

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Tema 6 – Otros conceptos de Java

En este tema veremos algunos conceptos más de Java que han quedado pendientes. Concretamente veremos como gestionar las excepciones en nuestros programas para evitar que finalicen. Posteriormente profundizaremos más en algunos conceptos de POO que no se introdujeron en el primer tema por evitar hacerlo demasiado abstracto y tedioso.

Control de excepciones

Una excepción es un mecanismo utilizado para realizar un control claro (en cuanto a código y estructuración) de los posibles errores que puedan producirse en tiempo de ejecución (Divisiones por cero, archivos que no se pueden abrir, error en el casting, ...).

En principio, una excepción no controlada provoca una completa terminación del programa en ejecución. Pero el propio programador puede crear código para controlar dichas excepciones.

Una excepción se puede ver como un evento (un mensaje interno) que rompe el flujo normal del programa. Este evento hace que se cree además un objeto del tipo *Exception* donde se guarda información sobre el error en cuestión. Si dicho objeto lo “recoge” el programa se pueden tomar decisiones en cuanto a lo que va a realizar el programa en lugar de simplemente parar como nos sucede hasta ahora.

Vamos a ver en este apartado cómo controlar esos objetos *Exception* y también veremos cómo lanzarlas con el objetivo de tener el control de errores bien estructurado.

Cláusula try/catch

En general para poder tratar una excepción se usa una estructura *try..catch* de la siguiente forma:

```
try{
    //Código con posibilidad de que produzca un error
}catch (ClaseExcepcion1 objExcepcion){
    //Código que se ejecuta cuando se produce la excepción 1
}catch (ClaseExcepcion2 objExcepcion){
    //Código que se ejecuta cuando se produce la excepción 2
}
...
finally{
    //Parte opcional con código que se ejecuta se produzca o no el error
}
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Se puede usar más de un *catch* en el caso que queramos tener en cuenta varias excepciones en el mismo código.

Veamos como manejarlo de forma práctica. Escribe el siguiente código directamente en la función *main()*:

```
Scanner sc=new Scanner(System.in);
int dividendo, divisor;

System.out.println("Por favor, introduce dividendo");
dividendo=Integer.parseInt(sc.nextLine());

System.out.println("ahora el divisor");
divisor=Integer.parseInt(sc.nextLine());

System.out.printf("El resultado es %d\n", dividendo/divisor);
```

Si ejecutas el programa y se escribe algo que no sea un número entero, por ejemplo una letra "a", se produce un error de conversión de tipo. El mensaje será similar al siguiente:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "a" at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:481)
    at java.lang.Integer.parseInt(Integer.java:514)
    at Excepciones.main(Excepciones.java:18)
```

Esto haría finalizar el programa de forma imprevista (y poco usable), lo que no nos interesa. Este mensaje de error da mucha información y hay que aprender a entenderla porque ayuda. Veámoslo:

- En la primera línea dice la excepción que salta y el motivo (for input string "a", es decir, se metió la cadena "a" y no es válida). Es quizá la información más importante.
- En la segunda nos da la línea de código de error pero en código que no es nuestro. Pulsa en el enlace y compruébalo, es código de Oracle.
- Lo mismo ocurre con las tercera y cuarta líneas, que corresponden a la clase Integer y en concreto dos sobrecargas del método estático parseInt.
- Finalmente en la última línea me dice el número de línea donde se produjo el error en nuestro código. Esta también es básica entenderla y es la que hay que buscar cuando salen un montón de líneas de mensaje de error. Lo que indica es el archivo .java donde se produce y la línea.

Por supuesto los números de línea pueden salirte distintos.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Vamos a modificar el código con la estructura anterior para evitar esto:

```
try {
    System.out.println("Por favor, introduce dividendo");
    dividendo = Integer.parseInt(sc.nextLine());
    System.out.println("ahora el divisor");
    divisor = Integer.parseInt(sc.nextLine());
    System.out.printf("El resultado es %d\n", dividendo / divisor);
} catch (NumberFormatException e) {
    System.out.println("Ha introducido un dato que no es un número entero");
}
```

El funcionamiento es el siguiente: Si se produce una excepción del tipo **NumberFormatException** el programa en lugar de parar y dar el mensaje de error mostrado anteriormente, salta a la cláusula **catch** y realiza el código ahí indicado. Tras ello el programa continua normalmente.

Pero ahora que está corregido, prueba a introducir como divisor un 0 ¿Qué es lo que sucede? Una excepción aritmética hace su aparición.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at excepciones.Excepciones.main(Excepciones.java:28)
```

La solución a esto es muy sencilla, se trata de añadir otro *catch* con una nueva excepción para controlar el error.

```
catch (ArithmeticException e){
    System.out.println("División por cero");
}
```

El objeto *e* que aparece contiene información sobre la excepción. Prueba a poner un punto tras la *e* para ver los distintos campos. No usaremos demasiado esto, pero puede valer para dar un error genérico añadiendo el siguiente *catch* o incluso poniendo uno único para todo:

```
catch (Exception e){
    System.out.println("Se ha producido un error del tipo: "+e.toString());
}
```

Cambia uno de los *parseInt* por *nextInt* y prueba a meter algo que no es un número para que salte esta excepción.

Al tener varios *catch* el try/catch funciona de forma similar a un switch, si hay error se salta al catch correspondiente a la excepción generada.

Si quieres probar el catch del exception, borra uno de los catch anteriores para que salte este último.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

La cláusula **finally** no la usaremos mucho por el momento, pero se debe saber que es una parte que se ejecuta SIEMPRE después de los `catch` o aunque no se produzca ningún `catch`. Incluso en el caso en el que haya un *return* dentro de un *catch*, se ejecutaría el *finally* antes de hacer el *return*. Cambia la última excepción como viene a continuación y añade el *finally*:

```
catch (Exception e) {
    System.out.println("Se ha producido un error del tipo: " + e.toString());
    return;
} finally {
    System.out.println("Siempre pasa por aquí");
}
System.out.println("Por aquí solo si no hay error");
```

Nota ED: Existe la plantilla (snippet) **try_catch** en vscode para crear automáticamente esta estructura.

Las excepciones se pueden anidar para dar un uso más particular como se puede ver en este ejemplo:


```
try {
    System.out.println("Por favor, introduce dividendo");
    dividendo = Integer.parseInt(sc.nextLine());
    System.out.println("ahora el divisor");

    //Si hay división por 0 no queremos que pida el dividendo
    boolean error;
    do {
        error = false;
        try {
            divisor = sc.nextInt();
            System.out.printf("El resultado es %d\n", dividendo/ divisor);
        } catch (ArithmeticException e) {
            System.out.println("División por cero. " +
                "\nPor favor introduzca de nuevo el divisor");
            error = true;
        }
    } while (error);
} catch (NumberFormatException e) {
    System.out.println("Ha introducido un dato que no es un número entero");
}
```

Si necesitas una lista de las excepciones puedes buscar en internet. Una página ejemplo sería directamente la lista de subclases de `Exception` que se puede ver aquí:

<http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html>

De cualquier manera la mejor forma de ir tratando las excepciones es añadirlas a los

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

catch a medida que las vamos detectando (o si ya sabemos que pueden aparecer) o ver la documentación de las funciones ya que estas indican la excepción que se puede lanzar.

Más adelante veremos que muchas excepciones están relacionadas entre sí mediante lo que se denomina herencia en POO.

Finalmente indicar que si la operación es la misma para distintos catch, se puede usar un OR a nivel de bit entre las excepciones. Para probarlo amplía el catch de *NumberFormatException* de la siguiente forma:

```
catch (NumberFormatException | InputMismatchException e) {
    System.out.println("Ha introducido un dato que no es un número entero");
}
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Lanzamiento de excepciones

Es posible que nuestro programa (o una función en una clase determinada) lance una excepción en caso de que se produzca una situación que a nosotros no nos interese. Esto puede tener distintas utilidades. Por ejemplo una función que no queremos hacer interfaz de usuario pero si se produce un error lanzamos una excepción para que sea recogido en otra función donde sí haya IU (piensa por ejemplo en un set de un objeto con un valor no válido). O como en el ejemplo siguiente que nos permite tener todos los mensajes de error ordenados en varios *catch*.

Veamos el siguiente ejemplo:

```
try{
    System.out.println("Introduce un nº positivo");
    numero=sc.nextInt();
    if (numero<0){
        throw new Exception("No se admiten valor negativo");
    }
} catch (InputMismatchException e){
    System.out.println("Ha introducido un dato que no es un número entero");
} catch (Exception e){
    System.out.println(e.getMessage());
}
```

En el caso de hacer una función que lance una excepción pero se desea que dicha excepción no sea controlada dentro de la propia función hay que añadir a la cabecera de la función la terminación *throws Exception* tal y como se ve en el ejemplo:

```
public static int numeroPositivo() throws Exception {
    Scanner sc = new Scanner(System.in);
    int numero=0;

    System.out.println("Introduce un nº positivo");
    numero=sc.nextInt();
    if (numero<0){
        throw new Exception("No se admiten valor negativo");
    }
    return numero;
}
```

por supuesto el tipo de excepción debe coincidir con el que se lanza.

Otro ejemplo:

```
public void setMes(int mes){
    if (mes<1||mes>12){
        throw new IllegalArgumentException("Mes inválido");
    }
    this.mes=mes;
}
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

En documentación *javadoc* puedes usar el tag **@exception** o **@throws** para explicar el motivo del lanzamiento de una excepción.

Hay un grupo de Excepciones denominadas **Unchecked Exceptions** que no necesitan *throws* como pueden ser *NullPointerException*, *IllegalArgumentException* o *ArrayIndexOutOfBoundsException*.

Puedes ver la lista de ambos grupos (Checked y Unchecked) aquí:

<https://stackoverflow.com/a/39607374>

En principio salvo casos muy particulares y justificados, no se deben capturar excepciones de objeto nulo o de salida de índices en arrays, colecciones,... pues pueden ser comprobadas de forma más concreta con *if*. Si no se corre el riesgo de estar capturando algo que no corresponde.

No vamos a profundizar más hasta el final del tema, pero si se desea aquí hay un buen artículo con múltiples ejemplos en el siguiente enlace, aunque también en este caso se recomienda su lectura una vez finalizado el tema pues usa conceptos de POO que aún no vimos:

<http://www.javaworld.com/article/2076700/core-java/exceptions-in-java.html>

Actividades para practicar:

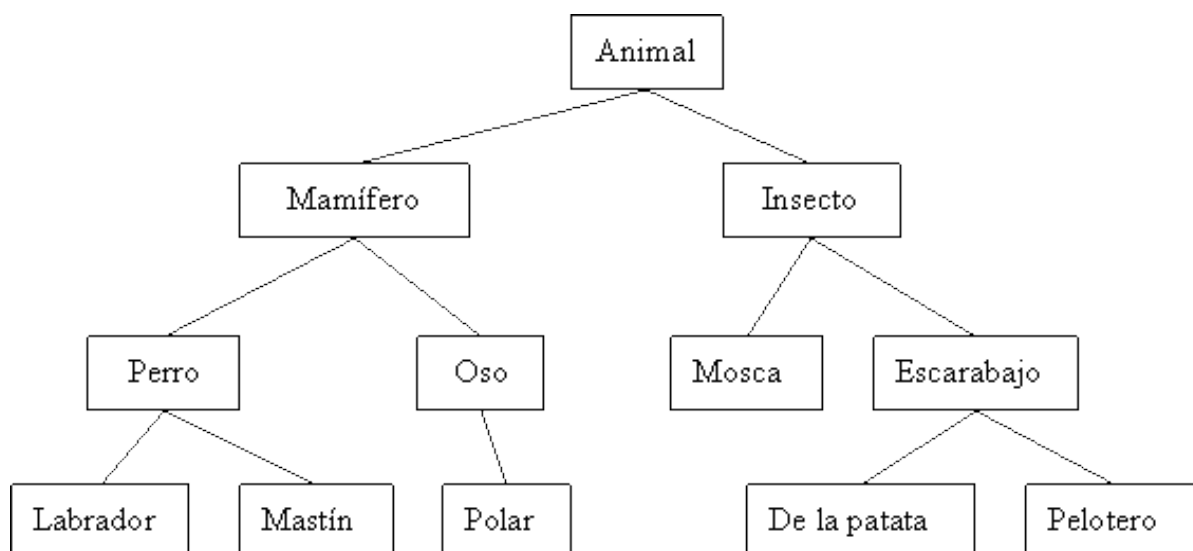
1. Coge el ejercicio 5 del primer boletín (Conversor temperaturas) y añádele control de excepciones de forma que si el usuario mete algo que no sea un número indique el error y vuelva a salir el menú.
2. En el ejercicio de la clase fecha haz que cuando se meta un día o un mes inválido en el set correspondiente se lance una excepción (*IllegalArgumentException*) en vez de poner el día o mes a 1. Haz además control de excepciones **en el programa principal** según sea necesario.
3. Realiza una función genérica de petición de dato entero. Debe pedir un dato al usuario y si no introduce un nº entero volver a pedir el dato. La función devuelve el dato cuando sea correcto.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM	
	MÓDULO	Programación			CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:	DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)			

Otros aspectos de POO

Herencia

Este concepto de la OO está relacionado con la jerarquía de definiciones existente en el mundo real. Un ejemplo se ve en el siguiente esquema:



- Animal: superclase (definición más genérica): Es un ser vivo eucariota pluricelular.
- Mamífero: subclase: Es un **Animal** (hereda de esta forma todas las propiedades de animal) con pelo, se desarrolla en el interior de la madre y se alimenta de leche materna la primera etapa de vida)

Actividad: Busca la definición de perro en la Wikipedia y sube en la jerarquía de herencia hasta animal para comprobar como funciona en el mundo real.

En programación, la subclase hereda todos los miembros de la superclase como si fueran suyos. Esto asegura que la complejidad crezca en menor medida que con técnicas habituales.

En el primer boletín de OO teníamos por ejemplo las clases Empleado y Directivo que ambas disponían de nombre y apellidos. Lo lógico sería definir estas en una clase denominada, por ejemplo, Persona y que tanto Empleado como Directivo heredaran de Persona para no tener que repetir código.

Una clase en Java, aunque no se indique, hereda de la clase genérica *Object*. Se puede comprobar esto al acceder a los miembros de la clase *Perro*. Aparecen miembros que no han sido definidos, que han sido heredados.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

En Java se establecerá la herencia por medio de la cláusula *extends*. Veamos el uso con un ejemplo guiado:

1. Escribe las siguientes clases

```
public class Animal {
    public String nombreCientifico;

    private int edad;

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}

public class Mosca extends Animal {
    public Mosca () {
        this.nombreCientifico="Drosophila Melanogaster";
    }
}

class Perro extends Animal {
    public String raza;
    public String nombreHumano;

    public Perro() {
        this.nombreCientifico="Canis Familiaris";
    }

    public Perro(int edad, String raza, String nombreHumano) {
        this.setEdad(edad);
        this.raza=raza;
        this.nombreHumano=nombreHumano;
        this.nombreCientifico = "Canis Familiaris";
    }

    public void ladrar() {
        System.out.println("GUAU!!!");
    }
}
```

2. Crea los objetos **objMosca** y **objPerro** en el **main**. Fíjate que en cuanto pones el punto aparecen tanto las propiedades propias como las heredadas (salvo que sean privadas).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

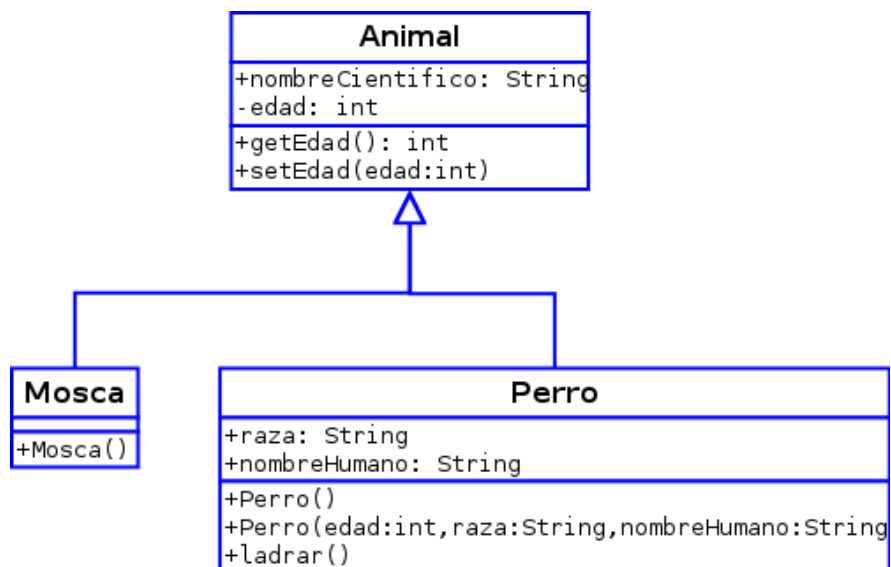
3. Prueba a poner en **ladrar()**

```
System.out.printf("GUAU!!! tengo %d años\n", getEdad());
```

4. Y luego **edad** en lugar de **getEdad()**;

5. Por último en **edad** sustituye **private** por la palabra **protected** y ejecuta de nuevo. Entenderemos esto en el apartado siguiente.

En UML la herencia se identifica mediante una flecha cerrada:



Para indicar que un miembro es **protected** se usa **#**

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Encapsulación y control de acceso

Lo visto en el último punto del ejemplo anterior se debe a los modificadores de control de acceso dentro de las clases heredadas. Estos modificadores permiten que ciertos miembros estén o no visibles a otras clases. Son los siguientes:

public: Puede ser accedido desde cualquier código: clase, subclases y otras clases. Se usa + en UML.

private : Sólo puede ser accedido desde el código de la clase a la que pertenece. Ni siquiera permite el acceso desde subclases. Se usa – en UML.

sin modificador (package-private): Cuando no se especifica nada en un miembro toma este nivel por defecto que es como público pero sólo dentro del package en el que se encuentra. Fuera del package no se puede acceder (se ve como privado). En ocasiones se denomina **default**. *No pondremos nada en UML.*

protected : Permite el acceso además de desde dentro del *package* en subclases heredadas fuera del package. Pondremos # en UML.

Ejemplo: Tenemos 2 paquetes. En el primero las clases A y B y en el segundo subA y C. La clase subA es subclase de la clase A (subA hereda de A). En este caso se cumple la siguiente tabla de visibilidad para los **miembros de la clase A** (Es decir ¿Se puede ver un miembro de A desde las otras clases?).

Visibilidad	A (pkg1)	B (pkg1)	SubA (pkg2)	C (pkg2)
private	Sí	No	No	No
protected	Sí	Sí	Sí	No
public	Sí	Sí	Sí	Sí
Sin modificador	Sí	Sí	No	No

Se debe tener en cuenta que a las clases también se les puede aplicar modificadores, pero sólo dos: **public** para que se accese desde cualquier otro punto o **sin modificador** lo que la hace accesible sólo dentro de las clases del mismo **package**.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Llamadas entre constructores

En ocasiones se hace necesario o cómodo ejecutar un constructor desde otro. Para ello, si el constructor es una sobrecarga de otro en la misma clase se usará la palabra reservada **this** para dicha llamada. Si el constructor al que se desea llamar está en la clase padre de la actual se usará la palabra reservada **super**.

Veámoslo con un ejemplo. Añade estos dos constructores a la clase Animal:

```
public Animal(){
}

public Animal (int edad, String nombreCientifico){
    this.setEdad(edad);
    this.nombreCientifico = nombreCientifico;
}
```

En Perro, borra los constructores de apartados anteriores y vete añadiendo los siguientes. El primero llama al constructor de la clase padre para inicializar la edad y el nombre científico y el resto lo inicializa en el propio constructor:

```
public Perro(int edad, String raza, String nombreHumano) {
    super(edad, "Canis Familiaris");
    this.raza=raza;
    this.nombreHumano=nombreHumano;
}
```

Ahora añadimos un constructor que inicialice la raza y el nombre llamando a la sobrecarga anterior:

```
public Perro(String raza, String nombreHumano){
    this(0,raza,nombreHumano);
}
```

Por último añadimos o sustituimos dos más, el primero inicializa edad a partir de la sobrecarga sita en la clase padre y el segundo sin parámetros inicializa a partir del primer constructor con valores "nulos":

```
public Perro() {
    this(0, "", "");
}

public Perro (int edad){
    super(edad, "");
}
```

¿Cómo harías para que el constructor Animal() inicialice a 0 años y a nombre científico cadena vacía llamando al otro?

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Sobreescritura

La sobreescritura de un método es la redefinición en una subclase de una función que existe en una clase padre. Para ello la función redefinida tiene que tener la misma signatura (nombre y lista de parámetros) que la de la clase base.

También debe cumplir que el método redefinido tiene que tener un nivel de acceso mayor que el de la clase padre. Es decir, no puedo definir como público una función que en la clase base sea privada o *protected*.

Escribe el siguiente método en la clase Animal:

```
public void muestraDatos(){
    System.out.printf("Tengo %d años", this.edad);
}
```

Ahora en la clase Mosca:

```
@Override
public void muestraDatos(){
    System.out.printf("Las moscas no tienen edad\n");
}
```

Esto es la función sobrecargada para que realice una tarea distinta. Si queremos realizar lo que hace en la clase base y añadirle algo más podemos ejecutar el método sobreescrito usando la palabra reservada **super**.

```
@Override
public void muestraDatos(){
    super.muestraDatos();
    System.out.printf("Sin embargo las moscas no tienen edad\n");
}
```

Se recomienda el uso (de hecho será imprescindible en este curso) de la anotación **@Override** para indicar al compilador que se va a realizar una sobreescritura. Esto permite la comprobación de errores verificando si la función que hay a continuación tiene un equivalente en la clase padre e informando si no es así.

Pruébalo escribiendo **muestraDatos (sin la r)** en lugar muestraDatos. Al existir la anotación @Override da error. Si la quitas no te informa de dicho error y puede pasar desapercibido.

Podemos **sobreescribir** también funciones que no hayamos escrito nosotros. Por ejemplo podemos hacerlo con **toString()** en cualquier otra clase. Veámoslo: Prueba en el programa principal lo que devuelve la siguiente línea:

```
Perro p=new Perro(5,"Pastor","Laika");
System.out.println(p.toString());
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Verás que sale una información que no vale de mucho (es el puntero de memoria). Incluso puedes quitar el `toString()` ya que el **`System.out.println()` llama automáticamente al `toString()`**.

Añade ahora en la clase Perro la sobreescritura de la siguiente forma:

```
@Override
public String toString(){
    return this.nombreHumano;
}
```

Y prueba de nuevo el programa principal.

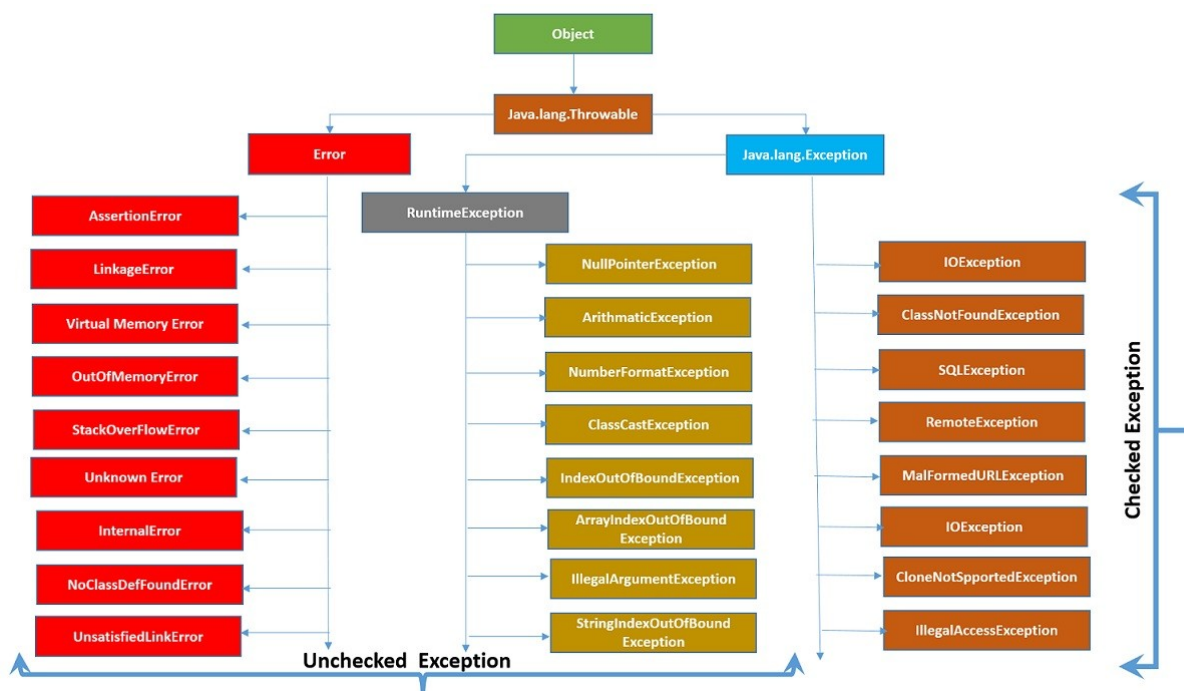
Nota ED: En el menú contextual de Source Action hay dos opciones en relación con la sobreescritura. Por un lado `Override/Implement methods` permite sobrecribir funciones de la clase padre.

Por otro `Generate toString()` es una ayuda para automatizar la generación del `toString()` sobreescrito devolviendo valores de ciertas propiedades que se seleccionen.

La sobreescritura hay que gestionarla con cuidado en ciertas situaciones, sobre todo en constructores, de ahí que diversos IDEs den Warnings en ciertas situaciones. Sobre este problema puedes leer más en el [Apéndice II](#).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Sobre herencia y excepciones.



Fuente <https://facingissuesonit.com/java-exception-handling/>

Cuando se trabaja con excepciones, realmente de lo que se dispone es de una clase Throwable de la cual heredan las clases Error y Exception que son dos grandes grupos de excepciones. El resto heredan de una de estas dos.

Siguiendo este método, esto me permite crear nuevas excepciones de forma muy simple. Veamos un ejemplo:

Creo una nueva excepción:

```
class ExcesivoCalorException extends IllegalArgumentException {
}
```

Y luego puedo hacer una función como esta:

```
public void setTemperatura(double temperatura) {
    if (temperatura>30.0) {
        throw new ExcesivoCalorException();
    }
    this.temperatura = temperatura;
}
```

Además puedo sobrescribir métodos de la excepción según convenga. Puedes probar por ejemplo a sobrescribir toString() o getMessage();

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Polimorfismo

Capacidad de la POO que permite referenciar objetos de una determinada clase mediante objetos de clases padre o superiores en la jerarquía. La limitación es el acceso a los miembros que existen en dicha clase ancestral.

Por ejemplo:

```
Perro objPerro=new Perro(5,"Pastor","Laika");
Animal objAnimal;

objAnimal=objPerro;

System.out.println(objAnimal.getEdad());
```

Este código es perfectamente válido y muestra en pantalla la edad del perro. La limitación está en que al ser objAnimal de la clase Animal no puede acceder directamente a las propiedades de objPerro aunque apunte a un Perro.

Si escribes:

```
System.out.println(objAnimal.nombreHumano);
```

Te dará error ya que nombreHumano no existe en la clase Animal. Sin embargo puedes decirle al programa, mediante casting, que objAnimal está referenciando a un objeto del tipo Perro y entonces acceder a sus miembros:

```
System.out.println(((Perro)objAnimal).nombreHumano);
```

Esto tiene diversas utilidades, por ejemplo puedo crear un array o colección de animales y este puede contener tanto perros como moscas:

```
Animal[] bichos=new Animal[3];
bichos[0]=new Perro(5, "Pastor", "Laika");
bichos[1]=new Mosca();
bichos[2]=new Mosca();
```

O incluso un array (o colección) que pueda contener cualquier objeto que existe en Java aprovechando el hecho de que todo hereda de la clase *Object*.

```
Perro perro =new Perro(5, "Pastor", "Laika");
Integer numero=2;
String cadena="Hola";
ArrayList<Object> col=new ArrayList<Object>();
col.add(numero);
col.add(cadena);
col.add(perro);
for (Object o:col) {
    System.out.printf("valor %s : %s\n", o, o.getClass());
}
```


COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Para saber si un objeto es de una clase determinada se pueden hacer comparaciones como:

```
if (obj.getClass()==String.class)
```

Siendo obj el objeto de clase desconocida. En este caso comprobaría si el objeto obj es de la clase String.

Existe también la posibilidad de comparar con la sentencia **instanceof**. Por ejemplo:

```
if (obj instanceof String)
```

Pero hay que tener cuidado porque en este caso además de comparar clases, si la primera es subclase de la segunda también devuelve true, sin embargo el uso de getClass es para clases exactas. Puedes ver más información en:

<https://stackoverflow.com/a/37282578/2586768>

El polimorfismo también sirve para pasarle un tipo genérico como parámetro a una función.

```
public void alimentar(Animal bicho){ ...
```

```
public void procesar(Object o) {...
```

en el primer caso puedo pasarle como parámetro cualquier animal (objetos que hereden de Animal) y en el segundo podría pasarle absolutamente cualquier objeto Java que desee. Un ejemplo de este último caso lo veremos a continuación al sobrecribir *equals*.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Sobreescribir equals para comparar cualquier objeto.

En el siguiente ejemplo juntaremos la **sobreescritura** con el polimorfismo en el método **equals**, que al igual que *toString()* es una función definida en la clase *Object* y que es muy habitual sobreescribirla ya que sirve para comprobar igualdad entre objetos (como hacemos con Strings).

Veamos un ejemplo: Crea el programa principal con el siguiente código:

```
ArrayList<Perro> perros = new ArrayList<>();

perros.add(new Perro(2, "Pastor", "Richard"));
perros.add(new Perro(5, "Palleiro", "Lucky"));
perros.add(new Perro(5, "Pekines", "Buff"));
perros.add(new Perro(5, "Caniche", "Thor"));

Perro p = new Perro(5, "Pekines", "Buff");
System.out.printf("%s %s existe en la colección\n",
p.nombreHumano, perros.contains(p) ? "" : "No");
```

Si lo ejecutas comprobarás que dice que no existe en la colección, lo cual no es correcto pues tiene exactamente las mismas propiedades.

¿Qué está sucediendo? Este método (y otros como *indexOf* o *remove*) lo que comparan son directamente las posiciones de memoria de los objetos. De hecho si el objeto **p** lo obtenéis de la siguiente forma:


```
Perro p = perros.get(2);
```

Entonces pasa a funcionar como a nosotros nos interesa pues en este caso *p* apunta directamente a un elemento de la colección. Pero volviendo a nuestra situación ¿Cómo solucionarlo?

Internamente, métodos como *contains* e *indexOf* llaman a su vez a otro método denominado **equals** que se encuentra definido en la clase *Object* y por tanto existe en todos los objetos Java por herencia.

Por defecto *equals* compara las direcciones de memoria de los objetos implicados, pues es como está programado en la clase *Object*. Pero se puede sobreescribir en clases heredadas. Si recuerdas, al comparar cadenas sí que funciona bien. Esto es así porque en la clase *String* sí que está sobreescrito el método *equals* para que compare las cadenas como nos interese (por el contenido y no por la dirección de memoria).

Vamos por tanto a **sobreescribir** el método **equals** dentro de la clase **Perro** para indicarle al programa como debe hacer para comparar la igualdad de elementos tipo *Perro*.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Escribe dentro de la clase Perro lo siguiente:

```
public boolean equals(Object o) {
    if (o != null && o.getClass() == Perro.class) {
        Perro p = (Perro) o;
        if (this.nombreHumano.equals(p.nombreHumano)
            && this.raza.equals(p.raza)
            && this.getEdad() == p.getEdad()) {
            return true;
        }
    }
    return false;
}
```

Si ejecutas de nuevo el programa verás que funciona correctamente. Esto es porque ahora cuando se ejecutan funciones como contains, estas usan la versión sobreescrita del equals.

Aquí se ve la potencia enorme de la sobreescritura, ya que puede llegar a ejecutar funciones que aún no están escritas para cambiar/mejorar el comportamiento.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Interfaces

En Java no existe la denominada herencia múltiple, es decir, que una clase no puede heredar de varias. Esta propiedad es complicada y muchos lenguajes de programación lo resuelve de una forma sencilla mediante las llamadas *interfaces*.

Una interfaz es una definición de una clase con ciertos métodos pero que **no están codificados**. Si no que hay que realizarlos en la clase que "implemente la interfaz". Esto es muy útil para potenciar aún más el concepto de polimorfismo entre clases que no heredan de la misma superclase. Entendámoslo con un ejemplo.

Supongamos que tenemos en cierta clase el método **competir**. A dicho método nos gustaría pasarle como parámetro uno o varios objetos que puedan ejecutar el método correr, por ejemplo un atleta, un caballo, un coche, etc... Pero resulta que la herencia de estas clases no es común de forma que no se puede meter el método correr en la superclase.

La solución mediante interfaces sería la siguiente:

```
interface ICorredor {
    void correr();    //sólo la definición
}

class Coche implements ICorredor {
    public void correr(){
        //código para mover las ruedas
    }
}

class Caballo extends Animal implements ICorredor {
    public void correr(){
        //código para mover las patas
    }
}
```

La función competir que estaría en alguna otra clase (clase Carrera por ejemplo) sería:

```
public void competir(ICorredor[] c){
    for (int i=0; i<c.length; i++)
        c[i].correr();
}
```

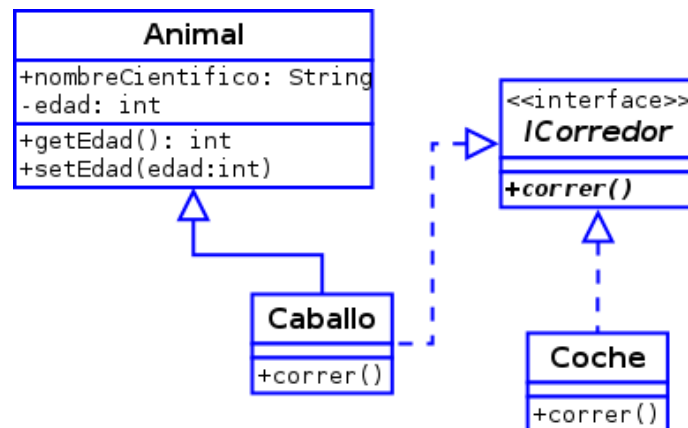
Pueden entenderse las interfaces como unas reglas o protocolos que deben cumplir las clases que las implementen, para asegurar que disponen de unas funciones determinadas que pueden ser llamadas desde distintos sitios.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Puedes ver más ejemplos de uso de colecciones e interfaces en los apendices [III](#) y [IV](#)

En una interfaz, además de funciones, se permite definir constantes con un valor predefinido y también cuerpo de las funciones pero por ahora no usaremos estas características.

En UML un Interface se representa como una clase poniendo el "estereotipo" interface y todo su contenido en cursiva.



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Clases Abstractas

En ocasiones sucede que una superclase nunca es instanciada. Por ejemplo el caso de animal. Tendremos objetos Mosca, objetos Perro, etc. pero un objeto tipo Animal no tiene sentido. En realidad usamos la clase Animal exclusivamente para reunir una serie de miembros comunes a todos los animales y luego para aplicar polimorfismo.

En casos así conviene declarar la clase como abstracta:

```
abstract public class Animal {
...

```

Además esto tiene otra ventaja y es que si se desea que todas las subclases de dicha clase estén obligadas a implementar cierto método, no es necesario crear un Interfaz a mayores si no que llega con crear un método abstracto que se define solo con la cabecera como hacemos en los interfaces.

Prueba a introducir en animal el siguiente método abstracto:

```
abstract public void comer();
```

Inmediatamente tras escribirlos van a aparecer errores tanto en la clase Mosca como en Perro que indican que debe sobreescribirse dicho método. Deja que el Netbeans lo corrija.

Por supuesto un método abstracto solo se puede crear en una clase abstracta.

En UML una clase abstracta se indica poniendo su nombre en cursiva (Itálica).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Apéndice I: Enumerados

Un enumerado es un tipo que contiene una serie de campos que son constantes. La principal utilidad es darle claridad al programa que estemos haciendo ya que nos permite usar palabras en lugar de números para codificar algún elemento de interés.

Realmente un enumerado es una clase que se define de forma ligeramente diferente, lo que le da un comportamiento especial. Podemos definir por ejemplo un enumerado para los días de la semana:

```
enum DiaSemana {
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}
```

Fíjate que escribimos la primera con mayúscula por ser una clase. Las constantes definidas en su interior (así como otros miembros que veremos) son estáticas por lo que no es necesario instanciar el enumerado.

En el programa principal podemos realizar sentencias válidas como las siguientes:

```
DiaSemana dia = DiaSemana.LUNES;
System.out.printf("dias: %s y %s\n", DiaSemana.DOMINGO, dia);
```

O también:

```
for (DiaSemana d : DiaSemana.values())
    System.out.print(d + " ");
```

Esta última permite plantear un menú con los enumerados para luego seleccionar una de las opciones. Suele ser un método muy cómodo.

El método *values()* devuelve un array con todos los enumerados.

Otra opción es pedirle al usuario un dato como *String* y convertirlo a enumerado. Es importante pasarlo a mayúsculas ya que debe coincidir exactamente el *String* con el enumerado:

```
System.out.println("\n\nSelecciona un día");
dia=DiaSemana.valueOf(sc.nextLine().toUpperCase());
System.out.printf("dia seleccionado: %s\n", dia);
switch (dia) {
case LUNES:
    System.out.println("Día horrorooooo!");
    break;
case SABADO:
    System.out.println("¡¡¡Fin de semana!!!");
    break;
case DOMINGO:
    System.out.println("A recuperarse de la noche anterior");
    break;
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

default:
    System.out.println("¡A trabajar!");
}

```

Uso avanzado

Los enumerados son clases lo que me permite realizar ciertas tareas y darle cierto valor añadido. A un enumerado se le pueden asignar distintos valores mediante un constructor privado y propiedades también privadas. Veámoslo con un ejemplo:

```

enum Naves {
    TIE_FIGHTER(1,"Tie Fighter", 100),
    DESTRUCTOR_ESTELAR (2,"Destructor Estelar", 1000),
    ESTRELLA_MUERTE (3,"Estrella de la Muerte", 100000),
    X_WING(4,"X-Wing", 120),
    HALCON_MILENARIO (5,"Halcón Milenario", 500);

    private int codigo;
    public int getCodigo(){
        return codigo;
    }

    private String nombre;
    @Override
    public String toString(){
        return nombre;
    }

    private int poder;
    public int getPoder() {
        return poder;
    }

    private Naves(int codigo, String nombre, int poder){
        this.codigo=codigo;
        this.nombre=nombre;
        this.poder =poder;
    }
}

```

Aquí se le asocia a cada uno de los enumerados un código único, un String que representa lo que queremos que se escriba (de hecho sobreescribimos *toString()*) y la potencia de la nave. El sentido de esto es como sigue: Cuando se usa una constante enumerada en un programa, es como si se llamara al constructor con los valores colocados entre paréntesis en la definición de la constante. Esto me permite acceder a otros valores asociados a dicha constante.

A continuación un ejemplo simple de uso:

```

Naves[] n=Naves.values();
for (int i=0; i<n.length; i++){
    System.out.printf("%d.-%25s ... %7d\n",

```


COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

        n[i].getCodigo(), n[i], n[i].getPoder());
    }

```

Teniendo claros los conceptos vistos hasta el momento no deberías tener problemas para leer y entender este código.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Apéndice II: Sobre la sobreescritura en constructores

Como ya se ha comentado más de una vez, los IDEs como Netbeans y otros muestran un Warning en el caso de usar una llamada a un setter dentro de los constructores. En general este aviso lo dará siempre que se llame dentro de un constructor a un método que puede sobreescribirse. ¿Por qué sucede?

Cuando se ejecuta un constructor de una subclase, este primero llama al constructor de su clase padre y si en dicho constructor de clase padre se hacen llamadas a las funciones sobreescritas sin terminar de inicializar todo, puede ser fuente de problemas. Veámoslo con un ejemplo:

En la clase Mosca sobreescribe el setEdad de la siguiente manera. El objetivo es que mosca o tiene 0 años o tiene 1.

```
@Override
public void setEdad(int edad){
    super.setEdad(0);
    if (edad>1){
        super.setEdad(1);
    }
}
```

Añade también en Mosca el siguiente constructor:

```
public Mosca(int edad){
    super(edad, "");
}
```

Modifica el constructor de Animal para que “utilice” la edad (aunque solo sea para mostrarlo):

```
public Animal (int edad, String nombreCientifico){
    this.setEdad(edad);
    this.nombreCientifico = nombreCientifico;
    System.out.println(edad);
}
```

Ahora en el main escribe el siguiente código y ejecuta:

```
Mosca m= new Mosca(22);
```

Verás que en pantalla se muestra 22 en lugar de 1 que sería lo correcto para la mosca. Esto es porque se ejecuta el setEdad de Animal antes que el setEdad de mosca y entre uno y otro se ha utilizado la edad para mostrarla.

Ojo, porque este problema se puede dar con cualquier método que pueda ser sobreescrito y al que se llame en el constructor, no solo a los set.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Hay varias soluciones, pero quizá lo que más se recomienda es o declarar los set (o el método usado) como final ya que esto impide que se sobrescriban o inicializar sin llamar a ningún método (esto dependiendo del caso puede no funcionar o no ser viable).

El tema no es baladí y como siempre hay que analizar cada caso. Incluso puede usarse "indebidamente" pero documentarlo bien para que quien vaya a usar ciertas clases lo tenga en cuenta.

Si quieres leer más sobre el tema:

<http://stackoverflow.com/questions/3404301/whats-wrong-with-overridable-method-calls-in-constructors>

<http://stackoverflow.com/questions/8501735/using-setter-methods-in-constructor-bad-practice>

<http://stackoverflow.com/questions/4893558/calling-setters-from-a-constructor>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Apéndice III: Interfaz Comparable

En ocasiones nos puede interesar comparar objetos creados por nosotros o, lo que es mejor, usar funciones de ciertas clases que permiten cosas como ordenar objetos.

Por ejemplo si tenemos una colección de Strings denominada nombres y realizamos la ordenación mediante la función Collections.sort() funciona perfectamente. Es decir:

```
ArrayList<String> nombres= new ArrayList<>();
nombres.add("Paul");
nombres.add("George");
nombres.add("Ringo");
nombres.add("John");

Collections.sort(nombres);
for(String s:nombres){
    System.out.println(s);
}
```

Sin embargo si tenemos una clase a medida como puede ser la siguiente:

```
class Pelicula {
    String titulo;
    int año;

    public Pelicula(String titulo, int año) {
        this.titulo = titulo;
        this.año = año;
    }
}
```

Al intentar ordenar una colección de películas nos dará error. Pruébalo:

```
ArrayList<Pelicula> peliculas = new ArrayList<>();
peliculas.add(new Pelicula("El Padrino",1972));
peliculas.add(new Pelicula("Tres anuncios en las afueras",2018));
peliculas.add(new Pelicula("Cadena Perpetua",1995));
peliculas.add(new Pelicula("El Imperio Contraataca",1980));
peliculas.add(new Pelicula("Blade Runner",1983));

Collections.sort(peliculas);
```

Para poder indicar orden en una clase, podemos establecerlo con el interface Comparable que nos obliga a implementar la función compareTo. Redefine la clase película de la siguiente manera:

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

class Pelicula implements Comparable<Pelicula>{
    String titulo;
    int año;

    public Pelicula(String titulo, int año) {
        this.titulo = titulo;
        this.año = año;
    }

    @Override
    public int compareTo(Pelicula p) {
        // Si el valor devuelto es negativo implica menor, si positivo mayor
        // Si cero es que son iguales
        if(this.año<p.año)
            return -1;
        if(this.año>p.año)
            return 1;
        return 0;
    }
}

```

Ahora el programa principal ya no da error y puedes ver la colección ordenada haciendo:

```

for(Pelicula p:peliculas){
    System.out.println(p.titulo);
}

```

La forma en que trabaja compareTo es como se indica en el comentario, cuando queramos indicar que un elemento es menor que otro, devolvemos un número negativo, en este caso y como se hace de forma muy habitual devolvemos -1. Si el elemento es mayor se devuelve un positivo y si son iguales 0.

El motivo que el interface Comparable sea tipado es porque en ciertas clases puede interesar hacer una comparación con otra clase y no con la misma. Por ejemplo podrías cambiar la clase Pelicula de la siguiente forma (aunque dejaría de funcionar el programa principal anterior):

```

class Pelicula implements Comparable<String>{
    String titulo;
    int año;

    public Pelicula(String titulo, int año) {
        this.titulo = titulo;
        this.año = año;
    }

    @Override

```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

public int compareTo(String p) {
    return p.compareToIgnoreCase(this.titulo);
}

```

En este caso podrías pasar a comparar objetos de tipo película con cadenas directamente.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Apéndice IV: Crea tu propio tipo de colección.

La clase ArrayList que hemos estado usando implementa la interfaz List, lo que le da sus capacidades como colección. Existe otra colección que es probable que veáis en algún ejemplo que es LinkedList que es prácticamente idéntica y es porque también implementa la interfaz List.

La principal diferencia entre ambas es que mientras ArrayList está más optimizada para búsquedas, LinkedList está optimizada en inserción y borrado de elementos.

Dispones de más información en: <https://beginnersbook.com/2013/12/difference-between-arraylist-and-linkedlist-in-java/>

Por supuesto sabiendo esto podríais crearos vuestra propia colección implementando el interface List de la siguiente manera:

```
class MiColeccion<T> implements List<T>{}
```

Luego deja que el IDE autoimplemente los métodos y ahí tendrás que programar el add, remove y otros según lo que desees hacer.

La T es lo que se denomina en Java tipos genéricos que nos permite jugar en una clase con distintos tipos aunque la definición sea la misma como en el caso de los ArrayList.

También puedes ver el código fuente de ArrayList aquí:

<http://developer.classpath.org/doc/java/util/ArrayList-source.html>

Y podéis ver un ejemplo sobre genéricos aquí:

<http://jonsegador.com/2012/10/clases-y-tipos-genericos-en-java/>

Finalmente comentar que desde Java 8 se permite la implementación de métodos por defecto dentro de la propia interface y también de métodos estáticos. Nosotros no usaremos esa característica pues su objetivo principal es la escalabilidad. De todas formas si quieres leer más sobre esta nueva funcionalidad accede a este enlace:

http://aitorm.github.io/java%208/java_8_interfaces/