

Fundamentos de Programación

Tema 5: Programación avanzada de clases

Contenidos

1.- Introducción	2
2.- Encapsulamiento.....	2
3.- Herencia	7
4.- Sobreescritura de métodos.....	16
5.- Métodos de la clase Object.....	21
6.- La interfaz Comparable<T>	25
7.- Clases abstractas.....	26
8.- Clases anónimas.....	30
9.- Polimorfismo	32
10.- Clases finales	38
11.- Limitaciones de la herencia.....	40
12.- Composición y agregación	41
13.- Patrones de diseño	43

Tema

5

PROGRAMACIÓN AVANZADA DE CLASES

La programación orientada a objetos que hemos estudiado nos permite realizar muchos programas. Sin embargo, para conseguir un nivel de calidad profesional y los máximos beneficios es necesario utilizar conceptos avanzados que vamos a estudiar en este tema, y conocer las buenas prácticas que se han desarrollado en los más de 20 años de experiencia de la programación Java.

1.- Introducción

Los programas bien contruidos se diferencian de los demás en que son fáciles de mantener y de ampliar. Cuando un programa ha sido bien diseñado, es mucho más fácil añadirle nuevas características y corregir sus errores, lo cual se traduce en una mayor facilidad de mantenimiento, y ya sabemos que esta es una cualidad muy apreciada en todo programa.

Cuando se usan los conceptos avanzados de la programación orientada a objetos, se favorece la creación de aplicaciones fáciles de mantener y de ampliar. Por este motivo, en la programación profesional se utilizan constantemente los siguientes principios:

- Encapsulamiento
- Herencia
- Polimorfismo

El uso de estas técnicas proporciona la máxima potencia a la programación orientada a objetos, favoreciendo la reutilización del código programado, el reparto de las tareas en equipo y la escalabilidad de los programas. Su único inconveniente es que diseñar un programa haciendo uso de estos principios es más difícil que hacerlo sin ellos. No obstante, el tiempo invertido en usarlos es algo que sin duda merece la pena.

Al contrario de lo que ocurre en otros lenguajes, Java tiene la ventaja de que es un lenguaje veterano que se ha utilizado en cientos de proyectos en lo que va de siglo. Esto hace que los puntos fuertes y débiles de Java sean bien conocidos, y que se conozca “lo que funciona” y lo que “no funciona” cuando se va a diseñar un programa para este lenguaje.

2.- Encapsulamiento

El encapsulamiento es una característica fundamental que debe tenerse en cuenta siempre que se va a diseñar una clase. En realidad es algo con lo que ya hemos trabajado de forma implícita durante todo el curso, sin mencionarlo expresamente.

El encapsulamiento es el principio según el cual, las propiedades de un objeto deben ser privadas, y los programadores de aplicaciones solo pueden modificarlas mediante métodos.

Por tanto, una clase “está bien encapsulada” cuando tiene sus propiedades privadas y ofrece métodos (getters por ejemplo) para acceder y cambiar su valor de forma controlada.

El encapsulamiento es el concepto opuesto a que las propiedades sean públicas y los programadores de la aplicación puedan acceder directamente a ellas y asignarle valores sin control. Por ejemplo, la siguiente clase estaría mal encapsulada, puesto que las propiedades son públicas y en el programa de la derecha vemos cómo se pueden modificar indebidamente:

```

01. public class Persona{
02.     public String nombre;
03.     public int edad;
04. }

01. public class Prueba{
02.     public static void main(String[] args){
03.         Persona p=new Persona();
04.         p.edad=-238;
05.     }
06. }

```

En cambio, la siguiente versión de la clase si está bien encapsulada: las propiedades están ocultas al programador de la aplicación, que debe usar métodos para modificarlas:

```

01. public class Persona{
02.     private String nombre;
03.     private int edad;
04.
05.     public void setEdad(int e){
06.         edad=e;
07.     }
08. }

01. public class Prueba{
02.     public static void main(String[] args){
03.         Persona p=new Persona();
04.         p.setEdad(10);
05.     }
06. }

```

El encapsulamiento nos permite **mantener los invariantes**, que son las relaciones entre las propiedades que deben mantenerse siempre. Por ejemplo, en la siguiente clase Rectángulo nos encontramos como propiedades la base, la altura y el área. Como sabemos, el área debe ser igual al producto de la base por la altura:

Rectángulo
+ int base + int altura + int area
+ Rectangulo(int base, int altura) + void setBase(int b) + void setAltura(int a) + int getArea() + int getBase() + int getAltura()

La clase Rectángulo, tal y como está escrita está mal encapsulada, y eso hace que un programador de aplicaciones pueda cambiar el área libremente así:

```

1. Rectangulo r = new Rectangulo(40,50);
2. r.area=10000;

```

Al hacer esto, el objeto rectángulo ya queda en un estado inconsistente porque el área no es la que corresponde a un triángulo con base 40 y altura 50. La relación entre el área, base y altura es un “invariante” que podemos conservar si encapsulamos la clase:

```

1. public class Rectangulo{
2.     private int base;
3.     private int altura;
4.     private int area;
5.     public Rectangulo(int base, int altura){
6.         if(base<0 || altura<0){
7.             throw new IllegalArgumentException("La base y la altura deben ser positivos");
8.         }
9.         this.base=base;
10.        this.altura=altura;
11.        this.area=base*altura;
12.    }
13.    public void setBase(int b){
14.        base = b;
15.        area = base*altura;    // si cambia la base, cambia también el área
16.    }
17.    public void setAltura(int a){
18.        altura = a;
19.        area = base*altura;    // si cambia la altura, cambia también el área
20.    }
21.    public int getArea(){
22.        return area;
23.    }
24. }

```

Aunque el encapsulamiento es muy necesario, cuando trabajamos en un equipo con más compañeros, a veces puede ser conveniente “relajar” un poco el nivel de encapsulamiento, para que algunos de nuestros compañeros tengan permiso para usar las propiedades de una clase. Eso lo vamos a conseguir con el *modificador de acceso por defecto* (también llamado modificador *package* o modificador *package-private*).

2.1.- El modificador de acceso por defecto

Cuando una propiedad o método tiene el modificador de acceso por defecto (en Java este modificador se consigue no poniendo ningún modificador a la propiedad), **se podrá acceder a ella desde todas las clases que estén dentro de su mismo paquete.** Por ejemplo, supongamos que estamos haciendo la siguiente clase Empleado:

```

1. package daw.oficina;
2. public class Empleado{
3.     private String nombre;
4.     int sueldo;
5.     public Empleado(String n, int s){
6.         nombre=n;
7.         sueldo=s;
8.     }
9. }

```

En esta clase la propiedad “sueldo” tiene modificador por defecto, lo que significa que cualquier persona que programe una clase dentro del paquete **daw.oficina** puede acceder y cambiar libremente el sueldo de cualquier objeto de la clase Empleado.

Por ejemplo, la siguiente clase “Jefe”, del paquete daw.oficina lo puede hacer así:

```

1. package daw.oficina;
2. public class Jefe {
3.     public void subirSueldo(Empleado empleado, double cantidad){
4.         empleado.sueldo+=cantidad;
5.     }
6. }

```

Como podemos ver, el método subirSueldo recibe un objeto empleado (creado por la aplicación) y una cantidad. Dentro del método, se accede a la propiedad “sueldo” del empleado y se le incrementa su valor. Esto es posible hacerlo porque “sueldo” tiene modificador de acceso por defecto, que permite ser accedida desde cualquier clase que se encuentre en el paquete daw.oficina. Si la clase Jefe hubiese estado en otro paquete que no fuese daw.oficina, la línea 4 del código anterior nos daría error de compilación por intentar acceder a una propiedad para la que no tendríamos permiso.

Cuando se usa el modificador por defecto, la clase ya no está completamente encapsulada, y deberemos fiarnos de que los compañeros que programen clases en el mismo paquete no le den valores indebidos. En caso de que sospechemos que las personas que van a usar nuestra clase puedan modificar malamente las propiedades, deberemos ocultarlas completamente (modificador private) y poner los setters correspondientes.

Por último, indicamos que en los diagramas de clases el modificador por defecto se suele representar con el signo ~ , debido a que Java no tiene un símbolo propio para él y es muy fácil que a veces por error se nos olvide y lo confundamos. Poniendo ese signo en el diagrama de clases, indicamos expresamente que queremos usar el modificador por defecto.

Ejercicio 1 : ¿En qué situaciones pondrías a una propiedad el modificador por defecto?

- a) Estoy haciendo un programa yo solo y hago muchas clases en el mismo paquete.
- b) Hago una librería de clases para ponerla en Internet y que todo el mundo la use.
- c) Le dejo a mi mejor amigo una clase que he hecho para que la use en su programa.
- d) Estoy haciendo una clase y mis compañeros van a hacer clases en el mismo paquete.

2.2.- El modificador de acceso por defecto en una clase

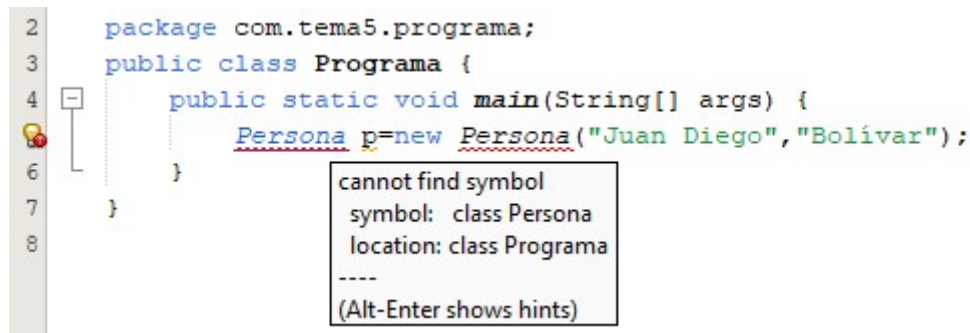
Todas las clases que hemos programado hasta este momento son siempre públicas y por tanto, cualquier programador puede usarlas. Sin embargo, un hecho no tan conocido es que podemos poner a una clase el modificador por defecto. Por ejemplo, a la siguiente clase Persona:

```

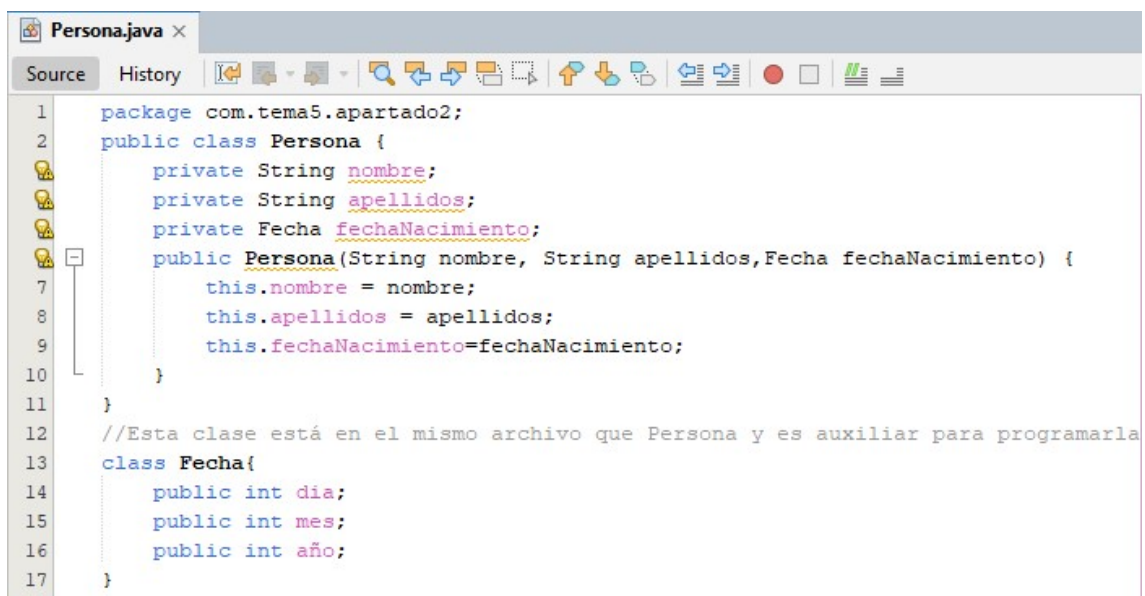
1. package com.tema5.apartado2
2. class Persona{ // observa que no es public
3.     private String nombre;
4.     private String apellidos;
5.     public Persona(String n, String a){
6.         nombre=n;
7.         apellidos=a;
8.     }
9. }

```

¿Qué ocurre cuando una clase tiene este modificador? Pues que solo es vista por las clases del mismo paquete. Observa que la clase `Persona` está en el paquete **com.tema5.apartado2**. Solamente las clases de ese paquete podrán crear y usar objetos de la clase `Persona`. Si hacemos un programa en otro paquete, nos dará error de compilación:



Las clases que tienen modificador por defecto son clases auxiliares que actúan como complementos de las clases principales que el analista quiere que sean las que usen los programadores de aplicaciones. De hecho, en un archivo pueden coexistir una clase pública y todas las clases auxiliares con modificador por defecto que hagan falta, aunque no se recomienda hacer esto.



Ejercicio 2: Consulta el diagrama de clases **Persona** y programa la interfaz `Persona`, la enum `EstadoCivil`, el record `Direccion` y la clase `ImplementacionPersona` (observa su modificador de acceso), teniendo en cuenta la documentación que aparece en la página siguiente a su diagrama.

Ejercicio 3: Consulta el diagrama de clases **Persona** y programa la clase `PersonaBuilder` tal y como se explica en la correspondiente documentación. Después, haz la clase **Programa** que aparece indicada en el mismo diagrama y documentación.

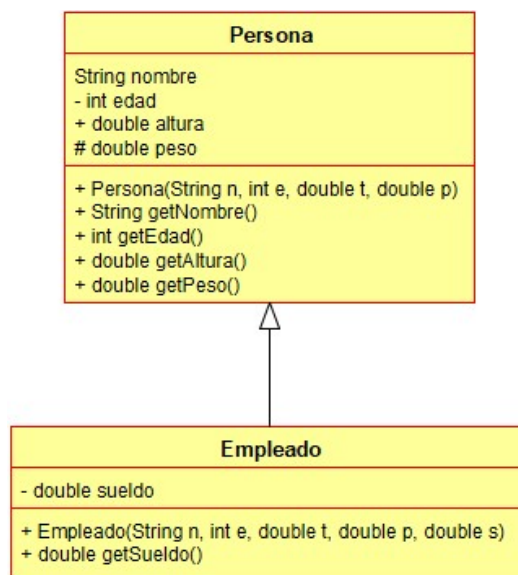
3.- Herencia

La herencia es uno de los conceptos más poderosos de la programación orientada a objetos porque permite que el código que programamos en una clase padre se propague automáticamente a todas sus clases hijas sin tener que reescribirlo en ellas. Por tanto, es una herramienta fundamental para conseguir reutilización de código.

Como ya sabemos, la herencia es la relación que se da entre dos clases A y B cuando podemos afirmar que A “es un” B. También sabemos que en el lenguaje Java:

- La herencia entre clases siempre es simple, lo que significa que una clase solo puede tener como mucho una clase padre.
- Las interfaces admiten herencia múltiple, y lo que significa que una interfaz puede tener muchas interfaces padres.
- Toda clase hereda automáticamente de Object.

En este apartado vamos a estudiar cómo podemos hacer que una clase que nosotros programamos tenga como padre una clase distinta de Object, lo que nos permitirá programar relaciones de herencia como esta:



Vamos a ver cómo se programaría la clase **Empleado** del diagrama anterior. Suponemos que la clase **Persona** ya está programada y comenzamos creando el archivo de código fuente para la clase **Empleado**:

```
1. public class Empleado{
2.
3. }
```

En este momento, la clase **Empleado** tiene como clase padre a **Object**, que es por defecto la clase padre de todas las clases de Java. Vamos a ver cómo cambiar la clase padre de **Empleado** para que sea **Persona**, en lugar de **Object**.

Basta con utilizar la palabra reservada **extends** para indicar que la clase Empleado va a ser una clase hija de la clase Persona, así:

```
1. public class Empleado extends Persona{
2.
3. }
```

Solo por hacer esto la clase Empleado se convierte en una clase hija de la clase Persona. Esto significa que la clase Empleado acaba de adquirir las propiedades nombre, edad, altura, peso y los métodos getNombre, getEdad, getAltura y getPeso. Sin embargo, en la clase hija Empleado solo podremos usar aquellas propiedades y métodos cuyo modificador de acceso en la clase padre Persona nos de permiso. Las reglas de visibilidad son:

- Podemos ver sin problema todas las propiedades y métodos **public**
- Podemos ver sin problema todas las propiedades y métodos **protected**
- No podemos ver las propiedades y métodos que tengan modificador por defecto, salvo que la clase hija (Empleado) esté en el mismo paquete que la clase padre (Persona).
- Nunca podremos ver las propiedades ni métodos **private** (pero sabemos que están ahí y que han sido heredados. Lo que ocurre, es que no tenemos permiso para acceder a ellos cuando programamos la clase hija Empleado).

En consecuencia, en la clase Empleado tenemos permiso para acceder a las propiedades “altura” y “peso”. Si además, la clase Empleado estuviese en el mismo paquete que la clase Persona entonces también podríamos acceder a la propiedad “edad”. En caso contrario, la única forma de poder conocer el valor de la edad sería usando método getEdad, que ha sido heredado por la clase Empleado.

Como ejemplo, vamos a añadir a la clase Empleado un método getIMC que calcula el valor del IMC del empleado¹.

```
1. public class Empleado extends Persona{
2.
3.     public double getIMC(){
4.         return peso/(altura*altura);
5.     }
6. }
```

Aquí vemos cómo podemos utilizar sin problemas las propiedades “peso” y “altura” aunque no las hayamos creado en la clase Empleado. Es posible hacer esto porque dichas propiedades han sido heredadas y además el modificador de acceso que tienen en la clase Persona nos da permiso para acceder a ellas cuando programamos la clase Empleado.

3.1.- El constructor de la clase hija

Cuando se hace herencia, se heredan todas las propiedades y métodos que tiene la clase padre, **excepto los constructores**. Por tanto, la clase Empleado debe tener su propio método constructor, que es el que viene en su diagrama de clases.

¹ Realmente este es un método que debería estar en la clase Persona, ya que todas las personas tienen un IMC, no solo las que son empleados. En este ejemplo lo ponemos solo en la clase Empleado para que se vea cómo podemos acceder a las propiedades heredadas con modificadores public y protected.

Como ya sabemos del tema anterior, la misión del método constructor es dar el valor inicial de las propiedades. Un examen detallado del constructor de la clase hija nos permite identificar que recibe todas las propiedades de Persona, y además, la propiedad “sueldo”.

+ Empleado(String n, int e, double t, double p, double s)

Propiedades de un empleado
como persona

Propiedades exclusivas
del empleado

Esto tiene sentido, porque como un Empleado “es una” Persona, es lógico que si queremos crear un Empleado, tengamos que indicar también el valor que tienen que tener las propiedades que hereda como Persona (recordemos que todas las propiedades siempre se heredan, aunque después algunas queden ocultas por su modificador de acceso).

Para inicializar las propiedades heredadas de la clase padre es **obligatorio** llamar a un constructor de la clase padre para que haga ese trabajo. Esto se consigue con la palabra reservada **super**, que lo que hace es invocar al constructor de la clase padre que queramos. Debemos pasar entre paréntesis los parámetros que nos pida dicho constructor, y también hay tener en cuenta que:

- La llamada a super debe ser siempre la primera línea del constructor de la clase hija.
- Podemos omitir la llamada a super si lo que tenemos que escribir es **super()**
- Tampoco es necesario hacer la llamada a super cuando estamos haciendo un constructor secundario que llama a otro de su clase con **this**

Por tanto, la programación del constructor principal de la clase Empleado sería así:

```
1. public class Empleado extends Persona{
2.     private double sueldo;
3.     public Empleado(String n, int e, double t, double p, double s){
4.         super(n,e,t,p); // llama al constructor de la clase padre
5.         sueldo=s;      // inicializa la propiedad de la clase hija
6.     }
7. }
```

Hay veces que no es necesario poner la llamada a super. Por ejemplo, supongamos que queremos añadir a la clase Empleado este constructor secundario:

- + Empleado(String n, double t, double p) → Crea un empleado con el nombre, altura y peso que se pasa como parámetro, con edad 18 años y sueldo 1000 €.

Podríamos programar dicho constructor “pasando” del constructor principal, así:

```
1. public class Empleado extends Persona{
2.     private double sueldo;
3.     public Empleado(String n, int e, double t, double p, double s){
4.         super(n,e,t,p); // llama al constructor de la clase padre
5.         sueldo=s;      // inicializa la propiedad de la clase hija
6.     }
7.     public Empleado(String n, double t, double p){
8.         super(n,18,t,p); // llama al constructor de la clase padre
9.         sueldo=1000;    // inicializa la propiedad de la clase hija
10.    }
11. }
```

Pero como ya sabemos, tiene más fácil mantenimiento llamar al constructor principal usando la palabra this. Lo haríamos así:

```

1. public class Empleado extends Persona{
2.     private double sueldo;
3.     public Empleado(String n, int e, double t, double p, double s){
4.         super(n,e,t,p); // llama al constructor de la clase padre
5.         sueldo=s;       // inicializa la propiedad de la clase hija
6.     }
7.     public Empleado(String n, double t, double p){
8.         this(n,18,t,p,1000);
9.     }
10. }

```

De esta forma, no es necesario escribir super en el nuevo constructor, porque lo que hace es llamar a this, y el constructor principal sí que comienza con la palabra super. En general, podemos encadenar llamadas sucesivas a constructores con this, de forma que el último de ellos realice una llamada a super.

Ejercicio 4 : Consulta el diagrama de clases **Edificios** y su documentación para programar las clases Edificio y Rascacielos.

Ejercicio 5 : Siguiendo con el mismo proyecto y diagrama del ejercicio anterior, programa las clases Hotel y CasaRural.

Ejercicio 6 : Consulta el diagrama de clases **Trabajadores** y programa la interfaz Teclado y las clases TecladoJava y TecladoConsolaDAW, de acuerdo a su documentación

Ejercicio 7 : Siguiendo con el mismo proyecto y diagrama del ejercicio anterior, programa las clases Trabajador, TrabajadorTecleante y Administrativo.

Ejercicio 8 : Programa la clase Programador del diagrama de los ejercicios anteriores.

3.2.- El modificador de acceso protected

En el apartado anterior vimos que cuando se hace herencia el modificador de acceso protected hace que las propiedades y métodos que lo llevan puedan ser vistos en clases hijas.

Sin embargo, el modificador protected también tiene un uso menos conocido (de hecho, muchos programadores profesionales lo ignoran), y es que en Java también incorpora la funcionalidad del modificador por defecto. Es decir, las propiedades y métodos protected pueden ser vistos en las clases hijas y también en las clases que están en el mismo paquete.

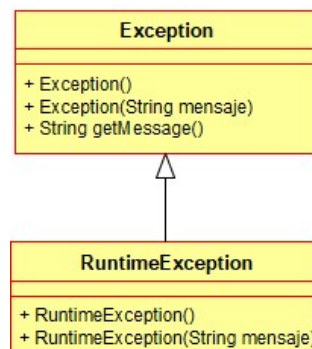
La siguiente tabla resume los modificadores de acceso:

¿Es posible acceder a una propiedad o método ...	private	defecto	protected	public
... dentro de su propia clase?	✓	✓	✓	✓
... al programar una clase que esté en su mismo paquete?	✗	✓	✓	✓
... al programar una clase hija?	✗	✗	✓	✓
... desde cualquier lugar?	✗	✗	✗	✓

A la vista de esta tabla, podemos ver que el modificador de acceso más restrictivo (según la cantidad de personas que pueden ver la propiedad o método) sería `private`, a continuación el modificador por defecto, luego el `protected` y por último el `public`, que daría acceso a todos.

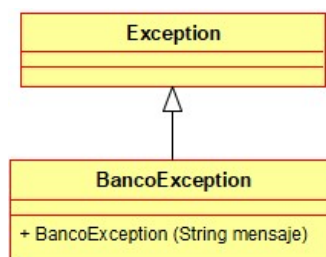
3.3.- Creación de excepciones propias

Es muy habitual que en los proyectos los desarrolladores creen sus propios tipos de excepciones. De esa forma, pueden ofrecer un tratamiento diferenciado a las situaciones que se presentan en cada proyecto concreto. Tenemos dos clases para poder hacer esto:



- **Exception** es la clase padre de todas las excepciones en Java. Representa una `CheckedException`, y clases como `IOException` o `SQLException` son hijas de ella.
- **RuntimeException** es la clase padre de todas las `RuntimeExceptions`. Clases como `NullPointerException` o `ArrayIndexOutOfBoundsException` son hijas de ella.

Para crear tipos personalizados de excepciones, bastará con heredar de esas clases. Si queremos crear una nueva `CheckedException` heredaremos de `Exception`, y si queremos crear una nueva `RuntimeException` heredaremos de `RuntimeException`. Por ejemplo, vamos a programar una clase `BancoException` que representa una `checked exception`:



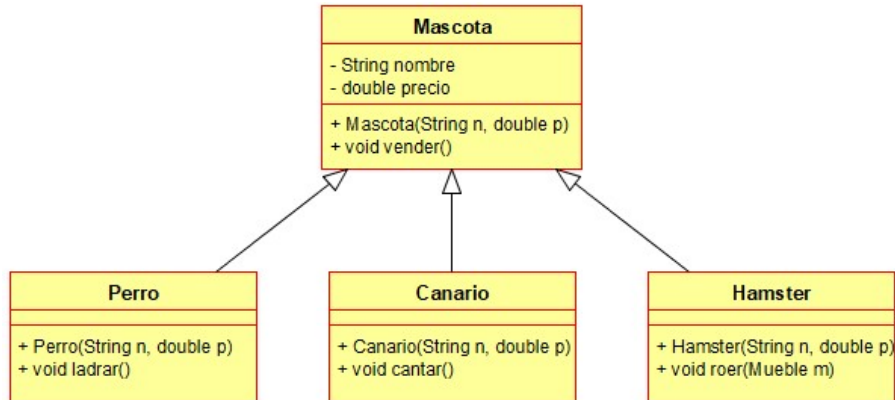
Basta con crear una clase hija de `Exception` y usar el constructor de `Exception` que recibe el mensaje asociado:

```
1. public class BancoException extends Exception{
2.     public BancoException(String mensaje){
3.         super(mensaje);
4.     }
5. }
```

Ejercicio 9: Programa todas las clases del diagrama de clases **Adivinanzas**

3.5.- El operador instanceof

Cuando tenemos un árbol de herencia hay veces en las que creamos objetos de una clase, pero los guardamos en una variable de su clase padre. Por ejemplo, supongamos que al programar una aplicación para una tienda de animales tenemos este diagrama:



Para hacer una lista que reuniese a todos los animales de la tienda deberíamos usar el tipo de dato `List<Mascota>`, y rellenarla con objetos concretos. Por ejemplo, así:

```
1. List<Mascota> tienda = new List<>();
2. tienda.add( new Perro("Tom",300) );
3. tienda.add( new Canario("Tim",20) );
4. tienda.add( new Perro("Bob",300) );
5. tienda.add( new Hamster("Jhon",6) );
```

Si ahora un cliente quiere comprar una mascota, deberemos sacarla de la lista y guardarla en una variable de tipo **Mascota**, que es lo que sabemos que guarda la lista. Por ejemplo, supongamos que el cliente escribe por teclado la posición de la mascota que quiere comprar:

```
1. System.out.println("Escriba la posición de la mascota que quiere comprar");
2. int posicion = sc.nextInt();
3. Mascota mascotaComprada = tienda.get(posicion);
```

En la línea señalada, la mascota que sacamos debe ser guardada en un objeto de la clase **Mascota**. Podemos llamar a su método `vender` (lo tienen todas las mascotas) y con esto, la mascota estará vendida.

```
1. Mascota mascotaComprada = tienda.get(posicion);
2. mascotaComprada.vender();
```

Es muy importante darse cuenta de que hemos podido vender la mascota de forma independiente de su tipo concreto, ya que el método `vender` lo tiene la clase **Mascota** y no necesitamos nada más para realizar la venta. En muchos programas esto va a ser así y **no** será necesario recuperar el objeto concreto.

Sin embargo, hay situaciones donde **si** puede interesarnos recuperar el objeto concreto. Por ejemplo, supongamos que antes de vender un Canario, queremos escucharlo. En ese caso, no podemos llamar al método `"cantar"` sobre el objeto **Mascota**, porque `"cantar"` es un método que solo se encuentra en la clase **Canario**.

La solución consiste en comprobar si la mascota sacada guarda en realidad un Canario usando la palabra **instanceof**, que devolverá true si la mascota guarda un objeto Canario. Además, en las últimas versiones de Java, la palabra instanceof sirve para que, en caso de que el objeto Mascota sea un Canario, convertir dicho objeto Mascota en el correspondiente objeto Canario²

La sintaxis para comprobar si la variable “mascotaComprada”, cuyo tipo es Mascota, realmente guarda en su interior un objeto de la clase Canario, y además, obtener una variable llamada “canario” con la correspondiente conversión es esta:

mascotaComprada instanceof Canario canario

variable Mascota

clase (o interfaz) a comprobar

variable con la conversión

El operador **instanceof** devuelve true si la variable escrita en su lado izquierdo es un objeto de la clase (o interfaz) que se escribe en el lado derecho. Es importante destacar que no se admiten tipos básicos, y que esta comprobación solo tiene sentido para tipos referencia.

Como instanceof devuelve un boolean, se suele escribir dentro de un if, así:

```
1. if(mascotaComprada instanceof Canario canario){
2.     // acciones en caso de haber seleccionado un Canario
3. }else {
4.     mascota.vender();
5. }
```

Dentro del if podemos usar la variable “canario”, que guarda el mismo objeto que hay dentro de “mascotaComprada”, pero convertido en un objeto de la clase Canario. Observa que el objeto “canario” si tiene el método cantar porque su tipo es Canario. En cambio, mascotaComprada no tiene ese método porque su tipo es Mascota (aunque en realidad guardara dentro un Canario).

```
1. if(mascotaComprada instanceof Canario canario){
2.     canario.cantar();
3.     System.out.println("¿Te quedas con el canario?");
4.     String respuestaCliente = sc.nextLine();
5.     if(respuestaCliente.equals("si")) {
6.         c.vender();
7.     }
8. }else {
9.     mascota.vender();
10. }
```

Como resumen de este apartado, debemos quedarnos con las siguientes ideas:

- Es algo muy habitual tener objetos de clases hijas que son guardadas en variables de una clase padre (aquí hemos visto como podemos guardar objetos Perro, Canario, Hamster en objetos de su clase padre Mascota).

² Esto se llama **pattern matching for instanceof** y apareció en Java 17. En las versiones anteriores, si instanceof nos devolvía true, teníamos que hacer un casting para convertir el objeto de la clase padre en el de la clase hija, lo que es una operación propensa a conversiones incorrectas (**ClassCastException**)

- En muchos casos, los métodos de la clase padre serán suficientes para lo que tengamos que hacer y no hará falta nada más (lo que pasaba al vender una mascota)
- En otros casos, deberemos recuperar un objeto con el tipo de dato original de los objetos usando el operador **instanceof**.
- El operador instanceof solo se puede aplicar a tipos de datos referencia y nunca a básicos.

Ejercicio 10: Consulta el diagrama **Equipo de fútbol** y programa las clases Empleado, CuerpoTecnico y CabreoException

Ejercicio 11: Siguiendo el diagrama de clases del ejercicio anterior, programa las clases EmpleadoPrimable, Futbolista y Entrenador, descritas así:

Ejercicio 12: Siguiendo el diagrama de los ejercicios anteriores, haz la clase EquipoFutbol

3.6.- Desestructuración de los records

Como muestra de que Java está actualizándose dentro de sus capacidades para adaptarse a lo que sucede en otros lenguajes, Java 19 (septiembre 2022) ha añadido al lenguaje una versión preliminar³ de los **records patterns**, que viene a ser lo que en lenguajes como Python es la **desestructuración de objetos**⁴, aunque en Java solo sirve para desestructurar records cuando son detectados con el operador instanceof.

Desestructurar un objeto consiste en guardar en variables sus propiedades o datos internos utilizando la asignación.

Java no admite la desestructuración de objetos, así que para ver cómo funciona acudiremos a otro lenguaje. Por ejemplo, en Python podemos crear una lista de 3 elementos y “desestructurarla” guardando en variables sus datos así:

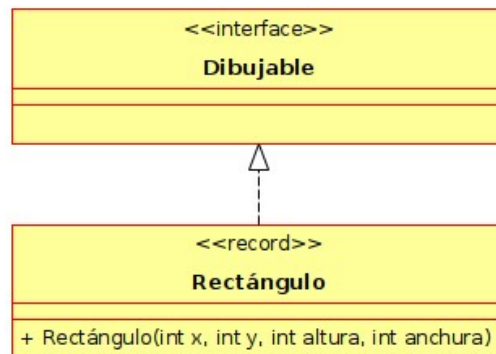
```
1. nombres = ["Susana", "Patricia", "Matilde"]
2. a,b,c = nombres # desestructuramos la lista guardando en a,b y c sus elementos
3. print(a)      # imprime Susana
4. print(b)      # imprime Patricia
5. print(c)      # imprime Matilde
```

Vamos a poner un ejemplo en el que veremos cómo Java 19 admite de forma experimental desestructurar records cuando estos son detectados con instanceof.

³ Como Java se actualiza cada 6 meses, suele ofrecer versiones preliminares (Preview) de las nuevas características del lenguaje, con objeto de obtener feedback. Estas características preliminares pueden ser modificadas en las siguientes versiones, por lo que no se recomienda su uso en producción. De hecho, por defecto están deshabilitadas y hay que añadir la opción **--enable-preview** al compilar y ejecutar el programa.

⁴ Actualmente Python es el lenguaje que mejor soporte da al concepto de desestructuración de objetos.

Supongamos que estamos haciendo un programa y tenemos una interfaz que define la habilidad que tiene un objeto para ser dibujado. Como un rectángulo es una figura que tiene esa habilidad, podemos programar el siguiente record:



Ahora supongamos que en algún lugar de nuestro programa tenemos que hacer un método que dibuje un objeto Dibujable usando un objeto Graphics (como el que nos daría la CapaCanvas de la ConsolaDAW) que recibe como parámetro. Para programar este método primero hay que detectar con instanceof el tipo de figura, para poder dibujarla de acuerdo a su forma:

```

1. public static void dibujar(Dibujable figura, Graphics g){
2.     // primero detecto el tipo de figura usando instanceof
3.     if( figura instanceof Rectangulo r){
4.         // en caso de un rectángulo, la pinto con el Graphics
5.         g.drawRect(r.x(),r.y(),r.anchura(),r.anchura());
6.     }
7. }
  
```

En esta situación podemos **desestructurar** el rectángulo, mediante la asignación automática de las propiedades en variables, así:

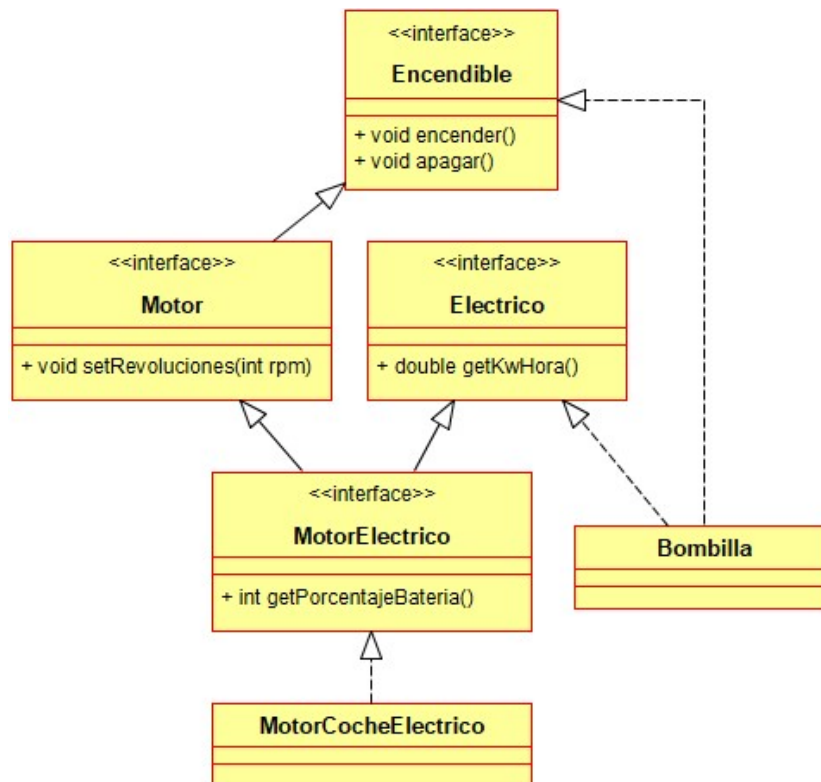
```

1. public static void dibujar(Dibujable figura, Graphics g){
2.     // primero detecto el tipo de figura usando instanceof
3.     if( figura instanceof Rectangulo(int x, int y, int w, int h) ){
4.         // en caso de un rectángulo, la pinto con el Graphics
5.         // usando las variables obtenidas al desestructurar el record
6.         g.drawRect(x,y,w,h);
7.     }
8. }
  
```

3.7.- Herencia de interfaces

Al igual que en las clases existe el concepto de herencia, en las interfaces también: una **interfaz hereda de otra interfaz** cuando se da la relación “es un” entre ellas y como consecuencia, recibe todos sus métodos. La novedad de la herencia de interfaces es que se puede hacer herencia múltiple, es decir, una interfaz puede tener múltiples interfaces padre.

Como ejemplo, consideremos el siguiente diagrama:



En este diagrama podemos ver que existe herencia simple en la interfaz Motor, ya que hereda de Encendible. Esto significa que los métodos de Motor en realidad son: setRevoluciones, encender y apagar. Del mismo modo vemos que hay herencia múltiple en la interfaz MotorEléctrico, ya que hereda a la vez de la interfaz Motor y de la interfaz Eléctrico. Esto significa que sus métodos son: getPorcentajeBatería, setRevoluciones y getKwHora.

Es muy importante no confundir las relaciones de herencia (representadas con la flecha continua) con las de realización (representada con flecha discontinua). La realización significa que una clase implementa la interfaz. Por ejemplo, en el diagrama se ve que MotorCocheElectrico es una clase que implementa la interfaz MotorElectrico. Esto significa que cuando programemos dicha clase, deberemos programar todos sus métodos: getPorcentajeBateria, setRevoluciones, getKwHora, encender y apagar.

Como vemos, es posible que una clase como Bombilla implemente dos interfaces a la vez (Eléctrico y Encendible). En este caso, al programar Bombilla deberemos programar todos los métodos de esas interfaces: getKwHora, encender y apagar.

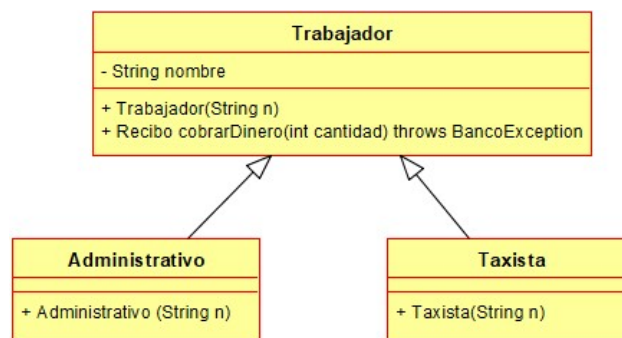
4.- Sobreescritura de métodos

Cuando se hace herencia, ya hemos visto que la clase hija recibe los métodos de la clase padre sin necesidad de programarlos. Sin embargo, en muchísimas situaciones nos encontramos con que el método heredado no se comporta como nos gustaría y tenemos que cambiarlo, bien por completo, o bien añadiéndole cosas nuevas.

Diremos que **sobreescribimos un método** cuando lo “reprogramamos” en la clase hija. Es decir, en la clase hija incluimos una nueva versión de ese método, pero cumpliendo que:

1. Tenemos que escribir exactamente igual el nombre del método y su lista de parámetros.
2. Tenemos que escribir exactamente igual el tipo de dato de retorno del método. Hay una excepción, y es que cuando el tipo de dato de retorno es referencia, se permite que el método sobreescrito devuelva una clase hija⁵.
3. El método sobreescrito no puede lanzar Checked Exceptions que no lance el método original.
4. El modificador de acceso del método sobreescrito no puede ser más restrictivo (por ejemplo, si en la clase padre el modificador es public, en la hija no puede ser private)

Vamos a verlo con ejemplos. Supongamos que tenemos el siguiente diagrama de clases:



En este diagrama el método “cobrarDinero” que hay programado en la clase Trabajador ingresa el sueldo de cada empleado en su cuenta corriente y devuelve un objeto Recibo. Si hay algún problema para hacer la transferencia, se lanza una BancoException.

El método “cobrarDinero” programado de esa forma pasa por herencia a las clases Administrativo y Taxista. Imaginemos que para Administrativo nos vale así, pero para taxista no, ya que estos cobran el dinero en mano. Por tanto, tenemos que dar una nueva programación al método cobrarDinero en la clase Taxista, es decir, hay que **sobreescribirlo**.

Entonces, nos vamos a la clase Taxista y le añadimos un método “cobrarDinero”, que tiene que tener el mismo nombre y lista de parámetros que el de la clase Trabajador (recordar el punto 1).

Por ejemplo, si el método cobrarDinero original es este:

```
1. public class Trabajador{
2.     private String nombre;
3.     public Trabajador(String n){
4.         nombre=n;
5.     }
6.     public Recibo cobrarDinero(int cantidad) throws BancoException {
7.         // programación del método ingresando dinero en el banco
8.     }
9. }
```

⁵ En esta situación, se dice que el método sobreescrito devuelve un **tipo covariante**

Una posibilidad es reescribir en la clase Taxista el método tal cual, y reprogramarlo:

```
1. public class Taxista{
2.     public Taxista(String n){
3.         super(n);
4.     }
5.     public Recibo cobrarDinero(int cantidad) throws BancoException {
6.         // programación del método cobrando el dinero en mano
7.     }
8. }
```

Sin embargo, los puntos 2 y 3 anteriores nos dicen que se permite cierta flexibilidad a la hora de escribir el tipo de retorno y la lista de excepciones del método sobreescrito.

Por ejemplo, en lugar de devolver un Recibo, el método sobreescrito podría devolver cualquier clase hija de Recibo, como por ejemplo, un Ticket. Entonces, esta forma sobrecargar el método “cobrarDinero” también sería correcta:

```
1. public class Taxista{
2.     public Taxista(String n){
3.         super(n);
4.     }
5.     public Ticket cobrarDinero(int cantidad) throws BancoException {
6.         // programación del método cobrando el dinero en mano
7.     }
8. }
```

Es importante señalar que esta norma de devolver una clase hija solo se permite cuando el retorno es un objeto. En caso de que el método de la clase padre devolviese un tipo básico, el método sobreescrito debería devolver exactamente el mismo tipo básico.

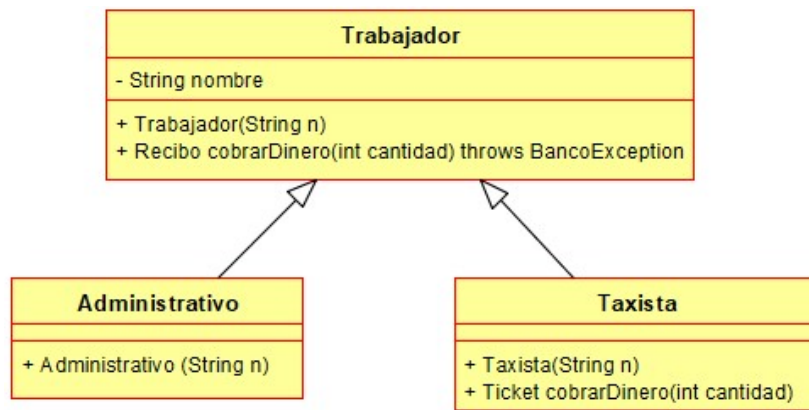
Por último, tampoco hay que respetar al 100% la lista de excepciones del método original. Se admite que el método sobreescrito lance menos excepciones que el de la clase padre, pero nunca se admite que lance una excepción nueva.

Por ejemplo, como para cobrar dinero en mano no interviene el banco, el método de la clase Taxista no necesita lanzar una BancoException y entonces se podría sobreescibir así:

```
1. public class Taxista{
2.     public Taxista(String n){
3.         super(n);
4.     }
5.     public Recibo cobrarDinero(int cantidad) {
6.         // programación del método cobrando el dinero en mano
7.     }
8. }
```

En cualquier caso, la forma concreta que debemos darle al método sobreescrito debe venir claramente indicada en el diagrama de clases, porque si no, se supondrá que la clase Taxista aceptará la programación del método “cobrarDinero” definida en su clase padre.

Por ejemplo, para tener claro que hay que sobreescibir el método “cobrarDinero” en la clase Taxista y cómo debemos programarlo, el diagrama de clases debería incluirlo en su clase Taxista:



Ahora, viendo el diagrama, ya sabemos claramente que el método “cobrarDinero” debe ser sobrescrito en la clase Taxista y además, que tenemos que programarlo así:

```

1. public class Taxista{
2.     public Taxista(String n){
3.         super(n);
4.     }
5.     public Ticket cobrarDinero(int cantidad) {
6.         // programación del método cobrando el dinero en mano
7.     }
8. }
  
```

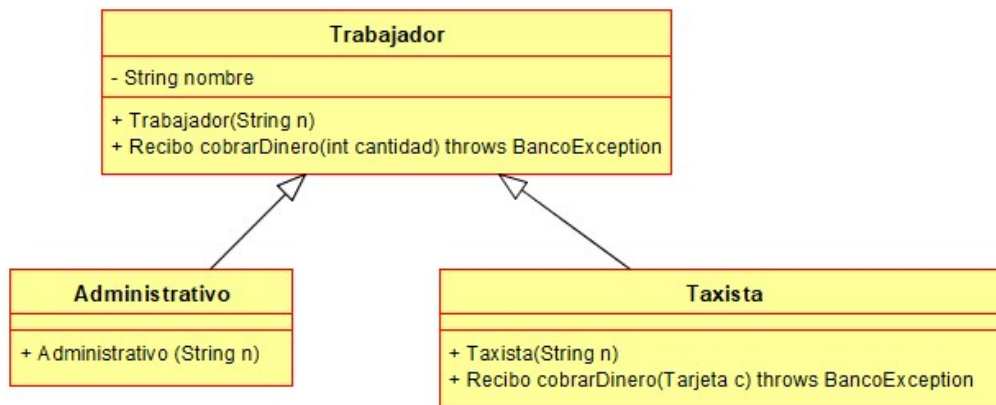
Cuando sobrescribimos un método, le podemos poner la anotación **@Override**, que es una indicación para que sepamos que el método que va a continuación es un método que sobrescribe a otro. No es obligatorio hacerlo, pero eso nos ayuda a reconocer los métodos que han sido sobrescritos, y además hace que se produzca un error de compilación si el método que sobrescribimos no se ajusta a la forma que tiene el método original.

```

1. public class Taxista{
2.     public Taxista(String n){
3.         super(n);
4.     }
5.     @Override
6.     public Ticket cobrarDinero(int cantidad){
7.         // programación del método cobrando el dinero en mano
8.     }
9. }
  
```

Por último, es muy importante destacar que la lista de parámetros del método sobrescrito debe ser idéntica (en longitud y tipos de datos usados) a la del método de la clase padre. En caso de que la lista de parámetros sea diferente, entonces ya no estamos sobrescribiendo un método. Lo que estamos haciendo es añadir una versión adicional del método con otros parámetros. Estas versiones del método se llaman **sobrecargas**.

En este diagrama lo podemos ver claramente:

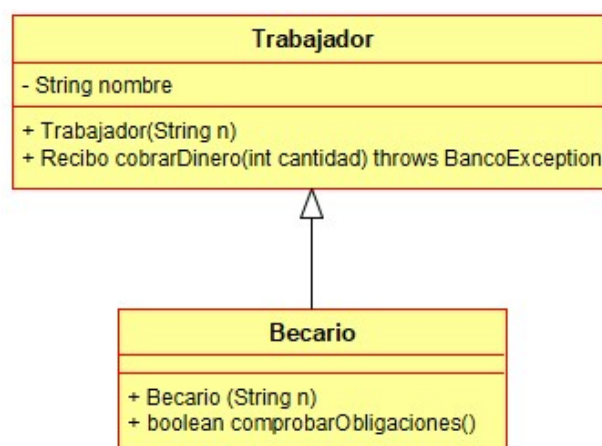


La clase Taxista ahora tiene dos versiones diferentes del método cobrarDinero. Una es la que hereda de la clase Trabajador, que ingresa el dinero en la cuenta corriente del taxista. Otra versión sería el método cobrarDinero que está programado en la clase Taxista, y que permite a un cliente pagar con su tarjeta. En este caso, el método cobrarDinero está sobrecargado (porque existe dos veces en la clase Taxista), pero no está sobrescrito (porque no se ha cambiado el método heredado de la clase Trabajador).

4.1.- Ampliar la funcionalidad de un método heredado

En la práctica a veces sucede que cuando vamos a sobrescribir un método no nos interesa cambiarlo del todo, sino ampliar las cosas que hace. Es decir, queremos que siga haciendo lo mismo que ya hacía en la clase padre, pero además necesitamos que haga más cosas.

Por ejemplo, supongamos que añadimos una nueva clase hija de Trabajador llamada Becario. A los becarios se les ingresa el dinero en su cuenta solo si se ve que han cumplido sus obligaciones de becario. Aquí tenemos el diagrama de clases:



Como vemos, en la clase Becario el método “cobrarDinero” heredado de Trabajador no nos sirve porque solo hace una parte del trabajo (ingresar el dinero en el banco). Por este motivo, es necesario sobrescribir dicho método en la clase Becario. Tendemos dos opciones:

- **Forma chapuza:** Copiamos y pegamos el método “cobrarDinero” de la clase Trabajador en la clase Becario y le añadimos las nuevas cosas que nos hacen falta. Haciendo esto nos cargamos el fácil mantenimiento del diagrama y además no siempre podremos hacerlo, porque necesitamos el código fuente de la clase Empleado, que puede haber sido hecha por otra persona.
- **Forma inteligente:** Programamos el método “cobrarDinero” de la clase Empleado y cuando necesitemos lo que hace el “cobrarDinero” de la clase padre, lo llamamos usando la palabra **super**, así:

```

1. public class Becario extends Trabajador{
2.     public Becario(String n){
3.         super(n);
4.     }
5.     public boolean comprobarObligaciones(){
6.         // comprueba que el becario ha estado trabajando en ese mes
7.     }
8.     public Recibo cobrarDinero(int cantidad) throws BancoException {
9.         Recibo recibo=null;
10.        if(comprobarObligaciones()) {
11.            recibo = super.cobrarDinero(cantidad);
12.        }
13.        return recibo;
14.    }
15. }

```

Gracias a la palabra **super**, podemos invocar a la versión del método “cobrarDinero” programada en la clase padre, y así no perdemos lo que hace.

Ejercicio 13 : Consulta el diagrama de clases **Contraseñas** y programa la clase **GeneradorContraseñas**, que es un objeto que genera contraseñas aleatorias y utiliza para ello la clase **StringBuilder**, de la librería estándar de Java.

Ejercicio 14 : Siguiendo el diagrama del ejercicio anterior, programa la clase **GeneradorContraseñasÚnicas**

Ejercicio 15 : Siguiendo el diagrama de los ejercicios anteriores, programa la clase **GeneradorContraseñasArchivo**

Ejercicio 16 : Siguiendo el diagrama de los ejercicios anteriores, programa las clases **GeneradorContraseñasInvertidas** y **GeneradorContraseñasRaras**.

Ejercicio 17 : ¿Sería posible hacer un generador de contraseñas únicas que además guardase en un archivo todas las que va generando?

5.- Métodos de la clase Object

La programación avanzada de una clase implica no dejarla solo con los métodos que aparecen en su diseño, sino también sobrescribir el comportamiento por defecto que tienen algunos de los métodos heredados de la clase **Object**, concretamente estos:

- **toString:** El objetivo de este método es devolver una cadena de texto que proporcione una idea sobre el objeto y su estado actual.
- **equals:** Permite la comparación del objeto con otro para saber si son iguales o no.
- **hashCode:** Proporciona un número entero, de forma que objetos que sean iguales con el método equals, deben coincidir en dicho número.

Aunque podemos programar “a mano” las sobreescripciones de estos métodos, los IDE ofrecen facilidades para programarlos automáticamente.

5.1.- Sobreescritura de toString

El método toString heredado de la clase Object sirve de muy poco, ya que muestra el nombre completamente cualificado del objeto (su paquete junto con su clase), un signo arroba y el valor del hashcode. Por ejemplo, **daw.empresa.Empleado@1a16863**

Por tanto, una de las primeras cosas que se hacen cuando se programa una clase es proporcionar un método toString más apropiado, que muestre cosas importantes del objeto, como el valor de algunas de sus propiedades.

Para sobreescribir toString nos basta con volver a programarlo en cualquier clase. Por ejemplo, esta clase Empleado sobreescribe toString para devolver un texto formado por el dni, un guión, el nombre del empleado y el sueldo entre paréntesis.

```

1.  public class Empleado{
2.      private String dni;
3.      private String nombre;
4.      private double sueldo;
5.      public Empleado(String d, String n, double s){
6.          dni=d;
7.          nombre=n;
8.          sueldo=s;
9.      }
10.     public String toString(){
11.         return dni+" - "+nombre+" (" +sueldo+");"
12.     }
13. }
```

Es importante destacar que los **Records** ya tienen programado el método toString para que se muestre por pantalla el nombre y valor de todas sus propiedades.

5.2.- Sobreescritura de equals

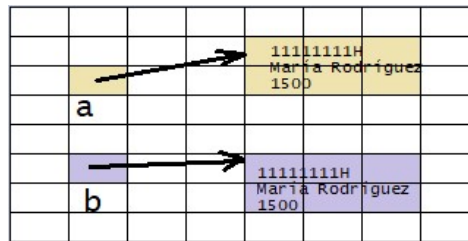
El objetivo de este método es devolver true si el objeto sobre el que actúa (this) coincide con el objeto que se pasa como parámetro (que es un Object).

Como equals es un método de Object, tenemos que todas las clases por defecto ya tienen un método equals heredado de Object. Sin embargo, este método heredado **considera distintos todos los objetos**, porque utiliza el signo == para comparar si dos objetos son iguales. Como ya sabemos, este operador compara el valor de los punteros en la memoria donde están los objetos, por lo que solo devolverá true cuando comparemos un objeto consigo mismo.

Por ejemplo, consideremos la clase Empleado del ejemplo anterior y este programa:

```
1. public class Empleado{
2.     public static void main(String[] args){
3.         Empleado a = new Empleado("11111111H", "María Rodríguez", 1500);
4.         Empleado b = new Empleado("11111111H", "María Rodríguez", 1500);
5.         System.out.println(a.equals(b));    // devuelve false
6.     }
7. }
```

El código anterior devuelve false aunque nosotros veamos que los empleados son “iguales”. El problema es que la clase Empleado usa el método equals recibido directamente de la clase Object, y ese método por defecto examina la memoria, viendo los objetos así:



Por este motivo, el método equals recibido de la clase Object devuelve false. Para ese método los dos objetos son diferentes.

Si queremos que los objetos puedan ser comparados según el valor de sus propiedades, no tenemos más remedio que sobrescribir el método equals. Para ello debemos garantizar que ese método devuelve true cuando comparamos objetos que sean de la misma clase y que coincidan en las propiedades que nosotros veamos que identifiquen al objeto.

Por ejemplo, podríamos sobrescribir el método equals de la clase Empleado así:

```
1. public class Empleado{
2.     private String dni;
3.     private String nombre;
4.     private double sueldo;
5.     public Empleado(String d, String n, double s){
6.         dni=d;
7.         nombre=n;
8.         sueldo=s;
9.     }
10.    public boolean equals(Object o){
11.        boolean r=false;
12.        if(o instanceof Empleado e){
13.            // compruebo si las propiedades coinciden con las del objeto pasado como parámetro
14.            r = dni.equals(o.dni) && nombre.equals(o.nombre) && sueldo==o.sueldo;
15.        }
16.        return r;
17.    }
18. }
```

Lo primero que hemos hecho ha sido comprobar que el Object recibido como parámetro sea un objeto de la clase Empleado. En caso de que no sea así, saltamos el if y directamente devolvemos false. En cambio, si se ha recibido un empleado, lo recuperamos en el instanceof y comparamos una por una si sus propiedades coinciden con las del objeto que estamos programando, devolviendo true en caso de que todas coincidan.

La programación del método equals se complica si se permite que algunas propiedades tomen valores nulos. Por ese motivo, para sobrescribir equals lo mejor es acudir a las herramientas que proporcionan los IDE para su programación.

Sobrescribir equals es importantísimo. Por ejemplo, si no lo hacemos, los métodos de las clases del Java Collection Framework pueden devolver resultados incorrectos (por ejemplo, decir que un objeto no se encuentra en una lista cuando nosotros vemos que si lo está)

Al igual que ocurría con toString, los **Records** ya tienen programado el método equals de forma que dos objetos sean iguales si coinciden una por una en todas sus propiedades.

5.3.- Sobreescritura de hashCode

El objetivo del método hashCode es devolver un número que sea igual para todos los objetos que sean iguales con equals. Por defecto, el método hashCode heredado de la clase Object devuelve un número distinto para cada objeto creado en el programa, lo cual es compatible con la versión de equals de la clase Object (que solo devuelve true cuando se hace equals de un objeto consigo mismo).

Por tanto, **si sobrescribimos el método equals, para que todo sea coherente será necesario sobrescribir también el método hashCode.**

El método hashCode se suele programar cogiendo las propiedades que intervienen en el método equals, tomar sus hashcodes (o sus valores si son de tipo básico), multiplicarlas por números primos que elijamos aleatoriamente y sumarlo todo.

```
1. public class Empleado{
2.     private String dni;
3.     private String nombre;
4.     private double sueldo;
5.     public Empleado(String d, String n, double s){
6.         dni=d;
7.         nombre=n;
8.         sueldo=s;
9.     }
10.    public boolean equals(Object o){
11.        boolean r=false;
12.        if(o instanceof Empleado){
13.            Empleado a = (Empleado)o;
14.            r = dni.equals(o.dni) && nombre.equals(o.nombre) && sueldo==o.sueldo;
15.        }
16.        return r;
17.    }
18.    public int hashCode(){
19.        return 3*dni.hashCode() + 7*nombre.hashCode() + 11 * sueldo;
20.    }
21. }
```

Debido a la complejidad que puede conllevar (sobre todo cuando algunas de las propiedades que intervienen puedan ser null), se recomienda usar la opción que tienen los IDE para sobrescribirlo. Los IDE permiten sobrescribir equals y hashCode a la vez.

6.- La interfaz Comparable<T>

Cuando programamos una clase es muy importante decidir si los objetos deberán poder ordenarse de menor a mayor. Por ejemplo, si hacemos una clase Alumno tal vez queramos que los alumnos se puedan ordenar por orden alfabético de sus apellidos.

Por defecto, cuando programamos una clase, no existe ningún orden entre los objetos que luego creamos de esa clase. Si queremos que los objetos se puedan ordenar de menor a mayor, deberemos añadir a la clase la habilidad de tener un orden, y eso significa que la clase deberá implementar la interfaz **Comparable<T>**, siendo T la clase que estamos programando⁶. Por ejemplo, si queremos que los objetos de la clase Alumno se puedan ordenar de menor a mayor, la clase Alumno deberá implementar la interfaz Comparable<Alumno>.

La interfaz **Comparable<T>** es una interfaz que viene en Java (no hay que programarla) y está en el paquete **java.util**.

Solo tiene un método: **+ int compareTo(T obj)** que habrá que sobrescribir.

El método **compareTo** es usado internamente por el método **Collections.sort** para ordenar de menor a mayor los objetos. Para programarlo, hay que devolver la “distancia” que existe entre el objeto sobre el que se llama el método (this) y el que se pasa como parámetro (obj), teniendo en cuenta el siguiente signo:

Si a es mayor que b	Positivo
Si a es menor que b	Negativo

En muchas ocasiones el método compareTo se programará **restando** los valores usados como criterio para comparar. Por ejemplo, si tenemos una clase Alumno y queremos ordenarlos por su nota media, el método compareTo devolverá la resta de sus notas medias.

```
1. public class Alumno implements Comparable<Alumno>{
2.     private String nombre;
3.     private List<Integer> notas;
4.     public Alumno(String n){
5.         nombre=n;
6.         notas=new ArrayList<>();
7.     }
8.     public void añadirNota(int n){
9.         notas.add(n);
10.    }
11.    public int getNotaMedia(){
12.        int suma=0;
13.        for(Integer n:notas){
14.            suma+=n;
15.        }
16.        return suma/notas.size();
17.    }
18.    public int compareTo(Alumno a){
19.        return this.getNotaMedia() - a.getNotaMedia();
20.    }
21. }
```

⁶ Cuando una clase implementa la interfaz Comparable<T> se dice que existe un **orden natural** en esa clase

Sin embargo, en otras ocasiones habrá que comparar por criterios que no sean números y por tanto, no se puedan restar. Por ejemplo, esto pasa si queremos ordenar los alumnos por orden alfabético de su nombre. Afortunadamente, siempre que estemos en esta situación tendremos dos objetos cuya distancia habrá que calcular, y esos objetos ya tendrán un método `compareTo` que nos la dará. Por ejemplo, si queremos ordenar los alumnos por su nombre, los dos nombres son `String`, que ya tienen un `compareTo` que nos da su distancia.

```
1. public class Alumno implements Comparable<Alumno>{
2.     private String nombre;
3.     public Alumno(String n){
4.         nombre=n;
5.     }
6.     public int compareTo(Alumno a){
7.         return this.nombre.compareTo(a.getNombre);
8.     }
```

Una vez que una clase implementa `compareTo`, ya podemos usar los métodos `Collections.sort` o `Arrays.sort` para ordenar una lista de esos objetos⁷.

Ejercicio 18 : Consulta el diagrama de clases **Paquetería** y programa la interfaz `EmpresaPaquetería` y las clases `Paquete` y `Transportista`.

Ejercicio 19 : Siguiendo con el ejercicio anterior, programa la clase `TransportistaOrdenado`.

Ejercicio 20 : Siguiendo con el ejercicio anterior, programa la clase `PaquetesPepe` y `EmpresaPremium`

Ejercicio 21 : Siguiendo con el ejercicio anterior, programa la clase `EmpresaLowCost`, que es una empresa de paquetería en la que guarda una lista de transportistas y los paquetes se asignan de forma cíclica a los transportistas.

7.- Clases abstractas

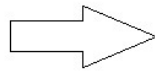
En la vida real, muchas veces encontramos clases que representan conceptos genéricos que no podemos asociar con ningún objeto concreto. Estas clases se llaman **clases abstractas**.

Por ejemplo, la clase `Comida` es una clase abstracta, ya que nunca vemos objetos `Comida`, sino que vemos objetos `Manzana`, `Bocadillo`, `Helado`,... Por tanto, la clase `Comida` representa el concepto abstracto de las cosas que podemos comer, pero nunca vamos a poder comer un objeto `Comida`, sino alguna forma concreta de esta. De igual forma, la clase `Deporte` sería una clase abstracta, ya que vemos o jugamos a deportes concretos.

⁷ ¿Qué pasa si una clase implementa `Comparable<T>` pero queremos ordenar por otro criterio? En esa situación deberemos pasar a `Collections.sort` un objeto `Comparator<T>`, cuyo método `compare(T a, T b)` nos devuelve la distancia entre “a” y “b” tal y como hace `compareTo`.



¿Comida?



Manzana



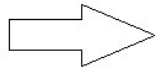
Bocadillo



Helado



¿Deporte?



Fútbol

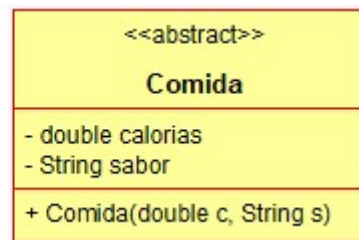
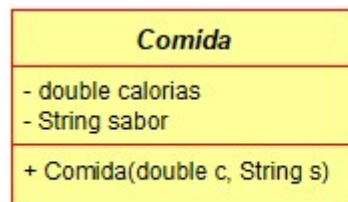


Baloncesto



Taewkondo

Las clases abstractas en UML se representan escribiendo su nombre en cursiva, o el estereotipo <<abstract>>:



Para programar una clase abstracta, basta con poner la palabra **abstract** justo en el momento de crear la clase, así:

```

1. public abstract class Comida{
2.     private double calorías;
3.     private String sabor;
4.
5.     public Comida(double c, String s){
6.         calorías=c;
7.         sabor=s;
8.     }
9. }
```

Una vez que convertimos una clase en abstracta, ya no podremos crear objetos de esa clase. Es decir, aunque el constructor de Comida sea public, ya no se podrá utilizar y nos dará este error de compilación:

```

public static void main(String[] args) {
    Comida c = new Comida(100, "salado");
}
```

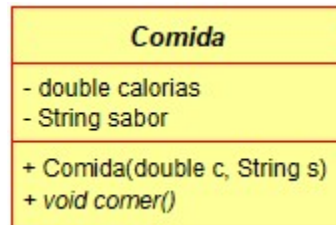
Comida is abstract; cannot be instantiated

 (Alt-Enter shows hints)

Por tanto, la única forma posible de obtener objetos de la clase Comida es mediante sus clases hijas. De hecho, el único sentido de tener clases abstractas es tener clases hijas.

7.1.- Métodos abstractos

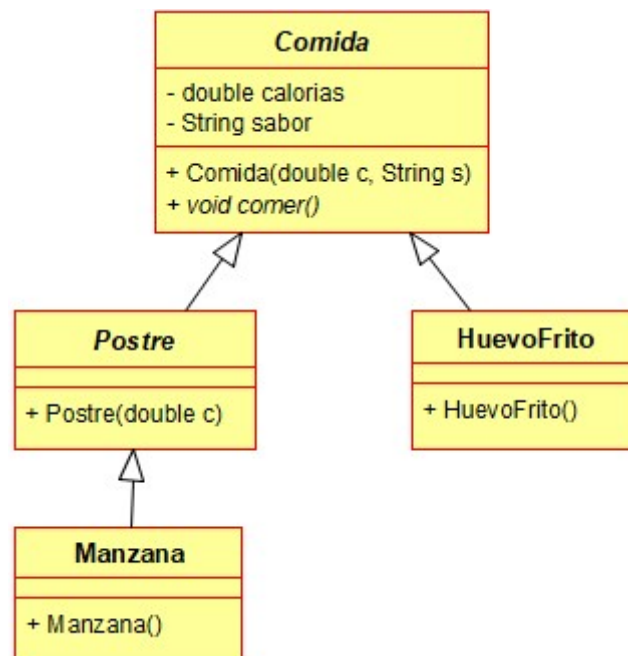
Las clases abstractas también pueden tener **métodos abstractos**. Un método abstracto es un método de una clase abstracta que está sin programar, para que sean las clases hijas quienes lo hagan. Los métodos abstractos se escriben en cursiva en los diagramas de clases:



Para programar un método abstracto basta con escribir la palabra **abstract** y dejarlo sin programar, tal y como hacíamos con los métodos que hemos estudiado en las interfaces⁸.

```
1. public abstract class Comida{
2.     private double calorias;
3.     private String sabor;
4.     public Comida(double c, String s){
5.         calorias=c;
6.         sabor=s;
7.     }
8.     public abstract void comer();
9. }
```

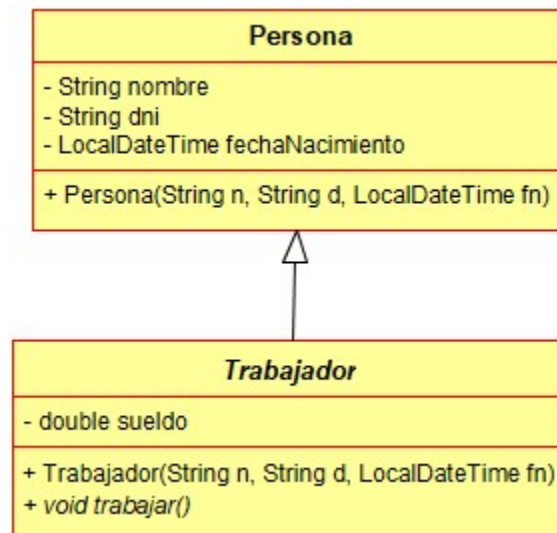
Son las clases hijas no abstractas quienes están obligadas a sobrescribir dicho método y proporcionar una implementación. Por ejemplo, observemos este diagrama:



⁸ Los métodos que hemos estudiado de las interfaces son abstractos porque en realidad siempre son “public abstract”. Lo que ocurre es que es opcional ponerle cualquiera de estas dos palabras.

En este ejemplo podemos ver como la clase abstracta Comida tiene una clase hija abstracta llamada Postre. Esta clase abstracta no está obligada a programar el método “comer” (porque es abstracta). Sin embargo, las clases Manzana y HuevoFrito si están obligadas a sobrescribir el método “comer”, ya que son **clases concretas** (es decir, no abstractas).

También puede ocurrir que una clase concreta tenga una clase hija abstracta. Aquí tenemos el ejemplo de esa situación:



En este ejemplo podemos ver que Persona es una clase concreta, y podemos crear objetos Persona con su constructor sin ningún problema. Sin embargo, tiene una clase hija abstracta que es Trabajador, con un método abstracto “trabajar”. Para obtener objetos trabajadores deberíamos usar clases hijas: Administrativo, Informático, etc.

Ejercicio 22 : Consulta el diagrama de clases **Animales** y programa en un paquete llamado **daw.zoo** las clases Animal, Tigre y León

Ejercicio 23 : Programa en el mismo paquete **daw.zoo** la clase ContenedorAnimales.

Ejercicio 24 : Programa en el mismo paquete **daw.zoo** las clases Jaula y TransporteAnimales

Ejercicio 25 : Consulta el diagrama de clases **Figuras geométricas** y programa las interface Coloreable, Centrabale y FiguraGeométrica.

Ejercicio 26 : Programa la interfaz Apoyable y la clase Círculo

Ejercicio 27 : Programa las clases Rectángulo y Cuadrado

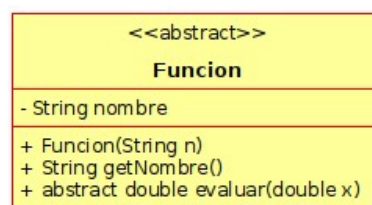
Ejercicio 28 : Programa la clase Triángulo, que es un triángulo definido por tres puntos que son sus tres vértices, y su color siempre es azul.

8.- Clases anónimas

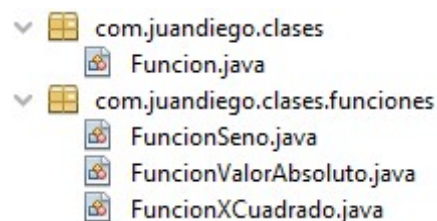
Hay ocasiones en las que estamos haciendo un programa y necesitamos un objeto de una clase hija que sería muy pequeña y rápida de escribir, de forma que ponernos a programar esa clase hija sería considerado “boilerplate” y nos llenaría el proyecto de clases muy pequeñas de un solo uso. Concretamente, esto sucede cuando se programan interfaces de usuario en Java para Android.

Las clases anónimas nos permiten crear un objeto de una clase hija sin tener que programar esa clase hija. El objeto recibirá todos los métodos de su clase padre y podrá sobrescribir los que crea necesario.

Por ejemplo, supongamos que tenemos la siguiente clase abstracta, que representa una función matemática $y=f(x)$



Si estamos haciendo un programa que trabaje con muchas funciones, en principio deberíamos tener una clase diferente para cada una de ellas, lo cual es un rollo porque son clases muy pequeñas y además se nos llenaría el proyecto de esas clases (lo suyo sería meterlas en un paquete separado para tenerlas separadas del resto de clases)



Por ejemplo, aquí tenemos lo que podría ser la clase FuncionXCuadrado:

```
1. public class FuncionXCuadrado extends Funcion{
2.     public FuncionXCuadrado(){
3.         super("x cuadrado");
4.     }
5.     @Override
6.     public double evaluar(double x){
7.         return x*x;
8.     }
9. }
```

Como podemos ver, es muy corta y no merece la pena tener muchas clases de ese estilo.

Para solucionar el problema tenemos las clases anónimas, que permiten al programador de aplicaciones guardar directamente dentro de un objeto de tipo **Función** un objeto de lo que sería una clase hija de Función, pero sin programarla en un archivo externo. Sería así:

```

1. public static void main(String[] args){
2.     Funcion f = new Funcion("x cuadrado") { // creo un objeto cuya clase hereda de Función
3.         @Override
4.         public double evaluar(double x){ // sobrescribo el método evaluar para ese objeto
5.             return x*x;
6.         }
7.     }
8.     System.out.println(f.evaluar(8)); // respuesta: 64
9. }

```

En este ejemplo vemos cómo la variable “f” se rellena con un objeto que es hijo de la clase Función y sobrescribe el método “evaluar” de la forma que le interesa, evitando así tener que hacer la clase FuncionXCuadrado en un archivo externo.

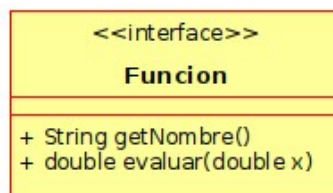
También es posible añadir propiedades a las clases anónimas, que se inicializarían dentro de un bloque de llaves que se pondría a continuación de ellas⁹. Por ejemplo, supongamos la siguiente clase anónima que calcularía una parábola de tipo Ax^2+Bx+C .

```

1. public static void main(String[] args){
2.     Funcion f = new Funcion("parábola") {
3.         // propiedades
4.         private int A;
5.         private int B;
6.         private int C;
7.         // bloque inicializador de las propiedades
8.         {
9.             A=-4;
10.            B=9;
11.            C=2;
12.        }
13.        @Override
14.        public double evaluar(double x){
15.            return A*x*x + B*x + C;
16.        }
17.    }
18.    System.out.println(f.evaluar(2)); // respuesta: 4
19. }

```

Con clases anónimas también podemos crear objetos que implementen interfaces de la misma forma. Por ejemplo, si Función hubiese sido esta interfaz:



Podríamos programar un objeto anónimo igual que antes, pero teniendo en cuenta que ahora hay dos métodos abstractos:

```

1. public static void main(String[] args){
2.     Funcion f = new Funcion() {
3.         @Override
4.         public String getNombre(){
5.             return "x cuadrado";
6.         }
7.         @Override
8.         public double evaluar(double x){
9.             return x*x;
10.        }
11.    }
12. }

```

⁹ Este bloque de llaves se denomina **bloque inicializador**, y también se puede usar en las clases normales para incluir código que se ejecutaría antes del constructor, pero su uso es poco frecuente.

Ejercicio 29 : Programa el record Persona y la siguiente interfaz Filtro:

<<interface>> Filtro
+ boolean aceptar(Persona p)

<<record>> Persona
+ Persona(String nombre, int edad, boolean soltero) + static void mostrarEnPantalla(List<Persona> personas, Filtro filtro)

- El método mostrarEnPantalla recibe una lista con personas y muestra en pantalla el nombre de aquellas que al aplicarles el filtro se obtiene como resultado true.

Una vez terminado el record y la interfaz, haz un programa que cree este List<Persona> de personas:

Juan Pérez	25	true
María López	39	false
Javier Presley	19	false
Robert Jiménez	30	false

Y muestre este mensaje al usuario:

- 1) Ver personas casadas
- 2) Ver personas solteras mayores de edad
- 3) Ver personas cuyo nombre empieza por la letra J.

Cada opción creará un objeto de una clase anónima que implemente la interfaz Filtro, de forma que acepte a las personas a las que hace referencia el texto de la opción (por ejemplo, el método aceptar del filtro de la opción 1 devolverá true si una persona está casada). Después se usará el método mostrarEnPantalla para aplicar el filtro a las personas de la lista.

9.- Polimorfismo

El polimorfismo es, posiblemente, la técnica más importante de toda la programación orientada a objetos¹⁰. Se utiliza constantemente en los programas profesionales de gran tamaño para conseguir que sean fáciles de mantener y de ampliar. Cuando un programa ha sido diseñado para conseguir polimorfismo, podemos conseguir estos beneficios:

- Adaptar el programa a un gran cambio del entorno, sin tener que tocar el núcleo principal de la aplicación, y solo aquella parte que interactúa con el entorno.

Ejemplo: Nuestro programa trabaja con datos guardados en archivos y un día los cambiamos por una base de datos. Este cambio, que en condiciones normales podría

¹⁰ Todos los lenguajes de programación, de una forma u otra incorporan el concepto de polimorfismo. En los lenguajes dinámicamente tipados como Python, el **duck typing** hace la labor de polimorfismo.

suponer tener que volver a programar entero el programa, es inmediato si el programa fue diseñado teniendo en mente el polimorfismo.

- Poder realizar una tarea de muchas formas posibles. El polimorfismo nos permite elegir la forma que queremos usar para resolver la tarea. Además, si en el futuro aparecen formas mejores de hacer la tarea, podremos “añadirla” al programa, sin tocar las anteriores y el núcleo de la aplicación.

Ejemplo: Los videojuegos utilizan librerías para dibujar en pantalla. Supongamos que un videojuego utiliza Direct3D. Con el polimorfismo podemos cambiar la forma de dibujar en la pantalla sin tener que cambiar para nada el código fuente del videojuego. Podemos añadir una nueva forma de dibujar la pantalla basada en otra librería, como OpenGL, y dar al usuario la posibilidad de elegir una de ellas.

- Ampliar las funciones que hace el programa mediante plugins, drivers o módulos adicionales.

Ejemplo: En el momento de su lanzamiento un programa posee una serie de funciones, pero con el tiempo se descubren otras nuevas que se le pueden añadir. Sin polimorfismo, sería necesario introducirlas manualmente y volver a compilarlo todo. Con el polimorfismo, podemos desarrollar un sistema de plugins, en el que cada uno codifica una nueva función que se va a añadir al programa. Los plugins se “enganchan” bien con el programa gracias al polimorfismo.

Como vemos, las ventajas del polimorfismo son evidentes y muy deseables. Sin embargo el polimorfismo no surge cuando estamos programando, sino en la fase de diseño de la aplicación. Es en esta etapa cuando se le da forma al programa y allí se decide si se debe realizar un diseño que contemple el polimorfismo. Como es lógico, los diseños que buscan el polimorfismo son más complicados que los que no.

Vamos a poner a continuación la definición de polimorfismo, aunque en este punto no se va a comprender bien lo que significa:

El polimorfismo se define como el principio por el cual, el código del método que se ejecuta se determina en tiempo de ejecución.

Como la definición no aclara mucho lo que significa, y parece no tener nada que ver con las ventajas que hemos comentado antes, vamos a ver un ejemplo donde aparece el polimorfismo.

En los siguientes apartados vamos a ver cómo podemos conseguir polimorfismo usando clases abstractas e interfaces¹¹.

¹¹ Existen otras formas de polimorfismo como el **polimorfismo por sobrecarga** o el **polimorfismo por reflexión**, que muchos autores no las consideran “auténtico polimorfismo”.

9.1.- Polimorfismo mediante clases abstractas

Supongamos que estamos programando un videojuego de naves espaciales¹², de forma que el jugador puede elegir su nave entre tres opciones:



Nave A




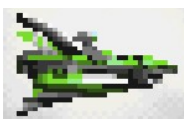







Nave B

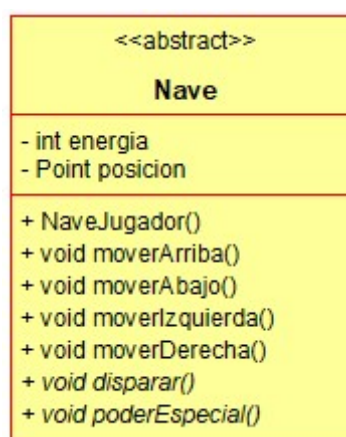


Nave C

Cada nave tiene su propia forma de disparar, y además tiene su propio poder especial:

Nave	Disparo normal	Poder especial
		
		
		

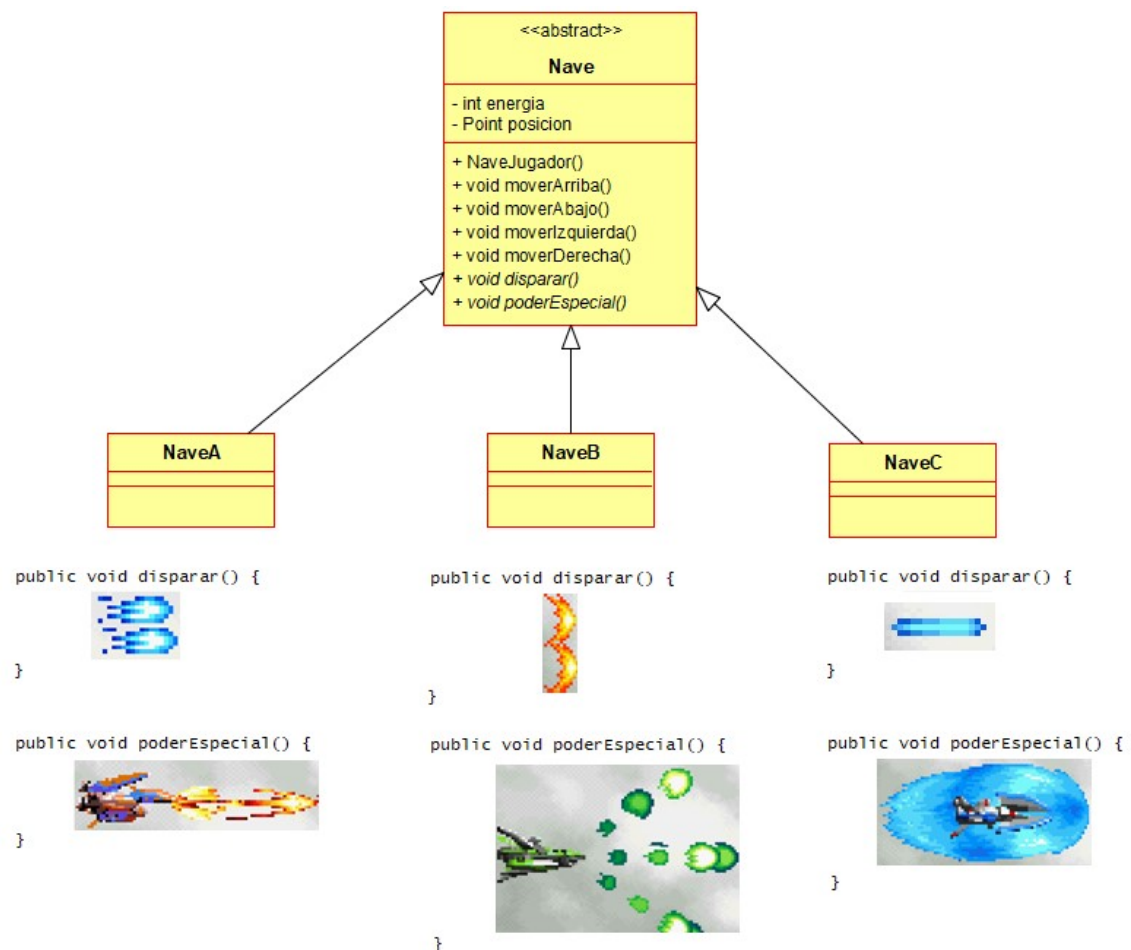
A la hora de programar este juego, haríamos una clase abstracta llamada **Nave** que tendría este diagrama:



En esta clase estarían todas las propiedades y métodos comunes a las tres naves, como por ejemplo su posición y el nivel de energía. Además, tendríamos como métodos abstractos a “disparar” y “poderEspecial”, ya que hemos visto que cada nave dispara de forma distinta.

¹² Imágenes tomadas del **Blazing Star**, de Neo Geo

Ahora, haríamos tres clases hijas que heredarían de la clase Nave y que programarían los métodos “disparar” y “poderEspecial” adaptados a su forma concreta de funcionar:



Cuando se inicia el juego, el programa crea una variable del tipo padre (Nave) que estará vacía, y pregunta al usuario la nave que quiere. Según la elección del usuario, esa variable se rellena con un objeto concreto del tipo elegido:

```

1. Nave jugador = null;
2. int opcion = ... // se detecta la opción elegida por el usuario
3. if(opcion==1){
4.     jugador=new NaveA();
5. }else if(opcion==2){
6.     jugador=new NaveB();
7. }else if(opcion==3){
8.     jugador=new NaveC();
9. }
  
```

Lo más importante es darse cuenta de que a partir de la línea 9, la variable jugador se rellena con un objeto de la clase elegida por el usuario. A continuación, programamos el juego utilizando solamente la variable jugador, de forma independiente del tipo de dato que contiene.

Veamos lo que ocurre cuando se pulsa el botón de disparar:

```

1. boolean disparoPulsado = ... // vemos si el usuario ha pulsado disparar
2. if(disparoPulsado){
3.     jugador.disparar();
4. }

```

Al pulsar el botón de disparo, se llama al método disparar sobre el objeto “jugador” y se activa el método disparar del objeto concreto con el que se rellenó esa variable. Por ejemplo, si el usuario seleccionó la nave B, la llamada de la línea 3 hará esto:



Observa como esto funciona para todos los tipos de nave. El programa es independiente del tipo de nave elegida, y en la línea 3 el método disparar toma la forma del método disparar con el que se rellenó la variable “jugador”. Esto se denomina **polimorfismo**.

El polimorfismo permite “olvidarnos” del tipo auténtico que se ha usado para rellenar la variable “jugador”, porque en realidad, **no nos hace falta dicho tipo de dato**. Lo único que nos hace falta para poder hacer el juego es una Nave, pero no es necesario saber si es de tipo NaveA, NaveB o NaveC. Cuando se programa de esta forma, una sola línea puede hacer muchas cosas distintas, según lo que esté programado en las clases hijas, y esto es algo muy potente.

¿Qué ventaja tiene haber programado el juego así? Imaginemos que queremos lanzar una actualización del juego que incluya dos naves más. Entonces:

- Lo único que tenemos que hacer simplemente es programar dos clases nuevas con las implementaciones de disparar y poderEspecial y permitir que el usuario pueda elegir las. Casi no hay que tocar nada en el código del programa principal¹³.
- Esas nuevas clases pueden programarse en paralelo por personas distintas, lo que favorece el trabajo en equipo.
- Si alguna de las clases nuevas tiene un fallo, sabremos que está en ellas, y no tendremos que ir buscando el error en el código fuente de la aplicación.

9.2.- Polimorfismo mediante interfaces

En el ejemplo anterior hemos visto que podemos conseguir polimorfismo cuando en un programa llamamos a un método abstracto sobre una variable cuyo tipo de dato es una clase abstracta y el método abstracto toma la forma del objeto que realmente habita en la variable.

¹³ De hecho, Java incorpora técnicas como **reflexión** y **SPI (Service Provider Interface)** que hacen que no haya que tocar absolutamente nada en el código principal. Solamente con añadir las clases nuevas el programa automáticamente las detecta y permite seleccionarlas. Es lo que se hace para programar **plugins**.

Esto mismo también puede conseguirse si en lugar de clases abstractas tenemos interfaces. Es decir, se producirá polimorfismo cuando tenemos una variable cuyo tipo de dato es una interfaz. Al llamar al método de la interfaz, se activará la implementación del método que esté definida en la clase que realmente habita en la variable.

El mismo ejemplo de las naves serviría tal cual, si Nave fuese ahora una interfaz y las clases NaveA, NaveB y NaveC implementasen dicha interfaz. En el programa seguiríamos teniendo una variable de tipo Nave (que ahora sería una interfaz) que se rellenaría con un objeto de la clase NaveA, NaveB o NaveC, según elija el usuario.

Los métodos de disparar y de poderEspecial siguen siendo abstractos y las clases lo tienen que implementar igual que antes. Si el usuario elige la nave B y pulsa el botón de poder especial, la línea **nave.poderEspecial();** hará que se active el método “poderEspecial” que hay programado en la clase B y se vea esto:



Por tanto, para conseguir polimorfismo lo que se hace es trabajar con variables cuyo tipo de dato es una clase abstracta o una interfaz y rellenarlas con objetos que den forma a los métodos abstractos que definen. De esa forma nos aseguramos de que en cualquier momento podamos cambiar las clases subyacentes por otras que sean mejores o que amplíen su funcionalidad.

Ejercicio 30 : Realiza un programa que utilice las clases del diagrama **Figuras geométricas** para hacer un programa que cree un List<FiguraGeometrica> y a continuación muestre al usuario este menú:

- 1) Añadir un rectángulo
- 2) Añadir un cuadrado
- 3) Añadir un círculo
- 4) Añadir un triángulo
- 5) Dibujar todo

Por cada cosa que elija el usuario el programa preguntará los datos necesarios para crearlo (coordenadas del centro y radio para círculo, etc). Entonces, creará la figura correspondiente y la añadirá a la lista de figuras. Cuando el usuario pulse la opción 5), el ordenador preguntará si desea dibujarlo en un PDF o en la ConsolaDAW. Por último, el programa dibujará todas las figuras en el lugar donde haya seleccionado el usuario.

Ejercicio 31 : Programa todo lo que hay en el diagrama **Efectos especiales**

Ejercicio 32 : Siguiendo con el ejercicio anterior, crea un paquete llamado **com.ieshlanz.tema5.programa** y haz en él un programa que pregunte una frase al usuario y a continuación muestre este menú:

- 1) Aplicar todos los efectos a la frase
- 2) Aplicar solo los efectos simétricos

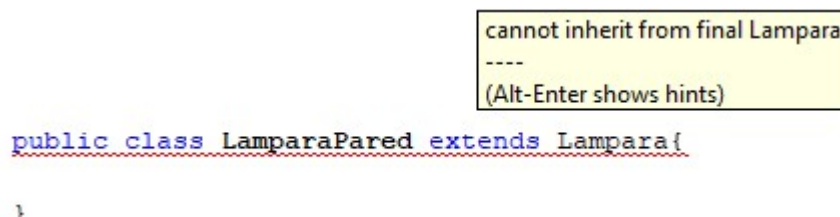
Según la opción elegida, el programa aplicará le aplicará todos los efectos especiales a la frase, o solo los simétricos.

10.- Clases finales

Una clase es **final** si no puede tener clases hijas. Podemos convertir cualquier clase que estemos programando en final si escribimos la palabra reservada “final” justo cuando vamos a empezar a programarla, de esta forma:

```
3. public final class Lampara{
4.     public Bombilla bombilla;
5.     public Interruptor interruptor;
6.     public Cable cable;
7. }
```

De esta forma, la clase Lámpara sería final y ya no podría tener clases hijas. Cualquier intento de crear una clase hija de ella lanzaría este error de compilación:



```
public class LamparaPared extends Lampara{
    ...
}
```

También existe el concepto de **método final**, que es un método (puede ser de cualquier clase) que no puede ser sobrescrito en sus clases hijas. O sea, cuando ponemos la palabra **final** a un método, significa que una clase hija no puede sobrescribir el método, o saldrá un error de compilación.

```
1. public class Alumno implements Comparable<Alumno>{
2.     private String nombre;
3.     private List<Integer> notas;
4.     public Alumno(String n){
5.         nombre=n;
6.         notas=new ArrayList<>();
7.     }
8.     public final void añadirNota(int n){ // este método no puede ser sobrescrito
9.         notas.add(n);
10.    }
```

Una pregunta que siempre surge en este momento es: ¿por qué es necesario hacer clases finales y métodos finales? ¿No es mejor que todas las clases puedan tener hijas? ¿qué ventaja tenemos impidiendo que una clase pueda tener clases hijas?

Así es como pensaban los diseñadores originales del lenguaje Java (y todo el mundo en aquella época), porque se pensaba que la herencia de clases era la panacea en el mundo de la programación. Todas las clases en Java por defecto son **abiertas** (pueden tener clases hijas) y los métodos por defecto, son **abiertos** (se pueden sobrescribir en clases hijas) para fomentar el uso de la herencia. Estamos hablando de la mentalidad que había en 1995 en el mundo de la programación, donde la programación orientada a objetos (C++, Java) comenzaba a imponerse a la programación imperativa (básicamente C).

Sin embargo, 20 años de experiencia masiva en Java en los que se han realizado cientos de proyectos en todo el mundo, han puesto de manifiesto que la herencia solo es buena cuando se usa bien. O dicho de otro modo, ha habido gente que ha usado mal la herencia y entonces no ha resultado ser tan buena. En el siguiente apartado veremos algunas de las limitaciones y problemas que tiene la herencia cuando se usa mal.

Actualmente, se recomienda que las clases sean siempre finales y que solamente sean abiertas las clases que realmente se diseñen y documenten para ello. Además, en las clases abiertas se recomienda que todos sus métodos sean finales, excepto aquellos pensados para ser sobrescritos¹⁴.

10.1.- Clases selladas

Como muestra del interés que existe para limitar el uso de la herencia, en Java 17 ha aparecido el concepto de **sealed class** (clase sellada), que es una clase que solo puede tener las clases hijas que indicamos en su declaración y ninguna más¹⁵.

Por ejemplo, si queremos hacer una clase `FiguraGeométrica` que solo tenga como clases hijas a `Rectángulo` y a `Círculo`, la sellaremos así:

```
1. public sealed abstract class FiguraGeometrica permits Rectangulo,Circulo{
2.     protected Point posicion;
3.     public FiguraGeometrica(Point p){
4.         posicion=p;
5.     }
6.     public abstract double getArea();
7. }
```

Las clases hijas se programan de la forma habitual, pero teniendo en cuenta que deben ser finales para que no exista la posibilidad de añadir más clases a la descendencia.

```
1. public final class Rectangulo extends FiguraGeometrica{
2.     private Dimension dimensiones;
3.     public Rectangulo(Point p, Dimension d){
4.         super(p);
5.         dimensiones=d;
6.     }
}
```

¹⁴ Tal es así que en lenguajes modernos como **Kotlin** las clases y los métodos por defecto son siempre finales y para que puedan ser abiertos necesitan marcarse con la palabra **open**.

¹⁵ Las clases selladas tienen interés en la programación funcional. Allí una clase sellada y sus clases hijas forman lo que se llama un **tipo de dato algebraico**. En Java aún no se le ha sacado todo el partido posible a las clases selladas y de momento solo sirven para limitar la herencia a unas clases conocidas.

```
7.     public double getArea(){
8.         return dimensiones.width*dimensiones.height;
9.     }
10. }
```

11.- Limitaciones de la herencia

En el apartado anterior hemos visto que muchas personas han usado mal la herencia de clases y que esto ha dado lugar a problemas. ¿Cuándo se usa mal la herencia? Básicamente ha habido tres causas:

- Se ha usado la herencia con el único objetivo de reutilizar código, haciendo que clases hijas representen conceptos que no tengan nada que ver en el mundo real con sus clases padre. Esto hace que cuando luego programemos surjan situaciones extrañas debido a la incoherencia del diseño, o fallos derivados de pasar a los métodos objetos que no son de aquello que se espera.

La solución a este problema es programar solo clases hijas cuando se cumpla el **principio de sustitución de Liskov (LSP)**, que nos dice: *“La clase A solo debe ser clase hija de la clase B si en cualquier lugar donde se pueda poner un objeto de tipo B se puede poner un objeto de tipo A sin problema”*.

- Se han hecho clases hijas de clases que no fueron diseñadas ni documentadas para ser clases padre. Esta situación nos lleva a clases hijas en las que aparecen errores difíciles de detectar cuando menos lo esperamos debido a no conocer el funcionamiento exacto de la clase padre.

La solución a este problema consiste en que las clases sean siempre finales y solo hacer clases abiertas cuando estemos 100% seguros de que van a tener clases hijas. No solo basta con esto, sino que además hay que documentar muy bien cómo debe ser programada una clase hija y poner como finales aquellos métodos que no tengamos previsto que sean sobreescritos.

- Las clases abiertas son vulnerables a un **finalizer attack**, que en algunos casos permite la ejecución de código sin permiso¹⁶.

La solución a este problema es sobreescibir en nuestra clase el método **protected void finalize()** para que no haga nada y ponerlo final para que no pueda ser sobreescrito.

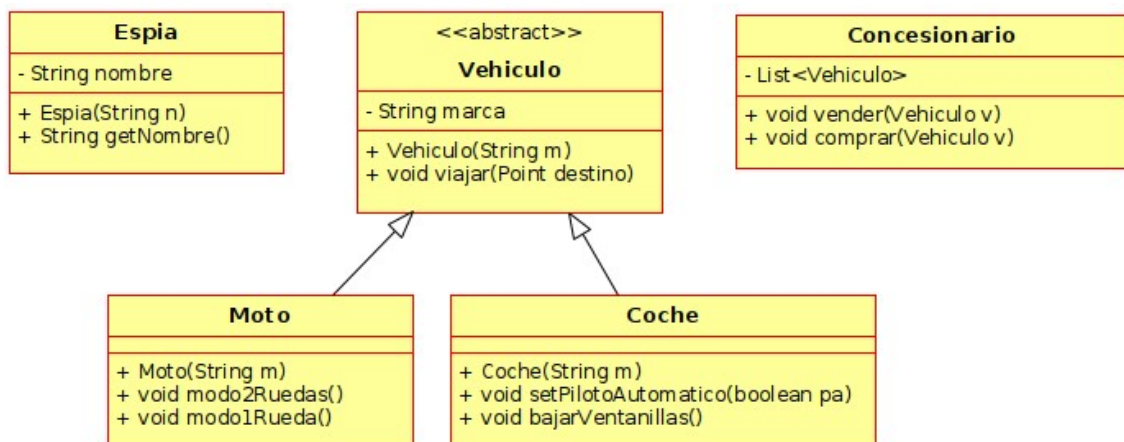
¹⁶ En la clase Object existe un método (hoy deprecado y reemplazado por los **Cleaners**) llamado **protected void finalize()**, que permite la ejecución de código justo antes de que el objeto sea eliminado de la memoria por el recolector de basura. Cualquier programador puede sobreescibir este método. Un finalizer attack aprovecha este método para poder conservar el objeto tras forzar una excepción en el constructor que se salte las protecciones, y poder llamar a los métodos del objeto abortado.

12.- Composición y agregación

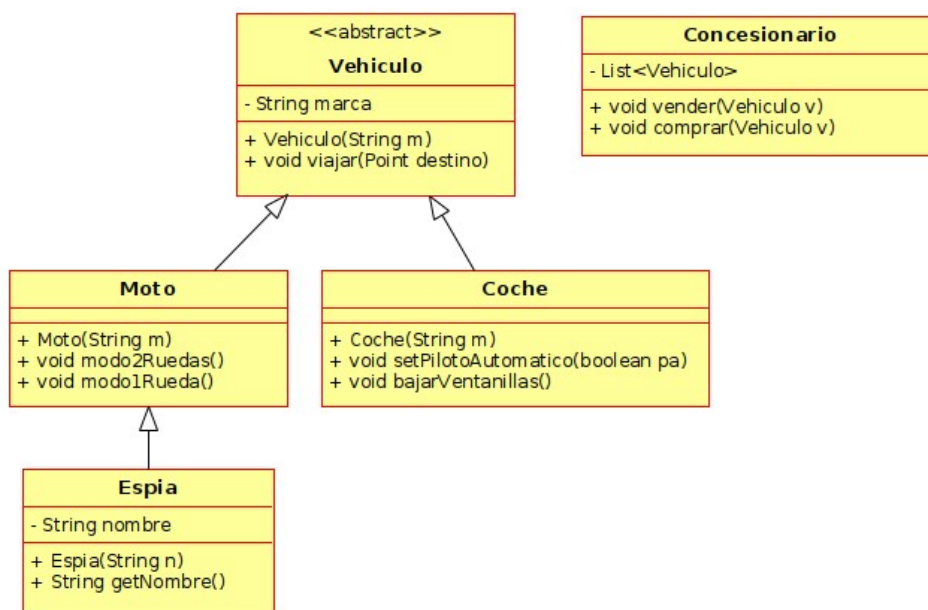
Como hemos visto, los principales problemas de la herencia se deben a que se ha usado en situaciones donde no se puede aplicar conceptualmente, o bien, nunca se planteó esa posibilidad por los diseñadores de las clases. ¿Qué alternativa hay entonces, al uso de la herencia cuando queremos reutilizar código y no es apropiado usar herencia?

La solución está en el uso de la **composición/agregación**¹⁷, en el que una clase “contiene” un objeto de la otra clase y delega en él.

Veamos un ejemplo de esto. Supongamos que tenemos este diagrama de clases en el que un Espía puede escapar en una moto o en un coche.



Si queremos que el espía escape en una moto, un **mal uso de la herencia** nos conduce a hacer que Espía sea una clase hija de la clase Moto.



¹⁷ La diferencia entre agregación y composición consiste en que los objetos de la clase

O sea, cuando programamos la clase Espía comenzamos así:

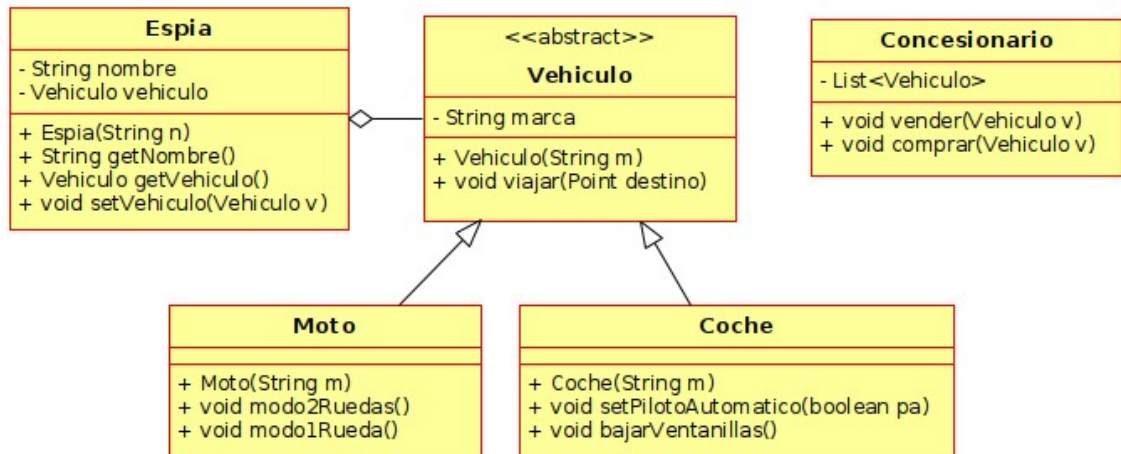
```
1. public class Espia extends Moto{
2.     ...
3. }
```

Pero esto, desde el punto de vista del mundo real no tiene sentido puesto que un Espía **no es** una moto. Además, como moto implementa Vehículo el espía automáticamente se convierte en un vehículo y pueda ser pasado a cualquier método que trabaje con vehículos, como por ejemplo los del concesionario, lo cual conduce a código potencialmente incorrecto. Por tanto, un programador de aplicaciones que use estas clases podría escribir este código:

```
1. Espia bond = new Espia("007");
2. bond.moda1Rueda();
3. Concesionario c=new Concesionario();
4. c.comprar(bond);
```

En este ejemplo tan sencillo ya puede verse que hay cosas que no tienen lógica y en programas grandes esta incoherencia puede ser causa de fallos muy difíciles de localizar. Además, el diseño limita que el espía pueda cambiar de moto o escapar en un coche, porque la herencia queda definida antes de que el programa se ponga en funcionamiento.

El diseño apropiado a esta situación es usar la relación “Espía **tiene** Vehículo”, que en este caso es de **agregación**, porque el espía y los vehículos pueden existir como objetos independientes uno de otro¹⁸.



Las relaciones de agregación y composición no tienen ninguna palabra en especial y basta con programar las clases de la forma habitual. Por ejemplo, la clase Espía sería así:

```
1. public class Espia{
2.     private String nombre;
3.     private Vehiculo vehiculo;
4.     public Espia(String n){
5.         nombre=n;
6.         vehiculo=null; // por defecto, el espía no tiene ningún vehículo
7.     }
8.     public String getNombre(){
9.         return nombre;
10.    }
```

¹⁸ En caso de que espía y vehículo fuesen objetos inseparables hablaríamos de **composición** y el rombo del diagrama de clases sería de **color negro**.

```

11. public Vehiculo getVehiculo(){
12.     return vehiculo;
13. }
14. public void setVehiculo(Vehiculo v){
15.     vehiculo=v;
16. }
17. }

```

No solo eso, ahora todo el código que se escriba tiene sentido desde el punto de vista de la situación del mundo real que se desea modela. Por ejemplo, ya podemos hacer esto:

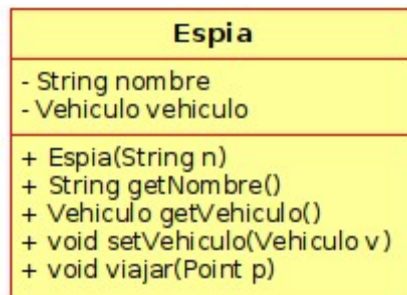
```

1. Espia bond=new Espia("007");
2. Vehiculo vehiculo=new Moto("Harley");
3. bond.setVehiculo(vehiculo);
4. bond.getVehiculo().modo1Rueda();
5. bond.getVehiculo().viajar(new Point(120,90));
6. Concesionario c=new Concesionario();
7. bond.vender(bond.getVehiculo());
8. Vehiculo vehiculo2 = new Coche("Mercedes");
9. bond.setVehiculo(vehiculo2);
10. bond.getVehiculo().viajar(new Point(10,50));

```

Una variación de la composición/agregación es la **delegación**, en la cual añadimos a la clase contenedora los métodos de la clase contenida y los programamos de forma que usen (o deleguen) en el objeto utilizado¹⁹.

Por ejemplo, en la clase Espía podemos añadir el método “viajar”, que delegará en el vehículo utilizado:



El método viajar de la clase Espía, simplemente delega en el método viajar de su objeto Vehículo, así:

```

1. public void viajar(Point p){
2.     if(vehiculo!=null){ // comprobamos que el espía tiene un Vehículo
3.         vehiculo.viajar(p); // delegamos en el método viajar de la propiedad vehiculo
4.     }
5. }

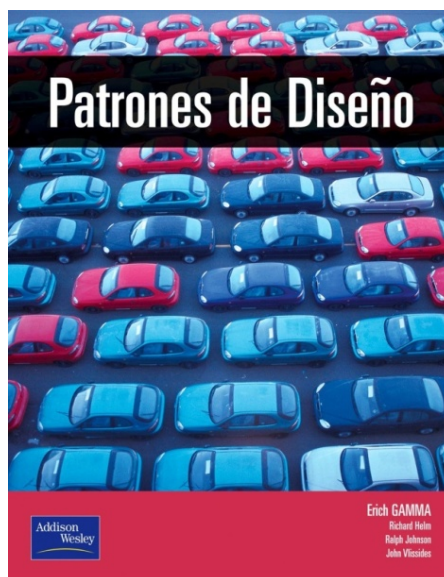
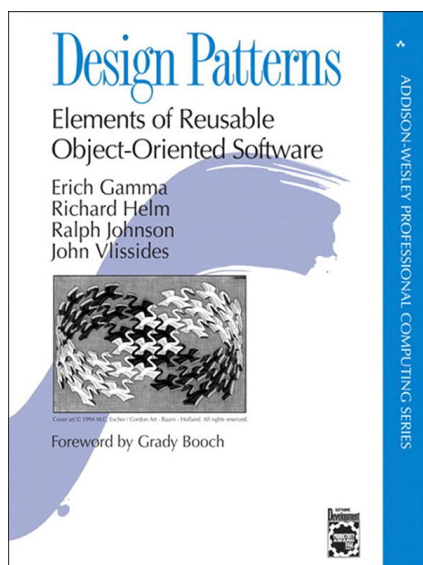
```

13.- Patrones de diseño

Para completar este tema de introducción al diseño y programación de un diagrama de clases vamos a comentar lo que son los **patrones de diseño**.

¹⁹ La delegación es algo que se usa tanto que lenguajes como **Ruby** o **Kotlin** tienen palabras específicas para facilitar su programación sin tener que escribir las llamadas al objeto delegado.

En el año 1994 apareció uno de los libros más importantes de toda la historia de la informática: **Design Patterns: Elements of Reusable Object-Oriented Software**, de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (The Gang of Four).



En este libro, sus autores identifican una serie de técnicas, llamadas “patrones de diseño”²⁰, que funcionan cuando hay que hacer un diagrama de clases y permiten diseñar sistemas escalables con fácil mantenimiento. Esto supuso una revolución en aquel momento, porque la programación orientada a objetos comenzaba a ser popular y ese libro ayudó a conocer cómo son los buenos diseños²¹. Aunque el libro original los explica para el lenguaje C++, estos patrones se han trasladado perfectamente a otros lenguajes como Java.

Durante el desarrollo de este tema, en los diagramas de los ejercicios han aparecido algunos de estos patrones. Vamos a comentarlos:

- **Builder:** Este patrón aparece en el diagrama **Persona**. Podemos ver como la clase Persona necesita muchos parámetros en su constructor y para un programador de aplicaciones sería muy pesado tener que pasar nada menos de 10 parámetros cada vez que quisiera crear una persona²².

La clase PersonaBuilder es un ejemplo de “builder”, que es una clase que nos facilita la creación de un objeto de otra clase (en este caso, un objeto Persona). Para ello, configuramos el objeto PersonaBuilder con los datos que necesitamos y el resto de parámetros tomará un valor por defecto. Cuando lo tenemos, usamos el método build, y tendremos un objeto Persona.

²⁰ Los autores identificaron estos patrones a partir de su experiencia en el desarrollo de muchísimos proyectos empresariales. En realidad, los patrones son cosas que ellos llevaron a la práctica en sus proyectos y que demostraron ser muy útiles.

²¹ Más adelante también se descubrieron las técnicas que no funcionan, que se denominan **antipatrones**. Uno de ellos, muy conocido por todos, es el **copy-paste**.

²² De hecho, usar directamente ese constructor sería un antipatrón (**telescopic constructor antipattern**)

Además, PersonaBuilder está diseñada de forma que se permita usar encadenando métodos a sí mismo, creando lo que se llama hoy día una “fluent api”²³.

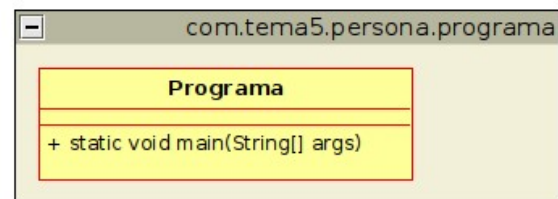
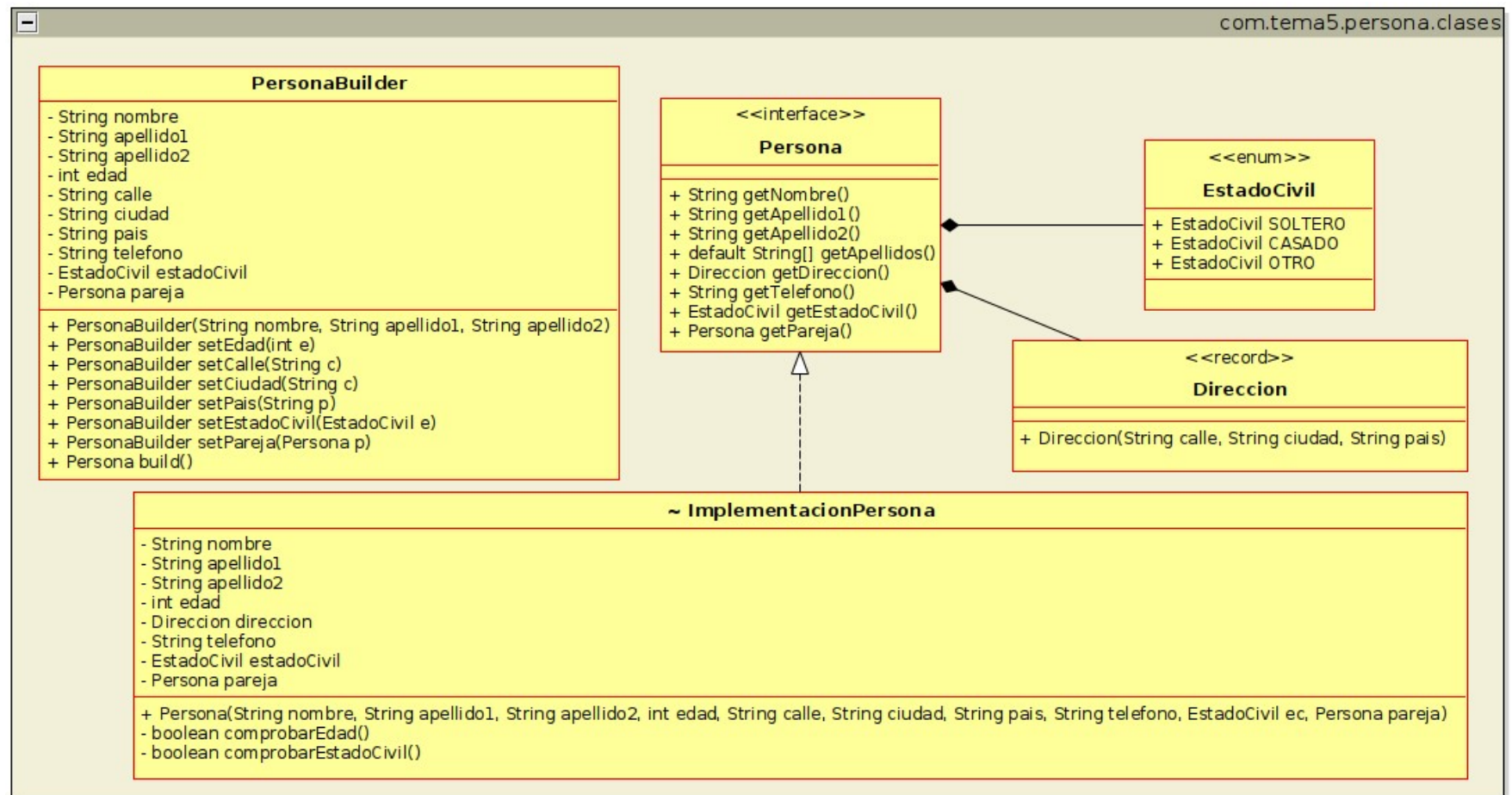
- **Factory:** Podemos ver este patrón en el diagrama **Efectos especiales**. Tenemos una interfaz EfectoEspecial que define un efecto especial y varias clases que la implementan. Esas clases no pueden ser instanciadas desde fuera del paquete de efectos, por lo que un programador de aplicaciones debe acudir a la clase EfectoFactory, que proporcionará un objeto EfectoEspecial de acuerdo al tipo que indique.

En general, la clase “factory” permite obtener un objeto que implementa una interfaz o hereda de una clase abstracta, sin que el programador de aplicaciones sepa realmente la clase verdadera de dicho objeto. Es muy usado para ocultar los detalles de la implementación a un programador de aplicaciones. O sea, al programador de aplicaciones le basta con tener un EfectoEspecial y poder llamar a su método aplicarEfecto, sin que le importe el tipo concreto del efecto.

- **Template:** Es un patrón que podemos ver en el diagrama **Animales**. La clase ContenedorAnimales nos define una “plantilla” que deben tener los tipos de contenedor. Esa clase obliga a las clases hijas a programar el método abstracto comprobarDisponibilidadPlaza. O sea, cada vez que queremos programar un tipo de contenedor de animales, creamos una clase que se ajuste a la plantilla definida en la clase ContenedorAnimales.

²³ Una **fluent api** es una librería diseñada para que se escriba en el código fuente resulte fácil de leer, tal y como si escribiéramos en lenguaje natural.

DIAGRAMAS DE CLASES: PERSONA



Persona: Interfaz que representa una persona

- Los getters devuelven el nombre, apellidos, dirección, teléfono, estado civil y pareja de la persona
- El método getApellidos devuelve un array donde el primer elemento es el primer apellido y el segundo el segundo apellido.

EstadoCivil: Enumeración que representa un estado civil. Las constantes de la enum son: SOLTERO, CASADO y OTRO

Dirección: Record que representa una dirección. Las propiedades de la enum son la calle, ciudad y país

ImplementacionPersona: Clase que proporciona la implementación de la interfaz Persona (cuidado con el modificador de acceso por defecto)

- El constructor inicializa las propiedades con los parámetros recibidos y a continuación llama a los métodos privados comprobarEdad y comprobarEstadoCivil para comprobar si la edad y el estado civil son correctos. En caso de que alguno de estos métodos devuelva false, se lanzará una IllegalArgumentException con el mensaje “Edad incorrecta” o “Estado civil incorrecto”
- El método comprobarEdad devuelve true si la edad es positiva
- El método comprobarEstadoCivil devuelve false si la persona tiene pareja (se entenderá que si la persona tiene pareja si esta es distinta de **null**) y su estado civil es SOLTERO, o si no tiene pareja y su estado civil es CASADO

PersonaBuilder: Sirve para facilitar al programador de aplicaciones la creación de un objeto de la clase ImplementacionPersona.

- El constructor de PersonaBuilder rellena sus propiedades nombre, apellido1 y apellido2 y pone los siguientes valores por defecto al resto de propiedades:
 - edad → 0
 - calle → null
 - ciudad → null
 - país → España
 - teléfono → null
 - estadoCivil → SOLTERO
 - pareja → null
- Los métodos setters de PersonaBuilder cambian el valor de la correspondiente propiedad, y devuelven el objeto PersonaBuilder que está siendo programado (this). Esto se hace así para permitir que el programador de aplicaciones pueda encadenar llamadas consecutivas al PersonaBuilder, creando una notación llamada “fluent api”.
- El método build crea un objeto ImplementacionPersona con los parámetros que PersonaBuilder tiene en sus propiedades, y lo devuelve.

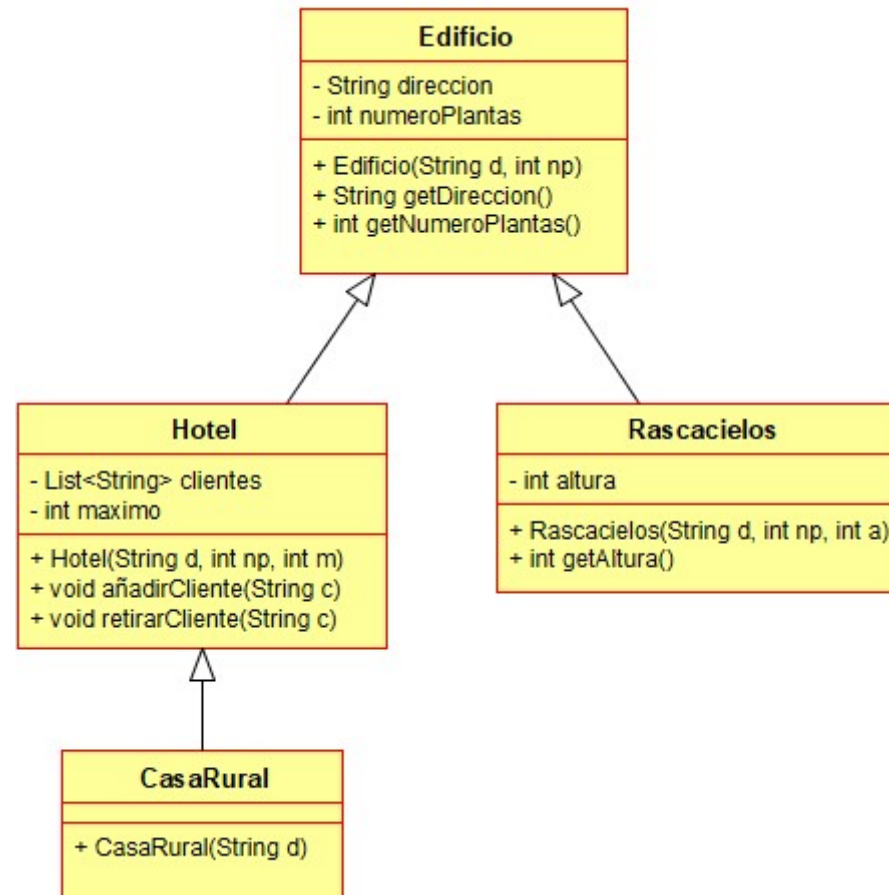
Programa: Esta clase estará en un paquete diferente al resto de clases del diagrama. Esta clase contiene un programa que simplemente usa la clase PersonaBuilder para crear dos objetos Persona con estos datos:

Nombre	Jaime
Apellidos	Lara Ramos
Teléfono	958000000
Ciudad	Granada
Edad	15

Nombre	María		
Apellidos	García Pérez		
Ciudad	Almería		
Pareja	Nombre	Juan	
	Apellidos	Molina Molina	
	Edad	30	
	País	Ecuador	

Intenta usar encadenamiento de métodos que puedas cuando uses el objeto PersonaBuilder.

DIAGRAMAS DE CLASES: EDIFICIOS



Edificio: Es una clase que representa un edificio con una dirección y una cantidad de plantas, que debe ser positiva.

- Sus getters devuelven el valor de las propiedades

Rascacielos: Es un tipo de edificio del que se guarda la altura, que debe ser positiva.

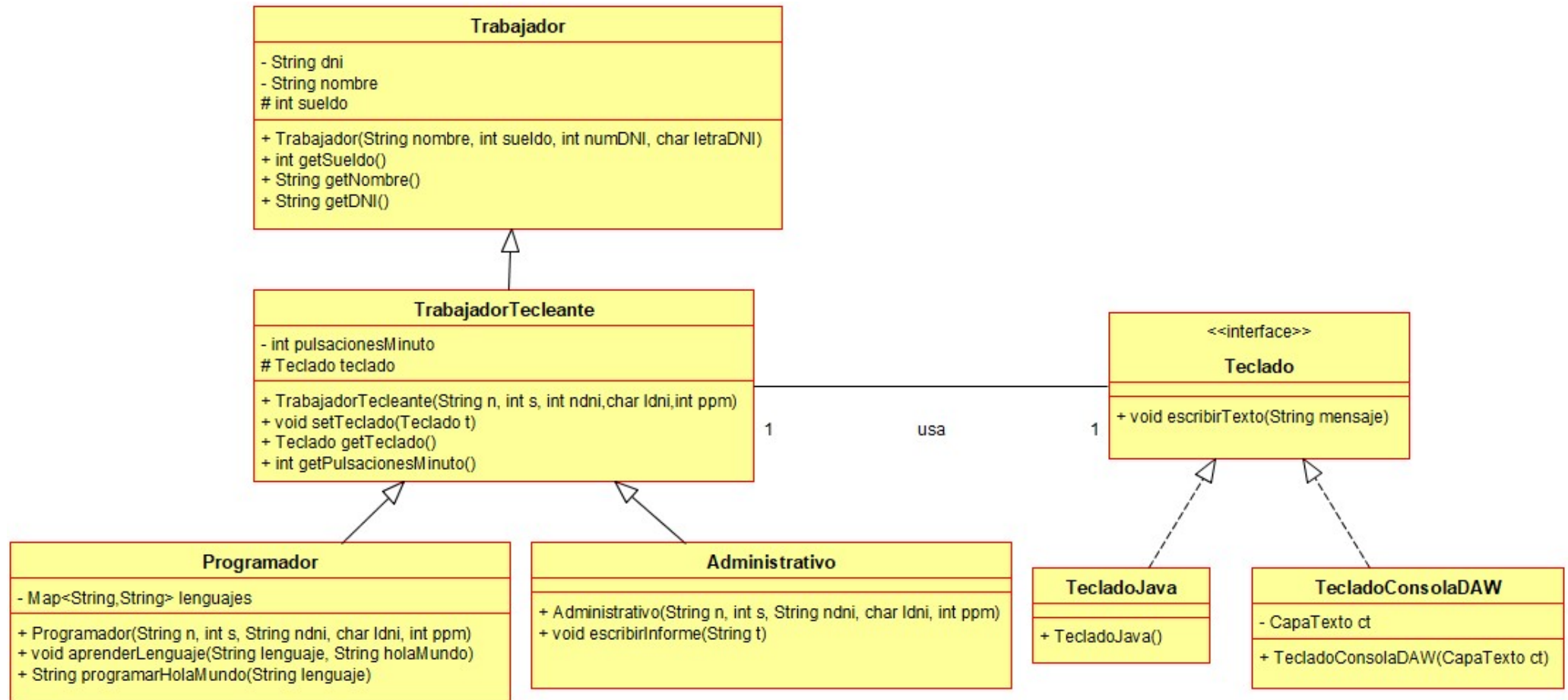
- El constructor lanzará una `IllegalArgumentException` si la altura del edificio es negativa

Hotel: Es un edificio que tiene una capacidad máxima de clientes. También tiene una lista con los nombres de los clientes que están en el hotel.

- El constructor lanzará una `IllegalArgumentException` si al crear un hotel se pasa una cantidad máxima de personas negativa
- `añadirCliente`: Añade a la lista de clientes el cliente cuyo nombre se pasa como parámetro. Si se añaden más clientes de lo que permite el hotel, lanzará una `IllegalStateException`
- `retirarCliente`: Retira del hotel el cliente cuyo nombre se pasa como parámetro. Si se intenta retirar un cliente que no está en el hotel, lanzará una `NoSuchElementException`

CasaRural: Es un tipo de hotel que solo tiene 3 plantas y admite hasta 6 clientes.

DIAGRAMAS DE CLASES: TRABAJADORES



Teclado: Es una interfaz que representa un teclado que puede ser usado por un empleado para escribir

TecladoJava: Es una implementación de teclado cuyo método escribirTexto recibe un mensaje y lo imprime por pantalla

TecladoConsolaDAW: Es una implementación de teclado que tiene como propiedad la capa de texto de una Consola DAW creada por el programa. En esta clase el método “escribirTexto” se programa escribiendo el texto en la capa de texto que tenemos en la propiedad.

Trabajador: Un Trabajador es un empleado que tiene un nombre, sueldo, dni (no es necesario comprobar que la letra es correcta) y métodos getters.

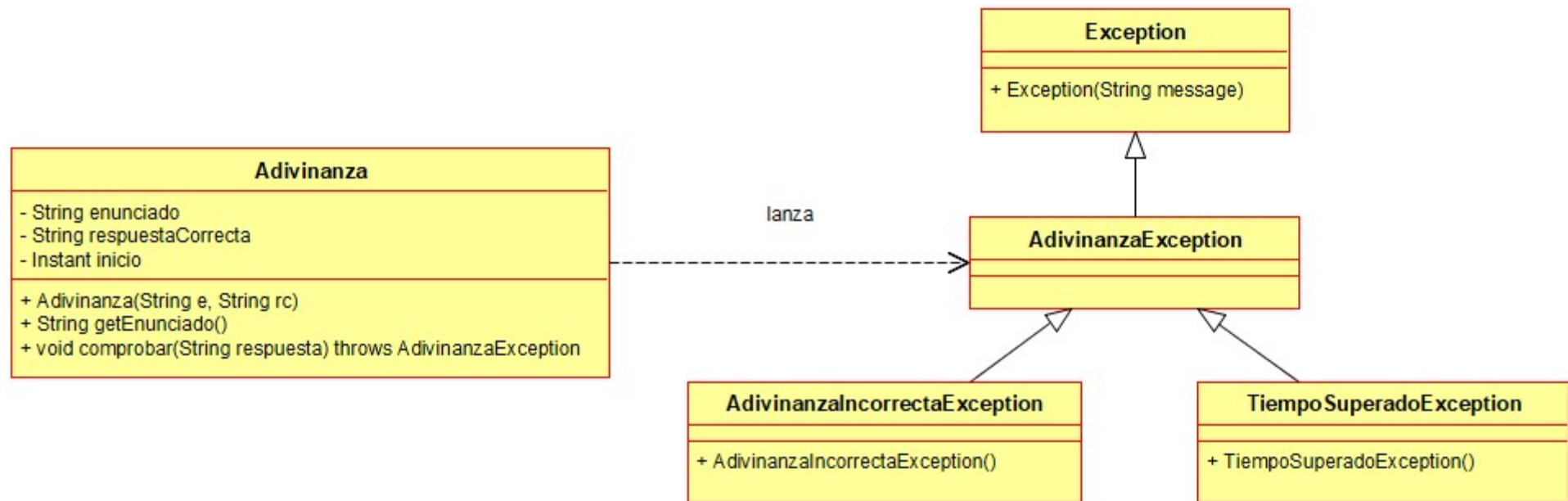
TrabajadorTecleante: Es un tipo de empleado que utiliza un teclado para trabajar, y además tiene unas determinadas pulsaciones por minuto.

Administrativo: Es un trabajador que escribe un informe en el teclado que está usando. Además, le pone la fecha y hora antes de escribirlo.

Programador: Un Programador es un empleado que conoce lenguajes de programación y sabe escribir el programa “Hola mundo” en ellos. Para eso dispone de un Map<String,String> donde guarda la asociación entre cada lenguaje conocido y el código fuente de su programa Hola Mundo.

- El método aprenderLenguaje recibe como parámetros el nombre de un lenguaje, y un String con el código fuente del programa “Hola mundo” en dicho lenguaje.
- El método programarHolaMundo recibe como parámetro el nombre de un lenguaje y nos devuelve el código fuente del programa Hola Mundo de dicho lenguaje. Si el el programador no conoce el lenguaje, se lanzará una IllegalStateException.

DIAGRAMAS DE CLASES: ADIVINANZAS



Adivinanza: Es una clase que representa una adivinanza que tiene que acertar el usuario. Posee un enunciado, un String con su respuesta correcta y también guarda el instante en que se llama al método “getEnunciado”. Este instante, al principio es null.

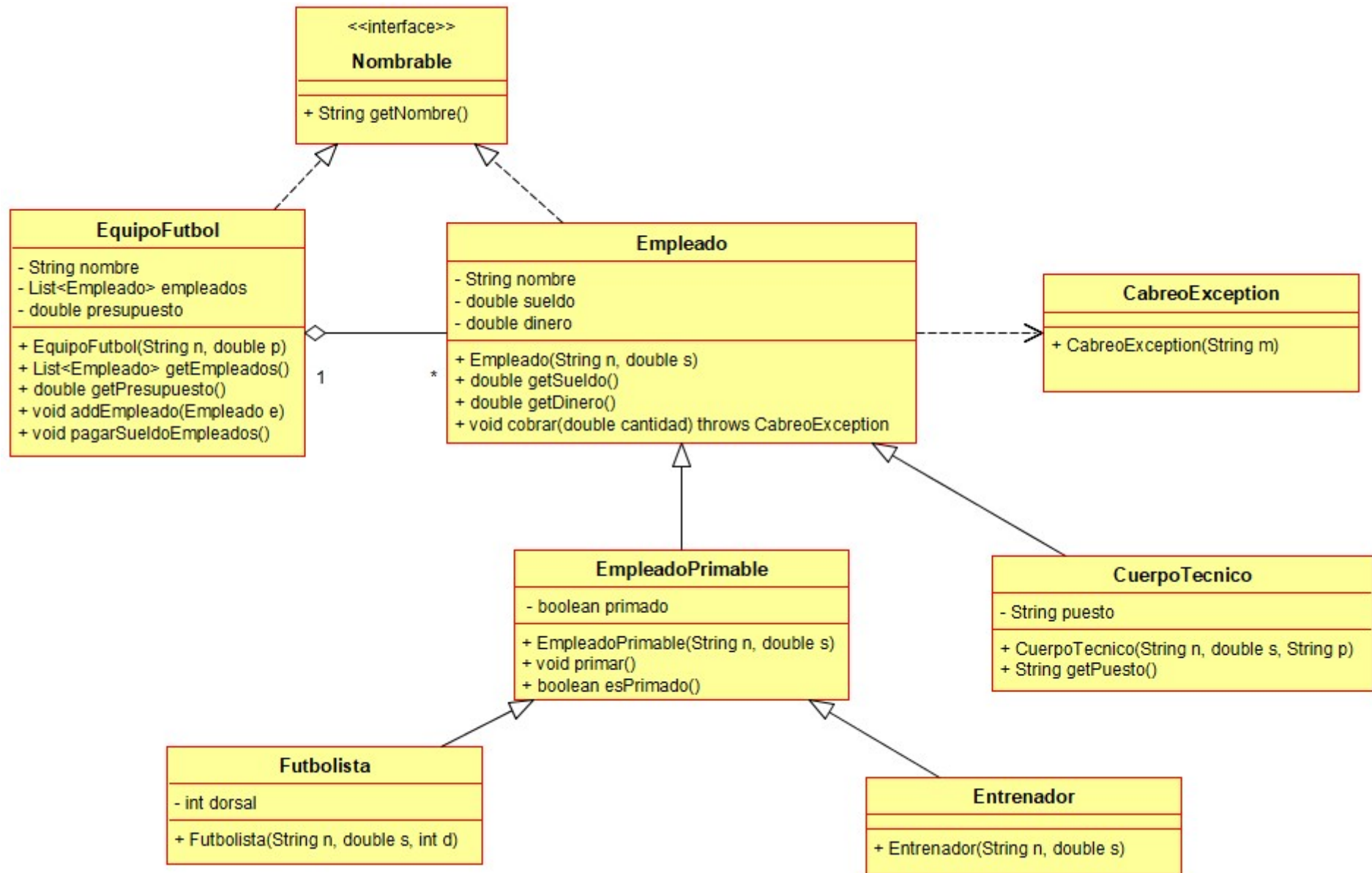
- Constructor: Crea una adivinanza con un enunciado y su respuesta correcta
- getEnunciado: Asigna la propiedad “inicio” con el instante actual y después devuelve el enunciado de la adivinanza
- comprobar: Comprueba si el String pasado como parámetro coincide con la respuesta correcta de la adivinanza. Se mirará el instante actual, y se comparará con el instante guardado en “inicio”. Si han pasado más de 30 segundos se lanzará una TiempoSuperadoException. En caso contrario, si la respuesta es correcta el método terminará sin hacer nada más, pero si es incorrecta se lanzará una AdivinanzaIncorrectaException.

AdivinanzaExcepcion: Clase que representa una excepción con una adivinanza

AdivinanzaIncorrectaException: Exception que se lanza si se falla una adivinanza

TiempoSuperadoException: Excepción que se lanza si se superan 30 segundos

DIAGRAMAS DE CLASES: EQUIPO DE FÚTBOL



Empleado: Es un empleado cualquiera del equipo de fútbol.

- nombre: El nombre del empleado
- sueldo: Dinero que gana el empleado
- dinero: La cantidad de dinero que tiene el empleado en su banco. Inicialmente un empleado no tiene dinero en su banco.
- cobrar: Método que ingresa al empleado la cantidad de dinero pasada como parámetro. Si dicha cantidad es menor que su sueldo, la ingresa, y a continuación lanza una `CabreoException`

CuerpoTécnico: Es un empleado que tiene asignado un puesto en el equipo, por ejemplo, entrenador de porteros, preparador físico, etc.

CabreoException: Tipo de excepción que se lanza cuando un empleado cobra menos que su sueldo. Su mensaje es siempre el mismo: “Al empleado ... no se le han pagado ... euros” (pon el nombre y la cantidad de su salario que no se le ha pagado).

EmpleadoPrimable: Clase que representa un empleado al que se le puede ofrecer una prima por buen rendimiento. Por defecto, ningún empleado está primado

- primar: método que prima al empleado
- esPrimado: método que devuelve true si el empleado ha sido primado.

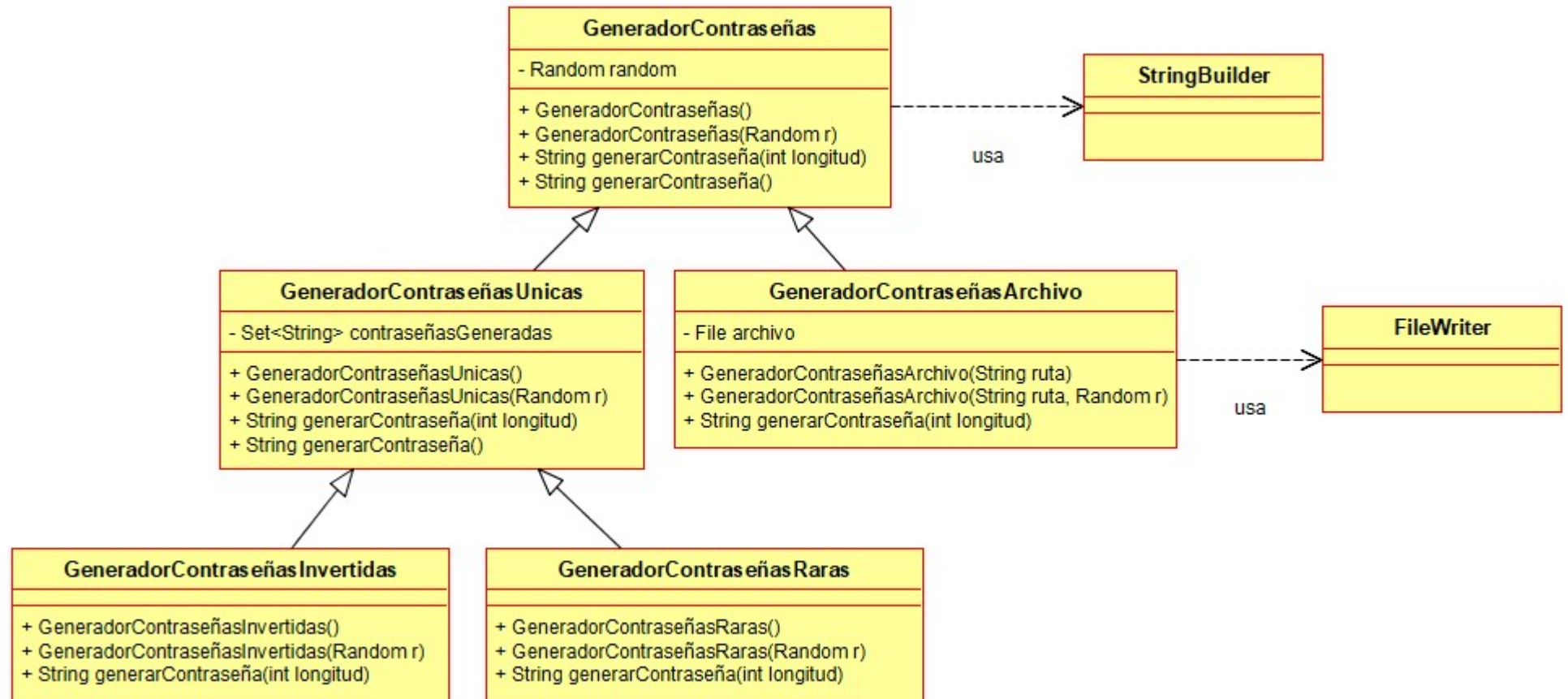
Futbolista: Tipo de empleado primable que tiene un dorsal en el equipo

Entrenador: Otro tipo de empleado primable.

EquipoFutbol: Es una clase que representa un equipo de fútbol

- nombre: El nombre del equipo de fútbol
- empleados: Una lista con todos los empleados del equipo. Inicialmente el equipo deberá crearse sin empleados.
- presupuesto: El total de dinero del equipo, del que deberá pagarse a los jugadores.
- addEmpleado: Añade un empleado al equipo. Solo debe haber un entrenador, por lo que si se intenta añadir más de un entrenador, se deberá lanzar una `IllegalArgumentException`. Igualmente, solo se admiten hasta 25 jugadores en plantilla.
- pagarSueldoEmpleados: Paga a cada empleado el sueldo pasado como parámetro y lo retira del presupuesto. Hay que tener en cuenta que a los empleados primados se les deberá pagar un 10% más de su sueldo. Si no hay presupuesto suficiente para pagar a un empleado, se dividirá el presupuesto restante entre el número de jugadores que falten por pagar, y se les abonará dicha cantidad. Se mostrará por pantalla el mensaje devuelto por la excepción correspondiente.

DIAGRAMA DE CLASES: CONTRASEÑAS



GeneradorContraseñas: Es una clase que genera contraseñas aleatorias y utiliza para ello la clase StringBuilder, de la librería estándar de Java.

- La propiedad “random” es el objeto que la clase usa internamente para generar números aleatorios que den lugar a las letras de las contraseñas.
- El primer constructor crea un generador de contraseñas, creando su Random interno.
- El segundo constructor crea un generador de contraseñas con el Random suministrado
- El primer generarContraseña genera una contraseña usando el total de caracteres pasado como parámetro. La contraseña se generará así:
 - Se empieza generando un número aleatorio entre 0, 1 y 2.
 - Si el número sale 0, generará aleatoriamente otro número entre 48 y 57 (ese es el rango de los códigos ASCII de los números).
 - Si el número sale 1, generará aleatoriamente otro número entre 65 y 90 (ese es el rango de los códigos ASCII de las letras mayúsculas).
 - Si el número sale 2, generará aleatoriamente un número entre 97 y 122, (ese es el rango de los códigos ASCII de las letras minúsculas).
 - Se añadirá a la contraseña el carácter cuyo código ascii se ha generado. Aunque no es imprescindible, se aconseja usar la clase StringBuilder para ir construyendo el String de la contraseña.
- El segundo generarContraseña genera una contraseña de 8 caracteres de longitud.

GeneradorContraseñasÚnicas: es un tipo de generador de contraseñas que no puede generar contraseñas repetidas.

- En las propiedades posee un Set<String> en el que se van guardando todas las contraseñas generadas.
- Se sobrescribirá el primer método generarContraseña para que las contraseñas se generan como en el ejercicio anterior, pero cada vez que se genera una, se compruebe que no está en el Set<String>, volviéndola a generar en caso de que sea así.
- Se sobrescribirá el segundo método generarContraseña, para que se genere una contraseña de 12 caracteres de longitud.

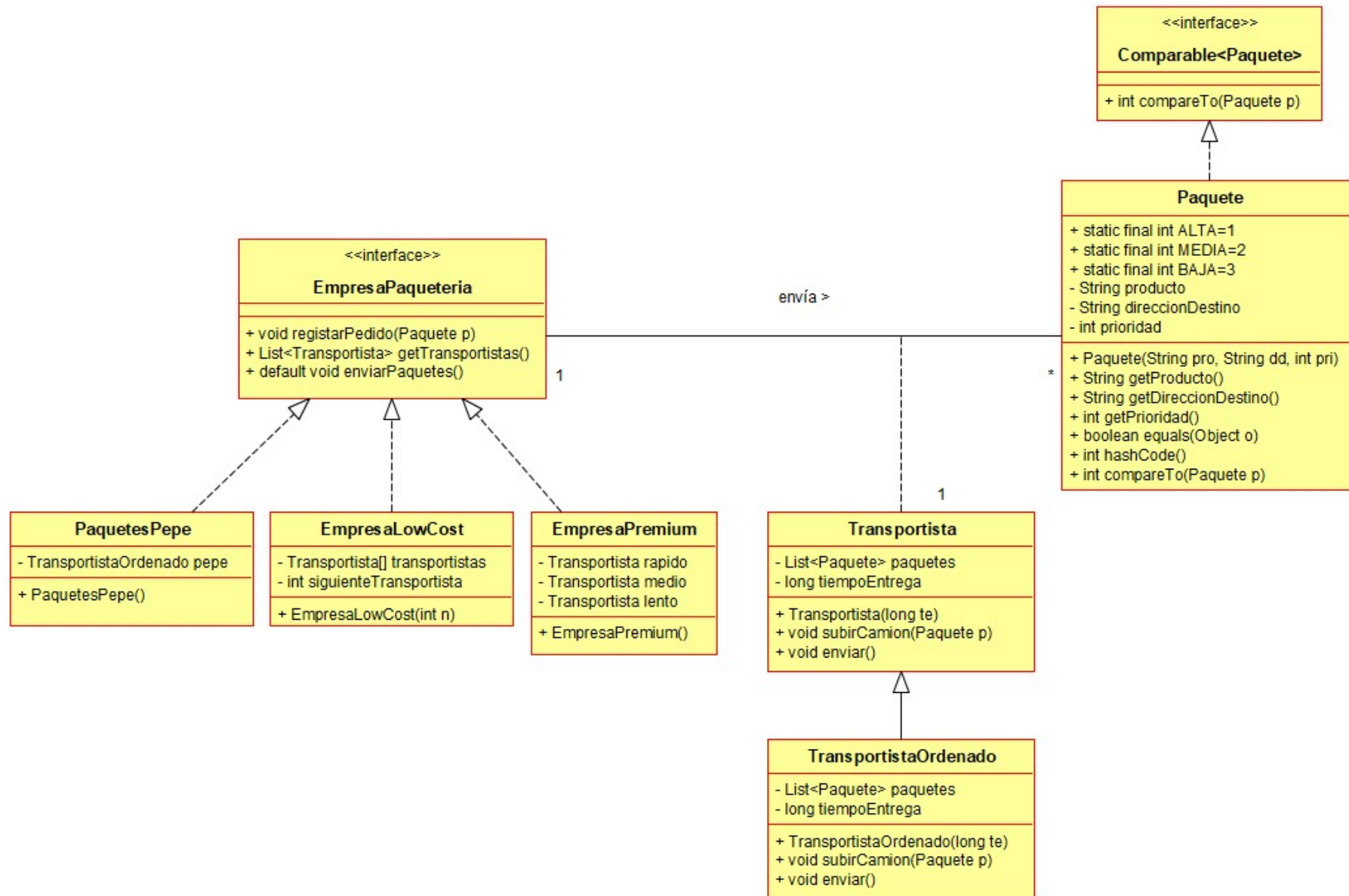
GeneradorContraseñasArchivo: es un tipo de generador de contraseñas que guarda todas las contraseñas que genera en un archivo.

- En las propiedades posee un File que indica la ruta del archivo donde se irán guardando las contraseñas generadas.
- Las contraseñas se generan como en el ejercicio 14, pero cada vez que se genera una, deberá añadirse al archivo (se añadirá al final del archivo una línea con la nueva contraseña). Se recomienda usar la clase FileWriter para programar el método.

GeneradorContraseñasInvertidas: es un tipo de generador de contraseñas únicas que genera una contraseña única y después le cambia todas las letras que estén en mayúsculas por minúsculas, y viceversa.

GeneradorContraseñasRaras: es un tipo de generador de contraseñas únicas que genera una contraseña única, y sustituye cada uno de sus caracteres por el que se obtiene sumando 122 a su código ascii. De esa forma, las letras y números de la contraseña generada son sustituidos por símbolos que no se corresponden con letras ni números.

DIAGRAMA DE CLASES: PAQUETERÍA



Paquete: Es una clase que representa un paquete que tiene un producto, una dirección de destino y un número que indica su prioridad.

- Las constantes de la clase representan los niveles de prioridad permitidos.
- El constructor crea un paquete con la dirección de destino y nivel de prioridad indicado. Si el nivel de prioridad no es válido, se lanzará una `IllegalArgumentException`.
- Dos paquetes se consideran iguales si tienen el mismo producto y la misma dirección de destino. Se recomienda usar el IDE para programar los métodos `equals` y `hashCode`.
- El método `compareTo` servirá para que los paquetes estén ordenados según su nivel de prioridad. Para ello, deberá devolver la diferencia entre la prioridad del paquete que se está programando y la del paquete que se recibe como parámetro.

Transportista: Clase que representa una persona que puede enviar un paquete a su destino. Posee una lista con todos los paquetes que tiene que enviar y además tiene una propiedad que indica el número de minutos que tarda en enviar un paquete.

- Constructor: crea un transportista que tarda en enviar un paquete el número de minutos que se pasa como parámetro, y no tiene ningún paquete asignado.
- `subirCamión`: recibe un paquete y lo guarda en su camión
- `enviar`: Simula el envío de paquetes por el transportista. Para ello el método recorre la lista de paquetes y por cada uno, hace una pausa de la cantidad de segundos indicada en la propiedad “`tiempoEntrega`” y luego muestra por pantalla el mensaje “El paquete (*producto*) con prioridad (*prioridad*) ha llegado a: (*destino paquete*)”. No se tendrá en cuenta la prioridad del paquete.

EmpresaPaquetería: Es una interfaz que define los requisitos que tiene que tener una clase para ser considerada una empresa de paquetería.

- `registrarPedido`: Consiste en aceptar un pedido y asignarlo a un repartidor
- `getTransportistas`: Devuelve una lista con los transportistas que trabajan para la empresa de transportes.
- `enviarPaquetes`: Es un método default que recorre la lista de transportistas y les ordena que envíen los paquetes que tienen asignados.

TransportistaOrdenado: Es un tipo de transportista, que cuando recibe la orden de enviar los paquetes, primero ordena la lista de paquetes por prioridad. A continuación, los envía de la forma indicada en el ejercicio anterior.

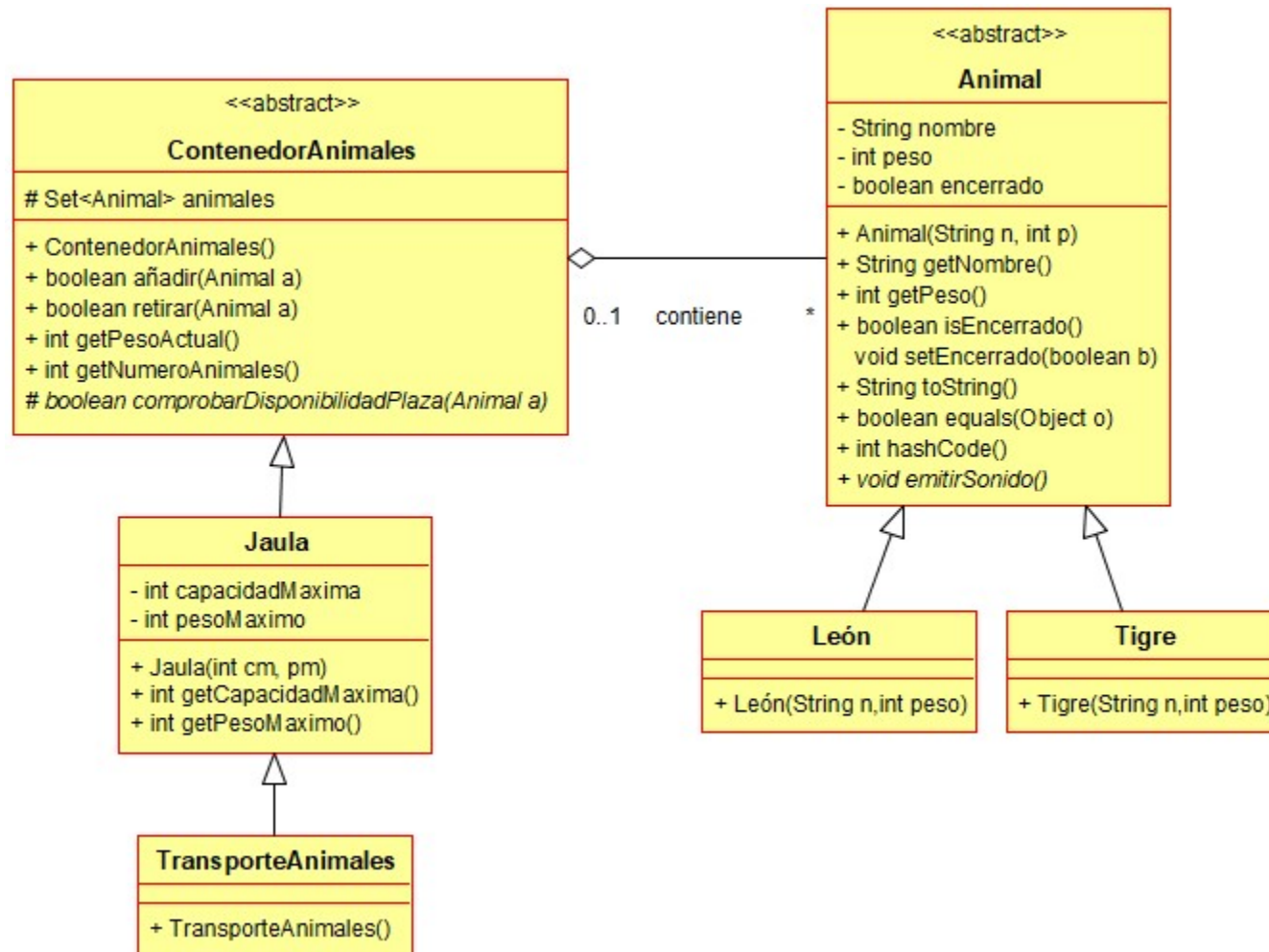
PaquetesPepe: Es una empresa de paquetería en la que solo hay un transportista (Pepe), que tiene la costumbre de ordenar los paquetes por prioridad antes de enviarlos, y tarda 30 minutos en enviar cada paquete.

EmpresaPremium: Es una empresa de paquetería en la que hay tres transportistas que tardan 10 minutos, 25 minutos y 50 minutos en enviar un paquete. Los paquetes se asignan a los transportistas según su prioridad, de forma que los paquetes de prioridad alta se asignan al transportista más rápido y así sucesivamente.

EmpresaLowCost, que es una empresa de paquetería en la que guarda una lista de transportistas y los paquetes se asignan de forma cíclica a los transportistas.

- Constructor: Crea una empresa con la cantidad de transportistas pasada como parámetro. Cada transportista tarda un tiempo entre 40 y 80 minutos en entregar un paquete. La propiedad “`siguienteTransportista`” guarda la posición del siguiente transportista al que se le asignará un paquete.
- `registrarPedido`: asigna el paquete al transportista que toca, según lo que haya en la propiedad “`siguienteTransportista`”, que pasará al siguiente.

DIAGRAMAS DE CLASES: ANIMALES



Animal: Es un animal que tiene un nombre, un peso y puede estar o no encerrado en un contenedor de animales.

- Haz que el método setEncerrado tenga modificador de acceso por defecto
- Sobreescribe los métodos heredados de la clase Object de esta forma:
 - toString: Muestra el nombre del animal y su peso
 - equals y hashCode: Se considera que un animal coincide con otro si ambos son objetos de la misma clase y además coincide su nombre de animal. Usa el IDE para programar estos métodos.
- emitirSonido: Método abstracto que muestra por pantalla un mensaje con el sonido del animal.

León y Tigre: clases que heredan de Animal y sobreescriben “emitirSonido”.

ContenedorAnimales: representa a cualquier cosa que pueda guardar un conjunto de animales.

- La propiedad “animales” es un conjunto con los animales que hay en el contenedor.
- añadir: Añade un animal al contenedor, solo si hay una plaza disponible para él. Esto nos lo devuelve el método auxiliar “comprobarDisponibilidadPlaza”. En caso de añadirlo, devuelve true y si no, devuelve false. En caso de añadir el animal, este deberá ser marcado como enjaulado.
- retirar: Retira el animal del contenedor. Si el animal pasado como parámetro no está en el contenedor, devuelve false. En otro caso, devuelve true.
- getPesoActual: Devuelve el peso de todos los animales que hay en el contenedor.
- getNumeroAnimales: Devuelve el número de animales que hay en el contenedor.
- comprobarDisponibilidadPlaza: Se implementará en las clases hijas para saber si el animal recibido como parámetro puede ser añadido o no, al contenedor.

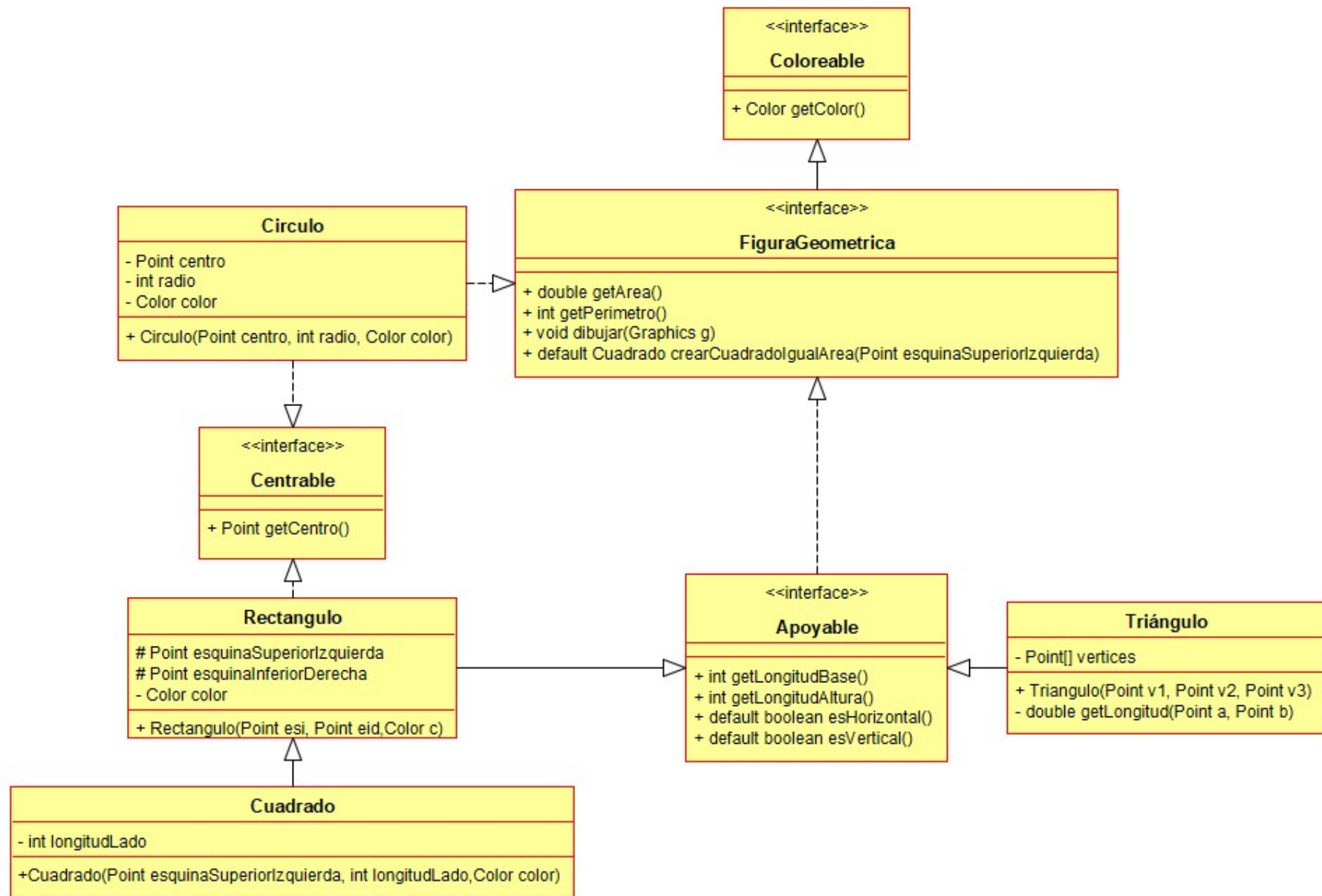
Jaula: Clase que representa una jaula que admite una capacidadMáxima de animales, pero sin que se supere un pesoMáximo.

- Se sobreescribirá el método “comprobarDisponibilidadPlaza” de forma que un animal se podrá añadir si no se supera el número máximo de animales permitido y tampoco se supera el peso máximo permitido.

TransporteAnimales: Clase que representa un vehículo especial que solamente puede admitir hasta 6 animales del mismo tipo, sin que se superen los 500kg de peso.

- Se sobreescribirá el método “comprobarDisponibilidadPlaza” de forma que un animal se podrá añadir si no se supera el número máximo de animales permitido, tampoco se supera el peso máximo permitido y su tipo es el mismo de los demás.

DIAGRAMAS DE CLASES: FIGURAS GEOMÉTRICAS



Coloreable: Es cualquier cosa que tiene un color.

Centrable: Es un objeto que tiene un centro y su método getCentro devuelve un Point con las coordenadas del centro del objeto.

FiguraGeométrica: Es una figura geométrica cualquiera.

- getArea: devuelve el área de la figura
- getPerímetro: devuelve el perímetro de la figura
- dibujar: recibe un objeto Graphics y dibuja con él la figura
- crearCuadradoIgualArea: Es un método default que devuelve un cuadrado que tenga igual área que la figura. Dicho cuadrado tiene como esquina superior izquierda el Point pasado como parámetro.

Apoyable: Es una figura geométrica que tiene una base y una altura.

- getLongitudBase y getLongitudAltura: devuelven dichas longitudes.
- esHorizontal: método que devuelve true si la base es mayor que la altura.
- esVertical: método que devuelve true si la altura es mayor que la base.

Circulo: Clase que representa un círculo e implementa todas las interfaces asociadas.

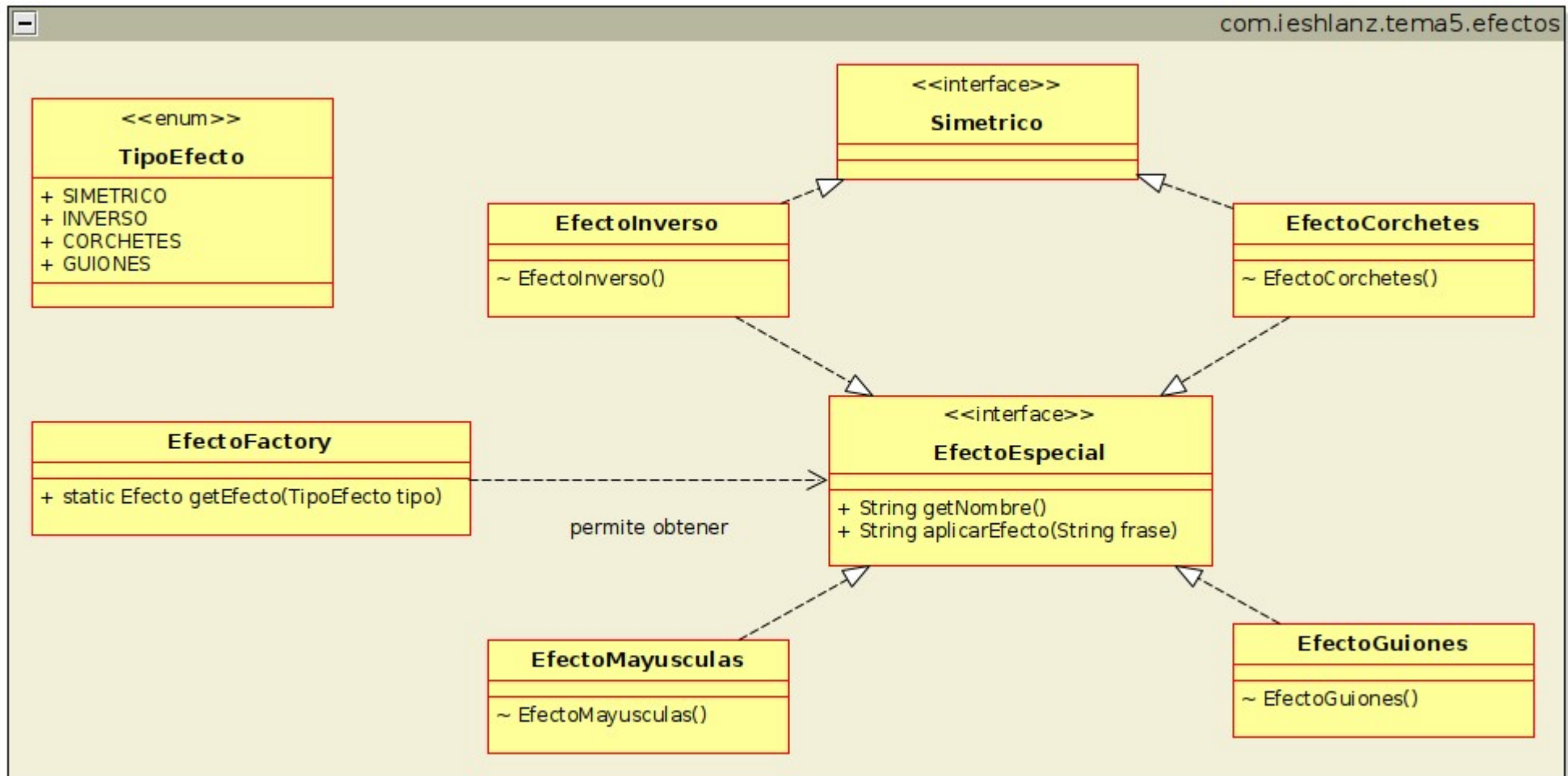
Rectángulo: Clase que representa un rectángulo definido por los dos puntos que se pasan como parámetro. Uno tiene las coordenadas de la esquina superior izquierda del rectángulo y el otro las de la esquina inferior derecha.

Cuadrado: Tipo de rectángulo que tiene todos sus lados iguales. Está definido mediante la longitud de su lado y el punto cuyas coordenadas son la esquina superior izquierda del cuadrado.

Triángulo: es un triángulo definido por tres puntos que son sus tres vértices, y su color siempre es azul.

- Constructor: Guarda los tres vértices recibidos en la propiedad que tiene como parámetro. Dicha propiedad deberá inicializarse con un array para guardar los tres vértices.
- getLongitud: Es un método auxiliar que nos da la longitud del segmento que une los puntos recibidos. Deberás buscar en Internet cómo calcular dicha longitud.
- getPerimetro: Se recomienda que se utilice el método auxiliar getLongitud
- getArea: Se recomienda usar el método auxiliar getLongitud y la **fórmula de Herón**, que deberás buscar en Internet.

DIAGRAMAS DE CLASES: EFECTOS ESPECIALES



EfectoEspecial: Clase que representa un efecto especial que puede aplicarse sobre una frase. En realidad el efecto lo que hace es transformar la frase en otra.

- El método getNombre devuelve el nombre del efecto especial
- El método aplicarEfecto se implementará en las clases hijas y lo que hace es transformar la frase recibida en otra, que será la “aplicación del efecto”
- La clase EfectoEspecial tendrá un bloque inicializador estático en el que rellenará la propiedad “EFECTOS” con un objeto de cada uno de los efectos especiales (una vez vayan siendo creados).

Simétrico: Es una interfaz sin métodos²⁴ que indica que un efecto especial es simétrico, es decir, si se aplica dos veces se vuelve a obtener el mismo texto de partida.

Resto de clases: Los efectos especiales se programan de la forma que indica la tabla, teniendo además en cuenta que en su constructor, se deberá mostrar por pantalla el mensaje “Creado el efecto especial (*nombre*)”. Ten además en cuenta que los efectos tienen el modificador de acceso por defecto.

Clase	Nombre del efecto	Lo que hace el efecto	Ejemplo
EfectoMayúsculas	Pasar a mayúsculas	Pasa a mayúsculas la frase.	“a jugar” → “A JUGAR”
EfectoInverso	Inversión de letras	Devuelve la frase al revés.	“a jugar” → “raguj a”
EfectoGuiones	Separador de guiones	Cambia los espacios por guiones bajos.	“a jugar” → “a_jugar”
EfectoCorchetes	Envoltura de corchetes	Encierra la frase entre [y], pero si el texto empieza y termina por ellos, los elimina.	“a jugar” → “[a jugar]” “[a jugar]” → “a jugar”

Nota: Usando los métodos apropiados de las clases *String* o *StringBuilder* los efectos se programan fácilmente

²⁴ Las interfaces sin métodos se llaman **marker interfaces**, y sirven para destacar que una clase posee alguna característica que la hace especial por algo.