

# Fundamentos de Programación

---

## Tema 5: Programación avanzada de clases

---

### Contenidos

1.- Introducción .....	2
2.- Encapsulamiento.....	2
3.- Herencia .....	6
4.- Sobreescritura de métodos.....	17
5.- Métodos de la clase Object.....	22
6.- Clases abstractas.....	27
7.- Polimorfismo .....	34

## Tema

## 5

**PROGRAMACIÓN AVANZADA DE CLASES**

*Los conceptos de programación orientada a objetos que hemos estudiado nos permiten realizar muchos programas. Sin embargo, para conseguir un nivel de calidad profesional y los máximos beneficios es necesario utilizar conceptos más avanzados que vamos a estudiar en este tema.*

**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.

**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. }
```

En algunas situaciones, es conveniente “relajar” un poco el nivel de encapsulamiento, y permitir que haya más personas que tengan permiso para usar las propiedades de una clase, pero sin que llegue a ser el programador de la aplicación. Eso lo vamos a conseguir con el modificador `protected` o el modificador de acceso por defecto (también llamado *package*).

**Ejercicio 1:** La siguiente clase está mal encapsulada. Prográmala de forma que esté bien encapsulada, poniendo todas las protecciones a las propiedades que creas necesarias. Si quieres, puedes añadir a la clase métodos privados que te ayuden a realizar el trabajo.

Sorteo
+ double dineroCupon + int pagosRealizados + int pagosPendientes + int totalParticipantes + double dineroRecogido
+ Sorteo(double dineroCupon, int totalPersonas) + void registrarPago() + void apuntarNuevoParticipante()

- La clase representa un sorteo en el que participan la cantidad de personas indicada en la propiedad “totalParticipantes”.
- El constructor crea un sorteo cuyos cupones cuestan la cantidad indicada y en el que participan inicialmente (luego se puede ampliar) la cantidad de personas indicada.
- La propiedad “dineroCupón” guarda el dinero que cuesta un cupón del sorteo.
- La propiedad “pagosRealizados” guarda la cantidad de personas que han pagado.
- La propiedad “pagosPendientes” guarda la cantidad de personas que faltan por pagar.
- La propiedad “dineroRecogido” guarda el total de euros que se han cobrado.
- El método “registrarPago” es llamado cada vez que una persona paga su cupón.

- La cantidad de gente que participa en el sorteo no es cerrada. Con el método “apuntarNuevoParticipante” hacemos que una persona no contemplada inicialmente se apunte en el sorteo. Dicha persona aún tendrá que pagar su cupón.

**Ejercicio 2 :** La siguiente clase está mal encapsulada. Prográmala de forma que esté bien encapsulada, poniendo todas las protecciones a las propiedades que creas necesarias. Si quieres, puedes añadir a la clase métodos privados que te ayuden a realizar el trabajo.

Compras
+ double presupuesto + List<Double> precios
+ Compras(double presupuesto) + void registrarCompra(double dinero) throws Exception

- La clase representa las compras que vamos haciendo, de forma que podemos comprar lo que queramos pero sin pasarnos de un presupuesto.
- La propiedad “presupuesto” es la cantidad dinero total que podemos gastarnos en compras.
- La propiedad “precios” guarda una lista con los precios de las cosas se han comprado.
- El constructor crea un objeto para hacer compras y recibe el presupuesto disponible.
- El método “registrarCompra” recibe un precio y registra en el objeto una compra con dicho precio. En caso de que no haya presupuesto para registrar una compra, se lanzará una Exception con el mensaje “No hay suficiente presupuesto. Faltan ...” euros.

**Ejercicio 3 :** La siguiente clase está mal encapsulada. Prográmala de forma que esté bien encapsulada, poniendo todas las protecciones a las propiedades que creas necesarias. Si quieres, puedes añadir a la clase métodos privados que te ayuden a realizar el trabajo.

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 representa un rectángulo que tiene una base, una altura y un área
- El método constructor crea un rectángulo a partir de una base y una altura
- El método setBase cambia la base del rectángulo
- El método getAltura cambia la altura del rectángulo
- El método getArea devuelve el área del rectángulo
- Los métodos getBase y getAltura devuelven la base y altura del rectángulo

## 2.1.- El modificador de acceso por defecto

Cuando una propiedad o método tiene el modificador de acceso por defecto (en Java sería no poner ningún modificador a la propiedad), **podrán acceder libremente a ella los programadores que hagan 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 el programador de una aplicación no puede tener acceso a ella. Sin embargo, dicho modificador permite que cualquier persona que programe una clase dentro del paquete **daw.oficina** pueda 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.

**Ejercicio 4 :** ¿En qué situaciones pondrías a una propiedad el modificador por defecto?

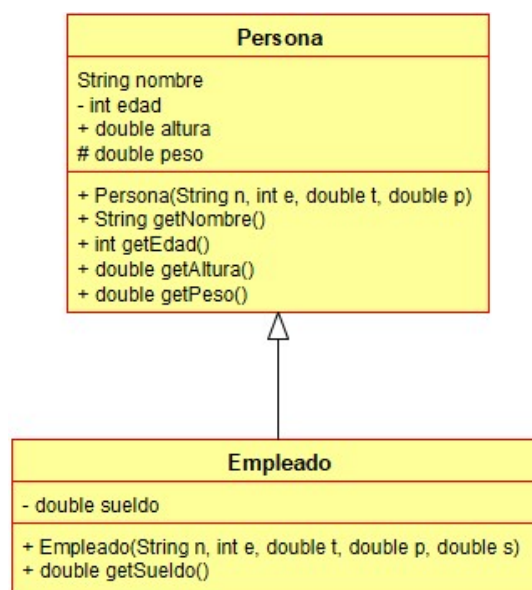
- a) Estoy haciendo un programa yo solo y hago muchas clases en el mismo paquete.
- b) Hago una librería de clases para ponerla en Internet y que todo el mundo la use.
- c) Le dejo a mi mejor amigo una clase que he hecho para que la use en su programa.
- d) Estoy haciendo una clase y tengo compañeros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

### 3.- Herencia

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 siempre es simple, lo que significa que una clase solo puede tener como mucho una clase padre.
- 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:



La principal dificultad a la hora de programar la herencia es reconocer que **el objeto que estamos programando automáticamente es un objeto de la clase padre**. Muchas personas no son capaces de entender esto y cuando van a programar la clase hija, piensan que necesitan crear objetos de la clase padre, cuando es innecesario.

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. }
```

A continuación, utilizamos 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. Por tanto, a partir de ese momento, todas las propiedades y métodos que tiene Persona automáticamente los adquiere la clase Empleado, como si fuesen propiedades y métodos que hubiésemos programado en ella. Sin embargo, solo podremos acceder a aquellas propiedades y métodos cuyo modificador de acceso nos de permiso, de esta forma:

- 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).

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 empleado<sup>1</sup>.

```
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.

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”.

---

<sup>1</sup> 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.

+ 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 nuestro constructor llama otro constructor de su clase por medio de **this**

Por tanto, la programación del método constructor 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 segundo constructor:

- + 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 “a mano”, así:

```
1. public class Empleado extends Persona{
2.     private double sueldo;
3.
4.     public Empleado(String n, int e, double t, double p, double s){
5.         super(n,e,t,p); // llama al constructor de la clase padre
6.         sueldo=s;      // inicializa la propiedad de la clase hija
7.     }
8.
9.     public Empleado(String n, double t, double p){
10.        super(n,18,t,p); // llama al constructor de la clase padre
11.        sueldo=1000;    // inicializa la propiedad de la clase hija
12.    }
13.
14. }
```

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



```

1. public class Empleado extends Persona{
2.     private double sueldo;
3.
4.     public Empleado(String n, int e, double t, double p, double s){
5.         super(n,e,t,p); // llama al constructor de la clase padre
6.         sueldo=s;       // inicializa la propiedad de la clase hija
7.     }
8.
9.     public Empleado(String n, double t, double p){
10.        this(n,18,t,p,1000);
11.    }
12.
13. }

```

De esta forma, no es necesario escribir super en el nuevo constructor, porque lo que hace es llamar a this, y el constructor invocado 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 5 :** Consulta el diagrama de clases **Edificios** y programa las clases Edificio y Rascacielos.

- Un Edificio tiene una dirección y una cantidad de plantas, que debe ser positiva.
- El Rascacielos es un tipo de edificio del que se guarda la altura, que debe ser positiva. En caso contrario se lanzará una `IllegalArgumentException`.

**Ejercicio 6 :** Siguiendo con el mismo proyecto y diagrama del ejercicio anterior, programa las clases Hotel y CasaRural.

- Un 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.
  - Si al crear un hotel se pasa una cantidad máxima de personas negativa, se lanzará una `IllegalArgumentException`
  - Si se añaden más clientes de lo que permite el hotel, el método `añadirCliente` lanzará una `IllegalStateException`
  - Si se intenta retirar un cliente que no está en el hotel, el método `retirarCliente` lanzará una `NoSuchElementException`
- La CasaRural es un tipo de hotel que solo tiene 3 plantas y admite hasta 6 clientes.
- La altura del Rascacielos es de 10 metros, más 3 metros por cada planta

**Ejercicio 7 :** Consulta el diagrama de clases **Trabajadores** y programa la interfaz Teclado y las clases TecladoJava y TecladoConsolaDAW.

- Un Teclado es cualquier cosa que un trabajador puede usar para escribir.
- Un TecladoJava simplemente recibe un mensaje y lo imprime por pantalla.
- Un TecladoConsolaDAW 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.

**Ejercicio 8 :** Siguiendo con el mismo proyecto y diagrama del ejercicio anterior, programa las clases Trabajador, TrabajadorTecleante y Administrativo:

- Un Trabajador es un empleado que tiene un nombre, sueldo, dni (no es necesario comprobar que la letra es correcta) y métodos getters.
- Un TrabajadorTecleante es un tipo de empleado que utiliza un teclado para trabajar, y además tiene unas determinadas pulsaciones por minuto.
- Un Administrativo es un trabajador que escribe un informe en el teclado que está usando. Además, le pone la fecha y hora antes de escribirlo.

**Ejercicio 9 :** Programa la clase 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.

### **3.2.- El modificador de acceso protected**

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

Sin embargo, el modificador protected también tiene un uso menos conocido (de hecho, muchos programadores profesionales lo ignoran), y es que 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?	✗	✗	✗	✓

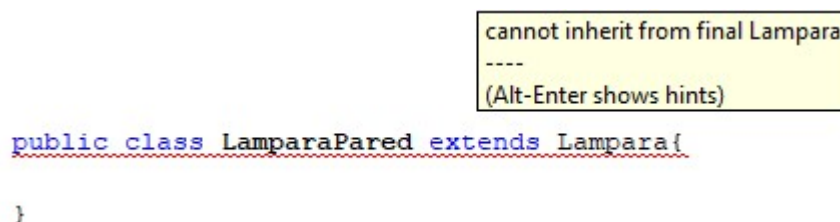
A la vista de esta tabla, podemos ver que el modificador de acceso más restrictivo (según la cantidad de personas que pueden ver la propiedad o método) sería el private, a continuación el modificador por defecto, luego el protected y por último el public, que daría acceso a todo el mundo.

### **3.3.- 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:

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

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:

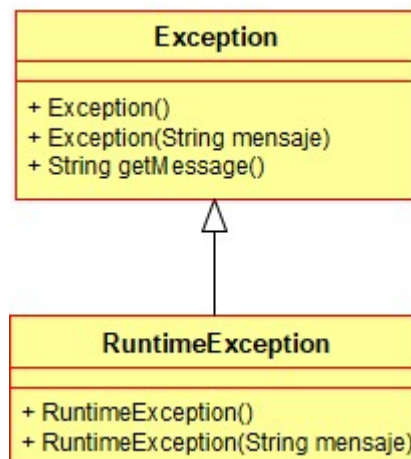


```
public class LamparaPared extends Lampara{  
  
}
```

Las clases finales se usan para impedir que haya programadores que creen clases hijas que les puedan añadir o modificar métodos. De esta forma las clases se conservan de la forma original que pensaron sus desarrolladores. Muchas de las clases que vienen con Java son finales, como por ejemplo la clase String. Esto hace que todos los programadores de Java cuando vean String sepan que se trata de la clase original y no de una clase hija.

### **3.4.- Creación de excepciones propias**

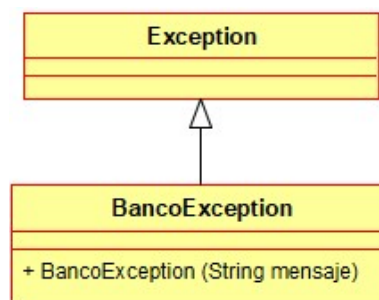
En este apartado vamos a estudiar una aplicación de la herencia muy utilizada. Es muy habitual que en los proyectos los desarrolladores creen sus propios tipos de excepciones. De esa forma, pueden ofrecer un tratamiento diferenciado a las situaciones que se presentan en cada proyecto concreto. En Java hay dos clases que representan las excepciones:



- **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 Checked Exception 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 cuando se trabaja con cuentas corrientes:



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

```

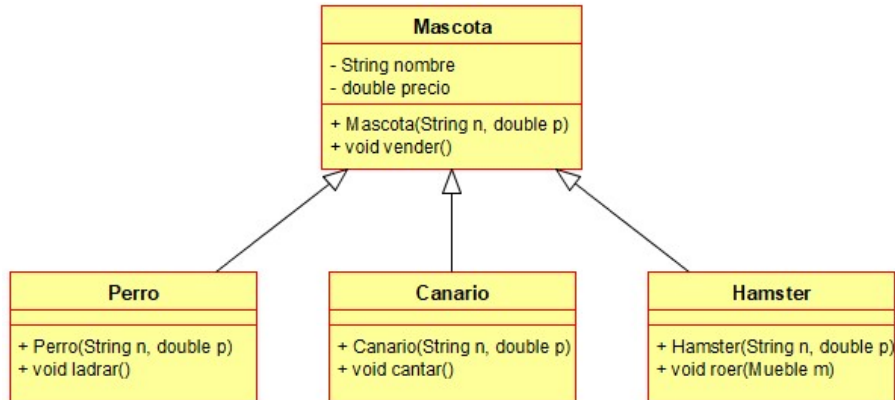
1. public class BancoException extends Exception{
2.     public BancoException(String mensaje){
3.         super(mensaje);
4.     }
5. }
  
```

### **Ejercicio 10 : Programa todas las clases del diagrama 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

### 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.

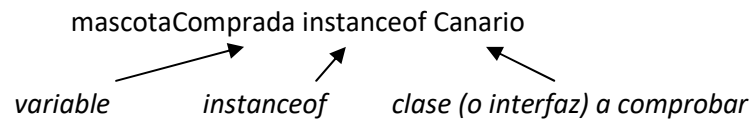
```
1. Mascota mascotaComprada = tienda.get(posicion);
2. mascotaComprada.vender();
```

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, y en caso afirmativo, hacerle un casting y entonces ya si se podrá llamar al método “cantar”.

Para detectar el tipo de dato “verdadero” que hay guardado en una variable se utiliza el operador **instanceof**, que funciona así:



Este operador 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 es solo para tipos referencia.

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

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

Dentro del if, podemos hacer un casting a la mascota y convertirlo en Canario. No va a producirse ningún fallo<sup>2</sup> porque instanceof nos ha asegurado de que la variable mascotaComprada guarda en realidad un objeto de la clase Canario. Una vez hecho el casting, ya si se puede llamar al método “cantar”.

```
1. if(mascotaComprada instanceof Canario){
2.     Canario c = (Canario) mascotaComprada;
3.     c.cantar();
4.     System.out.println("¿Te quedas con el canario?");
5.     String respuestaCliente = sc.nextLine();
6.     if(respuestaCliente.equals("si")) {
7.         c.vender();
8.     }
9. }else {
10.     mascota.vender();
11. }
```

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).
- 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 el tipo de dato original de los objetos usando el operador **instanceof** y luego un casting.
- El operador instanceof solo sirve para comprobar que una variable de tipo referencia tiene el tipo definido en una clase o una interfaz. No funciona para tipos básicos.

<sup>2</sup> En caso de hacer un casting a un objeto que no sea del tipo correcto se lanzará una **ClassCastException**

**Ejercicio 11 :** Consulta el diagrama **Equipo de fútbol** y programa las clases Empleado, CuerpoTecnico y CabreoException

- 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. El mensaje asociado es siempre el mismo: “Al empleado ... no se le han pagado ... euros”, donde se muestra el nombre del empleado y la cantidad de su salario que no se le ha pagado.

**Ejercicio 12 :** Siguiendo el diagrama de clases del ejercicio anterior, programa las clases EmpleadoPrimable, Futbolista y Entrenador

- 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.

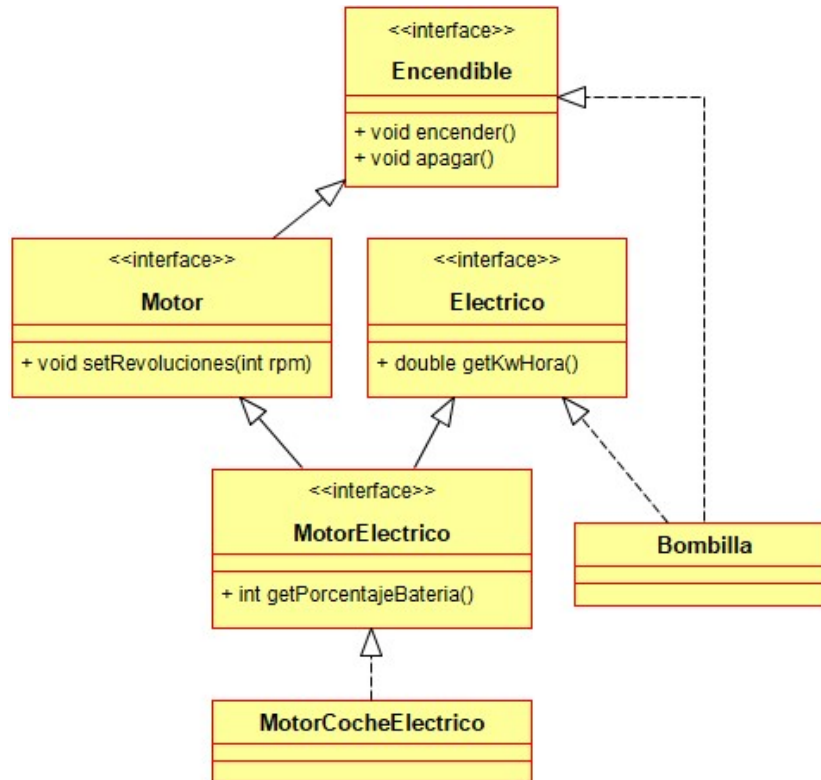
**Ejercicio 13 :** Siguiendo el diagrama de los ejercicios anteriores, haz la clase EquipoFutbol

- 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 disponible en el 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. De igual forma, 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.

### 3.6.- 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.



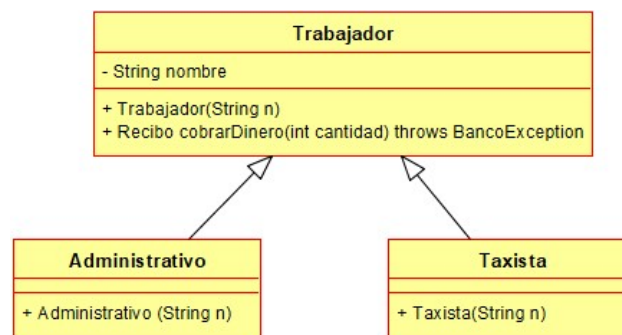
## 4.- Sobreescritura de métodos

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 hija<sup>3</sup>.
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).

<sup>3</sup> En esta situación, se dice que el método sobreescrito devuelve un **tipo covariante**

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. }
```

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.

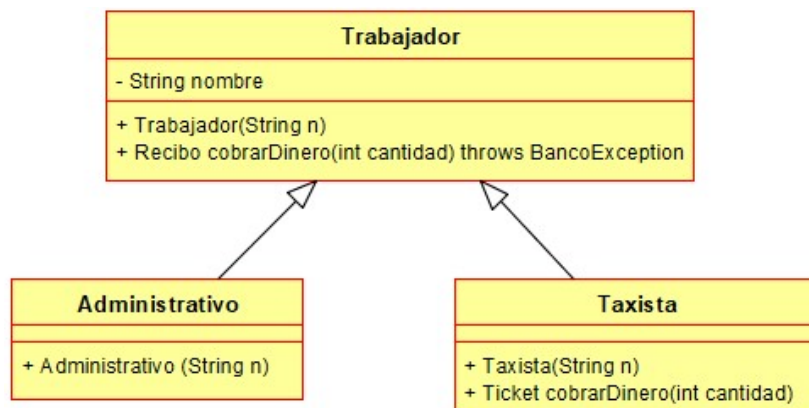
Por último, tampoco hay que respetar al 100% la lista de excepciones del método original. Se admite que el método sobreescrito lance menos excepciones que el de la clase padre, pero nunca se admite que lance una excepción nueva.

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 sobrescrito 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 sobrescribir 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:

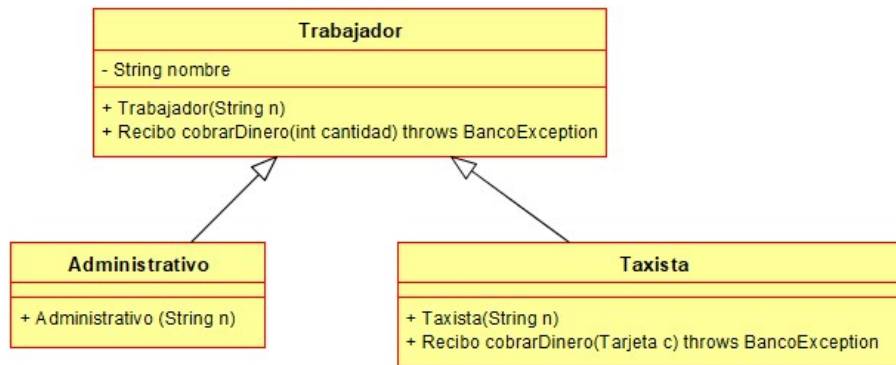
```
1. public class Taxista{
2.     public Taxista(String n){
3.         super(n);
4.     }
5.     public Ticket cobrarDinero(int cantidad) {
6.         // programación del método cobrando el dinero en mano
7.     }
8. }
```

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:

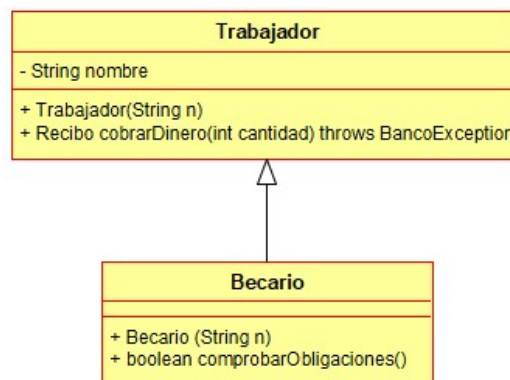


La clase **Taxista** ahora tiene dos versiones diferentes del método `cobrarDinero`. Una es la que hereda de la clase **Trabajador**, que ingresa el dinero en la cuenta corriente del taxista. Otra versión sería el método `cobrarDinero` que está programado en la clase **Taxista**, y que permite a un cliente 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 14 :** 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.

- 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.

**Ejercicio 15 :** Siguiendo el diagrama del ejercicio anterior, programa la clase `GeneradorContraseñasÚnicas`, que 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 generen 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.

**Ejercicio 16 :** Siguiendo el diagrama de los ejercicios anteriores, programa la clase `GeneradorContraseñasArchivo`, que 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.

**Ejercicio 17 :** Siguiendo el diagrama de los ejercicios anteriores, programa las clases `GeneradorContraseñasInvertidas` y `GeneradorContraseñasRaras`.

- `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.

**Ejercicio 18 :** ¿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. }
```

### **5.2.- Sobreescritura de equals**

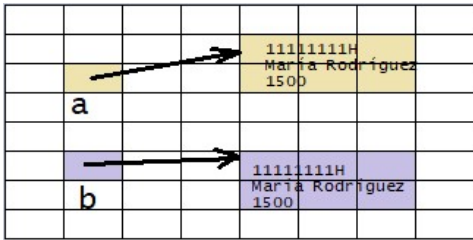
El método equals que todas las clases heredan de Object utiliza el signo == para comparar si dos objetos son iguales. Como ya sabemos, este operador compara el valor de los punteros en la memoria donde están los objetos, por lo que solo devolverá true cuando comparemos un objeto consigo mismo.

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){
13.            Empleado a = (Empleado)o;
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 con un casting 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.

Sobreescribir 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)



### 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 sobreescribimos el método equals, para que todo sea coherente será necesario sobreescribir 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 sobreescribirlo. Los IDE permiten sobreescribir equals y hashCode a la vez.

### 5.4.- Implementación de Comparable<T>

Aunque la interfaz Comparable<T> no tiene nada que ver con la clase Object, la programación avanzada de una clase puede necesitar que la implementemos.

Como se vio en el tema anterior, la interfaz Comparable<T> define un método **compareTo** que sirve para ordenar de menor a mayor los objetos de la clase que estamos programando.

El método compareTo se programa para que devuelva la “distancia” que existe entre el objeto que estamos programando y el que se pasa como parámetro, con el siguiente signo:

Si a es mayor que b	a.compareTo(b) > 0
Si a es menor que b	a.compareTo(b) < 0

Normalmente 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. }
```

**Ejercicio 19 :** Consulta el diagrama de clases **Paquetería** y programa la interfaz `EmpresaPaquetería` y las clases `Paquete` y `Transportista`.

- **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.

**Ejercicio 20 :** Siguiendo con el ejercicio anterior, programa la clase TransportistaOrdenado.

- 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.

**Ejercicio 21 :** Siguiendo con el ejercicio anterior, programa la clase PaquetesPepe y EmpresaPremium

- 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.

**Ejercicio 22 :** 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.

- 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.

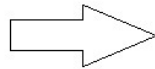
## 6.- Clases abstractas

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.



¿Comida?



Manzana



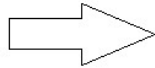
Bocadillo



Helado



¿Deporte?



Fútbol

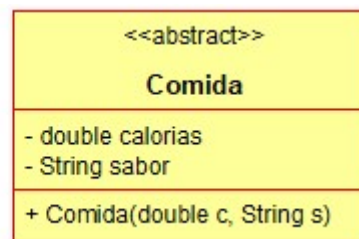
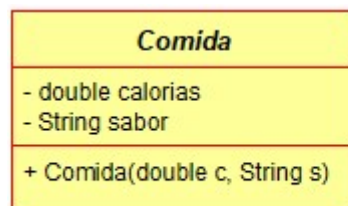


Baloncesto



Taekwondo

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:

```

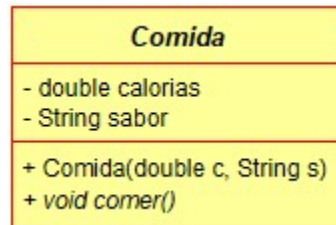
public static void main(String[] args) {
    Comida c = new Comida(100, "salado");
}
```

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.

## 6.1.- Métodos abstractos

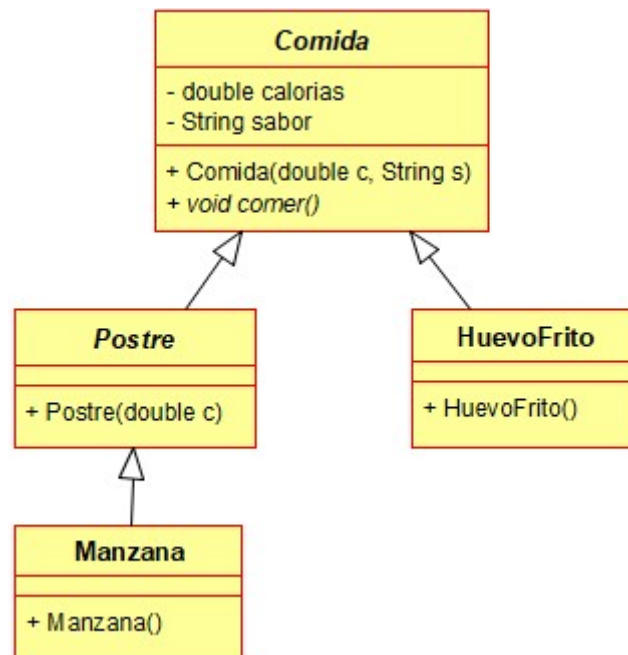
Las clases abstractas también pueden tener **métodos abstractos**. Un método abstracto es un método de una clase abstracta que está sin programar, para que sean las clases hijas quienes lo hagan. Los métodos abstractos se escriben en cursiva en los diagramas de clases:



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 interfaces<sup>4</sup>.

```
1. public abstract class Comida{
2.     private double calorias;
3.     private String sabor;
4.
5.     public Comida(double c, String s){
6.         calorias=c;
7.         sabor=s;
8.     }
9.
10.    public abstract void comer();
11. }
```

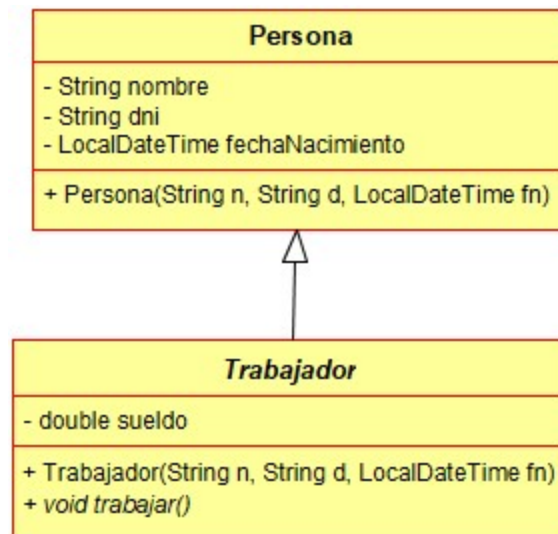
Son las clases hijas no abstractas quienes están obligadas a sobrescribir dicho método y proporcionar una implementación. Por ejemplo, observemos este diagrama:



<sup>4</sup> 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 23:** Consulta el diagrama de clases **Animales** y programa en un paquete llamado **daw.zoo** las clases Animal, Tigre y León

- 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 sobrescriben “emitirSonido”.

**Ejercicio 24:** Programa en el mismo paquete **daw.zoo** la clase **ContenedorAnimales**, que 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.

**Ejercicio 25 :** Programa en el mismo paquete **daw.zoo** las clases **Jaula** y **TransporteAnimales**

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

## 6.2.- ¿Clases abstractas o interfaces?

En muchas ocasiones, al diseñar un diagrama de clases surge la duda de si considerar un concepto como una clase abstracta o una interfaz, ya que las dos cosas podrían servir.

Por ejemplo, imaginemos que tenemos las clases **Persona**, **Perro** y **Gato**, que son animales terrestres. ¿Qué sería mejor, considerar una clase abstracta “**AnimalTerrestre**” de la que heredasen estas tres clases, o considerar una interfaz “**Terrestre**” que la implementasen las tres clases?



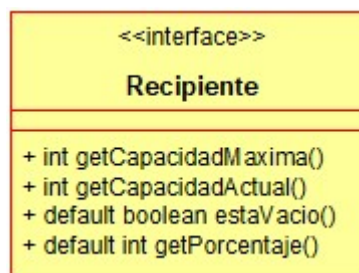
Aparentemente, el resultado es el mismo ya que las clases Persona, Perro y Gato deben programar los métodos andar, correr, comer y saltar.

La mejor solución dependerá de cada situación concreta, y habrá que tener en cuenta cosas como estas:

- Una clase solo puede heredar de otra. Si hacemos herencia, estamos “gastando” la única posibilidad que tenemos de heredar, que a lo mejor podría hacernos falta más adelante. Desde ese punto de vista, siempre que sea posible, debemos usar utilizar interfaces, porque además, con ellas podemos hacer herencia múltiple.
- Cuando heredamos, recibimos propiedades y métodos de la clase padre, cosa que con la interfaz no es posible. Si el “concepto padre” tiene muchas propiedades y métodos comunes a las clases hijas que se podrían programar en la clase padre, es preferible heredar, aunque ya estaríamos renunciando a la posibilidad de heredar de otra clase que pueda salirnos más adelante.

La tendencia actual en programación es utilizar siempre **interfaces**, en detrimento de las clases abstractas. Se ha comprobado que el código es más fácil de reutilizar cuando se usan interfaces (con clases se forma un árbol que es necesario trasladar completamente de un proyecto a otro). Además, para promover el uso de las interfaces, la versión 8 del lenguaje Java permite incluir en ellas **métodos estáticos** (tienen el mismo significado que los métodos estáticos de las clases) y **métodos default**.

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



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



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

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

**Ejercicio 26:** Consulta el diagrama de clases **Figuras geométricas** y programa las interfaces `Coloreable`, `Centrable` y `FiguraGeométrica`.

- `Coloreable`: Es cualquier cosa que tiene un color.
- `Centrable`: Es un objeto que tiene un centro.
  - `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.

**Ejercicio 27:** Programa la interfaz `Apoyable` y la clase `Círculo`

- `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.
- `Círculo`: Clase que representa un círculo e implementa todas las interfaces asociadas.

**Ejercicio 28:** Programa las clases `Rectángulo` y `Cuadrado`

- `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.

**Ejercicio 29:** Programa la clase `Triángulo`, que 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.

## 7.- Polimorfismo

El polimorfismo es, posiblemente, la técnica más importante de toda la programación orientada a objetos. 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 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.

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

En este momento, es difícil comprender lo que significa esta definición. Por ese motivo, vamos a ver ejemplos concretos de cómo conseguir el polimorfismo en un programa, para al final, comprender el concepto.

### **7.1.- Polimorfismo mediante clases abstractas**

Supongamos que estamos programando un videojuego de naves espaciales<sup>5</sup>, de forma que el jugador puede elegir su nave entre tres opciones:



Nave A








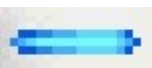



Nave B



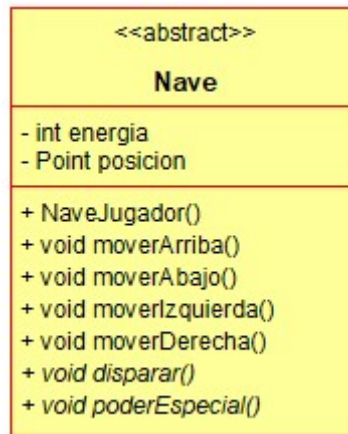
Nave C

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

Nave	Disparo normal	Poder especial
		
		
		

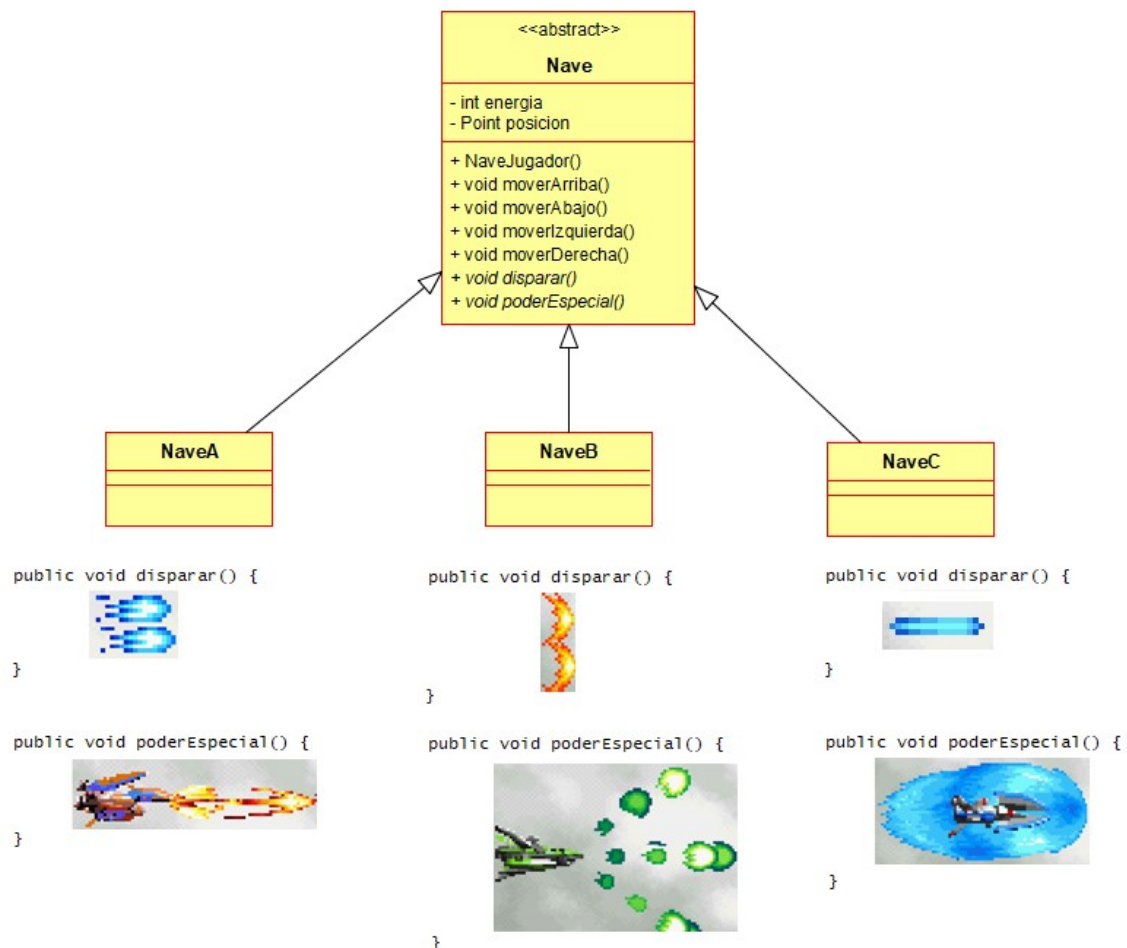
A la hora de programar este juego, haríamos una clase abstracta llamada **Nave** que tendría este diagrama:

<sup>5</sup> Imágenes tomadas del **Blazing Star**, de Neo Geo



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.

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 principal<sup>6</sup>.

<sup>6</sup> 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

- 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.

## **7.2.- 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.

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.

---

programa automáticamente las detecta y permite seleccionarlas. Es lo que se hace para programar **plugins**.

**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 :** Crea un paquete llamado **efectos.clases** y programa en él el diagrama 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.
  - La propiedad “nombre” es 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 métodos<sup>7</sup> que indica que un efecto especial es simétrico, es decir, si se aplica dos veces se vuelve a obtener el mismo texto de partida.
- 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

<sup>7</sup> Las interfaces sin métodos se llaman **marker interfaces**, y sirven para destacar que una clase posee alguna característica que la hace especial por algo.

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

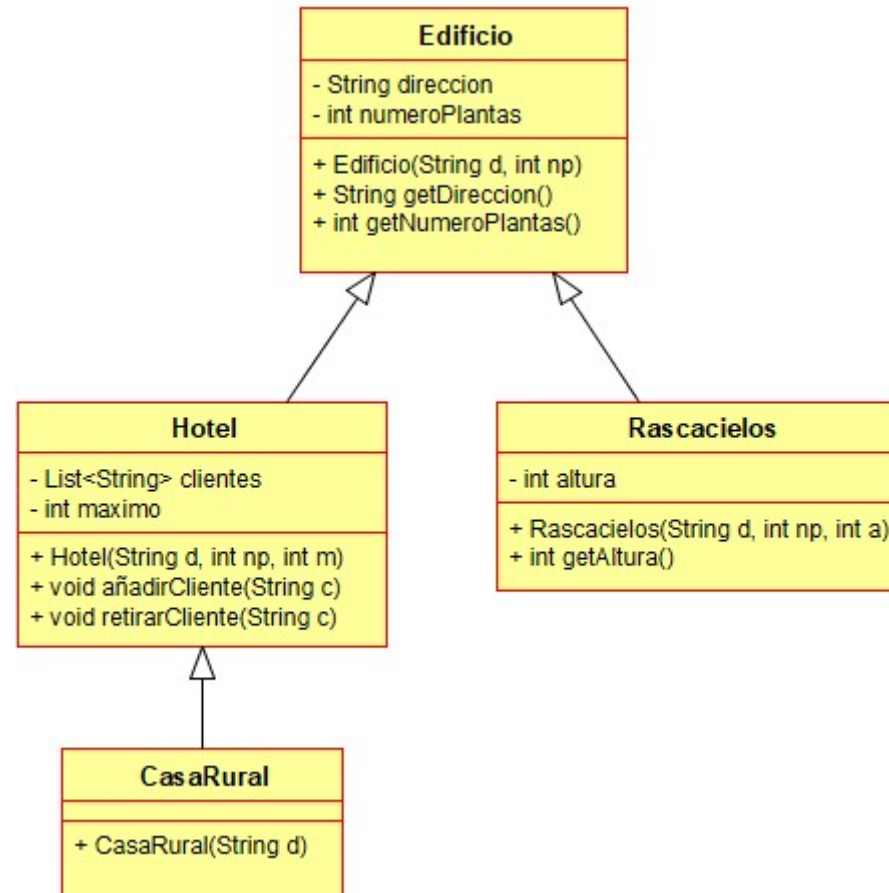
- 1) Aplicar todos los efectos a la frase
- 2) Aplicar solo los efectos simétricos

Según la opción elegida, el programa recorrerá todos los efectos especiales, o solo los simétricos, y los aplicará a la frase que ha introducido el usuario.

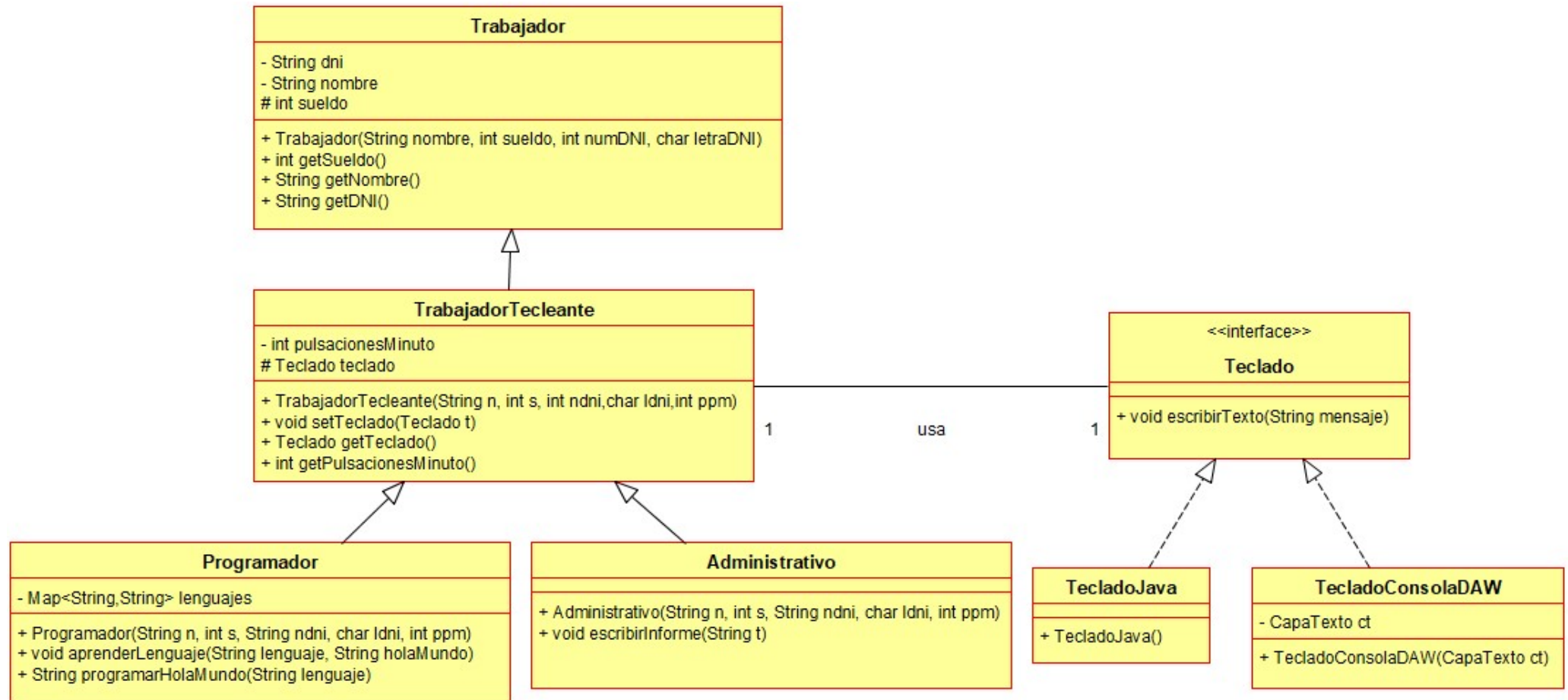
¿Encuentras algún motivo para que las clases de los efectos especiales tengan en su constructor el modificador por defecto?



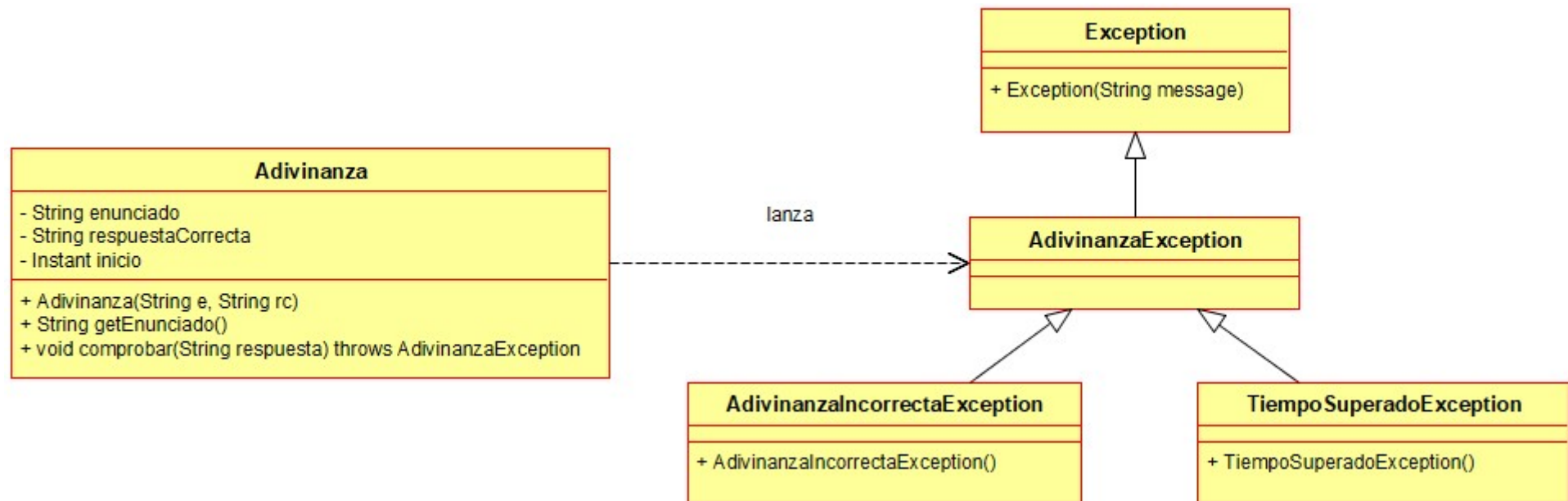
## DIAGRAMAS DE CLASES: EDIFICIOS



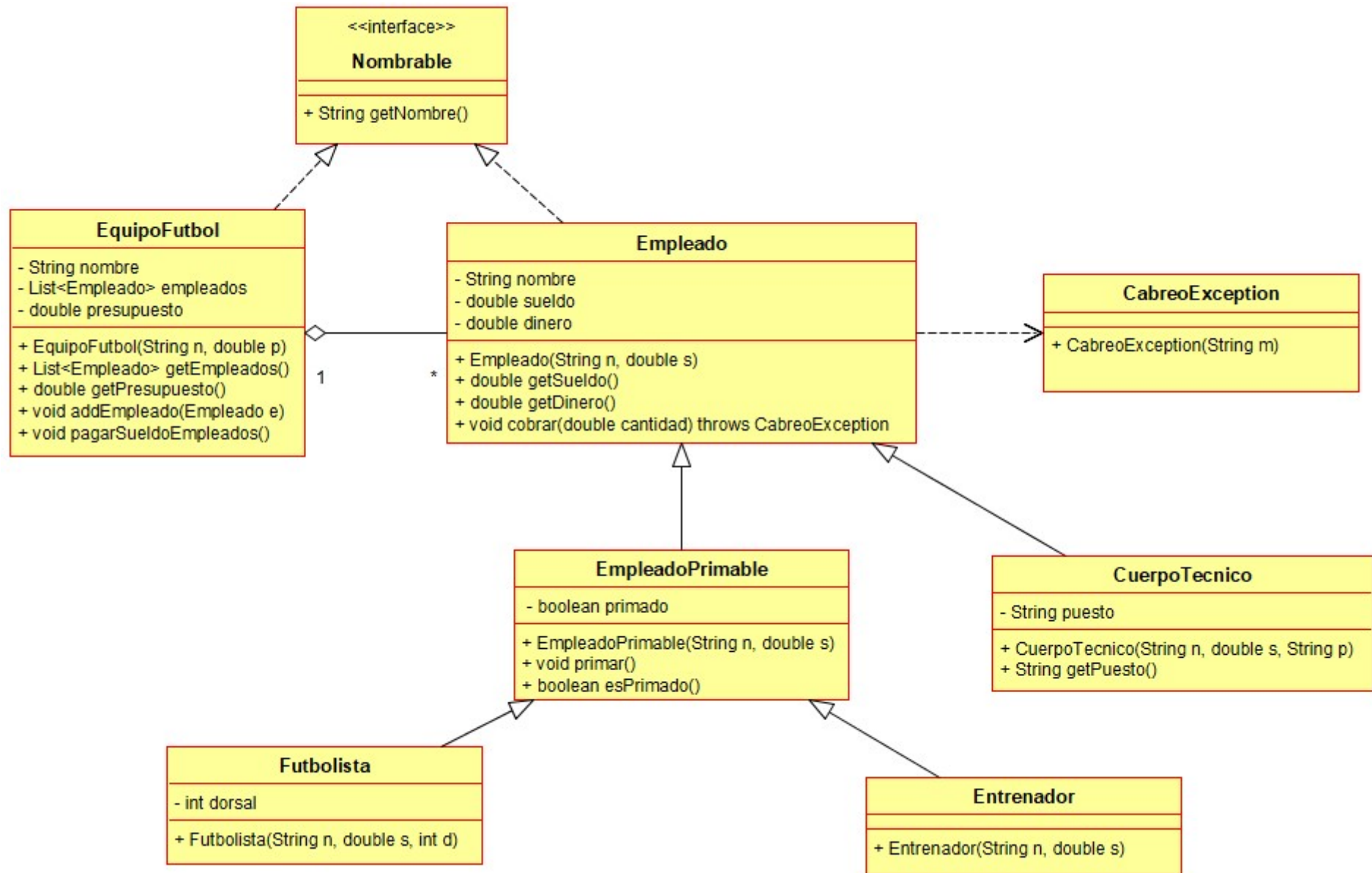
## DIAGRAMAS DE CLASES: TRABAJADORES



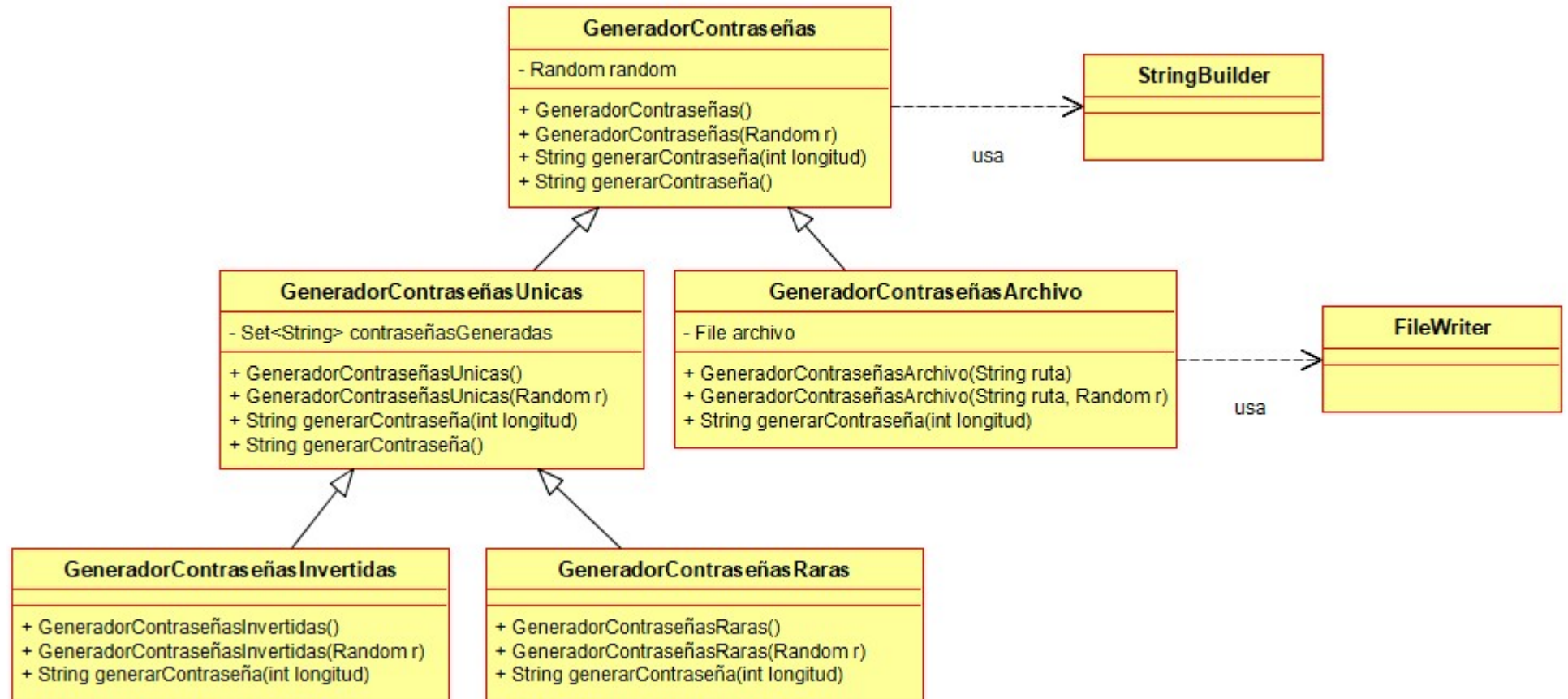
## DIAGRAMAS DE CLASES: ADIVINANZAS



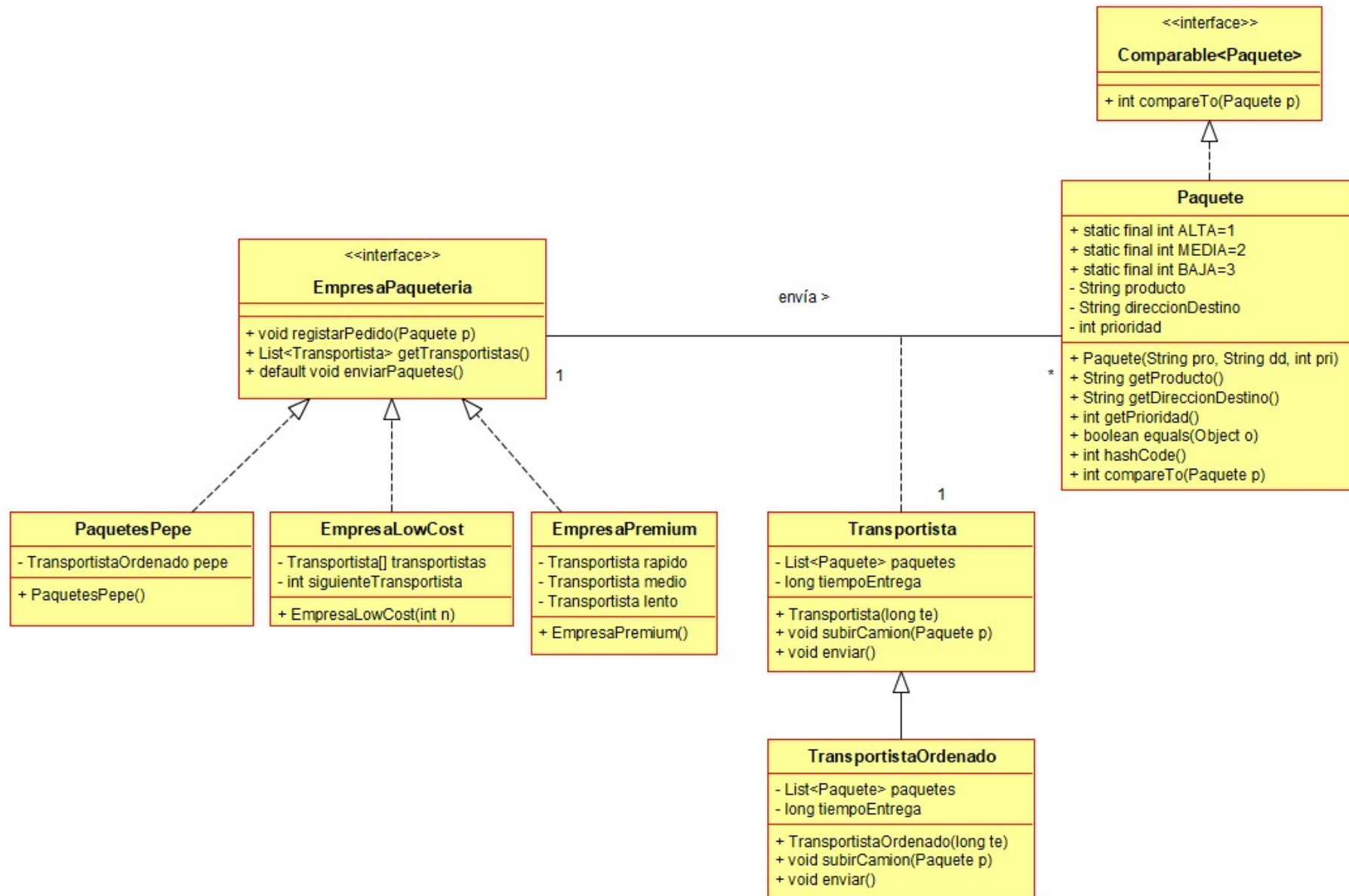
## DIAGRAMAS DE CLASES: EQUIPO DE FÚTBOL



## DIAGRAMA DE CLASES: CONTRASEÑAS

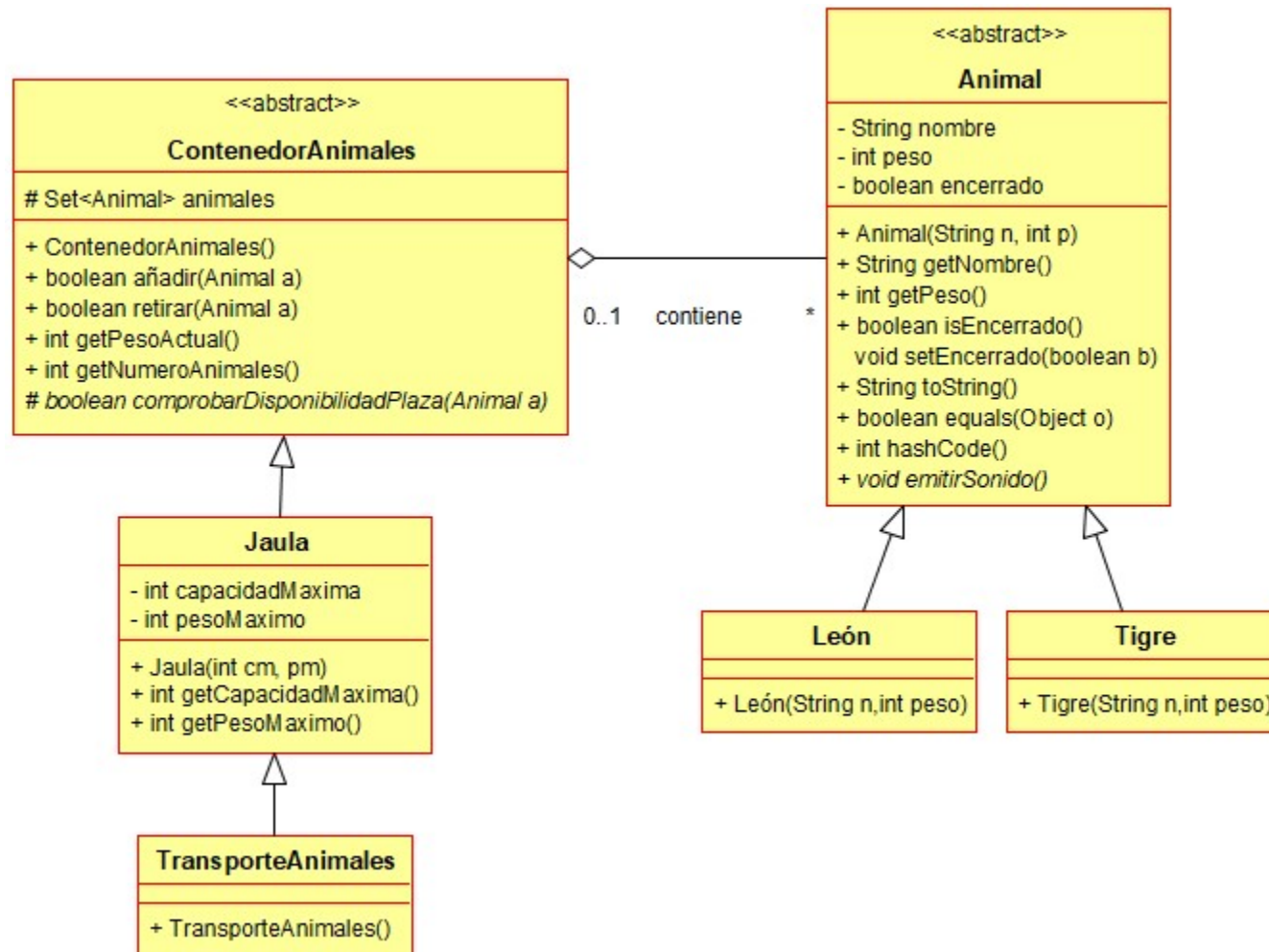


## DIAGRAMA DE CLASES: PAQUETERÍA

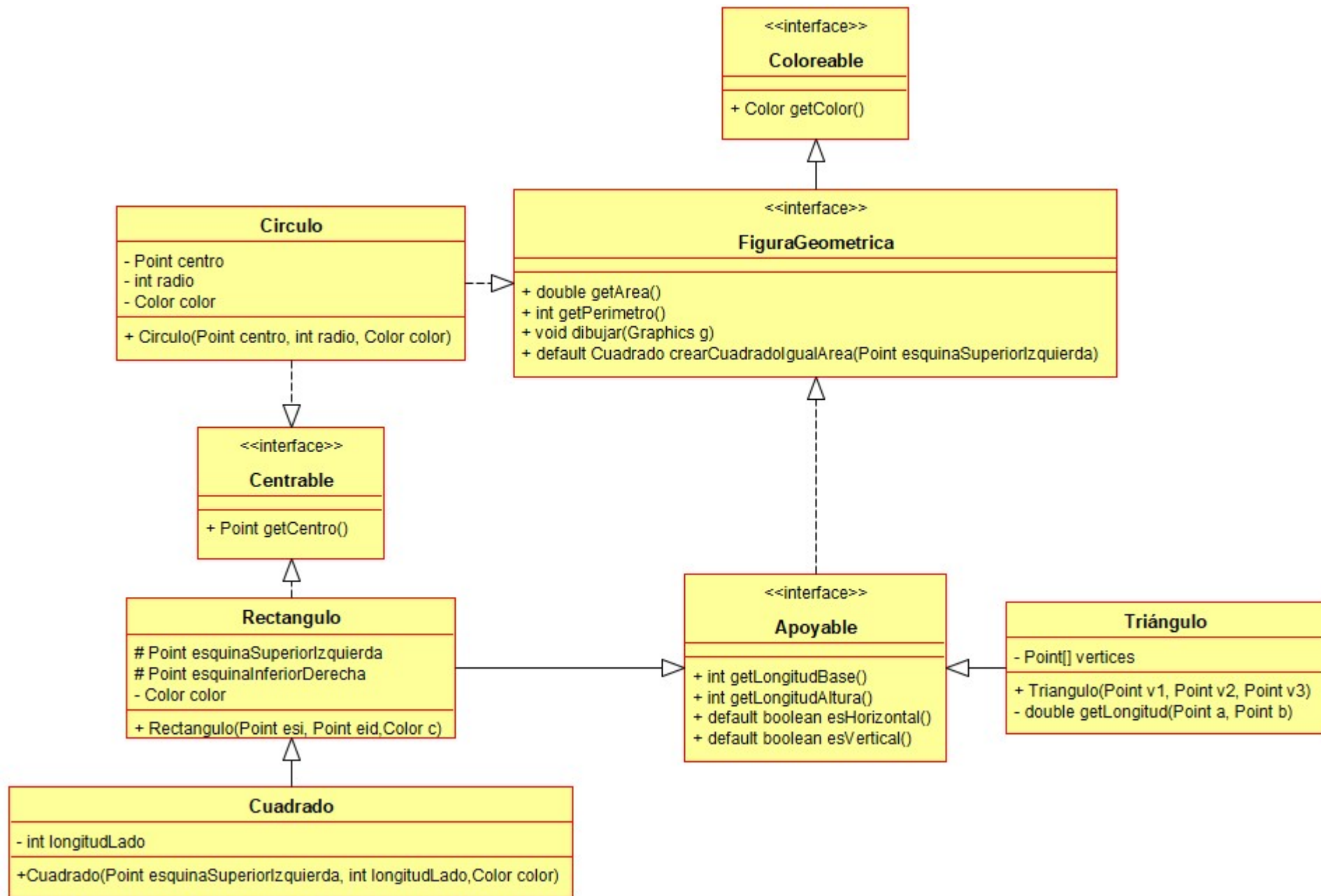




## DIAGRAMAS DE CLASES: ANIMALES



## DIAGRAMAS DE CLASES: FIGURAS GEOMÉTRICAS





## DIAGRAMAS DE CLASES: EFECTOS ESPECIALES

**Nota:** Este diagrama está expresado en notación UML

