

M3

PROGRAMACIÓN

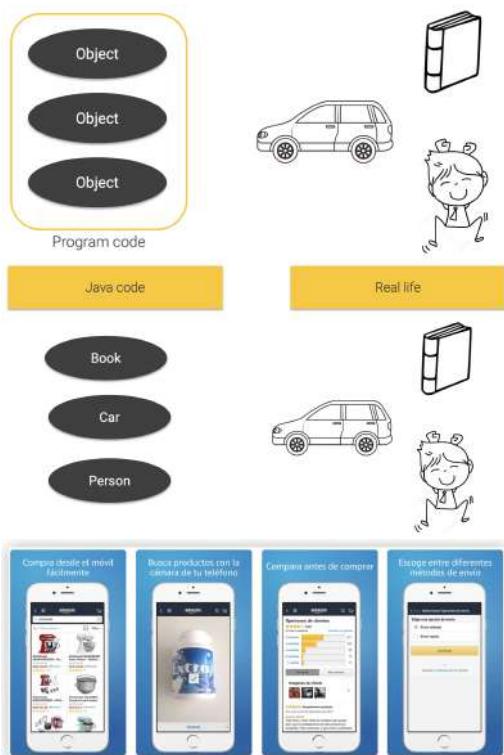
UF

4

JAVA PREDEFINED CLASSES

OPP INTRODUCTION

WHAT IS OPP?



La programación orientada a objetos o abreviado OOP o POO, es un tipo de programación que consiste en crear aplicaciones usando unidades llamadas objetos.

Cada unidad representa un objeto en la vida real, como por ejemplo un libro, un coche, una persona...

Hoy en día los lenguajes de programación más utilizados se basan en la programación orientada a objetos. Cuando programamos estamos resolviendo un problema real, entonces la idea es que el código tenga sentido con lo que estamos intentando resolver.

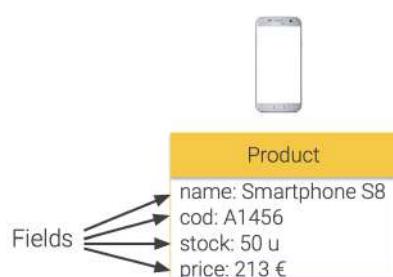
Una aplicación se puede entender como una simulación de un escenario del mundo real, donde un conjunto de elementos, los objetos, interactúan entre ellos para llevar a cabo una tarea que se quiere resolver.

Store application



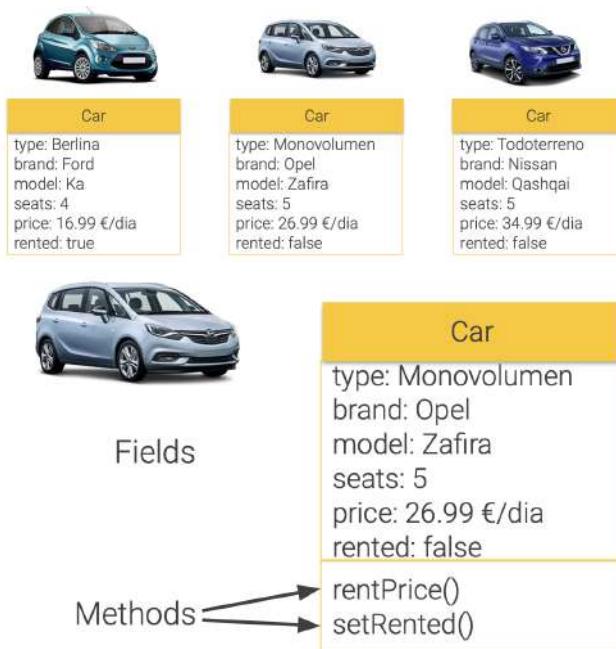
Por ejemplo, nuestra aplicación debería disponer de un objeto para cada uno de los productos que tenga la tienda. Cada objeto es responsable de tener los datos que le describan. El objeto producto, por ejemplo tendrá un nombre, un código, un stock y un precio.

Fields



Esto es lo que se denominan fields o atributos y son las características del objeto.

Renting a car application



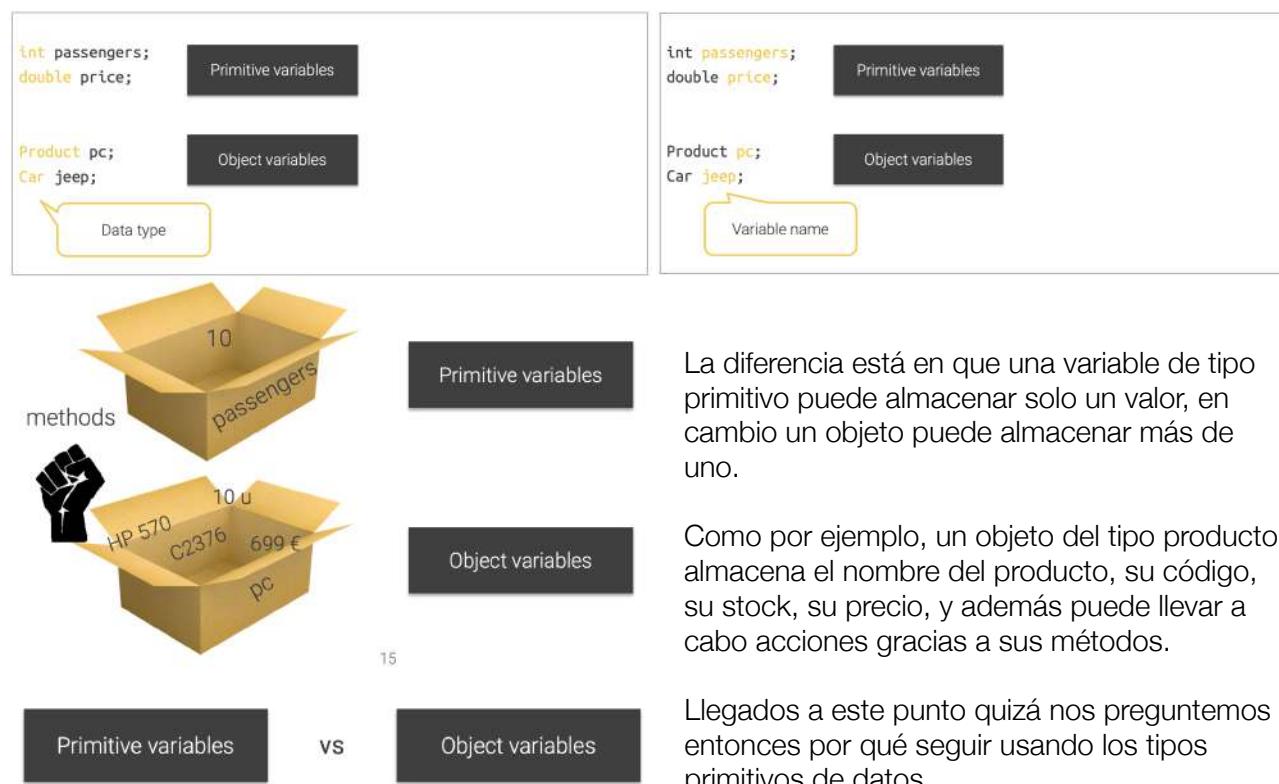
O por ejemplo si nuestra aplicación es para gestionar el alquiler de coches, el objeto coche tendría un tipo, una marca, un modelo, el número de asientos, el precio de alquiler, y está alquilado o no.

Los objetos además de tener características que registramos como atributos, pueden llevar a cabo acciones, estas acciones se registran como métodos.

Por ejemplo, de un coche, podemos querer calcular su precio de alquiler por disponer de él durante una semana.

Variable data types

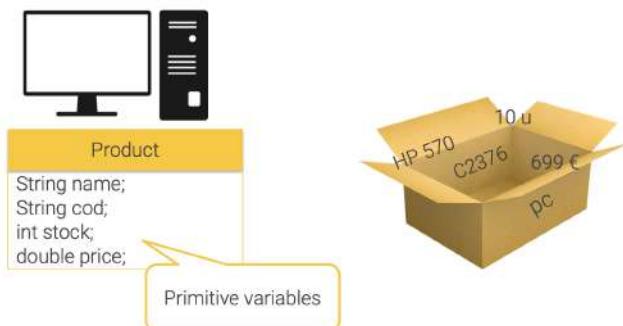
Ahora que ya tenemos una idea de lo que consiste la programación orientada a objetos. ¿Cómo usamos objetos en JAVA? La respuesta es fácil, como cualquier otro tipo de dato que hemos creado hasta ahora. Declararemos su tipo seguido del nombre de la variable. Un objeto no es más que otro tipo de dato.



La diferencia está en que una variable de tipo primitivo puede almacenar solo un valor, en cambio un objeto puede almacenar más de uno.

Como por ejemplo, un objeto del tipo producto almacena el nombre del producto, su código, su stock, su precio, y además puede llevar a cabo acciones gracias a sus métodos.

Llegados a este punto quizás nos preguntemos entonces por qué seguir usando los tipos primitivos de datos.



Ningún programa puede existir sin los tipos de datos primitivos, de hecho los objetos están hechos a partir de estos tipos de datos primitivos.

- Objetos representan objetos en la vida real.
- Atributos (fields): características de un objeto.
- Métodos (methods): acciones de un objeto.
- POO: código más organizado, más fácil de entender y más fácil de mantener.

Entonces podemos resumir que los objetos de nuestro código nos permiten representar objetos de la vida real, disponen de atributos o características que registraremos como variables denominadas fields, y pueden tener métodos que usaremos para llevar a cabo acciones sobre ellos.

Así la programación orientada a objetos hará que nuestro código esté mejor organizado, sea más fácil de entender y de mantener.

CLASSES AND OBJECTS

methods



Los objetos permiten almacenar muchos tipos de datos en una sola variable y además pueden llevar a cabo acciones gracias a sus métodos así que ayudan a que nuestro código quede bien organizado. Entonces, ¿cómo podemos los desarrolladores crear nuestros propios objetos? Pues lo hacemos creando clases.



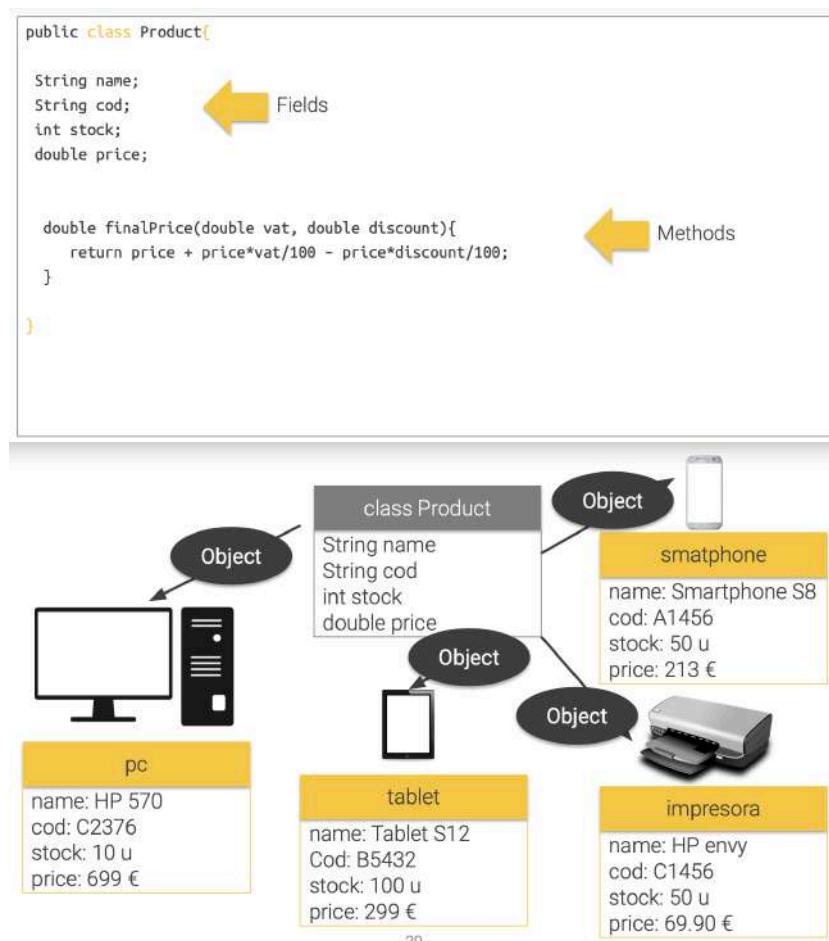
A class = A data type

Una clase simplemente es un tipo de dato. Para explicarlo de forma sencilla una clase es el plano que define cómo debería ser el objeto.



An object = A variable

Un objeto, por otro lado, es una variable. Es decir, es una entidad real que ha sido creada desde la clase.



En otras palabras, cuando estemos definiendo cómo debería ser el objeto en la clase es donde listaremos todos los atributos e implementaremos los métodos.

Así si tenemos definida la clase producto con una serie de atributos podremos crear un objeto PC de esta clase. Este objeto tendrá estos atributos con un determinado valor.

Por ejemplo, HP570 de nombre, C2376 de código, 10 unidad de stock, y un precio de 699 euros. Podemos crear muchos objetos a partir de esta clase. Y cada uno de estos objetos tendrá diferentes valores para los mismos atributos, pero todos incluirán el total de atributos definidos en la clase producto.



En JAVA cada clase debería ser creada en un fichero independiente. Este fichero tendrá por nombre el nombre de la clase y la extensión .java. Así el fichero producto.java contendrá el código de la clase producto.

Una aplicación típica en JAVA no es más que un conjunto de clase interactuando unas con otras. De hecho no puede existir código JAVA fuera de una clase.

Las clases y objetos entonces son términos diferentes que a veces pueden parecer iguales porque se refieren a la misma cosa, pero cada uno tiene un significado diferente. Una clase es un tipo de dato y un objeto es una variable. Una clase se define en un fichero propio con la extensión .java y un objeto puede aparecer en cualquier parte de código como variable.

Class	Object
<ul style="list-style-type: none"> • It's a data type • Has its own file • Defines the structure • CamelCase naming convention (starts with an upper case) 	<ul style="list-style-type: none"> • It's a variable • Scattered around the project • Used to implement logic • camelCase naming convention (starts with a lower case)
Examples	
Product	pc
Car	jeep

En una clase se define la estructura, esto son los atributos y métodos que tendrán todos los objetos creados a partir de ella. El nombre de una clase sigue la regla UpperCamelCase con la primera letra en mayúscula, y el identificador de un objeto, como el identificador de cualquier otra variable sigue la regla loweCamelCase con la primera letra en minúscula.

JAVA PRE-DEFINED CLASSES

JAVA PRE-DEFINED CLASSES



Como JAVA es un lenguaje de programación orientad a objetos, dispone de una gran cantidad de clases predefinidas que los programadores podemos usar cuando desarrollamos nuestro código. Como la clase `String` que permite representar cadenas de caracteres y trabajar con ellos; o la clase `Scanner` que permite leer datos introducidos por el usuario desde la consola; o la clase `File` que permite trabajar con ficheros y directorios...

Wrapper classes

Class	Primitive type
<code>Integer</code>	<code>int</code>
<code>Long</code>	<code>long</code>
<code>Double</code>	<code>double</code>
<code>Character</code>	<code>char</code>
<code>String</code>	<code>char[]</code>

Hasta incluye clases que envuelven a los tipos de datos primitivos para ofrecer funcionalidades extra a través de sus métodos. En la captura podemos verlas así como su equivalente en tipo primitivo. Nos permiten olvidarnos de los tipos primitivos y hacer que todas las variables de nuestro programa sean objetos.

Sin embargo, no es lo recomendable, la regla es usar tipos primitivos siempre que sea posible, para así hacer que la aplicación sea más eficiente.

PACKAGES

La enorme cantidad de clases de las que dispone la plataforma JAVA se organizan en librerías o paquetes. Un paquete es equivalente a una carpeta en nuestro sistema de ficheros. Permite agrupar clases relacionadas. Ahora vamos a ver los paquetes o librerías que probablemente más usaremos.

Java.lang



Es el más importante porque es el paquete base que contiene las clases fundamentales que se usan en cualquier programa JAVA, como son la clase string, math, integer, double...

Java.util



Este paquete es junto con java.lang uno de los paquetes más usados, porque contiene clases que permiten trabajar con colecciones de datos como arrayList, y otras muchas clases de utilidades como scanner, random...

Java.time



Contiene clases que permiten trabajar con fechas, horas, intervalos de tiempo...

Java.io



El paquete java.io permite realizar operaciones con ficheros y directorios desde aplicaciones JAVA.

Java.sql



Lo usaremos para gestionar información almacenada en bases de datos desde nuestras aplicaciones java.

Javax.swing



Y este paquete lo usaremos para desarrollar la interfaz gráfica de nuestras aplicaciones java.

Puedes ver más información en el siguiente enlace: [Java SE Platform API Specification](#)

JAVA PRE-DEFINED CLASSES: DOCUMENTATION

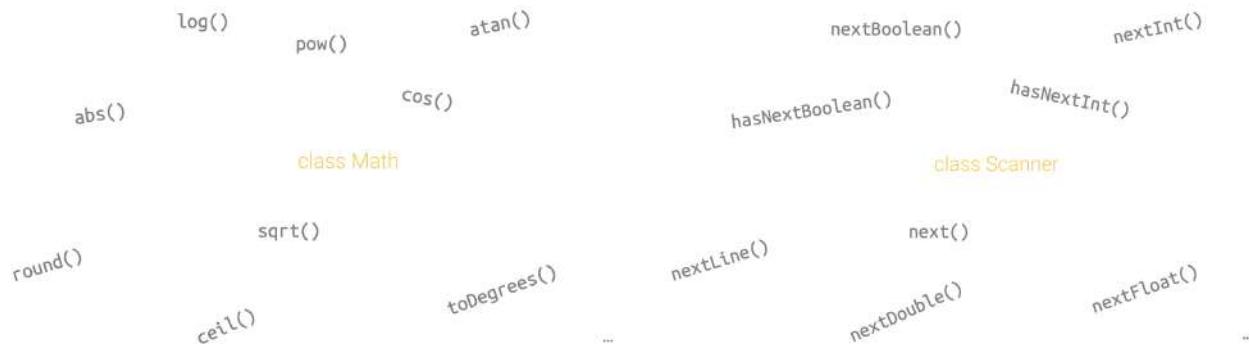
JAVA DOCUMENTATION



Las clases predefinidas del lenguaje JAVA disponen de atributos, pero sobretodo de potentes métodos que podemos usar para llevar a cabo tareas en nuestras aplicaciones.

Así por ejemplo la clase `String` dispone de métodos que nos permitirán trabajar con cadenas de caracteres, métodos como el método `charAt` que nos permitirá localizar el carácter que ocupa una determinada posición dentro de la cadena de texto, o el método `equals` que nos permitirá saber si 2 strings son iguales, o el método `length` que nos permitirá saber cuántos caracteres tiene la cadena de texto, y muchos más que iremos descubriendo si no los conocemos aún.

Debemos comprobar que cada método tiene un nombre que identifica de forma clara la operación que podremos hacer con él, y que los nombres de los métodos están escritos siguiendo la regla loweCamelCase (primera letra en minúscula y si contiene más de una palabra la inicial de cada una en mayúscula).



Otro ejemplo es la clase math que dispone de métodos para llevar a cabo operaciones matemáticas básicas o la clase scanner que usamos para leer datos desde consola.

Find out about built-in functions

Para saber cómo funcionan las clases del lenguaje JAVA como string, scanner, math y saber usar su métodos es clave que consultemos la documentación de JAVA. JAVA pertenece a Oracle que se encarga de mantener actualizada esta información en su web. Programadores de todos los niveles usamos esta documentación habitualmente para desarrollar nuestro código.

HOW TO USE JAVA DOCUMENTATION



Vamos a consultar juntos la documentación JAVA sobre la clase string para aprender cómo usarla. Hemos escrito en Google String (el nombre de la clase que queremos buscar) java 10 (que es la versión de java con la que estamos trabajando) y documentation.

El primer resultado que nos aparece es la página de docs.oracle que es la que nos interesa y clicamos sobre ella. Esto nos lleva a la página con la documentación de la clase string en JAVA. Vamos a ver qué hay en ella.

Class String

La estructura de esta página es la estructura general y que se repite en todas las páginas de documentación de Oracle sobre sus clases. Lo primero que encontramos es el nombre de la clase (Class String) y el paquete al que pertenece java.lang. Lo siguiente que encontramos es una descripción general y ejemplos de uso de la clase.

La clase string representa cadenas de caracteres y nos muestra que una forma de inicialización de objetos del tipo string básica es a través de literales. Para ello escribimos el tipo de dato, String str (identificamos el objeto) = "abc"; (y con el operador igual le asignamos el valor entre comillas dobles).

También nos informa de que tenemos otras muchas formas de inicializar strings que ya iremos viendo. Si seguimos hacia abajo nos informa de que la clase string tiene múltiples métodos para hacer diferentes acciones sobre estas cadenas de texto. Por ejemplo para examinar carácter a carácter su contenido, para comparar strings, para convertir de mayúsculas a minúsculas...

También nos informa por ejemplo de que disponemos del operador + para poder concatenar strings. Y otra mucha información que ya iremos consultando a medida que vaya siendo necesario. Si seguimos para abajo nos encontramos con la sección field summary, y en esta sección Oracle nos muestra los fields o atributos que dispone la clase.

Los fields de la clase string no nos interesa demasiado así que pasaremos a la siguiente sección que es constructor summary. En esta sección Oracle nos informa de los métodos constructores que son unos métodos especiales que disponen las clases para permitir inicializar objetos a partir de ellas.

Los métodos constructores siempre se identifican con el nombre de la clase. Por ejemplo, la clase string tiene diferentes métodos, todos ellos con el nombre string. La diferencia entre uno u otro son los parámetros de entrada de que disponen.

Por ejemplo la clase string permite inicializarse a través de una raíz de bytes o a través de una raíz de chars. Vamos a ver un ejemplo para saber cómo utilizar uno de estos métodos constructores. Para utilizar un método constructor, a parte de declarar el tipo de dato, usaremos el operador new seguido del nombre de clase y entre paréntesis si tiene los parámetros de entrada.

En el caso de ejemplo estamos utilizando un constructor que permite inicializar un objeto del tipo string a partir de un array de caracteres, el array data. Si seguimos navegando en la web hacia abajo nos encontramos con la última sección que es la sección método summary.

Aquí Oracle lista por orden alfabético todos los métodos que dispone la clase. Vamos para abajo para consultar uno de ellos, por ejemplo el método lenght.

A parte de la identificación del método en la parte derecha nos muestra una breve descripción de lo que va a hacer el método. Nos va a devolver la longitud o número de caracteres que tiene la cadena de texto. En la parte izquierda nos devuelve, si lo hace, el tipo de dato que retornará el método, en este caso un valor entero de tipo int.

Si pulsamos sobre el nombre del método nos llevará a una ampliación de la descripción del método. Vamos a consultar otro tipo de método en este caso que tenga parámetros de entrada, como por ejemplo el método charAt.

El método charAt devuelve el carácter que se encuentra en el índice especificado. Este método tiene un parámetro de entrada index y retorna un valor de tipo char. Si pulsamos sobre él vamos a la descripción del método y vemos que este parámetro de entrada index tiene que estar dentro de un rango de 0 a la longitud de la cadena de caracteres - 1.

Si no lo está se producirá en la llamada al método un error en tiempo de ejecución del tipo IndexOutOfBoundsException dado que no puede encontrar el carácter. Si sí que se encuentra dentro de rango devolverá el carácter que ocupe esta determinada posición.



- Class description
- Field Summary
- Constructor Summary
- Method Summary
 - Method without parameters
 - Method with parameters
 - Static methods

Al consultar la documentación JAVA de una de las clases más importantes de la plataforma, la clase string, hemos visto la información que ofrece Oracle a los desarrolladores para saber cómo interpretarla y usarla.

- Class description
- Field summary
- Constructor summary
- Method summary
 - Methods without parameters: length()
 - Methods with parameters: charAt()
 - Static methods.

Primero muestra una descripción de una clase y sus métodos, incluyendo ejemplos de uso. Luego muestra la sección field summary con los atributos de los que dispone la clase. A continuación está la sección constructor summary, con el detalle de los métodos constructores de la clase que permiten inicializar objetos. Y por último tenemos la sección method summary donde se listan por orden alfabético los métodos de los que dispone la clase.

Hemos visto cómo se ofrece la información de métodos que no disponen de parámetros de entrada como el método lenght y de métodos que sí necesitan de parámetros de entrada como el método charAt.

Class Math

Vamos a consultar ahora la documentación JAVA sobre otra clase, la clase math. Con esta consulta vamos a consultar especial atención a la sección field summary y a cómo se presenta la información de otro tipo de método que dispone JAVA, como son los métodos estáticos.

Vemos que tenemos el nombre de la clase Class Math y el paquete al que pertenece java.lang, nos dice que la clase math dispone de métodos que nos permiten llevar a cabo operaciones numéricas. Si seguimos haciendo scroll hacia abajo nos encontramos con la sección Field summary.

En esta sección nos encontramos con los atributos de la clase que son 2, el número E y el número PI que permiten representar los valores double (numéricos decimales) de estas 2 constantes matemáticas. Si nos damos cuenta, junto al tipo de dato aparece la palabra static. Esto indica que esos atributos son estáticos veremos más adelante cómo hacer uso de ellos en nuestro código.

Si seguimos hacia abajo vemos que esta clase no dispone de la sección constructor summary, es decir, no dispone de métodos constructores porque todos los métodos de esta clase son estáticos. Si seguimos hacia abajo hasta la sección method summary podemos ver que todos los métodos de la clase math tienen el modificador static delante del tipo de dato que retornan.

Un método estático significa que para poder usarlo no tenemos que crear un objeto de la clase, sino que podemos llamar directamente al método desde el nombre de la clase. Como hemos comentado antes más adelante veremos ejemplos sobre esto.

Vamos a ver uno de los métodos que dispone la clase math, como por ejemplo el método pow que permite calcular la potencia del número a elevado al exponente b. Igual que hemos visto en documentaciones anteriores de otras clases, vemos el nombre del método, los parámetros de entrada y su tipo (en este caso recibe 2 parámetros de entrada de tipo double), la descripción de lo que va a hacer el método, y el tipo, en el caso de que retorne un tipo, que devuelve, en este caso double.

Si pulsamos sobre el nombre del método accederemos a la descripción más detallada de lo que hace el método.



- Class description
- Field Summary
- Constructor Summary
- Method Summary
 - Method without parameters
 - Method with parameters
 - Static methods

En resumen la documentación que JAVA nos ofrece a los desarrolladores de sus clases predefinidas incluye una descripción de la clase y ejemplos de uso, una sección field summary con los atributos de los que dispone, estáticos o no, una sección constructor summary con los métodos constructores que nos permitirán crear objetos de la clase, y una sección method summary con las acciones que podremos hacer con la clase a través de los distintos tipos de métodos estáticos y no estáticos con y sin parámetros de entrada.

- Class description
- Field summary
 - Instance fields
 - Static fields
- Constructor summary
- Method Summary
 - Instance methods (with and without parameters)
 - Static methods (with and without parameters)

WORKING WITH JAVA PREDEFINED CLASSES

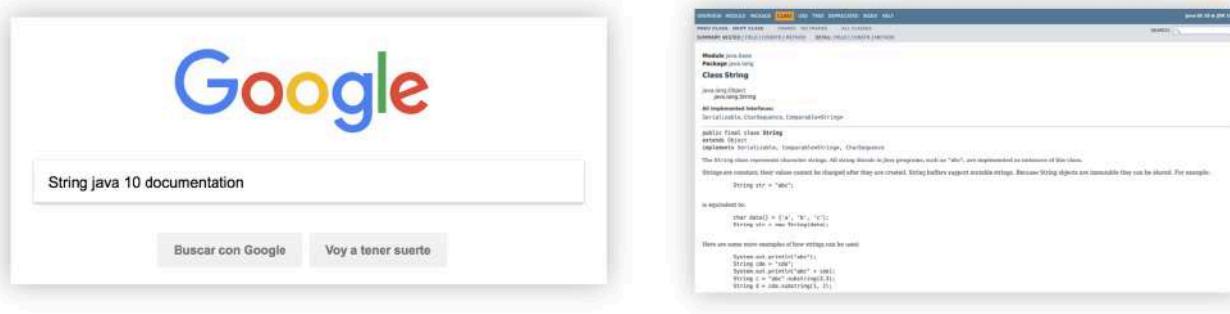
JAVA PREDEFINED CLASSES DOCUMENTATION

docs.oracle.com

- Class description
- Field summary
- Constructor summary
- Method summary

- Instance methods (with and without parameters)
- Static methods (with and without parameters)

Para poder trabajar con las clases predefinidas de JAVA en nuestras aplicaciones la documentación que Oracle nos ofrece de cada una de ellas incluye una descripción de la clase, una sección field summary con los atributos de los que dispone estáticos o no, una sección constructor summary con los métodos constructores que nos permitirán crear objetos de la clase, y una sección method summary donde encontraremos las acciones que podemos hacer con la clase a través de los distintos tipos de métodos estáticos o no estáticos con y sin parámetros de entrada.



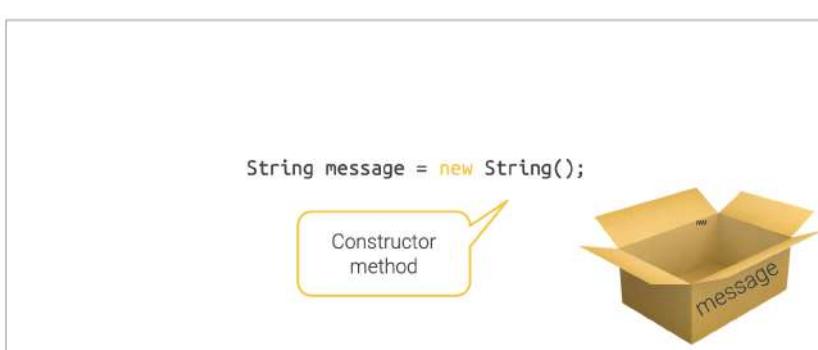
Accedemos a esta información escribiendo el nombre de la clase String, por ejemplo, seguido de la versión JAVA con la que estemos trabajando JAVA 10 en nuestro caso y documentation. Vamos a ahora a interpretar esta información usándola en nuestro código.

CREATING OBJECTS

docs.oracle.com

Para crear objetos de una clase la forma más habitual es usando cualquiera de sus métodos constructores que encontramos en la sección constructor summary.

New operator

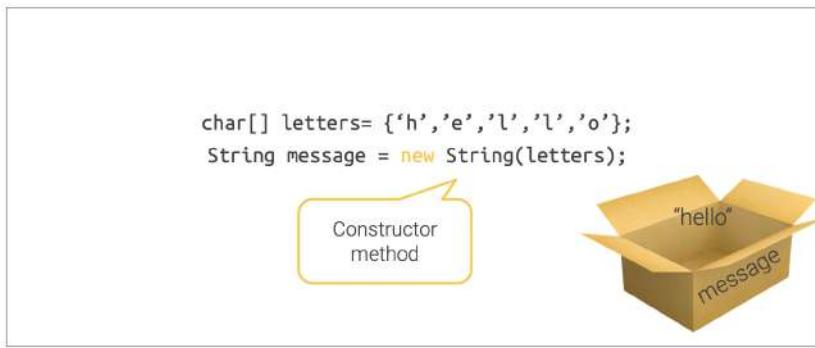


La sintaxis para usar un método constructor es la que vemos en la captura. Operador new seguido del método constructor.

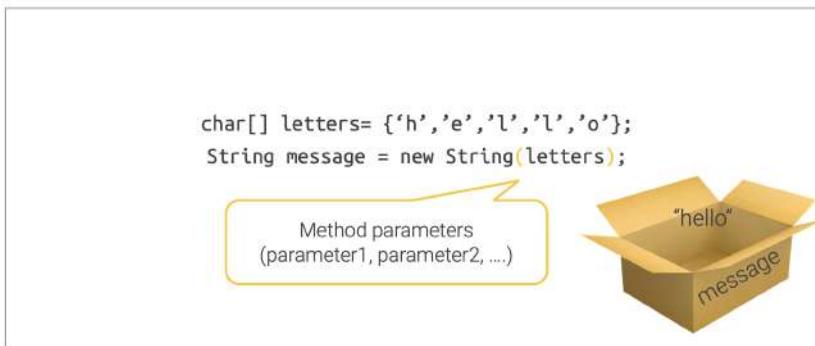
Como por ejemplo, la instrucción new String seguido de paréntesis vacíos que permite crear el objeto message inicializándolo a una cadena vacía, esto es una cadena sin caracteres.

Constructor Summary	
Constructors	Description
<code>String()</code>	Initializes a newly created <code>String</code> object so that it represents an empty character sequence.
<code>String(byte[] bytes)</code>	Constructs a new <code>String</code> by decoding the specified array of bytes using the platform's default charset.
<code>String(byte[] ascii, int hibyte)</code>	Deprecated. This method does not properly convert bytes into characters.
<code>String(byte[] bytes, int offset, int length)</code>	Constructs a new <code>String</code> by decoding the specified subarray of bytes using the platform's default charset.
<code>String(byte[] ascii, int offset, int offset, int count)</code>	Deprecated. This method does not properly convert bytes into characters.
<code>String(byte[] bytes, int offset, int length, String charsetName)</code>	Constructs a new <code>String</code> by decoding the specified subarray of bytes using the specified charset.
<code>String(byte[] bytes, int offset, int length, Charset charset)</code>	Constructs a new <code>String</code> by decoding the specified subarray of bytes using the specified charset.
<code>String(byte[], String charsetName)</code>	Constructs a new <code>String</code> by decoding the specified array of bytes using the specified charset.
<code>String(byte[], Charset charset)</code>	Constructs a new <code>String</code> by decoding the specified array of bytes using the specified charset.
<code>String(char[] value)</code>	Allocates a new <code>String</code> so that it represents the sequence of characters currently contained in the character array argument.
<code>String(char[] value, int offset, int count)</code>	Allocates a new <code>String</code> that contains characters from a subarray of the character array argument.
<code>String(int[] codePoints, int offset, int count)</code>	Allocates a new <code>String</code> that contains characters from a subarray of the Unicode code point array argument.
<code>String(String original)</code>	Initializes a newly created <code>String</code> object so that it represents the same sequence of characters as the argument; in other words, the newly created <code>String</code> is a copy of the argument <code>String</code> .

Normalmente una clase tiene más de un constructor, la diferencia entre uno y otro, está en el número y tipo de parámetros de entrada. En la captura podemos ver los distintos métodos constructores de los que dispone la clase string. Vamos a ver cómo usar uno de ellos que tenga parámetros de entrada, por ejemplo `String(char[] value)` que permite crear un objeto string a partir de un array de chars.



La instrucción `new String(letters)` creará el objeto `message` inicializándolo con la cadena de texto Hello que se formará a partir del array de chars.



Debemos observar que si el método tiene parámetros de entrada los incluimos dentro de los paréntesis separados por comas.

Method Summary	
All Methods	Static Methods
<code>copyValueOf(char[] data)</code>	Equivalent to <code>valueOf(char[])</code> .
<code>copyValueOf(char[] data, int offset, int count)</code>	Equivalent to <code>valueOf(char[], int, int)</code> .
<code>format(String format, Object... args)</code>	Returns a formatted string using the specified format string and arguments.
<code>format(Locale l, String format, Object... args)</code>	Returns a formatted string using the specified locale, format string, and arguments.
<code>join(CharSequence delimiter, CharSequence... elements)</code>	Returns a new <code>String</code> composed of copies of the <code>CharSequence</code> elements joined together with a copy of the specified delimiter.
<code>join(CharSequence delimiter, Iterable<? extends CharSequence> elements)</code>	Returns a new <code>String</code> composed of copies of the <code>CharSequence</code> elements joined together with a copy of the specified delimiter.
<code>valueOf(boolean b)</code>	Returns the string representation of the boolean argument.
<code>valueOf(char c)</code>	Returns the string representation of the char argument.
<code>valueOf(char[] data)</code>	Returns the string representation of the char array argument.
<code>valueOf(char[], int offset, int count)</code>	Returns the string representation of a specific subarray of the char array argument.
<code>valueOf(double d)</code>	Returns the string representation of the double argument.
<code>valueOf(float f)</code>	Returns the string representation of the float argument.
<code>valueOf(int i)</code>	Returns the string representation of the int argument.
<code>valueOf(long l)</code>	Returns the string representation of the long argument.
<code>valueOf(Object obj)</code>	Returns the string representation of the Object argument.



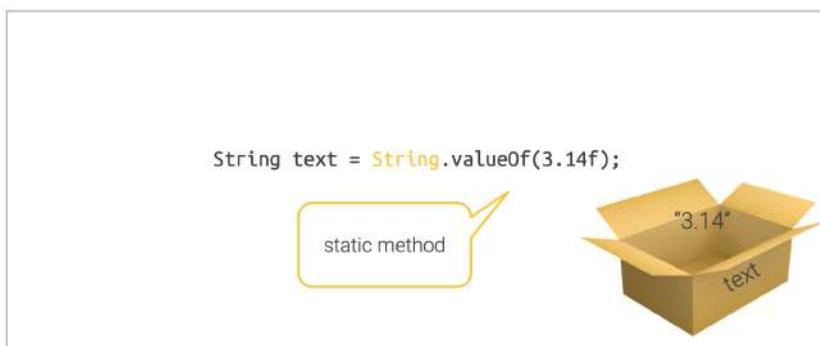
[String class Java Documentation](#)

Según la clase, nos podemos encontrar con otra posibilidad de inicialización, usando métodos estáticos. Por ejemplo la clase String dispone de los métodos estáticos que vemos en la captura.

Los reconocemos por llevar el modificador static. Estos métodos retornan un tipo string y por lo tanto permiten crear un objeto de este tipo. Por ejemplo el método `valueOf`, que recibe como parámetro de entrada, un valor numérico y retorna su equivalente cadena de texto.

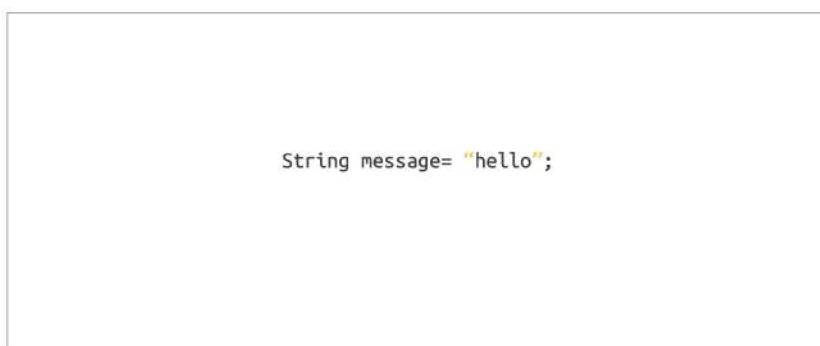
Static method

La sintaxis para usar un método estático, `valueOf` en el ejemplo de la captura, es el que podemos observar.



Sería nombre de la clase, punto y el método estático, debemos observar que no es necesario disponer de un objeto de tipo `String`, sino que escribimos el nombre de la clase, `String`, seguido de punto y el nombre del método `valueOf`, que retorna la cadena de texto resultante de convertir el número decimal introducido como parámetro.

String literal



La clase `String` es especial y aún permite otra forma de inicialización, usando literales. Esto es comillas dobles tal y como vemos en la captura.

Esta información la podemos encontrar en docs.oracle.com en la descripción de la clase.

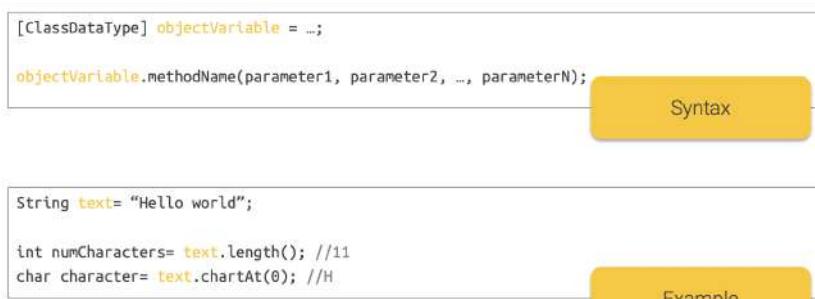
CALLING METHODS

docs.oracle.com

En la sección `method summary` es donde encontraremos información sobre los métodos de que dispone la clase predefinida. Una clase predefinida puede tener 2 tipos de métodos, de instancia y estáticos. Debemos recordar que los estáticos son los que llevan el modificador `static`.

¿Sabemos cómo usarlos en nuestro código? La sintaxis a utilizar es diferente entre ellos. Comenzaremos por los métodos de instancia o de objeto.

Instance method



Para usar un método de instancia de cualquier clase en JAVA usamos la sintaxis que vemos en pantalla, y necesario disponer de un objeto de la clase. Se usa el identificador del objeto, seguido de un punto, el nombre del método y paréntesis. Dependiendo del método puede ser necesario indicar información adicional, son los parámetros de entrada.

Éstos son los valores que se incluyen entre los paréntesis separados por comas.

A modo de ejemplo, creamos un objeto del tipo String que llamaremos text. Usamos los métodos de instancia lenght y charAt de la clase String que retornan el número de caracteres y el carácter de la posición indicada como parámetro de entrada respectivamente.

Debemos comprobar cómo es necesario disponer de un objeto de tipo String y usando el identificador del objeto llamamos a los métodos length y charAt.

Static method

```
ClassName.methodName(parameter1, parameter2, ..., parameterN);
```

Syntax

```
String texto = String.valueOf(3.1416f);
```

Example

Para usar un método estático escribimos el nombre de la clase seguido de un punto, el nombre del método y paréntesis.

Si el método tiene parámetros de entrada los incluimos entre los paréntesis separados por comas.

A modo de ejemplo usamos el método estático valueOf de la clase String. Debemos observar que no es necesario disponer de un objeto de tipo String, sino que escribimos el nombre de la clase String seguido de punto y el nombre del método valueOf que retorna la cadena de texto resultante de convertir el número decimal introducido como parámetro.

USING FIELDS

docs.oracle.com

En la sección field summary Oracle nos informará de qué atributos podemos usar de la clase predefinida. Igual que en los métodos hay 2 tipos de atributos, los estáticos y los de instancia.

Field Summary

Fields

Modifier and Type	Field	Description
static double	E	The double value that is closer than any other to e, the base of the natural logarithms.
static double	PI	The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.



[Math class Java Documentation](#)

Lo habitual es encontrar con atributos estáticos que se identifican igual que los métodos, porque llevan el modificador static, como por ejemplo los atributos E y PI de la clase math que representan el valor decimal de estas constantes matemáticas.

Static field

Para usar un atributo estático escribimos el nombre de la clase seguido de un punto y el nombre del atributo.



A modo de ejemplo usamos el atributo estático PI de la clase math. Debemos observar que no es necesario disponer de un objeto de tipo math, sino que escribimos el nombre de la clase, Math seguido de punto y el nombre del atributo que retorna el valor decimal de la constante PI que usamos para calcular el área de un círculo.

WORKING WITH JAVA PREDEFINED CLASSES: DEMO**JAVA PREDEFINED CLASSES DEMO**

Usaremos IntelliJ para desarrollar una aplicación que nos permita poner en práctica los conocimientos sobre algunas de las clases predefinidas del lenguaje más usadas. Veremos cómo crear objetos de estas clases, cómo usar sus distintos tipos de métodos y cómo usar sus atributos apoyándonos siempre en la documentación sobre las clases predefinidas que nos ofrece Oracle.

La aplicación permitirá elegir al usuario una figura geométrica para luego pedirle las dimensiones necesarias y así calcular su área. Entonces la aplicación hará uso de la consola de IntelliJ para interactuar con el usuario, mostrará datos usando las instrucciones SystemOut y leerá datos desde teclado usando la clase Scanner, trabajará con cadenas de texto y realizará operaciones matemáticas usando la clase Math.

Hemos dicho que vamos a solicitar al usuario qué tipo de figura es la que queremos utilizar para el cálculo del área, así que utilizaremos las instrucciones SystemOutPrint y Println para mostrar al usuario esta información y darle la bienvenida.

Una vez que le mostramos estos datos al usuario lo que queremos hacer es obtener y leer los datos que introduzca por consola, para esto necesitamos una variable de tipo scanner. Scanner al pertenecer al paquete java.util debemos que importarlo específicamente, solamente las clases que pertenecen al paquete java.lang no requiere de una importación.

La segunda cosa que vamos a ver es que hemos utilizado el constructor Scanner que recibe como parámetro de entrada system.in. Lo que necesitamos ahora es leer una cada de caracteres, un texto, bien cuadrado, triangulo o círculo del usuario.

Para esto desde la variable input. Tenemos que seleccionar uno de los métodos que dispone Scanner para hacer esta lectura de datos. En el caso de lectura de cadenas de caracteres Scanner dispone de dos métodos nextLine y next. La diferencia entre ellos es que next lee caracteres hasta encontrar un espacio en blanco, y nextLine permite leer cadenas de caracteres hasta encontrar un cambio de linea por parte del usuario. Así que nos interesaría leer solamente una palabra por lo tanto podemos seleccionar la instrucción input.next.

Y recoger el valor que nos retorna en una variable, por ejemplo figura. Una vez que hemos recogido en figura la opción seleccionada por el usuario, necesitaremos hacer unas operaciones u otras dependiendo de cuál sea así que usaremos una sentencia condicional de tipo switch.

Lo que hemos hecho ha sido crear la estructura de la sentencia condicional switch añadiendo las clases o las case necesarias para contemplar las acciones a realizar dependiendo de si la figura seleccionada el cuadrado, triángulo o círculo. Se ha añadido además la cláusula default para que en el caso de que el valor que se registra en figura no sea ninguna de estas opciones se muestre al usuario la información de que esta opción no es válida y el programa termine.

Tanto si es cuadrado, como si es círculo o triángulo lo que haremos es calcular el área y al final del todo mostrarla. Por lo tanto declaramos una variable en este punto de tipo doble que nos permita registrar este valor, y al finalizar la sentencia switch, mostraremos el resultado.

Actualizaremos el valor de área en cada una de las cláusulas case dependiendo de los datos de la figura. En el caso de un cuadrado necesitamos el lado, por lo tanto se lo pedimos al usuario y de nuevo necesitamos leer datos por parte del usuario, y para esto utilizamos la variable input y en este caso necesitamos leer un valor numérico y decimal, por lo tanto usamos nextDouble.

El valor que nos retorna es de tipo double que lo recogeremos en una variable que llamaremos lado. Haciendo uso de esta variable calculamos el área. El área de un cuadrado es lado por lado, lado². Necesitamos una operación matemática de elevar lado a la potencia 2. Vamos a hacer uso de la clase math.

La clase math dispone de métodos estáticos que nos permiten realizar estas operaciones, por esto escribimos directamente el nombre de la clase Math y con el punto seleccionamos el método que más nos interesa, en este caso pow. Necesitamos elevar lado a la potencia de 2.

Ya tenemos calculada el área que se mostrará después. Ahora vamos con el caso de que sea un triángulo. Podemos ver el resultado del código en la captura que tenemos a nuestra izquierda.

En el caso del círculo necesitamos el radio para poder calcular su área así que se lo pedimos al usuario. El área del círculo es PI al cuadrado. Así que de nuevo haremos uso de la clase math para el cálculo de este área.

DEMO CÓDIGO

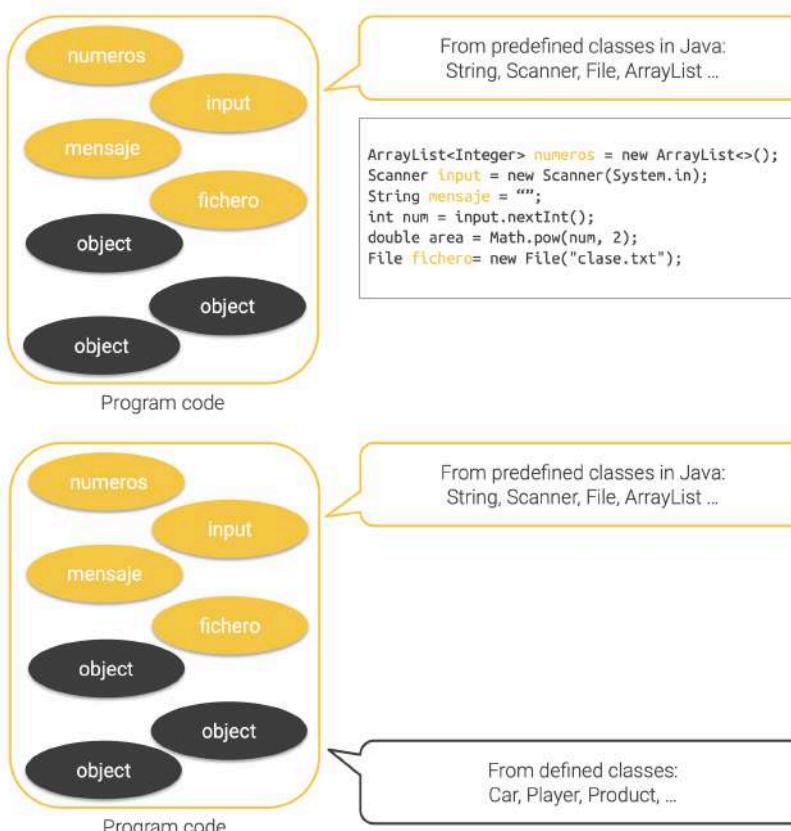
```
package com.company;  
import java.util.Scanner;  
public class Main {  
    public static void main(String[] args){  
        System.out.println("Bienvenido");  
        System.out.println("Selecciona la figura");  
        System.out.println("Cuadrado");  
        System.out.println("Triangulo");  
        System.out.println("Circulo");  
        Scanner input = new Scanner(System.in);  
        String figura = input.next();
```

```
double area = 0;
switch (figura.toLowerCase()){
    case "cuadrado":
        System.out.println("Longitud del lado (l) cm: ");
        double lado = input.nextDouble();
        area = Math.pow(lado,2);
        break;
    case "triangulo":
        System.out.println("Longitud de la base (b) cm: ");
        double base = input.nextDouble();
        System.out.println("Longitud de la altura (h) cm: ");
        double altura = input.nextDouble();
        area = base * altura /2;
        break;
    case "circulo":
        System.out.println("Longitud del radio (r) cm: ");
        double radio = input.nextDouble();
        area = Math.PI * Math.pow(radio,2);
        break;
    default:
        System.out.println("La opción seleccionada no es válida");
        System.exit(0);
}
System.out.println("El area es: " +area);
}
```

OPP INTRODUCTION

THINKING IN OBJECTS

THINKING IN OBJECTS



La programación orientada a objetos es un tipo de programación que consiste en crear aplicaciones usando unidades llamadas objetos.

En nuestro día a día como programador o programadora para desarrollar aplicaciones usando objetos podremos utilizar las clases predefinidas del lenguaje JAVA, como string, scanner, file, arraylist...

Pero también podremos definir nuevas clases, que nos permitan representar en nuestra aplicación los objetos de la vida real que necesitemos.

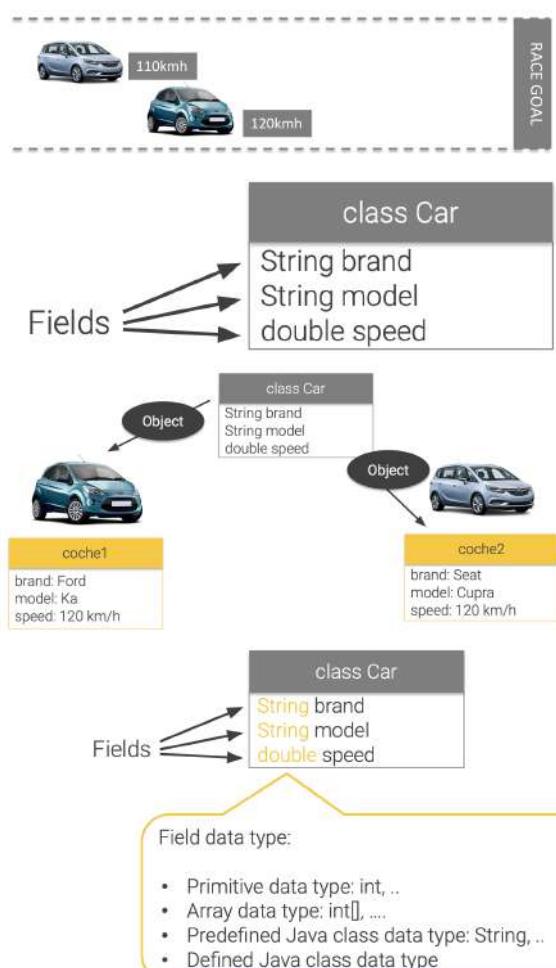
El objetivo de este tema es que aprendamos qué clases propias necesitamos definir.

Class	Object
<ul style="list-style-type: none"> • Es un tipo de dato • Define la estructura: fields, methods • CamelCase naming convention 	<ul style="list-style-type: none"> • Es una variable • Se usa para implementar la lógica • camelCase naming convention
<u>Examples</u>	
Product Car Scanner	pc jeep input

Una clase es un tipo de dato que define la estructura, esto son los atributos y métodos que tendrán los objetos creados a partir de ella.

Y los objetos disponen pues de atributos o características que registraremos como variables denominadas fields. Y pueden tener métodos que usaremos para llevar a cabo acciones sobre ellos.

- Atributos (fields): características de un objeto.
- Métodos (methods): acciones de un objeto.

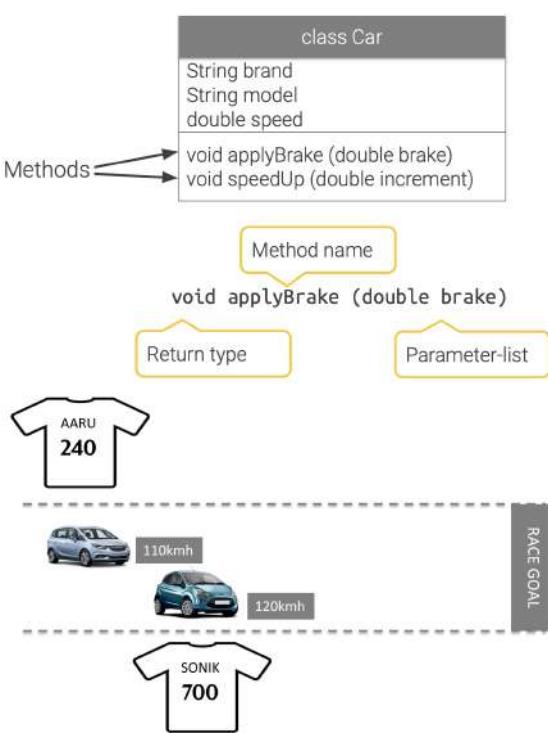


Por ejemplo, si vamos a desarrollar un juego de carreras de coches en JAVA, una buena idea sería definir la clase car para representar objetos coche de nuestra aplicación.

Los objetos coche podrían tener como características, la marca, el modelo y la velocidad.

Cada objeto que creamos del tipo car podrá tener diferentes valores para los mismos atributos, pero todos los objetos incluirán el total de atributos definidos en la clase.

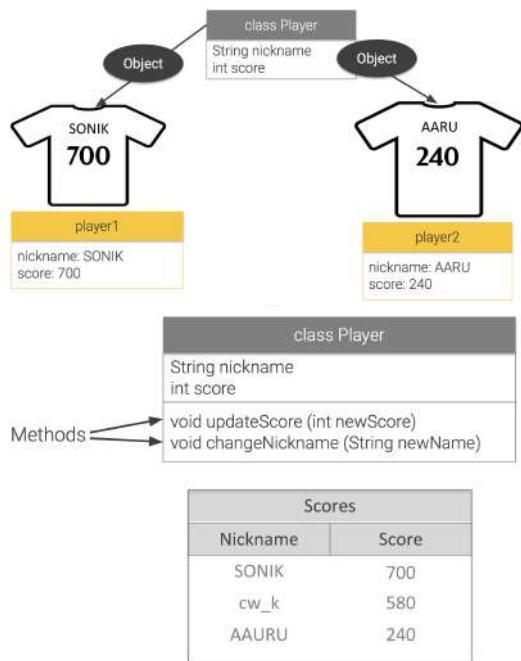
Un field o atributo es una variable por lo tanto, como cualquier variable hay que establecer cuál es su tipo de dato. Y también su identificador, que es el nombre que describe de forma simple lo que representa.



Para definir los métodos de una clase, debemos saber qué tendrá qué hacer la aplicación y por lo tanto qué se podrá hacer con cada objeto de la clase.

Por ejemplo, acciones que nos pueden interesar sobre los objetos car en un juego de carreras serían frenar y acelerar. Un método no solamente se define por un nombre, sino que es necesario el tipo que retorna, void si no devuelve nada, el nombre del método y entre paréntesis los parámetros de entrada si los tiene.

En el diseño de la aplicación del juego de carreras también podríamos necesitar representar objetos jugador, ya que cada coche puede necesitar un piloto.



La clase player o jugador tendrá como características su nickname y la puntuación máxima que ha obtenido.

Y como acciones por ejemplo, podría tener actualizar la puntuación o cambiar el nickname del jugador.

SUMMARY

Objeto: cada elemento importante se usa creando:

- Clases predefinidas JAVA. Ejemplo: string, scanner, ArrayList...
- Clases definidas propias. Ejemplo: car, player...

Fields o atributos:

- Son las características, propiedades del objeto.
- Son variables.
- Ejemplos: string nickname, int score...

Métodos:

- Son las operaciones que podemos hacer con el objeto.
- Ejemplos: void updateScore(int newScore)...

Cualquier aplicación se puede plantear como un conjunto de objetos que interactúan entre ellos para llevar a cabo tareas. En nuestro código para crear objetos podemos usar las clases predefinidas del lenguaje JAVA y también podemos crear nuestras propias clases para así representar objetos del mundo real según el escenario de nuestra aplicación.

Cada objeto tendrá unas características o propiedades que serán dadas por sus fields o atributos y dispondrá de métodos que nos permitirán realizar operaciones con el objeto.

APPLYING OPP

APPLYING OPP

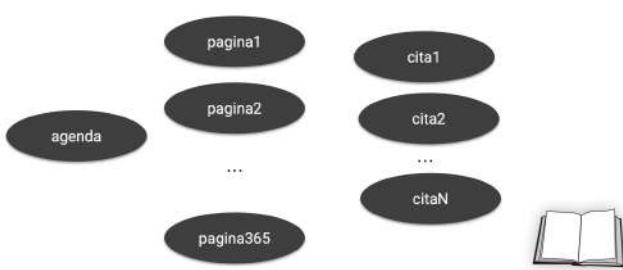
Una aplicación: Una simulación de un escenario del mundo real, donde los objetos interactúan entre ellos para llevar a cabo una tarea.

Una aplicación se puede entender como una simulación de un escenario del mundo real donde un conjunto de elementos, los objetos, interactúan entre ellos para llevar a cabo una tarea.

Vamos a ver un ejemplo de una aplicación que sirva de agenda, que permita consultar las fechas de un año concreto y permita apuntar las citas en unas horas concretas. Podemos pensar que la agenda es como un libro, en el que cada página representa un día.

Agenda: mapa de objetos

Una buena manera de descomponer el problema en objetos es a partir del elemento más general, y a partir de aquí ir extrayendo los elementos más sencillos. Así entonces, en este caso podemos partir de un objeto agenda y deducir los elementos que lo componen, las páginas.



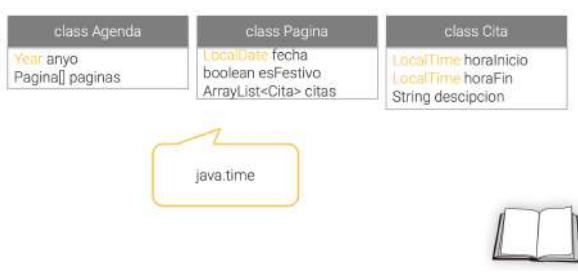
Tendremos un objeto página por cada uno de los días del año, ya que hemos decidido que la agenda sea anual así que tendríamos 365 objetos página o 366 si el año es bisiesto. Cuando se inicie la aplicación esta sería la estructura general, pero a medida que vayamos registrando citas iremos creando objetos cita en las páginas que seleccionemos.

En este punto vale la pena darnos cuenta que el número de citas es indeterminado, en contraposición del número de páginas que sabemos que es 365 o 366 si el año es bisiesto.

Agenda: clases

Una vez identificados los objetos, podemos establecer las clases sabiendo que cada tipo de objeto será una clase. En este caso es relativamente fácil ver que son 3 (agenda, página, y cita).

Agenda: clases (Fields)



Tras identificar las clases, es hora de establecer sus características.

La clase cita podría tener como atributos la hora de inicio, la hora de fin y una descripción. En la clase página la fecha, si es festivo o no y la colección de citas. Y en la clase agenda el año y la colección de páginas.

Nos fijamos ahora en el tipo de dato establecido para estos atributos. Usamos las clases predefinidas de java del paquete java.time para representar las horas de inicio y fin de una cita, la fecha de cada página y el año que representa la agenda. Usamos la clase String para la descripción de la cita, y usamos el tipo primitivo boolean para registrar la fecha de una página es festiva o no.



Para representar la página que contiene la agenda y las citas que se irán registrando en una página necesitaremos usar colecciones. Podremos usar un array estático de objetos página, ya que la cantidad de páginas no cambiará durante la ejecución de la aplicación, sino que será fijo.

Y tendremos que usar una colección dinámica para los objetos cita, como por ejemplo un arrayList, ya que el número de citas por página es indeterminado.

Agenda: clases (methods)

class Agenda	class Pagina	class Cita
Year año Pagina[] paginas crearCitaEnFecha borrarCitaEnFecha consultarCitasFecha cambiarDescripcionCita	LocalDate fecha boolean esFestivo ArrayList<Cita> citas crearCita borrarCita listarCitas	LocalTime horaInicio LocalTime horaFin String descripcion cambiarDescripcion



Para definir los métodos de cada clase tendremos que tener claro qué se podrá hacer con cada objeto. Por ejemplo, sobre un objeto del tipo agenda querremos crear una cita, borrarla, cambiar su descripción o consultar las citas que tenemos en una determinada fecha. Sobre un objeto del tipo página entonces tendremos que poder crear cita, borrarla o listar las citas que contiene. Y sobre un objeto del tipo cita tendremos que poder cambiar su descripción.

En el siguiente paso tendríamos que terminar de definir cada uno de los métodos estableciendo si devuelven algo o no, y qué parámetros de entrada son necesarios. No vamos a entrar en tanto detalle ahora ya que lo haremos más adelante.

Ejemplo pedidos supermercado

Sino que vamos a abordar otro ejemplo, éste no tan obvio como el anterior. Una aplicación que permita a un supermercado gestionar los pedidos que sus clientes realizan vía web o teléfono. El objetivo de nuevo es determinar qué objetos son necesarios y establecer las clases.



Cuando un cliente hace un pedido se registran sus datos personales y especifica las condiciones de entrega, la fecha y la dirección. En el pedido hace constar la lista de productos que quiere que le sirvan. Al final de la jornada el supermercado consulta los pedidos para servir los productos a sus clientes y organizar la ruta de los transportistas del día siguiente.

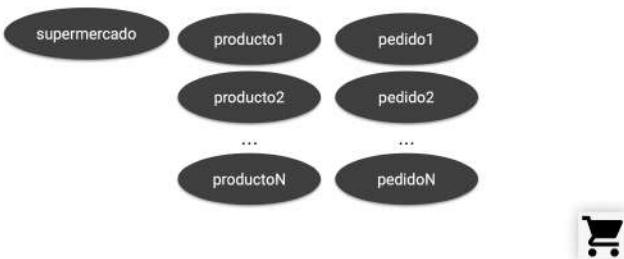
Pedidos Supermercado: descripción del escenario

Merece la pena que escribamos con el mayor detalle posible el escenario que tenemos que resolver ya que nos ayudará a plantear la solución.

Un supermercado quiere crear una aplicación que gestione los pedidos de clientes vía web o teléfono. Cuando un cliente hace un pedido, se registran sus datos personales y especifica las condiciones de entrega: día y hora y dirección.

En el pedido hace constar la lista de productos que quiere que le sirvan. Al final de la jornada, el supermercado consulta los pedidos para servir los productos a sus clientes y organizar la ruta de los transportistas del día siguiente.

Pedidos Supermercado: mapa de objetos



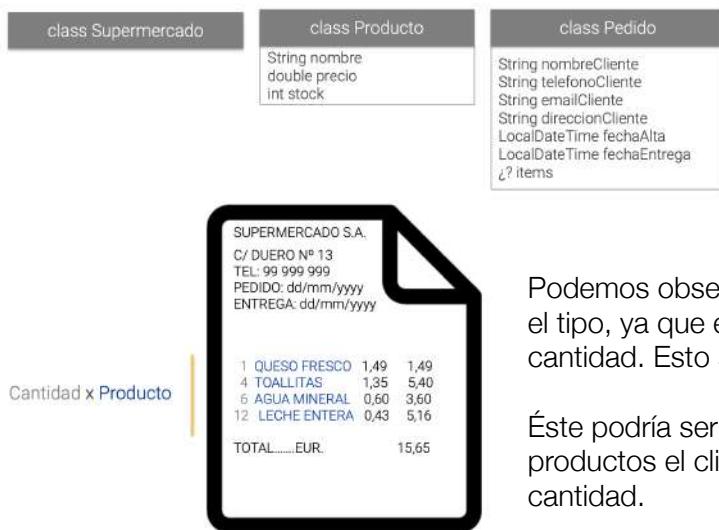
Igual que en el ejemplo anterior descompondremos el problema en objetos, partiendo del elemento más general y a partir de aquí extrayendo los elementos más sencillos. Así entonces en este caso podemos partir de un objeto supermercado que tiene productos.

Nos damos cuenta que el número de productos puede no ser fijo, ya que habitualmente un supermercado incorpora o retira productos de la venta.

Cuando se inicie la aplicación ésta sería la estructura inicial, pero a medida que vayamos necesitamos registrar pedidos iremos creando objetos pedido. De nuevo el número de pedidos es indeterminado. Podemos observar como en la descripción del escenario podemos destacar los objetos. Son los elementos más importantes y normalmente suelen ser sustantivos.

Pedidos Supermercado: clases (fields)

Una vez identificados los objetos, podemos establecer las clases. En este caso podemos plantear 3, que son supermercado, producto y pedido.



Alguno de los atributos que podríamos considerar para estas clases son, en la clase producto su nombre, precio y stock. En la clase pedido los datos personales y dirección del cliente, las fechas de alta y entrega del pedido, y la colección de productos que ha seleccionado el cliente.

Podemos observar que en este atributo es difícil establecer el tipo, ya que el cliente habrá indicado el producto y la cantidad. Esto son 2 cosas.

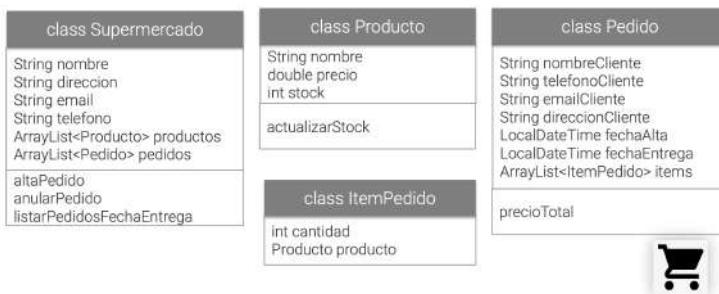
Éste podría ser un detalle del pedido. En la lista de productos el cliente selecciona para cada producto una cantidad.

Podremos resolver esto definiendo una clase auxiliar que denominemos ItemPedido, que tendrá como características la cantidad y el producto. Así en la clase pedido podremos disponer de una colección dinámica por ejemplo, un arrayList de objetos ItemPedido para registrar la lista de productos seleccionada por el cliente.

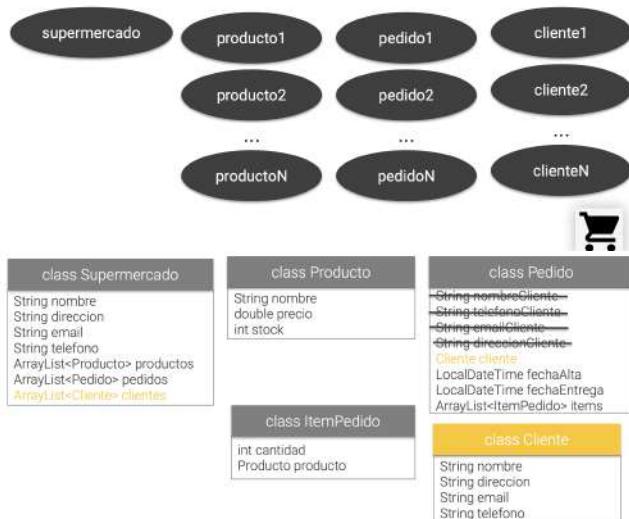
Vamos ahora con la clase supermercado, sus fields o atributos podrían ser sus datos de identificación como nombre dirección etc, así como las colecciones dinámicas de productos y pedidos.

Pedidos Supermercado: clases (methods)

¿Qué acciones queremos realizar sobre cada objeto? Sobre el objeto supermercado dar de alta o anular un pedido, así como listar los pedidos de una determinada fecha de entrega. De un objeto producto queremos actualizar stock para decremento las unidades disponibles en función de los pedidos de los clientes. Y de un objeto pedido queremos calcular el precio total a pagar.



Estas serían las operaciones fundamentales aunque un análisis más minucioso nos podría llevar a ver qué otras operaciones habituales de la aplicación como podrían ser, por ejemplo, cambiar la dirección de un pedido, o cambiar la fecha de entrega de un pedido, y por lo tanto sería necesario definir nuevos métodos.



Es difícil definir a la primera todas las clases atributos y métodos. Normalmente tendremos que hacer varias iteraciones. Además es importante que entendamos que un problema puede tener soluciones diferentes.

Así, por ejemplo, podríamos considerar qué otro elemento importante como objeto son los clientes, añadiríamos pues los objetos cliente a nuestro mapa de objetos y definiríamos la clase cliente, cuyos atributos serían los datos personales y la dirección. La clase pedido tendría entonces como atributo un objeto del tipo cliente, y en la clase supermercado podríamos añadir una colección de clientes.

SUMMARY

Entonces, para resolver una aplicación utilizando objetos podemos seguir los siguientes pasos.

- Plantear el escenario. Cuanto más detallada sea la descripción más fácil será resolver el problema.
- Localizar dentro de la descripción del escenario los elementos que se consideren más importantes, los que realmente interactúen. Éstos serán los objetos de la aplicación, normalmente suelen ser sustantivos.
- Definir las clases agrupando los diferentes objetos según el tipo. Cada tipo de objeto será una clase.
- También considerar qué objetos son de una cierta complejidad y redefinirlos como agrupaciones de objetos más simples como es el caso de ItemPedido.
- Identificar las características de los objetos de cada clase. Cuáles son sus propiedades, fields o atributos y qué operaciones tienen que ofrecer los métodos.
- Para después a partir de estos especificar formalmente sus atributos y métodos.

Debemos recordar que para un problema de cierta complejidad es muy difícil identificar a la primera todas las clases, atributos y operaciones. Normalmente tendremos que hacer varias iteraciones.

DEFINING CLASSES

DECLARING A CLASS

```

public class MyClass {

    // field declarations
    // method implementations

}

```

Para definir una clase usaremos la sintaxis que vemos en la captura. Public seguido de la palabra clave class seguida de nombre de la clase, en el ejemplo MyClass. Entre las llaves incluiremos el código que permita declarar sus atributos e implementar sus métodos. Los constructores son un tipo especial de método que nos permitirán crear objetos de la clase.

```
public class MyClass {  
  
    // field declarations  
    // constructors  
    // method implementations  
}
```

1
2
3

Predefined classes structure:

- Field Summary
- Constructor Summary
- Method Summary

1
2
3

<https://docs.oracle.com/...>



Siguiendo la línea de cómo nos presenta Oracle en su documentación la estructura de las clases predefinidas del lenguaje JAVA.

CLASS NAMING CONVENTIONS

```
public class MyClass {  
  
}
```

Class names should be nouns.
Try to keep names simple and descriptive

Class names examples:

class Car
class Player
class Producto
class Pedido

El identificador que pongamos a cada una de nuestras clases debería ser un sustantivo, como por ejemplo, car, player, producto, pedido... el idioma a utilizar, castellano, inglés...

Lo escogemos nosotros, o a veces nos vendrá impuesto por la empresa para la que desarrollamos la aplicación, eso sí siempre tenemos que intentar que el nombre que seleccionemos describa de forma simple lo que representa.

Usaremos la regla UpperCamelCase donde la primera letra es siempre mayúscula, y también en mayúscula la primera letra de las distintas palabras que formen el nombre de la clase.

De esta forma seguimos la misma linea que ha establecido Oracle para sus clases predefinidas.

Predefined class names examples:

String
Scanner
Math

Defined class names examples:

Producto
Car
Cliente

Aplicar esta forma de identificar nuestras clases hará que el código de nuestros programas sean más fácil de leer y por lo tanto de entender y mantener.

JAVA FILE

The image shows two separate windows displaying Java code. The left window contains the code for `MyClass.java`:

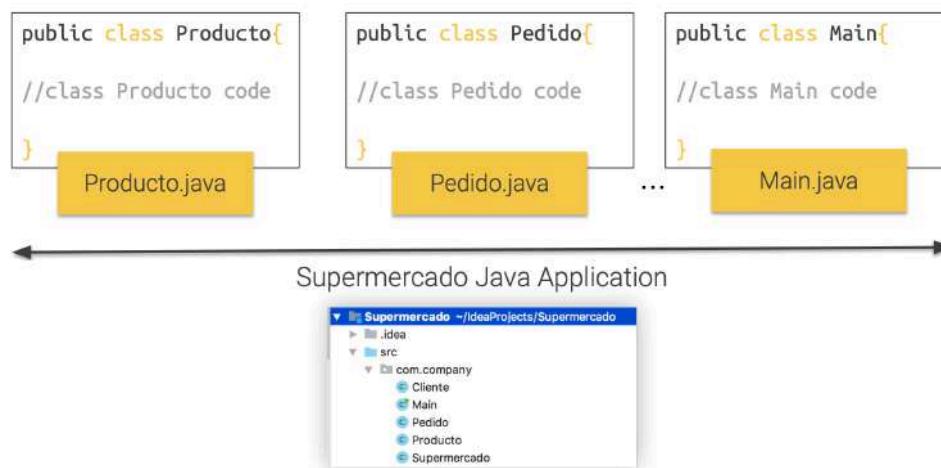
```
public class MyClass {  
    // field declarations  
    // constructors  
    // method implementations  
}
```

The right window contains the code for `Producto.java`:

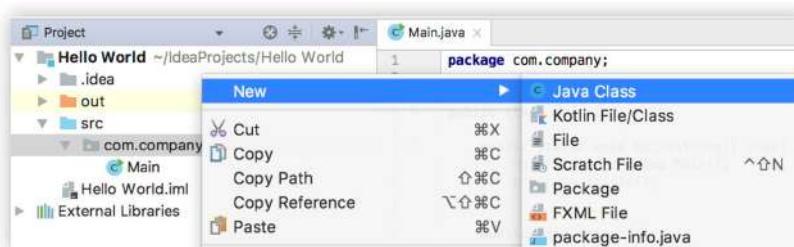
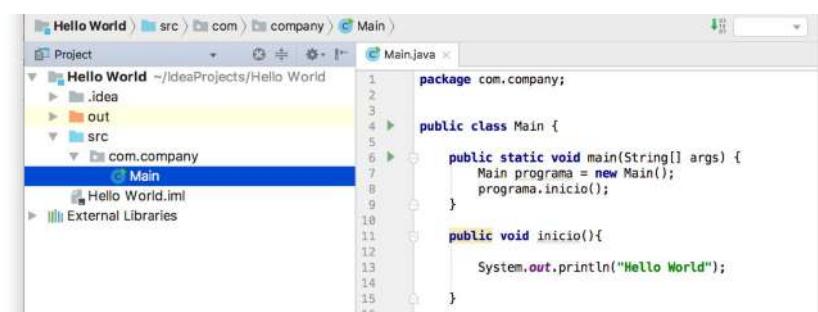
```
public class Producto{  
    //field declarations  
    String nombre;  
    int stock;  
    double precio;  
  
    //constructors  
    public Producto(String nombre, int stock, double precio){  
        this.nombre = nombre;  
        this.stock = stock;  
        this.precio = precio;  
    }  
  
    //method implementations  
    public void actualizarStock(int cantidad){  
        this.stock=this.stock-cantidad;  
    }  
}
```

En JAVA cada clase debería ser creada en un fichero independiente, este fichero tendrá por nombre, el nombre de la clase y la extensión .java así el fichero producto.java contendrá el código de la clase producto.

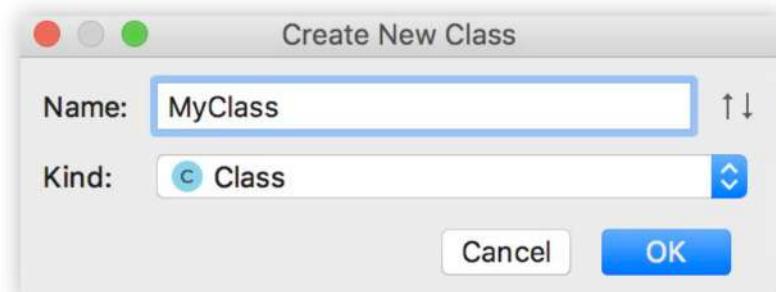
Una aplicación típica en JAVA no es más que un conjunto de clases interactuando unas con otras, de hecho no puede existir código JAVA fuera de una clase.



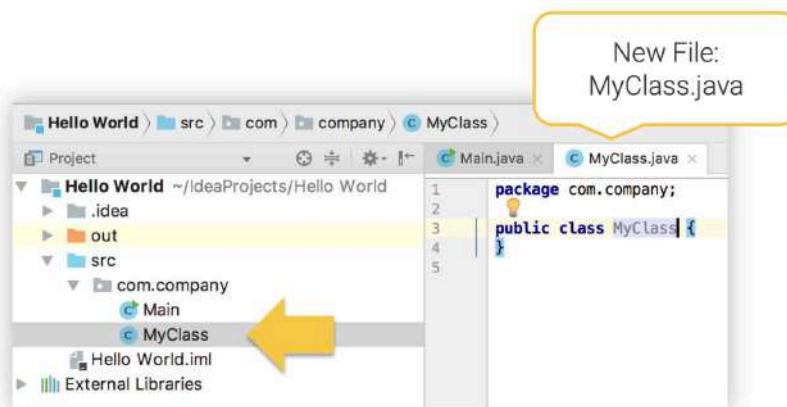
CREATING A CLASS IN INTELLIJ



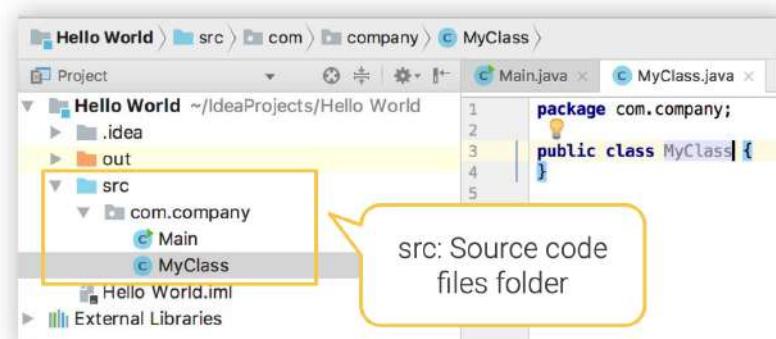
Para añadir una clase a cualquier Java Application Project en IntelliJ pulsamos con el botón derecho de ratón sobre el paquete `com.company` (sitio donde queremos crear la clase) y del menú que aparece seleccionaremos New Java Class.



Introduciremos el nombre de nuestra clase y pulsaremos ok.

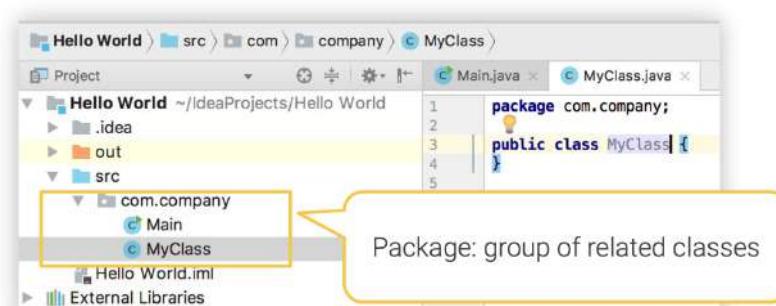


Y ya tenemos nuestra clase creada dentro de nuestro proyecto, debemos comprobar que se ha creado un fichero nuevo con el nombre especificado para la clase `MyClass.java`.

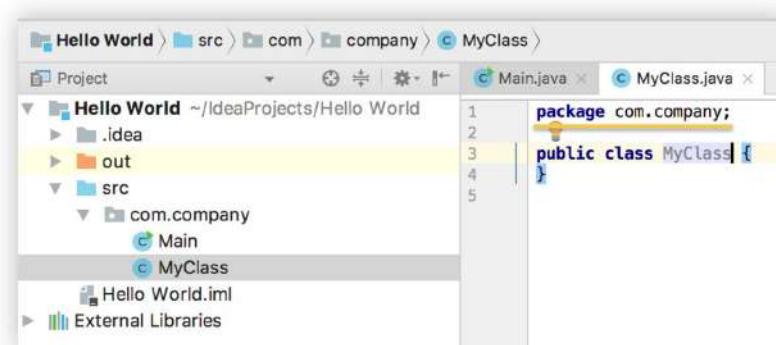


Nos debemos fijar ahora en dónde se ha creado esta clase, dentro del proyecto, pues concretamente dentro de la carpeta `src`.

El código fuente de nuestra aplicación siempre irá dentro de este directorio.

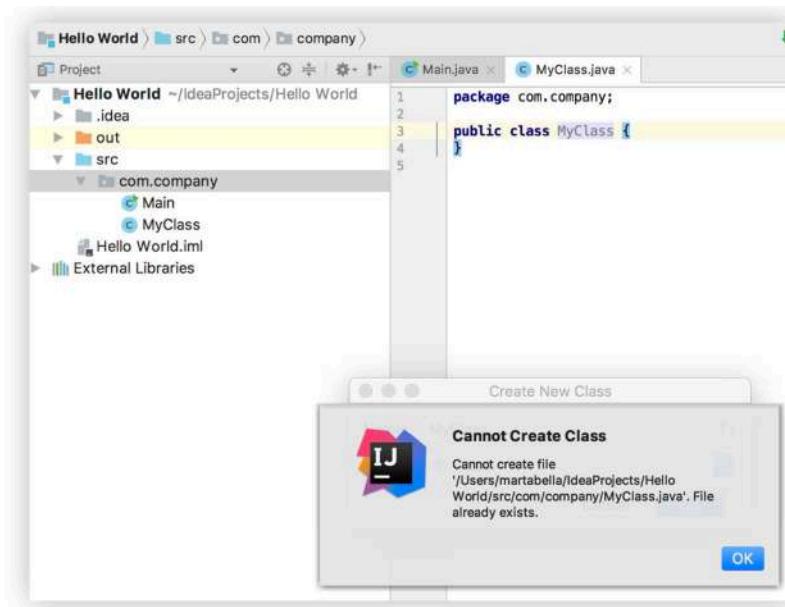


Dentro de la carpeta `src` pueden existir subcarpetas, cada subcarpeta se denomina package o paquete. Un paquete permite agrupar ficheros de código fuente que se relacionan entre sí.



En el ejemplo que vemos en la captura hay un único paquete denominado `com.company` que contiene dos ficheros, que son `Main.java` y `MyClass.java`.

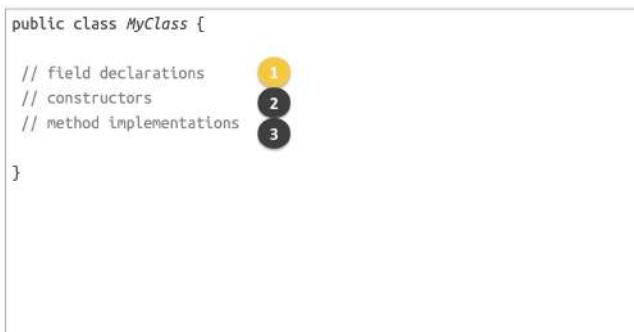
Debemos comprobar que la primera línea de código de cada uno de los ficheros con código fuente de un paquete es la sentencia `package com.company;` que permite indicar al ordenador que el código fuente del fichero pertenece a este paquete.



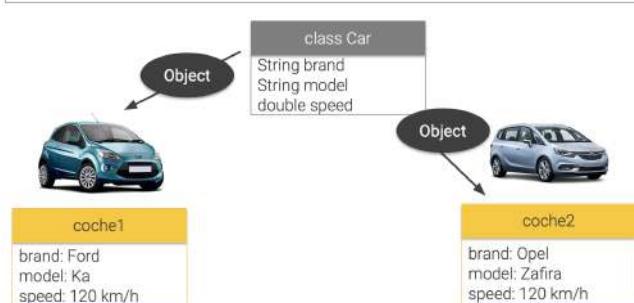
Debemos comentar también que dentro de un mismo paquete no pueden existir dos clases que tengan un mismo nombre.

FIELDS AND STARTING WITH CONSTRUCTORS

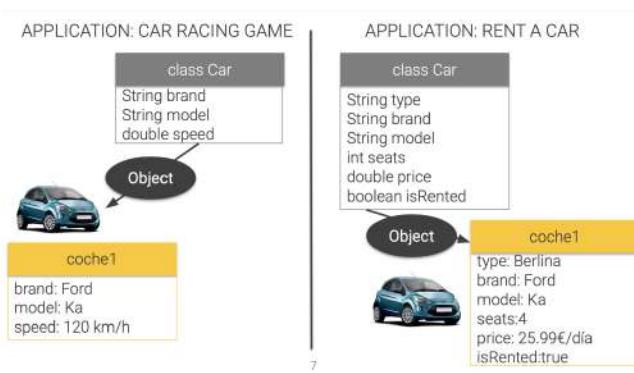
DECLARING FIELDS



Cualquier clase que definamos en nuestras aplicaciones JAVA podrá tener unos fields o atributos además de constructores y métodos. Al inicio del código de clase declararemos sus fields o atributos.



Los fields o atributos de un clase son las características del objeto, así por ejemplo una clase car podría tener como características, la marca, el modelo y la velocidad, y cada objeto que creemos del tipo car, podrá tener diferentes valores para estos atributos.



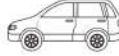
Las características del objeto dependen del escenario a resolver, así si estamos creando un juego de carreras de coches las características de los objetos del tipo car serían diferentes a las características de los objetos car de una aplicación de alquiler de coches.

Por lo tanto en cada aplicación la clase car tendría un número y un tipo de atributos diferente para registrar las propiedades de los objetos coche.

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors
    //method implementations
}
```

Field declarations = Variable declarations



```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;
}
```

Field declarations = Variable declarations

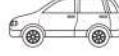
Data type



```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;
}
```

Field declarations = Variable declarations

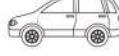
Identifier



```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;
}
```

Field declarations = Variable declarations

lower camel Case



Los fields o atributos son variables y por lo tanto usaremos la sintaxis ya conocida. El tipo de dato seguido del identificador de la variable y punto y coma. Debemos recordar que para identificar cada variable usamos la regla lowerCamelCase, primera letra en minúscula.

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors
    //method implementations
}
```

Primitive variables

Object variables



Los tipos de variables en fields o atributos de una clase pueden ser primitivos u objetos. Por ejemplo la clase car contiene el atributo speed que de tipo primitivo double, y atributos que son objetos como brand y model.

```
public class Producto{
    //field declarations
    String nombre;
    double precio;
    int stock;
}
```

Primitive variables

Object variables



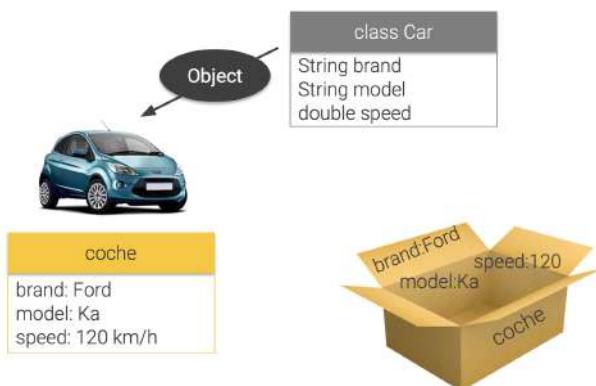
O por ejemplo, la clase producto que vemos en la captura contiene los atributos de tipo primitivo como precio y stock y atributos que son objetos como nombre.

```
public class Supermercado{
    //field declarations
    ArrayList<Producto> productos;
    String nombre;
    String direccion;
    String telefono;
    String email;
}
```




O por ejemplo, la clase supermercado. Un supermercado tiene una colección de productos por esto contiene un atributo productos de tipo array de objetos producto, siendo producto el tipo de objeto que hemos definido previamente.

CONSTRUCTORS



DEFAULT CONSTRUCTOR

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```

No constructor implementations:
Default constructor is available

Un constructor es un tipo especial de método que se encarga de crear e inicializar un objeto de una clase. El trabajo de un constructor es crear objetos de la clase y asignar valores a cada uno de sus atributos.

Cuando declaramos una clase podemos definir uno o más constructores o bien no definir ninguno. Entonces, JAVA automáticamente nos proporcionará un constructor por defecto que nos permitirá crear objetos de esta clase.

Creating objects with Default constructor

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```

Car coche = new Car();

Para entender qué es y cómo usar un método constructor en una clase JAVA usaremos la clase car que vemos en la captura que contiene una serie de fields o atributos y métodos, pero ningún constructor.

Entonces la clase car dispone del constructor por defecto que automáticamente genera JAVA.

¿Cómo creamos entonces un objeto del tipo car? Sabemos por nuestra experiencia con las clases predefinidas del lenguaje, que para usar un constructor tenemos que usar la palabra clave new con el constructor por defecto que no es más que el nombre de la clase y paréntesis vacíos, una lleva argumentos o parámetros de entrada.

Esta linea de código crea un objeto denominado coche de tipo car y le asigna a sus atributos los valores por defecto, null para la marca y el modelo, 0 para la velocidad, y quizás en este punto nos preguntemos por qué estos valores y no otros.

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```

Variable declarations
Declaration is not to declare "value" to a variable;
it's to declare the type of the variable

Bien, cuando simplemente declaramos una variable indicamos su tipo (string, int, double...) pero no su valor. Entonces, cuando declaras una variable JAVA le asigna el valor por defecto según su tipo.

Data Type	Descripción	Valor por defecto
long	Número entero	0
int		0
short		0
byte		0
double	Número decimal	0
float		0

Data Types	Descripción	Valor por defecto
boolean	valor lógico	false
char	carácter unicode	nul
String	cadena de caracteres	null
XXX	Cualquier objeto	null

El valor por defecto de tipos de datos primitivos numéricos es 0. El valor por defecto del tipo de dato boolean es false, nul para el tipo de dato char, y null con objetos tanto predefinidos del lenguaje como string, como los creados por nosotros.

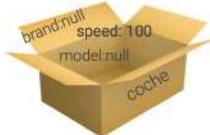
```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed=100;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```



Car coche = new Car();



Bien, si en la declaración de los atributos decidimos inicializar algún valor, por ejemplo 100 en el atributo velocidad. Cuando usemos el constructor por defecto para crear el objeto coche éste será el valor que se almacene en el este atributo y no, el de por defecto.

ACCESSING OBJECT FIELDS

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```



Car jeep = new Car();
jeep.speed



Para acceder al valor de cualquier atributo de un objeto, usamos el identificador del objeto seguido de punto y le nombre del atributo.

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```



Car jeep = new Car();
double n = jeep.speed



```
public class Car{
    //Field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```



```
Car jeep = new Car();  
System.out.println(jeep.speed);
```

Esto nos devuelve un valor de tipo double que podemos recoger en una variable, o usarla directamente en la llamada a un método, como por ejemplo System.out.println.

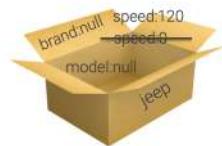
```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```



```
Car jeep = new Car();
jeep.speed = 120;
```



```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```



```
Car jeep = new Car();
jeep.speed = "Fast";
```



```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
}
```



```
Car jeep;
jeep.speed = 120;
System.out.println(jeep.speed);
```

Compilation error:
Variable 'jeep' might not have been
initialized

Para cambiar el valor de cualquier atributo de un objeto usamos el operador de asignación =, como si se tratara de una variable.

Por ejemplo, si hemos declarado o inicializado una variable jeep de tipo car, actualizaremos su velocidad escribiendo jeep.speed = y el valor que queramos asignar.

Igual que ocurre con cualquier otra variable, el valor que asignemos debe coincidir con el tipo de dato usado en su declaración en caso contrario se producirá un error de compilación.

Un error muy común es intentar acceder a un atributo de un objeto cuando éste no ha sido inicializado. Esto provocará un error de compilación.

CONSTRUCTORS

DEFINED CONSTRUCTORS

```
public class MyClass {
    // field declarations
    // constructors
    // method implementations
}
```



```
public class ClassName{
    //field declarations

    //constructors
    public ClassName(){
        //constructor block of code
        //Attributes initialization
    }

    public ClassName(parameters){
        //constructor block of code
        //Attributes initialization
    }

    //method implementations
}
```

Constructor without
input parameters

Constructor with input
parameters

Cuando declaramos una clase, si no definimos constructores, JAVA nos asigna el constructor por defecto que nos permite que creamos objetos de la clase. Pero lo habitual es que definamos constructores propios para así poder inicializar objetos dando los valores que queramos a sus atributos. Definiremos los constructores de una clase después de declarar sus atributos y antes de implementar sus métodos.

En el código que ves en la captura, hemos definido dos constructores propios para la clase denominada ClassName. Un constructor es un tipo especial de método responsable de la creación e inicialización de un objeto de la clase. Como cualquier método, los métodos constructores de una clase pueden tener o no parámetros de entrada.

La sintaxis es la que ves en la captura, modificador de acceso public seguido del nombre de la clase y entre paréntesis la lista de parámetros si tiene o paréntesis vacíos si no los tiene.

```
public class ClassName{
    //Field declarations

    //constructors
    public ClassName(){
        //constructor block of code
        //Attributes initialization
    }

    public ClassName(parameters){
        //constructor block of code
        //Attributes initialization
    }

    //method implementations
}
```

Constructors:

- Have the same name as the public class itself
- Don't have any return types

Debemos comprobar que el nombre del método constructor debe coincidir exactamente con el nombre de la clase y que nunca en un método constructor declaramos el tipo que retorna. En el bloque de código de cada método constructor incluiremos las sentencias de código que permitan inicializar los valores de los atributos de la clase de acuerdo a un determinado escenario.

Car racing game

```
public class Car{
    String brand;
    String model;
    double speed;
}
```



Por ejemplo, tenemos la clase CAR que tiene los atributos marca, modelo y velocidad. Cuando en la aplicación de carreras de coches tengamos que dar de alta un objeto del tipo car por ejemplo un Ford Ka o un Opel Zafira, nos darán su marca y modelo. Se supone que cuando creamos un nuevo objeto CAR no ha comenzado la carrera, así que su velocidad es de 0 km hora.

```
public class Car {
    String brand;
    String model;
    double speed;
}
```



Por tanto, no es necesario que sea un dato de entrada del constructor porque tendrá un valor de 0 para cualquier coche nuevo.

```
public class Car {
    String brand;
    String model;
    double speed;

    public Car(String b, String m) {
        //constructor block of code
        //Attributes initialization
    }
}
```



Bien, entonces necesitamos definir un constructor que tenga como datos de entrada la marca y el modelo.

```
public class Car {
    String brand;
    String model;
    double speed;

    public Car(String b, String m) {
        brand = b;
        model = m;
        speed = 0;
    }
}
```

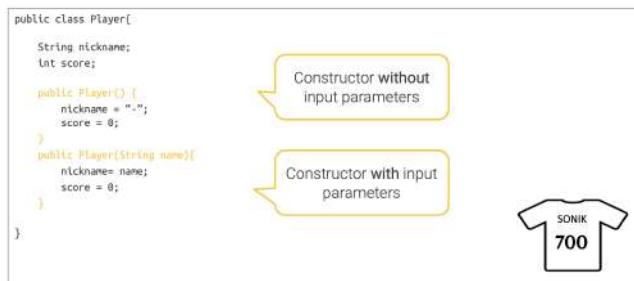
**Constructor block of code:
Attributes initialization**



En la captura podemos ver la definición de este constructor para la clase CAR. Escribimos public seguido del nombre de la clase y entre paréntesis los parámetros de entrada que hemos establecido en nuestro análisis. Dos strings, una para la marca y otra para el modelo.

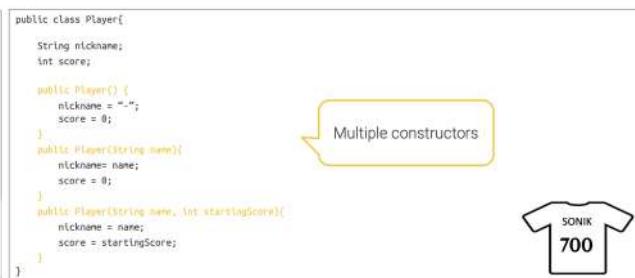
En el bloque de código de un constructor usaremos las sentencias de código necesarias para inicializar con algún valor los atributos de la clase. En el código que vemos en la captura se inicializan los atributos marca y modelo a partir de los datos de entrada del método y el atributo velocidad se inicializa con un valor por defecto de 0.

Bien, entonces en la clase Car hemos definido un constructor propio que nos permitirá crear objetos dando valores a todas sus características o atributos.



Por tanto, queremos proporcionar dos formas de inicializar un objeto player sin saber su nickname o sabiéndolo. En ambos casos la puntuación será de cero ya que todavía no ha comenzado la carrera.

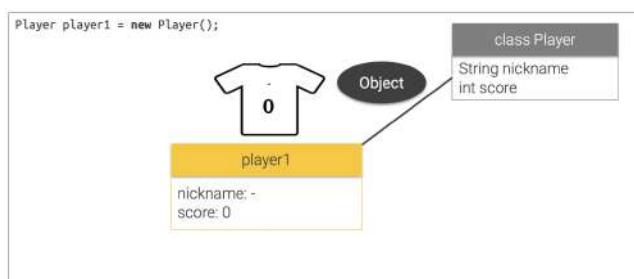
Por este motivo declaramos dos constructores en esta clase. Un constructor sin parámetros donde el atributo nickname se inicialice con un guión y el atributo score se inicialice a cero. Y un constructor con parámetros donde el atributo nickname se inicialice con el valor que tenga el parámetro de entrada y el atributo score se inicialice también a cero.



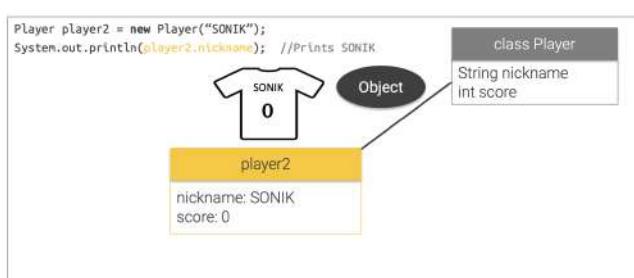
O incluso podemos añadir otro constructor para contemplar el caso de que se necesite crear un objeto jugador con un nickname dado y una puntuación inicial. Debemos comprobar que la clase Player tiene más de un constructor para permitir crear objetos de esta clase en los distintos escenarios que la aplicación necesita.

CREATE AN OBJECT WITH A CONSTRUCTOR

Car racing name



Cada vez que nuestra aplicación queramos crear un objeto de una clase, tendremos que llamar a algún constructor de la clase. En la captura podemos ver cómo, usando la palabra clave new seguida del constructor que queremos usar, por ejemplo, new Player seguido de paréntesis vacíos. Esta instrucción creará un objeto del tipo player llamado player1 usando el constructor sin parámetros de la clase Player.



Esto hará que el valor del atributo Nickname sea guión y el del atributo Score sea de cero.

Para crear un jugador que tenga un valor inicial para su atributo Nickname usaremos el segundo constructor de la clase tal y como vemos en la captura. De nuevo el operador New, seguido del constructor y entre paréntesis los parámetros de entrada.

Esta instrucción creará un objeto del tipo player llamado Player2 usando el constructor con parámetros de la clase Player, dando el valor de Sonic a su atributo NickName y de 0 a su atributo Score.

```
Player player3; //player1 is null
System.out.println(player3);
```

Compilation error:
Variable 'player3' might not have
been initialized

```
Car car1 = new Car("Ford","Ka");
```



Object

class Car
String brand
String model
double speed

car1

brand: Ford
model: Ka
speed: 0 km/h

Una vez que un objeto se ha inicializado usando el operador New, podremos acceder a sus atributos usando la técnica del punto tal y como vimos. Pero si no inicializamos un objeto usando la palabra clave new, entonces su valor será de null, que significa sin valor.

Y si intentamos acceder a un atributo de un objeto null, se producirá un error de compilación.

De forma equivalente, en nuestra aplicación del juego de carreras de coches, Para dar de alta un nuevo coche usaremos la palabra clave NEW seguida del constructor y entre paréntesis los parámetros de entrada. Esta instrucción que ves en la captura creará un objeto llamado car1 donde sus atributos tendrán el valor especificado en los argumentos del constructor.

DEFAULT VS DEFINED CONSTRUCTORS

Car racing name

```
public class Car {
    String brand;
    String model;
    double speed;
}
```

Default constructor available

Car myCar= new Car();



```
public class Car {
    String brand;
    String model;
    double speed;

    public Car(String b, String m) {
        brand = b;
        model = m;
        speed = 0;
    }
}
```

No default constructor available



```
public class Car {
    String brand;
    String model;
    double speed;

    public Car() {
        brand = "default";
        model = "default";
        speed = 0;
    }

    public Car(String b, String m) {
        brand = b;
        model = m;
        speed = 0;
    }
}
```

Default constructor available



Hemos dicho antes que JAVA dispone de un constructor por defecto para cada clase. Este privilegio se tiene mientras en esa clase no hayamos creado un constructor propio.

Así, la clase Car que ves en la captura donde se ha definido un constructor propio no tendrá el constructor por defecto.

Si intentamos crear un objeto de la clase Car usando el constructor por defecto, se producirá un error de compilación.

Si queremos disponer del constructor por defecto, tendremos que crearlo.

```
Car myCar= new Car("Ford", "Ka");
Car anotherCar = new Car();
```



Podemos observar como ahora la clase CAR dispone de dos métodos constructores, que podremos usar para crear distintos objetos de la clase tal y como ves en la captura.

Es lo que ha ocurrido en la clase Player, donde hemos definido el constructor por defecto y varios constructores con parámetros para permitir crear objetos de esta clase en los distintos escenarios que la aplicación necesitaba.

```
public class Player{
    String nickname;
    int score;

    public Player() {
        nickname = "-.-";
        score = 0;
    }

    public Player(String name) {
        nickname = name;
        score = 0;
    }

    public Player(String name, int startingScore) {
        nickname = name;
        score = startingScore;
    }
}
```



```
Player player1 = new Player();
Player player2 = new Player("SONIK");
```



THIS KEYWORD

Using this with a field

```
public class Car {
    String brand;
    String model;
    double speed;

    public Car(String b, String m) {
        brand = b;
        model = m;
        speed = 0;
    }
}
```

Constructor block of code:
Attributes initialization



```
public class Car {
    String brand;
    String model;
    double speed;

    public Car(String brand, String model) {
        this.brand = brand;
        this.model = model;
        this.speed = 0;
    }
}
```

this keyword



```
public class Player{
    String nickname;
    int score;

    public Player() {
        this.nickname = "-.-";
        this.score = 0;
    }

    public Player(String nickname){
        this.nickname= nickname;
        this.score = 0;
    }
}
```



```
public class Player{
    String nickname;
    int score;

    public Player() {
        this("-.-");
    }

    public Player(String nickname){
        this.nickname= nickname;
        this.score = 0;
    }
}
```



Hemos dicho que en el bloque de código de un constructor usaremos las sentencias de código necesarias para inicializar con algún valor los atributos de la clase. En un constructor con parámetros de entrada como el que ves en la captura, estos se usan para inicializar los atributos.

Un código más legible para cualquier constructor sería el que ves en la captura, donde los parámetros de entrada tienen el mismo nombre que los atributos de la clase.

Esto implica que tengamos que usar la palabra clave this dentro del bloque de código del método constructor, para separar entre atributos y parámetros de entrada, tal y como ves en la captura.

La palabra this también podemos usarla para llamar desde un constructor a otro de la clase.

Por ejemplo, en la clase Player que tiene estos dos constructores que permiten inicializar sus atributos con diferentes valores, podemos modificar el código del primer constructor llamando al segundo constructor con la palabra clave this. Entre paréntesis indicamos el valor del parámetro de entrada nickname, en este caso, guión.

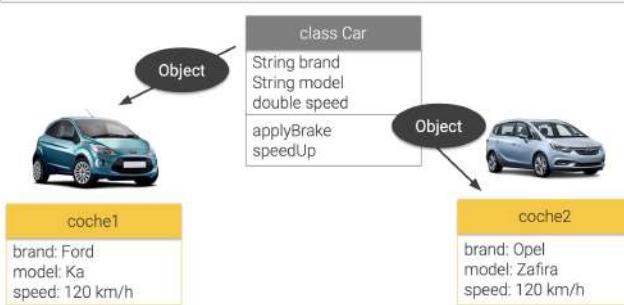
METHODS

METHOD IMPLEMENTATIONS

```
public class MyClass {
    // field declarations
    // constructors
    // method implementations
}
```

- 1
- 2
- 3

Cualquier clase que definamos en nuestras aplicaciones Java podrá tener unos métodos, además de atributos y constructores, si los tiene declararemos e implementaremos después de los constructores. Los métodos permiten que realicemos acciones sobre los objetos de la clase.



APPLICATION: CAR RACING GAME

class Car
String brand
String model
double speed
applyBrake
speedUp

APPLICATION: RENT A CAR

class Car
String type
String brand
String model
int seats
double price
boolean isRented
rentPrice
setRented

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;

    //constructors

    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```

Method implementations



Los métodos del objeto dependen del escenario a resolver. Así, si estamos creando un juego de carreras de coches, las acciones a realizar sobre los objetos del tipo CAR serían diferentes de las acciones de los objetos CAR en una aplicación destinada a alquiler de coches.

Por tanto, en cada aplicación, la clase CAR tendría unos métodos diferentes para permitir acciones específicas del escenario sobre los objetos coche.

Los métodos son funciones y, por tanto, usaremos la sintaxis ya conocida. Modificador de acceso, habitualmente public. Tipo de dato que devuelve. Usamos la palabra clave VOID si el método no devuelve ningún dato. Recuerda que si el método retorna algo en su bloque de código debe existir la sentencia return para devolver este algo.

El nombre del método usando la regla lowerCamelCase y paréntesis vacíos si el método no tiene datos de entrada o con los parámetros que sirven de datos de entrada separados por comas. Entre los paréntesis introduciremos el bloque del código, esto es el conjunto de instrucciones necesarias para llevar a cabo la tarea planificada.

Access modifier:

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;
    //constructors
    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```

Return type:

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;
    //constructors
    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```

Method name (lower Camel Case Rule):

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;
    //constructors
    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```

Parameters (parameter1, parameter2, ...):

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;
    //constructors
    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```

Method block of code:

```
public class Car{
    //field declarations
    String brand;
    String model;
    double speed;
    //constructors
    //method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }
    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```

ACCESSING OBJECT METHODS

Para llamar a cualquier método de un objeto, primero hay que crear el objeto. En la captura hemos creado el objeto Jeep de tipo Car haciendo uso del constructor por defecto de la clase. Para usar cualquier método de un objeto creado por nosotros, usamos la misma sintaxis que la que hemos usado con objetos predefinidos por el lenguaje, como por ejemplo el método length de la clase String.

Create an object:

```
Car jeep = new Car();
```

Object State:

```
brand:null
model:null
speed:0
jeep
```

Method Call:

```
Car jeep = new Car();
jeep.speedUp(100);
```

Object State after Method Call:

```
brand:null
model:null
speed:100
jeep
```

Si hemos declarado e inicializado una variable jeep de tipo char, llamaremos a cualquiera de sus métodos escribiendo el identificador del objeto jeep, seguido de punto, el nombre del método y entre paréntesis el valor de los parámetros de entrada del método si los tiene. En este ejemplo estamos llamando al método speedUp. Cuando llamamos a un método se ejecuta el bloque de código del método. En este caso el bloque de código del método speedUp hace que el atributo speed se incremente según el valor proporcionado como argumento de entrada.

```
public class Car {
    //Field declarations
    String brand;
    String model;
    double speed;

    //Constructors
    //Method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }

    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```



```
Car jeep = new Car();
jeep.speedUp(100);
jeep.speedUp(20);
jeep.applyBrake(50);
```



```
public class Car {
    //Field declarations
    String brand;
    String model;
    double speed;

    //Constructors
    //Method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }

    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```



```
Car jeep = new Car();
double currentSpeed = jeep.speedUp(100);
currentSpeed = jeep.speedUp(20);
currentSpeed = jeep.applyBrake(50);
System.out.println(currentSpeed);
```



```
public class Car {
    //Field declarations
    String brand;
    String model;
    double speed;

    //Constructors
    //Method implementations
    public double applyBrake(double decrement) {
        speed -= decrement;
        return speed;
    }

    public double speedUp(double increment) {
        speed += increment;
        return speed;
    }
}
```



```
Car jeep; //jeep is null
double currentSpeed = jeep.speedUp(100);
```

Compilation error:
Variable 'jeep' might not have been
initialized

Debemos recordar que la definición e implementación de un método se realiza una vez, pero que en nuestro programa podremos hacer tantas llamadas como necesitemos a un método.

Si el método retorna algo, como en este ejemplo que ambos métodos retornan un valor double, podemos almacenarlo en alguna variable para usarlo, por ejemplo, para mostrarlo por consola.

Recuerda que si no inicializamos un objeto usando la palabra clave new, entonces su valor será null, que significa sin valor. Si intentamos acceder a un método de un objeto null, se producirá un error de compilación.

OVERLOADING METHODS

El lenguaje de programación JAVA permite sobrecargar métodos. ¿Qué significa esto? Pues que dentro de una misma clase pueden existir varios métodos que tengan el mismo nombre si tienen distintos parámetros de entrada. Supongamos que tenemos una clase artista para dibujar varios tipos de datos, strings, enteros, etcétera, y que contiene un método para dibujar cada tipo de dato.

Podríamos usar un nuevo nombre de método para cada uno, por ejemplo drawString, drawInteger, drawDouble...

```
public class Artist {
    ...
    public void draw(String s) { ... }
    public void draw(int i) { ... }
    public void draw(double f) { ... }
}
```

Overloaded methods are differentiated by the number and the type of the parameters

Pero para simplificar nuestro código, en Java podemos usar el mismo nombre para todos los métodos draw, pero pasando a cada método una lista de parámetros diferente.

Así, la clase artista tendría tres métodos llamados draw, cada uno con una lista de parámetros diferente. En esta situación decimos que el método draw está sobrecargado. Debemos comprobar cómo cada método draw se diferencia de los otros por el tipo y número de parámetros de entrada.

```

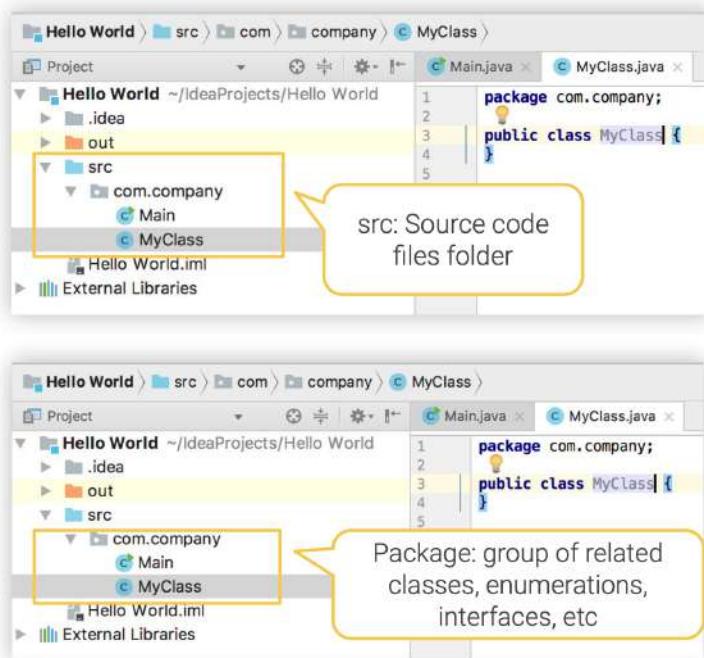
public class Artist {
    ...
    public void draw(String s) { ... }
    public boolean draw(String s) { ... }
    public void draw(int i) { ... }
    public void draw(double f) { ... }
}

```

Tenemos que saber que no es posible declarar más de un método que tenga el mismo nombre y los mismos parámetros de entrada. Esto provoca un error de compilación. Debemos observar que el tipo que retorna el método no sirve para diferenciar entre métodos. En definitiva, no podemos declarar dos métodos que se llamen igual y que tengan los mismos parámetros de entrada aunque el tipo que retornen sea diferente.

PACKAGES

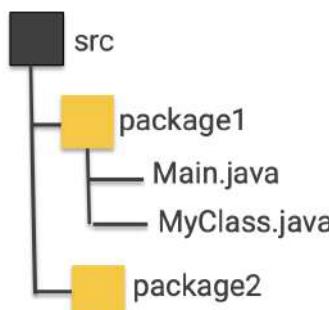
PACKAGES



Dentro de la carpeta src de nuestros proyectos JAVA sabemos que es donde tenemos el código fuente de la aplicación.

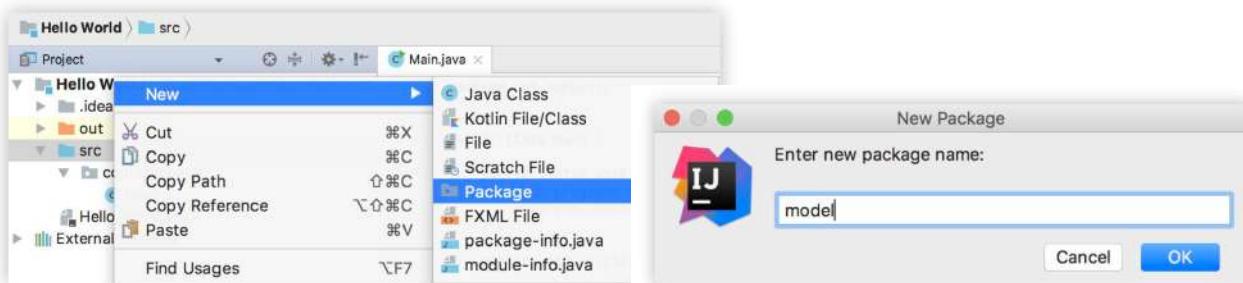
En esta carpeta pueden existir subcarpetas. Cada subcarpeta se denomina package o paquete. Un paquete permite agrupar ficheros de código fuente que se relacionan entre sí.

En la captura podemos ver un proyecto denominado Hello World, donde hay un único paquete denominado com.company, que contiene dos ficheros con código fuente, main.java y myclass.java.



Crearemos paquetes en nuestras aplicaciones para agrupar código que se relacione entre sí y así tenerlo más ordenado. Esto nos permitirá realizar cambios más rápidamente. Otra ventaja de estructurar nuestro proyecto usando paquetes es controlar el acceso a contenido, pero esto lo veremos más adelante con los modificadores de acceso.

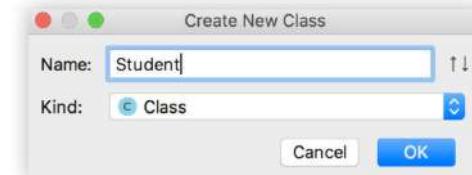
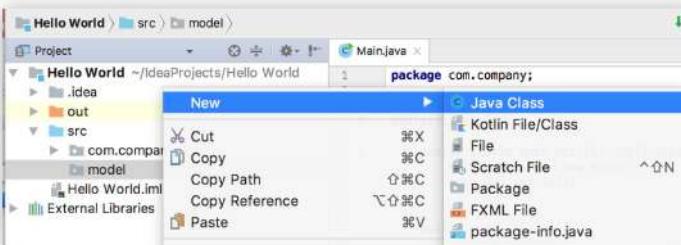
Creating a package



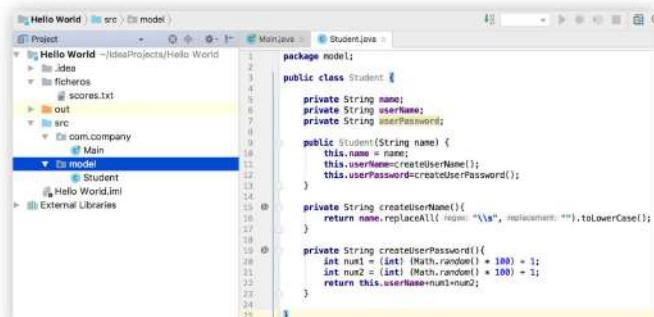
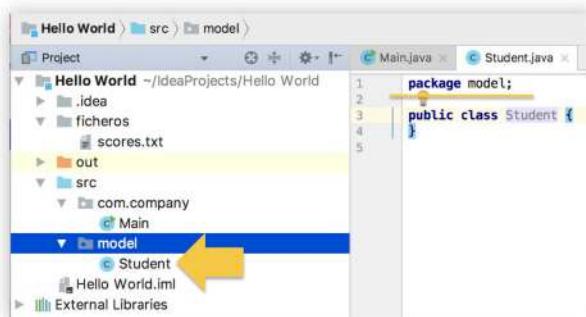
Para crear un paquete en IntelliJ, pulsamos con el botón derecho del ratón sobre la carpeta SRC, sitio donde podemos crear el paquete, y del menú que aparezca, seleccionaremos new package, introduciremos el nombre del paquete, por ejemplo model, y pulsaremos ok.



Y ya tendremos creado nuestro paquete Model dentro del proyecto.



Para crear ficheros de código dentro de un paquete, pulsamos con el botón derecho del ratón sobre el paquete, Model en este caso, y del menú que aparezca seleccionaremos New Java Class. Introduciremos el nombre de la clase y pulsaremos OK.



Con esto tenemos la clase Student creada dentro del paquete Model en el proyecto y por tanto podremos editar su código fuente según los requisitos de la aplicación.

```

package com.company;

public class Main {
    public static void main(String[] args) {
        Main programa = new Main();
        programa.inicio();
    }

    public void inicio(){
        Student estudiante = new Student("ElenaRoigPas");
    }
}

```

```

package com.company;

import model.Student;

public class Main {
    public static void main(String[] args) {
        Main programa = new Main();
        programa.inicio();
    }

    public void inicio(){
        Student estudiante = new Student("ElenaRoigPas");
    }
}

```

Cuando usemos esta clase fuera del paquete al que pertenece, por ejemplo desde la clase Main que se encuentra en el paquete com.company, IntelliJ nos marca que existe un error de compilación, destacando en rojo el nombre de la clase.

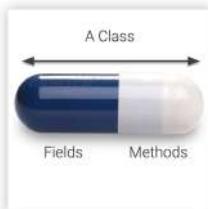
Si situamos el ratón sobre el nombre de la clase, IntelliJ nos muestra el detalle del error.

No sabe quién es student. ¿Qué tenemos que hacer? Pues es fácil usar la sentencia Import que vemos en la captura.

Import seguido del nombre del paquete, punto y el nombre de la clase. Esta sentencia permitirá usar objetos de la clase Student del paquete Model dentro de la clase Main. Debemos comprobar que la sentencia Import se sitúa antes de la declaración Public Class Main.

ENCAPSULATION INTRODUCTION

ENCAPSULATION



Uno de los principios básicos en la programación orientada a objetos para crear programas de calidad es la encapsulación, que significa que cada clase debería ser como una cápsula que contiene todo lo que necesita y nada más, y que cualquiera que la use no tiene por qué conocer su composición interna, sino simplemente usarla.

A class

Una idea que nos ayudará a la hora de aplicar este concepto al diseño de nuestros programas es pensar que cada clase es como una caja negra. Tenemos que pensar que cualquier otro programador o programadora que la vaya a usar no tiene por qué saber los detalles de su estructura interna.

Podemos hacer la equivalencia entre una clase y un cajero automático. La persona que usa el cajero automático se beneficia de él para, por ejemplo, sacar dinero, pero no debe saber cómo funciona internamente. Cuando definamos una clase solo hemos demostrar que pueden hacer sus objetos, pero no como lo hacen ni mediante qué datos.

Information hiding

Entonces conseguiremos aplicar encapsulación cuando ocluitemos información al exterior de nuestra clase. Es decir, definimos la clase de forma que los objetos creados a partir de ella mantengan su estado como privado, esto es que otros objetos no tengan acceso directo a sus atributos, a menos que explícitamente se lo permitamos mediante los métodos llamados getter o setter. Y cuando otros objetos solamente puedan llamar a los métodos públicos de la clase.

Java access modifiers

- private
 - no modifier (package private)
 - protected
 - public
-
- ```

 graph TD
 A[+ RESTRICTIVE] --> B["private"]
 B --> C["no modifier (package private)"]
 C --> D["protected"]
 D --> E["public"]
 E --> F[- RESTRICTIVE]

```

Java dispone de cuatro modificadores de acceso que podemos usar al definir la visibilidad de nuestras clases, de sus métodos y atributos. Cada 1 de ellos tiene un nivel de accesibilidad, solo podremos usar un modificador para cada clase, método o atributo. Cuál usar, pues la regla es siempre el más el más restrictivo posible.

En la captura podemos ver los modificadores de acceso de Java por orden del más restrictivo Private, al menos Public. Vamos a ver a través de un ejemplo, cada 1 de ellos y cuándo usarlos.

## School learning platform

**User.java**

```

public class User{
 String name;
 //more attributes like address, phone, etc
 String userName;
 String userPassword;

}

```

**User.java**

```

public class User{
 String name;
 String userName;
 String userPassword;

 User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 String createUserName(){ //block of code }

 String createUserPassword(){ //block of code }

 String display(){ //block of code }
}

```

Supongamos que tenemos que desarrollar una aplicación que permita la gestión de usuarios, alumnos, profesores de un colegio. Un usuario, bien sea alumno o profesor, tendrá como característica sus datos personales, nombre, apellidos, DNI, dirección... así como las credenciales de acceso a la aplicación, nombre de usuario y contraseña.

Definimos la clase user que permita gestionar los datos de cada usuario.

Por simplificar, tendrá como atributos un nombre y una cuenta para acceder a la plataforma de gestión con nombre de usuario y contraseña, en la vida real tendría más datos como dirección, teléfono, DNI...

**User.java**

```

public class User{
 String name;
 String userName;
 String userPassword;

 User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 String createUserName(){ //block of code }

 String createUserPassword(){ //block of code }

 String display(){ //block of code }
}

```

Cuando en nuestra aplicación creamos un usuario necesitaremos su nombre y apellidos a partir de estos, la aplicación generará automáticamente su nombre de usuario y contraseña según unas directrices que veremos luego. Por este motivo definimos un método constructor que tenga como dato de entrada el nombre.

```

public class User{
 String name;
 String userName;
 String userPassword;

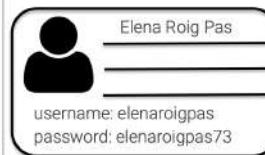
 User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 String createUserName(){
 return name.replaceAll("\\s","");
 }

 String createUserPassword(){
 int num1 = (int) (Math.random() * 100) + 1;
 int num2 = (int) (Math.random() * 100) + 1;
 return this.userName+num1+num2;
 }

 String display(){ //block of code }
}

```



Usaremos el método `createUserName` para crear el nombre del usuario en la aplicación. Este método permite crear una string a partir del nombre y apellidos del usuario, eliminando los espacios en blanco y convirtiendo todo en minúsculas.

Y usaremos el método `create password` para crear la contraseña del usuario calculando dos números aleatorios comprendidos entre 1 y 100 y concatenándolos al nombre de usuario.

## CLASS ACCESS MODIFIERS

School learning platform

```

public class User{
 String name;
 String userName;
 String userPassword;

 User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 String createUserName(){
 return name.replaceAll("\\s","");
 }

 String createUserPassword(){
 int num1 = (int) (Math.random() * 100) + 1;
 int num2 = (int) (Math.random() * 100) + 1;
 return this.userName+num1+num2;
 }

 String display(){ //block of code }
}

```

```

class User{
 String name;
 String userName;
 String userPassword;

 User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 String createUserName(){
 return name.replaceAll("\\s","");
 }

 String createUserPassword(){
 int num1 = (int) (Math.random() * 100) + 1;
 int num2 = (int) (Math.random() * 100) + 1;
 return this.userName+num1+num2;
 }

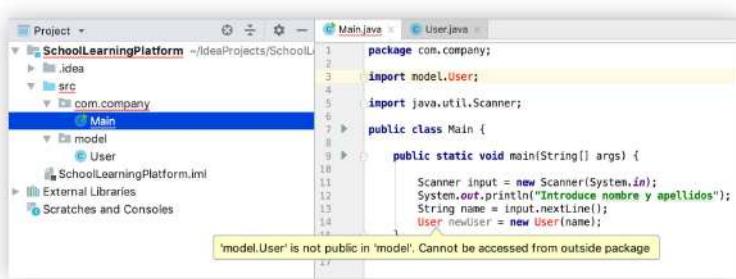
 String display(){ //block of code }
}

```

Debemos comprobar que en el Código desarrollado no hemos utilizado el modificador de acceso `Public`, como viene siendo habitual. Tan solo hemos usado `Public` en la declaración de la clase, que significa que podremos usarla desde cualquier otra clase de la aplicación. Si delante de la declaración de una clase no usamos modificador de acceso, significará que solamente podremos usarla desde otras clases que estén en el mismo paquete.

Lo podemos ver en la captura. En este proyecto, en el paquete `com.company` tenemos la clase `main` y la clase `user`. En la declaración de la clase `user` no hemos usado el modificador de acceso `Public`.

Si desde la clase `main` intentamos usar la clase `user`, no habrá problema, ya que es visible por estar en el mismo paquete.



Pero si las clases main y user están en paquetes distintos, como puedes ver en la captura correspondiente, la clase user al no tener modificador de acceso Public no es visible desde la clase main y al intentar usarla desde la clase main tendremos un error de compilación.

Entonces a nivel de clase, en Java tenemos dos posibles modificadores de acceso o Public o sin modificador de acceso. Si en la declaración de una clase no usamos el modificador de acceso Public, significará que la clase solo podrá ser usada por clases que están en su mismo paquete.

En cambio, si usamos el modificador de acceso Public podrá ser usada por todas las clases que estén dentro del mismo paquete o no. Por este motivo, habitualmente una clase se declara como Public.

## ENCAPSULATION: CONTROLLING ACCESS TO MEMBERS OF A CLASS

### ATTRIBUTES AND METHODS ACCESS MODIFIERS

Java dispone de cuatro modificadores de acceso para definir la visibilidad o no de los métodos y atributos de una clase. Si usamos el modificador de acceso Private delante de un atributo o método de una clase, sólo se podrá acceder a él dentro del código de la clase.

Por tanto, para lograr encapsulación, es el modificador de acceso que escogeremos para los atributos de nuestras clases así como los métodos internos que no queremos que sean llamados desde otras clases.

Cuando no proporcionemos un modificador de acceso a un atributo o método de una clase, significará que podemos acceder a él desde el código de la clase pero también desde las clases que se encuentran en el mismo paquete.

Esto es por lo que habitualmente se conoce a este modificador de acceso como Package Private o paquete privado. Lo usaremos si en nuestras aplicaciones definimos paquetes para controlar la visibilidad de los datos.

Por el contrario, el modificador de acceso public delante de un atributo o método significará que se puede acceder a él dentro del código de la clase pero también desde cualquier otra clase. Por tanto, si queremos lograr encapsulación es el modificador de acceso que escogeremos para los métodos externos que queremos que sean llamados desde otras clases como son los métodos constructores.

El modificador de acceso protected delante de un atributo o método significará que se puede acceder a él dentro de la clase, pero también desde las clases que estén en el mismo paquete y por todas las subclases.

El concepto de subclase todavía no lo conocemos ya que va asociado a herencia, que veremos más adelante. Por este motivo, no analizaremos por el momento el modificador de acceso Protected.

## School learning platform

The diagram illustrates three versions of the User.java class:

- User.java (Top):** Shows private attributes (name, userName, userPassword) and private methods (User constructor, createUserName, createUserPassword, display). A user interface shows a placeholder for a profile picture and two empty text fields labeled "Elena Roig Pas". Below it, the text "username: elenaroigpas" and "password: elenaroigpas73" is displayed.
- User.java (Middle):** Shows the same structure but with the User constructor and display method declared as public. The user interface remains the same.
- User.java (Bottom):** Shows the same structure but with the User constructor and display method declared as public. In addition, a Main.java file is shown with the following code:
 

```
User newUser = new User("Elena Roig Pas");
newUser.name = "Maria Elena Roig Pas";
String userPassword= newUser.createUserPassword();
```

 A callout box points to the line "String userPassword= newUser.createUserPassword();" with the text "Adding a setter method". A separate callout box below says "Error compilation".

## GETTER AND SETTER METHODS

Es bastante habitual que tras haber declarado un atributo como privado nos interese acceder a él para actualizar su valor o para leerlo. En estas situaciones definiremos el método setter para actualizar su valor y el método getter para leer su valor.

## A generic example

The diagram shows the MyClass.java code:

```
public class MyClass {
 private String myAttribute;

 public MyClass(String myAttribute) {
 this.myAttribute = myAttribute;
 }

 public void setMyAttribute(String myAttribute) {
 this.myAttribute = myAttribute;
 }
}
```

A callout box points to the line "public void setMyAttribute(String myAttribute) {" with the text "Adding a setter method".

Volviendo a la clase User, para lograr una correcta encapsulación de su información, declararíamos como Private todos sus atributos y los métodos de uso interno que no queremos que sean llamados desde otras clases, como son en este caso el método CreateUserName y el método CreateUserPassword.

Y declararemos como Public los métodos que queramos que sean accesibles desde otras clases.

En este caso, el método constructor y el método display que retorna un string con el valor de cada uno de los atributos de la clase.

Así, si desde la clase Main creamos un atributo del tipo User usando su método constructor, podemos porque hemos declarado este método como public, pero si intentamos acceder a un atributo o método declarado como private, esto dará un error de compilación.

Vamos a verlo. Dado el atributo myAttribute de la clase MyClass, definimos un método set escribiendo set y el nombre del atributo. Entre paréntesis situaremos el nuevo valor para el atributo, y en su bloque de código actualizaremos este valor. Un método set habitualmente no devuelve nada por esto, en su declaración indicamos void para el tipo que retorna.

```

public class MyClass {
 private String myAttribute;

 public MyClass(String myAttribute) {
 this.myAttribute = myAttribute;
 }

 public void setMyAttribute(String myAttribute) {
 this.myAttribute = myAttribute;
 }

 public String getMyAttribute() {
 return myAttribute;
 }
}

```

MyClass.java

```

public class MyClass {
 private String myAttribute;

 public MyClass(String myAttribute) {
 this.myAttribute = myAttribute;
 }

 public void setMyAttribute(String myAttribute) {
 this.myAttribute = myAttribute;
 }

 public String getMyAttribute() {
 return myAttribute;
 }
}

```

MyClass.java

Adding a getter method

myAttribute has a setter and  
getter method

Así si desde la clase Main creamos un objeto. Del tipo MyClass podremos usar el método getter y el método Setter del atributo denominado MyAttribute para leer y actualizar su valor respectivamente.

```

public class MyClass {
 private String myAttribute;

 public MyClass(String myAttribute) {
 this.myAttribute = myAttribute;
 }

 public void setMyAttribute(String myAttribute) {
 this.myAttribute = myAttribute;
 }

 public String getMyAttribute() {
 return myAttribute;
 }
}

```

MyClass.java

```

MyClass object = new MyClass("Name");
object.setMyAttribute("AnotherName");
System.out.println(object.getMyAttribute());

```

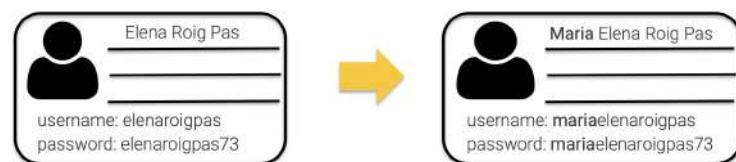
Main.java

Debemos tener en cuenta que si para todos los atributos terminamos declarando sus Getters y Setters, es decir, terminamos permitiendo que otras clases puedan leer y modificar su valor, ya no existiría encapsulación. Así que debemos usar getters y setters de forma racional y según el escenario.

### IMPORTANTE

The presence of numerous getter and setter methods is a red flag that the program isn't necessarily well designed == isn't well encapsulated information

School learning platform



**User.java**

```
public class User{
 private String name;
 private String userName;
 private String userPassword;

 public User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 public void setName(String name){
 this.name = name;
 }
}
```

**User.java**

```
public class User{
 private String name;
 private String userName;
 private String userPassword;

 public User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 public void setName(String name){
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }
}
```

**User.java**

```
public class User{
 private String name;
 private String userName;
 private String userPassword;

 public User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 public void setName(String name){
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 public String getName(){
 return this.name;
 }
}
```

**Adding a setter method**

**Adding a getter method**

**Main.java**

```
User newUser = new User("Elena Roig Pas");
newUser.setName("Maria Elena Perez Roig");
```

Vamos a ver el uso de métodos getters y setters en la aplicación de gestión de usuarios de la plataforma educativa que venimos trabajando. Para ello, suponemos que tras haber dado de alta un usuario, existe la posibilidad de habernos equivocado y es necesario actualizar su nombre o apellidos.

Como el nombre de usuario y contraseña están relacionados con este, también habrá que actualizarlos.

Necesitamos pues disponer de un método público que permita actualizar el atributo name. Definimos entonces el método setter del atributo name.

En esta aplicación, un cambio en el atributo name hace que también tengamos que cambiar el nombre de usuario y contraseña, ya que están relacionados con éste.

Por tanto, en el bloque de código del método setName tendremos que añadir el código que vemos en la captura.

Debemos comprobar que se pueden dar casos como este, donde el bloque de código de un método set no tiene por qué existir una única línea para actualizar el valor del atributo, sino que pueden existir más líneas.

Así, desde la clase Main, una vez creado un objeto del tipo User, podremos cambiar el valor del atributo name y con él el de userName y password.

## SUMMARY

Accesibility matrix

| Modifier    | Class | Package | Subclass | Other Classes |
|-------------|-------|---------|----------|---------------|
| Private     | Yes   | No      | No       | No            |
| No modifier | Yes   | Yes     | No       | No            |
| Protected   | Yes   | Yes     | Yes      | No            |
| Public      | Yes   | Yes     | Yes      | Yes           |

Aquí podemos ver el resumen de los diferentes modificadores de acceso que existen en JAVA y cómo afectan al acceso de atributos y métodos de una clase.

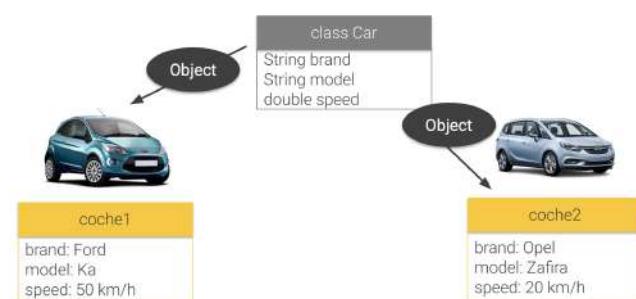
Por ejemplo, si usamos el modificador de acceso private en un atributo o método, significa que permitimos que se acceda a él desde el código de la misma clase pero no desde otras clase, sean que están en el mismo paquete o no. o por ejemplo, si usamos el modificador de acceso public en un atributo o método, significa que permitimos el acceso desde el código de la misma clase y desde otras clases que están en el mismo paquete o no.

Podemos entonces resumir que encapsular correctamente una clase consistiría en declarar como private los atributos de la clase, y definir los métodos públicos necesarios para acceder a sus valores métodos GETTER o modificar los métodos SETTER, definir como public los constructores de la clase para permitir crear objetos de la clase, y declarar como public los métodos de la clase que queramos que sean accesibles desde otras.

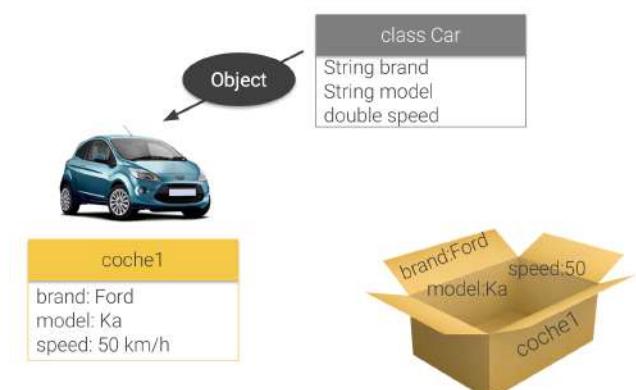
Declararemos como private los métodos auxiliares de forma que no sean accesibles desde otras clases.

## STATIC FIELDS AND METHODS

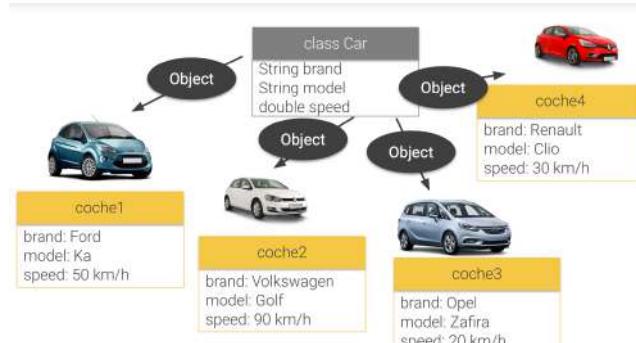
### OBJECT'S LIFE-TIME



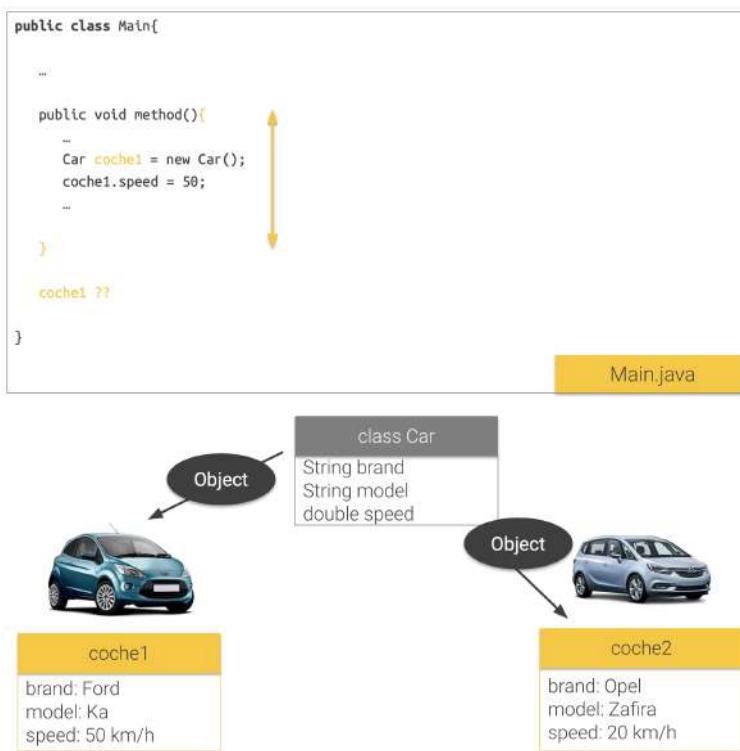
Los objetos que creamos a partir de una clase no son eternos, no viven para siempre.



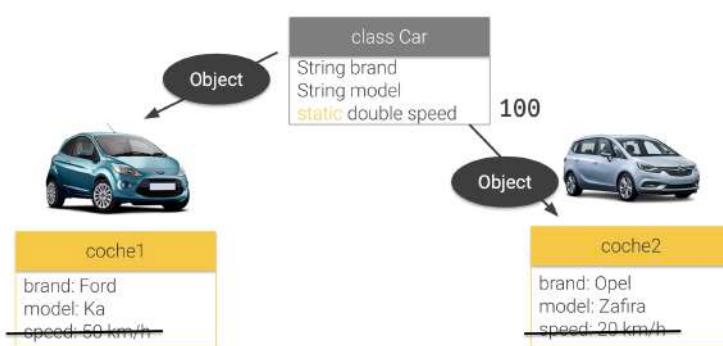
Normalmente creamos un objeto de la clase asignando valores a cada uno de sus fields u atributos.



En la aplicación podremos crear tantos objetos de la clase como sean necesarios. Cada uno podrá tener unos valores diferentes para los fields o atributos de la clase.



## STATIC FIELDS



Pero en algún momento todos estos objetos se destruirán, incluyendo los valores almacenados en cada uno de sus fields o atributos. Esto ocurrirá cuando el scope o ámbito de uso de estos objetos termine.

Por ejemplo, aquí, dentro de este método, hemos creado el objeto Coche1 del tipo Car. Una vez que el método termina, la variable Coche1 no existirá más, incluyendo todos los valores de todos sus fields o atributos.

Esto tiene lógica, ya que si no podemos acceder al objeto, no tiene sentido que los valores de sus fields o atributos sigan existiendo.

Sin embargo, en algunas situaciones podemos querer mantener el valor de un field incluso cuando no haya objetos creados de la clase.

En estos casos necesitamos añadir la palabra clave static delante de la declaración del atributo. Declarar un atributo como estático significa que sus valores no se guardan dentro del objeto, sino que lo hacen dentro de la clase.

Y cada objeto que se cree de la clase tendrá el mismo valor. Y cuando cada objeto de la clase se destruya, el valor del atributo estático se mantendrá dentro de la clase.



```
public class Main{
 ...
 public void method(){
 ...
 Car coche1 = new Car();
 coche1.speed = 50;
 ...
 }
}
```

Main.java

Entonces, como los atributos estáticos pertenecen a las clases y no a los objetos, la sintaxis Java permite acceder a los atributos estáticos directamente desde la clase, en lugar de tener que crear un objeto.

```
public class Main{
 ...
 public void method(){
 ...
 Car.speed = 50;
 ...
 }
}
```

Main.java

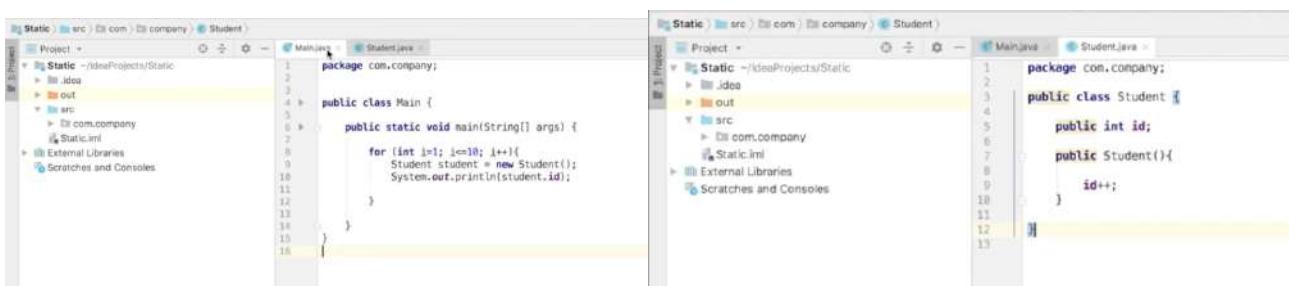
Por ejemplo aquí, accedemos directamente al atributo estático velocidad directamente desde la clase para poder cambiar su valor.

```
public class Main{
 ...
 public void method(){
 ...
 Car coche1 = new Car("Ford","Ka");
 Car coche2 = new Car("Opel","Zafira");
 Car.speed = 50;
 System.out.println(coche1.speed); //Prints 50
 System.out.println(coche2.speed); //Prints 50
 ...
 }
}
```

Main.java

Esto hará que todos los objetos del tipo car que se creen tengan este valor. Vamos a ver un ejemplo con IntelliJ.

En este ejemplo hemos creado la clase Student y queremos incrementar la variable ID de cada objeto que se cree de la clase Student. Así que para esto tenemos el atributo id como public y dentro del constructor de la clase incrementamos su valor en 1 lo que significa que cada vez que creamos un objeto nuevo de student se sumará 1 a esta variable id.



Si ahora vamos al método main, podemos ver que hemos creado un bucle que comienza en 1 y se ejecuta 10 veces y en su bloque de código creamos un objeto student y mostramos el valor de su variable id.

Así que cuando se ejecute este código esperamos que cada vez que se cree un objeto student se añada uno a la variable id y cuando lleguemos a crear el estudiante 10 el valor de id sea de 10. Si

ejecutamos el código vemos que el atributo ID vemos que no sucede lo esperado, sino que obtenemos todo 1s. La explicación por la que sucede esto es simple, si volvemos a la clase student, vemos que el atributo id no lleva la palabra clave Static.

Esto significa que este atributo pertenece al objeto y no a la clase. Así que cada vez que se crea un objeto del tipo Student, se crea esta variable, se inicializa a cero y en el método constructor se incrementa en uno. Por tanto, cada objeto del tipo Student tiene una variable id cuyo valor vale siempre 1. En cambio, si declaramos este atributo como static, significará que ya no pertenecerá al objeto, sino que pertenecerá a la clase.

Y cada vez que llamemos al constructor se añadirá 1 a la misma variable id. Si nos vamos ahora a ejecutar de nuevo el código, obtenemos distintos valores del atributo ID del 1 al 10 como esperábamos. Contar los objetos que se crean desde la clase es uno de los usos más comunes de usar static en un atributo.

## STATIC METHODS

```
public class Calculator{
 public static int add(int a, int b){
 return a + b;
 }
 public static int subtract(int a, int b){
 return a - b;
 }
}
```

Calculator.java

```
public class Calculator{
 public static int add(int a, int b){
 return a + b;
 }
 public static int subtract(int a, int b){
 return a - b;
 }
}
```

```
int r = Calculator.add(3,4);
```

Igual que los atributos estáticos, los métodos estáticos pertenecen a la clase y no al objeto. Se definen métodos estáticos cuando tenemos que crear un método que no necesite acceder a ningún atributo del objeto.

En otras palabras, un método que sea una función independiente. Un método estático toma sus argumentos de entrada y devuelve un resultado en base a esos valores de entrada. Y nada más.

Dentro de un método estático no podemos acceder a los atributos del objeto, pero sí podremos acceder a los atributos estáticos, ya que pertenecen a la clase. Aquí tenemos un ejemplo de la implementación de una calculadora con métodos estáticos.

Como los métodos ADD y SUBTRACT no necesitan valores específicos del objeto pueden ser declarados como estáticos y así podremos llamarlos directamente desde la clase sin necesidad de crear ningún objeto.

## ENUMS

### ENUMS

En Java podemos utilizar Enums para definir colecciones de constantes como pueden ser las estaciones del año, los días de la semana, etc.



Un enum en Java es un tipo de clase especial y a diferencia de las otras colecciones en Java, enum se encuentra en el paquete de java.lang.

### Creating an enum

```
public enum Season {SPRING,SUMMER,AUTUMN,WINTER}
```

Since elements are constants  
use capital their names

Creamos un enum de la siguiente manera. Como los elementos que contiene son constantes usamos mayúsculas en su definición.

### Fields in enums

```
public enum Season {
 SUMMER("summer"), AUTUMN("autumn"), WINTER("winter"), SPRING("spring")
 private String season;
 Season(String season) {
 this.season = season;
 }
}
```

enum constructor is implicitly private

En los enum podemos definir atributos. En tal caso deberemos especificar también un constructor, como los enums tienen valores constantes, este constructor se crea implícitamente como privado.

### Methods in enums

```
public enum Season {
 SUMMER("summer"), AUTUMN("autumn"), WINTER("winter"), SPRING("spring")
 private String season;
 Season(String season) {
 this.season = season;
 }
 public String getSeason() {
 return this.season;
 }
}
```

También podemos declarar métodos dentro de los enums, por ejemplo para devolver el valor de un elemento.

## Enums usage

```
public static void printSeason(Season season) {
 switch (season) {
 case SUMMER: {
 System.out.println(String.format("%s season", season.getSeason()));
 break;
 }
 case WINTER: {
 System.out.println(String.format("%s season", season.getSeason()));
 break;
 }
 case AUTUMN: {
 System.out.println(String.format("%s season", season.getSeason()));
 break;
 }
 case SPRING: {
 System.out.println(String.format("%s season", season.getSeason()));
 break;
 }
 }
}
```

Y aquí tenemos un ejemplo de cómo podríamos utilizar el enum Season, en este caso en una sentencia switch case, como si fuera un tipo primitivo de datos.

Evaluamos un objeto season como expresión en el switch. Podemos utilizar los valores constantes del enum para establecer los bloques case.

## Enums inside classes

```
public class Calendar {
 Season season;
 ArrayList<Day> days;
}
```

Y también podemos usar los enums que hayamos definido como tipo de dato de cualquier atributo de una clase.

Podemos encontrar información detallada de la clase enum en la documentación oficial de Oracle.

compact1, compact2, compact3  
java.lang  
**Class `Enum<E extends Enum<E>>`**  
java.lang.Object  
java.lang.Enum<E>  
**Type Parameters:**  
E - The enum type subclass  
**All Implemented Interfaces:**  
Serializable, Comparable<E>  
  
public abstract class `Enum<E extends Enum<E>>`  
extends Object  
implements Comparable<E>, Serializable  
  
This is the common base class of all Java language enumeration types. More information about enums, including descriptions of the implicitly declared methods synthesized by the compiler, can be found in section 8.9 of *The Java™ Language Specification*.  
  
Note that when using an enumeration type as the type of a set or as the type of the keys in a map, specialized and efficient set and map implementations are available.  
**Since:**  
1.5  
**See Also:**  
Class.getEnumConstants(), EnumSet, EnumMap, Serialized Form

## PROGRAMACIÓN ORIENTADA A OBJETOS II

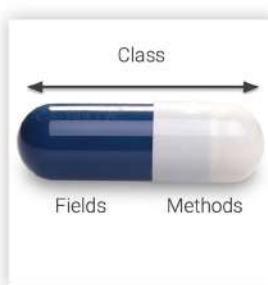
### OPP CONCEPTS

#### MORE ABOUT OPP



Como nos podemos imaginar por el título, la Programación Orientada a Objetos es más que dividir el código en clases. La programación orientada a objetos es cómo diseñamos nuestro código de manera que sea fácil de entender y lo más importante, fácil de extender para añadir nuevas funcionalidades. Para aprender cómo diseñar un buen programa en Java, es necesario que entendamos algunos de los conceptos básicos de la programación orientada a objetos.

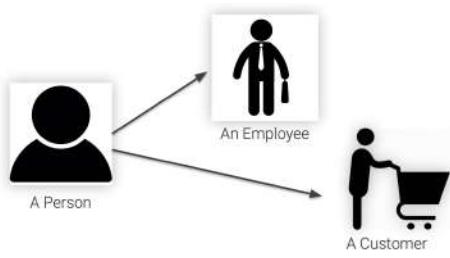
#### Encapsulation



Uno de ellos es encapsulación, que como ya sabemos, significa que cada clase es como una cápsula que contiene todo lo que necesita y nada más. De esta forma, si algo va mal, podemos saber exactamente qué código revisar.

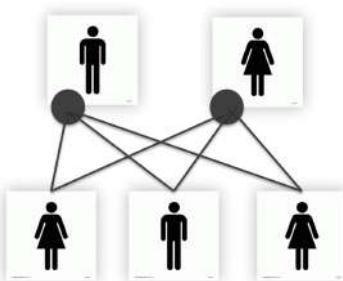
Por ejemplo, si hay un fallo en un objeto del tipo CAR, tendremos que revisar el código de la clase CAR. Además, esto permite al equipo de programadores y programadoras que desarrolla una aplicación trabajar en múltiples objetos simultáneamente, sin solaparse o duplicar código.

#### Polymorphism



Otro concepto importante es polimorfismo, una palabra algo complicada que intentaremos explicar de forma sencilla. Polimorfismo significa múltiples formas u aspectos. Así, por ejemplo, una persona puede ser un empleado o empleada en una oficina y también puede ser un cliente en un supermercado. En programación, polimorfismo define cómo los objetos pueden tener múltiples identidades, de forma que podemos agrupar diferentes objetos como si fueran del mismo tipo bajo ciertas condiciones.

#### Inheritance



Pero quizás el concepto más importante de todos es herencia. El concepto de herencia en programación es similar al concepto de herencia en la vida real. Significa pasar rasgos o características de padres a hijos, como el color de los ojos, del pelo o características faciales.

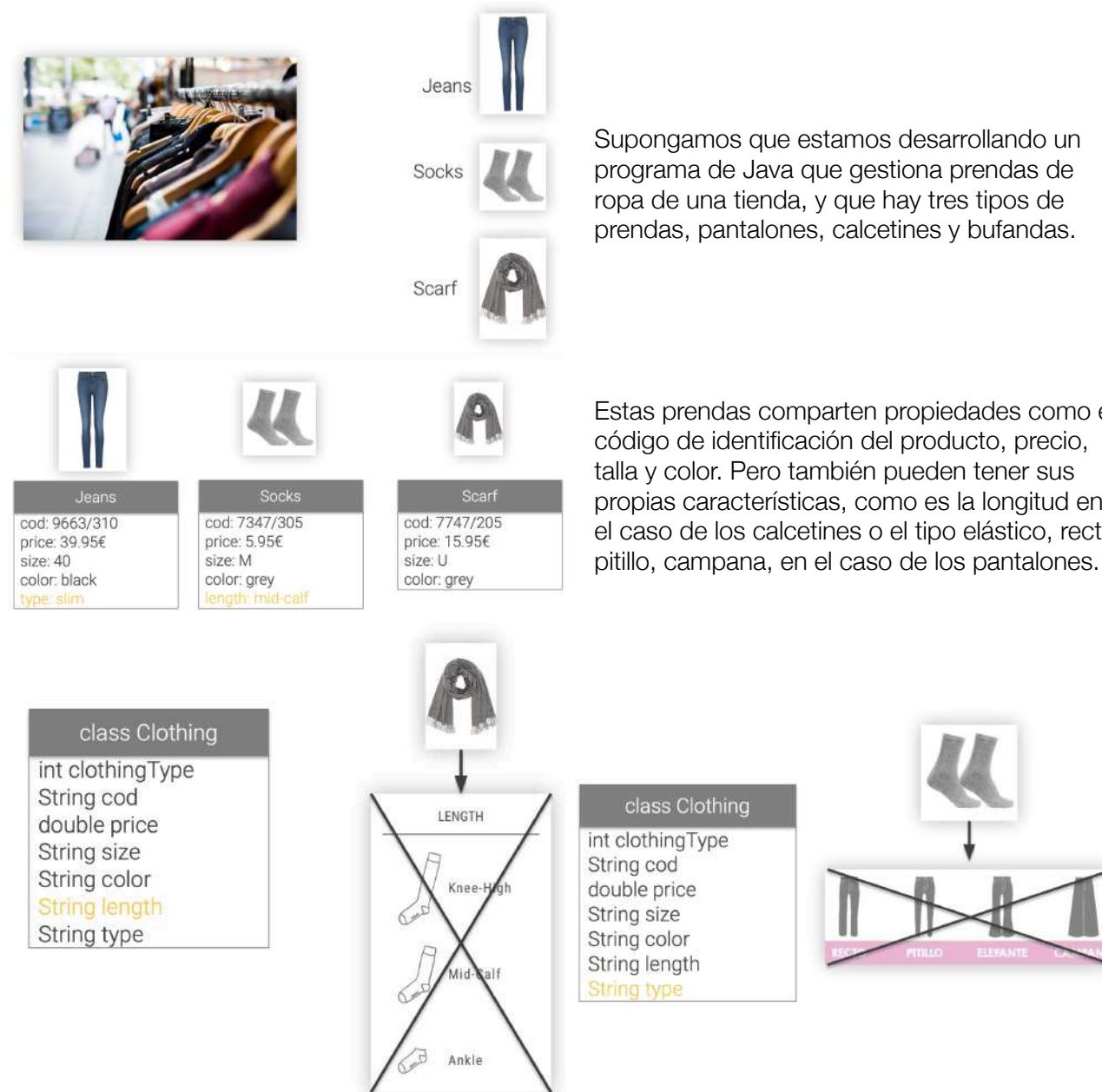
Las clases no sólo podrán usar otras clases, sino que podrán heredar sus características y métodos, para extender sus capacidades. Aplicar herencia es clave en un buen diseño de un programa Java porque ahorra tiempo a la hora de escribir código y le permite tener una estructura consistente y bien organizada.

Si nos vamos a docs.oracle podremos profundizar más en los conceptos explicados.

- <https://docs.oracle.com/javase/tutorial/java/concepts/index.html>

## INHERITANCE: INTRODUCTION

### INHERITANCE



Si quisieramos implementar esto en Java, podríamos hacerlo en una única clase llamada, por ejemplo, ropa o clothing.

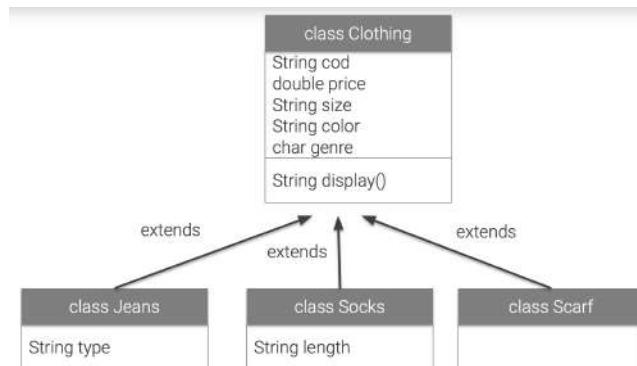
Y usar un atributo llamado algo así como clothing type que identificara el tipo de prenda cero por ejemplo para pantalones, uno para calcetines y dos para bufandas pero esto significaría que tenemos que incluir todos los atributos en esta clase. Y claro no tiene mucho sentido que una bufanda tuviera el atributo longitud, o que unos calcetines hubieran el atributo tipo.

Otra opción sería crear una clase para cada tipo de prenda.

| class Jeans                                                                            | class Socks                                                                              | class Scarf                                                                              |
|----------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| String cod<br>double price<br>String size<br>String color<br>char genre<br>String type | String cod<br>double price<br>String size<br>String color<br>char genre<br>String length | String cod<br>double price<br>String size<br>String color<br>char genre<br>String length |

De esta forma, cada clase contendría solamente los atributos y métodos que tiene sentido para esa clase. Pero también significaría repetir este código en cada una de las tres clases.

Y si por ejemplo quisiéramos añadir el género, masculino o femenino en cada una de las prendas tendríamos que añadirlo en las tres clases. Puede no parecernos mucho ahora, pero a medida que la aplicación fuera creciendo para gestionar más tipos de prendas, por ejemplo, camisetas, camisas, vestidos, un simple cambio como éste podría suponer bastante tiempo, además de duplicar código.



Así que, debe de haber una manera de hacerlo mejor. Java permite a una clase extender de otra. Esto es lo que se denomina herencia. Vamos a ver cómo usarla para resolver esto. Si comenzamos creando la clase ropa o clothing que contenga todos los atributos y métodos comunes, podemos luego crear una clase para cada uno de los tipos de prenda y hacer que extienda de la clase base Ropa.

Cada una de ellas tendrá sus propios métodos y atributos, así como los métodos y atributos de la clase base Ropa o Clothing.

```

public class Clothing{
 //field declarations
 String cod;
 double price;
 String size;
 String color;
 char genre; //M=Man, W=Woman

 //method implementations
 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre;
 }
}

```

Clothing.java

```

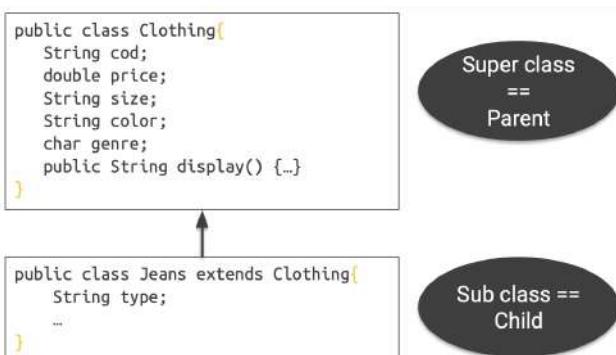
public class Jeans extends Clothing{
 //field declarations
 String type; //slim, fit, ...

 //method implementations
}

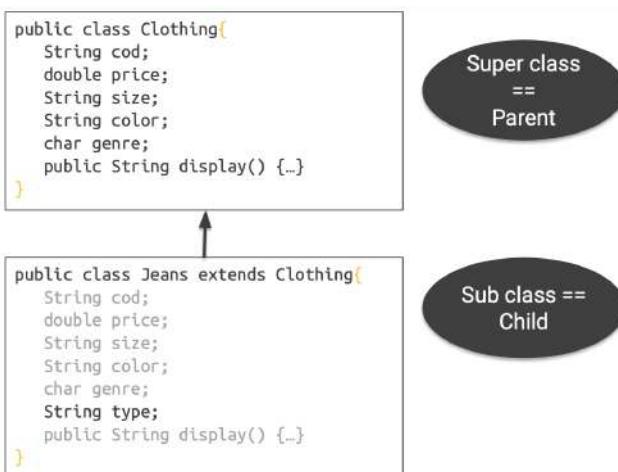
```

Trousers.java

Para hacer esto en Java, creariamos primero la clase base, y luego creariamos cada una de las otras clases, por ejemplo, la clase pantalón, haciendo que extienda de la clase base, en el ejemplo la clase Clothing.

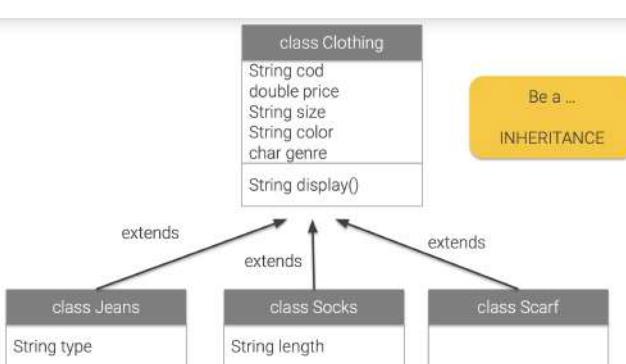
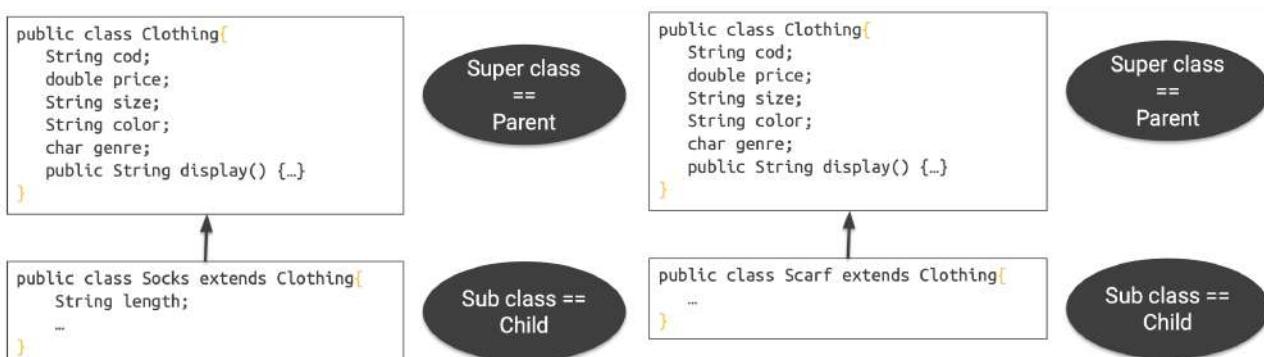


La clase Base, de la que extienden las otras, se denomina Superclase y es la clase Padre. Y la clase que extiende de la Superclase se denomina Subclase y es la clase hija. Cuando una clase extiende de otra, hereda todo lo que la superclase tenga.



Aunque no aparezca en el código, todos los atributos y métodos que estén en la superclase es como si estuvieran en la subclase.

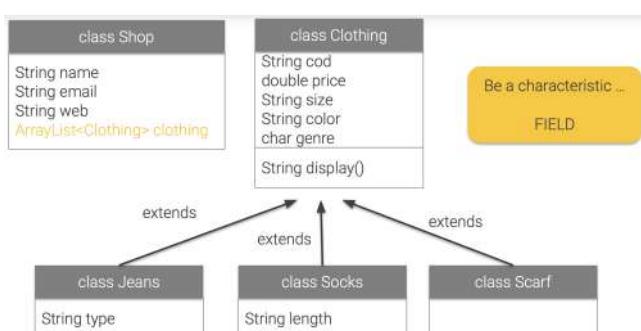
Siguiendo con el ejemplo crearemos la clase socks o calcetines que extendería de la clase clothing y que tendría su propio atributo longitud. Y la clase scarf o bufanda, que extendería también de la clase clothing.



Suponemos que nos estaremos dando o cuenta con este ejemplo que usar herencia nos permite minimizar repeticiones de código, a la vez que tenemos flexibilidad y una buena estructura de clases, separadas que encapsulan su información.

Ahora que sabemos qué es herencia, es importante que no confundamos conceptos. Ser un una corresponde a una relación de tipo especialización o generalización, esto es herencia.

Ser una característica o parte de corresponde a una relación de agregación, esto es, atributo.



Así, por ejemplo, en la aplicación de gestión de prendas de ropa, un pantalón, unos calcetines, una bufanda son un tipo de ropa, por esto su relación es de herencia. Pero si en la aplicación tuviéramos que gestionar tiendas de ropa, una tienda tiene como característica, además de un nombre, email, web, etc... una colección de prendas. Aquí la colección de objetos del tipo Clothing es un atributo de la clase Tienda.

Ahora es tu turno

Crea un nuevo proyecto en IntelliJ llamado ClothingManagerInit que contenga las clases Jeans, Socks y Scarf. Asegúrate que extienda de la clase Clothing que contiene las propiedades base y el método Display. No te preocupes ahora de implementar métodos constructores ni de aplicar encapsulación. Céntrate en el código más que en la lógica. Te recomiendo que sigas los siguientes pasos. En los materiales tienes el código solución.

## INHERITANCE: OVERRIDING METHODS

### OVERRIDING METHODS

```
public class Clothing {
 String cod;
 double price;
 String size;
 String color;
 char genre; //W==Woman, M==Man

 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre;
 }
}
```

Clothing.java

```
public class Clothing {
 String cod;
 double price;
 String size;
 String color;
 char genre;
 public String display() {...}
}

public class Jeans extends Clothing {
 String cod;
 double price;
 String size;
 String color;
 char genre;
 String type;
 public String display() {...}
}
```

Cuando una clase extiende de otra, todos los métodos public declarados en la superclase se incluyen automáticamente en la subclase, sin que tengamos que hacer nada. Así por ejemplo, si la clase clothing dispone del método display que retorna un string con el detalle de los valores de sus atributos.

Aunque no lo veamos, cada una de las subclases que extienda de ella, dispondrá de este método, igual que dispone de sus atributos.

En Java está permitido que en una subclase sobreescramos los métodos que se heredan de la superclase. Sobreescribir básicamente significa redeclarar el método en la subclase y modificar su bloque de código.

```
public class Main {
 public static void main(String[] args) {
 Jeans jeans = new Jeans();
 jeans.cod = "9663/310";
 jeans.price = 39.95;
 jeans.size = "40";
 jeans.color = "blue";
 jeans.genre = 'M';
 jeans.type = "slim";
 System.out.println(jeans.display());
 }
}
```

Main.java

```
public class Main {
 public static void main(String[] args) {
 Jeans jeans = new Jeans();
 jeans.cod = "9663/310";
 jeans.price = 39.95;
 jeans.size = "40";
 jeans.color = "blue";
 jeans.genre = 'M';
 jeans.type = "slim";
 System.out.println(jeans.display());
 //Prints: cod='9663/310', price=39.95, size='40', color='blue', genre=M
 }
}
```

Main.java

Volviendo a nuestro ejemplo, cuando en el método main creamos un objeto de una subclase, por ejemplo jeans, podemos llamar desde este objeto al método display, para mostrar por consola sus características.

```
public class Jeans extends Clothing{

 String type; //slim, fit, ..

 public String display() {
 return "Jeans" +
 "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre +
 ", type=" + type + '\'' +
 '}';
 }
}
```

Jeans.java

```
public class Main {

 public static void main(String[] args) {

 Jeans jeans = new Jeans();
 jeans.cod = "9663/310";
 jeans.price = 39.95;
 jeans.size = "40";
 jeans.color = "blue";
 jeans.genre = 'M';
 jeans.type = "slim";
 System.out.println(jeans.display());
 //Prints: Jeans[cod='9663/310', price=39.95, size='40', color='blue', genre=M,
 //type='slim']
 }
}
```

Main.java

```
public class Socks extends Clothing{

 String length; //knee-high, mid-calf, ...

 public String display() {
 return "Socks" +
 "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre +
 ", length=" + length + '\'' +
 '}';
 }
}
```

Socks.java

Implementemos una versión customizada del método display en la clase jeans, para mostrar que se trata de este tipo de prenda y también para mostrar el valor de su atributo propio, Type.

Debemos darnos cuenta de que la declaración del método en la superclase y en la subclase es idéntica y que lo que cambia es la implementación en la subclase, que permite mostrar el tipo de prenda, jeans en este caso, y el valor del atributo, type.

Si ejecutamos de nuevo el código del método main, la salida por consola al llamar al método display concuerda con el bloque de código desarrollado en la subclase. Es importante que nos demos cuenta de que una vez hemos sobreescrito un método en una subclase, se ignora que tenga este método en la superclase.

Literalmente, se ha sobreescrito. Bien, y podríamos entonces hacer lo mismo para la subclase SOCKS.

En los materiales del curso disponemos del código del proyecto denominado ClothingManagerOverrideMethod que hemos trabajado en este vídeo.

Vamos a comprobar si lo estamos entendiendo.

Suponemos que tenemos la definición de la clase Car tal y como ves en la captura y la clase Track que extiende de la clase Car. ¿Cuál será la salida por consola al ejecutar el siguiente código? En la primera línea de código se crea un objeto del tipo Track. En la segunda línea se llama el método M1. Este método está sobre escrito en la subclase Track, por tanto se ejecutará el bloque de código de la subclase imprimiendo Track1. En la segunda linea tenemos el método M2 definido en la superclase y por tanto heredado en la subclase que no lo modifica, así que imprime CAR2.

¿Cuál es la salida por consola al ejecutar el siguiente código?

```
Truck myCar = new Truck();
myCar.m1(); //Prints: truck 1
myCar.m2(); //Prints car 2
```

```
public class Car{
 public void m1(){ System.out.println("car 1"); }
 public void m2(){ System.out.println("car 2"); }

}
```

```
public class Truck extends Car{
 public void m1(){ System.out.println("truck 1"); }
}
```

## FINAL KEYWORD

En según qué situaciones permitir que cualquier subclase pueda sobreescribir el código de un método puede ser peligroso. Si queremos proteger algún método de ser sobreescrito por una subclase podemos. ¿Cómo? Pues etiquetándolo con la palabra clave FINAL. Un método etiquetado como FINAL es accesible desde las subclases pero no podrá ser sobreescrito. De esta forma tenemos la garantía de que cualquier método FINAL tendrá el mismo comportamiento que el definido en el código de la superclase.

```
public class Room{
 double width;
 double length;

 public final double getArea(){
 return width*length;
 }
}
```

Room.java

```
public class LivingRoom extends Room{
 // Not allowed to override getArea() here
}
```

LivingRoom.java

Aquí tenemos un ejemplo. La clase Room dispone del método GetArea, definido como Final. El cálculo del área de una habitación, sea cual sea, cocina, baño, sala de estar, será la misma, largo por ancho. Ahora, si tenemos una subclase como livingRoom no podremos sobreescribir el método getArea.

```
public class Main {
 public static void main(String[] args) {
 LivingRoom livingRoom = new LivingRoom(5,3);
 double area = livingRoom.getArea();
 System.out.println(area); //prints 15
 }
}
```

Main.java

```
public class Main {
 public static void main(String[] args) {
 final double MAX_ROOMS = 10;

 LivingRoom livingRoom = new LivingRoom(5,3);
 double area = livingRoom.getArea();
 System.out.println(area); //prints 15
 }
}
```

Main.java

Pero si será accesible desde cualquier objeto de la subclase, tal y como vemos en el ejemplo de la captura.

```
public class Main {
 public static void main(String[] args) {
 final double MAX_ROOMS = 10;
 MAX_ROOMS = 0; // This is not allowed and will show a compiler error!

 LivingRoom livingRoom = new LivingRoom(5,3);
 double area = livingRoom.getArea();
 System.out.println(area); //prints 15
 }
}
```

Main.java

Debemos recordar que la palabra clave final también podemos usarla con variables o atributos, pero en estas situaciones no tiene nada que ver con herencia.

Final delante de la declaración de una variable o atributo significa que es una constante y por tanto si intentamos actualizar su valor se nos mostrará un error de compilación.

## INHERITANCE: SUPER KEYWORD

### SUPER

Súper es otra palabra clave de Java como this o Break que vamos a conocer ahora. La palabra clave súper podemos usarla desde una subclase para acceder a atributos y métodos de la superclase.

```
public class Jeans extends Clothing{
 String type; //slim, fit, ...

 public String display() {
 return "Jeans{" +
 "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre +
 ", type=" + type + '\'';
 }
}
```

Jeans.java

Vemos un ejemplo, sabemos que una vez ha sobrescrito un método en una subclase, se ignora qué contenga este método en la superclase, literalmente, se ha sobrescrito.

Así, cuando llamemos al método Display desde cualquier objeto del tipo jeans se ejecutará el bloque de código que exista en la subclase jeans.

```
public class Clothing {
 String cod;
 double price;
 String size;
 String color;
 char genre; //W==Woman, M==Man

 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre;
 }
}
```

We want to continue to use the original method, and ADD extras for each subclass individually.



super

Clothing.java

En este caso, no queremos tirar a la basura la implementación del método display de la superclase Clothing, sino que nos interesa continuar usando el método original para añadir texto extra en la subclase.

Lo haremos gracias a la palabra clave de Java, super.

```
public class Jeans extends Clothing{
 String type; //slim, fit, ...
 public String display() {
 return "Jeans" +
 "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre +
 ", type=" + type + '\'' +
 '}';
 }
}
```

Jeans.java

```
public class Jeans extends Clothing{
 String type; //slim, fit, ...
 public String display() {
 return "Jeans" + super.display() +
 ", type=" + type + '\'' +
 '}';
 }
}
```

Access to super class  
display method

Jeans.java

Así que modificaremos el código del método Display de la clase Jeans. Rehusamos el método de la superclase en la subclase usando la palabra clave super seguido de punto y el nombre del método. La instrucción super.display hará que se ejecute el bloque de código del método de la superclase, en este caso retornando un string, que se concatenará con la cadena de texto declarada en el bloque de código del método display de la clase jeans.

```
public class Main {
 public static void main(String[] args) {
 Jeans jeans = new Jeans();
 jeans.cod = "9663/310";
 jeans.price = 39.95;
 jeans.size = "40";
 jeans.color = "blue";
 jeans.genre = 'M';
 jeans.type = "slim";
 System.out.println(jeans.display());
 //Prints: Jeans{cod='9663/310', price=39.95, size='40', color='blue', genre=M,
 //type='slim'}
 }
}
```

Main.java

Si ejecutamos el bloque de código del método Main donde hemos declarado e inicializado un objeto del tipo jeans, la salida por consola al llamar al método display concuerda con el bloque de código desarrollado en la subclase.

Antes de continuar vamos a comprobar si lo estamos entendiendo. Supongamos que tenemos la definición de la clase car tal y como ves en la captura y la clase truck que extiende de la clase car. ¿Cuál será la salida por consola al ejecutar el siguiente código? En la primera línea de código se crea un objeto del tipo truck. En la segunda línea se llama al método m1. Este método está sobreescrito en la subclase truck, por tanto se ejecutará el bloque de código de la subclase, imprimiendo track1. En la tercera línea se llama al método M2. Este método está sobrescrito en la subclase Truck, por tanto se ejecutará el bloque de código de la subclase, que llama al método M1 de la superclase, por tanto se imprimirá K1.

¿Cuál es la salida por consola al ejecutar el siguiente código?

```
Truck myCar = new Truck();
myCar.m1(); //Prints truck 1
myCar.m2(); //Prints car 1
```

```
public class Car{
 public void m1(){ System.out.println("car 1"); }
 public void m2(){ System.out.println("car 2"); }
}
```

```
public class Truck extends Car{
 public void m1(){ System.out.println("truck 1"); }
 public void m2(){ super.m1(); }
}
```

```
super()
```

```
public class Clothing {
 String cod;
 double price;
 String size;
 String color;
 char genre; //W=woman, M=man

 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre;
 }
}
```

Clothing.java

Podemos usar también Super para llamar al constructor de la superclase desde el constructor de la subclase. Se utiliza habitualmente cuando se define un constructor propio en la superclase. En el constructor de la subclase tendrás que llamar primero al constructor de la superclase para luego añadir más código.

```
public class Clothing {
 String cod;
 double price;
 String size;
 String color;
 char genre; //W=woman, M=man

 public Clothing(string cod, double price, String size, String color, char genre) {
 this.cod = cod;
 this.price = price;
 this.size = size;
 this.color = color;
 this.genre = genre;
 }

 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre;
 }
}
```

Clothing.java

Si definimos en la superclase Clothing un constructor propio que permite inicializar el valor de todos sus atributos.

```
public class Jeans extends Clothing{
 String type; //slim, fit, ..

 public Jeans(String cod, double price, String size,
 String color, char genre, String type) {
 super(cod, price, size, color, genre);
 this.type = type;
 }

 public String display() {
 return "Jeans{" + super.display() +
 ", type=" + type + '\'' +
 '}';
 }
}
```

1st call the superclass constructor

```
public class Jeans extends Clothing{
 String type; //slim, fit, ..

 public Jeans(String cod, double price, String size,
 String color, char genre, String type) {
 super(cod, price, size, color, genre);
 this.type = type;
 }

 public String display() {
 return "Jeans{" + super.display() +
 ", type=" + type + '\'' +
 '}';
 }
}
```

2nd add more code (if necessary)

Jeans.java

Jeans.java

En cualquiera de sus subclases, por ejemplo Jeans, tendremos que implementar al menos un constructor propio. Dentro del bloque de código de este constructor, tendremos que llamar primero al constructor de la superclase, usando la palabra clave super, y entre paréntesis los argumentos de entrada del constructor de la superclase si los tiene. Para luego, añadir más código si fuera necesario, como aquí donde necesitamos inicializar el atributo type por ser una característica propia de la subclase jeans.

```
public class Jeans extends Clothing{
 String type; //slim, fit, ..

 public Jeans(String cod, double price, String size,
 String color, char genre, String type) {
 this.type = type;
 super(cod, price, size, color, genre);
 }

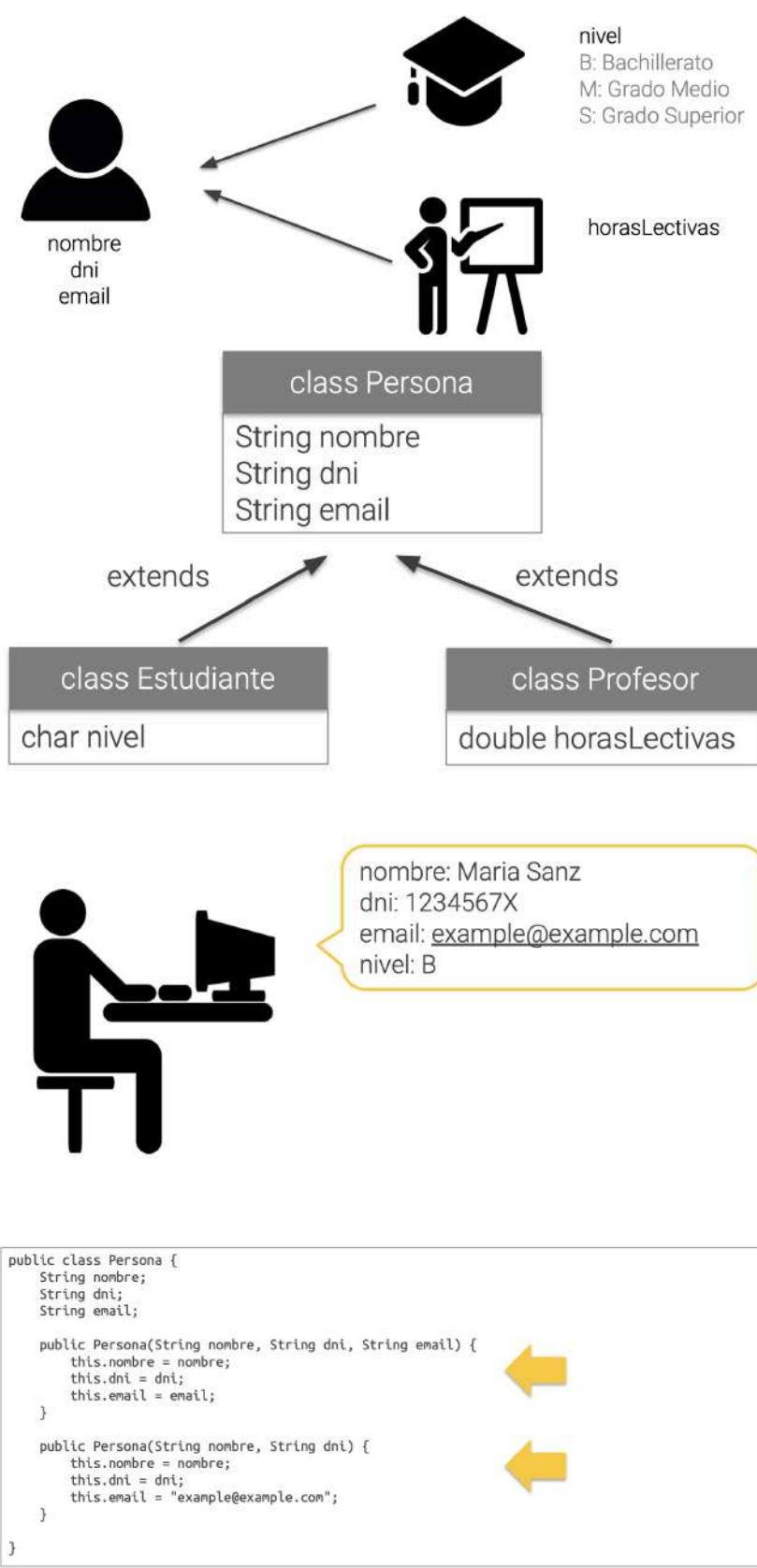
 public String display() {
 return "Jeans{" + super.display() +
 ", type=" + type + '\'' +
 '}';
 }
}
```

Error compilation!  
A call to super class constructor  
must be the first statement

Jeans.java

Debemos recordar que la llamada al constructor de la superclase debe ser la primera sentencia en el bloque del código del constructor de la subclase, sino se producirá un error de compilación.

Centro de estudios FP



Pongamos otro ejemplo. Tenemos ahora un centro de estudios de formación profesional con profesores y estudiantes. Profesores y alumnos se caracterizan por tener un nombre, un DNI y un e-mail. De los estudiantes además se registra el nivel de estudios, y de los profesores-profesoras las horas que imparten de clases.

Podríamos tener entonces la superclase Persona, que contenga todos los atributos y métodos comunes, y las subclases Estudiante y Profesor, cada una con sus atributos específicos.

Bien, en el momento de dar de alta un profesor o un alumno en el centro educativo puede no tener cuenta de e-mail.

En estos casos se le asigna una cuenta provisional, por ejemplo example@example.com y entonces a la hora de crear objetos del tipo profesor o alumno deberíamos disponer de dos constructores para permitir crear objetos tanto si tienen e-mail como si no.

En la clase Base Persona tendremos dos constructores, uno que reciba como parámetros de entrada el nombre DNI y la cuenta de email y otro que reciba el nombre el DNI y asigne al atributo email un valor por defecto, el de la cuenta provisional example@example.com.

Así, en cada subclase, por ejemplo en la subclase estudiante también dispondríamos de 2 constructores para permitir las 2 inicializaciones citadas anteriormente.

```
public class Estudiante extends Persona {
 char nivel;

 public Estudiante(String nombre, String dni, String email, char nivel){
 super(nombre, dni, email);
 this.nivel = nivel;
 }

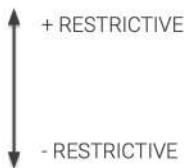
 public Estudiante(String nombre, String dni, char nivel){
 super(nombre, dni);
 this.nivel = nivel;
 }
}
```

Debemos comprobar que la primera línea de código de cada constructor es la llamada al constructor de la superclase, usando la palabra clave super y entre paréntesis los argumentos de entrada del constructor que necesitamos en cada caso.

## PROTECTED ACCESS MODIFIER

### ATTRIBUTES AND METHODS ACCESS MODIFIERS

- private
- no modifier (package private)
- protected
- public



Java nos permite el uso de los cuatro modificadores de acceso que ves en la captura para definir la visibilidad de los métodos y atributos de una clase. La regla es siempre usar el más restrictivo posible para encapsular o esconder correctamente la información de nuestras clases.

Hasta ahora, podemos resumir que encapsular correctamente una clase consiste en declarar como private los atributos de la clase y definir los métodos públicos getter y setter necesarios para acceder o modificar sus valores.

Por convención definiremos método getter para permitir leer o acceder al valor de un atributo y definiremos método setter para escribir o modificar el valor de un atributo. Definiremos como public los constructores de la clase para permitir crear objetos de la clase. Declararemos como public los métodos de la clase que queramos que sean accesibles desde otras. Y usaremos el modificador de acceso private en los métodos auxiliares de nuestra clase, de forma que no sean accesibles desde otras.

```
public class Clothing {
 String cod;
 double price;
 String size;
 String color;
 char genre; //W=Woman, M=Man

 public Clothing(String cod, double price, String size, String color, char genre) {
 this.cod = cod;
 this.price = price;
 this.size = size;
 this.color = color;
 this.genre = genre;
 }

 public String display() {
 return "cod=" + cod + "\n" +
 ", price=" + price +
 ", size=" + size + "\n" +
 ", color=" + color + "\n" +
 ", genre=" + genre;
 }
}
```

Clothing.java

```
public class Jeans extends Clothing{
 String type; //slim, fit, ...

 public Jeans(String cod, double price, String size,
 String color, char genre, String type) {
 super(cod, price, size, color, genre);
 this.type = type;
 }

 public String display() {
 return "Jeans(" + super.display() +
 ", type=" + type + "\n" +
 ")";
 }
}
```

Jeans.java

Ahora bien, ¿con la aplicación de herencia es necesario añadir alguna condición más para encapsular correctamente la información? Pues vamos a verlo. En la aplicación Clothing Manager que venimos usando de ejemplo hemos usado el modificador de acceso public en constructores y métodos tanto en la superclase Clothing como en cualquiera de las subclases, por ejemplo Jeans.

```
public class Clothing {
 private String cod;
 private double price;
 private String size;
 private String color;
 private char genre; //W==Woman, M==Man

 public Clothing(String cod, double price, String size, String color, char genre) {
 this.cod = cod;
 this.price = price;
 this.size = size;
 this.color = color;
 this.genre = genre;
 }
 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size='" + size + '\'' +
 ", color='" + color + '\'' +
 ", genre=" + genre;
 }
}
```

Clothing.java

```
public class Jeans extends Clothing{
 private String type; //slim, fit, ...

 public Jeans(String cod, double price, String size,
 String color, char genre, String type) {
 super(cod, price, size, color, genre);
 this.type = type;
 }

 public String display() {
 return "Jeans{" + super.display() +
 ", type='" + type + '\'' +
 '}';
 }
}
```

Jeans.java

### Accesibility matrix

Sabemos que marcar un atributo o método como private hará que ese atributo o método no sea accesible desde otras clases.

| Modifier    | Class | Package | Subclass | Other Classes |
|-------------|-------|---------|----------|---------------|
| Private     | Yes   | No      | No       | No            |
| No modifier | Yes   | Yes     | No       | No            |
| Protected   | Yes   | Yes     | Yes      | No            |
| Public      | Yes   | Yes     | Yes      | Yes           |



En nuestra aplicación donde existe una relación de herencia y la superclase Clothing tiene subclases, con el modificador de acceso Private, tampoco las subclases tendrán acceso a este atributo o método. Y esto puede no interesarnos en algunas situaciones.

```
public class Jeans extends Clothing{
 private String type; //slim, fit, ...

 public Jeans(String cod, double price, String size,
 String color, char genre, String type) {
 super(cod, price, size, color, genre);
 this.type = type;
 }

 public String display() {
 return "Jeans{" + super.display() +
 ", type='" + type + '\'' +
 '}';
 }
}
```

Subclasse Jeans doesn't  
need access to Clothing  
private attributes

Jeans.java

En la situación actual, desde el código de cualquiera de las subclases de la clase Clothing, no necesitamos acceso a los atributos de la superclase. Por tanto, podríamos dejar el modificador de acceso Private en todos ellos.

Nos faltaría entonces establecer como Private sus atributos y definir los métodos Getter y Setter que fueran necesarios.

Comenzaríamos en la superclase Clothing añadiendo el modificador de acceso Private en todos sus atributos.

Por simplificar, no definimos ningún método getter o setter, aunque en un entorno real probablemente necesitaríamos el método setPrice, para permitir modificar el precio de una prenda.

Y de la misma forma, en las subclases añadiríamos el modificador de acceso private a sus atributos como vemos por ejemplo en la captura, en la subclase jeans.

```

public class Jeans extends Clothing{

 private String type; //slim, fit, ..

 public Jeans(String cod, double price, String size,
 String color, char genre, String type) {
 super(cod, price, size, color, genre);
 this.type = type;
 }

 public String display() {
 return "Jeans{" + super.display() +
 ", type='" + type + '\'' +
 '}';
 }
 public double finalPrice(){
 double profitMargin=0.20;
 return price / (1-profitMargin);
 }
}

```

Subclasse Jeans needs access to Clothing private attribute price

public class Clothing{  
private String cod;  
private double price;  
...}

Jeans.java

```

Main.java x Clothing.java x Jeans.java x Socks.java x Scarf.java x
1 package com.company;
2
3 public class Jeans extends Clothing{
4 private String type; //slim, fit, etc
5
6 public Jeans(String cod, double price, String size, String color, char genre, String type) {
7 super(cod, price, size, color, genre);
8 this.type = type;
9 }
10
11 public String display() {
12 return "Jeans{" +
13 super.display() +
14 ", type='" + type + '\'' +
15 '}';
16 }
17
18 public double finalPrice(){
19 double profitMargin=0.20;
20 return price / (1-profitMargin);
21 }
22 'price' has private access in 'com.company.Clothing'
23 }

```

## PROTECTED ACCESS MODIFIER

| Modifier    | Class | Package | Subclass | Other Classes |
|-------------|-------|---------|----------|---------------|
| Private     | Yes   | No      | No       | No            |
| No modifier | Yes   | Yes     | No       | No            |
| Protected   | Yes   | Yes     | Yes      | No            |
| Public      | Yes   | Yes     | Yes      | Yes           |

↓

```

public class Clothing {
 private String cod;
 protected double price;
 private String size;
 private String color;
 private char genre; //Name, Name

 public Clothing(String cod, double price, String size, String color, char genre) {
 this.cod = cod;
 this.price = price;
 this.size = size;
 this.color = color;
 this.genre = genre;
 }
 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre;
 }
}

```

Clothing.java

```

public class Jeans extends Clothing{

 private String type; //slim, fit, ..

 public Jeans(String cod, double price, String size,
 String color, char genre, String type) {
 super(cod, price, size, color, genre);
 this.type = type;
 }

 public String display() {
 return "Jeans{" + super.display() +
 ", type='" + type + '\'' +
 '}';
 }
 public double finalPrice(){
 double profitMargin=0.20;
 return price / (1-profitMargin);
 }
}

```

public class Clothing{  
private String cod;  
**protected** double price;  
...}

Jeans.java

El modificador de acceso Protected delante de un atributo o método de la clase significará que se puede acceder a él desde el código de la clase, pero también desde las clases que estén en el mismo paquete y por todas las subclases.

Por lo tanto, si desde el código de una subclase tenemos que acceder a un atributo o método interno de la superclase, en lugar de usar el modificador de acceso Private, usaremos el modificador de acceso Protected.

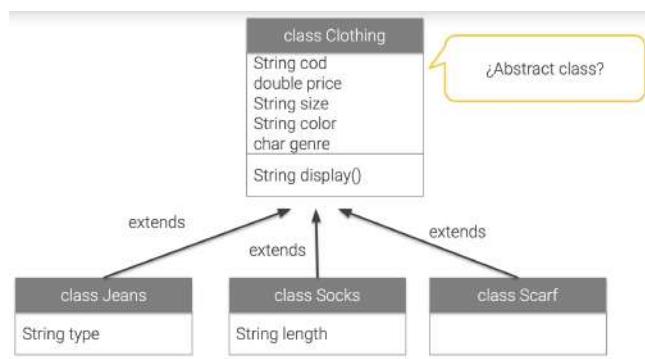
Y de esta forma podremos acceder a él desde el código de cualquiera de las subclases.

## SUMMARY

En resumen, podemos añadir que para encapsular correctamente una clase que sea superclase de otras, aplicaremos el modificador Protected en lugar de Private en los atributos y métodos que queramos que sean accesibles desde las subclases.

## INHERITANCE: ABSTRACT CLASSES

## ABSTRACT CLASSES



Al utilizar Herencia, todas las características y operaciones comunes de las distintas prendas de ropa, pantalones, calcetines, bufanda, quedan representadas en la superclase clothing.

Podemos definir la clase clothing como una clase abstracta si pensamos que solamente es una categorización y por lo tanto en la vida real nunca crearemos un objeto de este tipo, ya que una prenda de ropa en nuestra tienda será o un pantalón, o unos calcetines o una bufanda.

Ésta es la esencia de una clase abstracta, permite agrupar las propiedades y operaciones comunes de sus subclases, pero sin permitir que se creen objetos a partir de ella.

```

public abstract class Clothing {
 private String cod;
 protected double price;
 private String size;
 private String color;
 private char genre; //W==Woman, M==Man

 public Clothing(String cod, double price, String size, String color, char genre) {
 this.cod = cod;
 this.price = price;
 this.size = size;
 this.color = color;
 this.genre = genre;
 }

 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre;
 }
}

```

The code editor shows a file named "Clothing.java". The code defines an abstract class "Clothing" with private fields: cod, price, size, color, and genre. It has a constructor that takes these parameters and a public method "display()" that returns a string representation of the object's state. A yellow callout bubble to the right of the class definition contains the text: "An abstract class must be declared with the abstract keyword".

En la declaración de la clase usaremos la palabra clave Abstract para indicar que esta clase es abstracta.

```

package com.company;

public class Main {
 public static void main(String[] args) {
 Clothing clothing = new Clothing();
 Jeans jeans = new Clothing(); // "Clothing" is abstract; cannot be instantiated
 System.out.println(jeans.display());

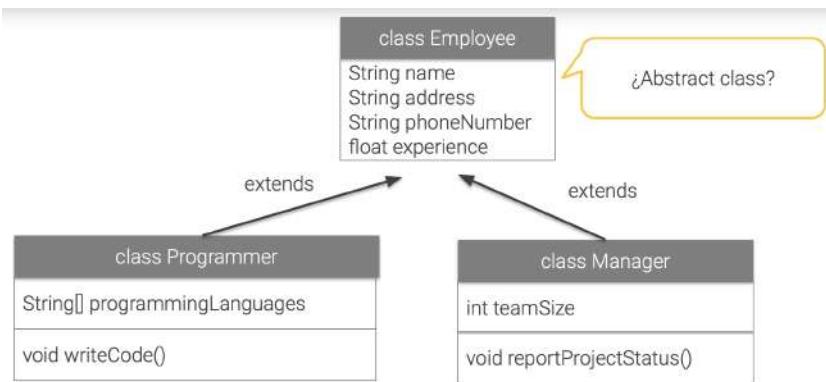
 Socks socks = new Socks();
 System.out.println(socks.display());

 Scarf scarf = new Scarf();
 System.out.println(scarf.display());
 }
}

```

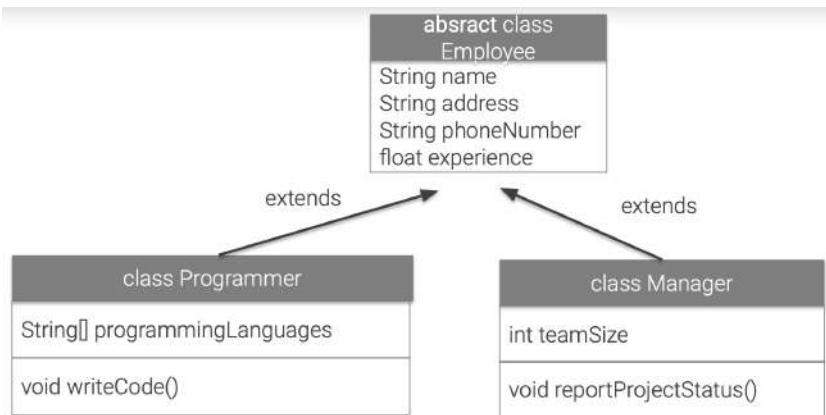
The code editor shows a file named "Main.java". It contains a main method that creates objects of the abstract class "Clothing", a concrete class "Jeans", and another concrete class "Socks". When the code is run, it prints the display strings of the jeans and socks objects. A yellow callout bubble highlights the line "Clothing clothing = new Clothing();". A tooltip appears over the word "Clothing" in the line, stating: "Clothing" is abstract; cannot be instantiated.

Cuando una clase es declarada como abstracta, no permite crear objetos a partir de ella. Si lo intentamos, se producirá un error de compilación.



Así los programadores se caracterizan por dominar distintos lenguajes de programación, y los managers por gestionar un equipo de trabajo de N programadores.

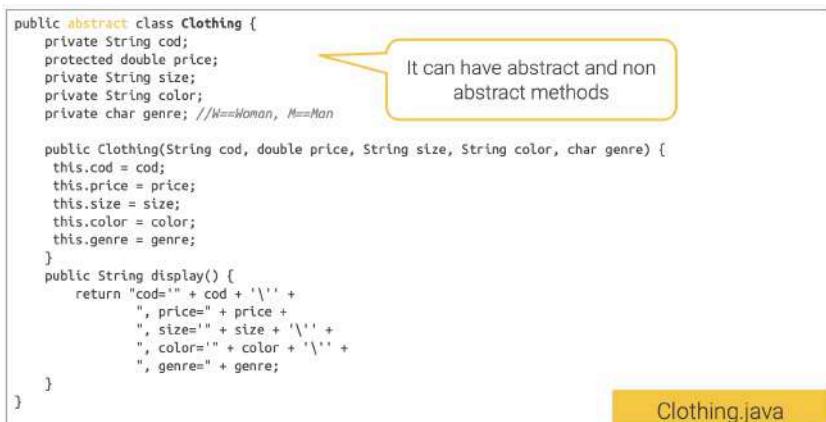
Suponemos que vemos clara la necesidad de aplicar herencia. Tendríamos la superclase Employee que agruparía las características principales de los empleados de la organización y las subclases Programmer y Manager para representar los distintos puestos de trabajo.



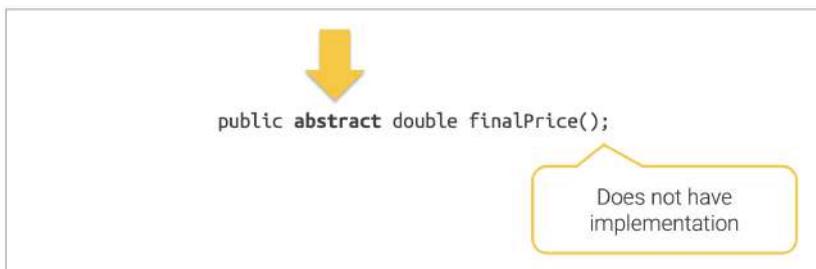
¿Es la clase Employee abstracta? La respuesta sería que sí.

Esta clase es una categorización que permite agrupar las características comunes de cualquier empleado de la organización, nunca será necesario instanciar o crear un objeto del tipo Employee ya que cualquier empleado de la organización será o programador o manager.

## ABSTRACT METHODS



Una clase abstracta puede tener cero o N métodos abstractos.



Las subclases de la clase abstracta, donde haya sido declarado, estarán obligadas a implementar su código.

```
finalPrice = price / (1-profitMargin)
```

| Jeans     | Socks | Scarf |
|-----------|-------|-------|
| 0.10-0.20 | 0.05  | 0.10  |

```
public abstract class Clothing {
 private String cod;
 protected double price;
 private String size;
 private String color;
 private char genre; //W==woman, M==man
 public Clothing(String cod, double price, String size, String color, char genre) {
 this.cod = cod;
 this.price = price;
 this.size = size;
 this.color = color;
 this.genre = genre;
 }
 public String display() {
 return "cod=" + cod + '\'' +
 ", price=" + price +
 ", size=" + size + '\'' +
 ", color=" + color + '\'' +
 ", genre=" + genre;
 }
 public abstract double finalPrice();
}
```

Clothing.java

```
package com.company;
public class Jeans extends Clothing{
 private String type;
 public Jeans(String cod, double price, String size, String color, char genre, String type) {
 super(cod, price, size, color, genre);
 this.type = type;
 }
 public String display() {
 return "Jeans[" +
 super.display() +
 ", type=" + type + '\'\' +
 "]";
 }
}
```

Un método se declara como abstracto usando la palabra clave `abstract`, tal y como vemos en la captura.

Un método abstracto simplemente se declara y no lleva nunca a implementación.

Vamos a verlo con un ejemplo. Supongamos que por cada prenda de ropa queremos calcular su precio final de venta al público, aplicando al precio base el margen de beneficio de cada producto.

Supongamos que el margen de beneficio para los pantalones varía del 10 al 20% dependiendo del tipo, para los calcetines es del 5% y para las bufandas es del 10%.

En la superclase `Clothing` declararíamos el método `Final Price` como abstracto, ya que cada tipo de prenda, pantalón, calcetines, etc... hará su implementación.

Al declarar el método `Final Price` como abstracto en la superclase `Clothing`, si en cualquiera de las subclases no lo implementamos, dará error de compilación.

Debemos comprobar que es una manera de obligarnos a desarrollar este método en las subclases y así no olvidarnos que toda prenda de ropa tiene que tener disponible esta acción, calcular el precio final de venta al público.

```
public class Jeans extends Clothing{
 private String type; //slim, fit, etc

 public double finalPrice(){
 double profitMargin;
 switch (type){
 case "slim":
 profitMargin=0.10;
 break;
 case "fit":
 profitMargin=0.15;
 break;
 default:
 profitMargin=0.20;
 break;
 }
 return price / (1-profitMargin);
 }
}
```

**Jeans.java**

Jeans



profitMargin  
0.10-0.20

```
public class Socks extends Clothing{
 private String length; //knee-high, mid-calf, ...
 ...

 public double finalPrice() {
 double profitMargin = 0.05;
 return price / (1-profitMargin);
 }
}
```

**Socks.java**

Socks



profitMargin  
0.05

```
public class Scarf extends Clothing{
 public Scarf(String cod, double price, String size, String color, char genre) {
 super(cod, price, size, color, genre);
 }

 public double finalPrice() {
 double profitMargin = 0.10;
 return price / (1-profitMargin);
 }
}
```

**Scarf.java**

Scarf



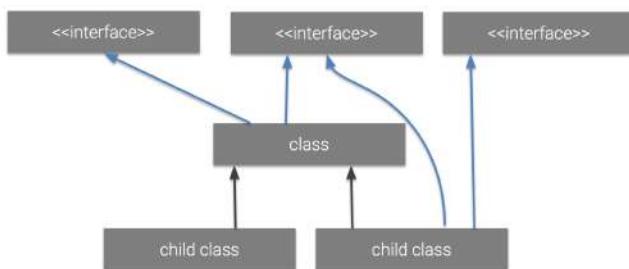
profitMargin:  
0.10

## MULTIPLE INHERITANCE: INTERFACES

### MULTIPLE INHERITANCE

Hemos visto cómo extender de una clase puede ser muy útil, sin embargo en JAVA hay una limitación, una clase solo puede extender de una única clase. Dicho con otras palabras, una clase puede tener varias subclases, pero solo puede tener una superclase.

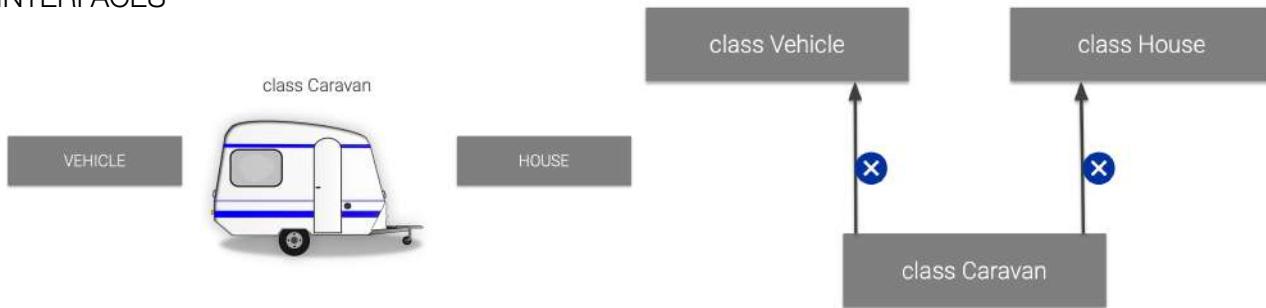
Esto es porque herencia múltiple puede causar ambigüedades si en las superclases tienen métodos similares. Si queremos ampliar información sobre herencia múltiple puedes echar un vistazo al conocido problema del diamante. La solución de Java para esto son las interfaces.



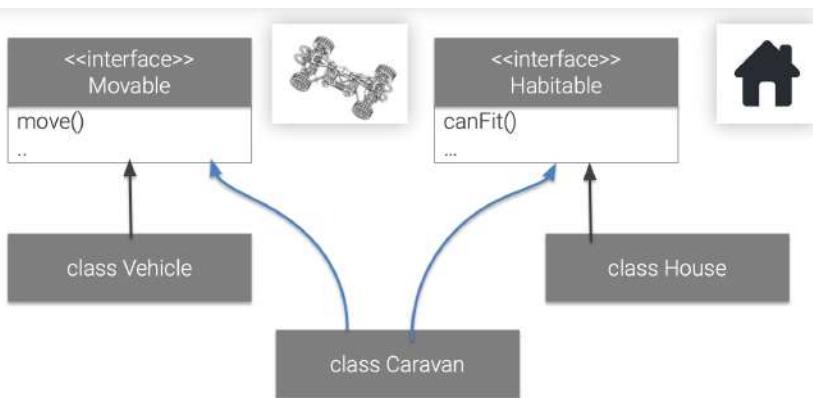
Las interfaces no tienen esta restricción, lo que significa que una clase puede implementar múltiples interfaces, permitiendo un diseño más flexible, pero eliminando el problema de la ambigüedad en la herencia múltiple.

Así que vamos a ver qué es una interface y cómo usarla.

## INTERFACES



Supongamos que tenemos una clase llamada Caravana. Una Caravana es mitad vehículo mitad casa. Si en la aplicación tenemos una clase vehículo y una clase casa, no podemos extender de ambas al mismo tiempo. La solución sería introducir interfaces.



Necesitaríamos definir una interfaz, denominada por ejemplo **Movable**, donde definiríamos los métodos que toda clase con la capacidad de moverse debería tener, como es la clase Vehículo. Otra interfase podría ser **Habitable**, donde definiríamos los métodos que toda clase con la capacidad de vivir en ella debería tener, como es la clase Casa.

Así, definidas estas dos interfaces, en la clase Caravana podríamos implementar ambas al mismo tiempo. Vamos a ver el código Java para esto.

```
public interface Movable{
 void move(int distance);
 boolean canMove();
}

public interface Habitable{
 boolean canFit(int inhabitants);
}
```

**Movable.java**

**Habitable.java**

Only list method signatures  
Implementation is responsibility of the class that will implement the interface

public and abstract methods

Crear una interfaz en Java es muy parecido a crear una clase. Simplemente cambia la palabra `class` por la palabra `interface`.

Dentro del código de la interface, como podemos ver, listamos la declaración de los métodos, pero no los implementamos porque esto es responsabilidad de la clase que decide implementar este interface.

Debemos comprobar que no hemos añadido ni el modificador de acceso `public` ni `abstract` delante de los métodos. Cualquier método declarado en una interface es `public` y es `abstract`. Añadir estos modificadores es redundante.

Lo mismo para en la interface habitable, que en este caso sólo incluye el método `canFit`.

```

public class Caravan implements Movable, Habitatable{
 void move(int distance) { ... }
 boolean canMove() { ... }
 boolean canFit(int inhabitants) { ... }
}

```

Implementation is responsibility of the class that will implement the interface

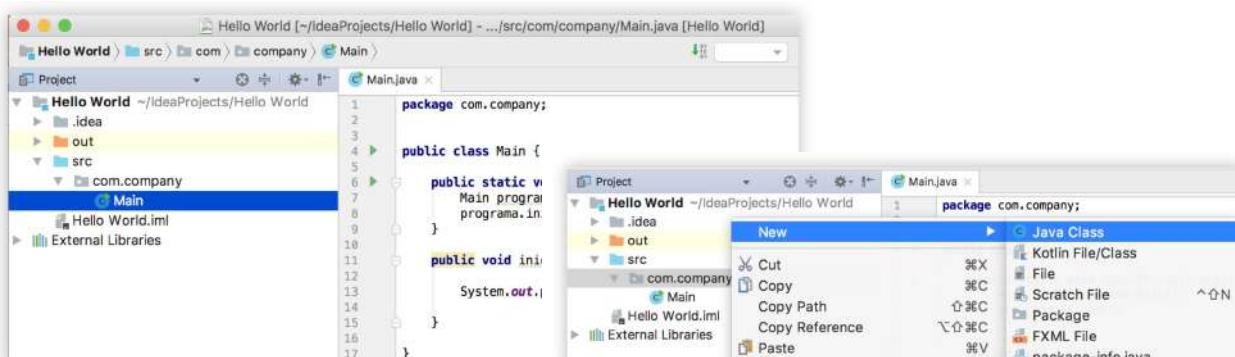
Caravan.java

Una vez que hemos declarado y creado estas dos interfaces, podemos comenzar a implementar la clase Caravana, que implementará las interfaces Movable y Habitatable. Tenemos que fijarnos que para indicar que la clase Caravana implementa ambas interfaces, hemos utilizado la palabra clave `Implements` y no `Extends` que usábamos para declarar relación de herencia.

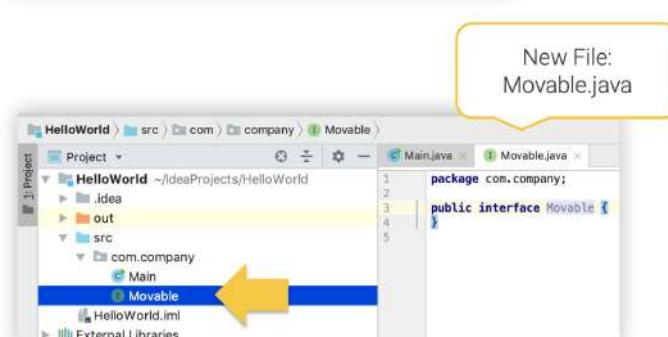
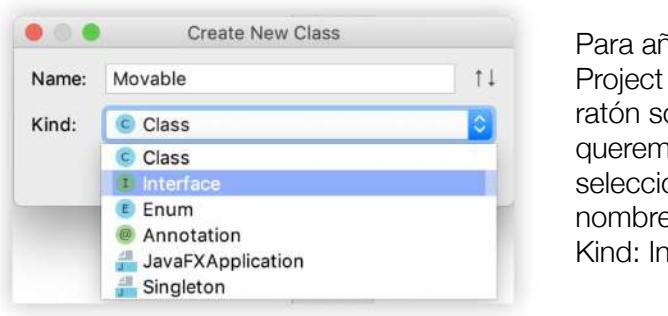
En el bloque de código de la clase tenemos que implementar el código de todos los métodos declarados en las interfaces.

Entonces las interfaces nos permiten definir qué debería hacer una clase pero no cómo lo tiene que hacer ya que en la interface declaramos los métodos y es en la clase donde implementamos el código de estos métodos. Las interfaces no sirven para crear objetos a partir de ellas, sino para listar una serie de métodos que serán implementados por las clases. Las interfaces no nos ayudan a evitar repeticiones de código son más una ayuda para diseñar la estructura de nuestros proyectos de forma óptima.

## CREATING AN INTERFACE IN INTELLIJ



Para añadir una interfaz a cualquier Java Application Project en IntelliJ, pulsamos con el botón derecho del ratón sobre el paquete com.company o sitio donde queremos crear la interfaz, y del menú que aparezca seleccionaremos New Java Class, introduciremos el nombre de la interfaz, seleccionaremos del menú Kind: Interface y pulsaremos OK.



Y ya habremos creado nuestra interfaz dentro de nuestro proyecto. Debemos comprobar que se ha creado un fichero nuevo con el nombre especificado para la interfaz, `movable.java`.

## JAVA PREDEFINED INTERFACES

**Module** java.base**Package** java.lang**Interface Comparable<T>****Type Parameters:**

T - the type of objects that this object

**Method Summary**

| All Methods       | Instance Methods | Abstract Methods                                          |
|-------------------|------------------|-----------------------------------------------------------|
| Modifier and Type | Method           | Description                                               |
| int               | compareTo(T o)   | Compares this object with the specified object for order. |

Este interface tiene un único método llamado compareTo que toma un objeto como parámetro y lo compara con otro objeto del mismo tipo. El objetivo principal de este interfaz es dar a cualquier clase la oportunidad de describir cómo comparar dos objetos. Esto es muy útil cuando queremos ordenar o buscar entre objetos del mismo tipo.

```
public class Book{
 private int numberOfPages;
 private String title;
 private String author;

 public Book(int numberOfPages, String title, String author) {
 this.numberOfPages = numberOfPages;
 this.title = title;
 this.author = author;
 }
}
```

Book.java

```
import java.util.ArrayList;
import java.util.Collections;

public class Main{
 public static void main(String[] args) {
 ArrayList<Book> books = ...
 Collections.sort(books);
 }
}
```

- Order criteria:
1. Sort by the title alphabetically
  2. If both books have the same title, then sort by the author alphabetically
  3. If both books have the same title and author, then sort by number of pages

Main.java

Por ejemplo, supongamos que tenemos la clase Book y en la aplicación tenemos una colección de objetos Book denominada Books que queremos ordenar siguiendo un determinado criterio, por ejemplo, alfabéticamente por su título. Pero si dos libros tienen el mismo título, entonces se ordenarán alfabéticamente por su autor, y si dos libros tienen igual título y autor, entonces se ordenarán según el número de páginas.

```
import java.util.ArrayList;
import java.util.Collections;

public class Main{
 public static void main(String[] args) {
 ArrayList<Book> books = ...
 Collections.sort(books);
 }
}
```

Book class needs to implement Comparable interface

Main.java

Ordenar la colección books haciendo uso de la instrucción collections.sort no será posible mientras la clase Book no implemente la interface Comparable.

Entonces en la declaración de la clase Book añadimos implementsComparable, nombre de la interface que queremos implementar. Con esta declaración la clase Book obligatoriamente tendrá que implementar el código del método compareTo que tiene declarado la interface.

The screenshot shows two snippets of Java code for a class named `Book`. The first snippet shows the declaration of the `Comparable` interface and its implementation. The second snippet shows the `compareTo` method overridden. A yellow arrow points from the `Comparable` interface declaration to the `compareTo` method implementation.

```

public class Book implements Comparable{
 private int numberofPages;
 private String title;
 private String author;

 public Book(int numberofPages, String title, String author) {
 this.numberofPages = numberofPages;
 this.title = title;
 this.author = author;
 }

}

public class Book implements Comparable{
 private int numberofPages;
 private String title;
 private String author;

 public Book(int numberofPages, String title, String author) {...}

 @Override
 public int compareTo(Book o) {
 return 0;
 }
}

```

**Method Detail**

**compareTo**

```

int compareTo(T o)
Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.) The implementor must also ensure that the relation is transitive: (x.compareTo(y) > 0 && y.compareTo(z) > 0) implies x.compareTo(z) > 0. Finally, the implementor must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z. It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."
```

In the foregoing description, the notation `sgn(expression)` designates the mathematical signum function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of expression is negative, zero, or positive, respectively.

**Parameters:**  
`o` - the object to be compared.

**Returns:**  
a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

**Book.java**

```

public class Book implements Comparable{
 private int numberofPages;
 private String title;
 private String author;

 public Book(int numberofPages, String title, String author) {...}

 @Override
 public int compareTo(Book o) {
 if (!this.title.equalsIgnoreCase(o.title)){
 return this.title.compareTo(o.title);
 //If titles are identical, lets compare authors
 }else if (!this.author.equalsIgnoreCase(o.author)){
 return this.author.compareTo(o.author);
 //If titles and authors are identical, lets compare number of pages
 }else{
 return this.numberofPages-o.numberofPages;
 }
 }
}

```

**Book.java**

Antes de escribir el código echamos un vistazo a la documentación de Java para ver cómo debería funcionar este método.

De acuerdo con la documentación, al comparar dos objetos del mismo tipo hay tres posibles salidas. Un número negativo, positivo, o cero, dependiendo de si el objeto considerado es menor, mayor o igual que el objeto especificado como argumento.

Entonces el bloque de código del método `compareTo` quedaría como podemos ver en la captura.

## POLYMORPHISM

### POLYMORPHISM

## polymorphism noun

poly·mor·phism | \pà-lè-'mòr-fizəm\

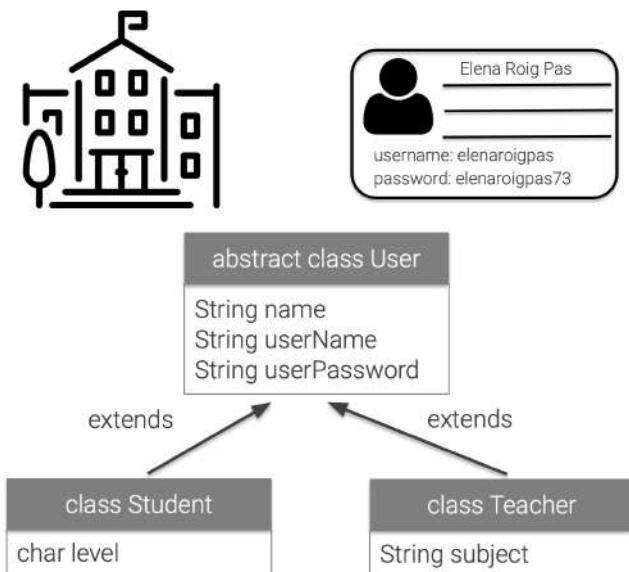
### Definition of polymorphism

- : the quality or state of existing in or assuming different forms: such as
- a(1)** : existence of a species in several forms independent of the variations of sex
- (2)** : existence of a gene in several allelic forms
- also** : a variation in a specific DNA sequence
- (3)** : existence of a molecule (such as an enzyme) in several forms in a single species

Polimorfismo literalmente significa algo que tiene múltiples formas.

En el mundo de la programación orientada a objetos, el uso de herencia permite un objeto ser polimórfico, porque cuando un objeto extiende de otra clase, no sólo es de su propio tipo, sino también del tipo de su superclase.

School Learning platform



```

public abstract class User{
 protected String name;
 protected String userName;
 protected String userPassword;

 public User(String name) {
 this.name = name;
 this.userName = createUserName();
 this.userPassword = createUserPassword();
 }

 private String createUserName(){ ... }
 private String createUserPassword(){ ... }
 public abstract String display();
}

```

Pongamos un ejemplo, una aplicación que gestione usuarios, alumnos, profesores de un colegio. Un usuario, bien sea alumno o profesor, tendrá como características sus datos personales, como nombre, apellidos... así como las credenciales de acceso a la aplicación, nombre de usuario y contraseña.

Podríamos, entonces, tener la clase usuario con los campos Nombre y las credenciales de acceso, y luego dos clases, Estudiante y Profesor, que serían subclases de Usuario, y por tanto tendrían los mismos atributos que la Superclase, y también añadirían sus propios atributos, Nivel de estudios para la clase Estudiante y Materia que imparte para la clase Profesor.

El código de la clase Usuario podría ser el que vemos en la captura. Se ha definido como abstracta porque no se quiere instanciar ningún objeto a partir de ella, sino que los objetos que se creen sean o bien del tipo estudiante o bien de tipo profesor. Dispone de tres atributos, el nombre, el nombre de Usuario y la contraseña.

Dispone de un constructor propio que permite inicializar un objeto User a partir del nombre del usuario, dando a su nombre de usuario y a su contraseña valores que se definen en los métodos privados CreateUserName y CreateUserPassword. Esta clase dispone además de un método abstracto, Display, que tendrá que ser implementado por las subclases de ésta.

```

public class Student extends User{
 private char level; //P:Primary, M:Middle School, H: High School

 public Student(String name, char level) {
 super(name);
 this.level = level;
 }

 public String display() {
 return "Student{" +
 "name='" + name + '\'' +
 ", userName='" + userName + '\'' +
 ", userPassword='" + userPassword + '\'' +
 ", level='" + level + '\'';
 }

 public char getLevel(){return this.level;}
}

```

La clase estudiante se declararía como subclase de la clase usuario. Dispondría del atributo propio nivel para registrar el nivel de estudios del estudiante. Dispondría de un constructor propio que en su bloque de código haría una llamada al constructor de la superclase y a su vez inicializaría el valor de su atributo propio.

Como Student extiende de la clase Usuario y en la clase Usuario tendría el método abstracto display, en esta subclase estamos obligados a implementar este método display. En la clase Student además hemos añadido el método getLevel que permite retornar el valor del atributo level.

```
public class Teacher extends User{
 private String subject; //Maths, Science, Music

 public Teacher(String name, String subject) {
 super(name);
 this.subject = subject;
 }

 public String display() {
 return "Teacher[" +
 "name:" + name + '\'' +
 ", userName=" + userName + '\'' +
 ", userPassword=" + userPassword + '\'' +
 ", subject=" + subject + '\'';
 }

 public String getSubject(){return this.subject;}
}
```

Teacher.java

Y la clase Profesor se declararía como subclase de la clase Usuario también. Dispondría de un atributo propio, que es la materia que imparte. Y de la misma forma que la clase Estudiante dispondría de un constructor propio, implementaría también el método Display declarado como abstracto en la Superclase.

Esta clase además dispone del método GetSubject que permite retornar el valor del atributo.

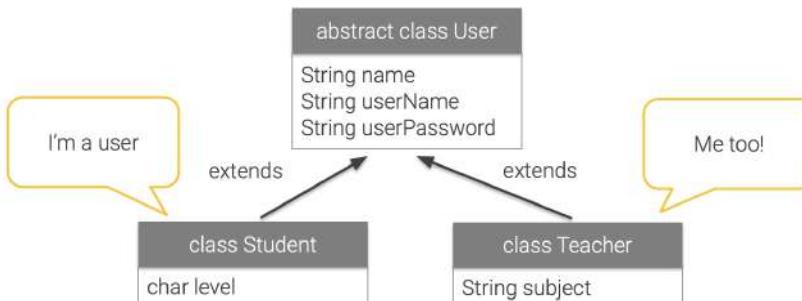
```
public class Main {
 public static void main(String[] args) {
 Student student = new Student("Elena Pérez Roig", 'P');

 Teacher teacher= new Teacher("Manuel Diaz Velasco", "Science");
 }
}
```

Main.java

Entonces para crear un objeto del tipo estudiante declararemos en primer lugar el tipo Student y usaremos New seguido del constructor de la clase.

Y para crear un objeto del tipo Profesor declararemos Teacher y usaremos New seguido del constructor de la clase.



Pero debemos recordar que la clase estudiante y profesor son subclases de la clase usuario, así que por defecto tienen también el tipo usuario.

Así que podríamos crear ambos objetos declarando el tipo de la superclase, User, y luego seguir inicializándolo según sus propios constructores. Esto da flexibilidad y vamos a ver por qué.

```
public class Main {
 public static void main(String[] args) {
 Student student = new Student("Elena Pérez Roig", 'P');

 Teacher teacher= new Teacher("Manuel Diaz Velasco", "Science");
 }
}
```

```
public class Main {
 public static void main(String[] args) {
 User student = new Student("Elena Pérez Roig", 'P');

 User teacher= new Teacher("Manuel Diaz Velasco", "Science");
 }
}
```

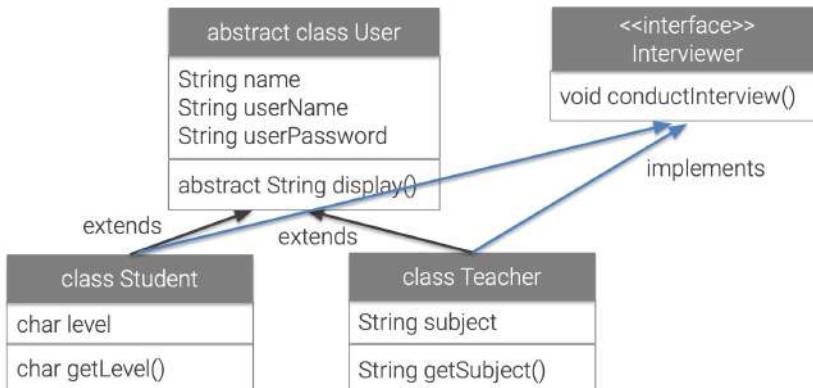
Main.java

```
public class Main {
 public static void main(String[] args) {
 ArrayList<User> primaryClassroom = new ArrayList<>();
 Student student = new Student("Elena Pérez Roig", 'P');
 primaryClassroom.add(student); //Elena
 student = new Student("Mercedes Requena Sanz", 'P');
 primaryClassroom.add(student); //Elena, Mercedes
 Teacher teacher= new Teacher("Manuel Diaz Velasco", "Science");
 primaryClassroom.add(teacher); //Elena, Mercedes, Manuel
 ...
 }
}
```

Main.java

Supongamos que tenemos que gestionar los alumnos y profesores de una clase de primaria. Podríamos crear una colección de objetos user, a la que ir añadiendo objetos de cualquiera de sus subclases, estudiante y profesor.

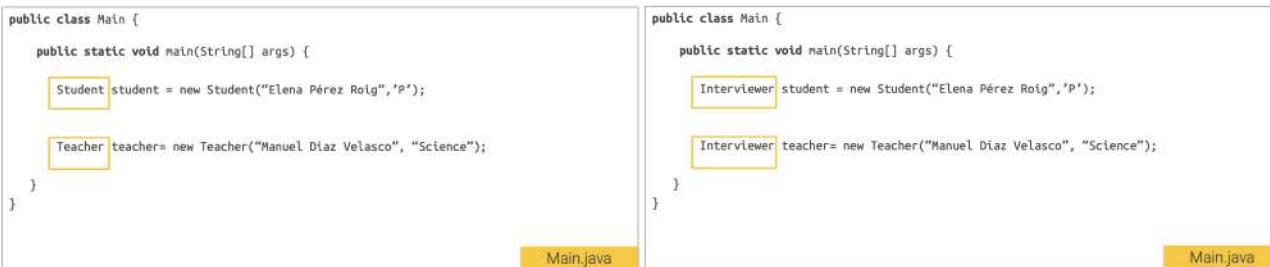
## POLYMORPHISM WITH INTERFACES



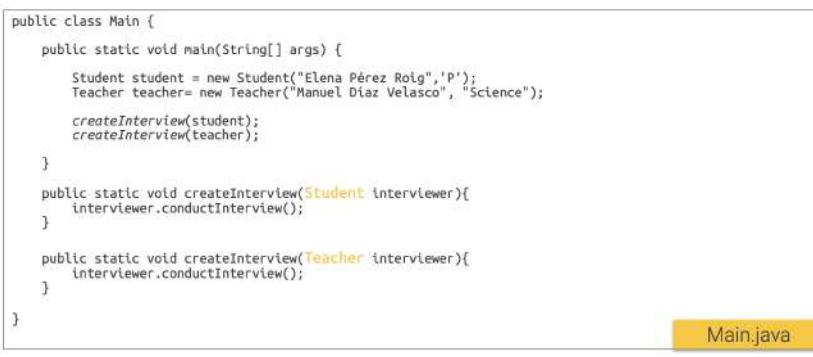
También se puede dar polimorfismo con interfaces. Supongamos que tanto estudiantes como profesores tienen la capacidad de dirigir entrevistas, cada uno de forma diferente.

Por ejemplo, los estudiantes podrían dirigir entrevistas con otros alumnos y los profesores podrían conducir entrevistas con familias.

Tendríamos entonces la interface Interviewer con el método ConductInterview y la clase Student y Teacher serían conformes a esta interfaz implementando el código del método ConductInterview.



Así que si tenemos que crear objetos del tipo Student y Teacher podríamos crearlos declarando como tipo de variable la interfaz y luego seguir inicializándolos usando sus propios constructores.



Esto es un ejemplo de uso de polimorfismo en interfaces, pero un escenario más real de uso sería éste.

Creados objetos Student y Teacher, en algún momento de la aplicación se necesita crear una entrevista dirigida bien por un estudiante o bien por un profesor.

La solución sería disponer de un único método que recibiera como parámetro de entrada cualquier objeto conforme la interfaz Interviewer. Por esto hemos declarado el método Public Static Void CreateInterview donde el parámetro de entrada es del tipo Interviewer, es del tipo de la interfaz.

Esto sería más eficiente que crear dos métodos, un método por cada tipo de objeto.

## INSTANCEOF

```
public class Main {
 public static void main(String[] args) {
 User student = new Student("Elena Pérez Roig", 'P');
 }
}
```

Super class type  
Interface type

Main.java

Hemos visto como en java la herencia e interfaces permiten a los objetos ser polimórficos y podemos en la declaración de un objeto establecer como tipo de variable el tipo de la superclase o la interface.

```
public class Main {
 public static void main(String[] args) {
 User student = new Student("Elena Pérez Roig", 'P');
 System.out.println(student.display());
 System.out.println(student.getLevel());
 }
}
```

Main.java

```
public class Main {
 public static void main(String[] args) {
 User student = new Student("Elena Pérez Roig", 'P');
 System.out.println(student.display());
 System.out.println(((Student)student).getLevel());
 }
}
```

Casting

Main.java

Haciendo esto, a partir de esta variable podemos invocar a los métodos públicos de la superclase o la interface, como es el método display de la superclase User. Pero si intentamos invocar algún método público que solo exista en la subclase, por ejemplo el método getLevel de la clase Student, se producirá un error de compilación.

¿Por qué? Porque el objeto Student se ha declarado como user. Tendríamos que hacer la conversión de tipo, tal y como ves en la captura, para poder invocar a este método.

```
public class Main {
 public static void main(String[] args) {
 ArrayList<User> primaryClassroom = new ArrayList<>();
 Student student = new Student("Elena Pérez Roig", 'P');
 primaryClassroom.add(student); //Elena
 student = new Student("Mercedes Requena Sanz", 'P');
 primaryClassroom.add(student); //Elena, Mercedes
 Teacher teacher= new Teacher("Manuel Diaz Velasco", "Science");
 primaryClassroom.add(teacher); //Elena, Mercedes, Manuel
 ...
 }
}
```

Main.java

Lo habitual es que usemos polimorfismo cuando necesitemos representar colecciones de objetos, tal y como hemos visto.

Cuando estemos recorriendo la colección con un bucle, necesitaríamos saber si es del tipo estudiante o profesor antes de hacer la conversión.

```
public class Main {
 public static void main(String[] args) {
 ArrayList<User> primaryClassroom = new ArrayList<>();
 ...
 for (User user : primaryClassroom){
 System.out.println(user.display());
 //Is Student?
 if (user instanceof Student){
 System.out.println(((Student) user).getLevel());
 }
 }
 }
}
```

Main.java

Aquí es donde entra en escena la palabra clave instanceof, que permite verificar si una variable es de un tipo. La instrucción user instanceof student devolverá true si el objeto user es del tipo student.

Cuando esto ocurra, podremos realizar casting para acceder a alguno de los métodos específicos de la subclase.

```
public class Main {
 public static void main(String[] args) {
 ArrayList<User> primaryClassroom = new ArrayList<>();
 ...
 int countTeachers = 0, countStudents = 0;
 for (User user : primaryClassroom){
 /*insert code here*/
 }
 }
}
```

Main.java

```
public class Main {
 public static void main(String[] args) {
 ArrayList<User> primaryClassroom = new ArrayList<>();
 ...
 int countTeachers = 0, countStudents = 0;
 for (User user : primaryClassroom){
 if (user instanceof Student) countStudents++;
 else if (user instanceof Teacher) countTeachers++;
 }
 }
}
```

Main.java

Vamos a comprobar que lo estamos entendiendo. ¿Qué código añadiríamos si queremos contar cuántos alumnos y cuántos profesores hay en esta clase de primaria?

Haríamos uso de Instance Of para comprobar uno por uno el tipo de los objetos user de la colección.

La condición user instanceof student devolverá true cuando el objeto user sea de tipo student y, por tanto, si se cumple esta condición, incrementaríamos en una unidad la variable count students.

Y de la misma forma, incrementaremos la variable count teachers en una unidad cuando la condición user instanceof teacher sea true.

**M3**

**PROGRAMACIÓN**

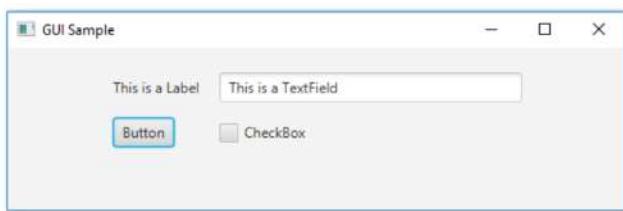
**UF**

**5**

# INTERFACES GRÁFICAS DE USUARIO

## GRAPHICAL USER INTERFACE

### INTRODUCTION TO GUI



Es muy habitual referirse a las interfaces gráficas de usuario con su acrónimo anglosajón GUI, que significa Graphical User Interface. Las GUI posibilitan la interacción entre el usuario y la aplicación de una forma amigable, mediante un conjunto de componentes gráficos tales como ventanas, botones, combos, listas, cajas de diálogo, campos de texto, etc.

Estos componentes gráficos se ubican en contenedores. Dentro de los contenedores, los componentes se organizan siguiendo un layout.

### JAVA GUIs

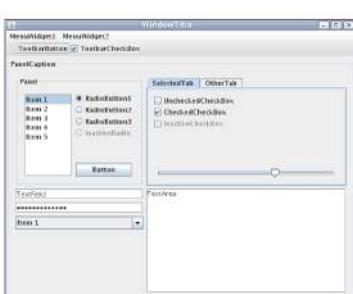
#### Awt



Java provee tres APIs con las que podemos trabajar para desarrollar interfaces gráficas.

La más básica es AWT, Abstract Window Toolkit. Es la API original que Java disponía para trabajar en modo gráfico. Actualmente está muy limitada puesto que no incluye controles tan básicos como por ejemplo una tabla.

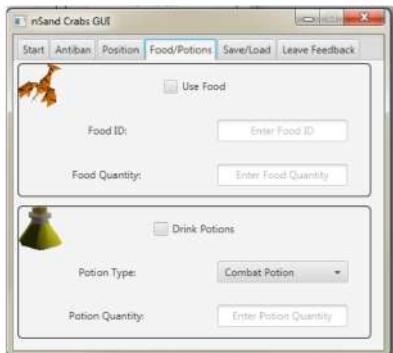
#### Swing



Aportó soluciones a las limitaciones de AWT y durante mucho tiempo ha sido la referencia en la creación de aplicaciones gráficas en Java.

Los controles se muestran igual en todos los sistemas operativos, puesto que a diferencia de AWT, los controles son objetos de Java. Como consecuencia, su apariencia se ve anticuada comparada con los botones que podemos encontrar en los sistemas operativos Windows o Mac actuales. Es interesante conocer que algunas clases de Swing mantienen dependencia con AWT y por este motivo encontramos referencias a este paquete cuando utilizamos Swing.

## JavaFX



JavaFX es la librería que Oracle ha impulsado para reemplazar Swing.

Permite crear interfaces de usuario mucho más rápidas con controles más sofisticados y estéticamente más agradables, además de aportar una mejor integración para sonido, imagen, vídeo o contenidos web.

## JAVAFX

### JavaFX main packages

| Packages                            | Description                                                                                                           |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <a href="#">javafx.application</a>  | Provides the application life-cycle classes.                                                                          |
| <a href="#">javafx.animation</a>    | Provides the set of classes for ease of use transition based animations.                                              |
| <a href="#">javafx.event</a>        | Provides basic framework for FX events, their delivery and handling.                                                  |
| <a href="#">javafx.css</a>          | Provides API for making properties styleable via CSS and for supporting pseudo-class state.                           |
| <a href="#">javafx.geometry</a>     | Provides the set of 2D classes for defining and performing operations on objects related to two-dimensional geometry. |
| <a href="#">javafx.stage</a>        | Provides the top-level container classes for JavaFX content.                                                          |
| <a href="#">javafx.scene</a>        | Provides the core set of base classes for the JavaFX Scene Graph API.                                                 |
| <a href="#">javafx.scene.canvas</a> | Provides the set of classes for canvas, an immediate mode style of rendering API.                                     |
| <a href="#">javafx.scene.chart</a>  | The JavaFX User Interface provides a set of chart components that are a very convenient way for data visualization.   |

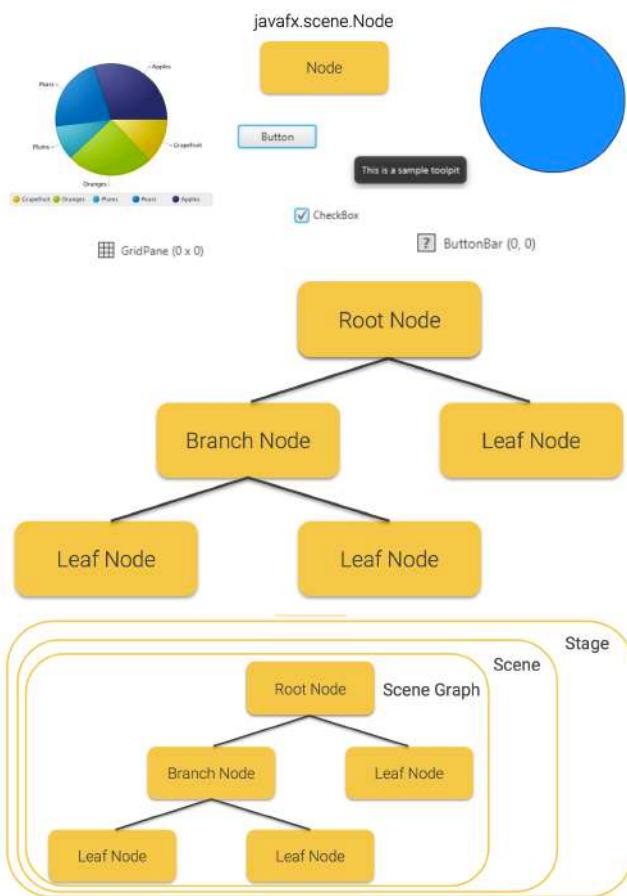
JavaFX proporciona una API completa con un amplio conjunto de clases e interfaces para crear aplicaciones GUI con gráficos enriquecidos. En la captura podemos ver algunos de los paquetes más destacados de esta API.

Podemos encontrar el listado completo en [docs.oracle.com](http://docs.oracle.com). JavaFX Application contiene el conjunto de clases y métodos responsables del ciclo de vida de una aplicación JavaFX, como por ejemplo el método Start de la clase Application.

JavaFX Animation contiene clases para dar funcionalidad a las transiciones entre elementos gráficos tales como Fill, Fade, Rotate, Scale o Transition. JavaFX Event contiene clases y interfaces para lanzar y capturar eventos. JavaFX CSS contiene las clases que nos permiten personalizar las aplicaciones gráficas Java utilizando hojas de estilo CSS. JavaFX Geometry contiene clases para definir y manejar objetos 2D. El paquete JavaFX Stage incluye los contenedores de más alto nivel, como es el propio Stage.

JavaFX Send provee clases e interfaces para soportar el modo gráfico. Incluye también los subpaquetes Canvas, Chart, Layout o Web, entre otros.

## Nodes



Llamamos nodos a todos los elementos gráficos que podemos utilizar en JavaFX.

Controles comunes como el Pattern, Checkbox, Figuras, como por ejemplo un círculo, contenedores que no siempre se ven pero muy útiles y necesarios como el GridPane o el ButtonBar, o otros elementos sencillos como un ToolTip, o más complejos como un gráfico. Todos los nodos son subclases de la clase `JavaFX.scene.Node`.

JavaFX contempla dos tipos de nodos, los nodos rama y los nodos hoja. Los nodos rama, también conocidos como parent nodes, pueden contener otros nodos hijo, mientras que los nodos hoja no pueden contener otros nodos.

De este modo, tenemos los nodos organizados en forma de árbol. El primer nodo del árbol se conoce como nodo raíz.

Esta estructura jerárquica en forma de árbol se conoce como Scene Graph.

Todos sus nodos están incluidos en una escena, que de este modo contendrá cualquier elemento gráfico que queramos mostrar en la aplicación. Las escenas deben estar siempre incluidas dentro de un Stage, que es el contenedor principal o elemento más exterior de la aplicación, típicamente una ventana.

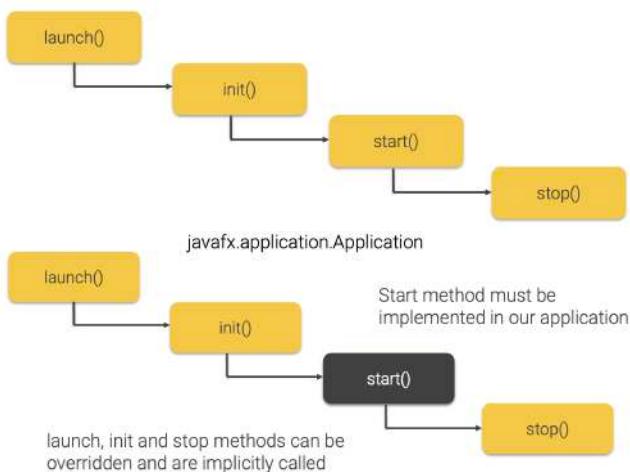
## JavaFX Application



Toda aplicación JavaFX tendrá como mínimo un Stage, que se crea al inicializar la aplicación. Un Stage puede mostrar una sola escena a la vez, pero es posible cambiar de escena tantas veces como sea necesario, por ejemplo, para pasar de un menú inicial a una escena de selección o mostrar cualquier otro tipo de controles. Si la aplicación lo requiere, se pueden crear ventanas adicionales que tendrán su propio stage.

Por ejemplo, para una ventana de diálogo o para iniciar un Wizard.

## JavaFX Application Lifecycle



El ciclo de vida de una aplicación JavaFX empieza con la llamada al método `launch`, que se puede realizar desde el método `main` de nuestra aplicación. Acto seguido, el método `init` inicializa la aplicación, al que seguirá el método `start` desde donde se inicializan los objetos `stage` y `scene`, y nuestra aplicación empezará a ser visible.

Finalmente, el método `stop` es llamado cuando se cierra la aplicación. Todos estos métodos pertenecen a la clase `Application` del paquete `javafx.application`. En nuestra aplicación, solo debemos implementar obligatoriamente el método `Start`.

Launch y Init son llamados automáticamente al iniciar la aplicación, y Stop al cerrarse la última ventana, aunque los tres métodos pueden ser sobreescritos y/o llamados explícitamente.

## Ejemplo DEMO sencillo

Veamos un proyecto sencillo de JavaFX en IntelliJ, donde veremos los componentes básicos de una aplicación con interfaz gráfica. Empecemos creando un nuevo proyecto de JavaFX. Utilizaremos la plantilla de JavaFX. Le pondremos DemoJavaFX. En este proyecto nos encontramos ya creado un ejemplo funcional de interfaz gráfica. Bueno, vemos además que se están cargando distintos componentes puesto que esto es un proyecto Maven, es la plantilla que nos ofrece JetBrains.

En este proyecto nos encontramos ya creado un ejemplo funcional de interfaz gráfica. Tenemos un fichero llamado `GeoApplication` en este fichero encontramos que la clase extiende a `Application` del paquete `javafx.application`. En esta clase nos encontramos el método `Start` imprescindible en toda aplicación JavaFX. Este método recibe un objeto `Stage` que podemos considerar el `primary Stage` y este objeto lo configuramos en las siguientes líneas para pasárselo como escena a este `Stage` y con el imprescindible método `Show` mostrarlo. Sin este método `Show` no podríamos mostrar nuestra interfaz gráfica.

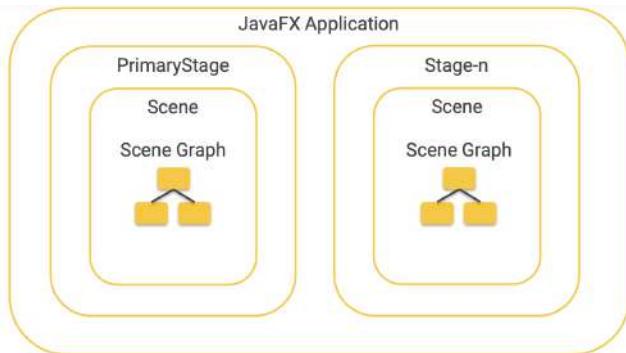
Un poco más abajo, tenemos el método `main` que está invocando al método `launch`. Sin este método `main` la aplicación funcionaría igualmente. Podemos ejecutar la aplicación con interfaz gráfica y si elimináramos este método `main` veríamos que la aplicación se ejecuta, se muestra igualmente. Lo que podemos añadir también son los métodos `init` y `stop`. La implementación de estos dos métodos nos podría ser útil para inicializar el estado de nuestra aplicación o cerrar recursos cuando nuestra aplicación termine.

Veamos que el método `Init` se ha ejecutado y al cerrar la aplicación se ha ejecutado el método `Stop`. Aparte de este `HelloApplication` encontramos un controlador, un fichero controlador. En él vemos código que sirve para gestionar las acciones que el usuario hace en la interfaz gráfica y tenemos otro fichero con extensión `fxml` que es donde está definida la escena que hemos cargado en la `Primary Stage`. Esta escena puede ser modificada con una interfaz gráfica llamada `SceneBuilder`. Por ejemplo, podríamos añadirle un botón y vemos que ahora hemos modificado la interfaz gráfica y aparece este botón.

## FIRST GUI PROJECT

## JAVAFX APPLICATION

## Java Application Structure



Veamos en primer lugar la estructura de una aplicación JavaFX. Esta tendrá como mínimo un Stage, pero puede tener más.

Los stages son las ventanas donde se desarrollará la aplicación. Cada stage tendrá una escena asociada, donde se dispondrá el scene graph, es decir, los elementos gráficos de la aplicación.

## JavaFX Application Code

```
import javafx.application.Application;
public class Main extends Application {
```

Primary launch class extending Application

Veamos ahora paso a paso el código que conforma una aplicación JavaFX. En primer lugar, necesitamos crear la clase principal como subclase de Application. Pero este código todavía no compila, puesto que necesitamos implementar obligatoriamente el método abstracto Start de la superclase Application.

```
import javafx.application.Application;
import javafx.stage.Stage;
public class Main extends Application {
 @Override
 public void start(Stage primaryStage) throws Exception{
 primaryStage.setTitle("Hello World");
 primaryStage.show();
 }
}
```

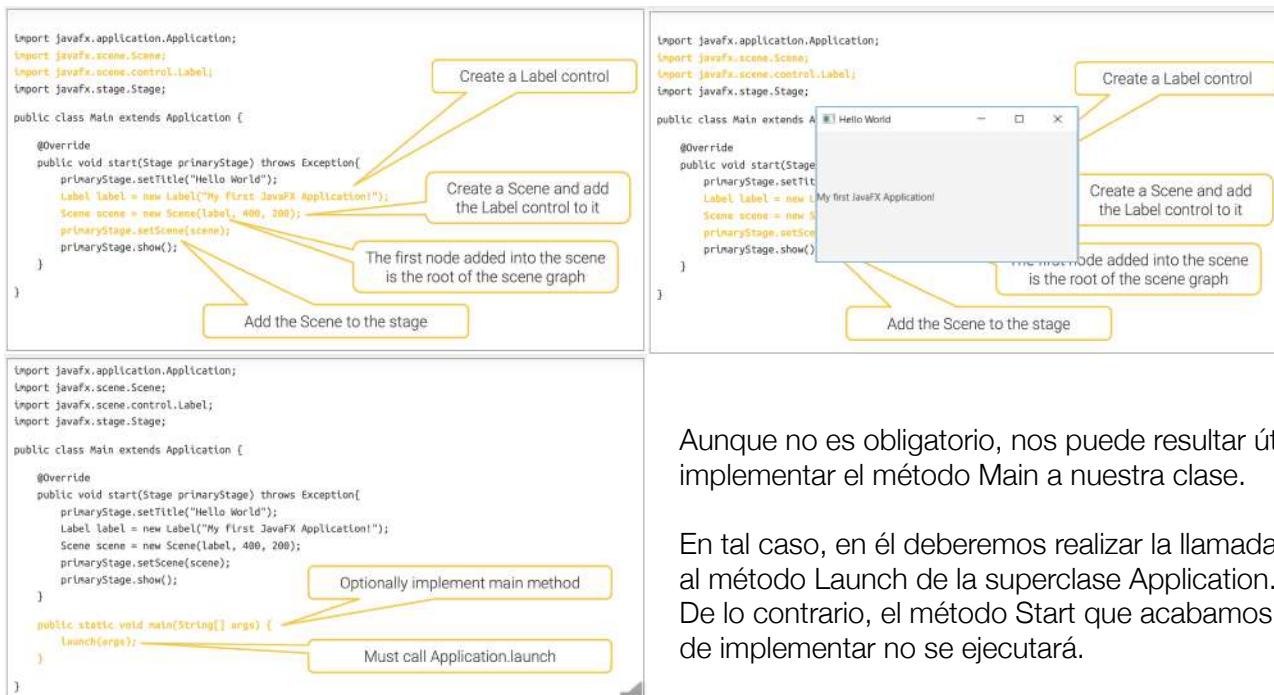
Implement the abstract method start()

Call the method show to display the Stage



El método Start recibe como parámetro un Stage, que como sabemos, representa la ventana de nuestra aplicación. Al Stage podemos, por ejemplo, añadirle un título, que se mostrará como título de la ventana. Pero muy importante es la llamada al método Show para que se muestre la ventana al iniciar el programa. Ahora la aplicación ya compila. Si la iniciamos, podemos ver una ventana vacía.

Podemos añadir contenido gráfico a nuestra ventana, como por ejemplo un control label. Para ello debemos crear una escena y añadirle los controles deseados. Este primer parámetro del constructor de la clase Scene es el elemento raíz del Scene Graph. En el constructor también podemos especificar las dimensiones de la escena. Por último, añadimos la escena a la clase Stage. Este es el resultado del código que hemos visto.



## FXML

### FXML Features

```
<?import javafx.scene.control.Label?>
<Label text="Hello, World!"/>
```

Con JavaFX podemos utilizar FXML. Se trata de un lenguaje de marcas para interfaces gráficas de usuario, desarrollado por Oracle expresamente para el proyecto JavaFX.

Este lenguaje sigue el estándar XML. Nos permite separar la definición del layout del resto de código de nuestra aplicación JavaFX.



En la captura vemos la misma definición de interfaz expresada en JavaFX y en FXML, podemos ver que la estructura jerárquica del lenguaje de marcas FXML hace más fácil la compresión de la interfaz y por lo tanto trabajar con ella añadiendo o modificando componentes de forma más sencilla. Entonces, ¿cuándo debo utilizar FXML y cuándo debo utilizar JavaFX? Por regla general, usaremos FXML para crear nuestras GUIs, porque el rendimiento es mejor, y solamente usaremos JavaFX cuando en tiempo de ejecución necesitemos modificar partes de una escena.

```
Import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class Main extends Application {

 @Override
 public void start(Stage primaryStage) throws Exception{
 primaryStage.setTitle("Hello World");
 Label label = new Label("My first JavaFX Application!");
 Scene scene = new Scene(label, 400, 200);
 primaryStage.setScene(scene);
 primaryStage.show();
 }

 public static void main(String[] args) {
 launch(args);
 }
}
```

Regresando al ejemplo de código visto anteriormente, podemos modificar la aplicación para que utilice FXML en la definición de la escena, en lugar de hacerlo con JavaFX.

The screenshot shows the transition from a direct Java-based approach to using FXMLLoader. It includes four panels: 
 1. Main.java: Initial code creating a Stage and adding a Label directly.
 2. myScene.fxml: The FXML file containing a single label with the text "My first JavaFX Application!".
 3. Main.java: Updated code using FXMLLoader to load the FXML file.
 4. myScene.fxml: The same FXML file as before.

```

Main.java
...
@Override
public void start(Stage primaryStage) throws Exception{
 primaryStage.setTitle("Hello World");
 Label label = new Label("My first JavaFX Application!");
 Scene scene = new Scene(label, 400, 200);
 primaryStage.setScene(scene);
 primaryStage.show();
}

myScene.fxml
<?xml version="1.0" encoding="UTF-8"?>
<Label text="My first JavaFX Application!" />

Main.java
...
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
...
@Override
public void start(Stage primaryStage) throws Exception{
 primaryStage.setTitle("Hello World");
 Parent root = FXMLLoader.load(getClass().getResource("myScene.fxml"));
 Scene scene = new Scene(root, 400, 200);
 primaryStage.setScene(scene);
 primaryStage.show();
}
...

myScene.fxml
<?xml version="1.0" encoding="UTF-8"?>
<Label text="My first JavaFX Application!" />
```

Para ello trabajaremos con un nuevo fichero con extensión.fxml, donde, con lenguaje FXML, definimos el label. Ahora podemos eliminar el label definido en el fichero main.java.

Y en su lugar, utilizando la clase FXML Loader, cargaremos nuestro fichero en un objeto de la clase parent, puesto que el constructor de la escena así lo requiere.

The screenshot shows the final state of Main.java and a screenshot of the application window. The window title is "Hello World" and the label text is "My first JavaFX Application!".

```

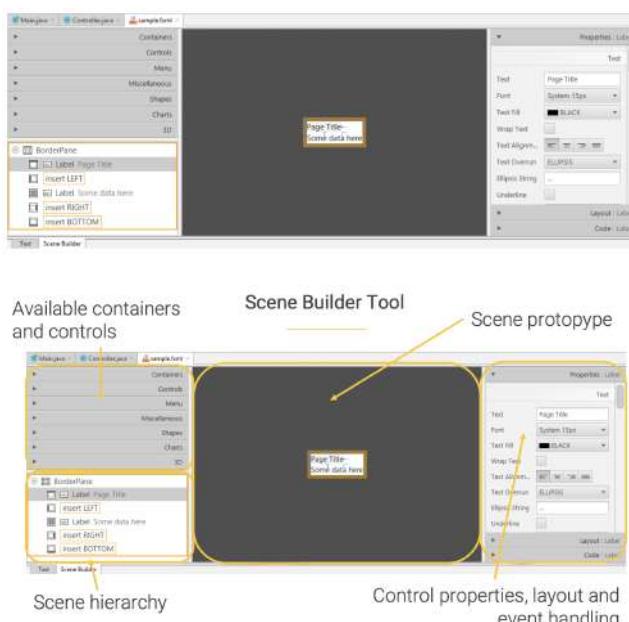
Main.java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.fxml.FXMLLoader;
import javafx.scene.Node;
import javafx.stage.Stage;

public class Main extends Application {
 @Override
 public void start(Stage primaryStage) throws Exception{
 primaryStage.setTitle("Hello World");
 Node root = FXMLLoader.load(getClass().getResource("sample.fxml"));
 Scene scene = new Scene(root, 400, 200);
 primaryStage.setScene(scene);
 primaryStage.show();
 }

 public static void main(String[] args) {
 launch(args);
 }
}
```

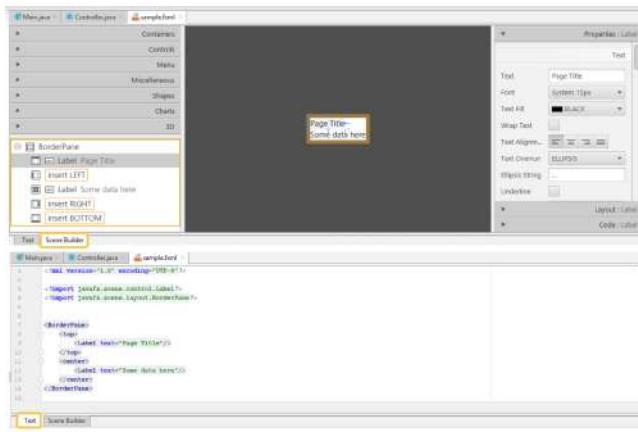
Así queda finalmente nuestro fichero main.java. Si ejecutamos la aplicación, seguimos viendo lo mismo que antes de implementar la escena en FXML. El texto de nuestro label pegado a la izquierda de la ventana.

### Scene Builder Tool



Para diseñar nuestras GUIs en FXML nos puede ser muy útil la herramienta Scene Builder.

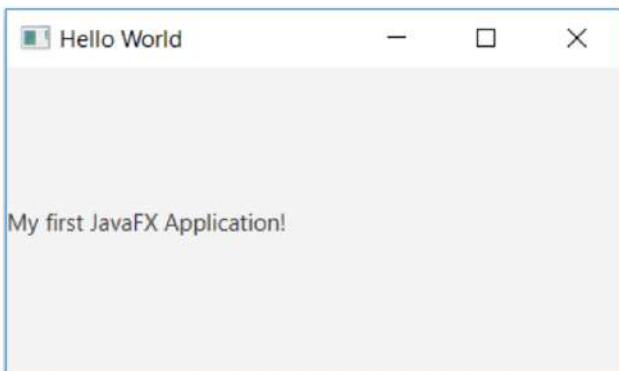
En esta captura podemos ver la interfaz de la herramienta Scene Builder, donde se dispone, en la parte central, el prototipo de nuestra escena. En la parte superior derecha, los controles y contenedores disponibles. Justo debajo, encontramos un esquema de la jerarquía de la escena, y ocupando el panel derecho, tenemos el editor de propiedades, layout y eventos del control que tengamos seleccionado en el prototipo central.



Siempre podemos intercambiar la vista de SceneBuilder a código o viceversa, para trabajar de la forma que mejor nos convenga.

## LAYOUTS

### LAYOUTS



En esta ventana vemos el texto de nuestro label pegado a la izquierda de la ventana, seguramente pensaremos que quedaría un poco mejor si estuviera desplazado a la izquierda o incluso centrado. Para disponer los controles en nuestra escena de una forma ordenada debemos utilizar layouts.

### Layout features



Un layout es un componente de los llamados contenedores, es decir, que puede contener otros componentes dentro de él. Asimismo controla el layout de los componentes que contiene.

Todo layout hereda de la superclase Pane. Al igual que los otros componentes JavaFX, deben formar parte del scene graph de una escena para ser visible. En la captura vemos las secciones de un Layout Border Pane.

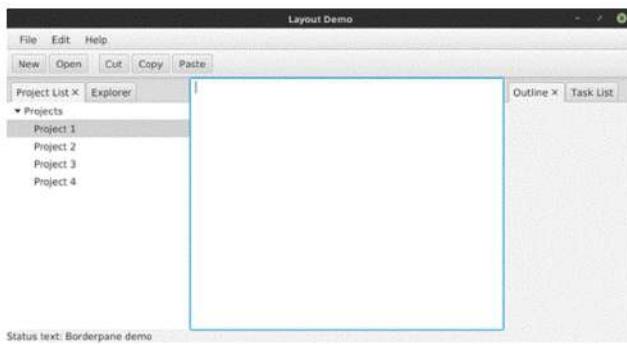
### Ejemplo DEMO

Esta aplicación muestra el contenido de la escena totalmente pegado a la izquierda del Stage. Para que el contenido en nuestra escena se muestre centrado debemos utilizar Layouts. Y esto lo hacemos en el fichero MyScene. Desplegamos los contenedores, que son los que deciden los layouts, y podremos, por ejemplo, coger el FlowPane. ¿Qué nos pasa ahora? Que nosotros teníamos

definido como raíz del, porque se ha definido de forma automática por defecto el primer nodo que hemos añadido a nuestro Scene Graph es la raíz y nosotros ahora lo que queremos como raíz es el FlowPane porque queremos anidar el Label dentro del FlowPane y no nos deja.

Pues lo que haremos es eliminarlo. Se elimina todo lo que es el Scene Graph y volvemos a añadirlo. Tenemos aquí el label, lo ponemos dentro del flowpaint y tendremos que volver a añadirle el texto. Ahora vemos que tenemos igualmente arriba a la izquierda pegado nuestro label pero, y fijaros donde lo hacemos, en el control contenedor, le decimos que queremos que los nodos dentro estén centrados. Y ahora si ejecutamos otra vez nuestra aplicación ya vemos el contenido centrado.

JavaFx.scene.layout.BorderPane



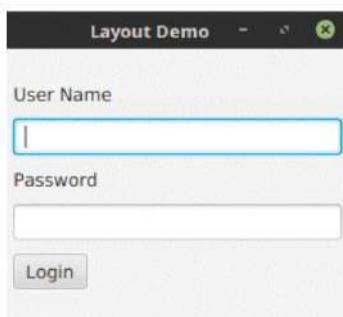
El Border Pane nos permite distribuir controles en 5 secciones distintas.

JavaFx.scene.layout.HBox



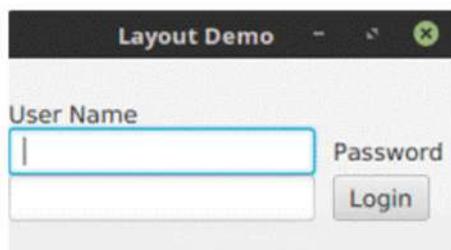
Otros layouts muy comunes son el HBox, donde podemos situar controles a lo largo de una única fila horizontal.

JavaFx.scene.layout.VBox



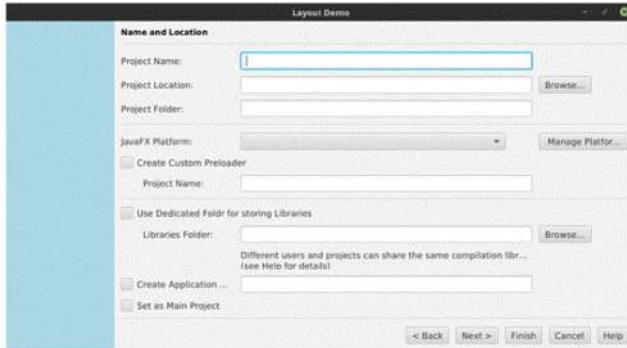
En VBox podemos distribuir los controles en vertical a lo largo de una única columna.

JavaFx.scene.layout.FlowPane



Si utilizamos un FlowPaintLayout, podremos contar con varias filas y columnas, cuyo contenido se reordena según las medidas de la ventana.

## JavaFx.scene.layout.GridPane



El GridPaint permite disponer los controles dentro de un grid flexible, donde podemos especificar exactamente en qué celda ubicar nuestro control. Es un control idóneo para la creación de formularios.

`<?xml version="1.0" encoding="UTF-8"?>`

```
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.FlowPane?>
```

The container stands as Parent

```
<FlowPane alignment="CENTER">
 <children>
 <Label text="My first JavaFX Application!" />
 </children>
</FlowPane>
```

The Parent sets the arrangement of the nested controls

Controls inside containers are Children

`<?xml version="1.0" encoding="UTF-8"?>`

```
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.FlowPane?>
```

Container stands as Parent

```
<FlowPane alignment="CENTER">
 <children>
 <Label text="M" />
 </children>
</FlowPane>
```

Arrangement controls

Controls inside containers are Children

Regresando de nuevo al ejemplo, podemos ubicar el control label que ya teníamos dentro de un contenedor FlowPaint y especificar que su contenido se muestre centrado. Ahora el componente contenedor FlowPaint ejerce de padre de los componentes hijo, ubicados dentro de él, en nuestro caso, del label. Ahora nuestro label se visualiza en el centro de la escena.

`<BorderPane nodeOrientation="LEFT_TO_RIGHT" prefHeight="400.0" prefWidth="600.0" xmlns:fx="http://javafx.com/fxml/1">`

```
<center>
 <FlowPane alignment="CENTER" prefHeight="200.0" prefWidth="200.0" BorderPane.alignment="CENTER">
 <children>
 <Label text="My First JavaFX Application!" />
 </children>
 </FlowPane>
</center>
<left>
 <VBox alignment="CENTER" prefHeight="317.0" prefWidth="149.0" spacing="20.0" BorderPane.alignment="CENTER">
 <children>
 <Button l...>
 <Button l...>
 </children>
 </VBox>
</left>
<top>
 <HBox alignment="CENTER" prefHeight="47.0" prefWidth="600.0" BorderPane.alignment="CENTER">
 <children>
 <Label text="My First JavaFX Application!" />
 </children>
 </HBox>
</top>
</BorderPane>
```

`<BorderPane nodeOrientation="LEFT_TO_RIGHT" prefHeight="400.0" prefWidth="600.0" xmlns:fx="http://javafx.com/fxml/1">`

```
<center>
 <FlowPane alignment="CENTER" prefHeight="200.0" prefWidth="200.0" BorderPane.alignment="CENTER">
 <children>
 <Label text="My First JavaFX Application!" />
 </children>
 </FlowPane>
</center>
<left>
 <VBox alignment="CENTER" prefHeight="317.0" prefWidth="149.0" spacing="20.0" BorderPane.alignment="CENTER">
 <children>
 <Button l...>
 <Button l...>
 </children>
 </VBox>
</left>
<top>
 <HBox alignment="CENTER" prefHeight="47.0" prefWidth="600.0" BorderPane.alignment="CENTER">
 <children>
 <Label text="My First JavaFX Application!" />
 </children>
 </HBox>
</top>
</BorderPane>
```

En este código hemos añadido más contenedores y controles en el fichero myscene.fxml para hacer nuestra aplicación un poco más interesante.

En este caso la raíz de nuestro scene graph es un contenedor border pane, del cual utilizaremos los paneles central izquierdo y superior. En el panel central hemos añadido un FlowPane, con dos figuras geométricas. En el panel izquierdo un VBox, con dos botones dentro de él. Y en el panel superior un HBox, donde hemos ubicado únicamente el label que ya teníamos.

Aunque solo queramos colocar un control, es conveniente hacerlo siempre dentro de un contenedor para poder controlar mejor su posición. Esta es ahora la apariencia de nuestra aplicación. En ella, hemos utilizado la combinación de varios layouts.

Normalmente, una ventana de una interfaz gráfica es la combinación de diferentes contenedores.

## DEMO

Lo que podríamos hacer ahora es jugar un poco más con los layouts y crear una interface gráfica más compleja. Vamos a hacer esto un poco más grande, así las vemos todas o la mayoría. Entonces volveremos a eliminarlo todo por lo que hemos hablando antes de la raíz del scene graph. Ahora queremos que la raíz de nuestro scene graph sea un Border Pane. Luego dentro del Border Pane tenemos varios paneles. El top estaría aquí arriba. Left... Vamos, se definen bastante bien.

```
public void start(Stage primaryStage) throws Exception {
 Label label = new Label("My First JavaFX Application");
 Circle circle = new Circle();
 circle.setStroke(Color.BLACK);
 circle.setStrokeType(StrokeType.INSIDE);
 circle.setFill(Color.DODGERBLUE);
 circle.setRadius(100);
 Polygon triangle = new Polygon();
 triangle.setStroke(Color.BLACK);
 triangle.setStrokeType(StrokeType.INSIDE);
 triangle.setFill(Color.DODGERBLUE);
 triangle.getPoints().addAll(new Double[]{-50.0,40.0,50.0, -40.0,0.0,-60.0 });
 Button button1 = new Button("Button");
 Button button2 = new Button("Button");
 ...
}
```



```
...
FlowPane flowpane = new FlowPane();
flowpane.setAlignment(Pos.CENTER);
flowpane.setPrefHeight(200);
flowpane.setPrefWidth(200);
flowpane.getChildren().addAll(triangle,circle);
HBox hbox = new HBox();
hbox.setAlignment(Pos.CENTER);
hbox.setPrefHeight(100);
hbox.setPrefWidth(200);
hbox.getChildren().addAll(label);
VBox vbox = new VBox();
vbox.setAlignment(Pos.CENTER);
vbox.setPrefHeight(200);
vbox.setPrefWidth(100);
vbox.setSpacing(40);
vbox.getChildren().addAll(button1,button2);
...
}
```



```
...
BorderPane borderPane = new BorderPane();
borderPane.setPrefHeight(400);
borderPane.setPrefWidth(600);
borderPane.setTop(hbox);
borderPane.setLeft(vbox);
borderPane.setCenter(flowpane);
Scene scene = new Scene(borderPane);
primaryStage.setTitle("Hello world");
primaryStage.setScene(scene);
primaryStage.show();
}
```



Y aquí lo que vamos a hacer es añadir más contenedores dentro de cada una de estas secciones que quiera utilizar. Por ejemplo, un HBox en la parte superior, una VBOX en el panel de la izquierda y en el panel central pues le vamos a poner un un flowpane.

Entonces aquí es donde vamos a ponerle, y ahora ya cerramos los contenedores, abrimos los controles, nos vamos a buscar el label que queríamos y lo ponemos aquí arriba. Y aquí, en el HBox, le decimos que lo queremos centrado.

Entonces, aquí vemos lo que sería la jerarquía de nuestro Scene Graph. En el VBox, aquí a la izquierda, le queremos añadir unos botones. Le añadimos dos botones que también vamos a querer que estén centrados. Y aparte de centrados, queremos que estén un poco separados. Y lo que sería en la parte central, le vamos a poner un par de formas.

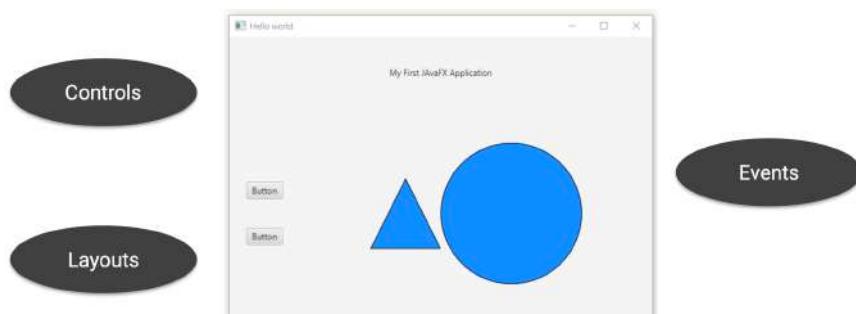
Y si ahora lo ejecutamo ésto se ha quedado un poco pequeño y es por culpa de que nosotros antes le hemos especificado aquí las medidas.

Pero ahora esto ya no es necesario. Y si ejecutamos, veremos que ahora la ventana coge las medidas definidas en el fichero MySceneFXML.

## GUI EVENTS

## GUI EVENTS

Main aspects of GUI



Una interfaz gráfica por simple o compleja que sea debe considerar 3 aspectos, los controles que son los elementos visuales que el usuario ve y utiliza, los layouts que definen cómo se deben organizar estos controles y qué aspecto deben tener, y el tercer elemento imprescindible son los eventos, que representan las cosas que pasan o afectan a la aplicación.



Cuando un usuario interactúa con los controles de una interfaz gráfica, se generan eventos. Los eventos también pueden ser generados por el propio sistema o por otras causas, podemos entender los eventos como acciones o circunstancias que son reconocidas por la aplicación.

### Event types

Foreground events
<ul style="list-style-type: none"> <li>Require the direct user interaction.</li> <li>Consequences of interplay with the graphical components.</li> <li>Clicking on a button, moving the mouse, key press,...</li> </ul>
Background events
<ul style="list-style-type: none"> <li>Require the interaction of a backend user</li> <li>Not directly generated by user interplay with the graphical components.</li> <li>OS interruptions, timer expiry, operation completion,...</li> </ul>

En general, existen dos tipos de eventos, los eventos de primer plano, que requieren la interacción del usuario, es decir, son generados como consecuencia directa de la utilización de los componentes gráficos, por ejemplo, pulsación de un botón, el movimiento del ratón, utilizar el teclado o seleccionar elementos en una lista.

Y los eventos de fondo, que son generados por un elemento ajeno a la interfaz gráfica, como por ejemplo, un timer expirado, una operación que se ha completado, etc.

### Events in JavaFX

En JavaFX, todos los eventos se extienden de la clase Event del paquete JavaFX Event. Algunos ejemplos de estos eventos comunes en JavaFX son MouseEvent, WindowsEvent, KeyEvent, TrackEvent.

mouse clicked	mouse pressed	window hiding	window shown
MouseEvent		WindowEvent	
mouse entered target		window hidden	
key pressed	key released	drag entered	drag dropped
KeyEvent		DragEvent	
key typed		drag entered target	

Estos eventos se generan como consecuencia de acciones realizadas por el ratón o el teclado, por una ventana o al arrastrar un control. En la captura vemos algunos ejemplos de estas acciones. Son clics o movimientos de ratón, esconder o mostrar una ventana, presionar o soltar una tecla o arrastrar un control dentro de otro.

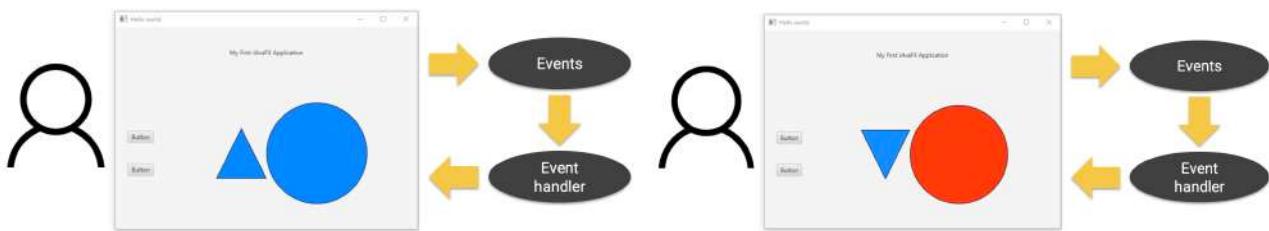
### JavaFX event properties



Todos los eventos tienen tres propiedades, source, type y target. Source, Type y Target. Source se refiere al elemento que ha ocasionado el evento, Type, la manera como ha ocurrido el evento, y Target representa al nodo sobre el que se ha producido la acción.

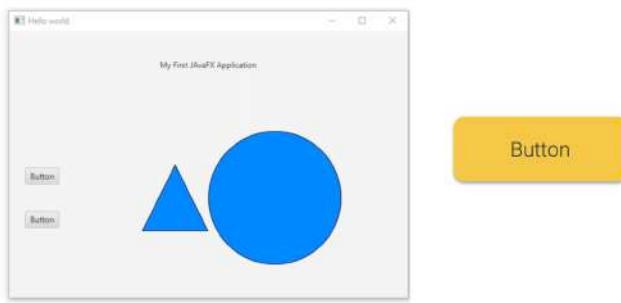
En el ejemplo que vemos, el origen sería el ratón, el tipo sería clic de ratón, y el Target sería el botón sobre el que se ha presionado.

## EVENT HANDLING



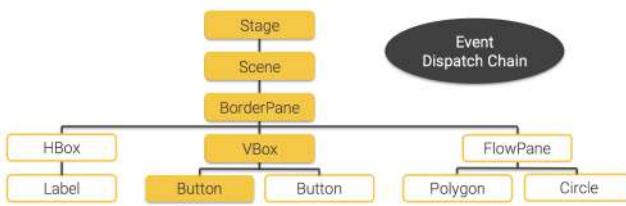
Sabemos que cuando un usuario utiliza los controles de una interfaz gráfica, por ejemplo presionando botones, se generan eventos. Si estos eventos no son gestionados, nada ocurre. Para aprovechar la interacción del usuario, debemos capturar los eventos que nos interesen, y si es el caso, realizar las acciones requeridas. Como por ejemplo, como vemos en la imagen, cambiando propiedades de algunos controles.

## Event Delivery Process: Target selection



Podemos capturar los eventos desde diferentes sitios y en diferentes momentos según nos interese, pero para poder decidir cuándo y cómo es mejor hacerlo, debemos conocer cómo se propagan los eventos. Cuando se pulsa el botón de nuestra aplicación de ejemplo, el sistema inicia el primer paso de la propagación de un evento, la determinación del nodo donde se ha originado, en nuestro caso, un button.

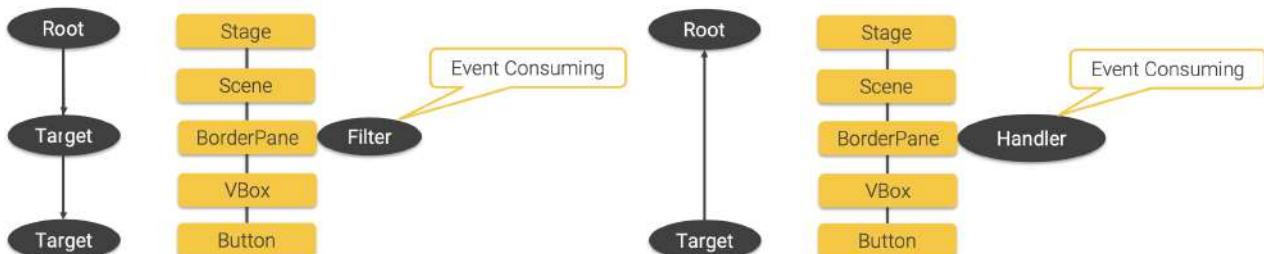
## Event Delivery Process: Route construction



Este botón se encuentra dentro de la jerarquía de nodos de la interfaz gráfica. El segundo paso de propagación de un evento es la construcción de una ruta desde el target hasta el stage.

A esta ruta se la conoce como cadena de propagación del evento, en inglés, Event Dispatch Chain.

## Event Delivery Process: Event Capturing



A continuación, se produce la captura del evento. El evento viaja de nodo a nodo a través de la cadena de propagación desde su raíz, que siempre será el stage, hasta el nodo target. Los nodos pueden ejecutar acciones como consecuencia del paso de un evento.

Para hacerlo, deben registrar un filtro para ese evento. Si esto sucede, se dice que el evento se consume. Como consecuencia de consumir un evento, la propagación del mismo se para, y el nodo que tiene registrado el filtro pasa a ser el nuevo target.

#### Event Delivery Process: Event Building

Después de que un evento haya llegado a su target y se hayan ejecutado todos los filtros registrados si los hubiera, el evento empieza un camino de vuelta hasta el root. En este camino, el evento también puede ser consumido, si algún nodo tiene registrado un Handler. Si ningún nodo consume el evento, se llega a la raíz y se finaliza el proceso.

#### Event handling methods

A la práctica nos encontramos cuatro métodos distintos para gestionar la captura de un evento, usando filtros, usando Handlers, con los métodos de conveniencia o bien con el método de referencia.

A continuación veremos cómo se implementan cada uno de ellos.

### DEMO

#### Demo 1

Veamos primero cómo capturar eventos en un proyecto que utiliza ficheros fxml. En este caso el método que gestiona la captura del evento lo escribiremos en un controlador, el que tenemos aquí arriba. Pero primero veamos cómo se linka este controlador con el fichero fxml.

Aquí lo vemos. Vayamos a escribir el método. Añadamos los imports que faltan. Y vemos que tampoco reconoce la variable circle. Esto es porque aquí no está definida. Tendremos que crearla de nuevo. Arreglemos también esto. Y muy importante lo que vamos a hacer ahora que es escribir la etiqueta fxml, esto lo que hace es linkarnos esta variable y este método con el fichero fxml. En el fichero fxml debemos terminar de linkar el objeto Circle que hemos creado en el código JavaFX con el círculo de nuestra interfaz gráfica. Así mismo también debemos asociar el método Handler del controlador con el evento click de mouse sobre el botón. Primero creemos la sustitución del círculo en el apartado de código debemos añadir el círculo que ya nos lo propone y luego nos vamos al button y también en su apartado, aquí no es tan importante que le definamos un identificador porque no lo hemos hecho en el fichero Java del controlador, pero lo que sí que es muy importante es irnos a los eventos de mouse y lo que es mouseClicked, asociarlo al método Handler.

Y ahora podemos ejecutar. Y como vemos, también funciona. Y aquí termina este vídeo donde hemos visto cómo capturar eventos con JavaFX.

#### Demo 2

Ahora haremos lo mismo pero en un proyecto que no utiliza ficheros FXML. En el método Start tenemos definida una interfaz gráfica. Y antes de empezar lo que haremos será modular un poco este escenario para tener la ventana más limpia y trabajar de una forma más cómoda. Vemos que la pulsación de los botones no hace nada. A continuación, capturaremos el evento de pulsación del ratón sobre este primer botón para que modifique el color del círculo. Para declarar un evento necesitamos crear un Event Handler. de tipo evento de ratón. Hemos dicho que lo que queríamos era capturar el clic del ratón.

Lo llamaremos Event Handler ChangeCircleColor y escribiremos el código que queremos que ejecute este handler. Cambiar el color del círculo. Ahora debemos registrar el evento a un nodo, en nuestro caso el botón 1. Para registrarlo podemos hacerlo con dos métodos, el addEvent Handler o el addEvent Filter. Sabemos que los filtros pueden consumir el evento en la fase de captura y el Handler lo consumirá en la fase de construcción. Pero ambos nos permitirán hacer lo mismo. Tienen, si vemos, la misma definición. Hay que basarle el tipo de evento que queremos capturar y el Handler que lo va a gestionar. Nosotros queremos capturar un tipo de evento de ratón, concretamente el click. Y el event handler que lo va a gestionar es el que hemos creado arriba.

Arreglamos este código. Y veamos si funciona. Otra manera de registrar un nodo para un evento es utilizando los métodos directos o en inglés convenience methods. Estos métodos siempre empiezan por setOn. Aquí tenemos toda la lista de eventos para los que podemos registrar este nodo. Nosotros queremos el mouseclick. Y como parámetro recibe el event handler que crearemos aquí mismo. Copiamos el código del handler que hemos definido antes y que ahora ya no vamos a necesitar, y tampoco vamos a necesitar registrar el evento. Vemos que también funciona.

#### Event Handling using filter

```
...
Main.java
EventHandler<MouseEvent> ehChangeCircleColor = new EventHandler<MouseEvent>() {
 @Override
 public void handle(MouseEvent event) {
 circle.setFill(Color.RED);
 }
};
button1.addEventFilter(MouseEvent.MOUSE_CLICKED,ehChangeCircleColor);
...
}
```

En este código, vemos un ejemplo de cómo se crea un Event Handler para el tipo de evento MouseEvent, y cómo se implementa el correspondiente método Handle. Luego vemos cómo se registra un filtro en el objeto Button1, para los EventType de tipo click de ratón, pasando al EventHandler que hemos creado anteriormente para que lo gestione.

#### Event handling using handler

```
...
Main.java
EventHandler<MouseEvent> ehChangeCircleColor = new EventHandler<MouseEvent>() {
 @Override
 public void handle(MouseEvent event) {
 circle.setFill(Color.RED);
 }
};
button1.addEventHandler(MouseEvent.MOUSE_CLICKED,ehChangeCircleColor);
...
}
```

En este código vemos cómo hacer lo mismo que anteriormente pero utilizando un Handler en lugar de un filtro.

El código del EventHandler es exactamente el mismo, únicamente cambia el método que registra el evento en el objeto botón, de addEventHandler a AddEventFilter.

#### Event handling using convenience method

```
...
Main.java
button1.setOnMouseClicked(new EventHandler<MouseEvent>() {
 @Override
 public void handle(MouseEvent event) {
 circle.setFill(Color.RED);
 }
});
...
}
```

En este código vemos cómo hacer la misma captura del evento, pero utilizando el método directo o Convenience Method.

Event handling using method reference

```
Controller.java
...
@FXML
public void changeCircleColor(Event e){
 circle.setFill(Color.RED);
}
...

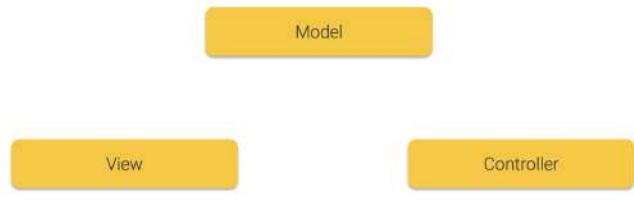
myScene.fxml
...
<Button mnemonicParsing="False" onMouseClicked="#changeCircleColor" text="Button" />
...
```

Y en este código vemos cómo hacer la captura del evento cuando trabajamos con ficheros FXML con el método asociado, definiendo el método en el controlador y asociando el objeto gráfico desde el fichero FXML.

## MVC

### MVC PATTERN

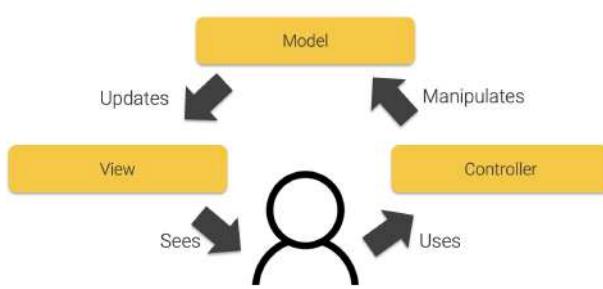
MVC o Modelo Vista Controlador es un patrón de diseño de software que consiste en dividir todos los objetos de una aplicación en 3 áreas y establecer correctamente la comunicación entre ellas.



El objetivo de este patrón de diseño es permitir la reutilización de código y el desarrollo en paralelo de las diferentes partes de una aplicación. Estos tres elementos o áreas son El modelo, que representa la lógica interna del programa, es donde se implementan las reglas de negocio y la estructura de datos, debe ser independiente de la interfaz gráfica.

La vista que representa la interfaz de usuario. El controlador que se encarga de traducir lo que el usuario hace en la vista a llamadas o métodos definidos en el modelo, para que de este modo se ejecute la lógica interna adecuada a la orden dada por el usuario.

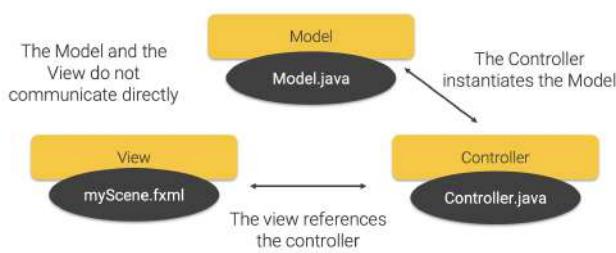
Interactions within the MVC pattern



Aquí podemos ver la comunicación entre los diferentes elementos que intervienen en el patrón MVC. El usuario visualiza la interfaz gráfica. Al utilizarla, realiza acciones que son interpretadas por el controlador. El controlador reacciona a las órdenes del usuario e interacciona con el modelo. En el modelo se realizan las operaciones de cálculo, actualización, consulta, etc que se acaban reflejando en la vista.

## MVC IN JAVAFX

## MVC Workflow



En JavaFX, los modelos y los controladores están implementados en ficheros.java, mientras que las vistas se crean con lenguaje FXML. Las vistas están asociadas con su correspondiente controlador, quien gestiona los eventos y actualiza los componentes de la vista.

El controlador se comunica con el modelo utilizando una instancia de su clase. El modelo y la vista no se comunican directamente.

## DEMO

En esta demo veremos cómo aplicar el patrón modelo vista controlador en proyectos JavaFX. Para ello utilizaremos esta aplicación llamada TicTacToe que simula el juego 3 en raya. Veamos qué contiene. Tenemos el fichero main encargado de mostrar la interfaz gráfica.

Tenemos la vista llamada GameFXML que contiene una representación gráfica del tablero de juego. En un GridPane representamos el tablero y luego tenemos labels para cada una de las casillas que tienen un identificador FXML único. Lo podemos ver aquí. Representa la fila y la columna de cada uno de los labels. Si exploramos el código FXML, vemos que el controlador asociado a esta vista se llama GameController. También vemos que cada uno de los Labels tiene el evento de clic asociado al método PlayTab.

También podemos ver esto en el apartado de los eventos de la vista en el Scene Builder. Vayamos al controlador. Vemos que tiene varios atributos label, que se llaman como cada una de las casillas y efectivamente estos labels están reverenciados mediante el tag @FXML a los controles que hemos visto en la vista. Luego tenemos una variable de tipo Game que se instancia al iniciar el controlador. Este objeto Game representa el modelo. Luego veremos cómo en la clase Game está implementada toda la lógica del juego. En el controlador también tenemos el método PlayTab. Al que hemos visto anteriormente están asociados los eventos clic de ratón de todos los Labels de la vista.

Está marcado también con el tag @fxml para poder ser visto desde la vista y recibe como parámetro el evento. En este método se obtiene del evento el identificador del Label pulsado. De este identificador se extrae fila y columna que luego usamos en la llamada al método makeMove del modelo. Vayamos entonces al modelo a ver qué hace. Vemos que la clase Game, que como hemos dicho representa el modelo, en esta aplicación JavaFX, tiene un atributo que cuenta las jugadas, una matriz de chars donde se almacena el estado del juego y un atributo string para almacenar el ganador. El constructor de la clase inicializa el contador a cero y el tablero con 9 posiciones 3x3. Y aquí tenemos el método makeMove, que hemos visto que se invoca en cada click del ratón sobre un casilla del tablero de la vista. Aquí se comprueba que la casilla esté vacía, y de estarlo, se coloca la ficha que toque, según el número de la jugada, en la correspondiente posición de la matriz Board. Este es el método que se encarga de comprobar que la casilla no esté ocupada. Arroja una excepción de estar ocupada. Luego veremos en el controlador cómo se gestiona esta excepción.

Tenemos un método getter para el tablero. Otro método que comprueba todos los posibles escenarios ganadores con la ayuda del helper method checkWinner y otro getter para obtener el ganador de haberlo. Como hemos visto en esta clase se gestiona la lógica y el estado del juego.

Volvamos donde nos hemos quedado. Tras colocar la ficha en el modelo pedimos el tablero actualizado también al modelo para actualizar la vista en el método updateBoard. Comprobamos si

hay ganador para mostrar un mensaje. En cualquier caso, se termina la ejecución del método a la espera que en la vista se pulsen más casillas que provoquen más eventos. Comentar que hemos visto que MakeMove podía arrojar excepciones. Lo que hacemos aquí es mediante un bloque TryCatch capturar cualquier excepción y mostrarla en un diálogo.

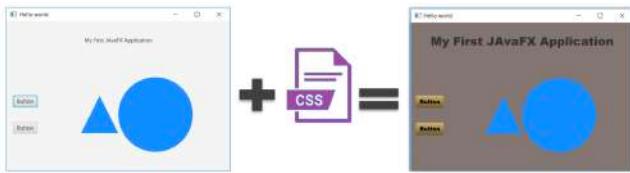
Al igual que hacemos por ejemplo, en el tratamiento de ficheros donde algunos métodos también pueden arrojar excepciones. Ejecutemos. Se abre la vista y podemos empezar a jugar. Si pulsamos dos veces sobre la misma casilla se muestra el mensaje de error esperado que recordemos es generado en el modelo, llega al controlador y éste lo traslada a la vista. Al igual que cada una de las pulsaciones en una casilla, llega al controlador mediante la captura del evento clic de ratón. El controlador llama al modelo para que actualice el estado.

El controlador recibe el estado del juego y actualiza la vista. Finalmente, si hay una combinación ganadora, también hemos visto que el modelo se lo comunica al controlador para mostrarlo en la vista.

## CSS

### CSS IN JAVAFX

#### Cascading Style Sheets



Sabemos que CSS es un lenguaje de diseño ampliamente utilizado en páginas web, puesto que permite separar la presentación del contenido. Podemos hacer uso de esta ventaja también en nuestras aplicaciones JavaFX.

De este modo, podemos definir los estilos de nuestros objetos gráficos fuera del fichero FXML, utilizando un fichero CSS, que nos permitirá poder cambiar la apariencia de nuestra aplicación rápidamente, simplemente cambiando el fichero de estilos.

### DEMO

En esta demo, veremos cómo utilizar hojas de estilo CSS en proyectos Java FX. Para ello, utilizaremos esta aplicación. Veamos qué contiene. Empecemos por el fichero main. Aquí podemos ver cómo se carga una interfaz gráfica desde un recurso FXML llamado MyScene.

Si buscamos y abrimos este fichero, podemos ver los elementos de la interfaz. Desde el panel de propiedades, podemos modificar la apariencia de estos elementos. Por ejemplo, de este label. Cambiaremos su tamaño de fuente a 32 píxeles. Inmediatamente lo vemos reflejado. Si ejecutamos la aplicación, deberíamos verlo igual. Pero podemos hacer lo mismo utilizando una hoja de estilos que, como sabemos, nos aporta mayor flexibilidad. Para ello creamos un fichero CSS. Para identificar que contendrá los archivos de MyScene, le ponemos el mismo nombre.

Y en él escribo el código CSS siguiendo las mismas reglas básicas que utilizaría si creara estilos para una página web. Recordemos, quiero cambiar el tamaño de fuente de este label. Para poder referirme a él, deberé asignarle un ID en el apartado de código. Ahora ya puedo referirme a él en el fichero CSS mediante su identificador utilizando la almohadilla.

Y como voy a utilizar los estilos definidos en este CSS, dejo sin estilos el fichero fxml. El siguiente paso es cargar el recurso CSS que he creado en la escena. Lo hago en el fichero main, utilizando el método `getStyleSheets` y añadiendo el recurso CSS. Una vez hecho, ejecuto y veo los estilos reflejados en el label de la aplicación. Ahora le añado un poco más de estilos al fichero CSS e incluso una imagen de fondo que añado al proyecto. Aquí tenemos la nueva apariencia de nuestra aplicación.

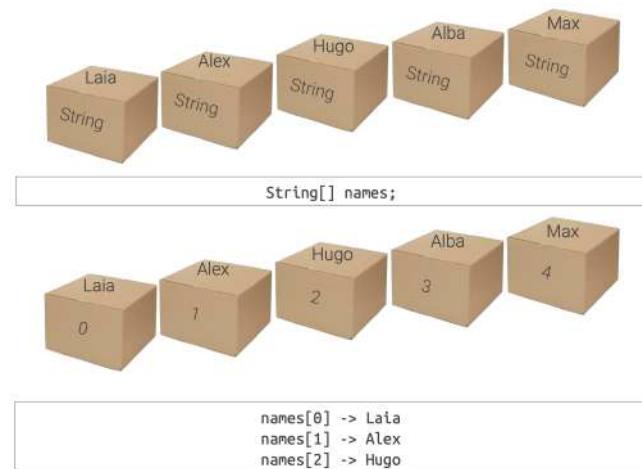
Para terminar, podría crear incluso un segundo fichero con estilos CSS. Aquí tenemos unos nuevos estilos y cambiar de esta forma tan sencilla la apariencia de la aplicación.

# CONTROL DE EXCEPCIONES Y COLECCIONES DE DATOS I

## ARRAYS AND COLLECTIONS

### ARRAYS AND COLLECTIONS

#### Arrays



Sabemos que la mejor manera de almacenar múltiples datos del mismo tipo es usar arrays. Los arrays son un grupo de celdas que almacenan valores, y cada valor se almacena de forma separada, pero el array se trata como si fuera una sola variable.

Cada elemento en el array tiene un índice, un número entero que comienza en cero e indica la posición que ocupa el elemento en el array.

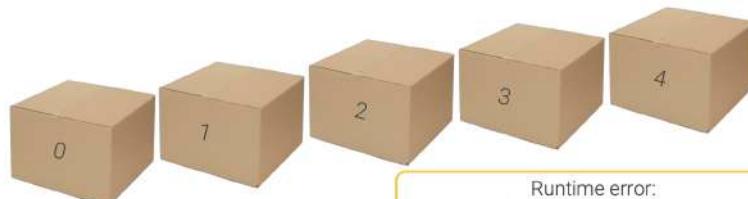
El índice permite acceder a cada elemento del array de forma individual.

#### Array Limitations

```
String[] names = new String[5];
```

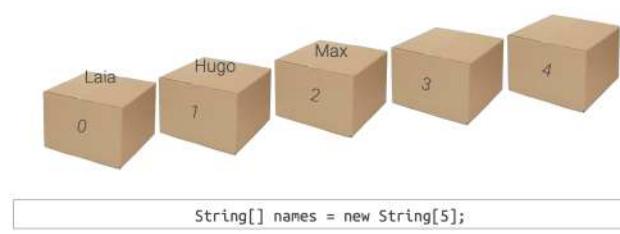
Los arrays tienen algunas limitaciones. La primera es que tenemos que saber el número de elementos que tendrá en el momento de inicializarlo.

Y una vez que hemos inicializado un array con un número específico, no podemos añadir o eliminar elementos.

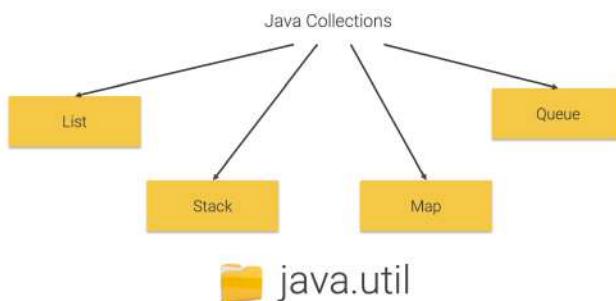


```
String[] names = new String[5];
names[10] = "Matt";
```

Esto significa que si intentamos acceder a un índice que no existe, porque supera el tamaño del array, se producirá un error en tiempo de ejecución y la aplicación se detendrá de forma inesperada.



## Java Collections

`add(E element)``iterator()``remove(E element)`

Otra limitación es que cuando comenzamos a llenar con valores el array no podemos añadir elementos adicionales sino que tendremos que reemplazar alguno de los existentes y cuando eliminemos valores se quedarán los huecos en el array y tendremos que cambiar la posición de los elementos manualmente para eliminar estos huecos.

Estas limitaciones son el motivo por el cual Java introdujo lo que se conoce como colecciones. Las colecciones son simplemente un grupo de clases e interfaces que Java nos ofrece para trabajar con agrupaciones de elementos. Todas ellas se encuentran en el paquete `java.util`. Las colecciones nos permitirán trabajar con agrupaciones de elementos cuyo tamaño sea variable.

`String``Double``Car``boolean`~~`int`~~`Integer``StringBuilder``Book`~~`double`~~

Es importante que entiendas que en una colección podremos almacenar objetos de cualquier tipo, o bien predefinidos por el lenguaje, o bien creados por nosotros, pero nunca tipos primitivos de datos.

## ArrayList: Example of a collection

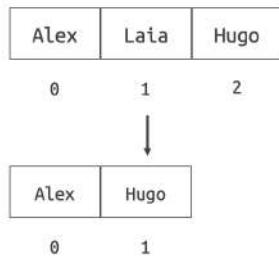
Una de las colecciones más conocidas es `ArrayList`. La veremos en detalle más adelante. Ahora la usaremos como ejemplo de colección Java para esbozar la potencia de usar colecciones en nuestras aplicaciones.

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
```

Alex	Laia	Hugo
0	1	2

Podremos inicializar una colección sin indicar el número de elementos que contendrá. Con la instrucción que ves en la captura hemos creado un `ArrayList` de strings, denominado `Names`, vacío, sin elementos. Para añadir un elemento a una colección, podremos usar el método `Add`, especificando como argumento el elemento a añadir.

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.remove("Laia");
```

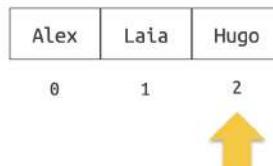


En este momento Names tiene tres elementos, y podríamos seguir añadiendo más usando el método Add. Usando el método Remove, podríamos eliminar de la colección el elemento especificado como argumento, además de reorganizar el resto de elementos para eliminar los huecos. Añadir y eliminar son las operaciones básicas.

Cualquier colección en Java tiene muchas operaciones más que iremos descubriendo.

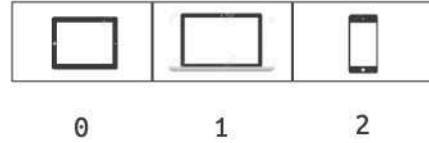
```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");

Iterator iterator = names.iterator();
while (iterator.hasNext()){
 System.out.print(iterator.next()+" ");
}
```



Toda colección dispone de un iterador, esto es un puntero, que con ayuda de un bucle nos permitirá recorrer la colección elemento a elemento. Y recuerda que una colección nos permitirá almacenar objetos de cualquier tipo, o bien predefinidos por el lenguaje, o bien creados por nosotros.

```
ArrayList<Producto> productos= new ArrayList<>();
productos.add(new Producto('Tablet',195.99));
productos.add(new Producto('PC',485));
productos.add(new Producto('Smartphone',95.99));
```



**LISTS****LIST**

```

Module java.base
Package java.util
Interface List<E>

Type Parameters:
E - the type of elements in this list

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Implementing Classes:
AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

```

Una lista en Java permite trabajar con una colección ordenada de elementos, también se conoce como secuencia. Una lista en Java es una interfaz que se comporta de forma muy similar a un array. Mantiene el orden de inserción, permite obtener y modificar los valores de sus elementos a partir de su posición, puede almacenar valores nul y valores duplicados.

La diferencia con respecto a los arrays es que al ser una colección, su tamaño puede cambiar y por tanto, permite añadir y borrar elementos. Puedes consultar la documentación de Oracle sobre esta interfaz en [docs.oracle.com](http://docs.oracle.com).

Some of the List's methods

Comprobarás que dispone de los métodos básicos de cualquier colección Java, como son add, remove, para añadir o eliminar elementos. Y que a estos métodos, la interfaz list, añade otros que permiten manipular sus elementos por la posición que ocupan en la lista, como son contains, para consultar si el elemento especificado está en la lista o no, size, para saber cuántos elementos contiene, add, remove, para añadir o eliminar un elemento dado su índice, get y set, para obtener o actualizar el valor de un elemento dado su índice.

La clase más popular que implementa la interfaz List es ArrayList. Comencemos con ella.

**ARRAYLIST**

```

Module java.base
Package java.util
Class ArrayList<E>

java.lang.Object
 java.util.AbstractCollection<E>
 java.util.AbstractList<E>
 java.util.ArrayList<E>

Type Parameters:
E - the type of elements in this list

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:
AttributeList, RoleList, RoleUnresolvedList

```

Un ArrayList es una clase que implementa la interfaz List. Es como un Array, pero con potentes métodos que permiten gestionar los elementos del array de forma simple.

## Creating an ArrayList

```
ArrayList<String> names = new ArrayList<String>();
```

```
ArrayList<String> names = new ArrayList<String>();
```

```
ArrayList<String> names = new ArrayList<>(); //names is an empty ArrayList
```

Declaration and initialization

```
ArrayList<String> anotherArrayList; //anotherArrayList is null
...
anotherArrayList= new ArrayList<>(); // Now anotherArrayList is an empty ArrayList
```

Initialization

Declaration

```
ArrayList<String> anotherArrayList; //anotherArrayList is null
...
anotherArrayList= new ArrayList<>(); // Now anotherArrayList is an empty ArrayList
```

Is null

Para declarar e inicializar un ArrayList usamos la instrucción que ves en la captura. Cuando creamos un ArrayList hay que informar a Java el tipo de dato que va a almacenar. Esto se indica entre los símbolos mayor que y menor que. En el ejemplo que ves en la captura se ha creado un ArrayList de objetos string.

Observa que la palabra String aparece dos veces, en el lado izquierdo del igual y en el lado derecho del igual. ¿Piensas que la segunda vez que aparece es redundante? Pues así es. Desde Java versión 7 no es necesario indicar el tipo de dato a la derecha, tal como ves en la captura.

Esta secuencia declara un ArrayList de objetos String y lo inicializa en la misma línea. Este ArrayList no tiene elementos. Está vacío.

Podemos también declarar una variable de tipo arraylist, como ves en la captura, y luego inicializarla en otra línea de código.

Mientras esta variable no haya sido inicializada, su valor es null, como la de cualquier otro objeto que no haya sido inicializado.

```
import java.util.ArrayList;

public class Main {
 public static void main(String[] args) {
 ArrayList<String> names = new ArrayList<>();
 }
}
```

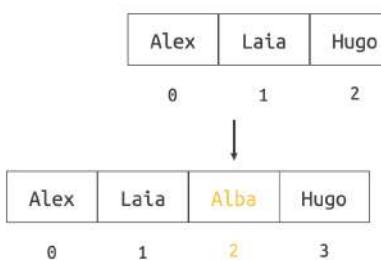
ArrayList pertenece al paquete java.util que no se importa implícitamente como el paquete Java.lang. Por lo tanto, tendremos que añadir la sentencia import para poder utilizar objetos de la clase ArrayList.

### Adding elements to an ArrayList

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
```

Alex	Laia	Hugo
0	1	2

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.add(2, "Alba");
```



Bien, hemos dicho que ArrayList es un Array dinámico, cuyo tamaño puede variar, así que vamos a empezar viendo cómo añadir elementos. Para añadir un elemento a un ArrayList usamos el método add, especificando como argumento el elemento añadir.

Debemos recordar que el tipo de dato del elemento debe coincidir con el tipo de dato que declaraste que iba a contener tu ArrayList. ¿Qué ha ocurrido? Hemos añadido tres elementos, tres objetos de tipo String a nuestro ArrayList, igual que en los Arrays el primer elemento de un ArrayList se almacena en la posición 0.

El método Add puede tener dos argumentos como ves en pantalla. El primer argumento indica en qué posición se quiere añadir el valor especificado como segundo argumento. Cuando un valor se añade a una posición que es ocupada por otro elemento, el elemento se desplaza a la derecha para dejar sitio al nuevo valor añadido.

Cuando se añade un elemento en una posición determinada, lo que hace Java es crear un nuevo array e inserta todos sus elementos en las posiciones siguientes a la posición que se ha especificado.

### NullPointerException

```
ArrayList<String> names;
```

```
names.add("Alex");
```

NullPointerException

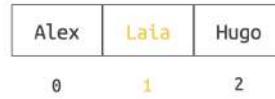
Antes de seguir, fíjate en una cosa importante. Si la variable ArrayList no ha sido inicializada, cuando llamemos a cualquiera de los métodos de la clase ArrayList, se producirá un error en tiempo de ejecución y el programa terminará.

Si no inicializamos la variable, su valor es null, e invocar a cualquiera de sus métodos produce una excepción del tipo `nullPointerException`.

### Accessing elements of an ArrayList

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");

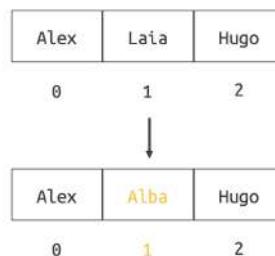
System.out.println(names.get(1)); //Prints Laia
```



Usamos el método `get` para acceder a un elemento en concreto, indicando su posición.

### Modifying the elements of an ArrayList

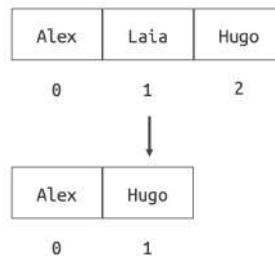
```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.set(1,"Alba");
```



Podemos modificar el valor de un elemento de un `ArrayList` con el método `SET`. Este método tiene dos argumentos. El primer argumento indica la posición a modificar y el segundo argumento el nuevo valor.

### Deleting the elements of an ArrayList

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.remove(1);
```



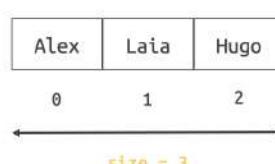
Para eliminar un elemento de un `ArrayList` usamos el método `REMOVE`. Este método tiene un argumento donde especificamos la posición a eliminar. Como parámetro de entrada del método Remove, también podemos especificar el valor a eliminar.

### Loops & ArrayList

```
ArrayList<String> names = new ArrayList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");

for (int i=0; i<names.size(); i++){
 System.out.print(names.get(i)+" ");
}

//Prints
Alex Laia Hugo
```



Usaremos bucles para recorrer los elementos de un `ArrayList`, de forma equivalente a como lo hacíamos con un `Array`. El método `Get` nos permitirá acceder a cada elemento según su posición. Y el método `Size`, que devuelve el número de elementos que tiene la colección, nos permitirá establecer la condición de salida del bucle.

Es el equivalente al length para Arrays.

Un ArrayList permite que lo recorramos de una forma más simple. Es el código que ves en el lado derecho de la pantalla.

```
ArrayList<String> names = ...
```

```
for (int i=0; i<arrays.size(); i++){
 System.out.print(names.get(i)+" ");
}
```

```
//Prints
Alex Laia Hugo
```

```
ArrayList<String> names = ...
```

```
for (String element: names){
 System.out.print(element+" ");
}
```

```
//Prints
Alex Laia Hugo
```

```
ArrayList<String> names = ...
```

```
for (int i=0; i<names.size(); i++){
 System.out.print(names.get(i)+" ");
}
```

```
//Prints
Alex Laia Hugo
```

```
ArrayList<String> names = ...
```

```
for (String element: names){
 System.out.print(element+" ");
}
```

```
//Prints
Alex Laia Hugo
```

Consiste en escribir el tipo de dato de los elementos que contiene el ArrayList, seguido del nombre de la variable, por ejemplo, element, que tomará el valor de cada uno de los elementos de la colección, seguido de dos puntos, y por último, el identificador de la variable de tipo ArrayList.

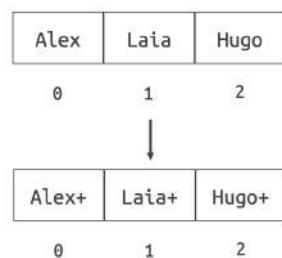
### Advanced Loops & ArrayList

Pero hay otra manera de recorrer los elementos de una ArrayList, que solo deberíamos usar en las situaciones donde al mismo tiempo que estamos recorriendo la colección queremos modificarla para eliminar algún elemento o modificar el valor de algún elemento. Se trata de usar ListIterator.

```
ArrayList<String> names = ...

ListIterator iterator = names.listIterator();

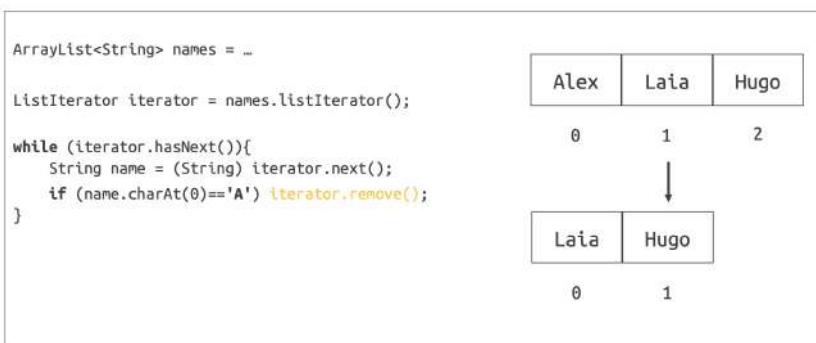
while (iterator.hasNext()){
 String name = (String) iterator.next();
 iterator.set(name+"+");
}
```



En el ejemplo que ves en pantalla, mientras se recorre la colección usando el ListIterator, se actualiza el valor de sus elementos. Para poder usar un iterador, primero tendremos que declararlo llamando al método ListIterator de la colección.

Un iterador es como un puntero. Con el método Next iremos moviendo el puntero al siguiente elemento de la colección.

El método HasNext retornará True siempre que hayan elementos para recorrer. Por este motivo será la condición a utilizar en el bucle. Un ListIterator dispone del método set para actualizar con un nuevo valor el elemento actual. En el ejemplo que ves en pantalla, cada stream de la colección se actualiza introduciendo un símbolo más al final de la cadena de texto.



Un ListIterator también dispone del método remove para eliminar el elemento actual. En el ejemplo que ves en pantalla, si el elemento de la colección comienza por A, se eliminará de la colección. Es recomendable echar un vistazo a la documentación de Oracle sobre la clase ArrayList y su lista completa de métodos.

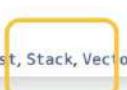
## STACK

**Module** java.base  
**Package** java.util  
**Interface List<E>**

**Type Parameters:**  
E - the type of elements in this list

**All Superinterfaces:**  
Collection<E>, Iterable<E>

**All Known Implementing Classes:**  
AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector



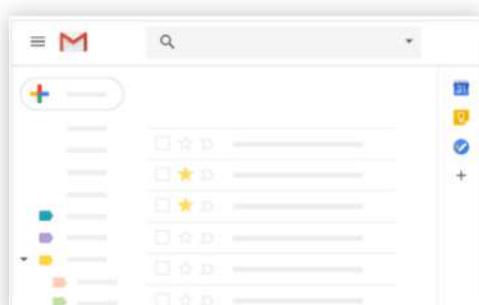
Otra clase popular que implementa la interfaz LIST es STACK.

**Module** java.base  
**Package** java.util  
**Class Stack<E>**

java.lang.Object  
java.util.AbstractCollection<E>  
java.util.AbstractList<E>  
java.util.Vector<E>  
java.util.Stack<E>

**All Implemented Interfaces:**  
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Una stack representa una pila de objetos, LIFO, LASTIN, FIRSTOUT, que significa que el último elemento introducido en la pila será el primer elemento en salir de ella. Como una pila de platos, el último plato que añades será el primer plato que coges. Igual que ArrayList, una stack internamente usa un array para almacenar los elementos.



Un ejemplo de cuando nos puede ser útil emplear una colección de tipo stack sería cuando desarrollamos algo parecido al sistema de correo electrónico. Cuando el servidor de correo electrónico recibe un nuevo email, debe añadir este email arriba en la pila de emails, de forma que el usuario pueda leer el email más reciente primero.

## Creating a Stack

```
Stack<String> newsFeed = new Stack();
```

Para declarar e inicializar una stack usamos la instrucción que ves en pantalla. Igual que con ArrayList debes especificar el tipo de objeto que almacenarás en la colección, en este caso strings, y tendrás que añadir la sentencia import para poder utilizarla.

```
import java.util.Stack;

public class Main {

 public static void main(String[] args) {
 Stack<String> newsFeed = new Stack();
 }
}
```

## Some of the List's methods

La clase Stack, como implementa la interfaz List, dispone de sus métodos de forma equivalente a lo que hemos visto con la clase ArrayList. Y añade los que ves en pantalla para permitir gestionar los elementos como una pila LIFO.

## Adding elements to a Stack

```
Stack<String> newsFeed = new Stack();
newsFeed.push("Morning news");
newsFeed.push("Afternoon news");
newsFeed.push("Evening news");
```

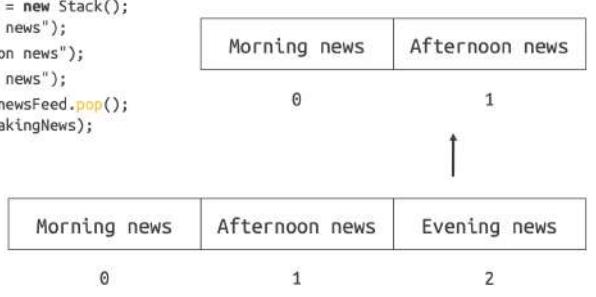
Morning news	Afternoon news	Evening news
0	1	2

Para añadir un elemento a una Stack LIFO usamos el método Push, especificando como argumento el elemento a añadir.

## Deleting elements from a Stack

```
Stack<String> newsFeed = new Stack();
newsFeed.push("Morning news");
newsFeed.push("Afternoon news");
newsFeed.push("Evening news");
String breakingNews = newsFeed.pop();
System.out.println(breakingNews);

//Prints
Evening news
```

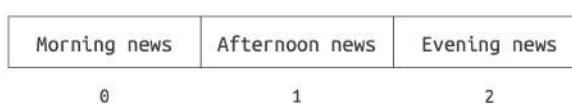


Y para eliminar un elemento de una stack LIFO usamos el método POP.

## Accessing elements from a stack

```
Stack<String> newsFeed = new Stack();
newsFeed.push("Morning news");
newsFeed.push("Afternoon news");
newsFeed.push("Evening news");
String breakingNews = newsFeed.peek();
System.out.println(breakingNews);

//Prints
Evening news
```



Y usaremos el método PICK si queremos acceder al último elemento de la stack sin eliminarlo. Es recomendable echar un vistazo a la documentación de Oracle sobre la clase Stack y su lista completa de métodos.

## QUEUES

## QUEUE



FIFO  
First In - First Out

Otro tipo de colección usado en Java son las queues o colas, como sugiere su nombre representa una línea de elementos situados uno detrás de otro.

A diferencia de la Stack o pila es FIFO, First In, First Out, donde el primer elemento que se añade a la cola es el primer elemento al que se accede o se elimina.

```
Module: java.base
Package: java.util
Interface: Queue<E>

Type Parameters:
E - the type of elements held in this queue

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Subinterfaces:
BlockingQueue<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingQueue, LinkedBlockingQueue, LinkedList,
LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue
```

Una queue en Java es una interfaz. Puedes consultar la documentación de Oracle sobre esta interfaz y sus métodos.

Some of the Queues's methods

Comprobarás que dispone de los métodos básicos de cualquier colección Java, como son Add y Remove, y que a estos métodos la interfaz Queue añade otros que permiten operaciones de inserción y extracción.

```
Module java.base
Package java.util

Interface Queue<E>

Type Parameters:
E - the type of elements held in this queue

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Subinterfaces:
BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue
```

La clase más popular que implementa la interfaz Queue es LinkedList. Veámosla.

## LINKEDLIST

```
Module java.base
Package java.util

Class LinkedList<E>

java.lang.Object
 java.util.AbstractCollection<E>
 java.util.AbstractList<E>
 java.util.AbstractSequentialList<E>
 java.util.LinkedList<E>

Type Parameters:
E - the type of elements held in this collection

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>
```

Una LinkedList es una clase que en su jerarquía extiende lista pero adicionalmente implementa la interfaz Queue. Entonces tenemos los métodos que tiene el ArrayList y además los que tienen las colas.

Creating a LinkedList

```
import java.util.LinkedList;

public class Main {

 public static void main(String[] args) {
 LinkedList<String> names = new LinkedList<>();
 }
}
```

Importamos, declaramos e inicializamos un LinkedList en nuestro proyecto de la misma forma que lo hacemos con un ArrayList.

## Adding elements to a LinkedList

```
LinkedList<String> names = new LinkedList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.add(2, "Alba");
```

Alex	Laia	Alba	Hugo
0	1	2	3

```
LinkedList<String> names = new LinkedList<>();
names.add("Alex");
names.add("Laia");
names.add("Hugo");
names.add(2, "Alba");

names.addFirst("Raj");
names.addLast("Maya");
```

Raj	Alex	Laia	Alba	Hugo	Maya
0	1	2	3	4	5

Y añadimos elementos de la misma forma utilizando el método Add.

O con los métodos adicionales de la interfaz Queue, AddFirst, AddLast, para añadir al principio o al final de la lista.

## Retrieving elements from a LinkedList

```
System.out.println("The LinkedList elements are:");
for (Iterator i = names.iterator(); i.hasNext();) {
 System.out.println(i.next());
}

//The LinkedList elements are
//Alex
//Laia
//Alba
//Hugo
```

Alex	Laia	Alba	Hugo
0	1	2	3

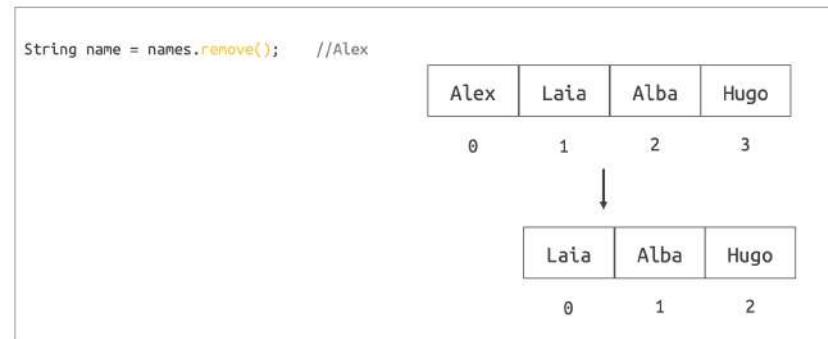
Podemos recorrer los elementos de una Linked List utilizando un iterador.

```
System.out.println(names.indexOf("Laia")); //1
System.out.println(names.indexOf("Lucas")); //-1
System.out.println(names.get(2)); //Alba
```

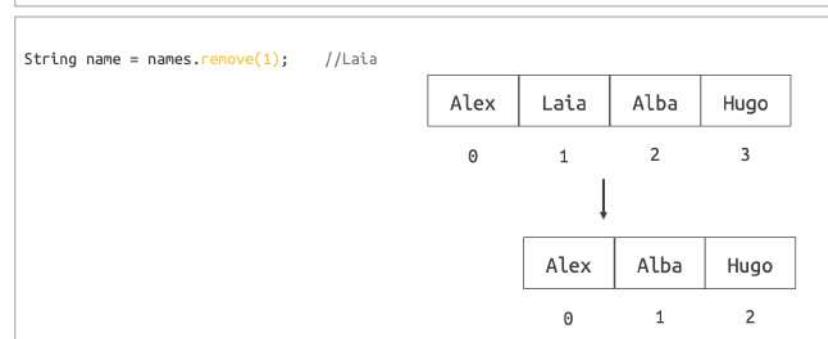
Alex	Laia	Alba	Hugo
0	1	2	3

Con el método de la interfaz ListIndexOf podemos consultar si un elemento está en la lista, nos devolverá su índice o menos uno en el caso de que no esté en la lista. Con el método Get y utilizando el índice como parámetro obtendremos el elemento.

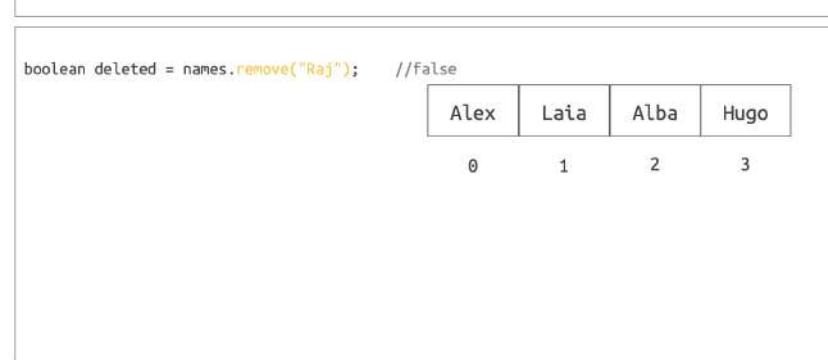
Deleting elements from a LinkedList



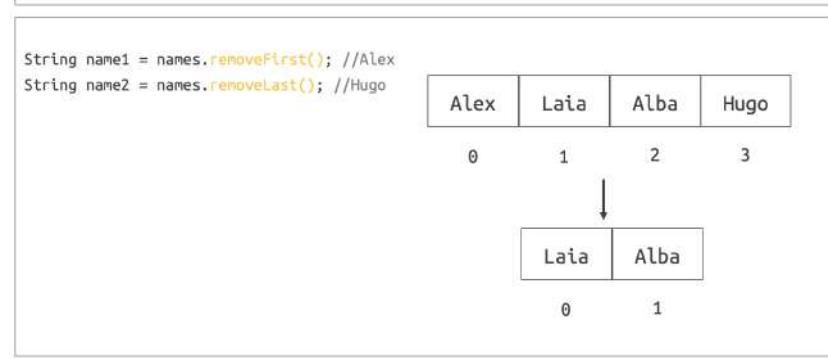
Para eliminar elementos de una Linked List podemos usar el método Remove. Si no indicamos parámetros de entrada en el método, se borrará el primer elemento de la lista.



Si como parámetro especificamos un entero, se eliminará el elemento que ocupe esta posición.



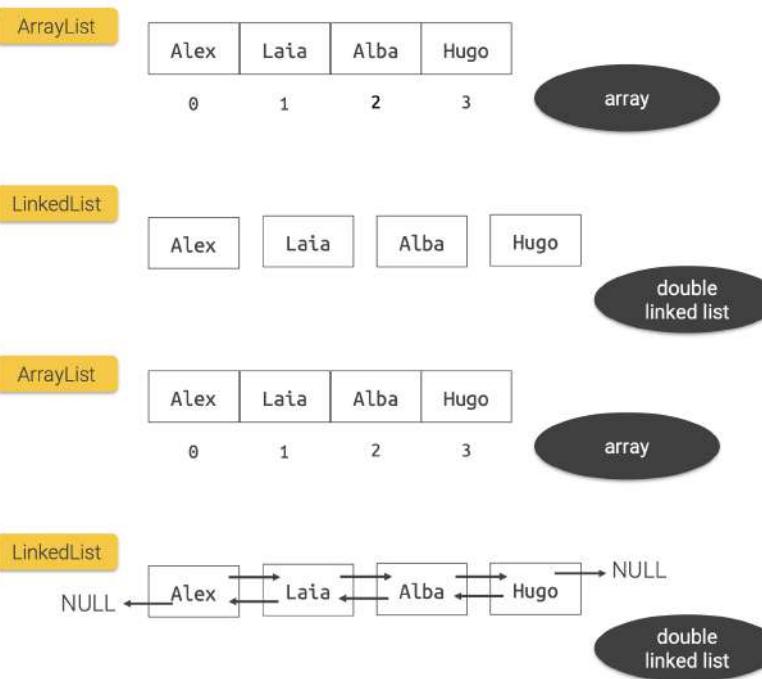
Y si especificamos un objeto, se eliminará la primera ocurrencia del elemento especificado si lo encuentra.



Y adivina. Igual que LinkedList, por implementar la Interface Queue, tenía los métodos addFirst y addLast, también dispone de removeFirst y removeLast para eliminar el primer y último elemento de la lista. Es recomendable echar un vistazo a la documentación de Oracle sobre la clase LinkedList y su lista completa de métodos.

## LINKEDLIST VS ARRAYLIST

Entonces, si un LinkedList es un ArrayList con más métodos ¿por qué no utilizar siempre LinkedList? Esta decisión dependerá del caso concreto en que nos encontremos, porque en unas situaciones un ArrayList será mejor y en otras ocasiones lo será un LinkedList. Y esto es debido a que ArrayList y LinkedList están implementadas de manera distinta.



Como sabemos, el ArrayList está implementado utilizando un Array. En cambio, una LinkedList se implementa utilizando una estructura de datos llamada Lista Dblemente Enlazada.

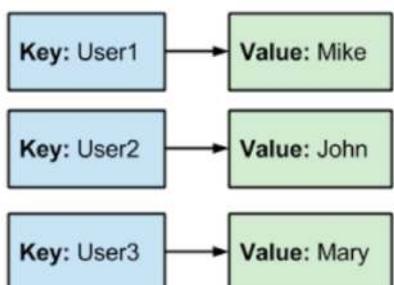
¿Qué significa esto? Pues que cada nodo de la lista contiene además del dato dos referencias, una al nodo anterior y otra al nodo siguiente.

Como consecuencia de estas implementaciones distintas, en una linked list podemos añadir elementos a la cabeza mucho más rápido, y también insertar o eliminar elementos en cualquier posición utilizando un iterador.

Sin embargo, una linked list ocupa más memoria que una arraylist debido a los punteros al anterior y posterior elemento. Pero la principal desventaja de una linked list frente a arraylist es que las operaciones de GET tardan bastante más. Por lo tanto, la recomendación es usar LinkedList en colecciones donde sean frecuentes las operaciones de añadir o eliminar elementos, ya que será mucho más rápido que si usas ArrayList. En el caso de que necesites una colección read-only, esto es, que apenas se modifique su número de elementos, mejor usar ArrayList.

## MAPS

### MAP



Los mapas son una colección que permite agrupar elementos a los que se accede mediante una clave, la cual debe ser única para los diferentes elementos de la colección. Un map en Java es una interfaz. Puedes consultar la documentación de Oracle sobre esta interfaz y sus métodos.

```

compact1, compact2, compact3
java.util
Interface Map<K,V>

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values

All Known Subinterfaces:
Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext,
NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:
AbstractMap, Attributes,AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap,
Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints,
SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

```

## Some of the Map's methods

El método get recupera un valor asociado a una clave dada. El método put permite añadir una pareja de clave-valor. Asocia la clave especificada en el key con el valor especificado en el value. El método remove elimina la clave-valor referenciada por la clave proporcionada.

El método keys devuelve un objeto en numeración con las claves que contiene la colección map. El método values devuelve un objeto collection con los valores que contiene la colección map. El método entrySet obtiene una colección set de los elementos del map. La clase más popular que implementa la interface queda es HighStable. Echemos un vistazo.

## HASHTABLE

```
Module java.base
Package java.util

Class Hashtable<K,V>

java.lang.Object
 java.util.Dictionary<K,V>
 java.util.Hashtable<K,V>

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values

All Implemented Interfaces:
Serializable, Cloneable, Map<K,V>

Direct Known Subclasses:
Properties, UIDefaults

public class Hashtable<K,V>
extends Dictionary<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Una Hashtable es una clase que implementa la interface map y que nos permite almacenar elementos con pares clave-valor.

## Creating a HashTable

```
import java.util.Hashtable;

public class Main {
 public static void main(String[] args) {
 Hashtable<String, String> hashtable = new Hashtable<>();
 }
}
```

Como vemos en la captura, para declarar una Hashtable debemos especificar el tipo tanto de la clave como del objeto que queremos almacenar.

```
Hashtable<String, Integer> numbers
 = new Hashtable<String, Integer>();

numbers.put("one", 1);
numbers.put("two", 2);
numbers.put("three", 3);
```

En el ejemplo que ves en la captura se crea la hash table numbers, que usa los nombres de los números como claves.

```
Hashtable<String, String> example1 = new Hashtable<>();

Hashtable<Integer, String> example2 = new Hashtable<>();

Hashtable<String, Person> example3 = new Hashtable<>();

Hashtable<int, double> example4 = new Hashtable<>();
```

Podemos utilizar tipos de datos diferentes tanto en la clave como en el valor. Siempre deberán ser objetos. Nunca tipos primitivos.

## Keys on HashTable

Hashtable **keys** must be of a **type** that implements **comparable** interface  
So they can be compared to secure that **keys are unique**

```
Hashtable<String, String> hashtable = new Hashtable<>();
Module java.base
Package java.lang
Class String
java.lang.Object
java.lang.String
All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>
```

En el tipo de dato de la key existe una restricción adicional. El tipo de dato debe permitir ser comparado con otro. Por este motivo, el tipo de dato es necesario que implemente la interface comparable. De este modo, las claves se pueden comparar para garantizar que son únicas.

En el ejemplo que ves en la captura, el tipo de dato key es String. La clase String implementa la interface comparable.

## Adding elements to a HashTable

```
Hashtable<String, String> hashtable = new Hashtable<>();

hashtable.put("Key1", "Chaitanya");
hashtable.put("Key2", "Ajeet");
hashtable.put("Key3", "Peter");
hashtable.put("Key4", "Ricky");
hashtable.put("Key5", "Mona");
```

Addinn kay-value pairs

Para añadir elementos a una hash table, utilizamos el método put. El primer parámetro es la clave, el segundo es el valor.

```
Hashtable<String, String> hashtable = new Hashtable<>();

hashtable.put("Key1", "Chaitanya");
hashtable.put("Key2", "Ajeet");
hashtable.put("Key3", "Peter");
hashtable.put("Key4", "Ricky");
hashtable.put(null, "Mona");
```

Si intentamos añadir un elemento con clave o valor null, dará error. Una hash table no permite almacenar ni claves ni valores null.

## Iterating a HashTable

Para poder recorrer la colección, el método keys nos proporcionará un enumerador con el que poder iterar la colección.

```

Use an Enumeration to iterate the
Hashtable

Enumeration names = hashtable.keys();
while(names.hasMoreElements()) {
 String key = (String) names.nextElement();
 System.out.println("Key: " +key+ " & Value: " + hashtable.get(key));
}

```

Call get method with the key to
retrieve the element

Una hash table es más rápida que otras colecciones si podemos prever la cantidad de elementos que almacenará, y si además no eliminamos ni añadimos demasiados elementos. Por lo tanto, utiliza una Hashtable para almacenar grandes cantidades de información que sufrirán pocos cambios y deban ser accedidos mediante una clave única.

Consulta la documentación de Oracle sobre la clase Hashtable y su lista completa de métodos.

## WORKING WITH COLLECTIONS

### COLLECTIONS CLASS

**java.util**

**Class Collections**

**java.lang.Object**  
**java.util.Collections**

---

**public class Collections**  
**extends Object**

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are null.

The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort` does not have to be a mergesort, but it does have to be *stable*.)

The "destructive" algorithms contained in this class, that is, the algorithms that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if the collection does not support the appropriate mutation primitive(s), such as the `set` method. These algorithms may, but are not required to, throw this exception if an invocation would have no effect on the collection. For example, invoking the `sort` method on an unmodifiable list that is already sorted may or may not throw `UnsupportedOperationException`.

This class is a member of the Java Collections Framework.

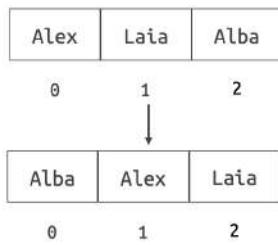
El framework de colecciones de Java proporciona la clase Collections. Es una clase famosa porque contiene métodos, todos ellos estáticos, que nos permiten trabajar con colecciones. Por ejemplo, para añadir una colección dentro de otra en una determinada posición o para obtener el valor mínimo o máximo de una determinada colección, invertir los elementos de una colección, etc.

## Sort method

```
ArrayList<String> names = new ArrayList<String>();
names.add("Alex");
names.add("Laia");
names.add("Alba");

Collections.sort(names);

for (String name:names){
 System.out.println(name);
}
```



Como muestra de la potencia de esta clase, vamos a ver uno de sus métodos, el método Sort, que ordena de menor a mayor la colección introducida como parámetro de entrada en el método sobreescribiéndola. Es recomendable echar un vistazo a la documentación de Oracle sobre la clase Collections y su lista completa de métodos.

## COLLECTIONS DEMO

### Demo 1

Veamos ahora una demo de uso de las colecciones LinkedList, Map y Stack. En esta demo veremos cómo utilizar algunas colecciones de Java. Comencemos viendo cómo funciona la pila. En esta clase de demostración, creamos un objeto Stack. Lo llenamos de información con el método `fillData`, fijaros que algunos elementos están repetidos y luego realizamos varias operaciones. El método `peek` utilizado aquí nos ha devuelto el último elemento que se ha añadido a la pila a continuación mostramos todo el contenido de la pila veremos que está en el mismo orden en el que hemos añadido los elementos.

Luego la primera llamada al método `pop` nos devuelve el mismo elemento que la anterior llamada a `peek`, pero a siguientes llamadas a este método nos van devolviendo los elementos siguientes. Fijaros, esto es debido a que `pop` retorna el primer elemento de la lista pero además lo elimina. Como podemos comprobar cuando volvemos a mostrar todo el contenido de la pila aquí debajo.

### Demo 2

En esta otra clase de demostración tenemos una Linked List.

En el método `FillData` vemos que le estamos añadiendo los mismos elementos y con el mismo orden que anteriormente hemos añadido a la pila. Después de llenar los datos, printamos el primer elemento, el último elemento y a continuación la colección entera, como vemos aquí. Podemos recuperar por delante y por detrás con `PollFirst` y `PollLast`. Aquí lo tenemos. Hemos obtenido los dos primeros y los dos últimos elementos respectivamente y los cuatro han sido eliminados de la colección, como podemos ver cuando volvemos a imprimir la lista aquí debajo.

Acto seguido, añadimos dos nuevos elementos a esta lista doblemente enlazada, uno por delante y otro por detrás. Por último, mostramos la lista por consola de tres formas distintas. Como lo hemos hecho siempre, utilizando un `for`, y luego empezando por delante y empezando por detrás, utilizando, en este caso, un iterador `ListIterator`. Aquí vemos el resultado, como lo hemos hecho siempre, imprimiendo desde delante e imprimiendo desde atrás.

### Demo 3

De la interfaz Map hemos visto la clase `Hashtable`, pero antes de desarrollar un ejemplo con esta clase os voy a mostrar otra colección la clase `HashMap` que también implementa esta interface. En esta clase declaramos una colección map de tipo `HashMap` y la instanciamos como tipo map, aquí lo vemos. Esto nos permitirá después cambiar el tipo de colección map sin hacer modificaciones en el

código. Una vez instanciado, el objeto map llamamos al método fillData que llenará de datos la colección. Fijaros que entre ellos tenemos valores null.

A continuación declaramos un iterador. Recordad que un iterador es un puntero que nos permitirá ir recorriendo uno a uno todos los elementos de una colección. Hemos visto inicializar un iterador para HighStable con el método keys, pero recordad que estamos utilizando esta HighStable como un objeto map, así que utilizaremos el método EntrySet, que es común para todas las colecciones map. Con el iterador vamos tratando elemento a elemento como un objeto clave valor entry y lo mostraremos por pantalla.

Acto seguido realizamos una inserción y una eliminación y volveremos a mostrar el contenido. Aquí finalmente recuperamos una de las entradas mediante su clave. Vemos que lo que nos devuelve el iterador no tiene por qué coincidir con el orden de inserción que hemos realizado antes. Si ahora cambiamos el tipo de HashMap por HashTable, vemos que no tenemos ningún error de compilación. Pero ejecutemos. Ahora sí obtenemos un error, nullPointerException en tiempo de ejecución, puesto que HashTable no admite valores ni claves null. Si comentamos este valor y volvemos a ejecutar, ya no tenemos ningún problema.

## ENUMS

### ENUMS

Enums es un tipo de dato especial de Java y otros lenguajes que nos permitirá definir colecciones de constantes. En Java podemos utilizar Enums para definir colecciones de constantes como pueden ser las estaciones del año, los días de la semana, etc. Un Enum en Java es un tipo de clase especial y a diferencia de las otras colecciones de Java, Enum se encuentra en el paquete java.lang.

#### Creating an enum

Creamos un Enum de la siguiente forma. Como los elementos que contiene son constantes, usamos mayúsculas en su definición.

#### Fields in enums

En los Enum podemos definir atributos. En tal caso, deberemos especificar también un constructor. Como que los Enums tienen valores constantes, este constructor se crea implícitamente como privado.

#### Method in enums

Podemos declarar métodos dentro de los enums, por ejemplo, para devolver el valor de un elemento.

#### Enums usage

Y aquí tenemos un ejemplo de cómo podríamos usar el enum Syson. En este caso, en una sentencia switch case, como si fuera un tipo primitivo de datos, evaluamos un objeto Syson como expresión en el switch y podemos utilizar los valores constantes del enum para establecer los bloques case.

#### Enums inside classes

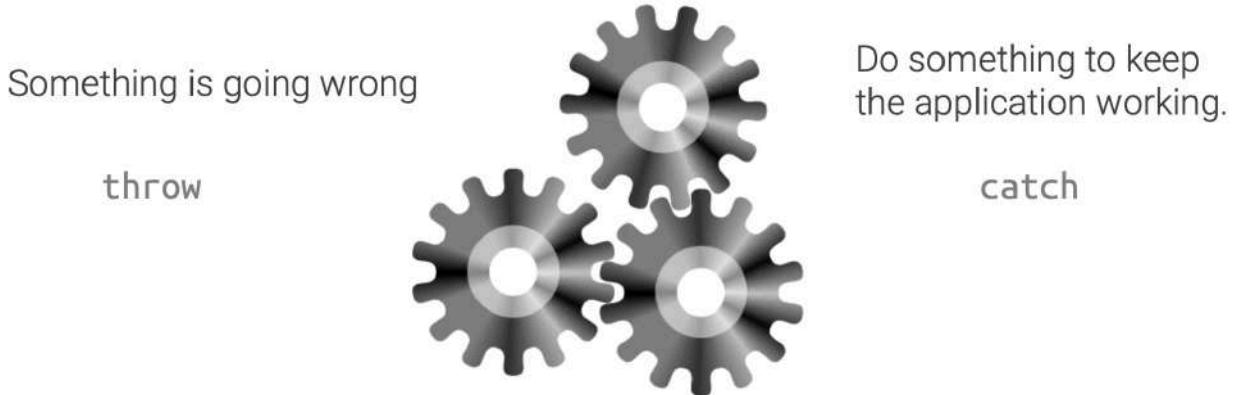
Y también podemos usar los enums que hayamos definido como tipo de dato de cualquier atributo de una clase. Puedes encontrar información detallada de la clase enum en la documentación oficial de oracle.

## CONTROL DE EXCEPCIONES Y COLECCIONES DE DATOS II

### EXCEPTIONS: INTRODUCTION

#### EXCEPTIONS: INTRODUCTION

Sabemos que en tiempo de ejecución pueden producirse situaciones que alteren el deseado transcurso de un programa. Por ejemplo, si un recurso deja de estar disponible y no podemos acceder a él.



Los lenguajes de programación deben ofrecer al programador mecanismos para gestionar estas situaciones con el fin de establecer alternativas en lugar de provocar una salida brusca del programa. La gestión de excepciones es el conjunto de mecanismos que un lenguaje de programación proporciona para detectar y gestionar los errores que se pueden producir en tiempo de ejecución.

Según el modelo de gestión de excepciones en Java, al producirse un error, la máquina virtual lanza un aviso, lo que conocemos como throw, que el programador debería poder capturar utilizando la cláusula catch para resolver la situación problemática y mantener la aplicación en funcionamiento.

#### Error VS Exception

Veamos ahora qué dos situaciones no deseadas pueden producirse en tiempo de ejecución. Java distingue entre error y excepción.

Error	Exception
Irrecoverable situation	Recoverable situation
Related to environment	Related to application
There is no programmable fix	May be anticipated and should be handled

`OutOfMemoryError`

`class Error`

`ArrayIndexOutOfBoundsException`

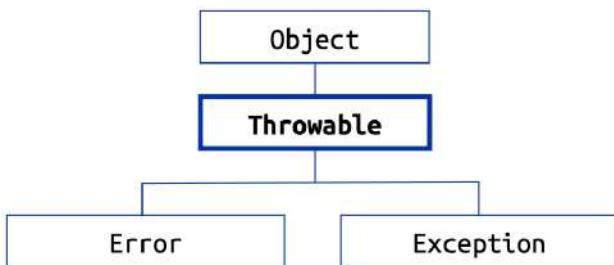
`class Exception`

Los errores son situaciones irrecuperables que no tienen solución y que no dependen del programador. No se deberían de producir nunca, pero cuando se producen provocan la interrupción brusca del programa. Por ejemplo, la máquina virtual se queda sin recursos para continuar la ejecución del programa.

Las excepciones, valga la redundancia, son situaciones excepcionales que los programas se pueden encontrar en tiempo de ejecución, incluyendo los errores de programación. El programador puede prever cada tipo de excepción y escribir el código para su gestión.

Por ejemplo, se producirá una excepción si intentamos acceder a una posición inexistente dentro de un array. Java engloba los posibles errores y excepciones en las clases Error y Exception según corresponda.

Class Throwable



Como podemos ver en la imagen, las clases Error y Exception se encuentran al mismo nivel dentro de la jerarquía de clases de Java, justo por debajo de la superclase Throwable. Esto significa que ambas heredan sus características y métodos.

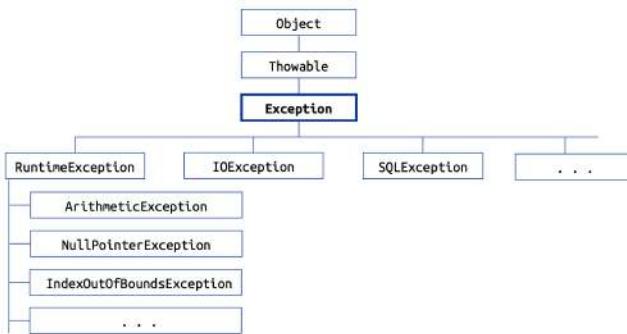
Constructors	
Modifier	Constructor and Description
	<code>Throwable()</code> Constructs a new throwable with <code>null</code> as its detail message.
	<code>Throwable(String message)</code> Constructs a new throwable with the specified detail message.
	<code>Throwable(String message, Throwable cause)</code> Constructs a new throwable with the specified detail message and cause.
protected	<code>Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)</code> Constructs a new throwable with the specified detail message, cause, <code>suppression</code> enabled or disabled, and writable stack trace enabled or disabled.
	<code>Throwable(Throwable cause)</code> Constructs a new throwable with the specified cause and a detail message of ( <code>cause==null ? null : cause.toString()</code> ) (which typically contains the class and detail message of <code>cause</code> ).

Como podemos ver en la documentación de la clase Throwable, disponible en Docs.Oracle.com, hay dos constructores que incorporan la posibilidad de crear un objeto throwable, indicando otro objeto throwable como causante del nuevo objeto. Esto permite encadenar los errores y las excepciones.

Methods	
Modifier and Type	Method and Description
void	<code>addSuppressed(Throwable exception)</code> Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.
Throwable	<code>fillInStackTrace()</code> Fills in the execution stack trace.
Throwable	<code>getCause()</code> Returns the cause of this throwable or <code>null</code> if the cause is nonexistent or unknown.
String	<code>getLocalizedMessage()</code> Creates a localized description of this throwable.
String	<code>getMessage()</code> Returns the detail message string of this throwable.
StackTraceElement[]	<code>getStackTrace()</code> Provides programmatic access to the stack trace information printed by <code>printStackTrace()</code> .
Throwable[]	<code>getSuppressed()</code> Returns an array containing all of the exceptions that were suppressed, typically by the <code>try-with-resources</code> statement, in order to deliver this exception.
Throwable	<code>initCause(Throwable cause)</code> Initializes the cause of this throwable to the specified value.
void	<code>printStackTrace()</code> Prints this throwable and its backtrace to the standard error stream.
void	<code>printStackTrace(PrintStream s)</code> Prints this throwable and its backtrace to the specified print stream.
void	<code>printStackTrace(PrintWriter s)</code> Prints this throwable and its backtrace to the specified print writer.
void	<code>setStackTrace(StackTraceElement[] stackTrace)</code> Sets the stack trace elements that will be returned by <code>getStackTrace()</code> and printed by <code>printStackTrace()</code> and related methods.
String	<code>toString()</code> Returns a short description of this throwable.

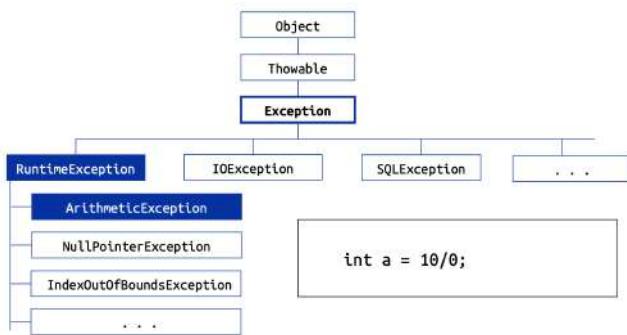
La clase throwable dispone de métodos para conocer el contexto en el que se ha producido la situación problemática y, por tanto, poder actuar en consecuencia. En este caso, nos será de especial utilidad los métodos GetCalls, que nos devuelve un objeto throwable que contiene la causa del error. GetMessage, que devuelve un mensaje descriptivo del error producido. PrintStackTrace, que muestra por el canal de errores el contexto donde se ha producido el error, incluyendo la cascada de llamadas desde el método main que han llevado al punto en el que se ha levantado el error, o toString, que devuelve una breve descripción del objeto en formato de texto.

## EXCEPTION: TYPES

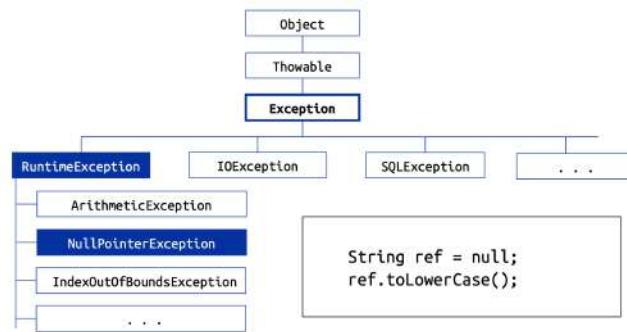


Al producirse una excepción en un programa, se crea un objeto de la subclase de Exception a la que pertenece la excepción. En pantalla podemos ver cómo se organiza la jerarquía de la clase Exception en Java. Las excepciones que se producen por algún problema dentro de la máquina virtual Java se engloban en la subclase RuntimeException. Según el problema ocurrido, el objeto que representará la excepción producida será una subclase de RuntimeException.

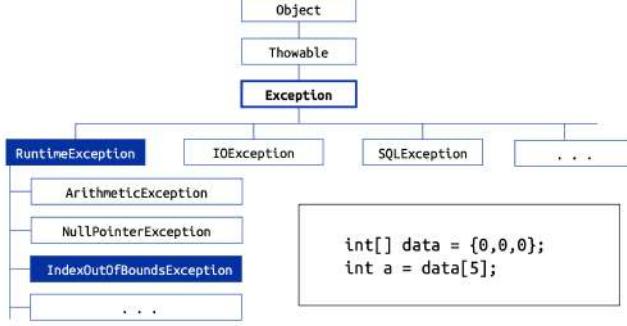
A continuación veremos algunas de ellas.



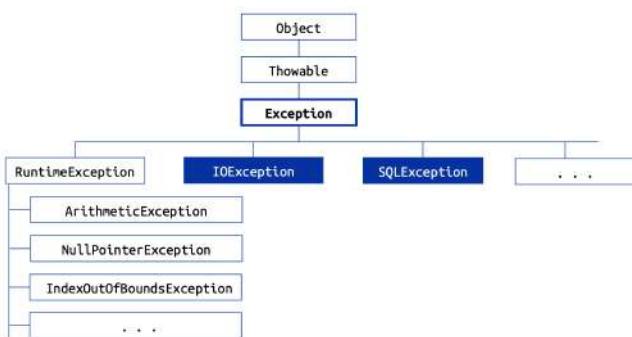
Obtendremos una ArithmeticException si el problema se ha producido durante una operación aritmética, como vemos en el código de ejemplo en pantalla, si intentamos dividir por cero.



Intentar utilizar una referencia a un objeto que no ha sido creado, es decir, cuyo valor es null, provocará una null pointer exception.



Si intentamos acceder a un índice inexistente, bien sea dentro de un string, un vector o como en el código del ejemplo, un array, hará que se produzca una index out of bounds exception.



Otras subclases que heredan directamente de Exception son por ejemplo IOException o SQLException.

### Implicit VS Explicit

Una clasificación no oficial pero que nos puede ayudar en nuestra tarea como programadores es diferenciar entre excepciones implícitas o explícitas.

Implicit	Explicit
RunTimeExceptions	Not RunTimeExceptions
May/Should be controlled	Must be controlled
Can't be worked around in run time	May be worked around in run time

Las excepciones implícitas son aquellas que pertenecen a una subclase de RuntimeException. Habitualmente representan errores de programación. La misma máquina virtual se encargará de gestionarlas si no lo hace el propio programador.

Algo recomendable para mejorar la percepción de los usuarios sobre la aplicación. Pero debemos ser conscientes que no podremos resolver estas excepciones en tiempo de ejecución.

### Implicit Exception

```

Public class Main {
 public static void main (String[]){
 String filename=null;
 if (filename.contains("aaaa")){
 //TODO code application
 }else{
 //TODO code application
 }
 }
}

```

#### Example

En el siguiente código, podemos ver un ejemplo de excepción implícita. Vemos que en la tercera línea se declara la variable StringFileName y a la vez se inicializa con un valor nulo. En la línea inmediatamente siguiente, intentamos utilizar el método contains de la variable stringFileName.

Implicit
RunTimeExceptions
May/Should be controlled
Can't be worked around in run time

#### NullPointerException

unchecked

Esto producirá una excepción en tiempo de ejecución de tipo runtimeException. Concretamente obtendremos una nullPointerException, que se produce siempre que intentamos utilizar métodos de un objeto null.

Las excepciones implícitas también se conocen como uncheck, puesto que el compilador de Java no las detecta. Un ejemplo de excepción implícita o uncheck es NullPointerException.

## Explicit Exception

```
Public class Main {
 public static void main (string[]){
 //TODO code application
 String filename="employees.dat";
 FileReader fr = new FileReader(filename);
 }
}
```

unreported exception FileNotFoundException must be caught or declared to be thrown

Example

Las excepciones explícitas son todas aquellas que no pertenecen a ninguna subclase de RuntimeException. Son excepciones que el programador está obligado a tener en cuenta, mediante el uso de cláusulas try-catch o declarando que el método pudo lanzar una excepción con la palabra reservada throws.

Ahora vemos un ejemplo de una excepción explícita. Después de escribir este código en nuestro IDE, el compilador nos alertará que algo no está bien, típicamente subrayando la línea donde está el problema.

## Explicit

Not RunTimeExceptions

Must be controlled

May be worked around in run time

## FileNotFoundException

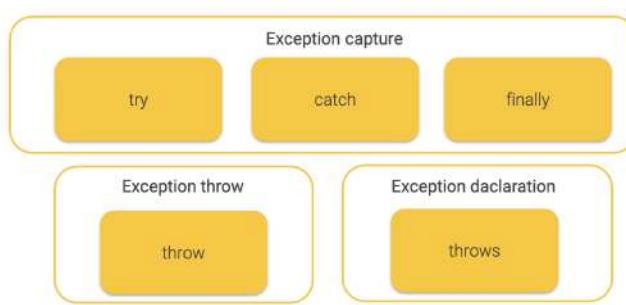
checked

Explorando la descripción textual obtendremos más información. En este caso, nos indica que la línea en cuestión puede lanzar una excepción del tipo FileNotFoundException. No se puede garantizar que el fichero Employees.dat vaya a estar siempre disponible. Es algo que no depende del lenguaje, sino de factores externos. Debemos, por lo tanto, controlar la posible excepción

Las excepciones explícitas también se conocen como check, puesto que el compilador de Java detectará que es posible que se produzcan e instará al programador a enmendar el código para su control. Un ejemplo de excepción explícita o check es FileNotFoundException.

**EXCEPTION HANDLING: CAPTURE****EXCEPTION HANDLING: INTRODUCTION**

## Exception handling keywords



El manejo de excepciones en Java se gestiona a través de cinco palabras clave. Veamos en primer lugar en qué ámbito se utilizan cada una de ellas. Try, catch y finally son utilizadas en la captura de las excepciones y su posterior tratamiento.

Throw se utiliza para el lanzamiento de las excepciones. Y throws se utiliza para su declaración.

## EXCEPTION CAPTURE

Try, catch & finally clauses

```
try{
 //code that may cause an Exception
}catch (nombreClasseException1 e1){
 //actions to do when de Exception1 happens
}catch (nombreClasseException2 e2){
 //actions to do when de Exception2 happens
} finally{
 //code to be executed always
}
```

Las palabras clave Try, Catch y Finally forman un subsistema interrelacionado en el que el uso de uno implica el uso de otro. Como vemos en pantalla, se emplean para enclausurar bloques de código. Dentro de un bloque Try, escribiremos código susceptible de causar una excepción.

Un bloque Try debe ir siempre seguido como mínimo de un bloque catch. En los bloques catch, escribiremos las acciones que queremos hacer cuando se produce la excepción. Un bloque catch va precedido de una declaración que define la excepción a la que el bloque da respuesta.

Podemos encadenar varios bloques catch para dar respuesta a distintos tipos de excepciones producidas dentro del bloque try. Al finalizar el bloque try o un bloque catch si se ha producido una excepción se ejecutará el código del bloque Finally y se continuará la ejecución del programa. Los bloques finally son de utilidad para cerrar recursos tales como conexiones o archivos ante una finalización prematura del programa. El bloque finally es opcional, pero cuando está su contenido se ejecutará siempre, incluso si los bloques try o catch que le preceden devuelven el método con un return.

## Demo ejemplo 1

Veamos un ejemplo práctico de la captura de excepciones utilizando las cláusulas try, catch y final. El siguiente código debe cargar el contenido de un fichero txt en una variable inputStream, utilizando para ello un objeto FileInputStream. Vemos que nuestro IDE, en este caso IntelliJ, realza el objeto FileInputStream indicando que puede lanzar una excepción de tipo FileNotFoundException, lo cual no se está capturando.

```
public static void main(String[] args) {
 InputStream in = null;
 try {
 System.out.println("About to open a file");
 in = new FileInputStream("missingfile.txt");
 System.out.println("File open");
 int s = in.read();
 } catch(FileNotFoundException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
 } catch(IOException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
 } finally{
 System.out.println("Application end");
 try{
 if(in != null){
 in.close();
 }
 } catch(IOException e){
 System.out.println("Failed to close file");
 }
 }
}
```

Para hacerlo utilizaremos la cláusula try. Una cláusula try siempre debe ir seguida de una cláusula catch o finally, capturaremos la excepción con una cláusula catch. En nuestro caso, capturaremos todo tipo de excepciones. Ahora el código ya no tiene errores de compilación, pero dejarlo así es una mala práctica. En nuestro caso, informaremos por consola que algo no ha ido bien.

También mostraremos la descripción del error.

```

public static void main(String[] args) {
 InputStream in = null;
 try {
 System.out.println("About to open a file");
 in = new FileInputStream("missingfile.txt");
 System.out.println("File open");
 int s = in.read();
 } catch(FileNotFoundException e){
 System.out.println(e.getClass().getName()+"-"+ e.getMessage());
 } catch(IOException e){
 System.out.println(e.getClass().getName()+"-"+ e.getMessage());
 } finally{
 System.out.println("Application end");
 try{
 if(in != null){
 in.close();
 }
 } catch(IOException e){
 System.out.println("Failed to close file");
 }
 }
}

```



## Demo ejemplo 2

```

public static void main(String[] args) {
 InputStream in = null;
 try {
 System.out.println("About to open a file");
 in = new FileInputStream("missingfile.txt");
 System.out.println("File open");
 int s = in.read();
 } catch(FileNotFoundException e){
 System.out.println(e.getClass().getName()+"-"+ e.getMessage());
 } catch(IOException e){
 System.out.println(e.getClass().getName()+"-"+ e.getMessage());
 } finally{
 System.out.println("Application end");
 try{
 if(in != null){
 in.close();
 }
 } catch(IOException e){
 System.out.println("Failed to close file");
 }
 }
}

```

java.io.IOException

Los necesitamos porque debemos capturar obligatoriamente las excepciones de tipo check que pueden producir el constructor FileInputStream, que puede arrojar una FileNotFoundException, o el método read de la clase InputStream, que puede arrojar una IOException.

La ordenación de los bloques Catch no es casual, puesto que estamos capturando dos excepciones que forman parte de la misma jerarquía. FileNotFoundException es una subclase de IOException. En estos casos, siempre debemos capturar primero la excepción más específica. Centrémonos ahora en el bloque Finally.

Sabemos que de ser especificado este bloque, se ejecutará siempre al final, y que lo debemos utilizar para liberar recursos abiertos dentro del bloque Try. En este caso, cerraremos el objeto InputStream, que podría haber sido abierto previamente. El método Close de la clase InputStream también puede arrojar una IOException, por lo que debemos capturar la posible excepción mediante otra cláusula try-catch.

Vemos así que también podemos utilizar este tipo de cláusulas de forma anidada.

Finalmente, de forma opcional, podemos incluir también una cláusula Finally, que utilizaremos para indicar que la aplicación ha terminado, independientemente de si se han producido errores o no. A continuación, iniciaremos la aplicación. En la consola, vemos los mensajes que ha dejado su ejecución, que nos da a conocer que se ha producido un error debido a que no se ha encontrado el fichero missingfile.txt.

Veamos ahora una versión extendida del código de la demo para adentrarnos en las características de la captura de excepciones en Java. En este caso, en lugar de tener un único bloque catch para capturar todo tipo de excepciones, tenemos dos bloques catch más específicos que capturan solo excepciones del tipo FileNotFoundException o IOException.

## Multi-catch

```

try {
 System.out.println("About to open a file");
 in = new FileInputStream("missingfile.txt");
 System.out.println("File open");
 int x = in.read();
 int y = 20 / x
} catch(ArithmeticException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(FileNotFoundException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(IOException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
}

```

java.lang.ArithmeticException

```

try {
 System.out.println("About to open a file");
 in = new FileInputStream("missingfile.txt");
 System.out.println("File open");
 int x = in.read();
 int y = 20 / x
} catch(FileNotFoundException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(IOException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(ArithmeticException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
}

```

java.lang.ArithmeticException

```

try {
 System.out.println("About to open a file");
 in = new FileInputStream("missingfile.txt");
 System.out.println("File open");
 int x = in.read();
 int y = 20 / x
} catch(FileNotFoundException | ArithmeticException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
} catch(IOException e){
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
}

```

```

try {
 System.out.println("About to open a file");
 in = new FileInputStream("missingfile.txt");
 System.out.println("File open");
 int x = in.read();
 int y = 20 / x
} catch(FileNotFoundException | IOException | ArithmeticException e){ ✗
 System.out.println(e.getClass().getName()+" - "+ e.getMessage());
}

```

Types in multi-cast must be disjoint: FileNotFoundException is a subclass of IOException

Fijémonos ahora en esta nueva versión de código con una instrucción adicional que puede arrojar otro tipo de excepción. En este caso, dependiendo del valor de X, podemos obtener una arithmetic exception. La posible nueva excepción se controla en su correspondiente bloque Catch.

De este modo, tenemos tres bloques Catch para capturar tres tipos distintos de excepciones. Esto nos permitiría poder gestionar cada una de ellas de una forma distinta. Pero en el ejemplo, vemos que estamos realizando la misma acción para cada una de ellas.

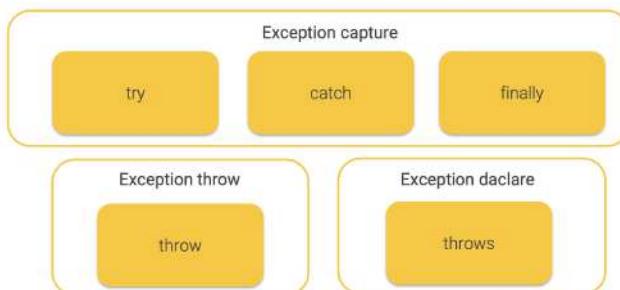
La característica Multicatch de Java nos ayuda a optimizar el código permitiendo la captura de más de una excepción en un mismo bloque catch. Como vemos en la imagen, ahora las excepciones FileNotFoundException y ArithmeticException son capturadas por el mismo bloque catch.

En la declaración del bloque Multicatch debemos separar las excepciones que queremos capturar mediante el símbolo or. Debemos tener en cuenta que no podemos capturar en un bloque Multi catch excepciones que formen parte de la misma jerarquía. De este modo, este bloque Multicatch sería incorrecto, porque FileNotFoundException es una subclase de IOException.

## EXCEPTION HANDLING: DECLARE AND THROW

### EXCEPTION HANDLING: INTRODUCTION

Exception handling keywords



Conocemos el uso de las cláusulas de captura de excepciones try catch y finally. Veamos ahora cómo se utiliza throw y Throws para declararlas.

### EXCEPTION THROW



La palabra clave Throw en Java se emplea para lanzar una excepción desde un método o cualquier bloque de código.

Podemos lanzar de este modo excepciones de tipo check o uncheck indistintamente, pero throw se utiliza principalmente para lanzar excepciones personalizadas.

### Throw y rethrow

```

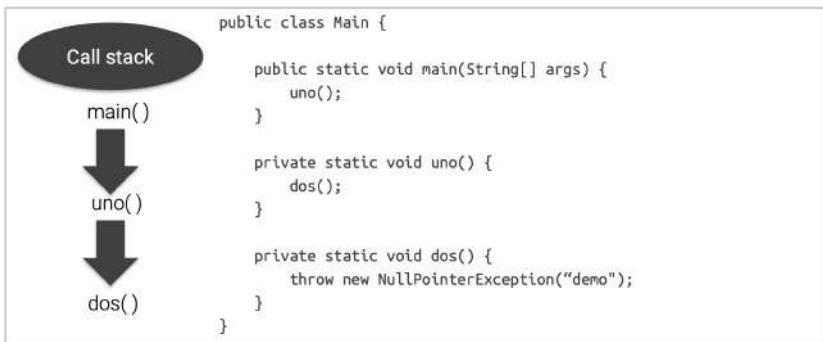
static void demoThrow()
{
 try
 {
 throw new NullPointerException("Explicitly thrown exception");
 }
 catch(NullPointerException e)
 {
 System.out.println("Caught inside demoThrow().");
 throw e; // rethrowing the exception
 }
}

```

En este ejemplo, vemos una sentencia Throw dentro de un bloque Try, y que es capturada en un bloque Catch dentro del mismo método. En determinados casos, también puede resultar útil relanzar esta misma excepción después de realizar alguna acción dentro del bloque Catch.

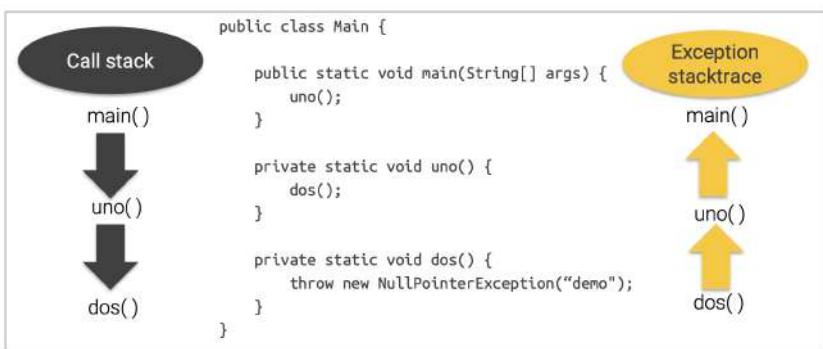
En tal caso, la excepción deberá ser capturada en un método anterior en la pila de ejecución del programa.

## Exception propagation



Veamos cómo funciona la propagación de las excepciones en la pila de ejecución del programa. En el código del ejemplo, vemos una aplicación que en su método `Main` realiza una llamada a otro método llamado `1`. El método `1` realiza una llamada al método `2`. Esta secuencia de métodos se conoce como pila de llamadas.

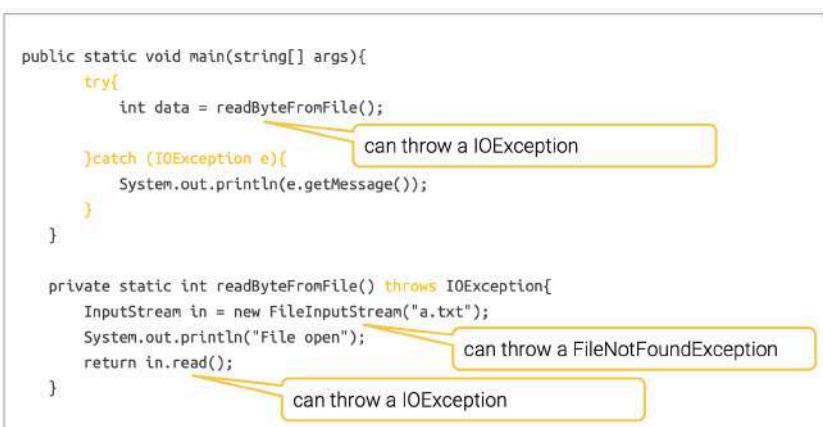
Si en tiempo de ejecución se lanza o se produce una excepción y ésta no es capturada dentro del mismo método, la máquina virtual aborta la ejecución del método donde se ha producido la excepción y propaga la excepción al método inmediatamente superior.



Si en el método superior tampoco se captura la excepción, se aborta la ejecución de éste y se propaga la excepción a su superior, y así sucesivamente hasta que la excepción es gestionada. Si la excepción no es gestionada y se llega al `main`, y aquí tampoco se gestiona, se interrumpe el programa.

Podemos encontrar información relativa al recorrido de la excepción desde que se lanza hasta que es capturada en la pila de llamadas de la excepción, más conocida por su término en inglés Stack Trace.

## EXCEPTION DECLARE



Cuando un método puede propagar una excepción hacia un método superior, siempre es conveniente especificarlo en su declaración. En el ejemplo, el método `ReadByteFromFile` puede producir dos tipos de excepción, una `File Not Found Exception` y una `EO Exception`. Debemos añadir, por lo tanto, la sentencia `Throws` en la declaración.

De este modo, quien utilice el método sabrá que éste puede arrojar una excepción y podrá tomar las medidas para su control.

### Demo ejemplo 1

Veamos un ejemplo práctico del uso de throw y throws para la declaración y lanzamiento de excepciones. Para ello, observemos el siguiente programa, el objetivo del cual es inicializar la variable data con un valor entero. Para ello, utilizamos el método getValue. Este método podemos suponer que contiene múltiples líneas y por el motivo que sea, deseamos lanzar una excepción. Por ejemplo, una IOException. Para lanzar una excepción de cualquier tipo, debemos empezar la sentencia con la palabra throw, seguido del operador new, y el nombre de la excepción que queremos lanzar.

Pasando como parámetro un texto, un mensaje del error producido. Ahora nuestro IDE ha detectado una excepción check que no está siendo controlada. Para hacerlo, podemos utilizar los bloques try-catch, pero en nuestro caso, escogeremos propagar la excepción al método superior. Podemos hacer esto utilizando la sentencia throws, seguido del nombre de la excepción que queremos propagar. Ahora vemos que el aviso que no estamos controlando una excepción check se ha desplazado al método main.

Concretamente, en la sentencia donde hacemos uso del método getValue, que ahora puede arrojar una IOException. En este caso, añadiremos las cláusulas tryCatch para controlarlo. Eliminamos este código, que es inalcanzable, puesto que por arriba estamos lanzando siempre una excepción, y obtenemos un programa libre de errores. Ahora podemos ejecutarlo.

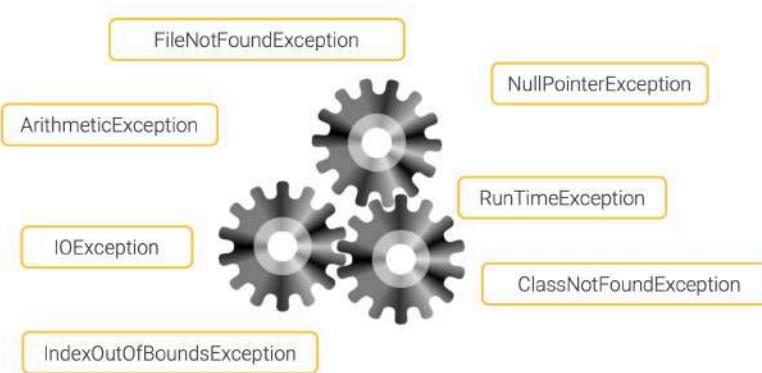
### Demo ejemplo 2

Hagamos el ejemplo más real utilizando un objeto scanner para que podamos introducir valores por consola. Supongamos que solo queremos aceptar números enteros. En tal caso, retornaremos el valor. Y de lo contrario, lanzaremos la excepción. Ejecutemos el programa. Introduzcamos un valor numérico, por ejemplo el 1, y vemos que el programa finaliza sin problemas.

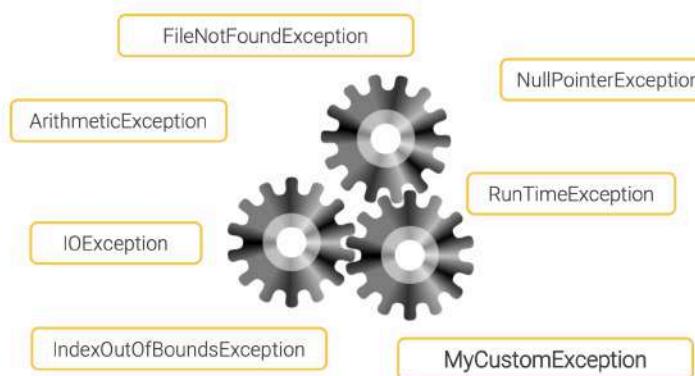
Introduzcamos ahora otro valor que no sea numérico. Vemos que se ha lanzado la excepción. Hemos visto que la propagación de excepciones es útil como medio para enviar un aviso a otra parte del programa, indicándole que algo está sucediendo y no es posible continuar con la ejecución normal del método. Utilizamos throw para lanzar una excepción de cualquier tipo en cualquier momento. Y debemos utilizar throws en la declaración de un método que propaga algún tipo de excepción.

## CUSTOM EXCEPTIONS

### CUSTOM EXCEPTIONS: INTRODUCTION



Sabemos que durante la ejecución de un programa pueden producirse excepciones por motivos diversos, mediante el uso de las técnicas de control de excepciones podemos gestionar estas situaciones y mantener nuestra aplicación en funcionamiento. Algunas de estas excepciones son lanzadas por la máquina virtual de Java y otras por el propio programador.

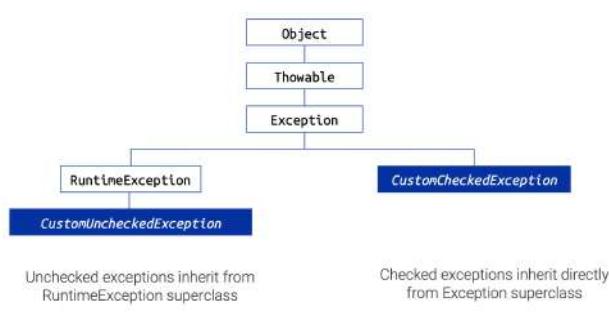


A veces el programador no encuentra entre las excepciones existentes ninguna adecuada a las características de la situación que quiere notificar.

En este caso, resulta práctico definir una excepción personalizada que sirva exactamente para el propósito específico que necesita el programador.

## CUSTOM EXCEPTIONS: CLASSES

Custom exception classes hierarchy



Las excepciones personalizadas que queramos crear tendrán su propia clase y ocuparán su propio lugar dentro de la jerarquía de clases de Java.

Si queremos una excepción de tipo Unchecked, deberemos extender la clase `RuntimeException`. Si lo que queremos es crear una excepción de tipo Check, extenderemos directamente la clase `Exception`.

## Custom exception class

```

Inherits from Exception superclass
public class MyCustomException extends Exception {
 public MyCustomException(String message) {
 super(message);
 }
}
We should call de superclass constructor

```

The code snippet shows the definition of a custom exception class named `MyCustomException`. It extends the `Exception` class. In the constructor, it calls the superclass constructor using the `super(message)` statement. Two yellow annotations are present: one pointing to the `super` call with the text "Inherits from Exception superclass", and another pointing to the `super(message)` call with the text "We should call de superclass constructor".

En el código de la pantalla podemos ver un ejemplo de excepción personalizada. En la declaración de la clase indicamos que queremos extender una excepción. En el constructor de la nueva clase realizaremos una llamada al constructor de la clase padre.

## Demo ejemplo

Veamos un uso práctico de las excepciones personalizadas en Java. Para ello, fijémonos en la siguiente clase `Account`, que representa una cuenta bancaria. Tiene un único atributo `Balance`, para reflejar el saldo de la cuenta. Dispone de dos constructores y tres métodos.

Uno para obtener el saldo actual y otros dos para realizar ingresos y reintegros. Por otro lado, tenemos la clase `Automated Teller Machine` que representa un cajero automático. Como único atributo, tiene un objeto `Account` y luego tiene el método `main` y dos métodos más para simular ingresos y reintegros. Ejecutemos el programa. Vemos que tras crear una cuenta a cero y realizar un ingreso de 100, podemos también realizar un reembolso de 120, resultando en un saldo negativo de 20. Podemos considerar este caso como un problema, puesto que esta cuenta no dispone del 120 y no debería haberse permitido un reembolso superior a su saldo.

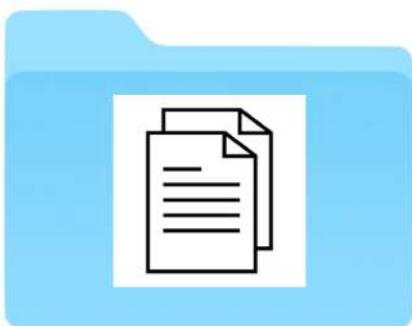
Vayamos entonces al método withdraw de la clase account a controlar esta casuística. Controlaremos que si el saldo actual es inferior al reembolso deseado abortaremos la ejecución del método lanzando una excepción. En caso contrario, permitiremos que se haga la operación. debemos especificar que este método propaga una excepción genérica y nos veremos obligados a controlar esta excepción cuando hagamos uso del método. Hemos utilizado la clase genérica Exception para lanzar la excepción. Esto prácticamente no nos da información de lo que ha ocurrido y dificulta, por ejemplo, comunicar al usuario qué ha sucedido y cómo puede proceder. Para mejorar la comunicación con el usuario, debemos también mejorar la comunicación entre clases. Para ello, haremos uso de una excepción personalizada.

Crearemos la clase LowBalanceException que extenderá de la clase exception. En el constructor llamaremos al constructor de la super clase con la keyword super. Ya tenemos la excepción personalizada creada. Vayamos a hacer uso de ella, en lugar de lanzar una excepción genérica, lanzaremos una LowBalanceException, vemos que nos marca que debemos especificar siempre un mensaje. Esto es porque solo hemos creado un constructor que obligatoriamente debe recibir un string. Crearemos pues el mensaje. Lanzaremos hacia método superior una LowBalanceException y vemos en el método main que no nos da ningún mensaje de error porque ya estamos capturando la excepción más genérica posible. Pero podemos añadir otro bloque catch para capturar excepciones específicas como la Low Balance Exception.y lo que haremos será mostrar el mensaje que nos da la descripción. Vemos que ahora el mensaje es más descriptivo cuando se produce la situación específica que queríamos controlar. Si se produciera algún otro tipo de excepción, ésta sería capturada en el segundo bloque Catch.

## LECTURA Y ESCRITURA EN FICHEROS

### JAVA I/O APIS AND FILES OPERATIONS

#### INTRODUCTION



Los ficheros son un recurso de almacenaje u obtención de información muy útil en el desarrollo de aplicaciones. Un fichero o archivo es un conjunto de bits de almacenados en un dispositivo. Los ficheros tienen un nombre y se ubican en directorios. El nombre del fichero debe ser único en ese directorio. Además, los ficheros pueden tener una extensión. Estas suelen ser de tres letras, PDF, DOC, GIF, etc., que permiten saber el tipo de fichero. Un fichero está formado por un conjunto de registros o líneas y cada registro por un conjunto de campos relacionados. La manera en que se agrupan los datos en un fichero depende de quien lo diseñe.

#### JAVA.IO

La primera API de Java que dio soporte a operaciones input-output fue Java.io. Una clase muy útil de este paquete es la clase File, que sirve como punto de entrada a todas las operaciones con ficheros y directorios. Esta clase nos permite crear o eliminar ficheros, pero tiene algunas limitaciones, como por ejemplo copiar, mover o renombrar ficheros, operaciones muy necesarias que más adelante veremos cómo realizar.

#### JAVA.NIO

En la versión 7 del JDK se introdujo el paquete java.nio para solventar las carencias de Java.io en el manejo de ficheros. Así pues, nos centraremos en el uso de Java.nio para el manejo de ficheros en Java a lo largo de este tema.

Algunas clases que ayudan a este propósito son Path, que nos es útil para localizar un fichero o directorio usando su ruta, Files, que usado conjuntamente con Path nos permite realizar operaciones con ficheros y directorios, o File System, para trabajar con el sistema de ficheros.

An object that may be used to locate a file in a file system.  
It will typically represent a system dependent file path.

Provides an interface to a file system and is the factory for objects to access files and other objects in the file system.

```
Path path = FileSystems.getDefault().getPath("logs", "access.log");
BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF_8);
```

This class consists exclusively of static methods that operate on files, directories, or other types of files.

En este código veremos un típico ejemplo de uso de las tres clases para obtener un objeto BufferedReader que permite la lectura de un fichero. Con la ayuda de File System obtenemos un objeto Path, esencial en el manejo de ficheros o directorios en Java. Un objeto Path contiene la información utilizada para determinar la ubicación de un fichero o directorio.

Los paths pueden ser instanciados mediante rutas relativas o absolutas. En la segunda línea de código, se utiliza el path como argumento de la función estática New Buffet Reader de la clase Files. Conozcamos más detalles sobre estas clases y Java.NEO.

## FILES OPERATIONS

### Chekking a File or Directory

<code>static boolean isWritable(Path path)</code>	Tests whether a file is writable.
<code>static boolean exists(Path path, LinkOption... options)</code>	Tests whether a file exists.
<code>static boolean isExecutable(Path path)</code>	Tests whether a file is executable.
<code>static boolean isHidden(Path path)</code>	Tells whether or not a file is considered <i>hidden</i> .
<code>static boolean isReadable(Path path)</code>	Tests whether a file is readable.

Antes de acceder a un fichero o directorio, se recomienda verificar que existe y que es posible acceder a él en el modo deseado. Lectura, escritura, ejecutable... La clase Files proporciona varios métodos estáticos para conocer si existe y de qué modo es posible acceder al fichero. Puede encontrar la lista completa de métodos de la clase Files en la documentación oficial de Oracle.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {

 public static void main(String[] args) {
 Path path = Paths.get("Gerard\\test.txt");

 System.out.println(Files.exists(path)); //true
 System.out.println(Files.isExecutable(path)); //true
 System.out.println(Files.isWritable(path)); //false
 System.out.println(Files.isReadable(path)); //true
 }
}
```

Permisos de Todos	Permitir	Denegar
Control total		
Modificar	✓	
Lectura y ejecución		
Lectura	✓	
Escritura		
Permisos especiales		

Aquí tenemos un ejemplo de uso de estos métodos. Para ello, creamos un objeto de tipo PATH, que inicializamos con el método estático GET de la clase PATH, que toma como parámetro una ruta absoluta hacia un fichero llamado test.txt. Suponiendo que el fichero existe y tiene los siguientes permisos, obtenemos el siguiente resultado. El fichero existe, es ejecutable, no tiene permisos de escritura, pero sí de lectura.

### Creating a File or Directory

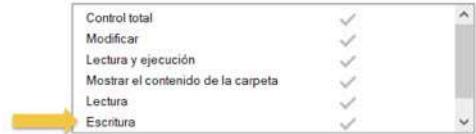
<code>static Path createDirectory(Path dir, FileAttribute&lt;?&gt;... attrs)</code>	Creates a new directory.
<code>static Path createFile(Path path, FileAttribute&lt;?&gt;... attrs)</code>	Creates a new and empty file, failing if the file already exists.

También en la clase Files encontramos los métodos CreateDirectory y CreateFiles para crear ficheros y directorios respectivamente.

```

public static void main(String[] args) throws IOException {
 Path path = Paths.get("newDirectory");
 Path newDir = Files.createDirectory(path);
 Path fileToCreatePath = newDir.resolve("newFile.txt");
 Files.createFile(fileToCreatePath);
 System.out.println("New file created: " + fileToCreatePath);
 System.out.println("New File exists: " + Files.exists(fileToCreatePath));
}

```



Aquí vemos un ejemplo donde primero creamos el directorio en NewDirectory, en esta ocasión utilizando una ruta relativa en la raíz del mismo proyecto. A continuación, utilizando el método Resolve, creamos un nuevo path, al que hemos añadido el nombre de un fichero. Y por último, con el método CreateFile, hacemos efectiva la creación del fichero.

Para poder crear ficheros y directorios, necesitamos tener permisos en el directorio destino. De lo contrario, obtendremos una IEO exception, de tipo java.nio.file.AccessDeniedException.

### Deleting File or Directory

<code>static void</code>	<code>delete(Path path)</code>	Deletes a file.
<code>static boolean</code>	<code>deleteIfExists(Path path)</code>	Deletes a file if it exists.

Para eliminar ficheros o directorios, utilizaremos el método delete o deleteIfExists. Con este segundo método, evitamos una posible excepción si el fichero o directorio no existe. Además, nos retorna un booleano indicando si se ha hecho la eliminación deseada o no. Se podrá eliminar un directorio si este no contiene otros ficheros o directorios, es decir, está vacío.

```

public void deletePath(String pathToFileOrDirecotry){
 try
 {
 Files.delete(Paths.get(pathToFileOrDirecotry));
 System.out.println("Deletion successful.");
 }
 catch(NoSuchFileException e)
 {
 System.out.println("No such file/directory exists");
 }
 catch(AccessDeniedException e)
 {
 System.out.println("Invalid permissions.");
 }
 catch(DirectoryNotEmptyException e)
 {
 System.out.println("Directory is not empty.");
 }
}

```

El siguiente código sirve para eliminar un fichero o directorio controlando las excepciones más comunes que pueden darse. Que no exista el fichero o directorio, que no tengamos permisos para realizar la eliminación o que el directorio no esté vacío.

Los paquetes Java.io y Java.nio no proporcionan ningún método que permita eliminar un directorio no vacío, así que deberemos tener presente esta limitación en nuestro código.

### Coping and Moving File or Directory

<code>static long</code>	<code>copy(Path source, OutputStream out)</code>	Copies all bytes from a file to an output stream.
<code>static Path</code>	<code>move(Path source, Path target, CopyOption... options)</code>	Move or rename a file to a target file.

Para copiar o mover ficheros o directorios se utilizan los métodos COPY y MOVE respectivamente. Ten presente que cuando se copian o mueven directorios no se copian los ficheros o directorios que se encuentran dentro de éste.

Java.nio.File StandardCopyOptions

Enum Constants	
Enum Constant	Description
ATOMIC_MOVE	Move the file as an atomic file system operation.
COPY_ATTRIBUTES	Copy attributes to the new file.
REPLACE_EXISTING	Replace an existing file if it exists.

En las operaciones de Copy o Move, es útil utilizar las opciones de copia definidas en el Enum Standard Copy Options, y que nos permite especificar el comportamiento que queremos ante determinados conflictos, por ejemplo mover el fichero de forma atómica, copiar atributos o reemplazar el fichero si éste ya existe.

```
import static java.nio.file.StandardCopyOption.*;

...
Path source = Paths.get("/users/personal/test/source/file.csv");
Path target = Paths.get("/users/personal/test/target/file.csv");
Files.move(source, target, REPLACE_EXISTING);

Files.move(source, target, REPLACE_EXISTING);
```

El siguiente código copia el fichero FileCSV del directorio source al directorio target. La siguiente instrucción mueve el mismo fichero del directorio source al directorio target donde ya existe una copia.

El uso de la est\'andar copy option ReplaceIfExists del paquete java.nio.file evit\'ara que obtengamos una excepci\'on del tipo FileNotFoundException.

## Listing Directory

static DirectoryStream<Path>	newDirectoryStream(Path dir)	Opens a directory, returning a DirectoryStream to iterate over all entries in the directory.
static DirectoryStream<Path>	newDirectoryStream(Path dir, String glob)	Opens a directory, returning a DirectoryStream to iterate over the entries in the directory.

La clase java.nio FileDirectoryStream proporciona la forma de iterar sobre el contenido de un directorio. La iteraci\'on se realiza sobre los objetos path. Podemos utilizar los m\'etodos NewDirectoryStream de la clase Files para obtener el iterador.



```

Path dir = Paths.get("Gerard");
DirectoryStream<Path> stream = Files.newDirectoryStream(dir);
for (Path file: stream) {
 System.out.println(file.getFileName());
}

//Demos
//test.txt

```

Returns the name of the file or directory denoted by this path as a Path object

En el siguiente bloque de código se lista el contenido del directorio Gerard. Obtendremos como resultado el listado de todos los ficheros y directorios que contengan la ruta especificada. El método getfilename retorna el nombre del elemento más alejado de la raíz del objeto path.

Ten presente que retorna indistintamente nombres de ficheros y directorios.

### Managing metadata

<code>static &lt;A extends BasicFileAttributes&gt; readAttributes(Path path, Class&lt;A&gt; type, LinkOption... options)</code>	Reads a file's attributes as a bulk operation.
<code>static Map&lt;String, Object&gt; readAttributes(Path path, String attributes, LinkOption... options)</code>	Reads a set of file attributes as a bulk operation.

Si en un programa se necesita obtener varios atributos de un fichero o directorio, no sería eficiente obtener de forma independiente cada uno de ellos, ya que acceder repetidamente al sistema de ficheros para conseguir un único atributo puede afectar de forma adversa el funcionamiento del programa.



```

import java.nio.file.attribute.BasicFileAttributes;
.....
Path file = Paths.get("Gerard\\test.txt");
BasicFileAttributes attr = Files.readAttributes(file, BasicFileAttributes.class);

System.out.println("creationTime: " + attr.creationTime());
System.out.println("lastAccessTime: " + attr.lastAccessTime());
System.out.println("lastModifiedTime: " + attr.lastModifiedTime());
System.out.println("isDirectory: " + attr.isDirectory());
System.out.println("isOther: " + attr.isOther());
System.out.println("isRegularFile: " + attr.isRegularFile());
System.out.println("isSymbolicLink: " + attr.isSymbolicLink());
System.out.println("size: " + attr.size());

```

Por este motivo, la clase Java Neo File Files proporciona dos métodos que permiten obtener los atributos de un fichero o directorio en una única instrucción.

En el siguiente bloque de código podemos observar cómo obtener los atributos básicos de un fichero en un objeto de tipo BasicFileAttributes, utilizando el método ReadAttributes de la clase Files.

**READING & WRITING ON TEXT FILES****INTRODUCTION**

Con frecuencia tendremos que guardar los datos de nuestro programa para poderlo recuperar más adelante. Hay varias formas de hacerlo. Una de ellas son los ficheros.

Un tipo de ficheros que resulta sencillo de manejar son los ficheros de texto. Son ficheros que podemos leer y escribir desde un programa Java y también leer y escribir con cualquier editor de textos o bien usar un programa tanto para leer como para escribir.

**READERS & WRITERS**

Las clases Reader y Writer del paquete Java.io nos permiten leer y escribir en ficheros de texto. Ambas clases disponen de una versión buffered. Veremos que estos objetos los podemos obtener utilizando los métodos NewBufferedReader y NewBufferedWriter de la clase Files. Visitemos la documentación oficial de Oracle. La clase Reader es una clase abstracta del paquete Java.io. Es la base para la lectura de ficheros en Java. Los únicos métodos que las subclases deben implementar son Read y Close, aunque muchas de las subclases sobreescriben más métodos para proporcionar una mejor funcionalidad. Vemos una de las subclases más utilizadas, BufferedReader. Esta clase implementa un buffer con el fin de proporcionar una lectura más eficiente. El método ReadLine realiza la lectura de una línea del fichero.

Retorna el contenido de la línea sin incluir los caracteres de fin de línea. Si ha alcanzado el final del fichero, retornará nul. Otro método muy popular es ReadAllLines de la clase Files en el paquete Java.neo. Este método retorna todas las líneas del fichero y lo cierra una vez ha terminado. El uso de este método no está recomendado para ficheros grandes. También en la clase Files encontramos el método NewBufferedReader, que recibiendo un path como argumento, abre el fichero en modo lectura y retorna un BufferedReader para poder leer de forma eficiente.

La clase Writer es también una clase abstracta, en este caso proporciona la base para la escritura de ficheros. Las subclases deben implementar los métodos abstractos write, close y flush. El método flush debe encargarse de comunicar al sistema operativo que realice la escritura efectiva en el fichero. Veamos BufferedWriter, una de las subclases más utilizadas. Esta subclase provee un buffer para optimizar la escritura en ficheros. El método Write escribirá en el fichero la cadena proporcionada, mientras que el método NewLine insertará un separador de línea.

**DEMO**

En este vídeo de demostración veremos un ejemplo de uso de las principales clases de lectura y escritura en ficheros de texto. Vemos que las Arrays tienen siete posiciones cada una de ellas y que los datos están relacionados entre ellos. Así, el empleado Fernández trabaja en el departamento 10 y tiene un salario de 1045. A continuación definimos el nombre del fichero datos.dat en la variable string filename. Si nos vamos al cuerpo del método main, vemos que utilizamos FileName para instanciar un objeto path que luego pasamos como parámetro al método WritingFile. Este método, que efectivamente recibe un objeto path, declara un objeto BufferedWriter, que obtiene de la llamada al método NewBufferedWriter de la clase Files. A este método le pasamos el path, el charSet y la manera como queremos escribir en el fichero, en este caso Create. Si nos vamos a la implementación de esta clase Enum obtendremos más información de qué significa exactamente cada uno de los tipos de abertura.

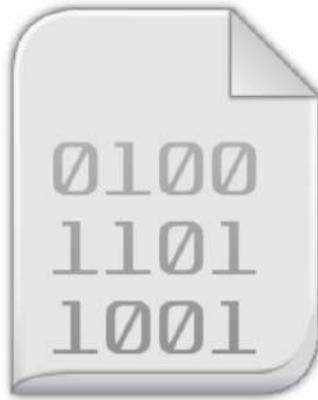
En nuestro caso se creará el fichero en caso de que no exista. Una vez tenemos el objeto BufferedWriter procedemos a iterar sobre el primero de los Arrays de Datos para ir escribiendo en el buffer, registro a registro, la información de los empleados. Una vez tenemos el buffer con los datos que queremos guardar llamamos al método Flash para hacer efectiva la escritura y luego al método Close para cerrar el fichero. Voy a poner aquí un punto de interrupción para ver cómo funciona y debugaremos este programa.

Aquí tenemos el nuevo fichero con los datos que acabamos de guardar en él. Aquí podemos verlos. A continuación llamamos al método ReadFromFile. Utilizamos un objeto BufferedReader con el que realizamos la acción de leer registro a registro el fichero que hemos generado anteriormente y mostrar la información por consola. Sigamos debugando. Aquí lo tenemos. Finalmente llamamos al método ReadAllLinesFromFiles.

Viajemos dentro de él. Utilizamos el método ReadAllLines para cargar todas las líneas del fichero en una raíz de strings. Si vamos avanzando paso a paso veremos como van apareciendo las líneas que ya tenemos precargadas en el array de strings como podemos ver aquí.

## READING & WRITING ON BYTES FILES

### INTRODUCTION



En ocasiones nuestras aplicaciones Java pueden requerir trabajar con ficheros de bytes.

Un fichero de byte o binario es un archivo que contiene información de cualquier tipo codificada en binario.

### STREAMS

Los ficheros de bytes almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario, como ocurre con los ficheros de texto. Ocupan menos espacio en disco. Las clases Input y OutputStream nos permiten realizar las operaciones de escritura y lectura con este tipo de ficheros.

Veremos que estos objetos los podemos obtener utilizando los métodos NewInputStream y NewOutputStream de la clase Files. Un stream no es más que una fuente o un destino para un bloque de información, en este caso de bytes. La clase InputStream es una clase abstracta del paquete Java.io. Es la base para la lectura de ficheros de bytes en Java. Una de las subclases más utilizadas es DataInputStream. Vemos que esta clase dispone de métodos de lectura para los diferentes tipos de datos primitivos, Boolean, Byte, Char, Double, Float, Int, Long o Short, y también dispone del método readUTF que es el indicado para la lectura de cadenas de texto. La clase files tiene el método new InputStream() que nos retorna un objeto InputStream. La clase OutputStream es también una clase abstracta del paquete java.io. En este caso es la base para la escritura de ficheros de bytes en Java empleando Streams. Una de las subclases más utilizadas es DataOutputStream.

Igual que hemos visto que DataInputStream tiene métodos específicos para la lectura de cada tipo de datos, DataInputStream también los tiene para la escritura, tenemos WriteChar, WriteDouble, WriteFloat, etc. Y al final tenemos WriteUTF, para la escritura de cadenas de texto.

## DEMO

En esta demostración veremos un ejemplo de uso de las principales clases de lectura y escritura en ficheros de bytes. En esta clase de ejemplo, tenemos tres arrays de distintos tipos de datos. Veremos que las tres arrays tienen siete posiciones cada una de ellas, y que los datos están relacionados entre ellos. El empleado Fernandez trabaja en el departamento 10 y tiene un salario de 1045 euros. A continuación definimos el nombre del fichero datos.dat en la variable string.filename.

En el cuerpo de la aplicación, usamos esta variable para instanciar un objeto path que le pasamos al método writingFile. Este método crea un objeto DataOutputStream utilizando el método newOutputStream de la clase Files, al que le pasamos el path que hemos recibido por parámetro y con la opción de apertura estándar Create. Luego, en un método que itera sobre el contenido del Array de Apellidos, utilizando los métodos de escritura de Data Output Stream según su correspondiente tipo de datos, vamos escribiendo en el fichero. Ejecutemos. Aquí tenemos el fichero.

El método ReadFromFile procederá a leerlo. Este método, utilizando un objeto DataInputStream, recupera la información del fichero hasta que salta la excepción EndOfFileException. Habremos llegado al final del fichero. Controlamos debidamente la excepción y ya terminamos. Aquí tenemos el resultado.

## RANDOM VS SEQUENTIAL ACCESS FILE

### INTRODUCTION

Las interfaces y clases que conocemos para leer o escribir con ficheros de texto y ficheros de bytes, como por ejemplo Writer, BufferedReader, OutputStream o DataInputStream, nos proporcionan un acceso secuencial, es decir, el contenido se lee o se escribe desde el inicio del fichero.



Esto nos puede resultar poco práctico si lo que queremos es acceder a un dato que se encuentra en mitad de un fichero. Existe otro tipo de acceso a ficheros que nos ayudará en este propósito, el acceso aleatorio o Random Access File que consiste en abrir el fichero y localizar una ubicación particular donde leer o escribir.

### RANDOM ACCESS FILE

La interfaz SeekableByteChannel provee métodos para establecer un apuntador a una determinada posición en la que leer o escribir. La clase ByteBuffer proporciona el soporte donde cargar los bytes que leeremos o escribiremos. Veremos que podemos obtener un objeto SeekableByteChannel llamando al método NewByteChannel de la clase Files. La interface SeekableByteChannel ofrece el soporte necesario para la lectura aleatoria en ficheros.

Al igual que las clases que dan soporte a la lectura y escritura secuencial, tiene métodos de lectura y escritura a los que pasaremos un objeto de la clase ByteBuffer de donde se obtendrán los bytes a escribir o donde se dejarán los bytes leídos. El método que permite el acceso aleatorio propiamente es Position pues mediante un número entero permite establecer exactamente el byte del fichero en que nos queremos situar.

## DEMO

En este vídeo demostración veremos un ejemplo de lectura y escritura aleatoria en un fichero. En esta clase de ejemplo tenemos definidas tres arrays de distintos tipos de datos.

Vemos que las arrays tienen siete posiciones cada una de ellas y que los datos están relacionados entre ellos. El empleado Fernández trabaja en el departamento 10 y tiene un salario de 1045 euros. A continuación definimos el nombre del fichero datos.dat en la variable filename y justo debajo la variable entera registerLength para establecer la longitud que tendrá cada registro que escribiremos o leeremos.

Vayamos al cuerpo de la aplicación. Como siempre creamos un objeto path que representará el fichero. Y lo pasamos al método WritingFile. Este método obtiene un objeto SeekableByteChannel de la llamada al método NewByteChannel de la clase Files, al que le pasamos el path y varias opciones de apertura estándar. Write para indicar que escriba al final del fichero y Create para indicar que crea un nuevo fichero. Luego creamos un ByteBuffer y con el método Allocate le indicamos cual será su capacidad. 36 bytes.

Ahora, iterando sobre la primera array, la de apellidos, limpiamos el buffer con un clear por si hubieran datos de una iteración anterior y lo vamos llenando 4 bytes para un entero, el índice 20 bytes para 10 caracteres, los apellidos 4 más para un entero, el departamento, y 8 para un double, el salario. Total, los 36 bytes del buffer. Luego llamamos al método flip para preparar la secuencia de bytes para su escritura, cosa que hacemos cuando pasamos el buffer al método write de nuestro objeto seekableByChannel.

Una vez hemos escrito en el fichero mostraremos por consola en qué posición ha quedado el apuntador del objeto seekableByChannel y seguiremos iterando. Cuando ya se han escrito todos los registros termina el método. Y hacemos la llamada al siguiente ReadFromFile. Aquí lo tenemos. De forma similar a como hemos creado anteriormente el objeto SeeCableByChannel, lo hacemos ahora pero con la opción de apertura Read. Creamos un buffer con la misma capacidad, un integer que nos permitirá mantener la posición de lectura dentro del fichero y que establecemos el cero para empezar desde el primer registro, y luego las variables del mismo tipo que queremos leer para ir recuperando la información.

En esta ocasión, iteraremos dentro de un bucle while, mientras la posición del apuntador en el objeto SeekableByChannel no haya sobrepasado su tamaño. Como antes, limpiamos el buffer por si estuviera lleno de la iteración anterior. Esta vez, llamamos al método read para cargar el buffer con un registro y rewind para establecer la posición de lectura dentro del buffer al inicio. Podemos poner un punto de interrupción en esta sentencia para ver cómo actúa cuando ejecutemos. A continuación, iremos asignando valor a las variables auxiliares con los métodos get de ByteBuffer, según su tipo de datos.

Mostraremos el registro por consola y, finalmente, desplazaremos un registro a la posición del apuntador del objeto seekableByteChannel, antes de empezar la siguiente iteración. Debuguemos. Nos hemos parado justo después de la lectura. Acabamos de llenar el buffer con 36 bytes, exactamente su capacidad, y tenemos su atributo Position apuntando al último byte. Aquí podemos verlo. El método Rewind setea a cero este apuntador. De esta forma, a medida que vayamos leyendo bytes, el apuntador se va desplazando. Es lo mismo que sucedía con el apuntador del objeto seekableByteChannel.

Como podemos ver en la salida por consola del método ReadFromFile. Si ahora continuamos con la ejecución vemos que iteración tras iteración se va recuperando y escribiendo registro a registro. Quitemos este y coloquemos otro punto de interrupción antes de la llamada al último método. A este método, aparte del fichero que queremos leer, le pasamos el registro que queremos leer, el 5. Vayamos allá.

Vemos que después de establecer el objeto SeekableByteChannel en modo lectura y el correspondiente buffer, con el tamaño de un registro, se establece la posición que queremos leer donde esperamos encontrar el quinto registro. E igual que antes, recuperaremos y luego mostraremos el registro por consola. En esta ocasión no encontramos ningún bucle de lectura sobre el objeto SeekableByteChannel puesto que sólo hemos querido recuperar un registro. Terminemos la ejecución del programa. Aquí tenemos el registro que hemos recuperado con acceso aleatorio.

## READING & WRITING ON XML FILES

### INTRODUCTION



Los ficheros XML se pueden utilizar para proporcionar datos a una base de datos o para almacenar copias de partes del contenido de la base de datos. También se utilizan para escribir ficheros de configuración de programas. En el protocolo SOAP, Simple Object Access Protocol, para ejecutar comandos en servidores remotos. Java dispone de varias herramientas para leer y escribir información en archivos XML.

Operaciones que también se conocen como Parsear. Tenemos DOM, Document Object Model, que es la API que veremos en detalle en este vídeo, la Simple API XML, el Framework Java Architecture for XML Binding y JDOM, un Document Object Model especialmente diseñado para Java.

### DOM

El DOM, o Modelo de Objetos de Documento, almacena toda la información del documento en memoria en forma de árbol, con nodos padre, nodos hijo y nodos finales, los que no tienen descendientes.

Una vez creado el árbol, se van recorriendo los diferentes nodos y se analiza a qué tipo particular pertenecen. Este procesamiento consume bastantes recursos de memoria y tiempo, sobre todo si los ficheros XML a parsear son grandes y complejos. El paquete javax.XML.Parsers nos proporciona la funcionalidad necesaria para abrir ficheros XML, mientras que el paquete org.w3c.dom nos aporta los elementos que conforman el DOM.

Como en toda operación de tratamiento de ficheros, haremos uso de los paquetes Java.io o Java.nio. En el paquete javax.xml.parsers encontramos las clases DocumentBuilderFactory y DocumentBuilder. El método newDocumentBuilder del documentBuilderFactory nos retorna un DocumentBuilder. Un objeto de esta clase nos servirá para iniciar el parseo del documento que le pasaremos por parámetro al método parse, que nos retornará un document. Otra interfaz indispensable, en este caso del paquete org.w3c.dom.

En este paquete encontramos las interfaces Document, Element y ATTR. Los documentos representan el fichero XML, los elementos los nodos y ATTR los atributos que pueden tener los nodos. Recomiendo explorar estas interfaces para conocer los distintos métodos Sets, Gets o Appends que ofrecen para manipular la información que contendrán los objetos de estos tipos.

## DEMO

En esta demostración veremos cómo parsear ficheros XMR utilizando DOM. En esta clase de ejemplo tenemos definidas tres arrays, tres distintos tipos de datos. Vemos que las arrays tienen siete posiciones cada una de ellas y que los datos están relacionados entre ellos. El empleado Fernández trabaja en el departamento 10 y tiene un salario de 1045 euros.

A continuación definimos el nombre del fichero datos.xml en la variable string.filename. Si nos vamos al cuerpo del método main, vemos que utilizamos filename para instanciar un objeto path, que luego pasamos como parámetro al método WritingFile. Este método utiliza una instancia de DocumentBuilderFactory para crear un objeto DocumentBuilder, que a su vez es utilizado para obtener un objeto Document. A este documento le creamos un elemento pasando un string que será su tagName. Este primer elemento, o nodo del documento, será la raíz, a la que dentro de un bucle iremos añadiendo más elementos, en este caso con tagName Employee. A Employee le creamos un atributo para informar su identificador. Y luego le añadimos 3 nodos hijo para cada uno de los elementos correspondientes a los tres arrays, empleados, departamento y salario. Una vez hemos añadido todos los elementos, salimos del bucle y utilizamos varios objetos del paquete javaX.XML.Transform.

Una instancia de TransformFactory, un objeto Transformer, un objeto DOMSource y un StreamResult, al que pasamos el fichero donde queremos escribir. De este modo, escribimos la estructura que hemos creado en el documento dentro del fichero datos.xml. Aquí tenemos el fichero que hemos creado. Veamos ahora el primer método de lectura. Esta vez, al objeto Builder le pasamos el fichero. Tenemos que llamar al método Normalize para que la estructura del documento quede organizada y lista para poder navegar. Ahora iremos obteniendo elementos, empezando desde la raíz y recuperando elementos y atributos de su estructura que ya conocemos, dentro de este bucle. Ejecutamos para ver el resultado. Finalmente, nos puede interesar abrir documentos de los que no conocemos su estructura.

Este último método es un ejemplo de este caso. Creamos y normalizamos el documento igual que antes. Pero en este caso llamamos a una función recursiva que como vemos aquí debajo se va adentrando en el árbol de nodos buscando y mostrando atributos o nodos hijos de cada elemento.

## READING & WRITING ON JSON FILES

### INTRODUCTION

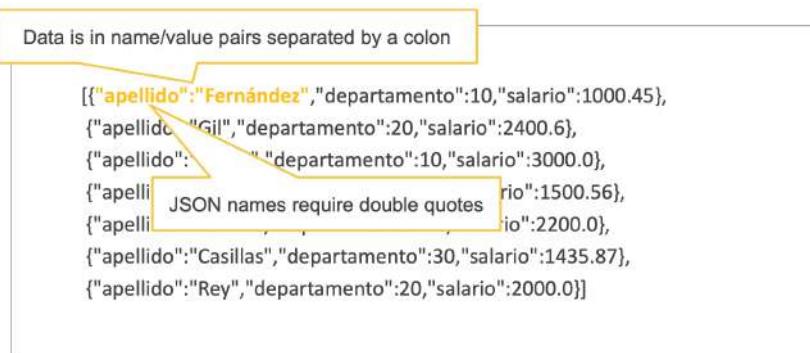


JSON es hoy en día un formato de fichero muy empleado en el manejo de datos, sobre todo entre los desarrolladores web, puesto que precisamente es un formato que surgió de la necesidad de almacenar datos estructurados en aplicaciones JavaScript.

El acrónimo JSON significa JavaScript Object Notation. Al igual que XML, es un estándar abierto y son ficheros que podemos abrir con un blog de notas y comprender su contenido.

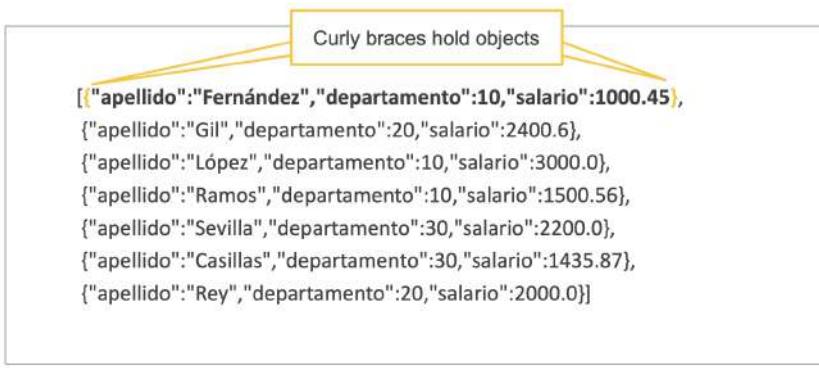
Los ficheros JSON son muy eficientes en el intercambio de información gracias a su estructura simple. Por ejemplo, en comparación con XML, donde el proceso de parseo puede llegar a consumir muchos recursos trabajando con grandes volúmenes de datos. Por el contrario, la sencillez de JSON delimita el modelado de los datos y a estructuras muy simples. En este aspecto, XML proporciona mayores ventajas.

### JSON Syntax



En esta captura se muestra el contenido de un fichero JSON sencillo, donde podremos ver las reglas de sintaxis JSON que derivan de la notación JavaScript. La información se estructura en pares clave-valor, separados por dos puntos.

Destacar que a diferencia de JavaScript, donde se permite el uso de comillas simple, en JSON siempre escribimos valores string utilizando comillas dobles. También utilizaremos comillas dobles en los identificadores de todos los atributos.



Los datos que conforman todos los atributos de un objeto están contenidos por unas llaves.



Los datos están separados por comas, ya sea entre los atributos de un mismo objeto o entre un objeto y el siguiente.



Cuando almacenamos más de un objeto, lo hacemos en forma de array. Para delimitar estos arrays de objetos, utilizaremos corchetes.

## PACKAGE JAVAX.JSON

Java no ofrece soporte nativo para trabajar con ficheros JSON. Para ello debemos añadir un paquete de utilidades. Nos podemos descargar, por ejemplo, JavaXJson. Esta API nos proporcionará las clases necesarias para el conveniente manejo de los datos.

Para representar los valores que leeremos o escribiremos, utilizaremos objetos JSONArray. Este objeto inmutable lo obtendremos de la clase JSONArrayBuilder en las operaciones de escritura, y luego, utilizando una instancia de JsonWriter, escribiremos su contenido a fichero. En las operaciones de lectura, obtendremos el objeto JSONArray con el contenido del fichero directamente de una instancia JSONReader.

## JSON Objects

Otras dos interfaces a tener en cuenta son JSONObject y JSONObjectBuilder. En el código de ejemplo, podemos ver cómo a partir de una instancia de JSONObjectBuilder al que podemos ir añadiendo varios atributos de distintos tipos de datos, construimos finalmente un JSONObject. Las instancias de JSONObject también son inmutables y representan el conjunto de atributos de un objeto en JSON. Si visitamos la documentación oficial de JSONObject, vemos que podemos obtener también un objeto JSONObject de la llamada al método readObject de la clase JsonReader.

```
JSONObjectBuilder json_object_builder = Json.createObjectBuilder();
json_object_builder.add("apellido","Fernando");
json_object_builder.add("departamento",10);
JSONObject jsonObject = json_object_builder.build();
```

Utilizaremos esta clase en las operaciones de lectura. Comentar también que un valor Json puede ser un string, puede ser también de tipo int o incluso un array, como vemos en este código de ejemplo. De este modo, podemos crear estructuras anidadas.

## DEMO

En esta demostración, veremos cómo leer y escribir información desde ficheros JSON. Antes de empezar, fíjemonos que la clase no compila. Es debido a que no disponemos ninguna API para manejo de Json. Podemos descargar la librería javax.json de la web jar-download.com Una vez descargado el fichero, depositamos el JAR en la carpeta de nuestro proyecto. Para mantener ordenado, en nuestro proyecto podemos crear una carpeta. Volvemos a nuestro proyecto , y en file project structure añadimos la correspondiente librería y ahora ya podemos empezar. En esta clase de ejemplo tenemos definidas tres arrays de distintos tipos de datos.

Vemos que las arrays tienen siete posiciones cada una de ellas y que los datos están relacionados. El empleado Fernández trabaja en el departamento 10 y tiene un salario de 1045 euros. A continuación definimos una variable String, FileName, con el nombre de un fichero, datos.json. Si nos vamos al cuerpo del Main, vemos que utilizamos este String FileName para instanciar un objeto path, que luego le pasaremos al método writingFile. En este método utilizaremos un outputStream de la librería Java.io. Luego declaramos un objeto Json\_array que de momento inicializaremos a null, también creamos una instancia de jsonArrayBuilder que obtenemos de la llamada al método static de la api de JSON, y por último declaramos un objeto jsonBuilder. En este bucle añadimos al objeto jsonArrayBuilder el retorno de la llamada a una función auxiliar, que tenemos aquí debajo. Vemos que retorna un objeto json construido mediante el uso de un jsonObjectBuilder al que vamos añadiendo,

uno a uno, los atributos correspondientes a cada registro extraídos de las Arrays apellido, departamento y salario. Cuando tenemos el ArrayBuilder lleno, inicializamos el objeto OutputStream, que hemos declarado anteriormente utilizando el path que tenemos apuntando al fichero Json. Y ahora sí, con este outputStream creamos un objeto JsonWriter que utilizamos para escribir en el fichero el contenido que hemos cargado anteriormente en el objeto JsonArrayBuilder.

Por último, cerramos todos los recursos. Ponemos un punto de introducción aquí y ejecutamos. Lo paramos y vemos que se nos ha creado un objeto llamado datos JSON con el contenido de las arrays apellido, departamento y salario. De este modo vemos mejor los siete registros. Después llamaremos a la función ReadFromFile. Ahora usaremos un input stream, también de la librería java.io. y lo inicializaremos con el path que apunta al fichero JSON donde antes hemos guardado los datos de los Arrays. Utilizamos este InputStream para crear un JSONReader y de la llamada a su método readArray obtenemos un JSONArray que podremos recorrer mostrando por consola como hacemos en este bucle.

Vemos que instanciamos cada elemento del Array como un JSONObject y luego vamos obteniendo los atributos por nombre, utilizando el correspondiente Get según su tipo de datos, para en este caso mostrarlo por pantalla. Ejecutemos. Y aquí tenemos el resultado de la lectura del fichero JSON.

**M3**

**PROGRAMACIÓN**

**UF**

**6**

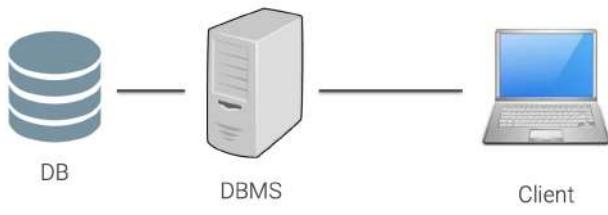
## APLICACIONES CON BBDD RELACIONALES Y ORIENTADAS A OBJETO

### DATABASE ACCESS INTRODUCTION

#### INTRODUCTION

Para que la información que manejan nuestras aplicaciones sea persistente, es decir, no se pierda una vez terminado el programa, sabemos que podemos almacenar los datos en ficheros.

Pero esta solución se queda corta si queremos operar con cantidades grandes de datos o si son diversas aplicaciones ejecutándose simultáneamente en distintos equipos las que tienen que acceder a estos datos.

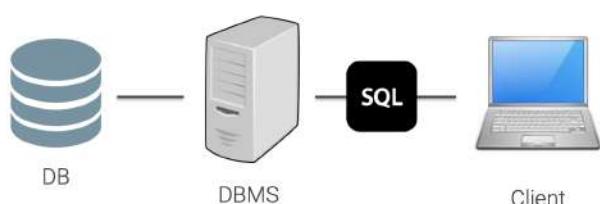


En estos casos, es más eficiente usar una base de datos, una poderosa herramienta que nos aportará fiabilidad y consistencia en el manejo de la información. Los datos que queremos guardar o consultar en una base de datos se encuentran en un servidor que opera un software llamado Sistema Gestor de Bases de Datos, en inglés Database Management System.

Un Sistema Gestor de Bases de Datos es un conjunto de programas que permiten el almacenamiento, modificación y extracción de la información en una base de datos y con el que nuestra aplicación, ejecutándose en un cliente, deberá interaccionar. Veremos que para realizar esta comunicación hay lenguajes específicos dependiendo del tipo de sistema gestor de bases de datos.

#### DBMS TYPES

##### Relational Database



Hay varios tipos de sistemas gestores de bases de datos según la manera como almacenan la información.

El más común es la base de dato relacional, que consiste en conjuntos de una o más tablas, estructuradas en registros y campos, que podemos entender como líneas y columnas. Esta información se vincula entre sí por un campo en común.

A este campo en común se le denomina identificador o clave, y también permite relacionar información de diferentes tablas. MySQL y SQL Server son las bases de datos relacionales más utilizadas.

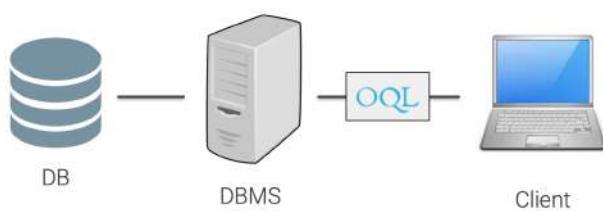
```

UPDATE clause {UPDATE country
 expression
SET clause {SET population = population + 1
 expression
WHERE clause {WHERE name = 'USA';
 predicate
}
}

```

<https://www.w3schools.com/sql/>

## Object-relational Database



Otro modo de almacenar la información es en una base de datos objeto-relacional. Este sistema utilizará también el modelo relacional, pero en este caso almacenando en sus tablas directamente los objetos. Oracle y PostgreSQL son dos ejemplos de base de datos objeto-relacional.

## Communicate with RBD and ORBD

El SQL, del inglés Structured Query Language, es un lenguaje de consulta estructurado diseñado para administrar y recuperar información de sistemas de gestión de bases de datos relacionales y objetos relacionales.

### SQL Statements

Cada fabricante de bases de datos ofrece su propia documentación del uso específico del SQL en su sistema, pero todos ellos tienen la base común definida por el estándar. Su sintaxis comprende cláusulas, expresiones y predicados para construir sentencias.

W3schools ofrece un buen tutorial para conocer cómo construir sentencias.

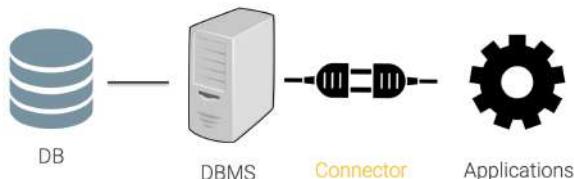
## Object Oriented Database

Un tercer modelo son las bases de datos orientadas a objetos que operan tal y como se hace en memoria, usando directamente las referencias de objetos.

## Communicate with OODB

El OQL del inglés Object Query Language es un lenguaje de consulta diseñado para administrar y recuperar información de sistemas de gestión de bases de datos orientadas a objetos. Cada fabricante tiene su propia definición del lenguaje.

## CONNECTING FROM APPS

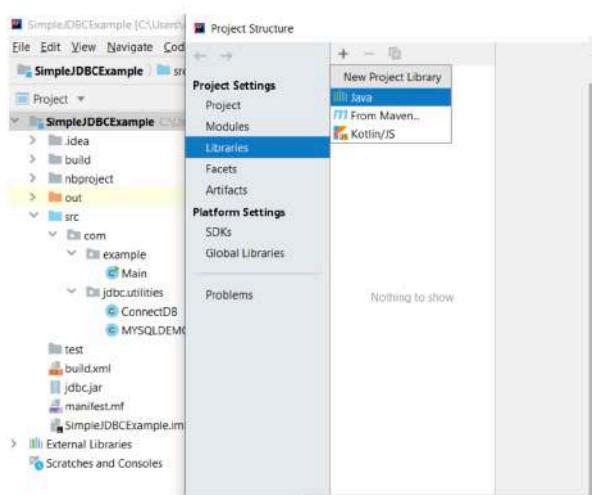


Para conectar a las bases de datos desde aplicaciones necesitamos drivers o conectores. Estos drivers deberán ser específicos para el fabricante de la base de datos. Por ejemplo, si estamos utilizando un driver para conectar a una base de datos MySQL, este mismo driver no nos servirá para conectar a una base de datos de Oracle o NeoDatis.

## Example: Download MYSQL Connector

Veamos a continuación, a modo de ejemplo, cómo instalar un conector MySQL en un proyecto IntelliJ. En primer lugar, necesitamos el conector o driver para MySQL.

Para obtenerlo, podemos descargarnos el paquete de utilidades de MySQL de su web oficial, que incluye la base de datos MySQL, una herramienta para administración con interfaz gráfica y varios conectores para distintos lenguajes de programación.



O si ya tenemos un servicio MySQL corriendo en nuestra máquina, como podría ser el que incorpora el paquete de programario exam, entonces podemos optar por simplemente descargar el fichero.jar e incorporarlo a nuestro proyecto.

Un fichero.jar o Java Archive es un paquete de ficheros de clases que permiten agregar funcionalidades no integradas originalmente en el JDK. Una vez tengamos el conector, debemos agregarlo al proyecto de IntelliJ.

Desde el menú Files, abrimos la ventana de Project Structure, y en la sección de Libraries, añadiremos el fichero.jar como una nueva librería Java.

Descarguemos directamente el archivo.jar de la página de Java 2S. Lo extraigo y me lo llevo al escritorio.

## ADD DRIVER TO THE PROJECT

El código de este proyecto IntelliJ simplemente comprueba que esté disponible el driver JDBC para MySQL. Si ejecutamos, vemos que nos da un error, puesto que no encuentra la clase. Debemos pues agregar el fichero.jar que anteriormente nos hemos descargado. Lo hacemos desde la opción Project Structure y en la sección Libraries añadimos el .jar que tenemos en el escritorio. Lo escogemos y ahora volvemos a ejecutar el proyecto. Y ahora ya no nos da el error. Tenemos nuestro proyecto listo para conectar con una base de datos MySQL.

## RELATIONAL DATABASE

### INTRODUCTION

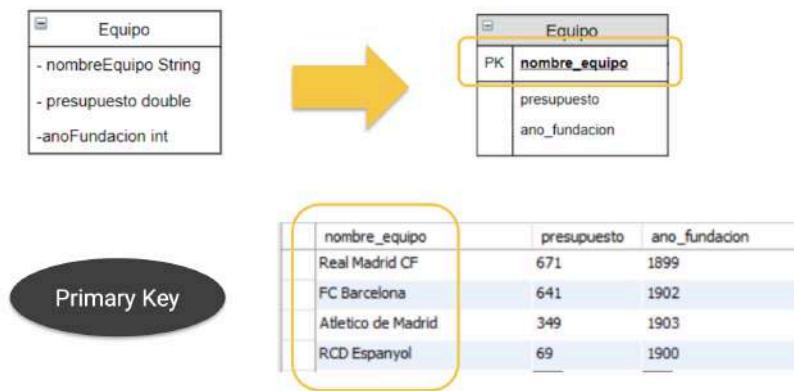
#### Relational model

Una base de datos relacional se compone de varias tablas, denominadas relaciones. No pueden existir dos tablas con el mismo nombre. Cada tabla es a su vez un conjunto de campos, columnas y registros, filas. Cada tabla debe tener una clave o identificador único. Cada tabla representa una entidad o tipo de datos. Las filas representan una instancia de ese tipo de datos.

Las columnas representan los valores o atributos de esa instancia. La relación entre una tabla padre y un hijo se lleva a cabo por medio de las claves primarias o claves foráneas o ajenas. Las claves primarias son la clave principal de un registro dentro de una tabla, y éstas deben cumplir con la integridad de datos. Las claves ajenas se colocan en la tabla hija. Contienen el mismo valor que la clave primaria del registro padre. Por medio de éstas se hacen las formas relacionales.

### MODEL TO BDR TRANSLATION

Así pues, en la traducción de nuestra aplicación orientada a objetos a un modelo relacional, cada clase instanciable se materializa en una tabla en la base de datos.



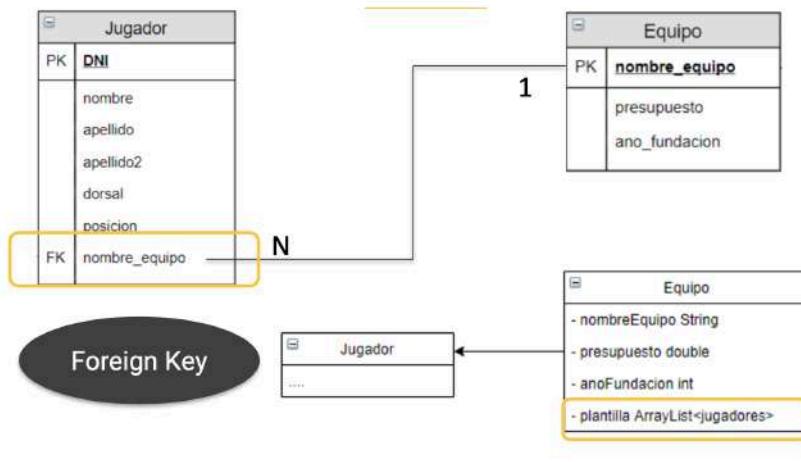
Cada atributo definido en una clase se materializa en la base de datos como una columna dentro de su tabla. La persistencia de cada objeto del modelo se materializa en la base de datos en una fila dentro de la tabla que corresponde a su clase.

En cada celda de la tabla se almacena el valor que cada objeto en concreto tiene asignado al atributo.

Para buscar un objeto concreto en una tabla es muy importante que siempre haya un atributo único, de forma que no haya ambigüedad. La columna que almacena este identificador único es la clave primaria de la tabla.

### 1-N relations

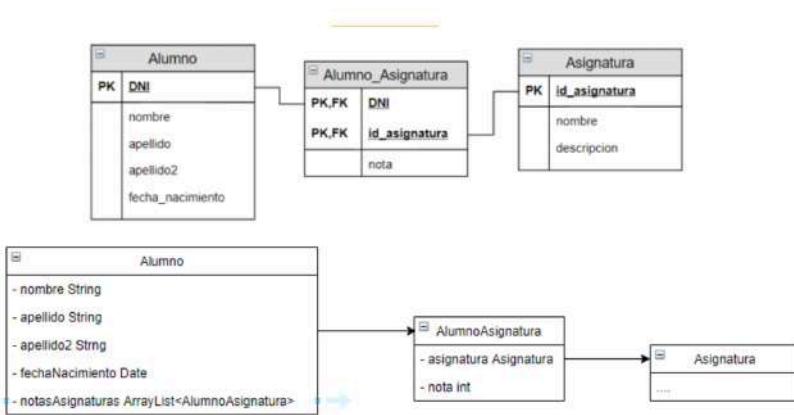
Esta clave primaria sirve también para establecer relaciones con otras tablas.



Como vemos en la imagen, jugador se relaciona con equipo mediante una foreing key, que toma el valor de la clave primaria de la tabla con la que se relaciona. De este modo, un jugador puede jugar en un equipo, mientras que un equipo puede tener N jugadores. Así podría ser el diagrama de clases de nuestro proyecto para representar esta relación, donde la clase Equipo tiene como atributo una colección de jugadores.

### N-M relations

Combinando claves primarias con una entidad débil, podemos establecer relaciones N-M.

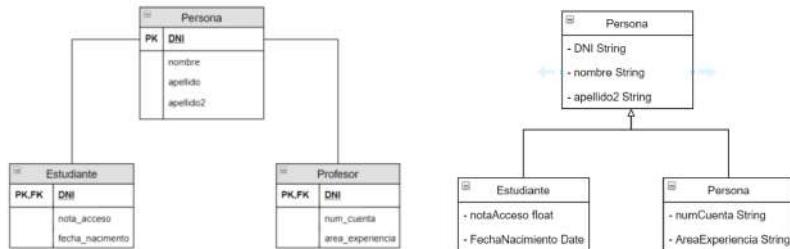


Una entidad débil es aquella que no puede existir sin participar en una relación, es decir, aquella que no puede ser únicamente identificada solamente por sus atributos.

Un alumno puede cursar varias asignaturas, mientras que una asignatura puede tener varios alumnos matriculados. Pero cada alumno tendrá una única nota de la asignatura.

El diagrama de clases en nuestra aplicación podría ser, por ejemplo, una clase asignadura asociada por alumno asignatura, que a su vez es asociada por la clase alumno, que contendría una colección de alumno asignadura con cada nota.

### Generalization



Y también podemos representar conceptos de herencia de la programación orientada de objetos, como la generalización. Persona concentra los atributos comunes en las entidades estudiante y profesor, que también cuentan con la primary key de Persona para poder establecer la relación.

## CONNECT TO MYSQL DATABASE

### PACKAGE JAVA.SQL

Para realizar operaciones con una base de datos, necesitamos en primer lugar, abrir una conexión. En el paquete Java SQL, encontramos la interfaz Connection. En el contexto de un objeto de este tipo, se producirá toda la comunicación con la base de datos. Para obtener una instancia de Connection, usaremos la clase Java.SQL.DriverManager. Concretamente, usaremos el método estático GetConnection y le pasaremos los datos de conexión a la base de datos.

Este método puede lanzar una excepción explícita que deberemos controlar, concretamente del tipo SQLException, también definida en el paquete JavaSQL. Si nos vamos a visitar la documentación oficial de Oracle sobre la clase DriverManager, ya nos anuncia qué es el servicio básico para gestionar los controladores JDBC. Si nos vamos a su lista de métodos, vemos que todos son estáticos.

El más utilizado sin duda es GetConnection, que dispone de tres sobrecargas, que retornan una conexión. Connection es una interfaz que representa la conexión a través de la cual nos comunicamos con la base de datos, enviándole sentencias SQL. El método CreateStatement retorna un objeto Statement que usaremos para lanzar sentencias en SQL contra la base de datos. Si las sentencias que queremos lanzar van acompañadas de parámetros, será más conveniente usar un objeto PreparedStatement, que está optimizado para este fin.

Lo obtendremos del método PrepareStatement. Como vemos, tanto CreateStatement como PrepareStatement pueden arrojar excepciones de tipo SQL.

### DEMO

En esta clase de ejemplo, establecemos una conexión con una base de datos MySQL. En primer lugar, comprobamos que tenemos el driver disponible en nuestro proyecto. Y luego utilizaremos esta clase auxiliar ConnectionDB para obtener la instancia de la conexión que queremos.

Esta clase utiliza el patrón Singleton. Esta técnica consiste en definir el constructor como privado y ofrecer un método estático que retorne la instancia del tipo deseado. Aquí lo tenemos. De este modo, nos aseguramos que sólo tendremos una única conexión activa en la base de datos.

Si existe, la retornará, y si no existe, la creará de nuevo. Para crear la conexión utilizamos el método `GetConnection` de la clase `DriverManager`. Le pasamos los datos de conexión, URL, Username y Password, que tenemos definido en otra clase en variables estáticas, como podemos ver aquí. En este caso, los parámetros de conexión contra mi base de datos MySQL son los que venían por defecto. El username es root y la password está en blanco.

En esta clase, también usamos un método para cerrar la conexión. Debemos llamarlo cuando hayamos terminado el uso de la conexión. Como podemos ver aquí, en el bloque Finally, pondré aquí un punto de interrupción y debuguemos. Entremos en el método que nos proporcionará la instancia de la conexión.

Como la instancia es NULL, llamaremos a `GetConnection` para obtener otra. Vemos que `GetConnection` nos ha lanzado una `SQLException`. El mensaje nos dice que la conexión ha sido rechazada. Esto es porque no tengo abierto el servicio MySQL y la base de datos no está disponible.

Desde el panel de exam, inicio el servicio. E intento de nuevo establecer la conexión. Ahora tengo abierta una conexión contra mi base de datos local. Ahora realizaría las operaciones necesarias contra la base de datos. Y al terminar debo cerrar la conexión. Ahora la conexión está cerrada.

## QUERY TO MYSQL DATABASE

### PACKAGE JAVA.SQL

Las consultas a una base de datos las haremos mediante sentencias de lenguaje SQL, a las que llamamos queries. Para lanzar estas queries necesitamos un objeto `Statement` que obtendremos de la conexión preestablecida. Este objeto nos devolverá un objeto `ResultSet` como resultado de la consulta. Ambos objetos, `Statement` y `ResultSet`, se encuentran en el paquete Java SQL. La interface `Statement` representa las sentencias SQL que queremos ejecutar contra la base de datos. Obtenemos una instancia de este objeto desde una conexión.

Encontramos varios métodos para ejecutar una consulta SQL que proporcionaremos mediante un objeto `String`. `ExecuteQuery` nos retorna un objeto `ResultSet`. Un `resultSet`, o conjunto de resultados, contiene los resultados de una consulta SQL. Lo podemos ver como un conjunto de filas o registros, pero también contiene metadatos sobre la consulta y los tamaños de cada columna.

En su lista de métodos, vemos varios GETs que permiten el acceso a las diferentes columnas de las filas. El método `Next` se usa para moverse a la siguiente fila del Result Set, convirtiendo a esta en la fila actual.

### DEMO

En este vídeo de demostración, veremos cómo obtener información almacenada en una base de datos. Concretamente, consultaremos el contenido de esta tabla que tengo en mi base de datos local en el esquema o catálogo MySQL Demo. Tiene esta única tabla, `Products`, con tres registros.

En primer lugar, comprobamos que tenemos el driver disponible en nuestro proyecto, y luego obtenemos la conexión de nuestra clase `Connect TV`. En una variable de tipo `String`, escribimos en lenguaje SQL la consulta que queremos realizar, en este caso obtener toda la información de la tabla `Products`. Luego, obtenemos de la conexión un objeto `Statement`, invocando al método `CreateStatement`.

Utilizamos este objeto para lanzar la consulta mediante el método ExecuteQuery, al que por parámetro le pasamos la consulta SQL. De esta llamada, obtenemos un objeto ResultSet, con todos los registros que retorne la consulta, sobre el que podemos iterar para ir mostrando la información por consola.

Aquí lo vemos. Dentro de este bloque Try, tenemos varios puntos donde se puede obtener una excepción de tipo SQLException, al crear la conexión, al obtener el objeto Statement, al lanzar la query y al recorrer el ResultSet. Aquí debajo, capturaremos estas posibles excepciones. Pongamos un punto de interrupción y debuguemos.

En el objeto Statement podemos ver la conexión a la que está asociada el objeto que conecta con la base de datos Localhost y que nos encontramos atacando al catálogo MySQL-Demo. Después de lanzar la query ya tenemos inicializado el objeto ResultSet. Si lo inspeccionamos, veremos que ha obtenido datos de dos campos, got y name. En RawData están los registros que ha obtenido de la consulta.

Tenemos tres registros de los que podemos leer los bytes. A medida que vayamos iterando, veremos que thisRow va tomando el valor del siguiente registro. Vemos que los valores van cambiando. Una vez recorrido todo el ResultSet, cerramos la conexión y terminamos. Si volvemos a ejecutar, vemos cómo hemos obtenido los registros que teníamos en la base de datos.

## INSERT, UPDATE, DELETE & DDL STATEMENTS TO DATABASE

### PACKAGE JAVA.SQL

Retrieving data from DataBase

```
String query = "SELECT COUNT(COD) FROM PRODUCTS";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

 [java.sql.Statement.executeQuery](#)

Sabemos que con el método executeQuery del objeto Statement podemos obtener información de la base de datos.

Pero este método solo acepta sentencias de este tipo, es decir, que solo obtengan información de la base de datos sin alterar su contenido ni estructura.

Altering DataBase data and estructure

INSERT	DELETE	UPDATE
CREATE	ALTER	DROP
GRANT	REVOKE	

 [java.sql.Statement.executeUpdate](#)

Pero si lo que queremos es modificar la información que contiene, bien insertando nuevos registros, eliminándolos o actualizándolos, o incluso lanzar sentencias DDL, del inglés Data Definition Language, para crear una nueva tabla, nuevos atributos o cambiar permisos, deberemos utilizar el método ExecuteUpdate.

```
String query = "INSERT INTO PRODUCTS VALUES (2,'Cookies 2')";
Statement stmt = con.createStatement();
stmt.executeUpdate(query);
```

 [java.sql.Statement.executeUpdate](#)

Este método nos retornará un entero con el número de filas afectadas por la ejecución de la sentencia, número de filas insertadas, número de filas eliminadas o número de filas modificadas.

Si lanzamos sentencias que no afectan registros, recibiremos un cero.

## DEMO

En este vídeo demostración veremos cómo realizar operaciones de modificación de datos o de estructura en una base de datos MySQL. El código que vemos en pantalla comprueba en primer lugar que dispongamos del driver JDBC. Y luego abre un bloque try con recursos. Este tipo de cláusula try nos permite inicializar objetos que una vez finalizado el bloque se cierran de forma automática. Nos podríamos ahorrar de este modo la cláusula finally, pero hay que tener en cuenta que los objetos aquí inicializados deben implementar la interface java.lang.AutoCloseable, condición que cumplen tanto el buffer reader como Connection. En nuestro caso, incluiremos igualmente la cláusula Finally para controlar mejor el cierre de la conexión.

A continuación, llamamos al método CreateTable, donde mediante una sentencia SQL Create que lanzamos utilizando el método ExecuteUpdate, crearemos la tabla Products si esta no existe ya. Luego entraremos en un método que muestra un menú al usuario. Este menú permite lanzar sentencias de consulta, lo que conocemos como queries, que ejecutamos con el método ExecuteQuery. En este menú también hay la opción de insertar un registro a la tabla Products.

En este caso utilizaremos el método ExecuteUpdate. Añadamos aquí un punto de interrupción y aquí tenemos otro. Debuguemos. Parémonos aquí justo antes de lanzar la creación de la tabla contra la base de datos. Vemos que la base de datos está completamente vacía, no tiene ninguna tabla. Y ahora ya vemos que la tabla Product se ha creado, pero no tiene registros. Sigamos. Ahora vemos el menú y escogeremos la opción de insertar un registro. Nos paramos aquí, avanzamos Y vemos que efectivamente se nos ha insertado el registro a la tabla Products.

## PREPARED STATEMENTS

### QUERYING WITH PARAMS

Bad practice

```
System.out.print("Enter user to search:");
employeeId = in.readLine();
String query = "SELECT * FROM Employee WHERE ID = "+ employeeId +";";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);

// employeeId → 1 returns the register with ID 1

// employeeId → 1 OR 1=1 returns all employee registers
```

SQL-injection

Si necesitamos generar una consulta en tiempo de ejecución con parámetros aportados por el usuario, podríamos hacerlo como vemos en pantalla, concatenando una variable string en medio de la sentencia.

En este caso, si el usuario pide el registro con identificador 1, obtendremos precisamente ese registro. Pero con un poco de imaginación, esta interfaz también podría ser utilizada para obtener toda la información de la tabla EMPLOYEE, añadiendo otra condición que modifique el sentido original de la consulta que teníamos previsto. Esta vulnerabilidad, conocida como SQL Injection, es muy común y supone un grave peligro para la integridad y confidencialidad de una base de datos.

Veamos un ejemplo práctico. Este código solicita un identificador de cliente al usuario y lo utiliza para construir la query que retornará todos los datos de este cliente. Aquí lo vemos. Ejecutemos. Podemos comprobar que el programa cumple su cometido. Veamos ahora qué ocurre si intentamos una inyección SQL.

La condición 1 igual a 1 siempre será cierta. Concatenada a un operador lógico OR provocará que todos los registros de la tabla sean devueltos. Para evitar este problema, es necesario que el código de nuestros programas procese cualquier cadena de texto que dependa de una entrada del usuario, antes de que entre a formar parte de una sentencia SQL.

## PREVENT SQL INJECTION

Good practice

```
System.out.print("Enter user to search:");
employeeId = in.readLine();
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pstmt = con.prepareStatement(query);
pstmt.setDouble(1, employeeId);
ResultSet rs = stmt.executeQuery();

// employeeId → 1 returns the register with ID 1

// employeeId → 1 OR 1=1 gets a NumberFormatException:
```

Las bases de datos relacionales, como MySQL, soportan sentencias preparadas. Son una buena solución para prevenir los problemas de SQL Injection. La clase PreparedStatement aporta la funcionalidad necesaria para trabajar con sentencias preparadas.

Igual que Statement, obtendremos un objeto PreparedStatement de la conexión. Cuando definamos la query, especificaremos un interrogante en las posiciones donde tengamos previsto ubicar un parámetro. Y con método set, según el tipo de datos, especificaremos el índice del parámetro y su valor. De esta forma, si el valor aportado por el usuario no coincide con el tipo esperado, obtendremos una excepción en lugar de exponer nuestra base de datos a posibles intrusiones malintencionadas.

Volvamos al ejemplo anterior. En esta ocasión, utilizaremos un objeto PreparedStatement en lugar de un statement. También hemos cambiado la query. En lugar de concatenar directamente el dato proporcionado por el usuario, colocamos un interrogante en esa posición.

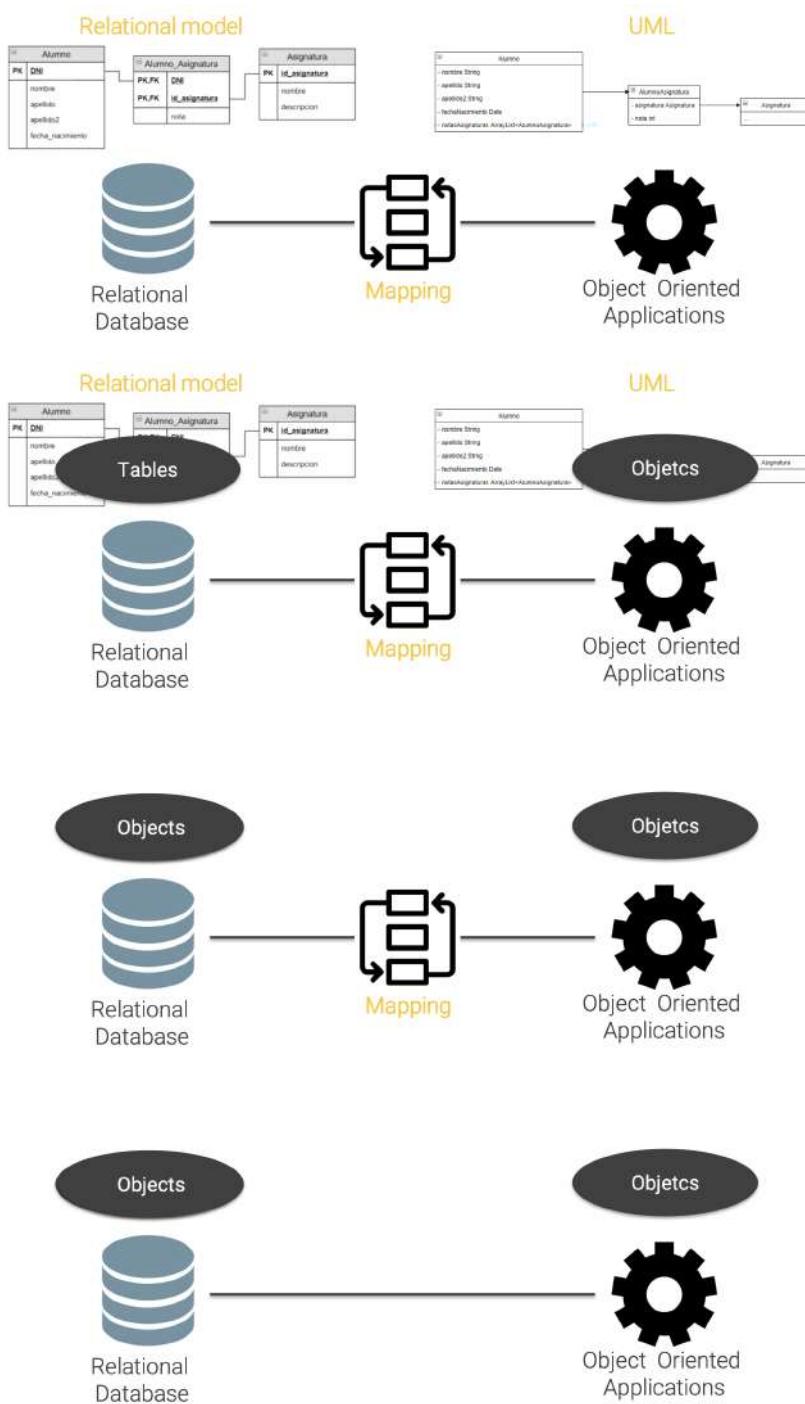
El parámetro lo informamos luego, utilizando el método setDouble, puesto que este es el tipo de dato que esperamos que el usuario introduzca. El resto del código es igual al ejemplo anterior. Ejecutemos. Sigue funcionando para un dato correcto.

Pero si intentamos la inyección SQL obtenemos un error NumberFormatException. Vemos que el error nos salta ya cuando leemos el valor, pero también nos saltaría en el método setDouble, puesto que acepta únicamente valores numéricos. De este modo, tenemos nuestra aplicación preparada para prevenir ataques de SQL e Inyección.

# APLICACIONES CON BBDD RELACIONALES Y ORIENTADAS A OBJETO II

## OBJECT ORIENTED DATABASES

### INTRODUCTION



Si bien las bases de datos relacionales son las más populares y las que tienen más aceptación, su utilización en una aplicación orientada a objetos implica un proceso de traducción del diagrama UML original a un modelo relacional.

Esta traducción o mapeo no siempre es trivial y puede tener varias soluciones que serán más o menos idóneas dependiendo del diseño de la aplicación y cómo maneje éste los datos.

Eso se da porque en la base de dato relacional tenemos tablas y en la aplicación orientada a objetos tenemos, valga la redundancia, objetos. Durante la traducción o mapeo se pierden muchas de las funcionalidades básicas de la orientación a objetos que se deben simular de alguna manera, referencias a objetos, clases asociativas, listas de objetos, herencia, etc.

Cuando el diagrama es de cierta complejidad, el mapeo puede complicarse muchísimo. Nos podemos ahorrar estos problemas de mapeo utilizando bases de datos orientadas a objetos, en lugar de bases de datos relacionales. La base de datos orientada a objetos se organiza exactamente igual que un diagrama UML en cuanto a las clases y a las relaciones entre ellas. Por esta razón ya no es necesario realizar ningún mapeo.

## ADVANTAGES

Hemos visto que en la base de datos almacenamos directamente los objetos, sin necesidad de realizar ningún mapeo. Otras ventajas de las bases de datos orientadas a objetos son su mayor capacidad de modelado. La utilización de objetos permite representar de una forma más natural los datos que se necesitan guardar.

Extensibilidad. Se pueden construir nuevos tipos de datos a partir de los existentes. Y un lenguaje de consulta más expresivo. El lenguaje de consultas es navegacional, de un objeto al siguiente, en contraste con el declarativo SQL.

## DISADVANTAGES

Pero las bases de datos orientadas a objetos también tienen algunas desventajas que las hacen poco populares. No existe ningún lenguaje estándar común para realizar consultas, como es el SQL para las bases de datos relacionales. En las orientadas a objetos, cada fabricante implementa su propio lenguaje OQL.

Tampoco hay un modelo de datos común, como son las tablas y las relaciones en las bases de datos relacionales, por lo que las diferentes soluciones existentes en el mercado son incompatibles entre ellas. Tienen una mayor complejidad. El incremento de funcionalidad provisto por un sistema gestor de base de datos orientadas a objetos lo hace más complejo que un sistema gestor de base de datos relacional. La complejidad conlleva productos más caros y difíciles de usar. Su uso está poco extendido en comparación con los sistemas relacionales tradicionales, por lo que el soporte y la información disponible es escasa. Debido a estos motivos, actualmente la aplicación de bases de datos orientadas a objetos se limita a ámbitos muy concretos vinculados a las áreas científicas. Su implementación en aplicaciones comerciales de ámbito general es muy baja.

## OQL & ODL

Así como el SQL es el lenguaje que nos proporciona acceso a una base de datos relacional, el OQL o Object Query Language es el lenguaje para comunicarnos con una base de datos orientada a objetos. Adicionalmente, hay un lenguaje de descripción de objetos, ODL, Object Description Language, que sirve para especificar el formato de una base de datos orientada a objetos, es decir, las características de los objetos que puede almacenar y sus relaciones. Ambos lenguajes están definidos por el ODMG, Object Data Base Management Group, un grupo formado por fabricantes de bases de datos con el objetivo de definir los estándares para los sistemas gestores de bases de datos orientadas a objetos.

### Object Description Language

El lenguaje, ODL, se utiliza para definir clases de objetos persistentes en una base de datos orientada a objetos, de forma que sus objetos se pueden almacenar. Dentro de la definición de cada clase se incluye el nombre de la clase, declaraciones opcionales de las claves primarias, declaraciones de elementos, atributos, relaciones y métodos.

### Object Query Language

Las características del Object Query Language son que su sintaxis básica es similar a SQL. No incluye operaciones de actualización, solo de consulta. Las modificaciones se realizan mediante los métodos que los objetos poseen. Dispone de operadores sobre colecciones, max, min, count, etc. Así como cuantificadores, for all, exists...

## INTRODUCTION TO NEODATIS

### INTRODUCTION

Para trabajar el concepto de base de datos orientadas a objetos usaremos Neodatis, una base de datos sencilla de utilizar que aporta una API disponible para Java, pero también para otros lenguajes de programación. Todo el contenido de la base de datos, modelo, objetos, índices, queda almacenado en un único fichero. Y además es de código abierto.

### INSTALLATION ON INTELLIJ

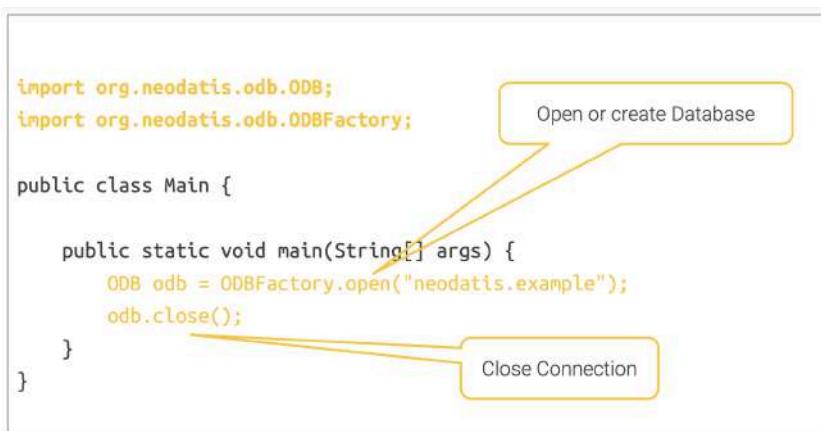
#### Installation Steps

Trabajar con NeoDatis es tan sencillo como descargar el fichero.jar que contiene el driver e importarlo a nuestro IDE, dentro de las librerías del proyecto. Una vez realizados estos dos pasos ya podremos empezar a utilizar la base de datos. Para utilizar NeoDatis lo primero que debemos hacer es conseguir el driver. Lo podemos descargar acudiendo a la web de NeoDatis.

Esta nos redirige a South Forge para realizar la descarga. En el paquete descargado encontramos efectivamente el driver en un fichero.jar. Adicionalmente tenemos un manual, disponible en formato PDF. Aquí podemos encontrar toda la información acerca del uso de Neodatis. Una pequeña descripción, cómo almacenar objetos, cómo recuperarlos, o cómo especificar criterios de búsqueda en las queries. Entre otra mucha información. En el apartado de documentación también encontramos un tutorial ejecutable, este fichero.bat. Dentro de la carpeta src encontramos el código fuente de los ejemplos.

En el paquete descargado encontramos también un explorador de base de datos con interfaz gráfica, odb-explorer.bat si lo ejecutamos a modo de ejemplo podemos ir a buscar la base de datos del tutorial donde podemos ver los objetos que contiene aunque no podemos acceder al contenido de los atributos, simplemente vemos su tipo de datos. Copiamos el driver e incorporemoslo a nuestro proyecto. Debemos añadirlo como librería desde File -> Project Structure y seleccionamos el .jar que acabamos de copiar. Ahora ya podemos empezar a utilizar el api de NeoDatis.

### DBCRAETION



Cuando ya tengamos la API de Neodatis disponible en nuestro proyecto, podremos crear una nueva base de datos con el método estático Open del objeto ODBFactory, al que proporcionaremos la ruta de lo que será nuestra base de datos. Este método nos retornará un objeto ODB, que representa una conexión con la base de datos. Si la base de datos no existe, se creará de nuevo.

Con el método Close del objeto ODB, cerraremos la conexión. Vemos que ya nos aparecen los objetos del API. Aquí abrimos la conexión con la base de datos. Como no existe, se creará de nuevo.

Y aquí simplemente la cerraremos. Ejecutemos. Aquí podemos ver que se nos ha creado el fichero de la base de datos.

## ADDING AND RETRIEVING OBJECTS USING NEODATIS OODB

### NEODATIS DOCUMENTATION

#### Storing Objects

```
try{
 // Create instance
 Sport sport = new Sport("volley-ball");
 // Open the database
 ODB odb = ODBFactory.open(ODB_NAME);
 // Store the object
 odb.store(sport);
} finally{
 if(odb!=null){
 // Close the database
 odb.close();
 }
}
```

En esta diapositiva vemos un ejemplo de cómo insertar un objeto en la base de datos.

#### Retrieving Objects

```
try{
 // Open the database
 odb = ODBFactory.open(ODB_NAME);
 // Get all object of type clazz
 Objects<Player> objects = odb.getObjects(Player.class);
 System.out.println(objects.size() + " player(s)");
 // display each object
 while(objects.hasNext()){
 System.out.println((i+1) + "\t: " + objects.next());
 }
} finally{
 // Closes the database
 if(odb!=null)
 odb.close();
}
```

Y en esta otra, un ejemplo de cómo recuperar los objetos.

Ambos ejemplos están extraídos de la documentación oficial de Neodatis. La documentación oficial de Neodatis está disponible en formato PDF en su web, y también la encontramos en el paquete de utilidades que acompaña el driver que debemos instalar en nuestro IDE.

En esta documentación, encontraremos cómo realizar las operaciones básicas de inserción y consulta. En el capítulo 7, Storing Objects, nos explica cómo guardar objetos. Debemos utilizar el método Store de una instancia de ODD.

Junto con el paquete del driver, también tenemos una sección JavaDoc. Está en formato web, así que podemos consultarla con nuestro navegador para saber cómo utilizar estos métodos. Si nos vamos al método ODB vemos su lista de métodos. Al método Store le pasamos directamente un objeto. Para saber cómo recuperar un objeto iremos al capítulo 8, Retrieving Objects. En el código de ejemplo vemos que debemos utilizar el método getObjects, también del objeto odb. Volviendo a JavaDoc, vemos que el método getObjects tiene tres sobrecargas. La primera recibe simplemente el tipo de clase de los objetos que queremos recuperar. Nos devolverá una lista de objetos, con todos los que haya. En este proyecto de ejemplo, donde ya tengo instalado el driver de Neodatis, he incorporado una de las clases de ejemplo disponibles en el paquete de instalación.

La clase Sport. Es una clase muy sencilla que solo tiene un atributo de tipo String, el correspondiente Getter y Setter, y un método String que devuelve el nombre. Para cerrar convenientemente la conexión colocaremos la llamada null y try. Así podemos cerrar la conexión aquí dentro, pero primero comprobaré que la base de datos no es null para evitar una posible nullPointerException.

Bien, ahora procedo a crear un deporte y luego a guardarlo en la base de datos. Lo haré con el método Store. Ahora que ya tengo el método guardado, procederé a recuperarlo.

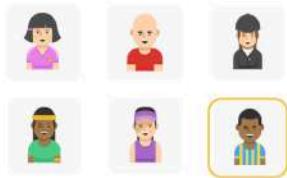
Para ello utilizaré el método GetObjects() que nos retorna una lista de objetos Sport, que es la clase que le especificaré como parámetro. Sobre esta lista que obtendré podremos iterar y mostrar su contenido. Ejecutemos. Hemos recuperado el objeto que hemos insertado aquí arriba.

## QUERIES WITH CRITERIA, UPDATES & DELETES USING NEODATIS OODB

### QUERIES WITH CRITERIA

Sabemos que para recuperar objetos de una base de datos Neodatis, debemos utilizar una instancia de la clase ORG NeodatisODB.odb y llamar al objeto GetObjects, especificando como parámetro el tipo de objeto que queremos recuperar. Esta llamada nos retornará todos los objetos del tipo especificado. Si queremos un resultado más concreto, deberemos especificar criterios de búsqueda. Lo haremos con un objeto de la clase eQuery, que combinado con las clases ICriterion, Where y CriteriaQuery nos permitirá especificar criterios de búsqueda.

```
ICriterion criterio = Where.equal("name", "Oliver");
IQuery query = new CriteriaQuery(Player.class,criterio);
Objects <Player>players = odb.getObjects(query);
```



Veamos cómo. Podemos especificar los criterios de búsqueda con la interfaz ICriterion mediante la cláusula Where y utilizando en este caso el método equal con el nombre del atributo y el valor que queremos para filtrar.

La cláusula Where dispone de muchos más métodos de filtro, contain, returnOrEqual, returnDone, like, or, and, not, entre otros.

Podéis encontrar el listado completo de métodos en la documentación JavaDoc de Neodatis.

Creamos luego un objeto iquery, llamando al constructor de la clase CriteriaQuery, y le pasamos el tipo de objeto que queremos recuperar, seguido del criterio. Una vez hayamos instanciado el objeto iquery, lo pasamos al método getObjects y obtendremos nuestro resultado filtrado.

## UPDATES

```
IQuery query = new CriteriaQuery(Player.class,Where.equal("name", "Oliver"));
Objects <Player>players = odb.getObjects(query);
Player player = (Player) players.getFirst();
player.setName("Eric");
odb.store(player);
```



Para actualizar un objeto de los que tengamos en la base de datos, lo recuperaremos mediante una query con criterios. Actualizaremos su valor y volveremos a guardarlo.

## DELETES

```
IQuery query = new CriteriaQuery(Player.class,Where.equal("name", "Eric"));
Objects <Player>players = odb.getObjects(query);
Player player = (Player) players.getFirst();
odb.delete(player);
```



Para eliminar un objeto de los que tengamos en la base de datos, lo recuperaremos mediante una query con criterios y utilizaremos el método delete en nuestra instancia odb de la base de datos.

## DEMO

En esta demo veremos cómo añadir, actualizar y eliminar objetos de una base de datos neodatis.

En esta clase de ejemplo vemos en primer lugar cómo creamos varios objetos relacionados entre ellos y los guardamos en la base de datos. Tenemos seis jugadores que practican tres deportes distintos y que juegan en cuatro equipos diferentes.

Una vez creados los equipos, los almacenamos en la base de datos y luego llamamos al método displayPlayers y lo que hace es mostrarlos todos. Ejecutemos el código hasta este punto. Se nos ha mostrado. Y si ahora nos vamos a lo de B-Explorer y abrimos la base de datos, vemos los deportes, los equipos y los jugadores. Y si exploramos los objetos vemos que junto con sus atributos, todos tienen un identificador único.

Cerramos la base de datos y regresemos a IntelliJ. El siguiente código lanza una query con criterios. Queremos todos los jugadores que se llamen Oliver. Sabemos que sólo hay uno. Lo recuperamos de la lista de resultados y le cambiamos el nombre por Eric.

Y en la siguiente línea lo guardamos actualizado en la base de datos. Pongamos aquí otro punto de interrupción y sigamos ejecutando. Vemos el nombre actualizado. El jugador se llama Eric. Recordad, el identificador único era 15. El último trozo de código recupera de nuevo el mismo jugador que ahora se llama Eric.

En esta ocasión, lo eliminamos de la base de datos utilizando el método DELETE. El jugador ya no aparece en la lista. Vemos que ahora sólo tenemos 5 jugadores. El identificador 15 ya no existe.