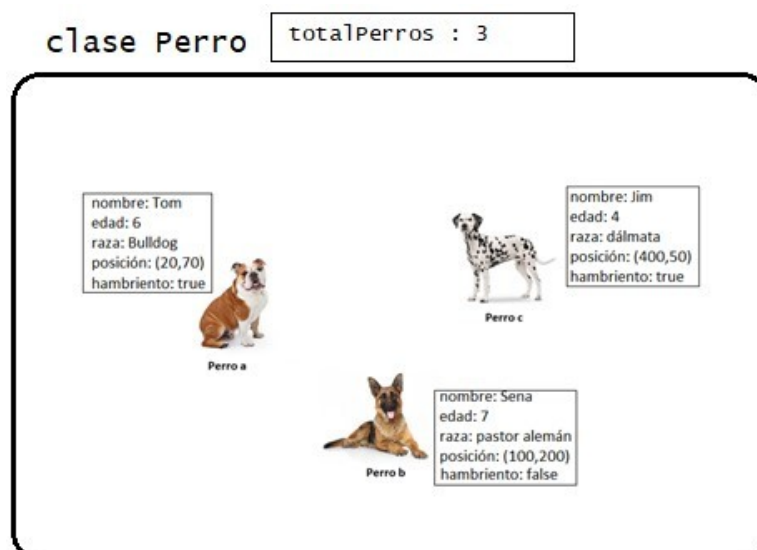
- getOficinaVacía: Devuelve una oficina de tipo 0
- getOficinaPequeña: Devuelve una oficina de tipo 1
- getOficinaMediana: Devuelve una oficina de tipo 2
- getOficinaGrande: Devuelve una oficina de tipo 3

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

9.- 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
- String nombre - int edad - String raza - Point posición - boolean hambriento - static int totalPerros
+ 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 escriben 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, se abre un **bloque inicializador estático**, que es un trozo de código que comienza con la palabra **static**, y en él se escriben las líneas que den el valor inicial a dichas propiedades. Ese trozo de código solo es ejecutado la primera vez que se utilice la clase (crear un objeto, llamar a un método, etc)

```
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.         totalPerros = 0;
10.    }
11.    // resto de la clase
12. }
```

Por último, como hemos puesto modificador private a la propiedad estática totalPerros, añadimos a la clase el método getTotalPerros que devuelve el valor de dicha propiedad. **Los métodos estáticos pueden acceder sin problemas a las propiedades estáticas.**

```
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.         totalPerros = 0;
10.    }
11.     public static int getTotalPerros(){
12.         return totalPerros;
13.    }
14.    // resto de la clase
}
```


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

Matrícula
- static int siguienteNúmeroMatrícula - int númeroMatrícula - String nombreAlumno - String nombreAsignatura
+ Matrícula(String nombreAlumno, String nombreAsignatura) + String getNombreAlumno() + String getNombreAsignatura() + int getNúmeroMatrícula()

- 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 “siguienteNúmeroMatrícula”, que deberá incrementarse una vez asignado dicho número a una matrícula concreta.
- Los getters devuelven los valores de las propiedades.

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.

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
- 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] *****-----

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

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.

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 tema 6 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.

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 rellenado 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 rellenado 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”

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

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

- El constructor crea un reloj que guarda la hora del momento actual.
- 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

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.

10.- 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 y 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.

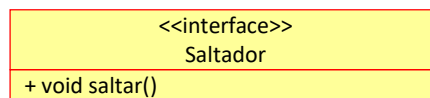
Por ejemplo, una habilidad puede ser “Nadador”. Para que una clase sea considerada un “nadador” bastará con que tenga estos métodos:

<<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:



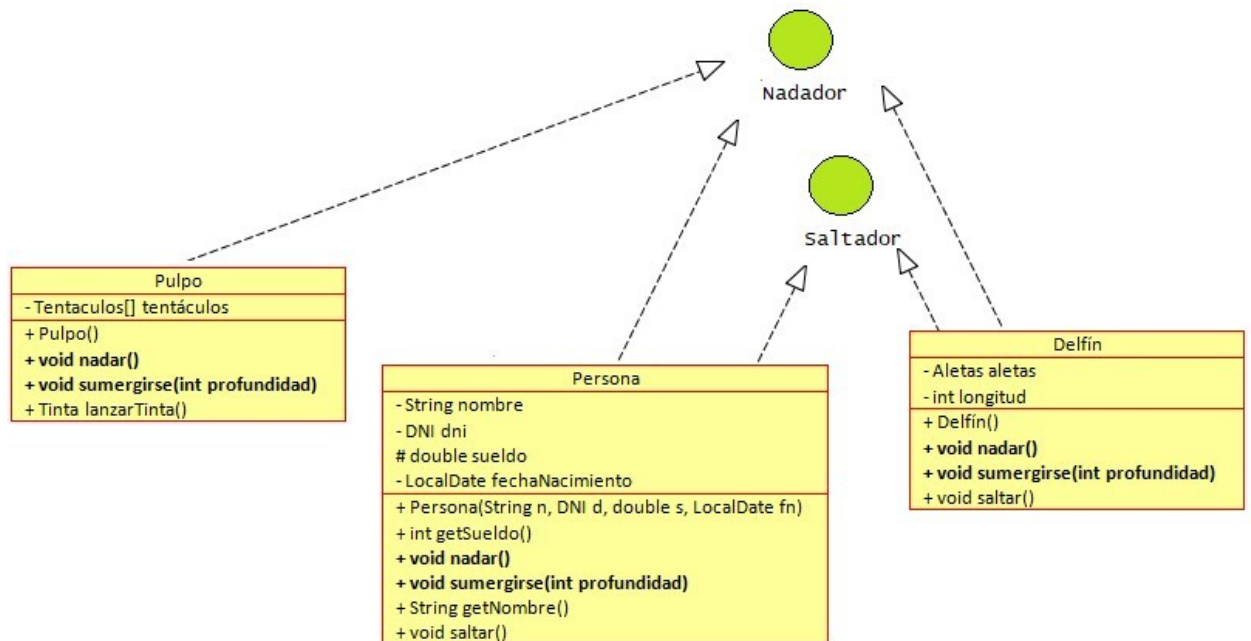
Una clase puede implementar todas las interfaces que necesite. Por ejemplo, supongamos que definimos esta interfaz llamada “Saltador”:



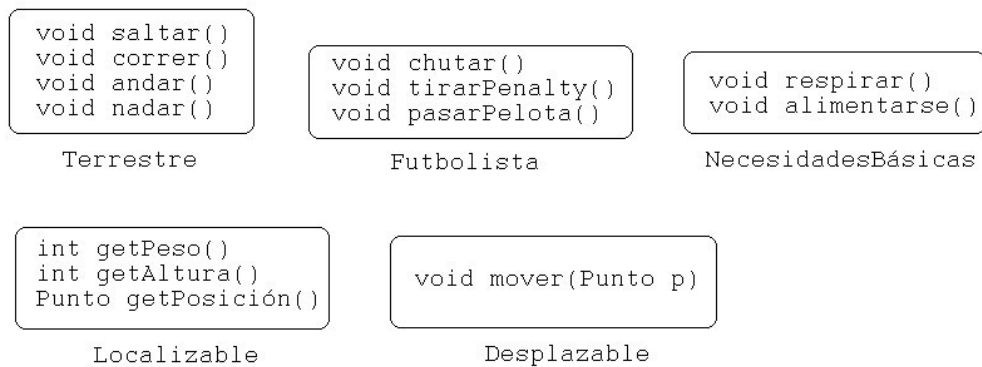
Entonces las clases Persona y Delfín también implementan la interfaz “Saltador”, porque ambas tienen el método saltar, tal y como lo describe la interfaz “Saltador”. En cambio, Pulpo implementa Nadador, pero no implementa Saltador.

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

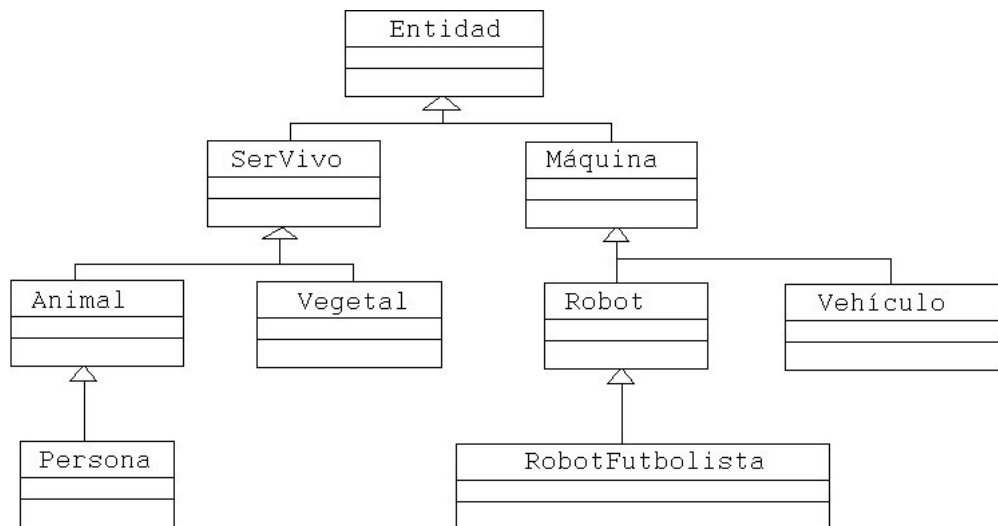
Hay varias formas de representar en un diagrama de clases que una clase implementa interfaces. Una de ellas consiste en representar cada interfaz con una bolita y dibujar una flecha discontinua desde la clase hasta la interfaz:



Ejercicio 36: Supongamos que tenemos las siguientes interfaces:



¿Qué interfaces implementarían las clases del siguiente diagrama?



10.1.- Programación de una interfaz

Programar una interfaz es muy sencillo, ya que lo único que tenemos que hacer es añadir al proyecto una interfaz, seleccionando el paquete con el botón derecho del ratón y seleccionando **New → interface**, según el IDE que estemos manejando.

En la ventana que aparece, pondremos el nombre de la interfaz, y dentro de su cuerpo lo único que haremos será escribir los métodos que definen la interfaz, sin programarlos¹¹.

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

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

¹¹¹¹ Como veremos en el próximo tema, los métodos que se dejan sin programar se denominan **métodos abstractos**.

Java no es capaz de reconocer automáticamente cuándo una clase está implementando una interfaz, por lo que hay que indicarlo en las clases que la implementen. Para ello, nos vamos a la línea **public class** de cada clase y después de su nombre escribimos la palabra reservada **implements** junto con el nombre de la interfaz que implementa.

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

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

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.
6.     public void nadar(){
7.         System.out.println("La persona está nadando");
8.     }
9.     public void sumergirse(int profundidad){
10.        System.out.println("La persona se sumerge "+profundidad+" metros");
11.    }
12.    public void saltar(){
13.        System.out.println("La persona ha saltado");
14.    }
15. }
```

Las interfaces, al igual que las clases, son tipos de datos referencia. Como veremos en el próximo tema, las interfaces van a servir **para almacenar 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 37: Programa la siguiente interfaz, que representa algo que tiene dinero:

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

- 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.

Monedero implements Adinerado
- double dinero
+ Monedero()

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

10.2.- Métodos default

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:

<<interface>>
Recipiente
+ int getCapacidadMaxima() + int getCapacidadActual() + default boolean estaVacio() + default int getPorcentaje()

```

1. public interface Recipiente{
2.     int getCapacidadMaxima();
3.     int getCapacidadActual();
4.     default boolean estaVacio(){
5.         return getCapacidadActual()==0;
6.     }
7.     default int getPorcentaje(){
8.         int p=0;
9.         if(!estaVacio()){
10.            p= 100*getCapacidadActual()/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 hijas. Sin embargo, la interfaz incorpora dos métodos default, que sirven para comprobar si el recipiente está vacío y para calcular su porcentaje. De esta forma, las clases que implementen la interfaz los recibirán tal cual. Por supuesto, las clases hijas pueden sobrescribir dichos métodos si lo desean.

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.

Ejercicio 38 : 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.

11.- 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 aquí¹²:

```
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.         numeroMatricula=nm;
8.         nombre=n;
9.         direccion=d;
10.        repetidor=r;
11.    }
12.    public int getNumeroMatricula(){
13.        return numeroMatricula;
14.    }
15.    public String getNombre(){
16.        return nombre;
17.    }
18.    public String getDireccion(){
19.        return direccion;
20.    }
21.    public boolean esRepetidor(){
22.        return repetidor;
23.    }
24. }
```

¹² 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 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 clase simple que porta una serie de datos de solo lectura (no hay setters). Java automáticamente proporciona:

- Un constructor (llamado **constructor principal**) que asigna directamente los valores en las propiedades (no hay posibilidad de hacer encapsulamiento)
- Un getter para cada propiedad. Es importante destacar que el nombre de los getters no va a empezar por get, sino que el getter ahora se llamará exactamente igual que la propiedad que devuelve¹³.
- Un método toString que muestra en pantalla el valor de las propiedades (y no la cadena clase@hashCode habitual)
- Una implementación de equals que devuelve true cuando dos objetos son iguales (con equals) en todas sus propiedades.
- Una implementación de hashCode compatible con equals.

La forma de programar un record es muy sencilla. Por ejemplo, la clase anterior 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 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 que se ejecuta después del constructor principal, en el que podemos validar que las propiedades tienen un valor correcto y en caso contrario, lanzar alguna excepción:

¹³ 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 !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.

11.1.- 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 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 como objetos inmutables.

Ejercicio 39: 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()

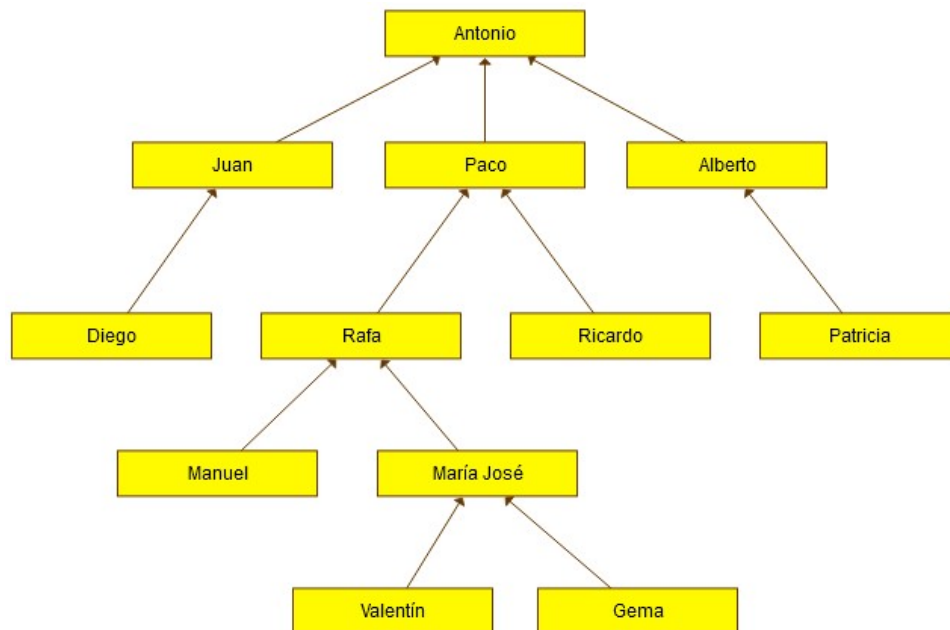
- El constructor principal declara las propiedades y les asigna el valor solamente si son positivas
- El segundo constructor crea un depósito vacío con la capacidad máxima pasada como parámetro.
- añadirLitro devuelve un depósito de agua que tiene las mismas características del que estamos programando, 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

Ejercicio 40: 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)

Haz un programa que cree esta estructura de objetos Empleado:



A continuación, el programa llamará al método mostrarSubordinados de Antonio y deberá verse esto:

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

12.- Enumeraciones

Una enumeración (**enum**) es un tipo de dato referencia¹⁴ 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:

¹⁴ Internamente, al compilar el proyecto las enumeraciones se convierten en clases

- 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 se convierte en 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. }
```

Otra forma de hacer ese método es comprobando si la posición que ocupa nuestra enum en la lista de valores es 5 o 6. Podemos acceder a dicho dato con el método **ordinal()**.

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

Ejercicio 41: Programa la siguiente enum, que representa las posibles canastas que pueden marcarse 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 enumerar al local y al visitante:

<<enum>> TipoEquipo	
+ LOCAL	
+ VISITANTE	

Por último, programa la siguiente interfaz llamada MarcadorBaloncesto, que representa todo 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 mostrarMarcador()

- El método mostrarMarcador muestra en la misma línea, el nombre del equipo local, sus puntos, y luego el nombre del equipo visitante y sus puntos, con este formato:

CB Granada 83 – Estudiantes 75

Ejercicio 42: 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

Ejercicio 43: 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
- 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.

Usa el record 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

13.- Test Driven Development (TDD)

Como hemos visto a lo largo de todo el tema, JUnit es la herramienta que hemos usado todo el rato para hacer los test. Hay otras herramientas, más o menos sofisticadas, pero la idea que persiguen es la misma y JUnit fue la primera que apareció.

Tal fue el éxito de JUnit, que inmediatamente fue portada a otros lenguajes (Python, PHP, etc también disponen de sus versiones) y su creador Ken Beck, desarrolló una nueva forma de programar que se basa en el uso constante de ella. Esta metodología de programación se denomina **TDD (Test-Driven Development, o desarrollo guiado por pruebas)**

La idea principal del TDD es que no debe programarse nada si no hay hecho antes un test que pruebe qué es lo que vamos a hacer.

Por ejemplo, si quiero programar una clase Calculadora para sumar dos números, primero debo hacer un test, aunque **no compile**.

O sea, comenzaríamos escribiendo esto:

```
1. @Test
2. public void testSumar(){
3.     Calculadora c=new Calculadora();
4.     int suma=c.sumar(3,2);
5.     assertEquals(5,suma);
6. }
```

Una vez que está hecho el test, que ni siquiera compila, inmediatamente vamos dando pequeños pasos hasta que el test pase al estado green (la barra verde).

Por ejemplo, el primer paso sería conseguir a toda costa que el test compile:

```
1. public class Calculadora {
2.     public int sumar(int a, int b){
3.         return 0;
4.     }
5. }
```


Sin embargo, aún la barra sigue en rojo, lo que indica que debemos seguir trabajando. No obstante, el hecho de que compile ya indica avance.

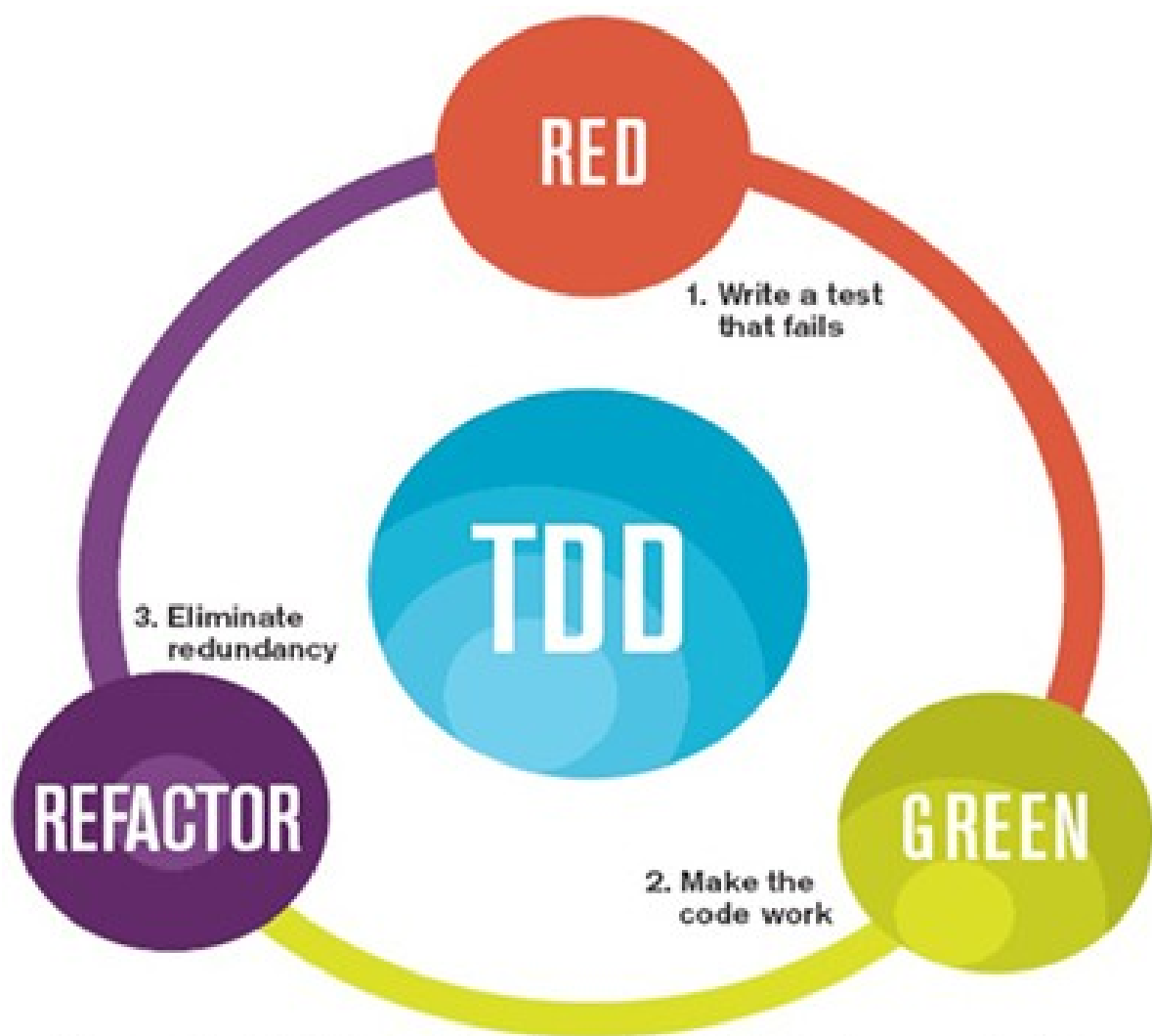
Lo siguiente es dar otro pequeño paso hasta conseguir que el test funcione correctamente.

```
1. public class Calculadora {  
2.     public int sumar(int a, int b){  
3.         return a+b;  
4.     }  
5. }
```

Una vez que el test funciona, la clase debe **refactorizarse**, que consiste en optimizar su diseño y su funcionamiento con vistas a facilitar el mantenimiento posterior. En este ejemplo tan sencillo que hemos visto, no hay ninguna refactorización posible.

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 test de las clases (aunque no compilen).
- 3) Se programan esqueletos (**stub**) de las clases que se van a programar. Un esqueleto es una versión inicial de la clase en la que los métodos tienen lo mínimo para compilar, pero no hacen nada útil. Normalmente lo que hacen es simplemente lanzar una runtime exception llamada **UnsupportedOperationException** o proporcionar un valor por defecto. Como es lógico, los esqueletos de las clases no superan las pruebas (**estado red**).
- 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.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."