

CURSO DE PROGRAMACIÓN FULL STACK

INSTRUCTIVO INSTALACIÓN UML

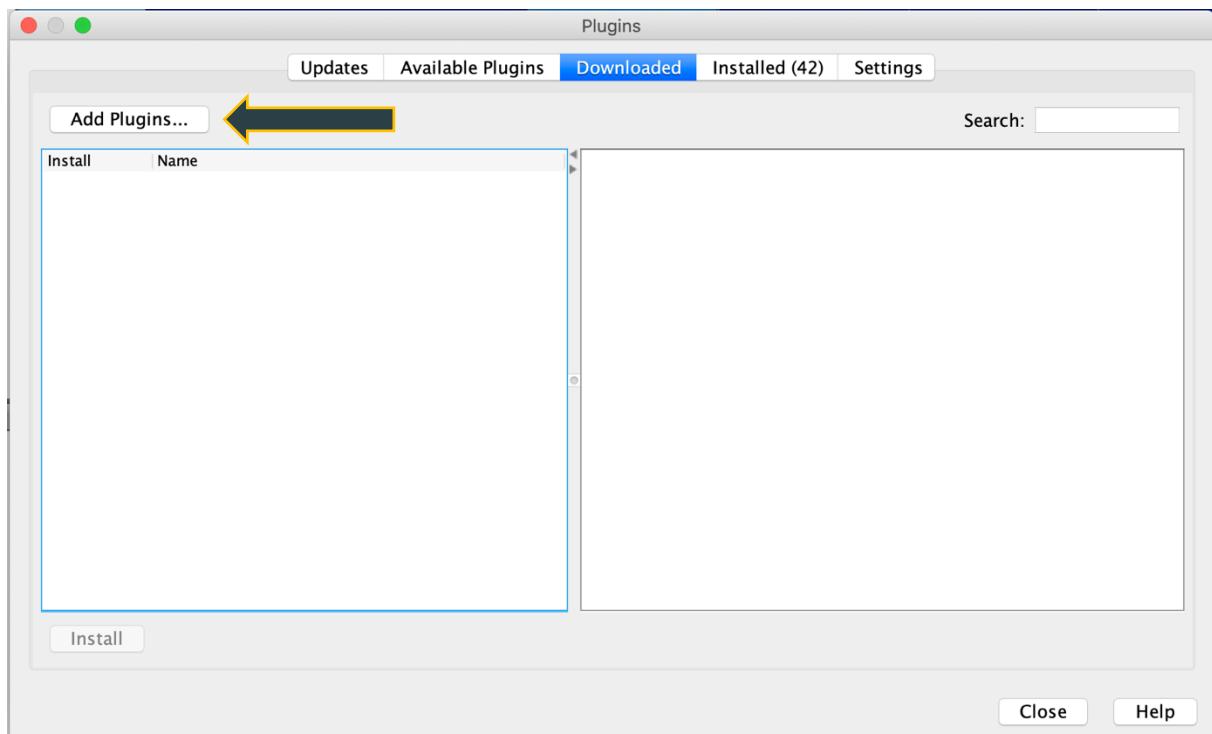


EGG

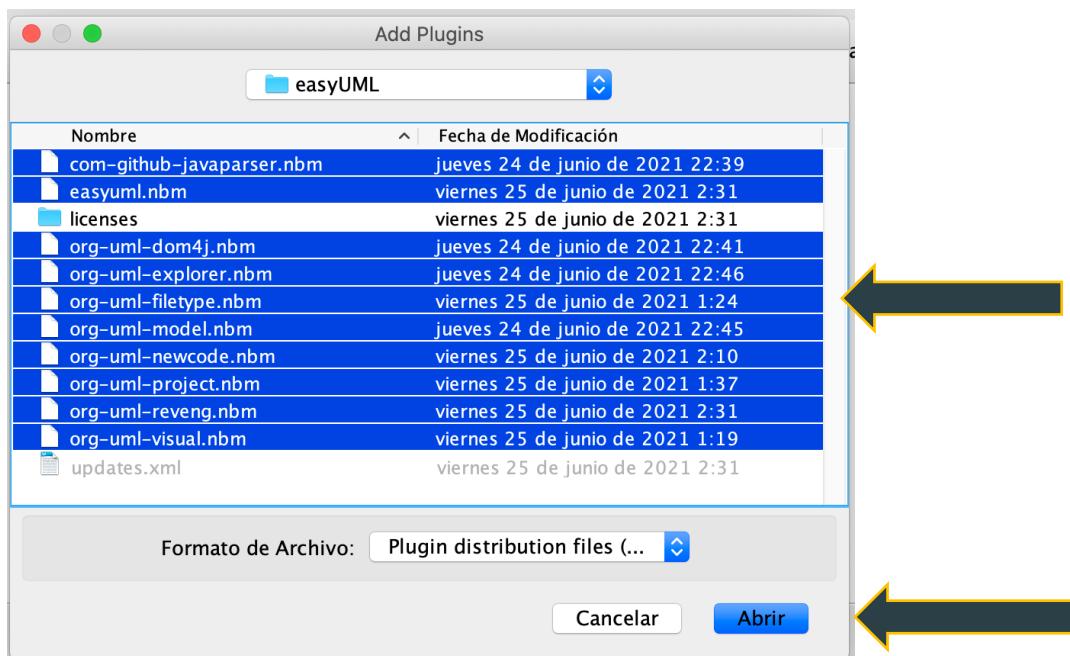
INSTALACIÓN DIAGRAMA UML

Para instalar el plugin

- Descomprimir la carpeta de easyUML en la ubicación que desee.
- En NetBeans ir a: Tools → Plugins → Downloaded → Add Plugins

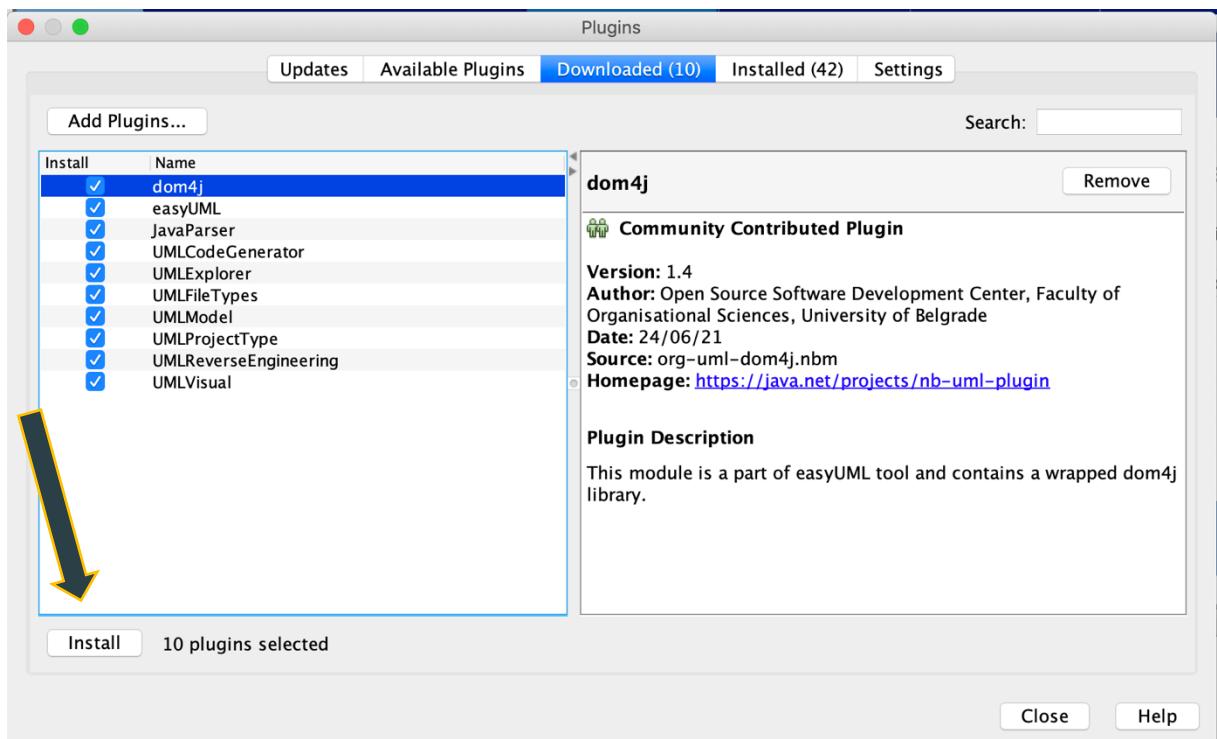


- Seleccionar todos los archivos de la carpeta descargada → Abrir

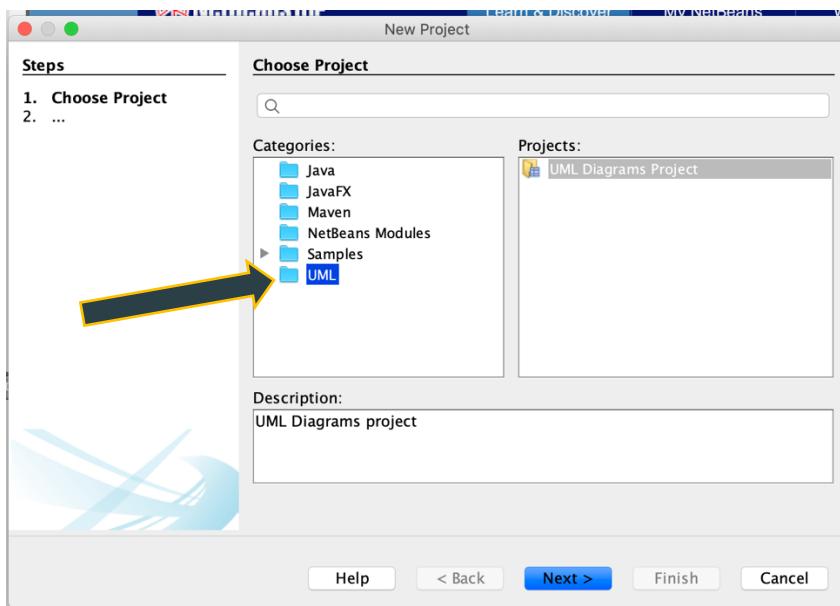


Nota: si nos salen los dos archivos licenses y updates.xml, esos no hay que sumarlos. No va a dejarnos sumarlos igualmente pero aclaramos.

- Después darle a la opción → Install. Ahí vamos a tener que seguir una pequeña instalación, muy sencilla.



- Por ultimo, reiniciar NetBeans
- Dentro de NetBeans, darle a la opción de New Project. Ahí, deberemos crear un proyecto, del tipo UML. Elegir ruta y nombre con que guardar la carpeta. Todos nuestros diagramas los podemos guardar en la misma carpeta.



Para crear un Diagrama desde Cero

- Posicionarse sobre el paquete del proyecto ya creado de diagramas.
- BOTON DERECHO → New Class Diagram

- A la derecha tenemos una ventana “*Palette*”, que es donde tenemos los componentes y relaciones posibles. Es cuestión de deslizar e ir armando el bosquejo.
- **Recordar:** Parte superior de cada componente se determinan los atributos, indicando el tipo de dato. Parte inferior se detallan los métodos que precisaremos crear en esa clase específica.

Importante: Es super importante que al crear un diagrama de clases desde cero o lo creemos en base a un proyecto, que **no haya caracteres especiales** (comas, ñ, símbolos,etc.) de ningún tipo. Ya sea en variables, métodos, nombres de paquetes, nombre del proyecto, nombre de clases, etc. **Los caracteres especiales van a hacer que no podamos crear nuestro diagrama de clases.**

Para crear un proyecto en base al Diagrama creado

- Posicionarse sobre el diagrama creado. BOTON DERECHO → easyUML generate Code
- Creara de forma automática, un proyecto, con las clases, atributos, relaciones y métodos definidos en nuestro diagrama.

Para crear un Diagrama de un proyecto ya existente

- Posicionarse sobre el proyecto que quiero armar su diagrama, BOTON DERECHO, easyUML Create. Me sugerirá guardarlo en la carpeta ya creada de diagramas.
- Editar la imagen para tener visibles las relaciones. Puedo desplazar los cuadros a mi gusto y orden.

Puedo exportar la imagen de mi diagrama, posicionándome sobre el diagrama ya creado, BOTON DERECHO → Export as Image.

Recordatorios de simbologías en el diagrama UML



CURSO DE PROGRAMACIÓN FULL STACK

INTRODUCCIÓN A JAVA

FUNDAMENTOS DEL LENGUAJE



INTRODUCCIÓN A JAVA

Hasta el momento hemos aprendido los diferentes tipos de estructuras de control comunes a todos los lenguajes de programación, dentro del paradigma de programación imperativa, haciendo uso del pseudo intérprete PSeInt. A partir de esta guía comenzaremos a introducir cada uno de los conceptos vistos hasta el momento, pero haciendo uso de un lenguaje de programación de propósito general como lo es Java.

JAVA

Java es un tipo de lenguaje de programación y una plataforma informática, creada y comercializada por Sun Microsystems en el año 1995 y desde entonces se ha vuelto muy popular, gracias a su fácil portabilidad a todos los sistemas operativos existentes.

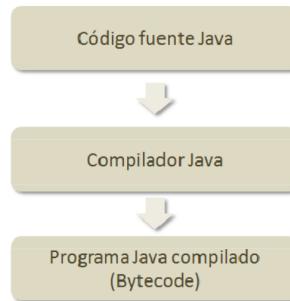
Java es un lenguaje de programación de alto nivel, estos, permiten escribir código mediante idiomas que conocemos (ingles, español, etc.) y luego, para ser ejecutados, se traduce al lenguaje de máquina mediante traductores o compiladores. Java es un lenguaje de alto nivel donde sus palabras reservadas están en inglés.

COMPILADOR EN JAVA

Permite traducir todo un programa de una sola vez, haciendo una ejecución más rápida y puede almacenarse para usarse luego sin volver a hacer la traducción. Los programas de Java se compilan a un lenguaje intermedio, denominado ByteCode. Este código es interpretado por la maquina virtual de Java (JVM) del entorno de ejecución (JRE) y así se consigue la portabilidad en distintas plataformas.

EJECUCIÓN DE UNA APLICACIÓN

En Java, todo el código fuente se escribe en archivos de texto plano cuya extensión es **.java**, en el ejemplo: MyProgram.java. Posteriormente, al compilar el código fuente, el compilador (javac) realiza un análisis de sintaxis del código escrito en los archivos fuente de Java (con extensión *.java). Si no encuentra errores en el código genera los archivos compilados con extensión **.class**. En caso de encontrar errores muestra la línea o líneas erróneas en el código. Los archivos **.class** no contienen código nativo del procesador, sino que contienen un conjunto de instrucciones que se denominan bytecodes, el lenguaje máquina de la **JVM**. Las instrucciones en bytecodes son independientes del tipo de computadora. Luego, el lanzador de aplicaciones java corre su aplicación con una instancia de la **JVM** en una computadora específica (Windows, Unix, MacOS, etc).



ARCHIVO FUENTE EN JAVA

Un archivo fuente de la tecnología Java tiene la siguiente forma:

```
[<declaración_paquete>]
<declaración_importacion>*
<modificador>* <declaración_clase>{
    <atributos>*
    <constructores>*
    <metodos>*
}
```

El orden de estos puntos es importante. Cualquier sentencia de importación debe preceder todas las declaraciones de clases. Si usa una declaración de paquetes, el mismo debe preceder tanto a las clases como a las importaciones. El nombre del archivo fuente tiene que ser el mismo que el nombre de la declaración de la clase pública en ese archivo. Un archivo fuente puede incluir más de una declaración de clase, pero sólo una puede ser declarada pública. Si un archivo fuente no contiene clases públicas declaradas, el nombre del archivo fuente no está restringido. Sin embargo, es una buena práctica tener un archivo fuente para cada clase declarada y que el nombre del archivo sea idéntico al nombre de la clase.

ESTRUCTURA DE UN PROGRAMA JAVA

Un programa describe cómo un ordenador debe interpretar las órdenes del programador para que ejecute y realice las instrucciones dadas tal como están escritas. Un programador utiliza los elementos que ofrece un lenguaje de programación para diseñar programas que resuelvan problemas concretos o realicen acciones bien definidas.

El siguiente programa Java muestra un mensaje en la consola con el texto "Hola Mundo".

```
/*
 * Este programa escribe el texto "Hola Mundo" en la consola * utilizando el
 * método System.out.println()
*/

```

```
package primerprograma;  
public class HolaMundo {  
    public static void main (String[] args) {  
        System.out.println("Hola Mundo");  
    }  
}
```

En este programa se pueden identificar los siguientes elementos del lenguaje Java: comentarios, paquete, definiciones de clase, definiciones de método y sentencias.

COMENTARIO

El programa comienza con un comentario. El delimitador de inicio de un comentario es `/*` y el delimitador de fin de comentario es `*/`.

El texto del primer comentario de este ejemplo seria: 'Este programa escribe el texto "Hola Mundo" en la consola utilizando el método System.out.println()'. Los comentarios son ignorados por el compilador y solo son útiles para el programador. Los comentarios ayudan a explicar aspectos relevantes de un programa y lo hacen más legible. En un comentario se puede escribir todo lo que se desee, el texto puede ser de una o más líneas.

PAQUETES

Después del comentario viene está escrito el nombre del paquete. Los paquetes son contenedores de clases y su función es la de organizar la distribución de las clases. Los paquetes y las clases son análogos a las carpetas y archivos utilizadas por el sistema operativo, respectivamente.

El lenguaje de programación de la tecnología Java le provee la sentencia package como la forma de agrupar clases relacionadas. La sentencia package tiene la siguiente forma:

```
package <nombre_paq_sup>[.<nombre_sub_paq>]*;
```

La declaración package, en caso de existir, debe estar al principio del archivo fuente y sólo la declaración de un paquete está permitida. Los nombres de los paquetes los pondrá el programador al crear el programa y son jerárquicos (al igual que una organización de directorios en disco) además, están separados por puntos. Es usual que sean escritos completamente en minúscula.

CLASES

La primera línea del programa, después del package. Define una clase que se llama HolaMundo. En el mundo de orientación a objetos, todos los programas se definen en término de objetos y sus relaciones. Las clases sirven para modelar los objetos que serán utilizados por nuestros programas. Los objetos, las clases y los paquetes **son conceptos que serán abordados con profundidad más adelante en el curso.**

Una clase está formada por una parte correspondiente a la declaración de la clase, y otra correspondiente al cuerpo de la misma:

```
Declaración de clase {  
Cuerpo de clase  
}
```

En la plantilla de ejemplo se ha simplificado el aspecto de la Declaración de clase, pero sí que puede asegurarse que la misma contendrá, como mínimo, la palabra reservada *class* y el nombre que recibe la clase. La definición de la clase o cuerpo de las comienza con una llave abierta (*{*) y termina con una llave cerrada (*}*). El nombre de la clase lo define el programador

MÉTODOS

Después de la definición de clase se escribe la definición del método *main()*. Pero que es un método?. Dentro del cuerpo de la clase se declaran los atributos y los métodos de la clase. Un método es una secuencia de sentencias ejecutables. Las sentencias de un método quedan delimitadas por los caracteres *{* y *}* que indican el inicio y el fin del método, respectivamente. Si bien es un tema sobre el que se profundizará más adelante en el curso, los métodos son de vital importancia para los objetos y las clases. En un principio, para dar los primeros pasos en Java nos alcanza con esta definición.

MÉTODO MAIN()

Ahora sabemos lo que es un método, pero en el ejemplo podemos ver el método *main()*. El *main()* sirve para que un programa se pueda ejecutar, este método, vendría a representar el Algoritmo / FinAlgoritmo de pseint y tiene la siguiente declaración:

```
public static void main(String[] args){
```

A continuación, describirnos cada uno de los modificadores y componentes que se utilizan *siempre* en la declaración del método *main()*:

public: es un tipo de acceso que indica que el método *main()* es público y, por tanto, puede ser llamado desde otras clases. Todo método *main()* debe ser público para poder ejecutarse desde el intérprete Java (JVM).

static: es un modificador el cual indica que la clase no necesita ser instanciada para poder utilizar el método. También indica que el método es el mismo para todas las instancias que pudieran crearse.

void: indica que la función o método *main()* no devuelve ningún valor.

El método *main()* debe aceptar siempre, como parámetro, un vector de strings, que contendrá los posibles argumentos que se le pasen al programa en la línea de comandos, aunque como es nuestro caso, no se utilice.

Luego, al indicarle a la máquina virtual que ejecute una aplicación el primer método que ejecutará es el método *main()*. Si indicamos a la máquina virtual que corra una clase que no contiene este método, se lanzará un mensaje advirtiendo que la clase que se quiere ejecutar no contiene un método *main()*, es decir que dicha clase no es ejecutable.

Si no se han comprendido hasta el momento muy bien todos estos conceptos, los mismos se irán comprendiendo a lo largo del curso.

SENTENCIA

Son las unidades ejecutable más pequeña de un programa, en otras palabras una línea de código escrita es una sentencia. Especifican y controlan el flujo y orden de ejecución del programa. Una sentencia consta de palabras clave o reservadas como expresiones, declaraciones de variables, o llamadas a funciones.

En nuestro ejemplo, del método *main()* se incluye una sentencia para mostrar un texto por la consola. Los textos siempre se escriben entre comillas dobles para diferenciarlos de otros elementos del lenguaje. **Todas las sentencias de un programa Java deben terminar con el símbolo punto y coma.** Este símbolo indica al compilador que ha finalizado una sentencia.

Una vez que el programa se ha editado, es necesario compilarlo y ejecutarlo para comprobar si es correcto. Al finalizar el proceso de compilación, el compilador indica si hay errores en el código Java, donde se encuentran y el tipo de error que ha detectado: léxico, sintáctico o semántico.

ELEMENTOS DE UN PROGRAMA

Los conceptos vistos previamente, son la estructura de un programa, pero también existen los elementos de un programa. Estos son, básicamente, los componentes que van a conformar las sentencias que podamos escribir en nuestro programa. Recordemos que toda sentencia en nuestro programa debe terminar con el símbolo **punto y coma**. Nos van a ayudar para crear nuestro programa y resolver sus problemas. Estos elementos siempre estarán dentro de un programa/goritmo.

Los elementos de un programa son: **identificadores, variables, constantes, operadores, palabras reservadas.**

PALABRAS RESERVADAS

Palabras que dentro del lenguaje significan la ejecución de una instrucción determinada, por lo que no pueden ser utilizadas con otro fin. En Java, al ser un lenguaje que está creado en inglés, todas nuestras palabras reservadas van a estar en ese idioma.

IDENTIFICADOR

Los identificadores son los nombres que se usan para identificar cada uno de los elementos del lenguaje, como ser, los nombres de las variables, nombres de las clases, interfaces, atributos y métodos de un programa. Si bien Java permite nombres de identificadores tan largos que se desee, es aconsejable crearlos de forma que tengan sentido y faciliten su interpretación. El nombre ideal para un identificador es aquel que no se excede en longitud (lo más corto posible) y que califique claramente el concepto que intenta representar en el contexto del problema que se está resolviendo.

VARIABLES Y CONSTANTES

Recordemos que en Pseint dijimos que los programas de computadora necesitan **información** para la resolución de problemas. Esta información puede ser un numero, un nombre, etc. Para utilizar la información, vamos a guardarla en algo llamado, **variables y constantes**. Las variables y constantes vendrían a ser como pequeñas cajas, que guardan algo en su interior, en este caso información. Estas, van a contar como previamente habíamos mencionado, con un identificador, un nombre que facilitara distinguir unas de otras y nos ayudara a saber que variable o constante es la contiene la información que necesitamos.

Dentro de toda la información que vamos a manejar, a veces, necesitaremos información que no cambie. Tales valores son las **constantes**. De igual forma, existen otros valores que necesitaremos que cambien durante la ejecución del programa; esas van a ser nuestras **variables**.

DECLARACIÓN DE VARIABLES EN JAVA

Normalmente los identificadores de las variables y de las constantes con nombre deben ser declaradas en los programas antes de ser utilizadas. La sintaxis de la declaración de una variable en java suele ser:

```
<tipo_de_dato> <nombre_variable>;
```

TIPOS DE DATO EN JAVA

Java es un lenguaje de *tipado estático*, esto significa que todas las variables *deben ser declaradas* antes que ellas puedan ser utilizadas y que no podemos cambiar el tipo de dato de una variable, a menos que, sea a traves de una conversión.

TIPOS DE DATOS PRIMITIVOS

Primitivos: Son predefinidos por el lenguaje. La biblioteca Java proporciona clases asociadas a estos tipos que proporcionan métodos que facilitan su manejo.

byte	Es un entero con signo de 8 bits, el mínimo valor que se puede almacenar es -128 y el máximo valor es de 127 (inclusive).
short	Es un entero con signo de 16 bits. El valor mínimo es -32,768 y el valor máximo 32,767 (inclusive).
int	Es un entero con signo de 32 bits. El valor mínimo es -2,147,483,648 y el valor máximo es 2,147,483,647 (inclusive). Generalmente es la opción por defecto.
long	Es un entero con signo de 64 bits, el valor mínimo que puede almacenar este tipo de dato es -9,223,372,036,854,775,808 y el máximo valor es 9,223,372,036,854,775,807 (inclusive).
float	Es un número decimal de precisión simple de 32 bits (IEEE 754 Punto Flotante).
double	Es un número decimal de precisión doble de 64 bits (IEEE 754 Punto Flotante).
boolean	Este tipo de dato sólo soporta dos posibles valores: verdadero o falso y el dato es representado con tan solo un bit de información.
char	El tipo de dato carácter es un simple carácter unicode de 16 bits. Su valor mínimo es de '\u0000' (En entero es 0) y su valor máximo es de '\uffff' (En entero es 65,535). Nota: un dato de tipo carácter se puede escribir entre comillas simples, por ejemplo 'a', o también indicando su valor Unicode, por ejemplo '\u0061'.
String	<p>Además de los tipos de datos primitivos el lenguaje de programación Java provee también un soporte especial para cadena de caracteres a través de la clase String.</p> <p>Encerrando la cadena de caracteres con comillas dobles se creará de manera automática una nueva instancia de un objeto tipo String.</p> <pre>String cadena = "Sebastián";</pre> <p>Los objetos String son inmutables, esto significa que una vez creados, sus valores no pueden ser cambiados. Si bien esta clase no es técnicamente un tipo de dato primitivo, el lenguaje le da un soporte especial y hace parecer como si lo fuera.</p>

VALORES POR DEFECTO

En Java no siempre es necesario asignar valores cuando nuevos atributos son declarados. Cuando los atributos son declarados, pero no inicializados, el compilador les asignará un valor por defecto. A grandes rasgos el valor por defecto será cero o null dependiendo del tipo de dato. La siguiente tabla resume los valores por defecto dependiendo del tipo de dato.

byte	0
short	0
int	0
long	0
float	0.0
double	0.0
boolean	false
char	'\u0000'
String	null
Objetos	null

Las variables locales son ligeramente diferentes; el compilador no asigna un valor predeterminado a una variable local no inicializada. Las variables locales son aquellas que se declaran dentro de un método. Si una variable local no se inicializa al momento de declararla, se debe asignar un valor antes de intentar usarla. El acceso a una variable local no inicializada dará lugar a un error en tiempo de compilación.

OPERADORES

Los operadores son símbolos especiales de la plataforma que permiten especificar operaciones en uno, dos o tres operandos y retornar un resultado. También aprenderemos qué operadores poseen mayor orden de precedencia. Los operadores con mayor orden de precedencia se evalúan siempre primero.

Primeramente, proceden los operadores unarios, luego los aritméticos, después los de bits, posteriormente los relacionales, detrás vienen los booleanos y por último el operador de asignación. La regla de precedencia establece que los operadores de mayor nivel se ejecuten primero. Cuando dos operadores poseen el mismo nivel de prioridad los mismos se evalúan de izquierda a derecha.

OPERADOR DE ASIGNACIÓN

- = Operador de Asignación Simple

OPERADORES ARITMÉTICOS

- + Operador de Suma
- Operador de Resta
- * Operador de Multiplicación
- / Operador de División
- % Operador de Módulo

OPERADORES UNARIOS

- + Operador Unario de Suma; indica que el valor es positivo.
- Operador Unario de Resta; indica que el valor es negativo.
- ++ Operador de Incremento.
- Operador de Decremento.

OPERADORES DE IGUALDAD Y RELACIÓN

- == Igual
- != Distinto

>	Mayor que
\geq	Mayor o igual que
<	Menor que
\leq	Menor o igual que
OPERADORES CONDICIONALES	
$\&\&$	AND
$\ $	OR
!	Operador Lógico de Negación.

TIPOS DE INSTRUCCIONES

Además de los elementos de un programa/algoritmo, tenemos las instrucciones que pueden componer un programa. Las instrucciones —acciones— básicas que se pueden implementar de modo general en un algoritmo y que esencialmente soportan todos los lenguajes son las siguientes:

- ✓ **Instrucciones de inicio/fin**, son utilizadas para delimitar bloques de código.
- ✓ **Instrucciones de asignación**, se utilizan para asignar el resultado de la evaluación de una expresión a una variable. El valor (dato) que se obtiene al evaluar la expresión es almacenado en la variable que se indique.

`<nombre de la variable> ← <expresión>`

expresión es igual a una expresión matemática o lógica, a una variable o constante.

- ✓ **Instrucciones de lectura**, se utilizan para leer datos de un dispositivo de entrada y se asignan a las variables.
- ✓ **Instrucciones de escritura**, se utilizan para escribir o mostrar mensajes o contenidos de las variables en un dispositivo de salida.
- ✓ **Instrucciones de bifurcación**, mediante estas instrucciones el desarrollo lineal de un programa se interrumpe. Las bifurcaciones o al flujo de un programa puede ser según el punto del programa en el que se ejecuta la instrucción hacia adelante o hacia atrás.

INSTRUCCIONES PRIMITIVAS

Dentro de las instrucciones previamente vistas, existe un subdivisión que son las instrucciones primitivas, las instrucciones primitivas van a ser las instrucciones de asignación, lectura y escritura.

ASIGNACIÓN

La instrucción de asignación permite almacenar un valor en una variable (previamente definida). Esta es nuestra manera de guardar información en una variable, para utilizar ese valor en otro momento.

`<variable> = <expresión>`

En Java, podemos definir una variable y al mismo tiempo podemos asignarle un valor a diferencia de Pseint:

`<tipo_de_dato> <nombre_variable> = expresion;`

Al ejecutarse la asignación, primero se evalúa la expresión de la derecha y luego se asigna el resultado a la variable de la izquierda. El tipo de la variable y el de la expresión deben coincidir.

ENTRADA Y SALIDA DE INFORMACIÓN

Los cálculos que realizan las computadoras requieren, para ser útiles la entrada de los datos necesarios para ejecutar las operaciones que posteriormente se convertirán en resultados, es decir, salida.

Las operaciones de entrada permiten leer determinados valores y asignarlos a determinadas variables y las operaciones de salida permiten escribir o mostrar resultados de determinadas variables y las operaciones, o simplemente mostrar mensajes.

ESCRITURA EN JAVA

En nuestro ejemplo de código al principio de la guía, usábamos la instrucción `System.out.println()` para mostrar el mensaje Hola Mundo. Esta instrucción permite mostrar valores en el **Output**, que es la interfaz grafica de Java. Todo lo que quisiéramos mostrar en nuestra interfaz grafica, deberá ir entre comillas dobles y dentro del paréntesis.

`System.out.println("Hola Mundo");`

Si quisiéramos que concatenar un mensaje y la impresión de una variable deberíamos usar el símbolo más para poder lograrlo.

`System.out.println("La variable tiene el valor de: " + variable);`

Si quisiéramos escribir sin saltos de línea, deberíamos quitarle el `In` a nuestro `System.out.println`.

`System.out.print("Hola");
System.out.print("Mundo");`

LECTURA O ENTRADA EN JAVA

En Java hay muchas maneras de ingresar información en el Output por teclado en nuestro programa Java, en el curso vamos a usar la clase Scanner.

Scanner es una clase en el paquete `java.util` utilizada para obtener la entrada de los tipos primitivos como int, double etc. y también String. Es la forma más fácil de leer datos en un programa Java.

Ejemplo definición clase Scanner:

```
Scanner leer = new Scanner(System.in);
```

- Este objeto Scanner vamos a tener que importarlo para poder usarlo, ya que es una herramienta que nos provee Java. Para importarlo vamos a utilizar la palabra clave `import`, seguido de la declaración de la librería donde se encuentra el Scanner. Esta sentencia, va debajo de la sentencia package. La sentencia se ve así: `import java.util.Scanner;`
- Para crear un objeto de clase Scanner, normalmente pasamos el objeto predefinido System.in, que representa el flujo de entrada estándar.
- Se le puso el nombre leer, pero se le puede el nombre que nosotros quisiéramos.
- Para utilizar las funciones del objeto Scanner, vamos a utilizar el nombre que le hemos asignado y después del nombre ponemos un punto(.), de esa manera podremos llamar a las funciones del Scanner.
- Para leer valores numéricos de un determinado tipo de datos, la función que se utilizará es `nextInt()`. Por ejemplo, para leer un valor de tipo int (entero), podemos usar `nextInt()`, para leer un valor de tipo double(real), usamos `nextDouble()` y etc. Para leer un String (cadenas), usamos `nextLine()`.

Podemos usarlo cuando definimos la variable:

```
int numero = leer.nextInt();
```

Podemos usarlo con una variable pre definida:

```
int numero;  
numero = leer.nextInt();
```

Nota: pueden encontrar un ejemplo lectura y entrada en Java en Moodle.

INSTRUCCIONES DE BIFURCACIÓN

Mediante estas instrucciones el desarrollo lineal de un programa se interrumpe. Las bifurcaciones o al flujo de un programa puede ser según el punto del programa en el que se ejecuta la instrucción hacia adelante o hacia atrás. De esto se encargan las estructuras de control.

ESTRUCTURAS DE CONTROL

Las estructuras de control son construcciones hechas a partir de palabras reservadas del lenguaje que permiten modificar el flujo de ejecución de un programa. De este modo, pueden crearse construcciones de alternativas mediante sentencias condicionales y bucles de repetición de bloques de instrucciones. Hay que señalar que un bloque de instrucciones se encontrará encerrado mediante llaves {} si existe más de una instrucción.

ESTRUCTURAS CONDICIONALES

Los condicionales son estructuras de control que cambian el flujo de ejecución de un programa de acuerdo a si se cumple o no una condición. Cuando el flujo de control del programa llega al condicional, el programa evalúa la condición y determina el camino a seguir. Existen dos tipos de estructuras condicionales, las estructuras *if / else* y la estructura *switch*.

IF/ELSE

La estructura *if* es la más básica de las estructuras de control de flujo. Esta estructura le indica al programa que ejecute cierta parte del código sólo si la condición evaluada es verdadera («true»). La forma más simple de esta estructura es la siguiente:

```
if(<condición>){  
    <sentencias>  
}
```

En donde, *<condición>* es una expresión condicional cuyo resultado luego de la evaluación es un dato booleano(lógico) verdadero o falso. El bloque de instrucciones *<sentencias>* se ejecuta si, y sólo si, la expresión (que debe ser lógica) se evalúa a true, es decir, se cumple la condición.

Luego, en caso de que la condición no se cumpla y se quiera ejecutar otro bloque de código, otra forma de expresar esta estructura es la siguiente:

```
if(<condición>){  
    <sentencias A>  
} else {  
    <sentencias B>  
}
```

El flujo de control del programa funciona de la misma manera, cuando se ejecuta la estructura if, se evalúa la expresión condicional, si el resultado de la condición es verdadero se ejecutan las sentencias que se encuentran contenidas dentro del bloque de código if (<sentencias A>). Contrariamente, se ejecutan las sentencias contenidas dentro del bloque else (<sentencias B>).

En muchas ocasiones, se anidan estructuras alternativas *if-else*, de forma que se pregunte por una condición si anteriormente no se ha cumplido otra y así sucesivamente.

```

if (<condicion1>) {
<sentencias A>
} else if(<condicion2>){
<sentencias B>
} else {
<sentencias C>
}

```

Al contrario de la estructura if / else, la estructura *switch* permite cualquier cantidad de rutas de ejecución posibles. Un switch funciona con los datos primitivos byte, short, char e int. También funciona con Enumeraciones, tema que se verá más adelante en el curso, y con unas cuantas clases especiales que «envuelven» a ciertos tipos primitivos: **Character**, **Byte**, **Short**, e **Integer** (tema que trataremos cuando se profundice en Orientación a Objetos).

SWITCH

El bloque *switch* evalúa qué valor tiene la variable, y de acuerdo al valor que posee ejecuta las sentencias del bloque case correspondiente, es decir, del bloque case que cumpla con el valor de la variable que se está evaluando dentro del switch.

```

switch(<variable>) {
case <valor1>:
<sentencias1>
break;
case <valor2>:
<sentencias2>
break;
default:
<sentencias3>
}

```

El uso de la sentencia break que va detrás de cada case termina la sentencia switch que la envuelve, es decir que el control de flujo del programa continúa con la primera sentencia que se encuentra a continuación del cierre del bloque switch. Si el programa comprueba que se cumple el primer valor (valor1) se ejecutará el bloque de instrucciones <sentencias1>, pero si no se encuentra inmediatamente la sentencia break también se ejecutarían las instrucciones <sentencias2>, y así sucesivamente hasta encontrarse con la palabra reservada break o llegar al final de la estructura.

Las instrucciones dentro del bloque default se ejecutan cuando la variable que se está evaluando no coincide con ninguno de los valores case. Esta sentencia equivale al else de la estructura if-else.

Nota: pueden encontrar un ejemplo de **estructuras condicionales** en Moodle.

ESTRUCTURAS REPETITIVAS

Durante el proceso de creación de programas, es muy común, encontrarse con que una operación o conjunto de operaciones deben repetirse muchas veces. Para ello es importante conocer las estructuras de algoritmos que permiten repetir una o varias acciones, un número determinado de veces.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan *bucles*, y se denomina *iteración* al hecho de repetir la ejecución de una secuencia de acciones.

Todo bucle tiene que llevar asociada una condición, que es la que va a determinar cuándo se repite el bucle y cuando deja de repetirse.

WHILE

La estructura *while* ejecuta un bloque de instrucciones mientras se cumple una condición. La condición se comprueba antes de empezar a ejecutar por primera vez el bucle, por lo tanto, si la condición se evalúa a «false» en la primera iteración, entonces el bloque de instrucciones no se ejecutará ninguna vez.

```
while (<condición>) {  
    <sentencias>  
}
```

DO / WHILE

En este tipo de bucle, el bloque instrucciones se ejecuta siempre al menos una vez. El bloque de instrucciones se ejecutará mientras la condición se evalúe a «true». Por lo tanto, entre las instrucciones que se repiten deberá existir alguna que, en algún momento, haga que la condición se evalúe a «false», de lo contrario el bucle será infinito.

```
do {  
    <sentencias>  
} while (<condición>);
```

La diferencia entre *do-while* y *while* es que do-while evalúa su condición al final del bloque en lugar de hacerlo al inicio. Por lo tanto, el bloque de sentencia después del “do” siempre se ejecutan al menos una vez.

FOR

La estructura *for* proporciona una forma compacta de recorrer un rango de valores cuando la cantidad de veces que se debe iterar un bloque de código es conocida. La forma general de la estructura *for* se puede expresar del siguiente modo:

```
for (<inicialización>; <terminación>; <incremento>) {  
    <sentencias>  
}
```

La expresión <inicialización> inicializa el bucle y se ejecuta una sola vez al iniciar el bucle. El bucle termina cuando al evaluar la expresión <terminación> el resultado que se obtiene es false. La expresión <incremento> se invoca después de cada iteración que realiza el bucle; esta expresión incrementa o decrementa un valor hasta que se cumpla la condición de <terminación> del bucle.

La estructura for también ha sido mejorada para iterar de manera más compacta las colecciones y los arreglos, tema que se verá más adelante en este curso. Esta versión mejorada se conoce como for-each.

Como regla general puede decirse que se utilizará el bucle for cuando se conozca de antemano el número exacto de veces que ha de repetirse un determinado bloque de instrucciones. Se utilizará el bucle do-while cuando no se conoce exactamente el número de veces que se ejecutará el bucle, pero se sabe que por lo menos se ha de ejecutar una. Se utilizará el bucle while cuando es posible que no deba ejecutarse ninguna vez.

Nota: pueden encontrar un ejemplo de [estructuras repetitivas](#) en Moodle.

SENTENCIAS DE SALTO

En Java existen dos formas de realizar un salto incondicional en el flujo “normal” de un programa. A saber, las instrucciones break y continue.

BREAK

La instrucción break sirve para abandonar una estructura de control, tanto de las condicionales (if-else y switch) como de las repetitivas (for, do-while y while). En el momento que se ejecuta la instrucción break, el control del programa sale de la estructura en la que se encuentra contenida y continua con el programa.

CONTINUE

La sentencia *continue* corta la iteración en donde se encuentra el continue, pero en lugar de salir del bucle, continúa con la siguiente iteración. La instrucción continue transfiere el control del programa desde la instrucción continue directamente a la cabecera del bucle (for, do-while o while) donde se encuentra.

Nota: pueden encontrar un ejemplo de [sentencias de salto](#) en Moodle.

SUBPROGRAMAS

Un método muy útil para solucionar un problema complejo es dividirlo en subproblemas — problemas más sencillos — y a continuación dividir estos subproblemas en otros más simples, hasta que los problemas más pequeños sean fáciles de resolver. Esta técnica de dividir el problema principal en subproblemas se suele denominar “divide y vencerás”.

El problema principal se soluciona por el correspondiente programa o algoritmo principal, mientras que la solución de los subproblemas será a través de subprogramas, conocidos como **procedimientos** o **funciones**. Un subprograma es un como un mini algoritmo, que recibe los *datos*, necesarios para realizar una tarea, desde el programa y devuelve los *resultados* de esa tarea.

FUNCIONES

Las funciones o métodos son un conjunto de líneas de código (instrucciones), encapsulados en un bloque, usualmente según los parámetros definidos en la función, esta recibe argumentos, cuyos valores se utilizan para efectuar operaciones y adicionalmente retornan un valor. En otras palabras, una función según sus parámetros, puede recibir argumentos (algunas no reciben nada), hace uso de dichos valores recibidos como sea necesario y retorna un valor usando la instrucción return, si no retorna es otro tipo de función. Los tipos que pueden usarse en la función son: int, doble, long, boolean, String y char.

A estas funciones les vamos a asignar un tipo de acceso y un modificador. Estos dos conceptos los vamos a ver mejor más adelante, pero por ahora siempre vamos a crear las funciones con el acceso public y el modificador static. **Para saber más sobre estos dos temas, leer la explicación del método main.**

```
[acceso] [modificador] [tipo] nombreFuncion([tipo] nombreArgumento, .....){  
    /*  
     * Bloque de instrucciones  
     */  
    return valor;  
}
```

PROCEDIMIENTOS (FUNCIONES SIN RETORNO)

Los procedimientos son básicamente un conjunto de instrucciones que se ejecutan sin retornar ningún valor, hay quienes dicen que un procedimiento no recibe valores o argumentos, sin embargo, en la definición no hay nada que se lo impida. En el contexto de Java un procedimiento es básicamente una función cuyo tipo de retorno es void, los que indica que devuelven ningún resultado.

```
[acceso] [modificador] void nombreFuncion([tipo] nombreArgumento){  
    /*  
     * Bloque de instrucciones  
     */  
}
```

Acerca de los argumentos o parámetros:

Hay algunos detalles respecto a los argumentos de un método, veamos:

- Una función, un método o un procedimiento pueden tener una cantidad infinita de parámetros, es decir pueden tener cero, uno, tres, diez, cien o más parámetros. Aunque habitualmente no suelen tener más de 4 o 5.
- Si una función tiene más de un parámetro cada uno de ellos debe ir separado por una coma.
- Los argumentos de una función también tienen un tipo y un nombre que los identifica. El tipo del argumento puede ser cualquiera y no tiene relación con el tipo del método.
- Al recibir un argumento nada nos obliga a hacer uso de éste al interior del método, sin embargo, para que recibarlo si no lo vamos a usar.
- En java los argumentos que sean variables de tipos primitivos (int, double, char, etc.) se van a pasar por valor, mientras que los objetos (String, Integer, etc.) y los arreglos se van a pasar por referencia. Nota: el concepto de objetos lo vamos a ver más adelante.

Consejos acerca de return:

- Cualquier instrucción que se encuentre después de la ejecución de return NO será ejecutada. Es común encontrar funciones con múltiples sentencias return al interior de condicionales, pero una vez que el código ejecuta una sentencia return lo que haya de allí hacia abajo no se ejecutará.
- El tipo de valor que se retorna en una función debe coincidir con el del tipo declarado a la función, es decir si se declara int, el valor retornado debe ser un número entero.
- En el caso de los procedimientos (void) podemos usar la sentencia return pero sin ningún tipo de valor, sólo la usaríamos como una manera de terminar la ejecución del procedimiento.

ARREGLOS: VECTORES Y MATRICES

Un arreglo es un contenedor de objetos que tiene un número fijo de valores del mismo tipo. El tamaño del arreglo es establecido cuando el arreglo es creado y luego de su creación su tamaño es fijo, esto significa que no puede cambiar. Cada una de los espacios de un arreglo es llamada elemento y cada elemento puede ser accedido por un índice numérico que arranca desde 0 hasta el tamaño menos uno. Por ejemplo, si tenemos un vector de 5 elementos mis índices serian: 0-1-2-3-4

Al igual que la declaración de otros tipos de variables, la declaración de un arreglo tiene dos componentes: el tipo de dato del arreglo y su nombre. El tipo de dato del arreglo se escribe como *tipo[]*, en donde, tipo es el tipo de dato de cada elemento contenido en él. Los corchetes sirven para indicar que esa variable va a ser un arreglo. El tamaño del arreglo no es parte de su tipo (es por esto que los corchetes están vacíos).

Una vez declarado un arreglo hay que crearlo/dimensionarlo, es decir, hay que asignar al arreglo un tamaño para almacenar los valores. La creación de un arreglo se hace con el operador *new*. Recordemos que las matrices son bidimensionales por lo que tienen dos tamaños, uno para las filas y otro para las columnas de la matriz.

Declaración y creación de un Vector

```
tipo[] arregloV = new tipo[Tamaño];
```

Declaración y creación de una Matriz

```
tipo[][] arregloM = new tipo[Filas][Columnas];
```

ASIGNAR ELEMENTOS A UN ARREGLO

Cuando queremos ingresar un elemento en nuestro arreglo vamos a tener que elegir el subíndice en el que lo queremos guardar. Una vez que tenemos el subíndice decidido tenemos que invocar nuestro vector por su nombre y entre corchetes el subíndice en el que lo queremos guardar.

Después, pondremos el signo de igual (que es el operador de asignación) seguido del elemento a guardar. En las matrices vamos a necesitar dos subíndices y dos corchetes para representar la posición de la fila y la columna donde queremos guardar el elemento.

Asignación de un Vector

```
vector[0] = 5;
```

Asignación de una Matriz

```
matriz[0][0] = 6;
```

Esta forma de asignación implica asignar todos los valores de nuestro arreglo de uno en uno, esto va a conllevar un trabajo bastante grande dependiendo del tamaño de nuestro arreglo.

Entonces, para poder asignar varios valores a nuestro arreglo y no hacerlo de uno en uno usamos un bucle **Para**. El bucle Para, al poder asignarle un valor inicial y un valor final a una variable, podemos adaptarlo fácilmente a nuestros arreglos. Ya que, pondríamos el valor inicial de nuestro arreglo y su valor final en las respectivas partes del Para. Nosotros, usaríamos la variable creada en el Para, y la pasaríamos a nuestro arreglo para representar todos los subíndices del arreglo, de esa manera, recorriendo todas las posiciones y asignándole a cada posición un elemento.

Para poder asignar varios elementos a nuestra matriz, usaríamos dos bucles **Para** anidados., ya que un **Para** recorrerá las filas (*variable i*) y otro las columnas (*variable j*).

Asignación de un Vector

```
for (int i = 0; i < 5; i++) {  
    vector[i] = 5;  
}
```

Asignación de una Matriz

```
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        matriz[i][j] = 6;  
    }  
}
```

Nota: pueden encontrar un ejemplo de **vectores y matrices** en Moodle.

VECTORES Y MATRICES EN SUBPROGRAMAS

Los arreglos se pueden pasar como parámetros a un subprograma (función o procedimiento) del mismo modo que pasamos variables, poniendo el tipo de dato delante del nombre del vector o matriz, pero deberemos sumarle las llaves para representar que es un vector o matriz. Sin embargo, hay que tener en cuenta que la diferencia entre los arreglos y las variables, es que los arreglos siempre se pasan por referencia.

```
public static void llenarVector(int[] vector){  
}  
  
public static void mostrarMatriz(int[][] matriz){  
}
```

A diferencia de Pseint, en Java si podemos devolver un vector o una matriz en una función para usarla en otro momento. Lo que hacemos es poner como tipo de dato de la función, el tipo de dato que tendrá el vector y así poder devolverlo.

```
public static int devolverVector(){  
    int[] vector = new int[5];  
    return vector;
```

CLASES DE UTILIDAD

Dentro del API de Java existe una gran colección de clases que son muy utilizadas en el desarrollo de aplicaciones. Las clases de utilidad son clases que definen un conjunto de métodos que realizan funciones, normalmente muy reutilizadas. Estas nos van a ayudar junto con las estructuras de control, a lograr resolver problemas de manera más sencilla.

Entre las clases de utilidad de Java más utilizadas y conocidas están las siguientes: Arrays, String, Integer, Math, Date, Calendar y GregorianCalendar. En esta guía solo vamos a ver la **Math**, **String** para hacer algunos ejercicios y después veremos el resto en mayor profundidad.

CLASE STRING

La clase String está orientada al manejo de cadenas de caracteres y pertenece al paquete java.lang del API de Java. Los objetos que son instancias de la clase String, se pueden crear a partir de cadenas constantes también llamadas literales, las cuales deben estar contenidas entre comillas dobles. Una instancia de la clase String es inmutable, es decir, una vez que se ha creado y se le ha asignado un valor, éste no puede modificarse (añadiendo, eliminando o cambiando caracteres).

Al ser un objeto, una instancia de la clase String no sigue las normas de manipulación de los datos de tipo primitivo con excepción del operador concatenación. El operador + realiza una concatenación cuando, al menos, un operando es un String. El otro operando puede ser de un tipo primitivo. El resultado es una nueva instancia de tipo String.

Método	Descripción.
charAt(int index)	Retorna el carácter especificado en la posición index
equals(String str)	Sirve para comparar si dos cadenas son iguales. Devuelve true si son iguales y false si no.
equalsIgnoreCase(String str)	Sirve para comparar si dos cadenas son iguales, ignorando la grafía de la palabra. Devuelve true si son iguales y false si no.
compareTo(String otraCadena)	Compara dos cadenas de caracteres alfabéticamente. Retorna 0 si son iguales, entero negativo si la primera es menor o entero positivo si la primera es mayor.
concat(String str)	Concatena la cadena del parámetro al final de la primera cadena.
contains(CharSequence s)	Retorna true si la cadena contiene la secuencia tipo char del parámetro.
endsWith(String suffix)	Retorna verdadero si la cadena es igual al objeto del parámetro
indexOf(String str)	Retorna el índice de la primera ocurrencia de la cadena del parámetro
isEmpty()	Retorna verdadero si la longitud de la cadena es 0

<code>length()</code>	Retorna la longitud de la cadena
<code>replace(char oldChar, char newChar)</code>	Retorna una nueva cadena reemplazando los caracteres del primer parámetro con el carácter del segundo parámetro
<code>split(String regex)</code>	Retorna un arreglo de cadenas separadas por la cadena del parámetro
<code>startsWith(String prefix)</code>	Retorna verdadero si el comienzo de la cadena es igual al prefijo del parámetro
<code>substring(int beginIndex)</code>	Retorna la sub cadena desde el carácter del parámetro
<code>substring(int beginIndex, int endIndex)</code>	Retorna la sub cadena desde el carácter del primer parámetro hasta el carácter del segundo parámetro
<code>toCharArray()</code>	Retorna el conjunto de caracteres de la cadena
<code>toLowerCase()</code>	Retorna la cadena en minúsculas
<code>toUpperCase()</code>	Retorna la cadena en mayúsculas

Java al ser un lenguaje de tipado estático, requiere que para pasar una variable de un tipo de dato a otro necesitemos usar un conversor. Por lo que, para convertir cualquier tipo de dato a un String, utilicemos la función `valueOf(n)`.

Ejemplo:

```
int numEntero = 4;
String numCadena = String.valueOf(numEntero);
```

Si quisiéramos hacerlo al revés, de String a int se usa el método de la clase Integer, `parseInt()`.

Ejemplo:

```
String numCadena = "1";
int numEntero = Integer.parseInt(numCadena);
```

CLASE MATH

En ocasiones nos vemos en la necesidad de incluir cálculos, operaciones, matemáticas, estadísticas, etc en nuestro programas Java.

Es cierto que muchos cálculos se pueden hacer simplemente utilizando los operadores aritméticos que java pone a nuestra disposición, pero existe una opción mucho más sencilla de utilizar, sobre todo para *cálculos complicados*. Esta opción es la **clase Math** del paquete **java.lang**.

La clase Math nos ofrece numerosos y valiosos métodos y constantes estáticos, que podemos utilizar tan sólo anteponiendo el nombre de la clase.

Método	Descripción.
abs(double a)	Devuelve el valor absoluto de un valor double introducido como parámetro.
abs(int a)	Devuelve el valor absoluto de un valor Entero introducido como parámetro.
abs(long a)	Devuelve el valor absoluto de un valor long introducido como parámetro.
max(double a, double b)	Devuelve el mayor de dos valores double
max(int a, int b)	Devuelve el mayor de dos valores Enteros.
max(long a, long b)	Devuelve el mayor de dos valores long.
min(double a, double b)	Devuelve el menor de dos valores double.
min(int a, int b)	Devuelve el menor de dos valores enteros.
min(long a, long b)	Devuelve el menor de dos valores long.
pow(double a, double b)	Devuelve el valor del primer argumento elevado a la potencia del segundo argumento.

random()	Devuelve un double con un signo positivo, mayor o igual que 0.0 y menor que 1.0.
round(double a)	Devuelve el long redondeado más cercano al double introducido.
sqrt(double a)	Devuelve la raíz cuadrada positiva correctamente redondeada de un double.
floor(double a)	Devuelve el entero más cercano por debajo.

MÉTODO RANDOM() DE LA CLASE MATH

El método `random` podemos utilizarlo para generar **números al azar**. El rango o margen con el que trabaja el método `random` oscila entre 0.0 y 1.0 (Este último no incluido)

Por lo tanto, para generar un número entero entre 0 y 9, hay que escribir la siguiente sentencia:

```
int numero = (int) (Math.random() * 10);
```

Nota: pueden encontrar un ejemplo de las **funciones de la clase String y Math** en Moodle.

EJERCICIOS DE APRENDIZAJE

A partir de ahora comenzaremos a aprender cómo los mismos algoritmos que diseñamos en PSeInt podemos escribirlos también en Java, simplemente haciendo una traducción de cada una de las estructuras de control vistas en PSeInt a Java.

Si bien en esta guía se proponen nuevos problemas, se sugiere que los mismos ejercicios ya implementados en PSeInt sean traducidos al lenguaje de programación Java.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

Para la realización cada uno de los siguientes ejercicios se debe definir una clase y colocar toda la implementación dentro del método *main()* de dicha clase, tal cual se indica en el video.

Dificultad Baja

Dificultad Media

Dificultad Alta

1. Escribir un programa que pida dos números enteros por teclado y calcule la suma de los dos. El programa deberá después mostrar el resultado de la suma
2. Escribir un programa que pida tu nombre, lo guarde en una variable y lo muestre por pantalla.
3. Escribir un programa que pida una frase y la muestre toda en mayúsculas y después toda en minúsculas. **Nota:** investigar la función `toUpperCase()` y `toLowerCase()` en Java.
4. Dada una cantidad de grados centígrados se debe mostrar su equivalente en grados Fahrenheit. La fórmula correspondiente es: $F = 32 + (9 * C / 5)$.
5. Escribir un programa que lea un número entero por teclado y muestre por pantalla el doble, el triple y la raíz cuadrada de ese número. **Nota:** investigar la función `Math.sqrt()`.

Condicionales en Java

6. Implementar un programa que dado dos números enteros determine cuál es el mayor y lo muestre por pantalla.
7. Crear un programa que dado un numero determine si es par o impar.
8. Crear un programa que pida una frase y si esa frase es igual a "eureka" el programa pondrá un mensaje de Correcto, sino mostrará un mensaje de Incorrecto. **Nota:** investigar la función `equals()` en Java.

9. Realizar un programa que solo permita introducir solo frases o palabras de 8 de largo. Si el usuario ingresa una frase o palabra de 8 de largo se deberá de imprimir un mensaje por pantalla que diga "CORRECTO", en caso contrario, se deberá imprimir "INCORRECTO". **Nota: investigar la función Length() en Java.**
10. Escriba un programa que pida una frase o palabra y valide si la primera letra de esa frase es una 'A'. Si la primera letra es una 'A', se deberá de imprimir un mensaje por pantalla que diga "CORRECTO", en caso contrario, se deberá imprimir "INCORRECTO". **Nota: investigar la función Substring y equals() de Java.**
11. Considera que estás desarrollando una web para una empresa que fabrica motores (suponemos que se trata del tipo de motor de una bomba para mover fluidos). Definir una variable tipoMotor y permitir que el usuario ingrese un valor entre 1 y 4. El programa debe mostrar lo siguiente:
- o Si el tipo de motor es 1, mostrar un mensaje indicando "La bomba es una bomba de agua".
 - o Si el tipo de motor es 2, mostrar un mensaje indicando "La bomba es una bomba de gasolina".
 - o Si el tipo de motor es 3, mostrar un mensaje indicando "La bomba es una bomba de hormigón".
 - o Si el tipo de motor es 4, mostrar un mensaje indicando "La bomba es una bomba de pasta alimenticia".
 - o Si no se cumple ninguno de los valores anteriores mostrar el mensaje "No existe un valor válido para tipo de bomba"

Bucles y sentencias de salto break y continue

12. Escriba un programa que valide si una nota está entre 0 y 10, sino está entre 0 y 10 la nota se pedirá de nuevo hasta que la nota sea correcta.
13. Escriba un programa en el cual se ingrese un valor límite positivo, y a continuación solicite números al usuario hasta que la suma de los números introducidos supere el límite inicial.
14. Escriba un programa que lea 20 números. Si el número leído es igual a cero se debe salir del bucle y mostrar el mensaje "Se capturó el numero cero". El programa deberá calcular y mostrar el resultado de la suma de los números leídos, pero si el número es negativo no debe sumarse. **Nota: recordar el uso de la sentencia break.**
15. Realizar un programa que pida dos números enteros positivos por teclado y muestre por pantalla el siguiente menú:



El usuario deberá elegir una opción y el programa deberá mostrar el resultado por pantalla y luego volver al menú. El programa deberá ejecutarse hasta que se elija la opción 5. Tener en cuenta que, si el usuario selecciona la opción 5, en vez de salir del programa directamente, se debe mostrar el siguiente mensaje de confirmación: ¿Está seguro que desea salir del programa (S/N)? Si el usuario selecciona el carácter 'S' se sale del programa, caso contrario se vuelve a mostrar el menú.

16. Realizar un programa que simule el funcionamiento de un dispositivo RS232, este tipo de dispositivo lee cadenas enviadas por el usuario. Las cadenas deben llegar con un formato fijo: tienen que ser de un máximo de 5 caracteres de largo, el primer carácter tiene que ser X y el último tiene que ser una O.

Las secuencias leídas que respeten el formato se consideran correctas, la secuencia especial "|||||&&&&&" marca el final de los envíos (llamémosla FDE), y toda secuencia distinta de FDE, que no respete el formato se considera incorrecta.

Al finalizar el proceso, se imprime un informe indicando la cantidad de lecturas correctas e incorrectas recibidas. Para resolver el ejercicio deberá investigar cómo se utilizan las siguientes funciones de Java **Substring()**, **Length()**, **equals()**.

17. Dibujar un cuadrado de N elementos por lado utilizando el carácter "*". Por ejemplo, si el cuadrado tiene 4 elementos por lado se deberá dibujar lo siguiente:

```
*****  
* * *  
* * *  
*****
```

18. Realizar un programa que lea 4 números (comprendidos entre 1 y 20) e imprima el número ingresado seguido de tantos asteriscos como indique su valor. Por ejemplo:

```
5 *****  
3 ***  
11 *****  
2 **
```

Vectores y Matrices en Java

19. Crea una aplicación que a través de una función nos convierta una cantidad de euros introducida por teclado a otra moneda, estas pueden ser a dólares, yenes o libras. La función tendrá como parámetros, la cantidad de euros y la moneda a convertir que será una cadena, este no devolverá ningún valor y mostrará un mensaje indicando el cambio (void).

El cambio de divisas es:

- * 0.86 libras es un 1 €
- * 1.28611 \$ es un 1 €
- * 129.852 yenes es un 1 €

Funciones en Java

20. Realizar un algoritmo que rellene un vector con los 100 primeros números enteros y los muestre por pantalla en orden descendente.

21. Realizar un algoritmo que rellene un vector de tamaño N con valores aleatorios y le pida al usuario un numero a buscar en el vector. El programa mostrará donde se encuentra el numero y si se encuentra repetido
22. Recorrer un vector de N enteros contabilizando cuántos números son de 1 dígito, cuántos de 2 dígitos, etcétera (hasta 5 dígitos).
23. Realizar un programa que rellene un matriz de 4×4 de valores aleatorios y muestre la traspuesta de la matriz.

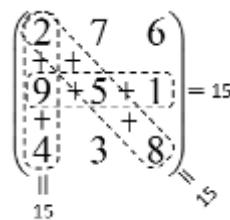
¿Cómo es la transpuesta de una matriz?

24. Realice un programa que compruebe si una matriz dada es anti simétrica. Se dice que una matriz A es anti simétrica cuando ésta es igual a su propia traspuesta, pero cambiada de signo. Es decir, A es anti simétrica si $A = -A^T$. La matriz traspuesta de una matriz A se denota por A^T y se obtiene cambiando sus filas por columnas (o viceversa).

Matriz			Matriz Traspuesta		
0	-2	4	0	2	-4
2	0	2	-2	0	-2
-4	-2	0	4	2	0

En este caso la matriz es anti simétrica.

25. Un cuadrado mágico 3×3 es una matriz 3×3 formada por números del 1 al 9 donde la suma de sus filas, sus columnas y sus diagonales son idénticas. Crear un programa que permita introducir un cuadrado por teclado y determine si este cuadrado es mágico o no. El programa deberá comprobar que los números introducidos son correctos, es decir, están entre el 1 y el 9.



26. Dadas dos matrices cuadradas de números enteros, la matriz M de 10×10 y la matriz P de 3×3 , se solicita escribir un programa en el cual se compruebe si la matriz P está contenida dentro de la matriz M. Para ello se debe verificar si entre todas las submatrices de 3×3 que se pueden formar en la matriz M, desplazándose por filas o columnas, existe al menos una que coincida con la matriz P. En ese caso, el programa debe indicar la fila y la columna de la matriz M en la cual empieza el primer elemento de la submatriz P.

Ejemplo:

Matriz de 10×10

1	26	36	47	5	6	72	81	95	10
11	12	13	21	41	22	67	20	10	61
56	78	87	90	09	90	17	12	87	67
41	87	24	56	97	74	87	42	64	35
32	76	79	1	36	5	67	96	12	11
99	13	54	88	89	90	75	12	41	76
67	78	87	45	14	22	26	42	56	78
98	45	34	23	32	56	74	16	19	18
24	67	97	46	87	13	67	89	93	24
21	68	78	98	90	67	12	41	65	12

Matriz de 3×3

36	5	67
89	90	75
14	22	26

Como podemos observar nuestra submatriz P se encuentra en la matriz M en los índices: 4,4 - 4,5 - 4,6 - 5,4 - 5,5 - 5,6 - 6,4 - 6,5 - 6,6.

EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termenes para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Dado un tiempo en minutos, calcular su equivalente en días y horas. Por ejemplo, si el usuario ingresa 1600 minutos, el sistema debe calcular su equivalente: 1 día, 2 horas.
2. Declarar cuatro variables de tipo entero **A**, **B**, **C** y **D** y asignarle un valor diferente a cada una. A continuación, realizar las instrucciones necesarias para que: **B** tome el valor de **C**, **C** tome el valor de **A**, **A** tome el valor de **D** y **D** tome el valor de **B**. Mostrar los valores iniciales y los valores finales de cada variable. Utilizar sólo una variable auxiliar.
3. Elaborar un algoritmo en el cuál se ingrese una letra y se detecte si se trata de una vocal. Caso contrario mostrar un mensaje. **Nota:** investigar la función **equals()** de la clase **String**.
4. Elaborar un algoritmo en el cuál se ingrese un número entre 1 y 10 y se muestre su equivalente en romano.

¿No te acordás los números romanos? [Consultalos acá.](#)

5. Una obra social tiene tres clases de socios:
 - o Los socios tipo 'A' abonan una cuota mayor, pero tienen un 50% de descuento en todos los tipos de tratamientos.
 - o Los socios tipo 'B' abonan una cuota moderada y tienen un 35% de descuento para los mismos tratamientos que los socios del tipo A.
 - o Los socios que menos aportan, los de tipo 'C', no reciben descuentos sobre dichos tratamientos.

Solicite una letra (carácter) que representa la clase de un socio, y luego un valor real que represente el costo del tratamiento (previo al descuento) y determine el importe en efectivo a pagar por dicho socio.

6. Leer la altura de N personas y determinar el promedio de estaturas que se encuentran por debajo de 1.60 mts. y el promedio de estaturas en general.

- 7.** Realice un programa que calcule y visualice el valor máximo, el valor mínimo y el promedio de n números ($n > 0$). El valor de n se solicitará al principio del programa y los números serán introducidos por el usuario. Realice dos versiones del programa, una usando el bucle “while” y otra con el bucle “do - while”.
- 8.** Escriba un programa que lea números enteros. Si el número es múltiplo de cinco debe detener la lectura y mostrar la cantidad de números leídos, la cantidad de números pares y la cantidad de números impares. Al igual que en el ejercicio anterior los números negativos no deben sumarse. **Nota: recordar el uso de la sentencia break.**
- 9.** Simular la división usando solamente restas. Dados dos números enteros mayores que uno, realizar un algoritmo que calcule el cociente y el residuo usando sólo restas. Método: Restar el dividendo del divisor hasta obtener un resultado menor que el divisor, este resultado es el residuo, y el número de restas realizadas es el cociente. Por ejemplo: $50 / 13$:

$50 - 13 = 37$ una resta realizada

$37 - 13 = 24$ dos restas realizadas

$24 - 13 = 11$ tres restas realizadas

dado que 11 es menor que 13, entonces: el residuo es 11 y el cociente es 3.

¿Aún no lo entendiste? [Consultá acá.](#)

- 10.** Realice un programa para que el usuario adivine el resultado de una multiplicación entre dos números generados aleatoriamente entre 0 y 10. El programa debe indicar al usuario si su respuesta es o no correcta. En caso que la respuesta sea incorrecta se debe permitir al usuario ingresar su respuesta nuevamente. Para realizar este ejercicio investigue como utilizar la función `Math.random()` de Java.
- 11.** Escribir un programa que lea un número entero y devuelva el número de dígitos que componen ese número. Por ejemplo, si introducimos el número 12345, el programa deberá devolver 5. Calcular la cantidad de dígitos matemáticamente utilizando el operador de división. **Nota: recordar que las variables de tipo entero truncan los números o resultados.**
- 12.** Necesitamos mostrar un contador con 3 dígitos (X-X-X), que muestre los números del 0-0-0 al 9-9-9, con la particularidad que cada vez que aparezca un 3 lo sustituya por una E. Ejemplo:

0-0-0

0-0-1

0-0-2

0-0-E

0-0-4

0-1-2

0-1-E

Nota: investigar función `equals()` y como convertir números a String.

13. Crear un programa que dibuje una escalera de números, donde cada línea de números comience en uno y termine en el número de la línea. Solicitar la altura de la escalera al usuario al comenzar. Ejemplo: si se ingresa el número 3:

```
1  
12  
123
```

14. Se dispone de un conjunto de N familias, cada una de las cuales tiene una M cantidad de hijos. Escriba un programa que pida la cantidad de familias y para cada familia la cantidad de hijos para averiguar la media de edad de los hijos de todas las familias.
15. Crea una aplicación que le pida dos números al usuario y este pueda elegir entre sumar, restar, multiplicar y dividir. La aplicación debe tener una función para cada operación matemática y deben devolver sus resultados para imprimirlas en el main.
16. Diseñe una función que pida el nombre y la edad de N personas e imprima los datos de las personas ingresadas por teclado e indique si son mayores o menores de edad. Después de cada persona, el programa debe preguntarle al usuario si quiere seguir mostrando personas y frenar cuando el usuario ingrese la palabra "No".
17. Crea una aplicación que nos pida un número por teclado y con una función se lo pasamos por parámetro para que nos indique si es o no un número primo, debe devolver true si es primo, sino false.

Un número primo es aquel solo puede dividirse entre 1 y si mismo. Por ejemplo: 25 no es primo, ya que 25 es divisible entre 5, sin embargo, 17 si es primo.

¿Qué son los números primos?

18. Realizar un algoritmo que calcule la suma de todos los elementos de un vector de tamaño N, con los valores ingresados por el usuario.
19. Escriba un programa que averigüe si dos vectores de N enteros son iguales (la comparación deberá detenerse en cuanto se detecte alguna diferencia entre los elementos).
20. Crear una función rellene un vector con números aleatorios, pasándole un arreglo por parámetro. Después haremos otra función o procedimiento que imprima el vector.
21. Los profesores del curso de programación de Egg necesitan llevar un registro de las notas adquiridas por sus 10 alumnos para luego obtener una cantidad de aprobados y desaprobados. Durante el periodo de cursado cada alumno obtiene 4 notas, 2 por trabajos prácticos evaluativos y 2 por parciales. Las ponderaciones de cada nota son:

Primer trabajo práctico evaluativo 10%
Segundo trabajo práctico evaluativo 15%
Primer Integrador 25%
Segundo integrador 50%

Una vez cargadas las notas, se calcula el promedio y se guarda en el arreglo. Al final del programa los profesores necesitan obtener por pantalla la cantidad de aprobados y desaprobados, teniendo en cuenta que solo aprueban los alumnos con promedio mayor o igual al 7 de sus notas del curso.

22. Realizar un programa que rellene una matriz de tamaño NxM con valores aleatorios y muestre la suma de sus elementos.
23. Construya un programa que lea 5 palabras de mínimo 3 y hasta 5 caracteres y, a medida que el usuario las va ingresando, construya una "sopa de letras para niños" de tamaño de 20×20 caracteres. Las palabras se ubicarán todas en orden horizontal en una fila que será seleccionada de manera aleatoria. Una vez concluida la ubicación de las palabras, rellene los espacios no utilizados con un número aleatorio del 0 al 9. Finalmente imprima por pantalla la sopa de letras creada.

Nota: Para resolver el ejercicio deberá investigar cómo se utilizan las siguientes funciones de Java substring(), Length() y Math.random().

24. Realizar un programa que complete un vector con los N primeros números de la sucesión de Fibonacci. Recordar que la sucesión de Fibonacci es la sucesión de los siguientes números:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Donde cada uno de los números se calcula sumando los dos anteriores a él. Por ejemplo:

La sucesión del número 2 se calcula sumando (1+1)

Análogamente, la sucesión del número 3 es (1+2),

Y la del 5 es (2+3),

Y así sucesivamente...

La sucesión de Fibonacci se puede formalizar de acuerdo a la siguiente fórmula:

$Fibonacci_n = Fibonacci_{n-1} + Fibonacci_{n-2}$ para todo $n > 1$
 $Fibonacci_n = 1$ para todo $n \leq 1$

Por lo tanto, si queremos calcular el término "n" debemos escribir una función que reciba como parámetro el valor de "n" y que calcule la serie hasta llegar a ese valor.

Para conocer más acerca de la serie de Fibonacci consultar el siguiente link:
<https://quantdare.com/numeros-de-fibonacci/>

CURSO DE PROGRAMACIÓN FULL STACK

PROGRAMACIÓN ORIENTADA A OBJETOS

PARADIGMA ORIENTADO A OBJETOS



EGG

GUÍA DE PARADIGMA ORIENTADO A OBJETOS

PARADIGMAS DE PROGRAMACIÓN

Un **paradigma de programación** es una manera o estilo de programación. Existen diferentes formas de diseñar un programa y varios modos de trabajar para obtener los resultados que necesitan los programadores. Por lo que un paradigma de programación se trata de un conjunto de métodos sistemáticos aplicables en todos los niveles del diseño de programas para resolver problemas.

PROGRAMACIÓN ORIENTADA A OBJETOS

La Programación Orientada a Objetos (POO) es un paradigma de programación, es decir, un modelo o un estilo de programación que se basa en el concepto de **clases y objetos**. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de código (clases) para crear instancias individuales de objetos.

Con el paradigma de Programación Orientado a Objetos lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma.

La programación orientada a objetos se enfoca en los **objetos, sus atributos y las interacciones** que se producen entre ellos para diseñar programas.

Un programa orientado a objetos es, esencialmente, un **conjunto de objetos** que se crean, **interaccionan entre sí** y dejan de existir cuando ya no son útiles durante la ejecución de un programa. Un programa puede llegar a ser muy complejo. La complejidad es más manejable cuando se descompone y se organiza en partes pequeñas y simples, los objetos.

¿POR QUÉ POO?

La Programación Orientada a objetos permite que el código sea reutilizable, organizado y fácil de mantener. Sigue el principio de desarrollo de *software* utilizado por muchos programadores **DRY (Don't Repeat Yourself)**, para evitar duplicar el código y crear de esta manera programas eficientes. Además, evita el acceso no deseado a los datos o la exposición de código propietario mediante la encapsulación y la abstracción, de la que hablaremos en detalle más adelante.

CLASES Y OBJETOS

Una **clase es un molde** para crear múltiples objetos que encapsula datos y comportamiento. Una clase es una combinación específica de atributos y métodos y puede considerarse un tipo de dato de cualquier tipo no primitivo.

Así, una clase es una especie de plantilla o prototipo de objetos: define los atributos que componen ese tipo de objetos y los métodos que pueden emplearse para trabajar con esos objetos. En su forma más simple, una clase se define por la palabra reservada class seguida del nombre de la clase. El nombre de la clase debe empezar por mayúscula.

Si el nombre es compuesto, entonces cada palabra debe empezar por mayúscula. La definición de la clase se pone entre las llaves de apertura y cierre.

```
public class NombreClase {  
    // atributos  
    // constructores  
    // metodos  
}
```

Una vez que se ha declarado una clase, se pueden crear objetos a partir de ella. A la creación de un objeto se le denomina instancia. Por esta razón que se dice que un objeto es una instancia de una clase y el término instancia y objeto se utilizan indistintamente. Para crear objetos, basta con declarar una variable de alguno de los tipos de las clases definidas.

```
NombreClase nombreObjeto;
```

Para crear el objeto y asignar un espacio de memoria es necesario realizar la instancia con el operador new. El operador new instancia el objeto y reserva espacio en memoria para los atributos y devuelve una referencia que se guarda en la variable.

```
nombreObjeto = new NombreClase();
```

Tanto la declaración de un objeto como la asignación del espacio de memoria se pueden realizar en un solo paso:

```
NombreClase nombreObjeto = new NombreClase();
```

A partir de este momento los objetos ya pueden ser referenciados por su nombre.

ACCESO A LOS ATRIBUTOS

Desde un objeto se puede acceder a los atributos mediante la siguiente sintaxis:

```
nombreObjeto.atributo;
```

ESTADO Y COMPORTAMIENTO

En términos más generales, un objeto es una abstracción conceptual del mundo real que se puede traducir a un lenguaje de programación orientado a objetos. Los objetos del mundo real comparten dos características: Todos poseen *estado* y *comportamiento*. Por ejemplo, el perro tiene estado (color, nombre, raza, edad) y el comportamiento (ladrar, caminar, comer, acostarse, mover la cola). Por lo tanto, un estado permite informar cuáles son las características del objeto y lo que este representa, y el comportamiento, consiste en decir lo que sabe hacer.

El estado de un objeto es una lista de variables conocidas como sus atributos, cuyos valores representan el estado que caracteriza al objeto.

El comportamiento es una lista de métodos, procedimientos, funciones u operaciones que un objeto puede ejecutar a solicitud de otros objetos. Los objetos también se conocen como instancias.

ELEMENTOS DE UNA CLASE

Una clase describe un tipo de objetos con características comunes. Es necesario definir la información que almacena el objeto y su comportamiento.

ATRIBUTOS

El estado o información de un objeto se almacena en atributos. Los atributos pueden ser de tipos primitivos de Java (descriptos en la guía Intro Java) o del tipo de otros objetos. La declaración de un atributo de un objeto tiene la siguiente forma:

```
<modificador>* <tipo> <nombre> [ = <valor inicial> ];
```

- **<nombre>**: puede ser cualquier identificador válido y denomina el atributo que está siendo declarado.
- **<modificador>**: si bien hay varios valores posibles para el **<modificador>**, por el momento solo usaremos modificadores de visibilidad: public, protected, private.
- **<tipo>**: indica la clase a la que pertenece el atributo definido.
- **<valor inicial>**: esta sentencia es opcional y se usa para inicializar el atributo del objeto con un valor particular.

Estos atributos irán al principio de la clase.

CONSTRUCTORES

Además de definir los atributos de un objeto, es necesario definir los métodos que determinan su comportamiento. Toda clase debe definir un método especial denominado constructor para instanciar los objetos de la clase. Este método tiene el mismo nombre de la clase. La declaración básica toma la siguiente forma:

```
[<modificador>] <nombre de clase> ( <argumento>* ) {  
    <sentencia>*  
}
```

- **<nombre de clase>**: El nombre del constructor debe ser siempre el mismo que el de la clase.
- **<modificador>**: Actualmente, los únicos modificadores válidos para los constructores son public, protected y private.
- **<argumentos>**: es una lista de parámetros que tiene la misma función que en los métodos.

El método constructor se ejecuta cada vez que se instancia un objeto de la clase. Este método se utiliza para **inicializar** los atributos del objeto que se instancia.

Para diferenciar entre los atributos del objeto y los identificadores de los parámetros del método constructor, se utiliza la palabra this. De esta forma, los parámetros del método pueden tener el mismo nombre que los atributos de la clase.

La instanciación de un objeto consiste en asignar un espacio de memoria al que se hace referencia con el nombre del objeto. Los identificadores de los objetos permiten acceder a los valores almacenados en cada objeto.

El Constructor por Defecto

Cada clase tiene al menos un constructor. Si no se escribe un constructor, el lenguaje de programación Java le provee uno por defecto. Este constructor no posee argumentos y tiene un cuerpo vacío. Si se define un constructor que no sea vacío, el constructor por defecto se pierde, salvo que creamos un nuevo constructor vacío.

MÉTODOS

Los métodos son funciones que determinan el comportamiento de los objetos. Un objeto se comporta de una u otra forma dependiendo de los métodos de la clase a la que pertenece. Todos los objetos de una misma clase tienen los mismos métodos y el mismo comportamiento. Para definir los métodos, el lenguaje de programación Java toma la siguiente forma básica:

```
<modificador>* <tipo de retorno> <nombre> ( <argumento>* ) {  
    <sentencias>  
    return valorRetorno;  
}
```

- **<nombre>**: puede ser cualquier identificador válido, con algunas restricciones basadas en los nombres que ya están en uso.
- **<modificador>**: el segmento es opcional y puede contener varios modificadores diferentes incluyendo a public, protected y private. Aunque no está limitado a estos.
- **<tipo de retorno>**: el tipo de retorno indica el tipo de valor devuelto por el método. Si el método no devuelve un valor, debe ser declarado void. La tecnología Java es rigurosa acerca de los valores de retorno. Si el tipo de retorno en la declaración del método es un int, por ejemplo, el método debe devolver un valor int desde todos los posibles caminos de retorno (y puede ser invocado solamente en contextos que esperan un int para ser devuelto). Se usa la sentencia return dentro de un método para devolver un valor.
- **<argumento>**: permite que los valores de los argumentos sean pasados hacia el método. Los elementos de la lista están separados por comas y cada elemento consiste en un tipo y un identificador.

Existen tres tipos de métodos: métodos de consulta, métodos modificadores y operaciones. Los métodos de consulta sirven para extraer información de los objetos, los métodos modificadores sirven para modificar el valor de los atributos del objeto y las operaciones definen el comportamiento de un objeto.

Para acceder a los atributos de un objeto se definen los métodos get y set. Los métodos get se utilizan para consultar el estado de un objeto y los métodos set para modificar su estado. Un método get se declara public y a continuación se indica el tipo de dato que devuelve. Es un método de consulta. La lista de parámetros de un método get queda vacía. En el cuerpo del método se utiliza return para devolver el valor correspondiente al atributo que se quiere devolver, y al cual se hace referencia como this.nombreAtributo.

Por otra parte, un método set se declara public y devuelve void. La lista de parámetros de un método set incluye el tipo y el valor a modificar. Es un método modificador. El cuerpo de un método set asigna al atributo del objeto el parámetro de la declaración.

Por último, un método de tipo operación es aquel que realiza un cálculo o modifica el estado de un objeto. Este tipo de métodos pueden incluir una lista de parámetros y puede devolver un valor o no. Si el método no devuelve un valor, se declara void.

Invocación de métodos

Un método se puede invocar dentro o fuera de la clase donde se ha declarado. Si el método se invoca dentro de la clase, basta con indicar su nombre. Si el método se invoca fuera de la clase entonces se debe indicar el nombre del objeto y el nombre del método. Cuando se invoca a un método ocurre lo siguiente:

- En la línea de código del programa donde se invoca al método se calculan los valores de los argumentos.
- Los parámetros se inicializan con los valores de los argumentos.
- Se ejecuta el bloque código del método hasta que se alcanza return o se llega al final del bloque.
- Si el método devuelve un valor, se sustituye la invocación por el valor devuelto.
- La ejecución del programa continúa en la siguiente instrucción donde se invocó el método.

Parámetros y argumentos

Los parámetros de un método definen la cantidad y el tipo de dato de los valores que recibe un método para su ejecución. Los argumentos son los valores que se pasan a un método durante su invocación. El método recibe los argumentos correspondientes a los parámetros con los que ha sido declarado. Un método puede tener tantos parámetros como sea necesario. La lista de parámetros de la cabecera de un método se define con la siguiente sintaxis:

tipo nombre [tipo nombre,]

Durante la invocación de un método es necesario que el número y el tipo de argumentos coincidan con el número y el tipo de parámetros declarados en la cabecera del método. Durante el proceso de compilación se comprueba que durante la invocación de un método se pasan tantos argumentos como parámetros tiene declarados y que además coinciden los tipos. Esta es una característica de los lenguajes que se denominan "strongly typed" o "fuertemente tipado"

Paso de parámetros

Cuando se invoca un método se hace una copia de los valores de los argumentos en los parámetros. Esto quiere decir que, si el método modifica el valor de un parámetro, nunca se modifica el valor original del argumento.

Cuando se pasa una referencia a un objeto se crea un nuevo alias sobre el objeto, de manera que esta nueva referencia utiliza el mismo espacio de memoria del objeto original y esto permite acceder al objeto original.

El valor de retorno

Un método puede devolver un valor. Los métodos que no devuelven un valor se declaran void, mientras que los métodos que devuelven un valor indican el tipo que devuelven: int, double, char, String o un tipo de objeto.

Sobrecarga de métodos

La sobrecarga de métodos es útil para que el mismo método opere con parámetros de distinto tipo o que un mismo método reciba una lista de parámetros diferente. Esto quiere decir que puede haber dos métodos con el mismo nombre que realicen dos funciones distintas. La diferencia entre los métodos sobrecargados está en su declaración, y más específicamente, en la cantidad y tipos de datos que reciben.

ABSTRACCIÓN Y ENCAPSULAMIENTO

La abstracción es la habilidad de ignorar los detalles de las partes para enfocar la atención en un nivel más alto de un problema. El encapsulamiento sucede cuando algo es envuelto en una capa protectora. Cuando el encapsulamiento se aplica a los objetos, significa que los datos del objeto están protegidos, “ocultos” dentro del objeto. Con los datos ocultos, ¿cómo puede el resto del programa acceder a ellos? (El acceso a los datos de un objeto se refiere a leerlos o modificarlos.) El resto del programa no puede acceder de manera directa a los datos de un objeto; lo tiene que hacer con ayuda de los métodos del objeto. Al hecho de proteger los datos o atributos con los métodos se denomina encapsulamiento.

ABSTRACCIÓN

La abstracción es la propiedad que considera los aspectos más significativos o notables de un problema y expresa una solución en esos términos. La abstracción posee diversos grados o niveles de abstracción, los cuales ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. La abstracción encarada desde el punto de vista de la programación orientada a objetos es el mecanismo por el cual se proveen los límites conceptuales de los objetos y se expresan sus características esenciales, dejando de lado sus características no esenciales. Si un objeto tiene más características de las necesarias los mismos resultan difíciles de usar, modificar, construir y comprender. En el análisis hay que concentrarse en ¿Qué hace? y no en ¿Cómo lo hace?

ENCAPSULAMIENTO

La encapsulación o encapsulamiento significa reunir en una cierta estructura a todos los elementos que a un cierto nivel de abstracción se pueden considerar pertenecientes a una misma entidad, y es el proceso de agrupamiento de datos y operaciones relacionadas bajo una misma unidad de programación, lo que permite aumentar la cohesión de los componentes del sistema.

El encapsulamiento **oculta lo que hace un objeto** de lo que hacen otros objetos y del mundo exterior por lo que se denomina también ocultación de datos. Un objeto tiene que presentar “una cara” al mundo exterior de modo que se puedan iniciar sus operaciones.

Los métodos operan sobre el estado interno de un objeto y sirven como el mecanismo primario de comunicación entre objetos. Ocultar el estado interno y hacer que toda interacción sea a través de los métodos del objeto es un mecanismo conocido como encapsulación de datos.

MODIFICADORES DE ACCESO

Para lograr el uso correcto del encapsulamiento vamos utilizar los modificadores de acceso, estos, van a dejarnos elegir como se accede a los datos y a través de que se accede a dichos datos. Todas las clases poseen diferentes niveles de acceso en función del modificador de acceso (visibilidad). A continuación, se detallan los niveles de acceso con sus símbolos correspondientes:

- **Public:** Este modificador permite a acceder a los elementos desde cualquier clase, independientemente de que esta pertenezca o no al paquete en que se encuentra el elemento.
- **Private:** Es el modificador más restrictivo y especifica que los elementos que lo utilizan sólo pueden ser accedidos desde la clase en la que se encuentran. Este modificador sólo puede utilizarse sobre los atributos de una clase y sobre interfaces y clases internas, no sobre clases o interfaces de primer nivel, dado que esto no tendría sentido. Es importante destacar también que el modificador private convierte los elementos en privados para otras clases, no para otras instancias de la clase. Es decir, un objeto de una determinada clase puede acceder a los atributos privados de otro objeto de la misma clase.
- **Protected:** Este modificador indica que los elementos sólo pueden ser accedidos desde su mismo paquete y desde cualquier clase que extienda la clase en que se encuentra, independientemente de si esta se encuentra en el mismo paquete o no. Este modificador, como private, no tiene sentido a nivel de clases o interfaces no internas.

Si no especificamos ningún modificador de acceso se utiliza el nivel de acceso por defecto (Default), que consiste en que el elemento puede ser accedido sólo desde las clases que pertenezcan al mismo paquete. Los distintos modificadores de acceso quedan resumidos en la siguiente tabla

Visibilidad	Public	Private	Protected	Default
Desde la misma Clase	SI	SI	SI	SI
Desde cualquier Clase del mismo Paquete	SI	NO	SI	SI
Desde una Subclase del mismo Paquete	SI	NO	SI	SI
Desde una Subclase fuera del mismo Paquete		NO	SI, a través de la herencia	NO
Desde cualquier Clase fuera del Paquete	SI	NO	NO	NO

ATRIBUTOS Y MÉTODOS ESTÁTICOS

Un atributo o un método de una clase se puede modificar con la palabra reservada static para indicar que este atributo o método no pertenece a las instancias de la clase si no a la propia clase.

Se dice que son atributos de clase si se usa la palabra clave static: en ese caso la variable es única para todas las instancias (objetos) de la clase (ocupa un único lugar en memoria), es decir que, si se poseen múltiples instancias de una clase, cada una de ellas no tendrá una copia propia de este atributo, si no que todas estas instancias compartirán una misma copia del atributo. A veces a las variables de clase se les llama variables estáticas. Si no se usa static, el sistema crea un lugar nuevo para esa variable con cada instancia (la variable es diferente para cada objeto).

En el caso de una constante no tiene sentido crear un nuevo lugar de memoria por cada objeto de una clase que se cree. Por ello es adecuado el uso de la palabra clave static. Cuando usamos "static final" se dice que creamos una constante de clase, un atributo común a todos los objetos de esa clase.

```
public class Cuenta {
    private static String saldo;
}
```

ATRIBUTOS FINALES

En este contexto indica que una variable es de tipo constante: no admitirá cambios después de su declaración y asignación de valor. La palabra reservada **final** determina que un atributo no puede ser sobrescrito o redefinido, es decir, no funcionará como una variable "tradicional", sino como una constante. Toda constante declarada con *final* ha de ser inicializada en el mismo momento de declararla. El modificador *final* también se usa como palabra clave en otro contexto: una clase *final* es aquella que no puede tener clases que la hereden. Lo veremos más adelante cuando hablemos sobre herencia.

Cuando se declaran constantes es muy frecuente que los programadores usen letras mayúsculas (como práctica habitual que permite una mayor claridad en el código), aunque no es obligatorio.

```
Public class Perro {  
    private final int edad;  
}
```

EN RESUMEN

Antes de POO, la técnica estándar de programación era la programación procedural. Se denomina programación procedural porque en ella se destacan los procedimientos o tareas que resuelven un problema. Se piensa primero en lo que se quiere hacer: los procedimientos.

En contraste, el paradigma POO invita a pensar en lo que se desea que represente el programa. Normalmente se responde esta invitación identificando algunas cosas en el mundo que se desea que el programa modele.

Estas cosas podrían ser entidades físicas o conceptuales, por ejemplo, un libro. Una vez identificadas las cosas que se quiere modelar, se identifican sus propiedades/atributos básicos. Estos se pueden agrupar todos juntos en una estructura coherente llamada objeto que creamos a través de las clases.

Proyecto con ejemplos de POO

Encontrarán en Moodle un ejemplo descargable.

El ejemplo está pensado para una pagina que va a administrar perros en su pagina web, muestra las distintas maneras de llenar y de mostrar un objeto.

CLASE CONTROL

La clase control, va a ser una clase auxiliar que nos va a ayudar con el manejo de las clases y los objetos de esas clases, pero para poder explicar esto, primero vamos a tener que ver **los patrones generales de software GRASP**. Aunque se considera que más que **patrones** propiamente dichos, son una serie de "buenas prácticas" de aplicación recomendable en el diseño de software.

PATRONES GRASP

Un proceso de desarrollo sirve para normalizar quien hace que cosa en cada momento y como debe realizarse esta cosa.

GRASP es el acrónimo de General **R**esponsibility **A**ssignment **S**oftware **P**atterns. Una de las cosas más complicadas en Orientación a Objeto consiste en elegir las clases adecuadas decidir como estas clases deben interactuar.

PATRON EXPERTO

Dentro de los patrones GRASP, vamos a utilizar el patrón experto. El GRASP de experto en información es el principio básico de asignación de responsabilidades. Nos indica, por ejemplo, que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo. De este modo obtendremos un diseño con mayor cohesión y así la información se mantiene encapsulada (disminución del acoplamiento).

Problema

¿Cuál es el principio general para asignar responsabilidades a los objetos?

Solución

Asignar una responsabilidad al experto en información.

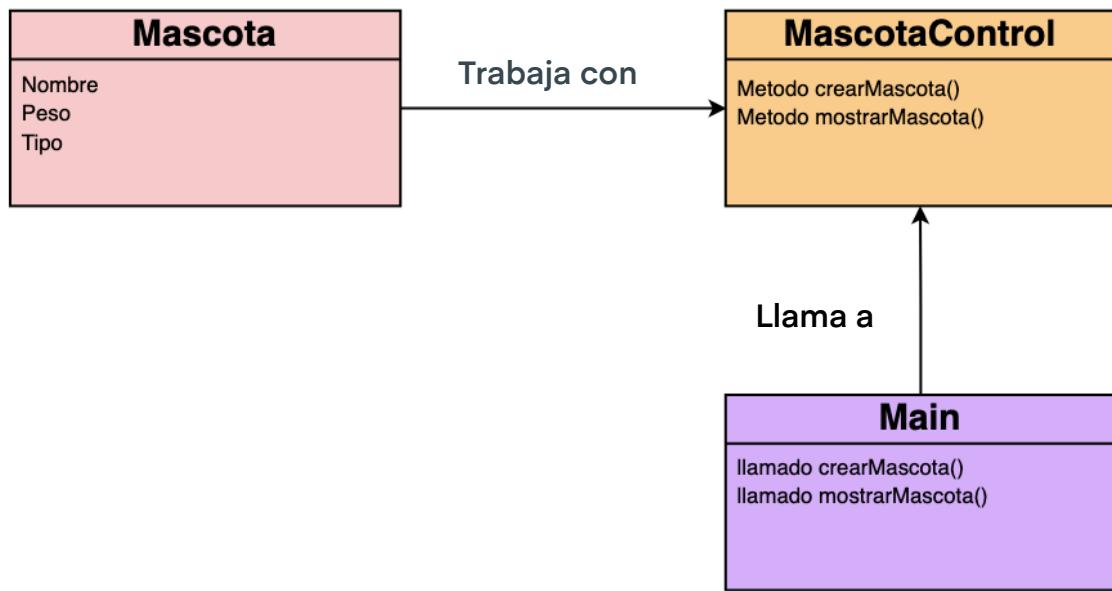
Beneficios

Se mantiene el encapsulamiento, los objetos utilizan su propia información para llevar a cabo sus tareas. Se distribuye el comportamiento entre las clases que contienen la información requerida. Son más fáciles de entender y mantener.

CLASE CONTROL

Esta clase del patrón experto, va a ser la clase control. La clase control, es una clase común y corriente pero que se va a encargar de crear los objetos y va a tener todos los métodos necesarios para la utilización de ese objeto. Supongamos que necesitamos un método que le sume un valor x a un atributo del objeto, ese método estará en la clase control.

Siempre se crea una clase control, por cada clase que tengamos, si tenemos las clases Perro y Gato, crearemos una clase control para Perro y otra para Gato. La idea es que una clase control, se encargue de solo una clase.



Como podemos ver en el diagrama, MascotaControl llama y trabaja con la clase Mascota y crea instancias de ese objeto a través de un método, también, tiene un método para mostrar la mascota.

Pero, el encargado de llamar a esos métodos de la clase MascotaControl va a ser la clase Main, que vendría a ser el usuario llamando a las cosas que quiere realizar con la Mascota.

CLASES DE UTILIDAD PARTE 2

Recordemos que las clases de utilidad son clases dentro del API de Java que son muy utilizadas en el desarrollo de aplicaciones. Las clases de utilidad son clases que definen un conjunto de métodos que realizan funciones, normalmente muy reutilizadas. Estas nos van a ayudar junto con las estructuras de control, a lograr resolver problemas de manera más sencilla.

Entre las clases de utilidad de Java más utilizadas y conocidas están las siguientes: Arrays, String, Integer, Math, Date, Calendar y GregorianCalendar. En la guía anterior vimos solo las clases Math y String. Ahora vamos a ver el resto de las clases.

CLASE ARRAYS

La clase Arrays es una clase de utilidad que posee una gran cantidad de métodos para manipular arreglos.

Método	Descripción.
Arrays.equals(arreglo1, arreglo2)	Retorna true o false, si dos arreglos del mismo tipo de dato son iguales.
Arrays.fill(arreglo, variable) Arrays.fill(arreglo, int desde, int hasta, variable)	Este método lo que hace es inicializar todo el arreglo con la variable o valor que pasamos como argumento. Este método se puede usar con cualquier tipo de dato y le podemos decir desde y hasta que índice queremos que llene con ese valor.
Arrays.sort(arreglo) Arrays.sort(arreglo, int desde, int hasta)	Este método sirve para ordenar un arreglo de manera ascendente. A este método también le podemos decir desde y hasta que índice queremos que ordene.
Arrays.toString(arreglo)	Este método imprime el arreglo como una cadena, la cadena consiste en una lista de los elementos del arreglo encerrados entre corchetes ("[]"). Los elementos adyacentes están separados por comas (",").

CLASE INTEGER

La clase Integer permite convertir un tipo primitivo de dato int a objeto Integer. La clase Integer pertenece al paquete java.lang del API de Java y hereda de la clase java.lang.Number.

Método	Descripción.
<code>Integer(String s)</code>	Constructor que inicializa un objeto con una cadena de caracteres. Esta cadena debe contener un número entero.
<code>compareTo(entero, otroEntero)</code>	Compara dos objetos Integer numéricamente. Retorna 0 si son iguales, entero negativo si el primer numero es menor o entero positivo si el primer numero es mayor.
<code>doubleValue()</code>	Retorna el valor del Integer en tipo primitivo double
<code>equals(Object obj)</code>	Compara el Integer con el objeto del parámetro. Devuelve true si son iguales y false si no.
<code>parseInt(String s)</code>	Convierte la cadena de caracteres numérica del parámetro en tipo primitivo int.
<code>toString()</code>	Retorna el valor del Integer en una cadena de caracteres.

CLASE DATE

La clase Date modela objetos o variables de tipo fecha. La clase Date representa un instante de tiempo específico con una precisión en milisegundos y permite el uso del formato Universal Coordinated Time (UTC). Por otro lado, muchas computadoras están definidas en términos de Greenwich Mean Time (GMT) que es equivalente a Universal Time (UT). GMT es el nombre estándar y UT es el nombre científico del estándar. La diferencia entre UT y UTC es que UTC está basado en un reloj atómico y UT está basado en un reloj astronómico.

Las fechas en Java, comienzan en el valor standar based time llamado "epoch" que hace referencia al 1 de Enero de 1970, 0 horas 0 minutos 0 segundos GMT.

La clase Date posee métodos que permiten la manipulación de fechas. La clase Date pertenece al paquete java.util del API de Java.

Método	Descripción.
<code>Date()</code>	Constructor que inicializa la fecha en el milisegundo más cercano a la fecha del sistema.
<code>Date(int dia, int mes, int año)</code>	Constructor que inicializa la fecha sumándole 1900 al año.

<code>after(Date fecha2)</code>	Retorna verdadero si la fecha esta después de la fecha del parámetro
<code>before(Date fecha2)</code>	Retorna verdadero si la fecha esta antes de la fecha del parámetro
<code>compareTo(Date fecha)</code>	Compara la fecha con la del parámetro. Retorna 0 si son iguales, entero negativo si el primer numero es menor o entero positivo si el primer numero es mayor.
<code>equals(Object obj)</code>	Compara el Date con el objeto del parámetro. Devuelve true si son iguales y false si no.
<code>getDay()</code>	Retorna el valor del día de la semana de la fecha. Ejemplo: si es lunes devuelve 0, martes 1, miércoles 2, jueves 3, viernes 4, sábado 5 y domingo 6.
<code>getDate()</code>	Retorna el numero del día de la fecha.
<code>getMonth()</code>	Retorna el mes de la fecha.
<code>getYear()</code>	Retorna el año de la fecha.
<code>getTime()</code>	Retorna la fecha en milisegundos a partir del "epoch".
<code> setDate(int dia)</code>	Asigna un día a la fecha.
<code> setMonth(int mes)</code>	Asigna un mes a la fecha.
<code> setYear(int anio)</code>	Asigna un año a la fecha.
<code> setTime(long time)</code>	Asigna la fecha en milisegundos a partir del "epoch".
<code>toString()</code>	Retorna la fecha en una cadena de caracteres.

Proyecto con ejemplos de Clase Utilidad

Encontrarán en Moodle un ejemplo descargable.

En este proyecto hay un Main con todas las clases de utilidad previamente vistas.

EJERCICIOS DE APRENDIZAJE

Antes de comenzar con esta guía, les damos algunas recomendaciones:

Este módulo es uno de los más divertidos ya que vamos a comenzar a modelar los objetos del mundo real con el lenguaje de programación Java. Es importante tener en cuenta que entender la programación orientada a objetos lleva tiempo y sobre todo PRÁCTICA, así que, a no desesperarse, con cada ejercicio vamos a ir entendiendo un poco más cómo aplicar este paradigma.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Crear una clase llamada Libro que contenga los siguientes atributos: **ISBN**, **Título**, **Autor**, **Número de páginas**, y un constructor con todos los atributos pasados por parámetro y un constructor vacío. Crear un método para cargar un libro pidiendo los datos al usuario y luego informar mediante otro método el número de ISBN, el título, el autor del libro y el numero de páginas.
2. Declarar una clase llamada Circunferencia que tenga como atributo privado el **radio de tipo real**. A continuación, se deben crear los siguientes métodos:
 - a) Método constructor que inicialice el radio pasado como parámetro.
 - b) Métodos get y set para el atributo radio de la clase Circunferencia.
 - c) Método para crearCircunferencia(): que le pide el radio y lo guarda en el atributo del objeto.
 - d) Método **area()**: para calcular el área de la circunferencia (**Area = $\pi * radio^2$**).
 - e) Método **perimetro()**: para calcular el perímetro (**Perímetro = $2 * \pi * radio$**).
3. Crear una clase llamada Operacion que tenga como atributos privados **numero1** y **numero2**. A continuación, se deben crear los siguientes métodos:
 - a) Método constructor con todos los atributos pasados por parámetro.
 - b) Método constructor sin los atributos pasados por parámetro.
 - c) Métodos get y set.
 - d) Método para crearOperacion(): que le pide al usuario los dos números y los guarda en los atributos del objeto.
 - e) Método **sumar()**: calcular la suma de los números y devolver el resultado al main.
 - f) Método **restar()**: calcular la resta de los números y devolver el resultado al main
 - g) Método **multiplicar()**: primero valida que no se haga una multiplicación por cero, si fuera a multiplicar por cero, el método devuelve 0 y se le informa al usuario el error. Si no, se hace la multiplicación y se devuelve el resultado al main
 - h) Método **dividir()**: primero valida que no se haga una división por cero, si fuera a pasar una división por cero, el método devuelve 0 y se le informa al usuario el error se le informa al usuario. Si no, se hace la división y se devuelve el resultado al main.

4. Crear una clase Rectángulo que modele rectángulos por medio de un atributo privado **base** y un atributo privado **altura**. La clase incluirá un método para crear el rectángulo con los datos del Rectángulo dados por el usuario. También incluirá un método para calcular la superficie del rectángulo y un método para calcular el perímetro del rectángulo. Por último, tendremos un método que dibujará el rectángulo mediante asteriscos usando la base y la altura. Se deberán además definir los métodos getters, setters y constructores correspondientes.

Superficie = base * altura / Perímetro = (base + altura) * 2.

5. Realizar una clase llamada Cuenta (bancaria) que debe tener como mínimo los atributos: **numeroCuenta** (entero), **el DNI del cliente** (entero largo), **el saldo actual**. Las operaciones asociadas a dicha clase son:

- Constructor por defecto y constructor con DNI, saldo, número de cuenta e interés.
- Agregar los métodos getters y setters correspondientes
- Método para crear un objeto Cuenta, pidiéndole los datos al usuario.
- Método ingresar(double ingreso): el método recibe una cantidad de dinero a ingresar y se la sumara a saldo actual.
- Método retirar(double retiro): el método recibe una cantidad de dinero a retirar y se la restará al saldo actual. Si la cuenta no tiene la cantidad de dinero a retirar, se pondrá el saldo actual en 0.
- Método extraccionRapida(): le permitirá sacar solo un 20% de su saldo. Validar que el usuario no saque más del 20%.
- Método consultarSaldo(): permitirá consultar el saldo disponible en la cuenta.
- Método consultarDatos(): permitirá mostrar todos los datos de la cuenta

6. Programa Nespresso. Desarrolle una clase Cafetera con los atributos **capacidadMaxima** (la cantidad máxima de café que puede contener la cafetera) y **cantidadActual** (la cantidad actual de café que hay en la cafetera). Implemente, al menos, los siguientes métodos:

- Constructor predeterminado o vacío
- Constructor con la capacidad máxima y la cantidad actual
- Métodos getters y setters.
- Método llenarCafetera(): hace que la cantidad actual sea igual a la capacidad máxima.
- Método servirTaza(int): se pide el tamaño de una taza vacía, el método recibe el tamaño de la taza y simula la acción de servir la taza con la capacidad indicada. Si la cantidad actual de café “no alcanza” para llenar la taza, se sirve lo que quede. El método le informará al usuario si se llenó o no la taza, y de no haberse llenado en cuanto quedó la taza.
- Método vaciarCafetera(): pone la cantidad de café actual en cero.
- Método agregarCafe(int): se le pide al usuario una cantidad de café, el método lo recibe y se añade a la cafetera la cantidad de café indicada.

7. Realizar una clase llamada Persona que tenga los siguientes atributos: nombre, edad, sexo ('H' hombre, 'M' mujer, 'O' otro), peso y altura. Si el alumno desea añadir algún otro atributo, puede hacerlo. Los métodos que se implementarán son:

- Un constructor por defecto.
- Un constructor con todos los atributos como parámetro.
- Métodos getters y setters de cada atributo.
- Método crearPersona(): el método crear persona, le pide los valores de los atributos al usuario y después se le asignan a sus respectivos atributos para llenar el objeto Persona. Además, comprueba que el sexo introducido sea correcto, es decir, H, M o O. Si no es correcto se deberá mostrar un mensaje
- Método calcularIMC(): calculará si la persona está en su peso ideal ($\text{peso en kg}/(\text{altura}^2 \text{ en mt}^2)$). Si esta fórmula da por resultado un valor menor que 20, significa que la persona está por debajo de su peso ideal y la función devuelve un -1. Si la fórmula da por resultado un número entre 20 y 25 (incluidos), significa que la persona está en su peso ideal y la función devuelve un 0. Finalmente, si el resultado de la fórmula es un valor mayor que 25 significa que la persona tiene sobrepeso, y la función devuelve un 1.
- Método esMayorDeEdad(): indica si la persona es mayor de edad. La función devuelve un booleano.

A continuación, en la clase main hacer lo siguiente:

Crear 4 objetos de tipo Persona con distintos valores, a continuación, llamaremos todos los métodos para cada objeto, deberá comprobar si la persona está en su peso ideal, tiene sobrepeso o está por debajo de su peso ideal e indicar para cada objeto si la persona es mayor de edad.

Por último, guardaremos los resultados de los métodos calcularIMC y esMayorDeEdad en distintas variables, para después en el main, calcular un porcentaje de esas 4 personas cuantas están por debajo de su peso, cuantas en su peso ideal y cuantos, por encima, y también calcularemos un porcentaje de cuantos son mayores de edad y cuantos menores.

CLASES DE UTILIDAD EN JAVA

Los métodos disponibles para las clases de utilidad Integer, Arrays y Date están en esta guía. Recordar que la clase String y Math están explicadas en la guía anterior de Intro Java.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

8. Realizar una clase llamada Cadena que tenga como atributos una frase (de tipo de String) y su longitud. En el main se creará el objeto y se le pedirá al usuario que ingrese una frase que puede ser una palabra o varias palabras separadas por un espacio en blanco y a través de los métodos set, se guardará la frase y la longitud de manera automática según la longitud de la frase ingresada. Deberá además implementar los siguientes métodos:

- Método mostrarVocales(), deberá contabilizar la cantidad de vocales que tiene la frase ingresada.
- Método invertirFrase(), deberá invertir la frase ingresada y mostrarla por pantalla. Por ejemplo: Entrada: "casa blanca", Salida: "acnalb asac".
- Método vecesRepetido(String letra), recibirá un carácter ingresado por el usuario y contabilizar cuántas veces se repite el carácter en la frase, por ejemplo: Entrada: frase = "casa blanca". Salida: El carácter 'a' se repite 4 veces.
- Método compararLongitud(String frase), deberá comparar la longitud de la frase que compone la clase con otra nueva frase ingresada por el usuario.
- Método unirFrases(String frase), deberá unir la frase contenida en la clase Cadena con una nueva frase ingresada por el usuario y mostrar la frase resultante.
- Método reemplazar(String letra), deberá reemplazar todas las letras "a" que se encuentren en la frase, por algún otro carácter seleccionado por el usuario y mostrar la frase resultante.
- Método contiene(String letra), deberá comprobar si la frase contiene una letra que ingresa el usuario y devuelve verdadero si la contiene y falso si no.

Método Static y Clase Math

9. Realizar una clase llamada Matemática que tenga como atributos dos números reales con los cuales se realizarán diferentes operaciones matemáticas. La clase deberá tener un constructor vacío, parametrizado y get y set. En el main se creará el objeto y se usará el Math.random para generar los dos números y se guardarán en el objeto con sus respectivos set. Deberá además implementar los siguientes métodos:

- Método devolverMayor() para retornar cuál de los dos atributos tiene el mayor valor
- Método calcularPotencia() para calcular la potencia del mayor valor de la clase elevado al menor número. Previamente se deben redondear ambos valores.
- Método calculaRaiz(), para calcular la raíz cuadrada del menor de los dos valores. Antes de calcular la raíz cuadrada se debe obtener el valor absoluto del número.

Clase Arrays

10. Realizar un programa en Java donde se creen dos arreglos: el primero será un arreglo A de 50 números reales, y el segundo B, un arreglo de 20 números, también reales. El programa deberá inicializar el arreglo A con números aleatorios y mostrarlo por pantalla. Luego, el arreglo A se debe ordenar de menor a mayor y copiar los primeros 10 números ordenados al arreglo B de 20 elementos, y llenar los 10 últimos elementos con el valor 0.5. Mostrar los dos arreglos resultantes: el ordenado de 50 elementos y el combinado de 20.

Clase Date

11. Pongamos de lado un momento el concepto de POO, ahora vamos a trabajar solo con la clase Date. En este ejercicio deberemos instanciar en el main, una fecha usando la clase Date, para esto vamos a tener que crear 3 variables, dia, mes y anio que se le pedirán al usuario para poner el constructor del objeto Date. Una vez creada la fecha de tipo Date, deberemos mostrarla y mostrar cuantos años hay entre esa fecha y la fecha actual, que se puede conseguir instanciando un objeto Date con constructor vacío.

Ejemplo fecha: Date fecha = new Date(anio, mes, dia);

Ejemplo fecha actual: Date fechaActual = new Date();

Si necesiten acá tienen más información en clase Date: [Documentacion Oracle](#)

12. Implemente la clase Persona. Una persona tiene un nombre y una fecha de nacimiento (Tipo Date), constructor vacío, constructor parametrizado, get y set. Y los siguientes métodos:

Nota: encontraras un ejemplo descargable de un **Date** como atributo en Moodle.

- Agregar un método de creación del objeto que respete la siguiente firma: crearPersona(). Este método rellena el objeto mediante un Scanner y le pregunta al usuario el nombre y la fecha de nacimiento de la persona a crear, recordemos que la fecha de nacimiento debe guardarse en un Date y los guarda en el objeto.
- Escribir un método calcularEdad() a partir de la fecha de nacimiento ingresada. Tener en cuenta que para conocer la edad de la persona también se debe conocer la fecha actual.
- Agregar a la clase el método menorQue(int edad) que recibe como parámetro otra edad y retorna true en caso de que el receptor tenga menor edad que la persona que se recibe como parámetro, o false en caso contrario.
- Método mostrarPersona(): este método muestra la persona creada en el método anterior.

EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Desarrollar una clase Cancion con los siguientes atributos: titulo y autor. Se deberá definir además dos constructores: uno vacío que inicializa el titulo y el autor a cadenas vacías y otro que reciba como parámetros el título y el autor de la canción. Se deberán además definir los métodos getters y setters correspondientes.
2. Definir una clase llamada Puntos que contendrá las coordenadas de dos puntos, sus atributos serán, x1, y1, x2, y2, siendo cada x e y un punto. Generar un objeto puntos usando un método crearPuntos() que le pide al usuario los dos números y los ingresa en los atributos del objeto. Después, a través de otro método calcular y devolver la distancia que existe entre los dos puntos que existen en la clase Puntos. Para conocer como calcular la distancia entre dos puntos consulte el siguiente link:

<http://www.matematicatuya.com/GRAFICAecuaciones/S1a.html>

3. Vamos a realizar una clase llamada Raices, donde representaremos los valores de una ecuación de 2o grado. Tendremos los 3 coeficientes como atributos, llamémosles a, b y c. Hay que insertar estos 3 valores para construir el objeto a través de un método constructor. Luego, las operaciones que se podrán realizar son las siguientes:
 - Método getDiscriminante(): devuelve el valor del discriminante (double). El discriminante tiene la siguiente formula: $(b^2)-4*a*c$
 - Método tieneRaices(): devuelve un booleano indicando si tiene dos soluciones, para que esto ocurra, el discriminante debe ser mayor o igual que 0.
 - Método tieneRaiz(): devuelve un booleano indicando si tiene una única solución, para que esto ocurra, el discriminante debe ser igual que 0.
 - Método obtenerRaices(): llama a tieneRaices() y si devolvió true, imprime las 2 posibles soluciones.
 - Método obtenerRaiz(): llama a tieneRaiz() y si devolvió true imprime una única raíz. Es en el caso en que se tenga una única solución posible.
 - Método calcular(): esté método llamará tieneRaices() y a tieneRaíz(), y mostrará por pantalla las posibles soluciones que tiene nuestra ecuación con los métodos obtenerRaices() o obtenerRaiz(), según lo que devuelvan nuestros métodos y en caso de no existir solución, se mostrará un mensaje.

Nota: Formula ecuación 2o grado: $(-b \pm \sqrt{((b^2)-(4*a*c))})/(2*a)$ Solo varia el signo delante de -b

4. Dígito Verificador. Crear una clase NIF que se usará para mantener DNIs con su correspondiente letra (NIF). Los atributos serán el número de DNI (entero largo) y la letra (String o char) que le corresponde. Dispondrá de los siguientes métodos:

- Métodos getters y setters para el número de DNI y la letra.
- Método crearNif(): le pide al usuario el DNI y con ese DNI calcula la letra que le corresponderá. Una vez calculado, le asigna la letra que le corresponde según el resultado del cálculo.
- Método mostrar(): que nos permita mostrar el NIF (ocho dígitos, un guion y la letra en mayúscula; por ejemplo: 00395469-F).

La letra correspondiente al dígito verificador se calculará a través de un método que funciona de la siguiente manera: Para calcular la letra se toma el resto de dividir el número de DNI por 23 (el resultado debe ser un número entre 0 y 22). El método debe buscar en un array (vector) de caracteres la posición que corresponda al resto de la división para obtener la letra correspondiente. La tabla de caracteres es la siguiente:

POSICIÓN	LETRA
0	T
1	R
2	W
3	A
4	G
5	M
6	Y
7	F
8	P
9	D
10	X
11	B
12	N
13	J
14	Z
15	S
16	Q
17	V
18	H
19	L
20	C
21	K
22	E

5. Crea una clase en Java donde declares una variable de tipo array de Strings que contenga los doce meses del año, en minúsculas. A continuación, declara una variable mesSecreto de tipo String, y hazla igual a un elemento del array (por ejemplo, mesSecreto = mes[9]. El programa debe pedir al usuario que adivine el mes secreto. Si el usuario acierta mostrar un mensaje, y si no lo hace, pedir que vuelva a intentar adivinar el mes secreto. Un ejemplo de ejecución del programa podría ser este:

Adivine el mes secreto. Introduzca el nombre del mes en minúsculas: febrero

No ha acertado. Intente adivinarlo introduciendo otro mes: agosto

¡Ha acertado!

6. Juego Ahorcado: Crear una clase Ahorcado (como el juego), la cual deberá contener como atributos, un vector con la palabra a buscar, la cantidad de letras encontradas y la cantidad jugadas máximas que puede realizar el usuario. Definir los siguientes métodos con los parámetros que sean necesarios:

- Constructores, tanto el vacío como el parametrizado.
- Método crearJuego(): le pide la palabra al usuario y cantidad de jugadas máxima. Con la palabra del usuario, pone la longitud de la palabra, como la longitud del vector. Después ingresa la palabra en el vector, letra por letra, quedando cada letra de la palabra en un índice del vector. Y también, guarda en cantidad de jugadas máximas, el valor que ingresó el usuario y encontradas en 0.
- Método longitud(): muestra la longitud de la palabra que se debe encontrar. Nota: buscar como se usa el vector.length.
- Método buscar(letra): este método recibe una letra dada por el usuario y busca si la letra ingresada es parte de la palabra o no. También informará si la letra estaba o no.
- Método encontradas(letra): que reciba una letra ingresada por el usuario y muestre cuantas letras han sido encontradas y cuantas le faltan. Este método además deberá devolver true si la letra estaba y false si la letra no estaba, ya que, cada vez que se busque una letra que no esté, se le restará uno a sus oportunidades.
- Método intentos(): para mostrar cuantas oportunidades le queda al jugador.
- Método juego(): el método juego se encargará de llamar todos los métodos previamente mencionados e informará cuando el usuario descubra toda la palabra o se quede sin intentos. Este método se llamará en el main.

Nota: encontraras un ejemplo descargable de un **vector** como atributo en Moodle.

Un ejemplo de salida puede ser así:

Ingrese una letra:

a

Longitud de la palabra: 6

Mensaje: La letra pertenece a la palabra

Número de letras (encontradas, faltantes): (3,4)

Número de oportunidades restantes: 4

Ingrese una letra:

z

Longitud de la palabra: 6

Mensaje: La letra no pertenece a la palabra

Número de letras (encontradas, faltantes): (3,4)

Número de oportunidades restantes: 3

Ingrese una letra:

b

Longitud de la palabra: 6

Mensaje: La letra no pertenece a la palabra

Número de letras (encontradas, faltantes): (4,3)

Número de oportunidades restantes: 2

Ingrese una letra:

u

Longitud de la palabra: 6

Mensaje: La letra no pertenece a la palabra

Número de letras (encontradas, faltantes): (4,3)

Número de oportunidades restantes: 1

Ingrese una letra:

q

Longitud de la palabra: 6

Mensaje: La letra no pertenece a la palabra

Mensaje: Lo sentimos, no hay más oportunidades

Bibliografía

Información sacada de libros:

- Fundamentos de Programación de Luis Joyanes Aguilar
- Fundamentos de Programación Java de Jorge Martínez Ladrón de Guevara en conjunto con la Facultad de Informática (Universidad Complutense de Madrid).
- Introducción a la programación Java de John S. Dean y Raymond H. Dean

Información sacada de las páginas:

- <https://profile.es/blog/que-son-los-paradigmas-de-programacion/>
- <https://profile.es/blog/que-es-la-programacion-orientada-a-objetos/>

CURSO DE PROGRAMACIÓN FULL STACK

COLECCIONES

PARADIGMA ORIENTADO A OBJETOS



EGG

GUÍA DE COLECCIONES

COLECCIONES

Previo a esta guía, nosotros manejábamos nuestros objetos de uno en uno, no teníamos manera de manejar varios objetos a la vez, pero, para esto existen **las colecciones**.

Una **colección** representa un grupo de objetos. Estos objetos son conocidos como **elementos**. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. Las colecciones nos dan la opción de almacenar cualquier tipo de objeto, esto incluye los objetos de tipo de datos. Por lo que, para crear una colección de un tipo de dato, no podremos usar los datos primitivos, sino sus objetos. Por ejemplo: en vez de `int`, hay que utilizar `Integer`.

Tipos de datos	
Primitivos	Objetos
<code>int</code>	<code>Integer</code>
<code>double</code>	<code>Double</code>
<code>long</code>	<code>Long</code>
<code>char</code>	<code>Character</code>
<code>boolean</code>	<code>Boolean</code>
String ya es un objeto, por lo que no tiene tipo primitivo	

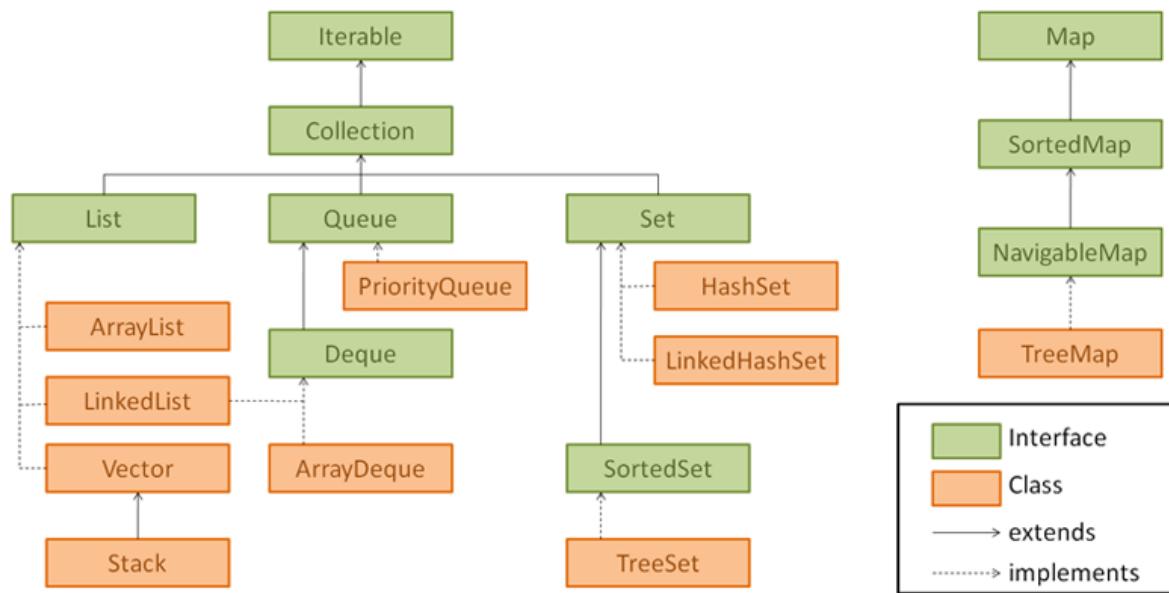
Dijimos que una colección es un grupo de objetos, pero obtener una colección vamos a utilizar unas clases propias de Java. Estas clases, que van a ser el almacén de los objetos, nos proveen con una serie de **métodos** comunes, para trabajar con los elementos de la colección, como, por ejemplo: **agregar** y **eliminar** elementos u obtener el tamaño de la colección, etc.

Las colecciones son una especie de arreglos de tamaño dinámico. Estas son parte del **Java Collections Framework** dentro del paquete `java.util`. El Collections Framework es una arquitectura compuesta de interfaces y clases. Dentro de este framework están las colecciones que vamos a trabajar, las listas, conjuntos y mapas. Nota: el concepto de interfaces lo vamos a explicar más adelante.

¿QUÉ ES UN FRAMEWORK?

Un framework es un marco de trabajo el cual contiene un conjunto estandarizado de conceptos, prácticas y criterios para hacer frente a un tipo de problemática particular y resolver nuevos problemas de índole similar.

Las clases del Java Collections Framework son las siguientes:



LISTAS

Las listas modelan una colección de objetos ordenados por posición. La principal diferencia entre las listas y los arreglos tradicionales, es que la lista crece de manera **dinámica** a medida que se van agregando objetos. No necesitamos saber de antemano la cantidad de elementos con la que vamos a trabajar. El framework trae varias implementaciones de distintos tipos de listas tales como **ArrayList**, **LinkedList**.

- **ArrayList**: se implementa como un arreglo de tamaño variable. A medida que se agregan más elementos, su tamaño aumenta dinámicamente. Es el tipo más común.

Ejemplo de un ArrayList de numeros:

```
ArrayList<Integer> numeros = new ArrayList();
```

- **LinkedList**: se implementa como una [lista de doble enlace](#). Su rendimiento al agregar y quitar es mejor que ArrayList, pero peor en los métodos set y get.

Ejemplo de una LinkedList de numeros:

```
LinkedList<Integer> numeros = new LinkedList();
```

CONJUNTOS

Los **conjuntos** o en inglés **Set** modelan una colección de objetos de una misma clase donde cada elemento aparece **solo una vez**, no puede tener duplicados, a diferencia de una lista donde los elementos podían repetirse. El framework trae varias implementaciones de distintos tipos de conjuntos:

- **HashSet**, se implementa utilizando una [tabla hash](#) para darle un **valor único** a cada elemento y de esa manera evitar los duplicados. Los métodos de agregar y eliminar tienen una complejidad de tiempo constante por lo que tienen mejor rendimiento que el TreeSet.

Ejemplo de un HashSet de cadenas:

```
HashSet<String> nombres = new HashSet();
```

- **TreeSet** se implementa utilizando una [estructura de árbol](#) (árbol rojo-negro en el libro de algoritmos). La gran diferencia entre el HashSet y el TreeSet, es que el TreeSet mantiene todos sus elementos de manera **ordenada**(forma ascendente), pero los métodos de agregar, eliminar son más lentos que el HashSet ya que cada vez que le entra un elemento debe posicionarlo para que quede ordenado.

Ejemplo de un TreeSet de numeros:

```
TreeSet<Integer> numeros = new TreeSet();
```

- **LinkedHashSet** está entre HashSet y TreeSet. Se implementa como una tabla hash con una lista vinculada que se ejecuta a través de ella, por lo que proporciona el orden de inserción.

Ejemplo de un LinkedHashSet de cadenas:

```
LinkedHashSet<String> frases = new LinkedHashSet();
```

MAPAS

Los mapas modelan un objeto a través de **una llave y un valor**. Esto significa que cada valor de nuestro mapa, va a tener una **llave única** para representar dicho **valor**. Las llaves de nuestro mapa no pueden **repetirse**, pero los valores sí. Un ejemplo sería una persona que tiene su DNI/RUT (llave única) y como valor puede ser su nombre completo, puede haber dos personas con el mismo nombre, pero nunca con el mismo DNI/RUT.

Los mapas al tener dos datos, también vamos a tener que especificar el tipo de dato tanto de la llave y de el valor, pueden ser de tipos de datos distintos. A la hora de crear un mapa tenemos que pensar que el primer tipo dato será el de la llave y el segundo el valor.

Son una de las estructuras de datos importantes del Framework de Collections. Las implementaciones de mapas son HashMap, TreeMap, LinkedHashMap y HashTable.

- **HashMap** es un mapa implementado a través de una tabla hash, las llaves se almacenan utilizando un algoritmo de hash solo para las llaves y evitar que se repitan.

Ejemplo de un HashMap de personas:

```
HashMap<Llave,Valor> personas = new HashMap();
```

```
HashMap<Integer,String> personas = new HashMap();
```

- TreeMap es un mapa que ordena los elementos de manera ascendente a través de las llaves.

Ejemplo de un TreeMap de personas:

```
TreeMap<Integer, String> personas = new TreeMap();
```

- LinkedHashMap es un HashMap que conserva el orden de inserción.

Ejemplo de un LinkedHashMap de personas:

```
LinkedHashMap<Integer, String> personas = new LinkedHashMap();
```

AÑADIR UN ELEMENTO A UNA COLECCIÓN

Las colecciones constan con funciones para realizar distintas operaciones, en este caso si queremos añadir un elemento a las listas o conjuntos vamos a tener que utilizar la función add(T), pero para los mapas vamos a utilizar la función put(llave,valor).

Listas:

```
ArrayList<Integer> numeros = new ArrayList(); //Lista de tipo Integer
int num = 20;
numeros.add(num); //Agregamos el numero 20 a la lista, en la posición 0
```

Conjuntos:

```
HashSet<Integer> numeros = new HashSet();
int num = 20;
numeros.add(20);
```

Mapas:

```
HashMap<Integer, String> alumnos = new HashMap();
int dni = 34576189;
String nombreAlumno = "Pepe";
alumnos.put(dni, nombreAlumno); //Agregamos la llave y el valor
```

ELIMINAR UN ELEMENTO DE UNA COLECCIÓN

Cada colección consta con métodos para poder remover elementos del tipo que sea la colección.

Listas:

Las listas constan de dos métodos:

remove(int índice): Este método remueve un elemento de un índice específico. Esto mueve los elementos, de manera que no queden índices sin elementos.

remove(elemento): Este método remueve la primer aparición de un elemento a borrar en una lista

Índice:

```
ArrayList<Integer> numeros = new ArrayList();
int num = 20;
numeros.add(num); // Este numero se encuentra en el índice 0
numeros.remove(0); // Eliminamos el numero que esté en el índice 0
```

Elemento:

```
ArrayList<Integer> numeros = new ArrayList();
int num = 30;
numeros.add(num);
numeros.remove(30); // Eliminamos el numero 30 o el primer 30 que encuentre
```

Conjuntos:

Ya que los conjuntos no constan de índices, solo se puede borrar por elemento.

remove(elemento): Este método remueve la primera aparición de un elemento a borrar en un conjunto

```
HashSet<Integer> numeros = new HashSet();
int num = 50;
numeros.add(50);
numeros.remove(50); //Eliminamos el numero 50
```

Mapas:

La parte más importante de los elementos de un mapa es la llave del elemento, que es la que hace el elemento único, por eso en los mapas solo podemos remover un elemento por su llave.

remove(llave): Este método remueve la primera aparición de la llave de un elemento a borrar en un mapa.

```
HashMap<Integer, String> estudiantes = new HashMap();
estudiantes.remove(123); //Borramos la llave 123
```

RECORRER UNA COLECCIÓN

Si quisieramos mostrar todos los elementos que le hemos agregado y que componen a nuestra colección vamos a tener que recorrerla.

Para recorrer una colección, vamos a tener que utilizar el bucle **forEach**. El bucle comienza con la palabra clave **for** al igual que un bucle *for normal*. Pero, en lugar de declarar e inicializar una variable contador del bucle, declara una variable vacía, que es del **mismo tipo que la colección**, seguido de dos puntos y seguido del nombre de la colección. La variable recibe en cada iteración un elemento de la colección, de esa manera si nosotros mostramos esa variable, podemos mostrar todos los elementos de nuestra colección.

Para recorrer mapas vamos a tener que usar el objeto Map.Entry en el for each. A través de el entry vamos a traer los valores y las llaves, si no, podemos tener un for each para cada parte de nuestro mapa sin utilizar el objeto Map.Entry.

Para saber más sobre la clase Map y el objeto Entry: [Map.Entry](#)

For Each:

```
for (Tipo de dato variableVacia : Colección){  
}
```

Listas:

```
ArrayList<String> lista = new ArrayList();  
for (String cadena : lista) {  
    System.out.println(cadena);  
    // mostramos los elementos a través de la variable  
}
```

Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();  
for (Integer numero : numerosSet) {  
    System.out.println(numero);  
    // mostramos los elementos a través de la variable  
}
```

Mapas:

```
HashMap<Integer, String> alumnos = new HashMap();  
// Recorrer las dos partes del mapa  
for (Map.Entry<Integer, String> entry : alumnos.entrySet()) {  
    System.out.println("documento=" + entry.getKey() +  
        ", nombre=" + entry.getValue());  
    // entry.getKey trae la llave y entry.getValue trae los valores del mapa
```

Sin Map.Entry:

```
// mostrar solo las llaves  
for (Integer dni : alumnos.keySet()) {  
    System.out.println("Documento: " + dni);  
}  
  
// mostrar solo los valores  
for (String nombres : alumnos.values()) {  
    System.out.println("Nombre: " + nombres);  
}
```

ITERATOR

El *Iterator* es una interfaz que pertenece al **framework de colecciones**. Este, nos permite recorrer, acceder a la información y eliminar algún elemento de una colección. Gracias al *Iterator* podemos eliminar un elemento, mientras recorremos la colección. Ya que, cuando queremos eliminar algún elemento mientras recorremos una colección con el bucle `ForEach`, nos va a tirar un error.

Como el *Iterator* es parte del framework de colecciones, todas las colecciones vienen con el método `iterator()`, este, devuelve las instrucciones para iterar sobre esa colección. Este método `iterator()`, devuelve la colección, lo recibe el objeto *Iterator* usando el objeto `Iterator`, podemos iterar sobre nuestra colección.

Para poder usar el *Iterator* es importante crear el objeto de tipo *Iterator*, con el mismo tipo de dato que nuestra colección.

El *Iterator* contiene tres métodos muy útiles para lograr esto:

1. `boolean hasNext()`: Retorna verdadero si al *Iterator* le quedan elementos por iterar
2. `Object next()`: Devuelve el siguiente elemento en la colección, mientras el método `hasNext()` retorne true. Este método es el que nos sirve para mostrar el elemento,
3. `void remove()`: Elimina el elemento actual de la colección.

Ejemplo Listas:

```
ArrayList<String> lista = new ArrayList();
lista.add("A");
lista.add("B");

// Iterator para recorrer la lista
Iterator iterator = lista.iterator(); // Devolvemos el iterator
System.out.println("Elementos de la lista: ");

// Usamos un while para recorrer la lista, siempre que el hasNext()
// devuelva true.
while (iterator.hasNext())

// Mostramos los elementos con el iterator.next()
System.out.print(iterator.next() + " ");
System.out.println();

}
```

ELIMINAR UN ELEMENTO DE UNA COLECCIÓN CON ITERATOR

Como pudimos ver más arriba para eliminar un elemento de una colección vamos a tener que utilizar la función `remove()` del *Iterator*. Esto se aplica para el resto de nuestras colecciones. Los mapas son los únicos que no podemos eliminar mientras las iteramos.

Listas:

```
ArrayList<String> palabras = new ArrayList();
Iterator<String> it = palabras.iterator();
while (it.hasNext()) {
    if (it.next().equals("Hola")) { // Borramos si está la palabra Hola
        it.remove();
    }
}
```

Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();
Iterator<Integer> it = numerosSet.iterator();
while (it.hasNext()) {
    if (it.next() == 3) { // Borramos si está el numero 3
        it.remove();
    }
}
```

ORDENAR UNA COLECCIÓN

Los elementos, que vamos agregando a nuestra colección se van a mostrar según se fueron agregando y nosotros capaz, necesitemos mostrar o tener todos los elementos ordenados.

Para ordenar una colección, vamos a tener que utilizar la función `Collections.sort(coleccion)`. La función, que es parte de la clase `Collections`, recibe la colección y la ordena para después poder mostrarla ordenada de manera ascendente.

Algunas colecciones, como los conjuntos o los mapas no pueden utilizar el `sort()`. Ya que por ejemplo los `HashSet`, manejan valores Hash y el `sort()` no sabe ordenar por hash, si no por elementos. Por otro lado, los mapas al tener dos datos, el `sort()` no sabe por cual de esos datos ordenar.

Entonces, para ordenar los conjuntos, deberemos convertirlos a listas, para poder ordenar esa lista por sus elementos. Y a la hora de ordenar un mapa como tenemos dos datos para ordenar, vamos a convertir el `HashMap` a un `TreeMap`.

Nota: recordemos que los `TreeSet` y `TreeMap` se ordenan por si mismos.

Listas:

```
ArrayList<Integer> numeros = new ArrayList();
Collections.sort(numeros);
```

Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();
// Se convierte el HashSet a Lista.
ArrayList<Integer> numerosLista = new ArrayList(numerosSet);
Collections.sort(numerosLista);
```

Mapas:

```
HashMap<Integer, String> alumnos = new HashMap();
// Se convierte el HashMap a TreeMap
TreeMap<Integer, String> alumnosTree = new TreeMap();
```

COLECCIONES CON OBJETOS

De la misma manera que podemos crear colecciones con los tipos de datos de Java, podemos crear colecciones de algún objeto, de una clase creada por nosotros, previamente. Esto, nos servirá para manejar varios objetos al mismo tiempo y acceder a ellos de una manera más sencilla. Por ejemplo, tener una lista de alumnos, siendo cada Alumno un objeto con sus atributos.

AÑADIR UN OBJETO A UNA COLECCIÓN

Para añadir un objeto a una colección tenemos que primero crear el objeto que queremos trabajar y después crear una colección donde su tipo de dato sea dicho objeto.

La manera de agregar los objetos a la colección es muy parecida a lo que habíamos visto previamente.

Las colecciones Tree, ya sean TreeSet o TreeMap, son las únicas que no vamos a poder agregar como siempre. Ya que, los Tree, siendo colecciones que se ordenan a sí mismas, debemos informarle al Tree como va a ordenarse. Ahora, pensemos que un objeto posee más de un dato(atributos), entonces, el Tree, no sabe por qué atributo debe ordenarse.

Para solucionar esto, vamos a necesitar un **Comparator**, este, le dará la pauta de como ordenarse y sobre que atributo. El Comparator está explicado más abajo en la guía y muestra como agregárselo a los Tree.

Listas:

```
ArrayList<Libro> libros = new ArrayList();
Libro libro = new Libro();
libros.add(libro);
```

Conjuntos:

```
HashSet<Perro> perros = new HashSet();
Perro perro = new Perro();
perros.add(perro);
```

Mapas:

```
HashMap<Integer, Alumno> alumnos = new HashMap();
int dni = 34576189;
Alumno alumno = new Alumno("Pepe", "Honguito");
alumnos.put(dni, alumno);
```

RECORRER UNA COLECCIÓN CON OBJETOS

Para recorrer una colección donde su tipo de dato sea un objeto creado por nosotros, vamos a seguir utilizando los métodos que conocemos, el for each o el iterator. Pero a la hora de mostrar el objeto con un System.out.println, no nos va a mostrar sus atributos. Sino que, nos va a mostrar el nombre de la clase, el nombre del objeto, una arroba y un código hash para representar los valores del objeto.

Ejemplo:

```
ArrayList<Libro> libros = new ArrayList();
Libro libro = new Libro();
libros.add(libro);
for (Libro libro : libros) {
System.out.println(libro);
}
```

Cuando queremos mostrar el libro, que está siendo recorrido por el for each, nos mostraría algo así: `Libreria.Libro@14ae5a5`

Para solucionar este problema, vamos a tener que sobrescribir(Override), un método de la clase String dentro de la clase de nuestro objeto. Este método va a transformar, el nombre de la clase, el nombre del objeto y el hash, en una cadena legible para imprimir.

Para poder usar este método vamos a ir a nuestra clase, ahí hacemos click derecho, insert code y le damos a `toString()`. Eso nos va a generar un método `toString()` con los atributos de nuestro objeto y que retorna una cadena para mostrar el objeto.

Ejemplo:

```
@Override
public String toString() {
    return "Libro{" + "titulo=" + titulo + '}';
}
```

Este método se va a llamar solo, sin necesidad que lo llamemos nosotros, siempre que queramos mostrar nuestro objeto en un `System.out.println`. Y mostrará la línea que se ve en el return.

Ejemplo:

```
ArrayList<Libro> libros = new ArrayList();
Libro libro = new Libro();
libro.setTitulo("La Odisea");
libros.add(libro);
for (Libro libro : libros) {
System.out.println(libro);
}
```

Output: Libro{titulo= La Odisea}

COMPARATOR

A la hora de querer ordenar una colección de objetos en Java, no podemos utilizar la función sort, ya que el sort se utiliza para ordenar colecciones con elementos uniformes. Pero los objetos pueden tener dentro distintos tipos de datos (atributos). Entonces, nuestra función sort no sabe porqué tipo de dato o atributo ordenar. Para esto, utilizamos la interfaz Comparator con su función compare() en nuestra clase entidad.

Supongamos que tenemos una clase Perro, que tiene como atributos el nombre del perro y la edad. Nosotros queremos ordenar los perros por edad, deberemos crear el método compare de la clase Comparator en la clase Perro.

Ejemplo:

```
public class Perro {
    public static Comparator<Perro> compararEdad = new Comparator<Perro>() {
        @Override
        public int compare(Perro p1, Perro p2) {
            return p2.getEdad().compareTo(p1.getEdad());
        }
    };
}
```

Explicación del método:

- El método crea un objeto estático de la interfaz Comparator. Este nos va a permitir utilizar a través de un sobrescribir (Override) el método compare, el mismo nos deja comparar dos objetos para poder ordenarlos. Este objeto se crea static para poder llamar al método solo llamando a la clase, sin tener que crear otro objeto Comparator, en este caso la clase Perro.
- Dentro de la creación de objeto se crea un método de la clase Comparator llamado compare, arriba del método se puede ver la palabra Override. Override, se usa cuando desde una subclase (Perro), queremos utilizar un método de otra clase (Comparator) en nuestra subclase.

- El método recibe dos objetos de la clase Perro y retorna una comparación entre los dos usando los get para traer el atributo que queríamos comparar y usa la función compareTo, que devuelve 0 si la edad es la misma, 1 si la primera edad es mayor a la segunda y -1 si la primera edad es menor a la segunda.
- Si quisiéramos cambiar el atributo que usa para ordenar, pondríamos otro atributo en el get del return.

USO DEL METODO COMPARATOR

Como el comparator se va a usar para ordenar nuestras colecciones, se va a poner en el llamado de la función Collections.sort() y se va a poner en la inicialización de cualquier tipo de Tree.

Listas:

```
ArrayList<Perro> perros = new ArrayList();
//Se llama al metodo estatico a traves de la clase y se pone la lista a
//ordenar.
perros.sort(Perro.compararEdad);
```

Conjuntos:

```
HashSet<Perro> perrosSet = new HashSet();
ArrayList<Perro> perrosLista = new ArrayList(perrosSet);
perrosLista.sort(Perro.compararEdad);
```

Crear un TreeSet

En los TreeSet necesitamos crearlos con el comparator porque como el TreeSet se ordena solo, necesitamos decirle al TreeSet, bajo que atributo se va a ordenar, por eso le pasamos el comparator en el constructor.

```
TreeSet<Perro> perros = new TreeSet(Perro.compararEdad);
Perro perro = new Perro();
perros.add(perro);
```

Mapas:

```
HashMap<Integer, Alumno> alumnos = new HashMap();
//Se usa una función de los mapas para traer todos valores.
ArrayList<Alumno> nombres = new ArrayList(map.values());
nombres.sort(Alumno.compararDni);
```

COLECCIONES EN FUNCIONES

A la hora de querer pasar una colección a una función, deberemos recordar que Java es fuertemente tipado, por lo que deberemos poner el tipo de dato de la colección y que tipo de colección es cuando la pongamos como argumento.

Listas:

```
Public void llenarLista(ArrayList<Integer> numeros){  
    numeros.add(20)  
}  
  
Main  
  
ArrayList<Integer> numeros = new ArrayList();  
llenarLista(numeros); // Le pasamos la lista a la función
```

Conjuntos:

```
Public void llenarHashSet(HashSet<String> palabras){  
    palabras.add("Hola")  
}  
  
Main  
  
HashSet<String> palabras = new HashSet();  
llenarHashSet(palabras); // Le pasamos el conjunto a la función
```

Mapas:

```
Public void llenarMapa(HashMap<Integer, String> alumnos){  
    alumnos.put(1, "Pepe");  
}  
  
Main  
  
HashMap<Integer, String> alumnos = new HashMap();  
llenarMapa(alumnos); // Le pasamos el conjunto a la función
```

DEVOLVER UNA COLECCIÓN EN FUNCIONES

Para devolver una colección en una función, tenemos que hacer que el tipo de dato de nuestra función sea la colección que queremos devolver, teniendo también el tipo de dato que va a manejar dicha colección.

Listas:

```
public ArrayList<Integer> llenarLista(){  
    ArrayList<Integer> numeros = new ArrayList();  
    numeros.add(20);  
    return numeros; // Devolvemos la lista llena.  
}
```

Conjuntos:

```
public HashSet<String> llenarHashSet(){  
    HashSet<String> palabras = new HashSet();  
    palabras.add("Hola")  
    return palabras }
```

Mapas:

```
public HashMap<Integer, String> llenarMapa(){  
    HashMap<Integer, String> alumnos = new HashMap();  
    alumnos.put(1, "Pepe");  
    return alumnos;  
}
```

PROYECTO CON EJEMPLOS DE COLECCIONES

El proyecto contiene un paquete con un main para cada tipo de colección y un paquete más que muestra como usar una lista con un objeto. Además, contiene algunos métodos que no están explicados en la teoría.

El ejemplo lo podrán encontrar en Moodle para descargar.

CLASE COLLECTIONS

La clase **Collections** es parte del framework de colecciones y también es parte del paquete `java.util`. Esta clase nos provee de métodos que reciben una colección y realizan alguna operación o devuelven una colección, según el método. Vamos a mostrar algunos de los métodos pero, hay muchos más.

Método	Descripción.
<code>fill(List<T> lista, Objeto objeto)</code>	Este método reemplaza todos los elementos de la lista con un elemento específico.
<code>frequency(Collection<T> colección, Objeto objeto)</code>	Este método retorna la cantidad de veces que se encuentra un elemento específico en una colección.
<code>replaceAll(List<T> lista, T valorViejo, T valorNuevo)</code>	Este método reemplaza todas las apariciones de un elemento específico en una lista, con otro valor.
<code>reverse(List<T> lista)</code>	Este método invierte el orden de los elementos de una lista.
<code>reverseOrder()</code>	Este método retorna un comparador que invierte el orden de los elementos de una colección.
<code>shuffle(List<T> lista)</code>	Este método modifica la posición de los elementos de una lista de manera aleatoria.
<code>sort(List<T> lista)</code>	Este método ordena los elementos de una lista de manera ascendente.

METODOS EXTRAS COLECCIONES

En la guía se muestran las acciones más realizadas con colecciones, con la ayuda de sus métodos, pero también existen otros métodos en las colecciones para realizar otras acciones. **Nota:** los métodos de los List y los Set, son los mismos, quitando el get y el set.

Listas y Conjuntos:

Método	Descripción.
<code>size()</code>	Este método retorna el tamaño de una lista / conjunto.

<code>clear()</code>	Este método se usa para remover todos los elementos de una lista / conjunto.
<code>get(int índice)</code>	Este método retorna un elemento de la lista según un índice de la lista.
<code>set(int índice, elemento)</code>	Este método guarda un elemento en la lista en un índice específico.
<code>isEmpty()</code>	Este método retorna verdadero si la lista / conjunto está vacío y falso si no lo está.
<code>contains(elemento)</code>	Este método recibe un elemento dado por el usuario y revisa si el elemento se encuentra en la lista o no. Si el elemento se encuentra retorna verdadero y si no falso.

Mapas:

Método	Descripción.
<code>clear()</code>	Este método se usa para remover todos los elementos de un mapa.
<code>containsKey(Llave)</code>	Este método recibe una llave dada por el usuario y revisa si la llave se encuentra en la lista o no. Si la llave se encuentra retorna verdadero y si no falso.
<code>containsValue(Valor)</code>	Este método recibe un valor dado por el usuario y revisa si el valor se encuentra en el mapa o no. Si el elemento se encuentra retorna verdadero y si no falso.
<code>get(Llave)</code>	Este método retorna un elemento del mapa según una llave dentro del mapa.
<code>isEmpty()</code>	Este método retorna verdadero si el mapa está vacío y falso si no lo está.
<code>size()</code>	Este método retorna el tamaño de un mapa.

<code>values()</code>	Este método crea una colección según los valores del mapa. Ósea, que retorna una lista, por ejemplo, con todos los valores del mapa.
<code>clear()</code>	Este método se usa para remover todos los elementos de un mapa.

EJERCICIOS DE APRENDIZAJE

En este módulo vamos a continuar modelando los objetos con el lenguaje de programación Java, pero ahora vamos a utilizar las colecciones para poder manejarlas de manera más sencilla y ordenada.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Diseñar un programa que lea y guarde razas de perros en un ArrayList de tipo String. El programa pedirá una raza de perro en un bucle, el mismo se guardará en la lista y después se le preguntará al usuario si quiere guardar otro perro o si quiere salir. Si decide salir, se mostrará todos los perros guardados en el ArrayList.
2. Continuando el ejercicio anterior, después de mostrar los perros, al usuario se le pedirá un perro y se recorrerá la lista con un Iterator, se buscará el perro en la lista. Si el perro está en la lista, se eliminará el perro que ingresó el usuario y se mostrará la lista ordenada. Si el perro no se encuentra en la lista, se le informará al usuario y se mostrará la lista ordenada.
3. Crear una clase llamada Alumno que mantenga información sobre las notas de distintos alumnos. La clase Alumno tendrá como atributos, su nombre y una lista de tipo Integer con sus 3 notas.

En el main deberemos tener un bucle que crea un objeto Alumno. Se pide toda la información al usuario y ese Alumno se guarda en una lista de tipo Alumno y se le pregunta al usuario si quiere crear otro Alumno o no.

Después de ese bucle tendremos el siguiente método en la clase Alumno:

Método notaFinal(): El usuario ingresa el nombre del alumno que quiere calcular su nota final y se lo busca en la lista de Alumnos. Si está en la lista, se llama al método. Dentro del método se usará la lista notas para calcular el promedio final de alumno. Siendo este promedio final, devuelto por el método y mostrado en el main.

Nota: encontrarán en Moodle un ejemplo de una Colección como Atributo.

4. Un cine necesita implementar un sistema en el que se puedan cargar películas. Para esto, tendremos una clase Película con el título, director y duración de la película (en horas). Implemente las clases y métodos necesarios para esta situación, teniendo en cuenta lo que se pide a continuación:

En el main deberemos tener un bucle que crea un objeto Pelicula pidiéndole al usuario todos sus datos y guardándolos en el objeto Pelicula.

Después, esa Pelicula se guarda una lista de Peliculas y se le pregunta al usuario si quiere crear otra Pelicula o no.

Después de ese bucle realizaremos las siguientes acciones:

- Mostrar en pantalla todas las películas.
- Mostrar en pantalla todas las películas con una duración mayor a 1 hora.
- Ordenar las películas de acuerdo a su duración (de mayor a menor) y mostrarlo en pantalla.
- Ordenar las películas de acuerdo a su duración (de menor a mayor) y mostrarlo en pantalla.
- Ordenar las películas por titulo, alfabéticamente y mostrarlo en pantalla.
- Ordenar las películas por director, alfabéticamente y mostrarlo en pantalla.

5. Se requiere un programa que lea y guarde países, y para evitar que se ingresen repetidos usaremos un conjunto. El programa pedirá un país en un bucle, se guardará el país en el conjunto y después se le preguntará al usuario si quiere guardar otro país o si quiere salir, si decide salir, se mostrará todos los países guardados en el conjunto.

Después deberemos mostrar el conjunto ordenado alfabéticamente para esto recordar como se ordena un conjunto.

Y por ultimo, al usuario se le pedirá un país y se recorrerá el conjunto con un Iterator, se buscará el país en el conjunto y si está en el conjunto se eliminará el país que ingresó el usuario y se mostrará el conjunto. Si el país no se encuentra en el conjunto se le informará al usuario.

6. Se necesita una aplicación para una tienda en la cual queremos almacenar los distintos productos que venderemos y el precio que tendrán. Además, se necesita que la aplicación cuente con las funciones básicas.

Estas las realizaremos en el main. Como, introducir un elemento, modificar su precio, eliminar un producto y mostrar los productos que tenemos con su precio (Utilizar Hashmap). El HashMap tendrá de llave el nombre del producto y de valor el precio. Realizar un menú para lograr todas las acciones previamente mencionadas.

EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Diseñar un programa que lea una serie de valores numéricos enteros desde el teclado y los guarde en un ArrayList de tipo Integer. La lectura de números termina cuando se introduzca el valor -99. Este valor no se guarda en el ArrayList. A continuación, el programa mostrará por pantalla el número de valores que se han leído, su suma y su media (promedio).

2. Crear una clase llamada CantanteFamoso. Esta clase guardará cantantes famosos y tendrá como atributos el nombre y discoConMasVentas.

Se debe, además, en el main, crear una lista de tipo CantanteFamoso y agregar 5 objetos de tipo CantanteFamoso a la lista. Luego, se debe mostrar los nombres de cada cantante y su disco con más ventas por pantalla.

Una vez agregado los 5, en otro bucle, crear un menú que le de la opción al usuario de agregar un cantante más, mostrar todos los cantantes, eliminar un cantante que el usuario elija o de salir del programa. Al final se deberá mostrar la lista con todos los cambios.

3. Implemente un programa para una Librería haciendo uso de un HashSet para evitar datos repetidos. Para ello, se debe crear una clase llamada Libro que guarde la información de cada uno de los libros de una Biblioteca. La clase Libro debe guardar el título del libro, autor, número de ejemplares y número de ejemplares prestados. También se creará en el main un HashSet de tipo Libro que guardará todos los libros creados.

En el main tendremos un bucle que crea un objeto Libro pidiéndole al usuario todos sus datos y los seteandolos en el Libro. Despues, ese Libro se guarda en el conjunto y se le pregunta al usuario si quiere crear otro Libro o no.

La clase Librería contendrá además los siguientes métodos:

- Constructor por defecto.
- Constructor con parámetros.
- Métodos Setters/getters

- Método prestamo(): El usuario ingresa el título del libro que quiere prestar y se lo busca en el conjunto. Si está en el conjunto, se llama con ese objeto Libro al método. El método se incrementa de a uno, como un carrito de compras, el atributo ejemplares prestados, del libro que ingresó el usuario. Esto sucederá cada vez que se realice un préstamo del libro. No se podrán prestar libros de los que no queden ejemplares disponibles para prestar. Devuelve true si se ha podido realizar la operación y false en caso contrario.
- Método devolucion(): El usuario ingresa el título del libro que quiere devolver y se lo busca en el conjunto. Si está en el conjunto, se llama con ese objeto al método. El método decrementa de a uno, como un carrito de compras, el atributo ejemplares prestados, del libro seleccionado por el usuario. Esto sucederá cada vez que se produzca la devolución de un libro. No se podrán devolver libros donde que no tengan ejemplares prestados. Devuelve true si se ha podido realizar la operación y false en caso contrario.
- Método toString para mostrar los datos de los libros.

4. Almacena en un HashMap los códigos postales de 10 ciudades a elección de esta página: <https://mapanet.eu/index.htm>. Nota: Poner el código postal sin la letra, solo el numero.

- Pedirle al usuario que ingrese 10 códigos postales y sus ciudades.
- Muestra por pantalla los datos introducidos
- Pide un código postal y muestra la ciudad asociada si existe sino avisa al usuario.
- Muestra por pantalla los datos
- Agregar una ciudad con su código postal correspondiente más al HashMap.
- Elimina 3 ciudades existentes dentro del HashMap, que pida el usuario.
- Muestra por pantalla los datos

CURSO DE PROGRAMACIÓN FULL STACK

RELACIONES ENTRE CLASES

PARADIGMA ORIENTADO A OBJETOS



EGG

GUÍA RELACIONES ENTRE CLASES

RELACIONES ENTRE CLASES

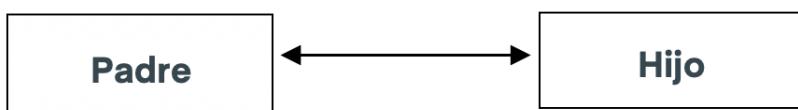
Una relación es una conexión semántica entre clases. Permite que una clase conozca sobre los atributos, operaciones y relaciones de otras clases. Las clases no actúan aisladas entre sí, al contrario las clases están relacionadas unas con otras. Una clase puede ser un tipo de otra clase —generalización— o bien puede contener objetos de otra clase de varias formas posibles, dependiendo de la fortaleza de la relación entre las dos clases.

En la programación orientada a objetos, un objeto se comunica con otro objeto para utilizar la funcionalidad y los servicios proporcionados por ese objeto. Por ejemplo: un objeto curso tiene varios objetos alumnos y un objeto profesor. Esto significa que, gracias a la relación entre objetos, el objeto curso puede tener toda la información que necesita. Además, nos ayuda para la reutilización de código, ya que podemos usar todo lo de la clase que ya habíamos escrito,

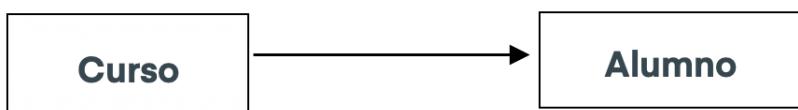
La relación entre dos clases separadas se establece a través de sus Objetos. Es decir, las clases se conectan juntas conceptualmente. Las relaciones entre clases realmente significan que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo.

La relación más simple es la asociación, esta relación es entre dos objetos como habíamos dicho previamente. En las relaciones de asociación se puede establecer una relación **bidireccional**, que los objetos que están al extremo de una relación pueden “conocerse” entre sí, o una relación **unidireccional** que solamente uno de ellos “conoce” a otro.

Bidireccional:



Unidireccional:



Dentro de la asociación simple existe la **composición** y la **agregación**, que son las dos formas de relaciones entre clases.

AGREGACIÓN

Representa un tipo de relación muy particular, en la que una clase es instanciada por otro objeto y clase. La agregación se podría definir como el momento en que dos objetos se unen para trabajar juntos y así, alcanzar una meta. Un punto a tomar muy en cuenta es que ambos objetos son independientes entre sí.

Para validar la agregación, la frase "**Usa un**" debe tener sentido, por ejemplo: El programador usa una computadora. El objeto computadora va a existir más allá de que exista o no el objeto programador.

En agregación, ambos objetos pueden sobrevivir individualmente, lo que significa que al borrar un objeto no afectará a la otra entidad.



COMPOSICIÓN

La composición es una relación más fuerte que la agregación, es una "relación de vida", es decir, que el tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye. La composición es un tipo de relación dependiente en dónde un objeto más complejo es conformado por objetos más pequeños. En esta situación, la frase "**Tiene un**", debe tener sentido, por ejemplo: el cliente tiene una cuenta bancaria. Esta relación es una composición, debido a que al eliminar el cliente la cuenta bancaria no tiene sentido, y también se debe eliminar, es decir, la cuenta existe sólo mientras exista el cliente.



RELACIONES EN CÓDIGO

Recordemos que las relaciones entre clases significan que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo. Pero a la hora de poner un atributo en una clase, debemos ver el tipo de relación de esas clases.

En las relaciones, tanto composición como agregación, las relaciones pueden ser de **uno a uno**, de **cero a uno**, de **uno a muchos** y de **cero a muchos**. El tipo de relación se ve representada a la hora de poner el objeto como forma de atributo en la clase que recibe la relación. Por ahora vamos a trabajar solo con **uno a uno** y **uno a muchos** porque son las más comunes.

UNO A UNO

Por cada objeto tenemos **una** relación con **un solo** objeto. Ejemplo: para un curso tengo un profesor. En código se representa con un atributo que sea un objeto.

```

public class Profesor {
    private String nombre;
    private String apellido;
}

public class Curso {
    private Profesor profesor;

    public Profesor getProfesor() {
        return profesor;
    }

    public void setProfesor(Profesor profesor) {
        this.profesor = profesor;
    }
}

```

En este ejemplo en el Main vamos a tener que crear un objeto Profesor, para poder guardarlo en el Curso. Para guardar el objeto podemos usar el set que se va a generar de dicho objeto Profesor, ya que es un atributo de la clase Curso.

Main

```

Profesor profesor = new Profesor();
profesor.setNombre("Agustín");
profesor.setApellido("Lima");
Curso curso = new Curso();
curso.setProfesor(profesor); // Setteamos un profesor en el Curso

```

UNO A MUCHOS

Por cada objeto tenemos **una** relación con **muchos** objetos de una clase. Ejemplo: para un curso tengo muchos alumnos. En java para guardar varios objetos de una clase utilizamos colecciones. Y como las listas son las colecciones más rápidas de llenar, utilizamos una lista

```

Public class Alumno {
    private String nombre;
    private String apellido;
}

Public class Curso {
    private List<Alumno> alumnos;

    public List<Alumno> getAlumnos() {
        return alumnos;
    }

    public void setAlumno(List<Alumno> alumnos) {
        this.alumnos = alumnos;
    }
}

```

En este ejemplo en el Main vamos a tener que crear varios objetos Alumno para después guardarlos en un ArrayList de tipo Alumno, para poder guardarlo en el Curso. Para guardar el objeto podemos usar el set que se va a generar de dicho ArrayList de tipo Alumno, ya que es un atributo de la clase Curso.

```
Main
Alumno alumno1 = new Alumno();
alumno1.setNombre("Mariela");
alumno1.setApellido("Gadea");
ArrayList<Alumno> alumnos = new ArrayList();
alumnos.add(alumno1); // Agregamos el alumno a la lista
Curso curso = new Curso();
curso.setAlumnos(alumnos); // Seteamos la lista de alumnos en el Curso
```

UML

El lenguaje de modelado unificado (UML) es un lenguaje de modelado de propósito general. El objetivo principal de UML es definir una forma estándar de visualizar la forma en que se ha diseñado un sistema mediante diagramas. Es bastante similar a los planos utilizados en otros campos de la ingeniería.

UML no es un lenguaje de programación, es más bien un lenguaje visual. Usamos diagramas UML para representar el comportamiento y la estructura de un sistema. Estos diagramas se hacen siempre previos a la codificación del programa en sí, también para facilitar después la creación del programa si ya tenemos en claro qué debemos crear.

Existen varios tipos de diagramas de programación que podemos hacer con UML, en el que vamos a hacer hincapié nosotros es el diagrama de clases.

DIAGRAMAS DE CLASES

Es el componente básico de todos los programas orientados a objetos. Usamos diagramas de clases para representar la estructura de un sistema mostrando las clases del sistema, sus métodos y atributos. Los diagramas de clases también nos ayudan a identificar la relación entre diferentes clases u objetos. Ya sea relaciones entre clases o herencia.

Cada clase está representada por un rectángulo que tiene una subdivisión de tres compartimentos: nombre, atributos y métodos.

Hay tres tipos de modificadores que se utilizan para decidir la visibilidad de atributos y métodos:

- + se usa para el modificador de acceso public.
- # se usa para el modificador de acceso protected.
- se usa para el modificador de acceso private.

Libro
-String título -Integer ejemplares
+void préstamo() +void devolución()

Si tuviéramos una interfaz, se vería así

<<interface>>
Interfaz
+void método()

Nota: El concepto de interfaz se desarrollará en la siguiente guía.

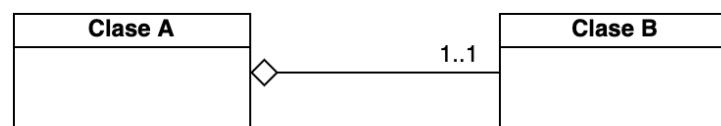
RELACIONES ENTRE CLASES

Las relaciones entre clases se representan con flechas entre las clases. La clase que recibe la relación de la otra clase, como un objeto de la otra clase, es la clase a la que lo toca el rombo.



Y para representar el tipo de relación, ya sea **uno a uno** o de **uno a muchos**, es con un símbolo para cada relación en la flecha. A continuación, veremos uno de los muchos ejemplos:

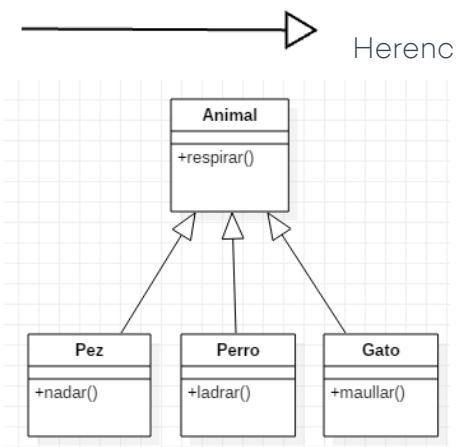
1...1	Uno a uno
1...*	Uno a muchos o muchos a uno



En el primer ejemplo la Clase A tiene a la clase B por ser una composición y en el segundo ejemplo, la clase A usa a la clase B por ser una agregación.

HERENCIA

El concepto de herencia se desarrollará en la siguiente guía. La herencia se representa con la siguiente flecha:



EJERCICIOS DE APRENDIZAJE

En este módulo de POO, vamos a empezar a ver cómo dos o más clases pueden relacionarse entre sí, ya sea por una relación entre clases o mediante una herencia de clases.

Nota: instalar el easyUML por si queremos ver nuestros proyectos como diagramas UML. Encontrarán como instalarlo en Moodle.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Realizar un programa para que una Persona pueda adoptar un Perro. Vamos a contar de dos clases. Perro, que tendrá como atributos: nombre, raza, edad y tamaño; y la clase Persona con atributos: nombre, apellido, edad, documento y Perro.
Ahora deberemos en el main crear dos Personas y dos Perros. Despues, vamos a tener que pensar la lógica necesaria para asignarle a cada Persona un Perro y por ultimo, mostrar desde la clase Persona, la información del Perro y de la Persona.
2. Realizar el juego de la ruleta rusa de agua en Java. Como muchos saben, el juego se trata de un número de jugadores, que, con un revolver de agua, el cual posee una sola carga de agua, se dispara y se moja. Las clases a hacer del juego son las siguientes:

Clase Revolver de agua: esta clase posee los siguientes atributos: posición actual (posición del tambor que se dispara, puede que esté el agua o no) y posición agua (la posición del tambor donde se encuentra el agua). Estas dos posiciones, se generarán aleatoriamente.

Métodos:

- llenarRevolver(): le pone los valores de posición actual y de posición del agua. Los valores deben ser aleatorios.
- mojar(): devuelve true si la posición del agua coincide con la posición actual
- siguienteChorro(): cambia a la siguiente posición del tambor
- toString(): muestra información del revolver (posición actual y donde está el agua)

Clase Jugador: esta clase posee los siguientes atributos: id (representa el número del jugador), nombre (Empezara con Jugador más su ID, "Jugador 1" por ejemplo) y mojado (indica si está mojado o no el jugador). El número de jugadores será decidido por el usuario, pero debe ser entre 1 y 6. Si no está en este rango, por defecto será 6.

Métodos:

- disparo(Revolver r): el método, recibe el revolver de agua y llama a los métodos de mojar() y siguienteChorro() de Revolver. El jugador se apunta, aprieta el gatillo y si el revolver tira el agua, el jugador se moja. El atributo mojado pasa a false y el método devuelve true, sino false.

Clase Juego: esta clase posee los siguientes atributos: Jugadores (conjunto de Jugadores) y Revolver

Métodos:

- llenarJuego(ArrayList<Jugador>jugadores, Revolver r): este método recibe los jugadores y el revolver para guardarlos en los atributos del juego.
 - ronda(): cada ronda consiste en un jugador que se apunta con el revolver de agua y aprieta el gatillo. Si el revolver tira el agua el jugador se moja y se termina el juego, sino se moja, se pasa al siguiente jugador hasta que uno se moje. Si o si alguien se tiene que mojar. Al final del juego, se debe mostrar que jugador se mojó.
- Pensar la lógica necesaria para realizar esto, usando los atributos de la clase Juego.

3. Realizar una baraja de cartas españolas orientada a objetos. Una carta tiene un número entre 1 y 12 (el 8 y el 9 no los incluimos) y un palo (espadas, bastos, oros y copas). Esta clase debe contener un método `toString()` que retorne el número de carta y el palo. La baraja estará compuesta por un conjunto de cartas, 40 exactamente.

Las operaciones que podrá realizar la baraja son:

- barajar(): cambia de posición todas las cartas aleatoriamente.
- siguienteCarta(): devuelve la siguiente carta que está en la baraja, cuando no haya más o se haya llegado al final, se indica al usuario que no hay más cartas.
- cartasDisponibles(): indica el número de cartas que aún se puede repartir.
- darCartas(): dado un número de cartas que nos pidan, le devolveremos ese número de cartas. En caso de que haya menos cartas que las pedidas, no devolveremos nada, pero debemos indicárselo al usuario.
- cartasMonton(): mostramos aquellas cartas que ya han salido, si no ha salido ninguna indicárselo al usuario
- mostrarBaraja(): muestra todas las cartas hasta el final. Es decir, si se saca una carta y luego se llama al método, este no mostrara esa primera carta.

EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Ahora se debe realizar unas mejoras al ejercicio de Perro y Persona. Nuestro programa va a tener que contar con muchas personas y muchos perros. El programa deberá preguntarle a cada persona, que perro según su nombre, quiere adoptar. Dos personas no pueden adoptar al mismo perro, si la persona eligió un perro que ya estaba adoptado, se le debe informar a la persona.
Una vez que la Persona elige el Perro se le asigna, al final deberemos mostrar todas las personas con sus respectivos perros.

2. Nos piden hacer un programa sobre un Cine, que tiene una sala con un conjunto de asientos (8 filas por 6 columnas). De Cine nos interesa conocer la película que se está reproduciendo, la sala con los espectadores y el precio de la entrada. Luego, de las películas nos interesa saber el título, duración, edad mínima y director. Por último, del espectador, nos interesa saber su nombre, edad y el dinero que tiene disponible.
Para representar la sala con los espectadores vamos a utilizar una matriz. Los asientos son etiquetados por una letra y un número la fila A1 empieza al final del mapa como se muestra en la tabla. También deberemos saber si el asiento está ocupado por un espectador o no, si esta ocupado se muestra una X, sino un espacio vacío.

8	A	X	1	8	B	X	1	8	C	X	1	8	D		1	8	E	X	1	8	F	X
7	A	X	1	7	B	X	1	7	C	X	1	7	D	X	1	7	E		1	7	F	X
6	A		1	6	B	X	1	6	C		1	6	D	X	1	6	E	X	1	6	F	
5	A	X	1	5	B		1	5	C	X	1	5	D	X	1	5	E	X	1	5	F	X
4	A	X	1	4	B	X	1	4	C	X	1	4	D	X	1	4	E	X	1	4	F	X
3	A		1	3	B	X	1	3	C	X	1	3	D		1	3	E	X	1	3	F	
2	A	X	1	2	B		1	2	C	X	1	2	D	X	1	2	E	X	1	2	F	
1	A	X	1	1	B	X	1	1	C	X	1	1	D	X	1	1	E	X	1	1	F	X

Se debe realizar una pequeña simulación, en la que se generen muchos espectadores y se los ubique en los asientos aleatoriamente (no se puede ubicar un espectador donde ya este ocupado el asiento).

Los espectadores serán ubicados de uno en uno y para ubicarlos tener en cuenta que sólo se podrá sentar a un espectador si tiene el dinero suficiente para pagar la entrada, si hay espacio libre en la sala y si tiene la edad requerida para ver la película. En caso de que el asiento este ocupado se le debe buscar uno libre.

Al final del programa deberemos mostrar la tabla, podemos mostrarla con la letra y numero de cada asiento o solo las X y espacios vacíos.

3. Ha llegado el momento de poner de prueba tus conocimientos. Para te vamos a contar que te ha contratado "La Tercera Seguros", una empresa aseguradora que brinda a sus clientes coberturas integrales para vehículos.

Luego de un pequeño relevamiento, te vamos a pasar en limpio los requerimientos del sistema que quiere realizar la empresa.

- a. Gestión Integral de clientes. En este módulo vamos a registrar la información personal de cada cliente que posea pólizas en nuestra empresa. Nombre y apellido, documento, mail, domicilio, teléfono.
- b. Gestión de vehículos. Se registra la información de cada vehículo asegurado. Marca, modelo, año, número de motor, chasis, color, tipo (camioneta, sedán, etc.).
- c. Gestión de Pólizas: Se registrará una póliza, donde se guardará los datos tanto de un vehículo, como los datos de un solo cliente. Los datos incluidos en ella son: número de póliza, fecha de inicio y fin de la póliza, cantidad de cuotas, forma de pago, monto total asegurado, incluye granizo, monto máximo granizo, tipo de cobertura (total, contra terceros, etc.). Nota: prestar atención al principio de este enunciado y pensar en las relaciones entre clases. Recuerden que pueden ser de uno a uno, de uno a muchos, de muchos a uno o de muchos a muchos.
- d. Gestión de cuotas: Se registrarán y podrán consultar las cuotas generadas en cada póliza. Esas cuotas van a contener la siguiente información: número de cuota, monto total de la cuota, si está o no pagada, fecha de vencimiento, forma de pago (efectivo, transferencia, etc.).

Debemos realizar el diagrama de clases completo, teniendo en cuenta todos los requerimientos arriba descriptos. Modelando clases con atributos y sus correspondientes relaciones. Para hacer el diagrama podes utilizar easyUML de Java o utilizar una pagina que nos permite hacer diagramas online: [diagramas online](#).

4. Desarrollar un simulador del sistema de votación de facilitadores en Egg-

El sistema de votación de Egg tiene una clase llamada Alumno con los siguientes atributos: nombre completo, DNI y cantidad de votos. Además, crearemos una clase Simulador que va a tener los métodos para manejar los alumnos y sus votaciones. Estos métodos serán llamados desde el main.

- La clase Simulador debe tener un método que genere un listado de alumnos manera aleatoria y lo retorne. Las combinaciones de nombre y apellido deben ser generadas de manera aleatoria. Nota: usar listas de tipo String para generar los nombres y los apellidos.
- Ahora hacer un generador de combinaciones de DNI posibles, deben estar dentro de un rango real de números de documentos. Y agregar a los alumnos su DNI. Este método debe retornar la lista de dnis.

- Ahora tendremos un método que, usando las dos listas generadas, cree una cantidad de objetos Alumno, elegidos por el usuario, y le asigne los nombres y los dnis de las dos listas a cada objeto Alumno. No puede haber dos alumnos con el mismo dni, pero si con el mismo nombre.
- Se debe imprimir por pantalla el listado de alumnos.
- Una vez hecho esto debemos generar una clase Voto, esta clase tendrá como atributos, un objeto Alumno que será el alumno que vota y una lista de los alumnos a los que votó.
- Crearemos un método votación en la clase Simulador que, recibe el listado de alumnos y para cada alumno genera tres votos de manera aleatoria. En este método debemos guardar a el alumno que vota, a los alumnos a los que votó y sumarle uno a la cantidad de votos a cada alumno que reciba un voto, que es un atributo de la clase Alumno.
Tener en cuenta que un alumno no puede votarse a sí mismo o votar más de una vez al mismo alumno. Utilizar un hashset para resolver esto.
- Se debe crear un método que muestre a cada Alumno con su cantidad de votos y cuales fueron sus 3 votos.
- Se debe crear un método que haga el recuento de votos, este recibe la lista de Alumnos y comienza a hacer el recuento de votos.
- Se deben crear 5 facilitadores con los 5 primeros alumnos votados y se deben crear 5 facilitadores suplentes con los 5 segundos alumnos más votados. A continuación, mostrar los 5 facilitadores y los 5 facilitadores suplentes.

CURSO DE PROGRAMACIÓN FULL STACK

HERENCIA

PARADIGMA ORIENTADO A OBJETOS



EGG

GUÍA DE HERENCIA

HERENCIA

La herencia es una relación fuerte entre dos clases donde una clase es **padre o madre** de otra. La herencia es un pilar importante de POO. Es el mecanismo mediante el cual una clase es capaz de heredar todas las características (atributos y métodos) de otra clase.

Las propiedades comunes se definen en la **superclase** (clase padre) y las **subclases** heredan estas propiedades (clase hija). En esta relación, la frase "**Un objeto es un-tipo-de**" debe tener sentido, por ejemplo: un perro es un tipo de animal, o también, una heladera es un tipo de electrodoméstico.

La herencia apoya el concepto de "reutilización", es decir, cuando queremos crear una nueva clase y ya existe una clase que incluye parte del código que queremos, podemos utilizar esa clase que ya tiene el código que queremos y hacer de la nueva clase una subclase. Al hacer esto, estamos reutilizando los campos y métodos de la clase existente.

La manera de usar herencia es a través de la palabra **extends**.

```
class subclase extends superclase {  
    // atributos y métodos  
}
```

HERENCIA Y ATRIBUTOS

La subclase (Hija) como hemos dicho recibe todos los atributos de la superclase (Madre), y además la subclase puede tener atributos propios.

```
Class Persona {  
    protected String nombre;  
    protected Integer edad;  
    protected Integer documento;  
}  
  
Class Alumno extends Persona {  
    private String materia;  
}
```

El siguiente programa crea una superclase llamada **Persona**, la clase crea personas según su nombre, edad y documento, y una subclase llamada **Alumno**, que recibe **todos los atributos de Persona**. De esta manera se piensa que los atributos de alumno son nombre, edad y documento, que son propios de cualquier Persona y materia que sería específico de cada Alumno. Usualmente la superclase suele ser un concepto muy general y abstracto, para que pueda utilizarse para varias subclases.

En la superclase podemos observar que los atributos están creados con el modificador de acceso **protected** y no private. Esto es porque el modificador de acceso protected permite que las subclases puedan acceder a los atributos **sin la necesidad** de getters y setters.

Los atributos se trabajan como protected también, porque una subclase no hereda los miembros privados de su clase principal. Sin embargo, si la superclase tiene métodos públicos o protegidos (como getters y setters) para acceder a sus campos privados, estos también pueden ser utilizados por la subclase. Entonces, para evitar esto, usamos atributos protected.

Visibilidad	Public	Private	Protected	Default
Desde la misma Clase	SI	SI	SI	SI
Desde cualquier Clase del mismo Paquete	SI	NO	SI	SI
Desde una Subclase del mismo Paquete	SI	NO	SI	SI
Desde una Subclase fuera del mismo Paquete	SI	NO	SI, a través de la herencia	NO
Desde cualquier Clase fuera del Paquete	SI	NO	NO	NO

HERENCIA Y CONSTRUCTORES

Una diferencia entre los constructores y los métodos es que los **constructores no se heredan**, pero los **métodos sí**. Todos los constructores definidos en una superclase pueden ser usados desde constructores de las subclases a través de la palabra clave **super**.

La palabra clave super es la que me permite elegir qué constructor, entre los que tiene definida la clase padre, es el que debo usar. Si la superclase tiene definido el constructor vacío y no colocamos una llamada explícita super, se llamará el constructor vacío de la superclase.

```
Class Persona {  
    public Persona(String nombre, Integer edad, Integer documento){  
        this.nombre = nombre;  
        this.edad = edad;  
        this.documento = documento;  
    }  
}
```

```

Class Alumno extends Persona {

public Alumno(String materia, String nombre, Integer edad, Integer
documento){

super.(nombre, edad, documento);

this.materia = materia;

}

}

```

En el ejemplo podemos ver que el constructor de la clase Alumno utiliza la palabra clave super para llamar al constructor de la superclase y de esa manera utilizarlo como constructor propio y además sumarle su atributo materia.

La palabra clave super nos sirve para hacer referencia o llamar a los atributos, métodos y constructores de la superclase en las subclases.

```

super.atributoClasePadre;
super.metodoClasePadre;

```

HERENCIA Y MÉTODOS

Todos los métodos accesibles o visibles de una superclase **se heredan** a sus subclases. Pero, ¿qué ocurre si una subclase necesita que uno de sus métodos heredados funcione de manera diferente?

Los métodos heredados pueden ser **redefinidos** en las clases hijas. Este mecanismo se lo llama **sobreescritura**. La sobreescritura permite que las clases hijas sumen sus particulares en torno al funcionamiento y agrega coherencia al modelo. Esto se logra poniendo la anotación **@Override** arriba del método que queremos sobreescribir, el método debe llamarse igual en la subclase como en la superclase.

```

Class Persona {
public void codear(){
System.out.println("Una persona comun no codea");
}

Class Alumno extends Persona {
@Override
public void codear(){
System.out.println("Está aprendiendo");
}
}

```

En el ejemplo podemos ver que tenemos el mismo método en la clase Persona, que en la clase Alumno, el método nos va a informar cuales son sus capacidades para codear según la clase que llamemos. Por lo que en la clase Alumno, cuando heredamos el método lo sobreescribimos para cambiar su funcionamiento, y hacer que diga algo distinto al método de Persona.

En los métodos de la superclase, también podemos hacer que tengan un modificador de acceso `protected`, esto hace que los únicos que puedan invocar a ese método sean las subclases.

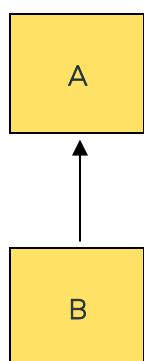
POLIMORFISMO

El término polimorfismo es una palabra de origen griego que significa "muchas formas". Este término se utiliza en POO para referirse a la propiedad por la que es posible enviar **mensajes** sintácticamente **iguales a objetos de tipos distintos**, es decir, que la misma operación se realiza en las clases de diferente forma. Estas operaciones tienen el mismo significado y comportamiento, pero internamente, cada operación se realiza de diferente forma. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía. Esto hace referencia a la idea de que podemos tener un método definido en la superclase y que las subclases tengan el mismo método, pero con distintas funcionalidades

TIPOS DE HERENCIA

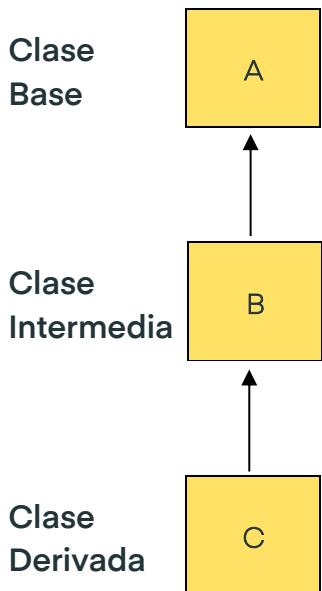
HERENCIA ÚNICA

En la herencia única, las subclases heredan las características de solo una superclase. En la imagen a continuación, la clase A sirve como clase base para la clase derivada B.



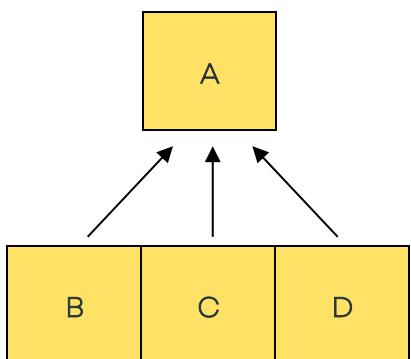
HERENCIA MULTINIVEL

En la herencia multinivel, una clase derivada heredará una clase base y, además, la clase derivada también actuará como la clase base de otra clase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, que a su vez sirve como clase base para la clase derivada C. En Java, una clase no puede acceder directamente a los miembros de los "abuelos".



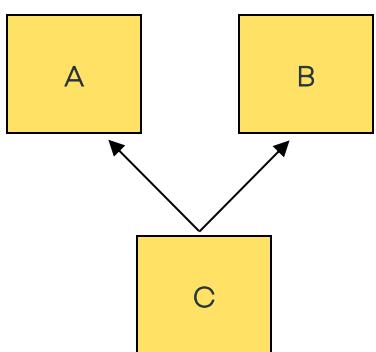
HERENCIA JERÁRQUICA

En la herencia jerárquica, una clase sirve como una superclase (clase base) para más de una subclase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, C y D.



HERENCIA MÚLTIPLE (A TRAVÉS DE INTERFACES)

En Herencia múltiple, una clase puede tener más de una superclase y heredar características de todas las clases principales. Tenga en cuenta que Java no admite herencia múltiple con clases. En Java, podemos lograr herencia múltiple **solo a través de Interfaces**. En la imagen a continuación, la Clase C se deriva de la interfaz A y B.



MODIFICADORES DE CLASES Y MÉTODOS

CLASES FINALES

El modificador final puede utilizarse también como modificador de clases. Al marcar una clase como final impedimos que se generen hijos a partir de esta clase, es decir, cortamos la jerarquía de herencia.

```
public final class Animal{ }
```

MÉTODOS FINALES

El modificador final puede utilizarse también como modificador de métodos. La herencia nos permite reutilizar el código existente y uno de los mecanismos es crear una subclase y sobrescribir alguno de los métodos de la clase padre. Cuando un método es marcado como final en una clase, evitamos que sus clases hijas puedan sobrescribir estos métodos.

```
public final void método(){ }
```

CLASES ABSTRACTAS

En java se dice que una clase es abstracta cuando no se permiten instancias de esa clase, es decir que no se pueden crear objetos. Nosotros haríamos una clase abstracta por dos razones. Usualmente las clases abstractas suelen ser las superclases, esto lo hacemos porque creemos que la superclase o clase padre, no debería poder instanciarse. Por ejemplo, si tenemos una clase Animal, el usuario no debería poder crear un Animal, sino que solo debería poder instanciar objetos de las subclases.

```
public abstract class Animal { }
```

Otra razón es porque decidimos hacer métodos abstractos en nuestra superclase. Cuando una clase posee al menos un método abstracto esa clase necesariamente **debe ser marcada como abstracta**.

MÉTODOS ABSTRACTOS

Un método abstracto es un método declarado, pero no implementado, es decir, es un método del que solo se escribe su nombre, parámetros, y tipo devuelto, pero no su código de implementación. Estos métodos se heredan y se sobreescreiben por las clases hijas quienes son las responsables de implementar sus funcionalidades. ¿Qué utilidad tiene un método abstracto? Podemos ver un método abstracto como una palanca que fuerza dos cosas: la primera, que no se puedan crear objetos de una clase. La segunda, que todas las subclases sobreescrbían el método declarado como abstracto.

```
public abstract void codear();  
Hija:  
@Override  
Public void codear(){  
System.out.println("Está aprendiendo"); }
```

INTERFACES

Una interfaz es sintácticamente similar a una clase abstracta, en la que puede especificar uno o más métodos que no tienen cuerpo. Esos métodos deben ser implementados por una clase para que se definan sus acciones.

Por lo tanto, una interfaz especifica qué se debe hacer, pero no cómo hacerlo. Una vez que se define una interfaz, cualquier cantidad de clases puede implementarla. Además, una clase puede implementar cualquier cantidad de interfaces.

Para implementar una interfaz, una clase debe proporcionar cuerpos (implementaciones) para los métodos descritos por la interfaz. Cada clase es libre de determinar los detalles de su propia implementación. Dos clases pueden implementar la misma interfaz de diferentes maneras, pero **cada clase aún admite el mismo conjunto de métodos**.

Para decirle a Java que estamos trabajando sobre una interfaz vamos a tener que utilizar la palabra **interface**, esto se vería así:

```
public interface nombreInterfaz { }
```

Para una interfaz, el modificador de acceso es **public** o no se usa. Cuando no se incluye ningún modificador de acceso, los resultados de acceso serán los predeterminados y la interfaz solo están disponibles para otros miembros de su paquete. Si se declara como **public**, la interfaz puede ser utilizada por cualquier otra clase. El nombre de la interfaz puede ser cualquier identificador válido.

INSTANCIAR UNA INTERFAZ

Aunque las interfaces van a ser implementadas por clases y van a tener métodos, al igual que una clase abstracta, esta, no se va a poder instanciar. La definición de un interfaz no tiene constructor, por lo que no es posible invocar el operador **new** sobre un tipo interfaz, por lo que no podemos crear objetos del tipo interfaz.

IMPLEMENTACIÓN DE INTERFAZ

Una vez que se ha definido una interfaz, una o más clases pueden implementar esa interfaz. Para implementar una interfaz, incluya la palabra reservada **implements** en la definición de clase.

```
public class Clase implements Interfaz { }
```

Los métodos de la interfaz los podemos implementar en nuestra y se van a sobrescribir desde la interfaz, métodos que recordemos, **no tendrán cuerpo**. Nuestra tarea será completar esos métodos que implementamos de la interfaz.

MÉTODOS

Los métodos de una interfaz se declaran utilizando solo su tipo de devolución y firma. Son, esencialmente, métodos abstractos. Por lo tanto, cada clase que incluye dicha interfaz debe implementar todos sus métodos. En una interfaz, los métodos son implícitamente públicos.

```
public interface Interfaz {  
    public void metodo();  
    public int sumar();  
}  
  
public class Clase implements Interfaz {  
    @Override  
    Public void metodo(){  
        System.out.println("Implementacion del método");  
    }  
}  
  
    @Override  
    Public int sumar(){  
        int suma = 2 + 2;  
        return suma;  
    }  
}
```

Como podemos ver en la interfaz teníamos dos métodos sin cuerpo y al implementar la interfaz en nuestra clase, los sobrescribimos y les dimos una funcionalidad a dichos métodos.

VARIABLES

Las variables declaradas en una interfaz no son variables de instancia. En cambio, son implícitamente *public* y *final*, además, deben inicializarse. Por lo tanto, son esencialmente constantes, recordemos que se escriben en **mayúsculas** para diferenciarlas de variables.

```
public interface Interfaz {  
    public final int CONSTANTE = 10;  
    public void metodo();  
}  
  
public class Clase implements Interfaz {  
    @Override  
    Public void metodo(){  
        System.out.println("La constante tiene un valor de " + CONSTANTE);  
    }  
}
```

Las constantes definidas en nuestra interfaz pueden ser llamadas en la clase, solo con el nombre y de esa manera podemos utilizar los valores definidos en la interfaz.

EJERCICIOS DE APRENDIZAJE

En este módulo de POO, vamos a empezar a ver cómo dos o más clases pueden relacionarse entre si mediante una herencia de clases.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Tenemos una clase padre Animal junto con sus 3 clases hijas Perro, Gato, Caballo. La clase Animal tendrá como atributos el nombre, alimento, edad y raza del Animal.
Crear un método en la clase Animal a través del cual cada clase hija deberá mostrar luego un mensaje por pantalla informando de que se alimenta. Generar una clase Main que realice lo siguiente:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         //-->Declaracion del objeto PERRA  
4         Animal perro = new Perro("Stich", "Carnivoro", 15, "Doberman");  
5         perro.Alimentarse();  
6         //-->Declaracion de otro objeto PERRA  
7         Perro perro1 = new Perro("Teddy", "Croquetas", 10, "Chihuahua");  
8         perro1.Alimentarse();  
9  
10        //-->Declaracion del objeto Gato  
11        Animal gato = new Gato("Pelusa", "Galletas", 15, "Siames");  
12        gato.Alimentarse();  
13        //-->Declaracion del objeto Caballo  
14        Animal caballo = new Caballo("Spark", "Pasto", 25, "Fino");  
15        caballo.Alimentarse();  
16    }  
17 }  
18 }
```

2. Crear una superclase llamada Electrodoméstico con los siguientes atributos: precio, color, consumo energético (letras entre A y F) y peso.

Los constructores que se deben implementar son los siguientes:

- Un constructor vacío.
- Un constructor con todos los atributos pasados por parámetro.

Los métodos a implementar son:

- Métodos getters y setters de todos los atributos.
- Método comprobarConsumoEnergetico(char letra): comprueba que la letra es correcta, sino es correcta usara la letra F por defecto. Este método se debe invocar al crear el objeto y no será visible.

- Método comprobarColor(String color): comprueba que el color es correcto, y si no lo es, usa el color blanco por defecto. Los colores disponibles para los electrodomésticos son blanco, negro, rojo, azul y gris. No importa si el nombre está en mayúsculas o en minúsculas. Este método se invocará al crear el objeto y no será visible.
- Método crearElectrodomestico(): le pide la información al usuario y llena el electrodoméstico, también llama los métodos para comprobar el color y el consumo. Al precio se le da un valor base de \$1000.
- Método precioFinal(): según el consumo energético y su tamaño, aumentará el valor del precio. Esta es la lista de precios:

<u>LETRA</u>	<u>PRECIO</u>
A	\$1000
B	\$800
C	\$600
D	\$500
E	\$300
F	\$100

<u>PESO</u>	<u>PRECIO</u>
Entre 1 y 19 kg	\$100
Entre 20 y 49 kg	\$500
Entre 50 y 79 kg	\$800
Mayor que 80 kg	\$1000

A continuación se debe crear una subclase llamada Lavadora, con el atributo carga, además de los atributos heredados.

Los constructores que se implementarán serán:

- Un constructor vacío.
- Un constructor con la carga y el resto de atributos heredados. Recuerda que debes llamar al constructor de la clase padre.

Los métodos que se implementara serán:

- Método get y set del atributo carga.
- Método crearLavadora (): este método llama a crearElectrodomestico() de la clase padre, lo utilizamos para llenar los atributos heredados del padre y después llenamos el atributo propio de la lavadora.
- Método precioFinal(): este método será heredado y se le sumará la siguiente funcionalidad. Si tiene una carga mayor de 30 kg, aumentará el precio en \$500, si la carga es menor o igual, no se incrementará el precio. Este método debe llamar al método padre y añadir el código necesario. Recuerda que las condiciones que hemos visto en la clase Electrodoméstico también deben afectar al precio.

Se debe crear también una subclase llamada Televisor con los siguientes atributos: resolución (en pulgadas) y sintonizador TDT (booleano), además de los atributos heredados.

Los constructores que se implementarán serán:

- Un constructor vacío.
- Un constructor con la resolución, sintonizador TDT y el resto de atributos heredados. Recuerda que debes llamar al constructor de la clase padre.

Los métodos que se implementara serán:

- Método get y set de los atributos resolución y sintonizador TDT.
- Método crearTelevisor(): este método llama a crearElectrodomestico() de la clase padre, lo utilizamos para llenar los atributos heredados del padre y después llenamos los atributos del televisor.
- Método precioFinal(): este método será heredado y se le sumará la siguiente funcionalidad. Si el televisor tiene una resolución mayor de 40 pulgadas, se incrementará el precio un 30% y si tiene un sintonizador TDT incorporado, aumentará \$500. Recuerda que las condiciones que hemos visto en la clase Electrodomestico también deben afectar al precio.

Finalmente, en el main debemos realizar lo siguiente:

Vamos a crear una Lavadora y un Televisor y llamar a los métodos necesarios para mostrar el precio final de los dos electrodomésticos.

3. Siguiendo el ejercicio anterior, en el main vamos a crear un ArrayList de Electrodomésticos para guardar 4 electrodomésticos, ya sean lavadoras o televisores, con valores ya asignados.

Luego, recorrer este array y ejecutar el método precioFinal() en cada electrodoméstico. Se deberá también mostrar el precio de cada tipo de objeto, es decir, el precio de todos los televisores y el de las lavadoras. Una vez hecho eso, también deberemos mostrar, la suma del precio de todos los Electrodomésticos. Por ejemplo, si tenemos una lavadora con un precio de 2000 y un televisor de 5000, el resultado final será de 7000 (2000+5000) para electrodomésticos, 2000 para lavadora y 5000 para televisor.

4. Se plantea desarrollar un programa que nos permita calcular el área y el perímetro de formas geométricas, en este caso un círculo y un rectángulo. Ya que este cálculo se va a repetir en las dos formas geométricas, vamos a crear una Interfaz, llamada calculosFormas que tendrá, los dos métodos para calcular el área, el perímetro y el valor de PI como constante.

Desarrollar el ejercicio para que las formas implementen los métodos de la interfaz y se calcule el área y el perímetro de los dos. En el main se crearán las formas y se mostrará el resultado final.

Área círculo: $\pi * \text{radio}^2$ / **Perímetro círculo:** $\pi * \text{diámetro}$.

Área rectángulo: $\text{base} * \text{altura}$ / **Perímetro rectángulo:** $(\text{base} + \text{altura}) * 2$

EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, puedes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. En un puerto se alquilan amarres para barcos de distinto tipo. Para cada Alquiler se guarda: el nombre, documento del cliente, la fecha de alquiler, fecha de devolución, la posición del amarre y el barco que lo ocupará.

Un Barco se caracteriza por: su matrícula, su eslora en metros y año de fabricación.

Sin embargo, se pretende diferenciar la información de algunos tipos de barcos especiales:

- Número de mástiles para veleros.
- Potencia en CV para barcos a motor.
- Potencia en CV y número de camarotes para yates de lujo.

Un alquiler se calcula multiplicando el número de días de ocupación (calculado con la fecha de alquiler y devolución), por un valor módulo de cada barco (obtenido simplemente multiplicando por 10 los metros de eslora).

En los barcos de tipo especial el módulo de cada barco, se calcula sacando el modulo normal y sumándole el atributo particular de cada barco. En los veleros se suma el número de mástiles, en los barcos a motor se le suma la potencia en CV y en los yates se suma la potencia en CV y el número de camarotes.

Utilizando la herencia de forma apropiada, deberemos programar en Java, las clases y los métodos necesarios que permitan al usuario elegir el barco que quiera alquilar y mostrarle el precio final de su alquiler.

2. Crear una superclase llamada Edificio con los siguientes atributos: ancho, alto y largo. La clase edificio tendrá como métodos:

- Método calcularSuperficie(): calcula la superficie del edificio.
- Método calcularVolumen(): calcula el volumen del edificio.

Estos métodos serán abstractos y los implementarán las siguientes clases:

- Clase Polideportivo con su nombre y tipo de instalación que puede ser Techado o Abierto, esta clase implementará los dos métodos abstractos y los atributos del padre.

- Clase EdificioDeOficinas con sus atributos número de oficinas, cantidad de personas por oficina y numero de pisos. Esta clase implementará los dos métodos abstractos y los atributos del padre.

De esta clase nos interesa saber cuantas personas pueden trabajar en todo el edificio, el usuario dirá cuantas personas entran en cada oficina, cuantas oficinas y el numero de piso (suponiendo que en cada piso hay una oficina). Crear el método cantPersonas(), que muestre cuantas personas entrarían en un piso y cuantas en todo el edificio.

Por ultimo, en el main vamos a crear un ArrayList de tipo Edificio. El ArrayList debe contener dos polideportivos y dos edificios de oficinas. Luego, recorrer este array y ejecutar los métodos calcularSuperficie y calcularVolumen en cada Edificio. Se deberá mostrar la superficie y el volumen de cada edificio.

Además de esto, para la clase Polideportivo nos interesa saber cuantos polideportivos son techados y cuantos abiertos. Y para la clase EdificioDeOficinas deberemos llamar al método cantPersonas() y mostrar los resultados de cada edificio de oficinas.

3. Una compañía de promociones turísticas desea mantener información sobre la infraestructura de alojamiento para turistas, de forma tal que los clientes puedan planear sus vacaciones de acuerdo a sus posibilidades. Los alojamientos se identifican por un nombre, una dirección, una localidad y un gerente encargado del lugar. La compañía trabaja con dos tipos de alojamientos: Hoteles y Alojamientos Extrahoteleros.

Los Hoteles tienen como atributos: Cantidad de Habitaciones, Número de Camas, Cantidad de Pisos, Precio de Habitaciones. Y estos pueden ser de cuatro o cinco estrellas. Las características de las distintas categorías son las siguientes:

- Hotel **** Cantidad de Habitaciones, Número de camas, Cantidad de Pisos, Gimnasio, Nombre del Restaurante, Capacidad del Restaurante, Precio de las Habitaciones.
- Hotel ***** Cantidad de Habitaciones, Número de camas, Cantidad de Pisos, Gimnasio, Nombre del Restaurante, Capacidad del Restaurante, Cantidad Salones de Conferencia, Cantidad de Suites, Cantidad de Limosinas, Precio de las Habitaciones.

Los gimnasios pueden ser clasificados por la empresa como de tipo “A” o de tipo “B”, de acuerdo a las prestaciones observadas. Las limosinas están disponibles para cualquier cliente, pero sujeto a disponibilidad, por lo que cuanto más limosinas tenga el hotel, más caro será.

El precio de una habitación debe calcularse de acuerdo a la siguiente fórmula: $\text{PrecioHabitación} = \$50 + (\$1 \times \text{capacidad del hotel}) + (\text{valor agregado por restaurante}) + (\text{valor agregado por gimnasio}) + (\text{valor agregado por limosinas})$.

Donde:

Valor agregado por el restaurante:

- \$10 si la capacidad del restaurante es de menos de 30 personas.
- \$30 si está entre 30 y 50 personas.
- \$50 si es mayor de 50.

Valor agregado por el gimnasio:

- \$50 si el tipo del gimnasio es A.
- \$30 si el tipo del gimnasio es B.

Valor agregado por las limosinas:

- \$15 por la cantidad de limosinas del hotel.

En contraste, los Alojamientos Extra hoteleros proveen servicios diferentes a los de los hoteles, estando más orientados a la vida al aire libre y al turista de bajos recursos. Por cada Alojamiento Extrahotelero se indica si es privado o no, así como la cantidad de metros cuadrados que ocupa. Existen dos tipos de alojamientos extrahoteleros: los Camping y las Residencias. Para los Camping se indica la capacidad máxima de carpas, la cantidad de baños disponibles y si posee o no un restaurante dentro de las instalaciones. Para las residencias se indica la cantidad de habitaciones, si se hacen o no descuentos a los gremios y si posee o no campo deportivo. Realizar un programa en el que se representen todas las relaciones descriptas.

Realizar un sistema de consulta que le permite al usuario consultar por diferentes criterios:

- todos los alojamientos.
- todos los hoteles de más caro a más barato.
- todos los campings con restaurante
- todas las residencias que tienen descuento.

4. Sistema Gestión Facultad. Se pretende realizar una aplicación para una facultad que gestione la información sobre las personas vinculadas con la misma y que se pueden clasificar en tres tipos: estudiantes, profesores y personal de servicio. A continuación, se detalla qué tipo de información debe gestionar esta aplicación:

- Por cada persona, se debe conocer, al menos, su nombre y apellidos, su número de identificación y su estado civil.
- Con respecto a los empleados, sean del tipo que sean, hay que saber su año de incorporación a la facultad y qué número de despacho tienen asignado.
- En cuanto a los estudiantes, se requiere almacenar el curso en el que están matriculados.
- Por lo que se refiere a los profesores, es necesario gestionar a qué departamento pertenecen (lenguajes, matemáticas, arquitectura, ...).
- Sobre el personal de servicio, hay que conocer a qué sección están asignados (biblioteca, decanato, secretaría, ...).

El ejercicio consiste, en primer lugar, en definir la jerarquía de clases de esta aplicación. A continuación, debe programar las clases definidas en las que, además de los constructores, hay que desarrollar los métodos correspondientes a las siguientes operaciones:

- Cambio del estado civil de una persona.
- Reasignación de despacho a un empleado.
- Matriculación de un estudiante en un nuevo curso.
- Cambio de departamento de un profesor.
- Traslado de sección de un empleado del personal de servicio.
- Imprimir toda la información de cada tipo de individuo. Incluya un programa de prueba que instancie objetos de los distintos tipos y pruebe los métodos desarrollados.

CURSO DE PROGRAMACIÓN FULL STACK

MANEJO DE EXCEPCIONES

PARADIGMA ORIENTADO A OBJETOS



EGG

GUÍA DE MANEJO DE EXCEPCIONES

EXCEPCIONES

El término excepción es una abreviación de la frase “Evento Excepcional”. Una *excepción* es un evento que ocurre durante la ejecución de un programa que interrumpe el flujo normal de las instrucciones del programa.

Existen muchas clases de errores que pueden provocar una excepción, desde un desbordamiento de memoria o un disco duro estropeado hasta un intento de dividir por cero o intentar acceder a un vector fuera de sus límites. Cuando esto ocurre, la máquina virtual Java crea un objeto de la clase **exception** o **error** y se notifica el hecho al sistema de ejecución. Se dice, que se ha **lanzado una excepción** (“*Throwing Exception*”). Luego, el objeto, llamado excepción, contiene información sobre el error, incluyendo su tipo y el estado del programa cuando el error ocurrió.

Después de que un método lanza una excepción, el sistema, en tiempo de ejecución, intenta encontrar algo que maneje esa excepción. El conjunto de posibles “algo” para manejar la excepción es la lista ordenada de los métodos que habían sido llamados hasta llegar al método que produjo el error. Esta lista de métodos se conoce como *pila de llamadas*. Luego, el sistema en tiempo de ejecución busca en la pila de llamadas el método que contenga un bloque de código que pueda manejar la excepción. Este bloque de código es llamado **manejador de excepciones**.

Concretamente, una *excepción* en java es **un objeto que modela un evento excepcional, el cual no debería haber ocurrido**. Como observamos anteriormente, al ocurrir estos tipos de evento la máquina virtual no debe continuar con la ejecución normal del programa. Es evidente que las excepciones son objetos especiales, son objetos con la capacidad de cambiar el flujo normal de ejecución. Cuando se detecta un error, una excepción debe ser lanzada.

Ejemplos de situaciones que provocan una excepción:

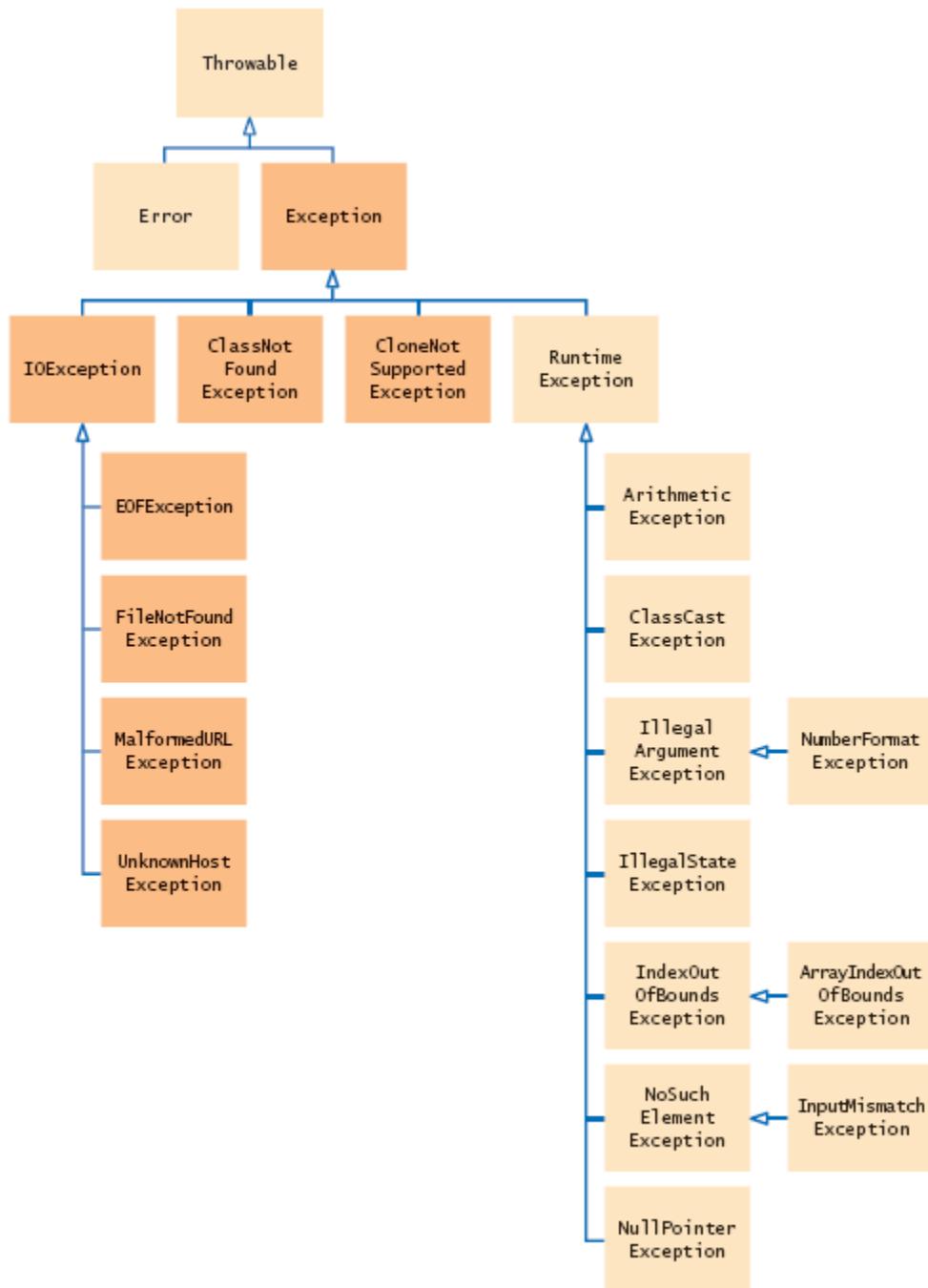
- No hay memoria disponible para asignar
- Acceso a un elemento de un array fuera de rango
- Leer por teclado un dato de un tipo distinto al esperado
- Error al abrir un fichero
- División por cero

JERARQUIA DE EXCEPCIONES

En Java, todas las excepciones están representadas por clases. Todas las clases de excepción se derivan de una clase llamada **Throwable**. Por lo tanto, cuando se produce una excepción en un programa, se genera un objeto de algún tipo de clase de excepción.

Hay dos subclases directas de **Throwable**: **Exception** y **Error**:

1. Las excepciones de tipo Error están relacionadas con errores que ocurren en la Máquina Virtual de Java y no en tu programa. Este tipo de excepciones escapan a su control y, por lo general, tu programa no se ocupará de ellas. Por lo tanto, este tipo de excepciones no se describen aquí.
2. Los errores que resultan de la actividad del programa están representados por subclases de Exception. Por ejemplo, dividir por cero, límite de matriz y errores de archivo caen en esta categoría. En general, tu programa debe manejar excepciones de estos tipos. Una subclase importante de Exception es RuntimeException, que se usa para representar varios tipos comunes de errores en tiempo de ejecución.



MANEJADOR DE EXCEPCIONES

El manejo de excepciones Java se gestiona a través de cinco palabras clave: **try**, **catch**, **throw** y **finally**. Forman un subsistema interrelacionado en el que el uso de uno, implica el uso del otro.

Las declaraciones del programa que desea supervisar para excepciones están contenidas dentro de un **bloque try**. Si se produce una excepción dentro del bloque **try**, se lanza. Tu código puede atrapar esta excepción usando **catch** y manejarlo de una manera racional. Las excepciones generadas por el sistema son lanzadas automáticamente por el sistema de tiempo de ejecución de Java. Para lanzar manualmente una excepción, use la palabra clave **throw**. En algunos casos, una excepción arrojada por un método debe ser especificada como tal por una cláusula **throws**. Cualquier código que debe ejecutarse al salir de un bloque try se coloca en un bloque **finally**.

Ahora vamos a ver en detalle cada palabra clave dentro del manejo de excepciones.

EL BLOQUE TRY

Lo primero que hay que hacer para que un método sea capaz de tratar una excepción generada por la máquina virtual Java o por el propio programa mediante una instrucción **throws** encerrar las instrucciones susceptibles de generarla en un bloque **try**. En el bloque try vamos a poner una serie de instrucciones que creemos que puede llegar a tirar una excepción durante su ejecución y queremos manejarla para evitar la finalización del programa.

```
try {  
    Instrucción1;  
    Instrucción2;  
    Instrucción3;  
    Instrucción4  
    ...  
}
```

Cualquier excepción que se produzca por alguna instrucción, dentro del bloque **try** será analizada por el bloque o bloques **catch**. En el momento en que se produzca la excepción, se abandona el bloque **try**, y las instrucciones que sigan al punto donde se produjo la excepción no son ejecutadas. Cada bloque **try** debe tener asociado por lo menos un bloque **catch**.

EL BLOQUE CATCH

Por cada bloque **try** pueden declararse uno o varios bloques **catch**, cada uno de ellos capaz de tratar un tipo u otro de excepción. Para declarar el tipo de excepción que es capaz de tratar un bloque **catch**, se declara un objeto cuya clase es la clase de la excepción que se desea tratar o una de sus superclases.

```
try {  
    BloqueDeInstrucciones  
} catch (TipoExcepción nombreVariable) {  
    BloqueCatch  
} catch (TipoExcepción nombreVariable) {  
    BloqueCatch  
}
```

Al producirse la excepción dentro de un bloque **try**, la ejecución del programa se pasa al primer bloque **catch**. Si la clase de la excepción se corresponde con la clase o alguna subclase de la clase declarada en el bloque **catch**, se ejecuta el bloque de instrucciones **catch** y a continuación se pasa el control del programa a la primera instrucción a partir de los bloques **try-catch**. Lo más adecuado es utilizar excepciones lo más cercanas al tipo de error previsto, ya que lo que se pretende es recuperar al programa de alguna condición de error y si “se meten todas las condiciones en la misma bolsa”, seguramente habrá que averiguar después qué condición de error se produjo para poder dar una respuesta adecuada.

```
try {  
    // Se intenta hacer la división  
    int division = 10 / 0;  
} catch (ArithmetricException a) {  
    // Si la división falla el programa va al bloque catch y se ejecuta el  
    System.out.println  
    System.out.println("Error: división por cero");  
}
```

En este ejemplo en el bloque **try** hacemos una división por cero, las divisiones por cero generan un tipo de excepción llamado, **ArithmetricException**. En el bloque **catch** ponemos como tipo de excepción la **ArithmetricException** y dentro del bloque ponemos un mensaje que explique cual ha sido el error.

MÉTODOS THROWABLE

Dentro del bloque **catch**, utilizamos un **System.out.print** para mostrar el error, pero no hemos estado haciendo nada con el objeto de excepción en sí mismo. Como muestran todos los ejemplos anteriores, una cláusula **catch** especifica un tipo de excepción y un parámetro.

El parámetro recibe el objeto de excepción. Como todas las excepciones son subclases de **Throwable**, todas las excepciones admiten los métodos definidos por Throwable.

Estos métodos son:

Método	Sintaxis	Descripción
getMessage	String getMessage()	Devuelve una descripción de la excepción
fillInStackTrace	Throwable fillInStackTrace()	Devuelve un objeto Throwable que contiene un seguimiento de pila completo. Este objeto se puede volver a lanzar.
toString	String toString()	Devuelve un objeto String que contiene una descripción completa de la excepción. Este método lo llama println() cuando se imprime un objeto Throwable.

```
try {
    int division = 10 / 0;
} catch (ArithmaticException a) {
    System.out.println("Error:" + a.getMessage());
    System.out.println("Error:" + a)
    System.out.println(a.fillStrakTrace());
}
```

Resultado:

```
Error: / by zero
Error: / by zero
Error: java.lang.ArithmaticException: / by zero
```

EL BLOQUE FINALLY

El bloque **finally** se utiliza para ejecutar un bloque de instrucciones sea cual sea la excepción que se produzca. Este bloque se ejecutará siempre, incluso si no se produce ninguna excepción. Sirve para no tener que repetir código en el bloque try y en los bloques catch. El bloque finally es un buen lugar en donde liberar los recursos tomados dentro del bloque de intento.

```

try {
    BloqueDeInstrucciones
} catch (TipoExcepción nombreVariable) {
    MensajeDeError
} catch (TipoExcepción nombreVariable) {
    MensajeDeError
} finally {
    CódigoFinal
}

```

Ejemplo:

```

try {
    // Se intenta hacer la división
    int division = 10 / 0;
} catch (ArithmaticException a) {
    // Si la división falla el programa va al bloque catch y se ejecuta el
    System.out.println
    System.out.println("Error: división por cero");
} finally {
    // Si el programa hizo la división o no, este System.out.println se va a
    ejecutar igual
    System.out.println("Saliendo del try");
}

```

LA CLAÚSULA THROWS

La cláusula **throws** lista las excepciones que un método puede lanzar. Los tipos de excepciones que lanza el método se especifica después de los paréntesis de un método, con una cláusula **throws**. Un método puede lanzar objetos de la clase indicada o de subclases de la clase indicada.

Java distingue entre las excepciones verificadas y errores. Las excepciones verificadas deben aparecer en la cláusula throws de ese método. Como las RuntimeExceptions y los Errores pueden aparecer en cualquier método, no tienen que listarse en la cláusula throws y es por esto que decimos que no están verificadas. Todas las excepciones que no son RuntimeException y que un método puede lanzar deben listarse en la cláusula throws del método y es por eso que decimos que están verificadas. El requisito de atrapar excepciones en Java exige que el programador atrape todas las excepciones verificadas o bien las coloque en la cláusula throws de un método.

Si la excepción no se trata, el manejador de excepciones realiza lo siguiente:

- Muestra la descripción de la excepción.
- Muestra la traza de la pila de llamadas.
- Provoca el final del programa.

Colocar una excepción en la cláusula *throws* obliga a otros métodos a ocuparse de la excepción. Esto se puede hacer colocando otro *throws* al método que llama al método, con el tipo de excepción que podría tirar o rodear el llamado del método con un try-catch, y de esa manera que el try-catch se encargue de manejar la excepción que podría tirar el método.

```
[acceso] [modificador] [tipo] nombreFuncion() throws TipoDeExcepcion {  
    Bloque de instrucciones  
}
```

Ejemplo:

```
// Tenemos un método que devuelve un resultado y que tira una  
// ArithmeticException  
  
public int division() throws ArithmeticException {  
    int division;  
  
    division = 20 / 0; // Si la división tira una excepción la manejará el  
                      // try/catch del llamado a la función.  
  
    return division; // Si tira una excepción el método no va a devolver ningún  
                     // resultado.  
}
```

Main

```
// Llamamos a la función, si tira una excepción va al bloque catch y  
// ejecuta el mensaje de error, sino imprime el resultado de la división.  
  
try {  
    System.out.println(division());  
} catch (ArithmeticException a) {  
    System.out.println("Error: división por cero");  
}
```

LA PALABRA THROW

Los programas escritos en Java pueden lanzar excepciones explícitamente mediante la instrucción **throw**, lo que facilita la devolución de un “código de error” al método que invocó el método que causó el error. La cláusula **throw** debe ir seguida del tipo de excepción que queremos que lance el método. Puede lanzarse cualquier tipo de excepción que implemente la interfaz **Throwable**.

Cuando se lanza una excepción usando la palabra throw, el flujo de ejecución del programa se detiene y el control se transfiere al bloque try-catch más cercano que coincide con el tipo de excepción lanzada. Si no se encuentra tal coincidencia, el controlador de excepciones predeterminado finaliza el programa. La palabra clave throw es útil para lanzar excepciones basadas en ciertas condiciones, por ejemplo, si un usuario ingresa datos incorrectos. También es útil para lanzar excepciones personalizadas específicas para un programa o aplicación.

Cuando utilicemos la palabra throw en un método, vamos a tener que agregarle la palabra throws al método con la excepción que va a tirar nuestro throw. De esa manera avisamos que cuando se llame al método hay que manejar una posible excepción.

```
throw new TipoExpcion("Mensaje de error");
```

Ejemplo:

En este método recibimos una lista y un número para agregar a dicha lista. El método contiene la palabra throws para avisar que este método puede tirar una excepción

```
public void agregarNumeroLista(List<Integer> lista, int numero) throws Exception {  
    // Validamos si la lista ya tiene el número a agregar  
    if(lista.contains(numero)){  
        // Si lo tiene tiramos un excepción de tipo Exception, poniéndole un  
        // mensaje entre los paréntesis  
        throw new Exception("El número ya está en la lista");  
    }  
    lista.add(numero); // Si no contiene el número, lo agregamos a la lista
```

Main:

```
List<Integer> lista = new ArrayList();  
// Llamamos al método dentro de un try/catch para manejar la posible  
// excepción, además le pasamos la lista al método y el número.  
try{  
    agregarNumeroLista(lista, 1);  
}catch (Exception e){  
    System.out.println(e.getMessage());  
    // Usamos el método getMessage, para obtener el mensaje que pusimos en el  
    // throw  
}
```

EJERCICIOS DE APRENDIZAJE

En este módulo vamos a empezar a manejar los errores y las excepciones de nuestro código para poder seguir trabajando sin que el código se detenga.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Inicializar un objeto de la clase Persona ejercicio 7 de la guía POO, a null y tratar de invocar el método esMayorDeEdad() a través de ese objeto. Luego, englobe el código con una cláusula try-catch para probar la nueva excepción que debe ser controlada.
2. Definir una Clase que contenga algún tipo de dato de tipo array y agregue el código para generar y capturar una excepción **ArrayIndexOutOfBoundsException** (índice de arreglo fuera de rango).
3. Defina una clase llamada DivisionNumero. En el método main utilice un Scanner para leer dos números en forma de cadena. A continuación, utilice el método parseInt() de la clase Integer, para convertir las cadenas al tipo int y guardarlas en dos variables de tipo int. Por ultimo realizar una división con los dos numeros y mostrar el resultado.
4. Todas estas operaciones puede tirar excepciones a manejar, el ingreso por teclado puede causar una excepción de tipo InputMismatchException, el método Integer.parseInt() si la cadena no puede convertirse a entero, arroja una NumberFormatException y además, al dividir un número por cero surge una ArithmeticException. Manipule todas las posibles excepciones utilizando bloques try/catch para las distintas excepciones
5. Escribir un programa en Java que juegue con el usuario a adivinar un número. La computadora debe generar un número aleatorio entre 1 y 500, y el usuario tiene que intentar adivinarlo. Para ello, cada vez que el usuario introduce un valor, la computadora debe decirle al usuario si el número que tiene que adivinar es mayor o menor que el que ha introducido el usuario. Cuando consiga adivinarlo, debe indicárselo e imprimir en pantalla el número de veces que el usuario ha intentado adivinar el número. Si el usuario introduce algo que no es un número, se debe controlar esa excepción e indicarlo por pantalla. En este último caso también se debe contar el carácter fallido como un intento.
6. Dado el método metodoA de la clase A, indique:
 - a) ¿Qué sentencias y en qué orden se ejecutan si se produce la excepción MioException?
 - b) ¿Qué sentencias y en qué orden se ejecutan si no se produce la excepción MioException?

```

class A {
    public void metodoA() {
        sentencia_a1
        sentencia_a2
        try {
            sentencia_a3
            sentencia_a4
        } catch (MioException e) {
            sentencia_a6
        }
        sentencia_a5
    }
}

```

7. Dado el método metodoB de la clase B, indique:

- ¿Qué sentencias y en qué orden se ejecutan si se produce la excepción MioException?
- ¿Qué sentencias y en qué orden se ejecutan si no se produce la excepción MioException?

```

class B {
    public void metodoB() {
        sentencia_b1
        try {
            sentencia_b2
        } catch (MioException e) {
            sentencia_b3
        }
        finally
            sentencia_b4
    }
}

```

8. Indique que se mostrará por pantalla cuando se ejecute cada una de estas clases:

```

class Uno{
    private static int metodo() {
        int valor=0;
        try {
            valor = valor+1;
            valor = valor + Integer.parseInt ("42");
            valor = valor +1;
            System.out.println("Valor final del try:" + valor) ;
        } catch (NumberFormatException e) {
            Valor = valor + Integer.parseInt("42");
            System.out.println("Valor final del catch:" + valor);
        } finally {
            valor = valor + 1;
            System.out.println("Valor final del finally: " + valor) ;
        }
        valor = valor +1;
}

```

```

        System.out.println("Valor antes del return: " + valor) ;
        return valor;
    }

    public static void main (String[] args) {
        try {
            System.out.println (metodo()) ;
        }catch(Exception e) {
            System.err.println("Excepcion en metodo() ") ;
            e.printStackTrace();
        }
    }

}

class Dos{
    private static int metodo() {
        int valor=0;
        try{
            valor = valor + 1;
            valor = valor + Integer.parseInt ("W");
            valor = valor + 1;
            System.out.println("Valor final del try: " + valor) ;
        } catch ( NumberFormatException e ) {
            valor = valor + Integer.parseInt ("42");
            System.out.println("Valor final del catch: " + valor) ;
        } finally {
            valor = valor + 1;
            System.out.println("Valor final del finally: " + valor) ;
        }
        valor = valor + 1;
        System.out.println("Valor antes del return: " + valor) ;
        return valor;
    }

    public static void main (String[] args) {
        try{
            System.out.println ( metodo ( ) ) ;
        } catch(Exception e) {
            System.err.println ( " Excepcion en metodo ( ) " ) ;
            e.printStackTrace();
        }
    }

}

class Tres{
    private static int metodo( ) {
        int valor=0;
        try{

```

```

        valor = valor + 1;
        valor = valor + Integer.parseInt ("W");
        valor = valor + 1;
        System.out.println("Valor final del try: " + valor);
    } catch(NumberFormatException e) {
        valor = valor + Integer.parseInt ("W");
        System.out.println("Valor final del catch: " + valor);
    } finally{
        valor = valor + 1;
        System.out.println("Valor final del finally:" + valor);
    }
    valor = valor + 1;
    System.out.println("Valor antes del return: " + valor) ;
    return valor;
}

public static void main (String[] args) {
    try{
        System.out.println( metodo ( ) ) ;
    } catch(Exception e) {
        System.err.println("Excepcion en metodo ( ) " ) ;
        e.printStackTrace();
    }
}
}

```

9. Dado el método metodoC de la clase C, indique:

- a) ¿Qué sentencias y en qué orden se ejecutan si se produce la excepción MioException?
- b) ¿Qué sentencias y en qué orden se ejecutan si no se produce la excepción MioException?
- c) ¿Qué sentencias y en qué orden se ejecutan si se produce la excepción TuException?

```

class C {
    void metodoC() throws TuException{
        sentencia_c1
        try {
            sentencia_c2
            sentencia_c3
        } catch (MioException e){
            sentencia_c4

        } catch (TuException e){
            sentencia_c5
            throw (e)
        }
    }
}

```

```
    finally
        sentencia_c6
    }
}
```

IMPORTANTE: A partir de la próxima guía se debe aplicar en todos los ejercicios el manejo de excepciones cada vez que sea necesario controlar una posible excepción.

EJERCICIO INTEGRADOR COMPLEMENTARIO

Este ejercicio va a requerir que utilicemos todos conocimientos previamente vistos en esta y otras guías. Estos pueden realizarse cuando hayas terminado todas las guías y tengas una buena base sobre todo lo que veníamos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con este ejercicio complementario, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. **Este ejercicio, no lleva nota y es solamente para medir nuestros conocimientos.** Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

Armadura Iron Man:

J.A.R.V.I.S. es una inteligencia artificial desarrollada por Tony Stark. Está programado para hablar con voz masculina y acento británico. Actualmente se encarga de todo lo relacionado con la información doméstica de su casa, desde los sistemas de calefacción y refrigeración hasta los Hot Rod que Stark tiene en su garage.

Tony Stark quiere adaptar a J.A.R.V.I.S. para que lo asista en el uso de sus armaduras, por lo tanto, serás el responsable de llevar adelante algunas de estas tareas.

El objetivo de **JARVIS** es que analice intensivamente toda la información de la armadura y del entorno y en base a esto tome decisiones inteligentes.

En este trabajo se deberá crear en el proyecto una clase llamada Armadura que modele toda la información y las acciones que pueden efectuarse con la Armadura de Iron Man. La armadura de Iron Man es un exoesqueleto mecánico ficticio usado por Tony Stark cuando asume la identidad de Iron Man. La primera armadura fue creada por Stark y Ho Yinsen, mientras estuvieron prisioneros.

Las armaduras de Stark se encuentran definidas por un color primario y un color secundario. Se encuentran compuestas de dos propulsored, uno en cada bota; y dos repulsores, uno en cada guante (los repulsores se utilizan también como armas). Tony los utiliza en su conjunto para volar.

La armadura tiene un nivel de resistencia, que depende del material con el que está fabricada, y se mide con un número entero cuya unidad de medida de dureza es Rockwell (https://es.wikipedia.org/wiki/Dureza_Rockwell). Todas las armaduras poseen un nivel de salud el cual se mide de 0 a 100. Además, Tony tiene un generador, el cual le sirve para salvarle la vida en cada instante de tiempo alejando las metrallas de metal de su corazón y también para alimentar de energía a la armadura. La batería a pesar de estar en el pecho de Tony, es considerada como parte de la armadura.

La armadura también posee una consola en el casco, a través de la cual JARVIS le escribe información a Iron Man. En el casco también se encuentra un sintetizador por donde JARVIS susurra cosas al oído de Tony. Cada dispositivo de la armadura de Iron Man (botas, guantes, consola y sintetizador) tienen un consumo de energía asociado.

En esta primera etapa con una armadura podremos: caminar, correr, propulsar, volar, escribir y leer.

- Al caminar la armadura hará un uso básico de las botas y se consumirá la energía establecida como consumo en la bota por el tiempo en el que se camine.
- Al correr la armadura hará un uso normal de las botas y se consumirá el doble de la energía establecida como consumo en la bota por el tiempo en el que se corra.
- Al propulsarse la armadura hará un uso intensivo de las botas utilizando el triple de la energía por el tiempo que dure la propulsión.
- Al volar la armadura hará un uso intensivo de las botas y de los guantes un uso normal consumiendo el triple de la energía establecida para las botas y el doble para los guantes.
- Al utilizar los guantes como armas el consumo se triplica durante el tiempo del disparo.
- Al utilizar las botas para caminar o correr el consumo es normal durante el tiempo que se camina o se corra.
- Cada vez que se escribe en la consola o se habla a través del sintetizador se consume lo establecido en estos dispositivos. Solo se usa en nivel básico.
- Cada vez que se efectúa una acción se llama a los métodos usar del dispositivo se le pasa el nivel de intensidad y el tiempo. El dispositivo debe retornar la energía consumida y la armadura deberá informar al generador se ha consumido esa cantidad de energía.

Modele las clases, los atributos y los métodos necesarios para poder obtener un modelo real de la armadura y del estado de la misma.

Mostrando Estado

Hacer un método que JARVIS muestre el estado de todos los dispositivos y toda la información de la Armadura.

Estado de la Batería

Hacer un método para que JARVIS informe el estado de la batería en porcentaje a través de la consola. Poner como carga máxima del reactor el mayor float posible. Ejecutar varias acciones y mostrar el estado de la misma.

Mostrar Información del Reactor

Hacer un método para que JARVIS informe el estado del reactor en otras dos unidades de medida. Hay veces en las que Tony tiene pretensiones extrañas. Buscar en Wikipedia la tabla de transformaciones.

Sufriendo Daños

A veces los dispositivos de la armadura sufren daños para esto cada dispositivo contiene un atributo público que dice si el dispositivo se encuentra dañado o no. Al utilizar un dispositivo existe un 30% de posibilidades de que se dañe.

La armadura solo podrá utilizar dispositivos que no se encuentren dañados.

Modifique las clases que sean necesarias para llevar adelante este comportamiento.

Reparando Daños

Hay veces que se puede reparar los daños de un dispositivo, en general es el 40% de las veces que se puede hacer. Utilizar la clase Random para modelar este comportamiento. En caso de estar dentro de la probabilidad (es decir probabilidad menor o igual al 40%) marcar el dispositivo como sano. Si no dejarlo dañado.

Revisando Dispositivos

Los dispositivos son revisados por JARVIS para ver si se encuentran dañados. En caso de encontrar un dispositivo dañado se debe intentar arreglarlo de manera insistente. Para esos intentos hay un 30% de posibilidades de que el dispositivo quede destruido, pero se deberá intentar arreglarlo hasta que lo repare, o bien hasta que quede destruido.

Hacer un método llamado revisar dispositivos que efectúe lo anteriormente descrito, el mecanismo insistente debe efectuarlo con un bucle do while.

Radar Versión 1.0

JARVIS posee también incorporado un sistema que usa ondas electromagnéticas para medir distancias, altitudes, ubicaciones de objetos estáticos o móviles como aeronaves barcos, vehículos motorizados, formaciones meteorológicas y por su puesto enemigos de otro planeta.

Su funcionamiento se basa en emitir un impulso de radio, que se refleja en el objetivo y se recibe típicamente en la misma posición del emisor.

Las ubicaciones de los objetos están dadas por las coordenadas X, Y y Z. Los objetos pueden ser marcados o no como hostiles. JARVIS también puede detectar la resistencia del objeto, y puede detectar hasta 10 objetos de manera simultánea.

JARVIS puede calcular la distancia a la que se encuentra cada uno de los objetos, para esto siempre considera que la armadura se encuentra situada en la coordenada (0,0,0).

Hacer un método que informen a qué distancia se encuentra cada uno de los enemigos. Usar la clase Math de Java.

Simulador

Hacer un método en JARVIS que agregue en radar objetos, hacer que la resistencia, las coordenadas y la hostilidad sean aleatorios utilizando la clase random. Utilizar la clase Random.

¿Qué ocurre si quiero añadir más de 10 objetos?

¿Qué ocurre si cuando llevo 8 enemigos aumento la capacidad del vector?

Destruyendo Enemigos

Desarrollar un método para que JARVIS que analice todos los objetos del radar y si son hostiles que les dispare. El alcance de los guantes es de 5000 metros, si el objeto se encuentra fuera de ese rango no dispara.

JARVIS al detectar un enemigo lo atacará hasta destruirlo, la potencia del disparo es inversamente proporcional a la distancia al a que se encuentra el enemigo y se descontará de la resistencia del enemigo. El enemigo se considera destruido si su resistencia es menor o igual a cero.

JARVIS solo podrá disparar si el dispositivo está sano y si su nivel de batería lo permite. Si tiene los dos guantes sanos podrá disparar con ambos guantes haciendo más daño. Resolver utilizando un for each para recorrer el arreglo y un while para destruir al enemigo.

Acciones Evasivas

Desarrollamos un método para que JARVIS que analice todos los objetos del radar y si son hostiles que les dispare. Modificar ese método para que si el nivel de batería es menor al 10% se corten los ataques y se vuelve lo suficientemente lejos para que el enemigo no nos ataque. Deberíamos alejarnos por lo menos 10 km enemigo. Tener en cuenta que la velocidad de vuelo promedio es de 300 km / hora.

Bibliografía

Información sacada de las paginas:

- <https://www.oracle.com/ar/database/what-is-a-relational-database/>

- <https://www.geeksforgeeks.org/sql-tutorial/>

CURSO DE PROGRAMACIÓN FULL STACK

BASES DE DATOS CON MYSQL



GUÍA DE BASE DE DATOS

BASE DE DATOS

Una base de datos es una colección organizada de información estructurada, o datos, típicamente almacenados electrónicamente en un sistema de computadora. Una base de datos es usualmente controlada por un sistema de gestión de base de datos (DBMS). En conjunto, los datos y el DBMS, junto con las aplicaciones que están asociados con ellos, se conocen como un sistema de base de datos, que a menudo se reducen a solo base de datos.

Los datos dentro de los tipos más comunes de bases de datos en funcionamiento hoy en día se modelan típicamente en filas y columnas en una serie de tablas para que el procesamiento y la consulta de datos sean eficientes. Luego se puede acceder, administrar, modificar, actualizar, controlar y organizar fácilmente los datos. La mayoría de las bases de datos utilizan lenguaje de consulta estructurado (SQL) para escribir y consultar datos.

¿QUÉ ES UNA TABLA EN BASE DE DATOS?

Una tabla de base de datos es similar en apariencia a una hoja de cálculo en cuanto a que los datos se almacenan en filas y columnas.

Para aprovechar al máximo la flexibilidad de una base de datos, los datos deben organizarse en tablas para que no se produzcan redundancias. Por ejemplo, si quiere almacenar información sobre los empleados, cada empleado debe especificarse solo una vez en la tabla que está configurada para los datos de los empleados. Los datos sobre los productos se almacenarán en su propia tabla y los datos sobre las sucursales se almacenarán en otra tabla. Este proceso se denomina normalización.

Cada fila de una tabla se denomina registro. En los registros se almacena información. Cada registro está formado por uno o varios campos. Los campos equivalen a las columnas de la tabla. Por ejemplo, puede tener una tabla llamada "Empleados" donde cada registro (fila) contiene información sobre un empleado distinto y cada campo (columna) contiene otro tipo de información como nombre, apellido, dirección, etc. Los campos deben designarse como un determinado tipo de datos, ya sea texto, fecha u hora, número o algún otro tipo. Vamos a ver una tabla de Empleado:

Empleado		
Nombre	Apellido	Edad
Agustín	Cocco	24
Martin	Bullón	21
Mariano	Fernández	18
Jerónimo	Gómez	23

Nombre de la tabla.

Nombre de las columnas: Nombre, Apellido y Edad

Fila / Registro:
Cada una de las
Filas tiene un empleado distinto.

¿CUÁL ES LA DIFERENCIA ENTRE UNA BASE DE DATOS Y UNA HOJA DE CÁLCULO?

Las bases de datos y las hojas de cálculo (como Microsoft Excel) son dos formas convenientes de almacenar información. Las principales diferencias entre las dos son:

- Cómo se almacenan y manipulan los datos
- Quién puede acceder a los datos
- Cuántos datos se pueden almacenar

Las hojas de cálculo se diseñaron originalmente para un usuario, y sus características lo reflejan. Son muy buenas para un solo usuario o un pequeño número de usuarios que no necesitan manipular una gran cantidad de datos complicados. Las bases de datos, por otro lado, están diseñadas para contener colecciones mucho más grandes de información organizada, cantidades masivas en ocasiones. Las bases de datos permiten a múltiples usuarios al mismo tiempo acceder y consultar los datos de forma rápida y segura utilizando una lógica y un lenguaje altamente complejos.

¿POR QUÉ INTERESA USAR UNA BASE DE DATOS?

- Mayor independencia. Los datos son independientes de las aplicaciones que los usan, así como de los usuarios.
- Mayor disponibilidad. Se facilita el acceso a los datos desde contextos, aplicaciones y medios distintos, haciéndolos útiles para un mayor número de usuarios.
- Mayor seguridad (protección de los datos). Por ejemplo, resulta más fácil replicar una base de datos para mantener una copia de seguridad que hacerlo con un conjunto de ficheros almacenados de forma no estructurada. Además, al estar centralizado el acceso a los datos, existe una verdadera sincronización de todo el trabajo que se haya podido hacer sobre estos (modificaciones), con lo que esa copia de seguridad servirá a todos los usuarios.
- Menor redundancia. Un mismo dato no se encuentra almacenado en múltiples archivos o con múltiples esquemas distintos, sino en una única instancia en la base de datos. Esto redonda en menor volumen de datos y mayor rapidez de acceso.
- Mayor eficiencia en la captura, codificación y entrada de datos.

CLASIFICACIÓN DE LAS BASES DE DATOS

Hay muchos tipos diferentes de bases de datos. La mejor base de datos para una organización específica depende de cómo la organización pretende utilizar los datos.

- Bases de datos relacionales: Los elementos de una base de datos relacional se organizan como un conjunto de tablas con columnas y filas. La tecnología de base de datos relacional proporciona la manera más eficiente y flexible de acceder a información estructurada. En la actualidad se usa de forma mayoritaria las bases de datos relacionales.

Nota: este es el tipo de base de datos que vamos a trabajar y sobre el que vamos a profundizar.

- Bases de datos orientadas a objetos: La información en una base de datos orientada a objetos se representa en forma de objetos, como en la programación orientada a objetos.
- Bases de datos NoSQL: Una NoSQL, o una base de datos no relacional, permite que los datos no estructurados y semiestructurados se almacenen y manipulen, a diferencia de una base de datos relacional, que define cómo deben componerse todos los datos insertados en la base de datos. Las bases de datos NoSQL se hicieron populares a medida que las aplicaciones web se hacían más comunes y más complejas.

BASE DE DATOS RELACIONALES

Las bases de datos relacionales se basan en el modelo relacional, una forma intuitiva y directa de representar datos en tablas. En una base de datos relacional, cada fila de la tabla es un registro con un ID único llamado clave. Las columnas de la tabla contienen atributos de los datos, y cada registro generalmente tiene un valor para cada atributo, lo que facilita el establecimiento de las relaciones entre los puntos de datos.

¿QUÉ ES UN MODELO DE BASE DE DATOS?

Un modelo de base de datos es la estructura lógica que adopta la base de datos, incluyendo las relaciones y limitaciones que determinan cómo se almacenan y organizan y cómo se accede a los datos. Así mismo, un modelo de base de datos también define qué tipo de operaciones se pueden realizar con los datos, es decir, que también determina cómo se manipulan los mismos, proporcionando también la base sobre la que se diseña el lenguaje de consultas.

En general, prácticamente todos los modelos de base de datos pueden representarse a través de un diagrama de base de datos (en esta entrada veremos algunos).

MODELO RELACIONAL

El modelo relacional, para el modelado y la gestión de bases de datos, es un modelo de datos basado en la lógica de predicados y en la teoría de conjuntos.

Su idea fundamental es el uso de relaciones. Estas relaciones podrían considerarse en forma lógica como conjuntos de datos llamados tuplas. Pese a que esta es la teoría de las bases de datos relacionales creadas por Codd, la mayoría de las veces se conceptualiza de una manera más fácil de imaginar, pensando en cada relación como si fuese una tabla que está compuesta por registros (cada fila de la tabla sería un registro o "tupla") y columnas (también llamadas "campos").

Es el modelo más utilizado en la actualidad para modelar problemas reales y administrar datos dinámicamente.

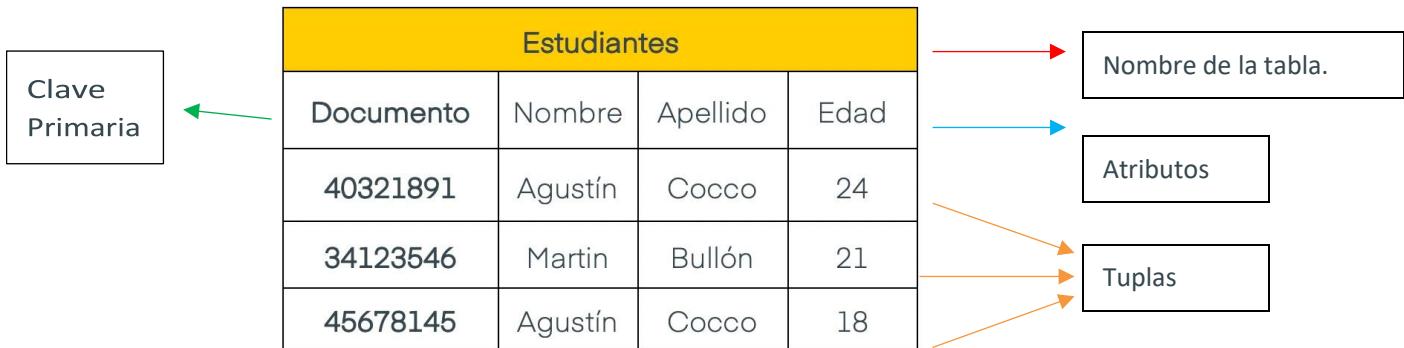
Este modelo está basado en que todos los datos están almacenados en tablas (entidades/relaciones) y cada una de estas es un conjunto de datos, por tanto una base de datos es un conjunto de relaciones. Las tablas están formadas por filas, también llamadas tuplas, donde se describen los elementos que configuran la tabla (es decir, los elementos de la relación establecida por la tabla), columnas o campos, con los atributos y valores correspondientes, y el dominio, concepto que agrupa a todos los valores que pueden figurar en cada columna.

CARACTERÍSTICAS DEL MODELO RELACIONAL

- Los datos de cualquier columna son de un solo tipo.
- Cada columna posee un nombre único.
- El orden de las columnas no es de importancia para la tabla.
- Las columnas de una relación se conocen como atributos.
- Cada atributo tiene un dominio.
- No se permiten filas repetidas mediante la clave primaria.
- No existen 2 filas en la tabla que sean idénticas.
- Los valores de los atributos son atómicos: en cada tupla, cada atributo (columna) toma un solo valor. Se dice que las relaciones están normalizadas.

- No hay dos atributos que se llamen igual.
- El orden de los atributos no importa: los atributos no están ordenados.

ELEMENTOS DEL MODELO RELACIONAL



ATRIBUTOS

Un Atributo en el Modelo Relacional representa una propiedad que posee esa tabla y equivale al atributo del Modelo E-R.

Se corresponde con la idea de campo o columna.

En el caso de que sean varios los atributos de una misma tabla, definidos sobre el mismo dominio, habrá que darles nombres distintos, ya que una tabla no puede tener dos atributos con el mismo nombre.

Por ejemplo, la información de los estudiantes de un curso se representa mediante la tabla Estudiantes, que tiene columnas para los atributos id_estudiantes, nombre, apellido y edad.

DOMINIO

El dominio dentro de la estructura del Modelo Relacional es el conjunto de valores que puede tomar un atributo.

Estudiantes			
Documento	Nombre	Apellido	Edad
Valor numérico	Cadena de texto	Cadena de texto	Valor numérico
Valor numérico	Cadena de texto	Cadena de texto	Valor numérico
Valor numérico	Cadena de texto	Cadena de texto	Valor numérico

- Un dominio contiene todos los posibles valores que puede tomar un determinado atributo. Dos atributos distintos pueden tener el mismo dominio.

- Un domino es un conjunto finito de valores del mismo tipo. Distintos tipos de dominios son: enteros, cadenas de texto, fecha, etc.

TUPLAS

Filas de una tabla que contiene valores para cada uno de los atributos (equivale a los registros). Ejemplo: 34563, José, Martínez, 19, Masculino. Representa un objeto único de datos implícitamente estructurados en una tabla. Un registro es un conjunto de campos que contienen los datos que pertenecen a una misma entidad.

CLAVES

Campo cuyo valor es único para cada registro. Primaria, identifica una tabla, y Foránea. Ejemplo: id estudiante.

CLAVE PRIMARIA

Se denomina clave primaria o identificador único o llave principal a uno o más atributos que identifican únicamente cada instancia de un registro; es conocido también como "clave candidata".

Una base de datos relacional está diseñada para imponer la exclusividad de las claves primarias permitiendo que haya sólo una fila con un valor de clave primaria específico en una tabla. Es decir, nunca puede existir dos instancias de un registro con el mismo valor de su atributo primario .

El o los atributos identificadores se señalan con el símbolo "@"(arroba), o de lo contrario con la sigla PK (clave primaria).

Para mejorar el desempeño de la base de datos se recomienda utilizar claves primarias numéricas; por lo tanto, si una tabla no posee un atributo identificador numérico, se debería agregar un atributo, comúnmente llamado id (abreviación de identificador) seguido por el nombre de la tabla. Por ejemplo: id_estudiante o si existe un atributo propio de una tabla que no se va a repetir puede ser ese, como documento.

Estudiantes			
Documento	Nombre	Apellido	Edad
40321891	Agustín	Cocco	24
34123546	Martín	Bullón	21
45678145	Agustín	Cocco	18

En esta tabla Estudiantes, tendremos dos estudiantes con el mismo nombre y apellido, pero con distintos identificadores y distintas edades, por lo que no se consideran datos duplicados. Si nosotros nos basáramos en el nombre para evitar datos duplicados, no podríamos ingresar dos alumnos con el mismo nombre, esta es otra de las ventajas del identificador único o clave primaria.

CLAVE FORANEA

Una clave foránea o llave foránea es una columna o un conjunto de columnas en una tabla cuyos valores corresponden a los valores de la clave primaria de otra tabla. A veces, esto también se denomina clave de referencia. Para poder añadir una fila con un valor de clave foránea específico, debe existir una fila en la tabla relacionada con el mismo tipo de valor de clave primaria.

La relación entre 2 tablas coincide con la clave primaria en una de las tablas con una clave foránea en la segunda tabla. Por ejemplo, si tenemos las tablas profesor y curso, para relacionarlas, tendríamos una clave foránea de la clave primaria de un profesor como columna en la tabla curso. De esta manera se dice que ese profesor pertenece a ese curso.

Profesor				Curso			
id_profesor	Nombre	Apellido	Edad	id_curso	Nombre	Costo	id_profesor
1	Agustín	Cocco	24	1	Curso de Programación	500	1

En este ejemplo la tabla Curso (tabla de la derecha), tiene una columna llamada id_profesor, esta columna es la que va a tener las claves foráneas y la que va mostrar que hay una relación entre las dos tablas. En este ejemplo, la columna id_profesor, tiene el id 1 del profesor Agustín, por lo que, sería correcto decir que Agustín es el Profesor del Curso de Programación.

RELACIONES

Uno de los aspectos fundamentales de las bases de datos relacionales son precisamente las relaciones. En pocas palabras, una “relación” es una asociación que se crea entre tablas, con el fin de vincularlas y garantizar la integridad referencial de sus datos.

Una relación es la abstracción de un conjunto de asociaciones que existen entre las tablas de dos tuplas, por ejemplo, existe una relación entre Película (tabla Películas) y PaísDeOrigen (tabla PaisesDeOrigen).

Para que una relación entre dos tablas exista, la tabla que deseas relacionar debe poseer una clave primaria o identificador único, mientras que la tabla donde estará el lado dependiente de la relación debe poseer una clave foránea o llave foránea de esa clave primaria.

- Las relaciones tienen sentido bidireccional.
- Las relaciones existen ya que las entidades representan aspectos del mundo real y en este mundo los componentes no están aislados, sino que se relacionan entre sí; es por esto que es necesario que existan las relaciones entre las entidades.

TIPOS DE RELACIONES

Las bases de datos relacionales tienen diversos tipos de relaciones que podemos utilizar para vincular nuestras tablas.

Este vínculo va a depender de la cantidad de ocurrencias que tiene un registro o fila de una tabla dentro de otra tabla (esto se conoce como cardinalidad).

Veamos los tipos de vínculos o relaciones:

Relaciones uno a uno

Se presentan cuando un registro de una tabla sólo está relacionado con un registro de otra tabla, y viceversa.

Por ejemplo, supongamos que nuestros empleados deben almacenar su información de contacto. Para este caso, pudiésemos leer la relación de esta manera:

Un empleado tiene una sola información de contacto. y

Una información de contacto pertenece a un solo empleado.

Dado que la información de contacto es la que depende principalmente del empleado, es en ella donde existirá la clave foránea para representar el vínculo.

Relaciones uno a muchos / muchos a uno

Esta relación es un poco más compleja que la anterior, así que vamos a usar las tablas A y B para explicarla.

Una relación de uno a muchos se presenta cuando un registro de la tabla A está relacionado con ninguno o muchos registros de tabla B, pero este registro en la tabla B solo está relacionado con un registro de la tabla A. Veamos un ejemplo de esto.

Supongamos que tenemos ciudades en las cuales viven nuestras personas, pero cada persona solo puede pertenecer a una ciudad. Para este caso, pudiésemos leer la relación de esta manera:

Una ciudad tiene muchas (o ningún) personas. y

Una persona vive en una sola ciudad.

Dado que la persona es el que necesita de la ciudad, es en él donde existirá la clave foránea para representar el vínculo.

Relaciones muchos a muchos

Estas son las relaciones más complejas, se presentan cuando muchos registros de una tabla se relacionan con muchos registros de otra tabla. Vamos a verlo en un ejemplo.

Supongamos que nuestros empleados trabajan turnos. Por ejemplo, Juan trabaja en el turno mañana y de la noche, pero en el turno de la mañana trabajan Juan, Pedro y María.

Para este tipo de relación se crea una tabla intermedia conocida como tabla asociativa. Por convención, el nombre de esta tabla debe estar formado por el nombre de las tablas participantes (en singular y en orden alfabético) separados por un guion bajo (_). Esta tabla está compuesta por las claves primarias de las tablas que se relacionan con ella, así se logra que la relación sea de uno a muchos, en los extremos, de modo tal que la relación se lea:

Un empleado trabaja en muchos turnos y

Un turno tiene muchos empleados.

La tabla recibirá las llaves primarias como llaves foráneas.

Supongamos que tenemos la siguiente relación. El ticket de las compras del supermercado, un cliente puede comprar varios productos y al mismo tiempo un producto puede ser comprado por varios clientes. Las tablas se verían así:

Producto	
Id	Nombre
1	Agua
2	Azúcar
3	Pan

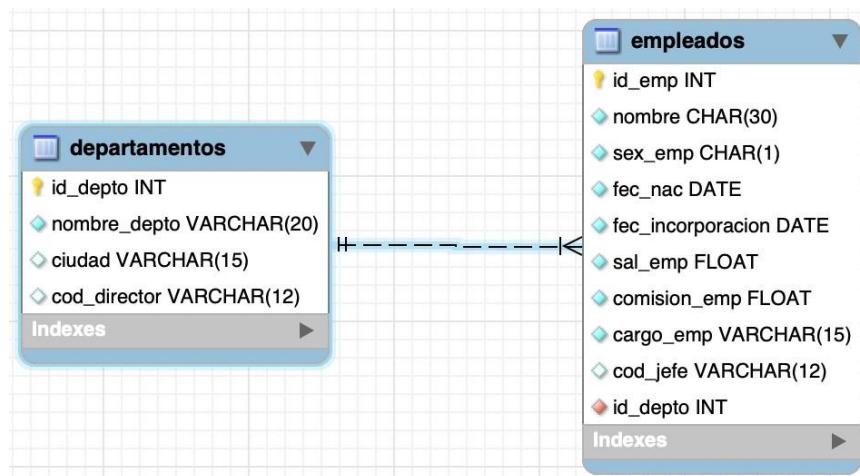
Ticket	
Id	Descripcion
1	Uno
2	Dos

Tabla Intermedia:

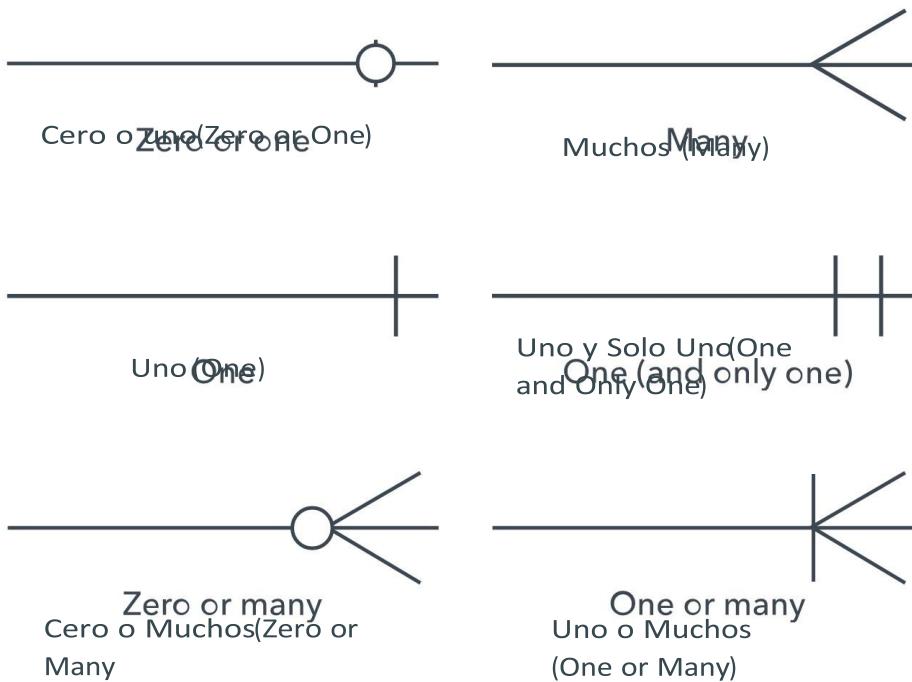
Producto_Ticket	
Id_Producto	Id_Ticket
1	1
3	1
2	2

DIAGRAMAS EER

Los diagramas de relación de entidades (EER) son representaciones visuales de bases de datos que muestran cómo los elementos dentro de una base de datos están relacionados entre sí. Un ERD se compone de dos tipos de objetos: entidades y relaciones. Las entidades van a ser lo que nosotros conocemos como tablas, y las relaciones tienen finales de línea especiales llamados cardinalidades que describen cómo dos elementos de la base de datos interactúan entre sí.

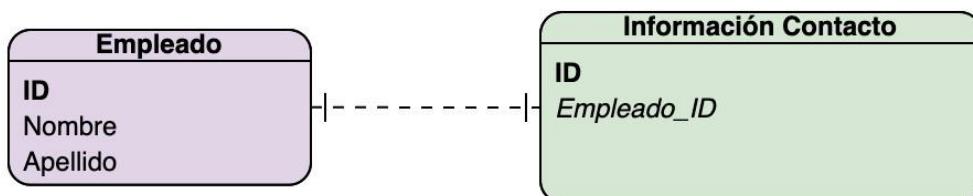


Estos diagramas nos van a servir para representar las relaciones previamente mencionadas, para ello existen las siguientes líneas que unen las tablas entre sí:

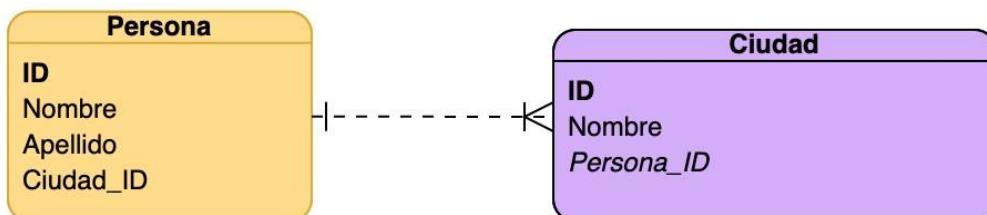


Teniendo en cuenta estas líneas vamos a mostrar como serían las relaciones previamente vistas:

Relaciones uno a uno



Relaciones uno a muchos / muchos a uno



Relaciones muchos a muchos



¿QUÉ ES EL SOFTWARE DE BASE DE DATOS?

Para poder trabajar con base de datos, tablas, sus columnas, filas, relaciones, con el modelo relacional, etc. Tenemos que utilizar un software de base de datos.

El software de base de datos se utiliza para crear, editar y mantener archivos y registros de bases de datos, lo que facilita la creación de archivos y registros, la entrada de datos, la edición de datos, las actualizaciones y los informes. El software también se encarga del almacenamiento de datos, las copias de seguridad y los informes, el control de acceso múltiple y la seguridad. La sólida seguridad de las bases de datos es especialmente importante hoy en día, ya que el robo de información se vuelve más frecuente. En ocasiones, el software de base de datos también se denomina "sistema de administración de bases de datos" (DBMS).

El software de base de datos simplifica la gestión de la información al permitirles a los usuarios almacenar datos en una forma estructurada y luego, acceder a ellos. Por lo general, tiene una interfaz gráfica para ayudar a crear y administrar los datos y, en algunos casos, los usuarios pueden crear sus propias bases de datos mediante el software de base de datos.

¿QUÉ ES UN SISTEMA DE ADMINISTRACIÓN DE BASE DE DATOS (DBMS)?

Una base de datos generalmente requiere un programa completo de software de base de datos, que se conoce como sistema de administración de bases de datos (DBMS). Un DBMS sirve como una interfaz entre la base de datos y sus usuarios o programas finales, lo que permite a los usuarios recuperar, actualizar y administrar cómo se organiza y optimiza la información. Un DBMS también facilita la supervisión y el control de las bases de datos, lo que permite una variedad de operaciones administrativas, como la supervisión del rendimiento, el ajuste, las copias de seguridad y la recuperación.

Algunos ejemplos de software de bases de datos o DBMS populares incluyen MySQL, Microsoft Access, Microsoft SQL Server, FileMaker Pro, Oracle Database y dBASE.

En este curso vamos a utilizar el software de bases de datos MySQL.

¿QUÉ ES UNA BASE DE DATOS DE MYSQL?

MySQL es un sistema de gestión de bases de datos relacionales de código abierto basado en SQL. Fue diseñado y optimizado para aplicaciones web y puede ejecutarse en cualquier plataforma. A medida que surgían nuevos y diferentes requisitos con Internet, MySQL se convirtió en la plataforma elegida por los desarrolladores web y las aplicaciones basadas en la web. Debido a que está diseñada para procesar millones de consultas y miles de transacciones. La flexibilidad bajo demanda es la característica principal de MySQL.

LENGUAJE DE CONSULTA ESTRUCTURADO SQL

SQL es un acrónimo en inglés para Structured Query Language, un Lenguaje de Consulta Estructurado. Un tipo de lenguaje de programación que te permite acceder, manipular y descargar datos de una base de datos mediante comandos, mejor conocido como consultas (Querys).

Tiene capacidad de hacer cálculos avanzados y álgebra. Es utilizado en la mayoría de empresas que almacenan datos en una base de datos. Ha sido y sigue siendo el lenguaje de programación más usado para bases de datos relacionales.

El lenguaje SQL también se usa para controlar el acceso a datos y para la creación y modificación de esquemas de Base de datos. SQL utiliza los términos tabla, fila y columna para los términos relación, tupla y atributo del modelo relacional formal, respectivamente. Por lo tanto, es posible utilizar todos estos términos indistintamente.

Existen dos tipos de comandos SQL:

1. Lenguaje de Definición de Datos (DDL): permite crear y definir nuevas bases de datos, campos e índices.
 - CREATE: Crea nuevas tablas, campos e índices.
 - DROP: Elimina tablas e índices.
 - ALTER: Modifica las tablas agregando campos o cambiando la definición de los campos.
2. Lenguaje de Manipulación de Datos (DML): permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos.
 - SELECT: Consulta registros de la base de datos que satisfagan un criterio determinado.
 - INSERT: Carga lotes de datos en la base de datos en una única operación.
 - UPDATE: Modifica los valores de los campos y registros especificados.
 - DELETE: Elimina registros de una tabla de una base de datos.

Pasos para Implementar una Base de Datos:

PASO	Descripción
1	Definir en el disco duro, el área física que contendrá las tablas de la base de datos. Sentencia SQL → CREATE DATABASE
2	Crear las diferentes tablas de la base de datos. Sentencia SQL → CREATE TABLE
3	Insertar en las filas los datos a las diferentes tablas, sin violar la integridad de los datos. Sentencia SQL → INSERT INTO
4	Actualizar los datos que cambien con el tiempo en las diferentes tablas. Sentencia SQL → UPDATE
5	Eliminar las filas que ya no se requieran en las diferentes tablas. Sentencia SQL → DELETE
6	Realizar las consultas deseadas a las tablas de la base de datos a través de la poderosa sentencia de consultas del SQL, llamada SELECT

7	Dar nombre a las consultas, elaboradas en el paso No.6 cuando se requiera ocultar el diseño y columnas de las tablas a través de la creación de vistas lógicas. Sentencia SQL → CREATE VIEW
---	---

CONSULTAS SQL

Estas son las consultas que vamos a escribir en nuestro software de base de datos para crear, actualizar, borrar, acceder y manipular información de nuestra base de datos.

CONSULTAS DE CREACIÓN

1. CREATE DATABASE

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] nombre_base_datos
```

- Esta sentencia sirve para crear una base de datos con un nombre específico.
- Para poder crear una base de datos, el usuario que la crea debe tener privilegios de creación asignados.
- IF NOT EXISTS significa: SI NO EXISTE, por lo tanto, esto es útil para validar que la base de datos sea creada en caso de que no exista, si la base de datos existe y se ejecuta esta sentencia, se genera un error.
- CREATE SCHEMA o CREATE DATABASE son sinónimos.

2. CREATE TABLE

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] nombre_de_tabla ( campo1
    tipo dato [NULL/NOT NULL] | CHECK (expresiónLógica) | [ DEFAULT
    expresiónConstante],
    campo2 tipo dato [NULL/NOT NULL] | CHECK (expresiónLógica) | [
    DEFAULT expresiónConstante ], campo-N,
    PRIMARY KEY(campo_llave),
    FOREIGN KEY (campo_llave) REFERENCES tabla2 (campo_llave-tabla2))
```

Ligaduras

Tipo: Integridad de Dominio o Columna

Especifica un conjunto de valores que son válidos a ingresar sobre una columna específica para una tabla de la base de datos. Esta integridad se verifica a través de una la validación de los valores de datos que se ingresan y el tipo de los datos a introducir (numérico, alfanumérico, alfabético, etc.).

- DEFAULT: Esta restricción asigna un valor específico a una columna cuando el valor para ello no haya sido explícitamente proporcionado para tal columna en una sentencia "INSERT" o de adición de un nuevo registro en la tabla.
- CHECK: Especifica los valores de datos que el DBMS acepta le sean ingresados para una columna.
- REFERENCES: Especifica los valores de datos que son aceptables para actualizar una columna y que están basados en valores de datos localizados en una columna de otra tabla.

Tipo: Integridad de Entidad o Tabla

Especifica que, en una tabla o entidad, todas sus filas tengan un identificador único que diferencie a una fila de otra y también que se establezcan columnas cuyo contenido es un valor único que las hace llaves candidatas para un futuro como, por ejemplo:
número de cédula, número de seguro social o cuenta de email.

- PRIMARY KEY: Este tipo de restricción se aplica a todas las filas permitiendo que exista un identificador, que se conoce como llave primaria y que se asegura que los usuarios no introduzcan valores duplicados. Además, asegura que se cree un índice para mejorar el desempeño. Los valores nulos no están permitidos para este tipo de restricción.
- UNIQUE: Con esta restricción se previene la duplicación de valores en columnas que tienen valor único y que no son llave primaria pero que pueden ser una llave alternativa o candidata para el futuro. Asegura que se cree (Por parte del DBMS) un índice para mejorar el desempeño. Y al igual que las llaves primarias, no se le está permitido que se introduzcan valores nulos.

Tipo: Integridad Referencial

La Integridad Referencial asegura que las relaciones que existe entre llave primaria (en la tabla referenciada) y la llave foránea (en las tablas referenciantes) serán siempre mantenidas. Una fila o registro en la tabla referenciada (tabla donde reside la llave primaria) no puede ser borrada o su llave primaria cambiada si existe una fila o registro con una llave foránea (en la tabla referenciante) que se refiere a esa llave primaria.

- FOREIGN KEY: En esta restricción se define una llave foránea, una columna o combinación de columnas en las cuales su valor debe corresponder al valor de la llave primaria en la misma u en otra tabla.

CONSULTAS PARA ACTUALIZAR Y BORRAR

Eliminación de Tablas:

La sentencia para eliminar una tabla y por ende todos los objetos asociados con esa tabla es DROP TABLE, donde r es el nombre de una tabla existente.

`DROP TABLE r`

Modificación de Tablas

Después que una tabla ha sido utilizada durante algún tiempo, los usuarios suelen descubrir que desean almacenar información adicional con respecto a las tablas. La sentencia ALTER TABLE se utiliza sobre tablas que ya poseen desde cientos a miles de filas por ser tablas de un sistema de

Base de Datos que ya está en producción.

`ALTER TABLE nombre_tabla acción`

Siendo acción una de las siguientes:

- RENAME TO nuevo_nombre
- ADD [COLUMN] nombre_atributo definición_atributo

- DROP [COLUMN] nombre_atributo
 - MODIFY nombre_atributo definición_atributo
 - CHANGE nombre_atributo nuevo_nombre nueva_definición
- ALTER COLUMN nombre_atributo nuevo_nombre nueva_definición

Los cambios que se pueden realizar con la sentencia SQL ALTER TABLE son:

- Añadir una definición de columna a una tabla. Puede crearse con valores nulos o con valores.
- Eliminar una columna de la tabla. Pero antes de su eliminación deben ser eliminados por ALTER TABLE todas las restricciones que estén definidas sobre esta columna.
- Eliminar la definición de: llave primaria, foránea o restricciones de ligaduras de integridad (check), existentes para una tabla. Esta acción no elimina a la columna con sus valores, ella permanece tal cual como está, solo se elimina su definición.
- Definir una llave primaria para una tabla. La columna(s) a la cual se le dará esta responsabilidad debe contener previamente valores únicos por fila.
- Definir una nueva llave foránea para una tabla. La columna a definir como llave foránea debe contener previamente valores que corresponden a la llave primaria de otra tabla.

3. INSERT INTO

En su formato más sencillo, INSERT se utiliza para añadir una sola fila a una tabla. Debemos especificar el nombre de la tabla y una lista de valores para la fila. Los valores deben suministrarse en el mismo orden en el que se especificaron los atributos correspondientes en el comando CREATE TABLE.

```
INSERT INTO nombre_tabla (columna1, columna2, columna3,...) VALUES  
(valor, valor2, valor3,...);
```

4. UPDATE

El comando UPDATE se utiliza para modificar los valores de atributo de una o más filas seleccionadas. Una cláusula WHERE en el comando UPDATE selecciona las filas de una tabla que se van a modificar. La sentencia UPDATE tiene la siguiente forma:

```
UPDATE nombre_tabla  
SET nombre_columna1 = valor1,  
nombre_columna2 = valor2, [ORDER  
BY ...] [WHERE condicion]
```

5. DELETE

El comando DELETE elimina filas de una tabla. Incluye una cláusula WHERE, para seleccionar las filas que se van a eliminar.

- Las filas se eliminan explícitamente sólo de una tabla a la vez. Sin embargo, la eliminación se puede propagar a filas de otras tablas si se han especificado opciones de acciones referenciales en las restricciones de integridad referencial del DDL.

- En función del número de filas seleccionadas por la condición de la cláusula WHERE, ninguna, una o varias filas pueden ser eliminadas por un solo comando DELETE. La ausencia de una cláusula WHERE significa que se borrarán todas las filas de la relación; sin embargo, la tabla permanece en la base de datos, pero vacía. Debemos utilizar el comando DROP TABLE para eliminar la definición de la tabla.

```
DELETE FROM nombre_tabla [WHERE condicion] [ORDER BY ...] [LIMIT cantidad_filas]
```

CONSULTAS PARA ACCEDER Y MANIPULAR INFORMACIÓN

6. SELECT

La sentencia SELECT es muy poderosa y ampliamente rica en sus cláusulas y variantes permitiendo la capacidad de atender en poco tiempo a consultas complejas sobre la base de datos. Está en el especialista desarrollador de aplicaciones conocerlo a profundidad para explotar las bondades y virtudes.

Se usa para listar las columnas de las tablas que se desean ver en el resultado de una consulta. Además de las columnas se pueden listar columnas a calcular por el SQL cuando actúe la sentencia. Esta cláusula no puede omitirse.

La sentencia SELECT, obtiene y nos permite mostrar filas de la base de datos, también permite realizar la selección de una o varias filas o columnas de una o varias tablas. Para seleccionar la tabla de la que queremos obtener dichas filas vamos a utilizar la sentencia FROM.

La sentencia FROM lista las tablas de donde se listarán las columnas enunciadas en el SELECT. Esta cláusula no puede omitirse.

```
SELECT nombres de las columnas FROM tablaOrigen;
```

SELECT nombre, apellido FROM Alumnos; Teniendo la siguiente tabla de alumnos:

Alumnos			
Id	Nombre	Apellido	Edad
1	Agustín	Cocco	24
2	Martin	Bullón	21

El resultado que mostraría sería:

Alumnos	
Nombre	Apellido
Agustín	Cocco
Martin	Bullón

Para mostrar todos los datos de una tabla usamos el símbolo (*). Esto nos mostraría la primera tabla.

```
SELECT * FROM Alumnos;
```

También en las consultas SELECT podemos hacer operaciones matemáticas entre los datos numéricos de las tablas que elijamos. Usualmente ponemos estas operaciones entre paréntesis para separar la operación del resto de la consulta.

```
SELECT nombre,(salario+comision) FROM Empleados;
```

Teniendo la siguiente tabla de Empleados:

Empleados			
Id	Nombre	Salario	Comisión
1	Agustín	5000	300
2	Martin	2000	250

El resultado que mostraría sería:

Empleados	
Nombre	(salario+comision)
Agustín	5300
Martin	2250

En este ejemplo hacemos una suma pero podemos hacer una resta (-), una multiplicación (*) y una división (/), también podemos poner agregarle números a nuestras operaciones.

```
SELECT nombre,(salario + comisión - 200) FROM Empleados;
```

Ahora el resultado que mostraría sería:

Empleados	
Nombre	(salario+comisión-200)
Agustín	5100
Martin	2050

A la consulta SELECT le podemos sumar cláusulas que van a alterar el resultado de filas que obtenga el SELECT, esto nos puede servir para traer ciertas filas y evitar algunas que no queremos mostrar.

CLÁUSULAS:

7. SELECT DISTINCT

El SELECT DISTINCT se utiliza cuando queremos traer solo registros diferentes. En las tablas a veces pueden haber valor repetidos, para evitarlos usamos esta sentencia.

```
SELECT DISTINCT nombres de las columnas FROM tablaOrigen;
```

```
SELECT DISTINCT nombre, apellido FROM Alumnos;
```

Teniendo la siguiente tabla de alumnos:

Alumnos			
Id	Nombre	Apellido	Edad
1	Agustín	Cocco	24
2	Martin	Bullón	21
3	Agustín	Cocco	1

El resultado que mostraría sería:

Alumnos	
Nombre	Apellido
Agustín	Cocco
Martin	Bullón

8. WHERE

Establece criterios de selección de ciertas filas en el resultado de la consulta gracias a las condiciones de búsqueda. Si no se requiere condiciones de búsqueda puede omitirse y el resultado de la consulta serán todas las filas de las tablas enunciadas en el FROM.

```
SELECT nombres de las columnas FROM tablaOrigen WHERE condición de  
Búsqueda;
```

```
SELECT nombre, apellido FROM Alumnos WHERE nombre = "Agustín";
```

En este ejemplo traerá todos los alumnos con nombre Agustín. Nótese que el nombre está en comillas dobles, esto es porque si vamos a poner una cadena en la condición debe estar entre comillas dobles, si fuese un numero no seria necesario.

Teniendo la siguiente tabla alumnos:

Alumnos			
Id	Nombre	Apellido	Edad
1	Agustín	Cocco	24
2	Martin	Bullón	21

El resultado que mostraría sería:

Alumnos	
Nombre	Apellido
Agustín	Cocco

En las condiciones WHERE podemos utilizar operadores lógicos, operadores relaciones operadores propios de SQL.

Operadores Relacionales

Operador	Significado	Ejemplo
>	Mayor que	SELECT * FROM Alumnos WHERE edad > 21;
<	Menor que	SELECT * FROM Alumnos WHERE edad < 18;
=	Igual que	SELECT * FROM Alumnos WHERE edad = 20;
>=	Mayor o igual que	SELECT * FROM Alumnos WHERE edad >= 30;
<=	Menor o igual que	SELECT * FROM Alumnos WHERE edad <= 10;
<> o !=	Distinto que	SELECT * FROM Alumnos WHERE edad <> 5;

En todos estos ejemplos estamos buscando las filas donde la edad de un alumno sea mayor, menor, etc, a x edad. Usamos edad pero puede ser cualquier valor numérico o valor de tipo cadena.

Operadores Lógicos

Operador	Significado	Ejemplo
AND	El operador AND muestra un registro si todas las condiciones separadas por AND son verdaderas	SELECT * FROM Alumnos WHERE edad = 18 AND edad = 21;
OR	El operador OR muestra un registro si algunas de las condiciones separadas por OR es verdadera.	SELECT * FROM Alumnos WHERE edad = 15 OR edad = 20;
NOT	El operador NOT muestra un registro si la/s condición/es no es verdadera.	SELECT * FROM Alumnos WHERE NOT edad = 20;

Los operadores lógicos sirven para filtrar resultados basados en más de una condición. **Operadores propios de SQL**

a) BETWEEN

El operador BETWEEN selecciona valores dentro de un rango determinado. Los valores pueden ser números, texto o fechas.

```
SELECT nombre/s de la/s columna/s FROM tablaOrigen WHERE condición de  
Búsqueda BETWEEN valor1 AND valor2;
```

```
SELECT * FROM Alumnos WHERE edad BETWEEN 21 AND 40;
```

Usamos edad pero puede ser cualquier valor numérico o valor de tipo cadena.

Teniendo la siguiente tabla:

Id	Nombre	Apellido	Edad
1	Agustín	Cocco	24
2	Martin	Bullon	39
3	Mariela	Lima	60
4	Juliana	Martínez	30
5	Gastón	Vidal	26

El resultado sería:

Id	Nombre	Apellido	Edad
1	Agustín	Cocco	24
2	Martin	Bullon	39
4	Juliana	Martínez	30
5	Gastón	Vidal	26

b) IN

El operador IN te permite especificar varios valores para una condición de una cláusula WHERE. Es un atajo para no escribir varias condiciones OR.

```
SELECT nombre/s de la/s columna/s FROM tablaOrigen WHERE condición de  
Búsqueda IN (valor1, valor2, valor3, ...);
```

```
SELECT * FROM Alumnos WHERE nombre IN ("Agustín", "Mariela", "Juliana");
```

Usamos nombre pero puede ser cualquier valor numérico o valor de tipo cadena.

Teniendo la siguiente tabla de alumnos:

Id	Nombre	Apellido	Edad
1	Agustín	Cocco	24
2	Martin	Bullon	15

3	Mariela	Lima	20
---	---------	------	----

El resultado que mostraría sería:

Id	Nombre	Apellido	Edad
1	Agustín	Cocco	24
3	Mariela	Lima	20

c) LIKE

El operador LIKE se usa en una cláusula WHERE para buscar un patrón específico en una columna. También se usa cuando queremos utilizar una cadena en una comparación WHERE

Hay dos símbolos que se utilizan a menudo junto con el operador LIKE:

El signo de porcentaje (%) representa cero, uno o varios caracteres y el guion bajo (_) para representar un carácter. Estos signos se pueden utilizar por separado o juntos.

`SELECT nombre/s de la/s columna/s FROM tablaOrigen WHERE condición de Búsqueda LIKE patrón de valores o cadena;`

`SELECT nombre, apellido FROM Alumnos WHERE nombre LIKE 's%';`

Esa consulta, lo que hace es traer todos los alumnos donde el nombre empiece con el carácter s.

Teniendo la siguiente tabla de alumnos:

Id	Nombre	Apellido	Edad
1	Sebastián	Gómez	24
2	Sabrina	Martínez	15
3	Mariela	Lima	20

El resultado que mostraría sería:

Id	Nombre	Apellido	Edad
1	Sebastián	Gómez	24
2	Sabrina	Martínez	15

Operador LIKE	Significado
WHERE nombre LIKE 'a%'	Encuentra cualquier nombre que empiece con "a".
WHERE nombre LIKE '%a'	Encuentra cualquier nombre que termine con "a".

WHERE nombre LIKE '%ar%'	Encuentra cualquier nombre que tenga "ar" en cualquier posición.
WHERE nombre LIKE '_e%'	Encuentra cualquier nombre que tenga "e" en la segunda posición.
WHERE nombre LIKE 'e_%'	Encuentra cualquier nombre que empiece con "e" y sea por lo menos de 2 de largo.
WHERE nombre LIKE 'e__%'	Encuentra cualquier nombre que empiece con "e" y sea por lo menos de 3 de largo.
WHERE nombre LIKE 'a%n%'	Encuentra cualquier nombre que empiece con "a" y termine con "n".

9. ORDER BY

La cláusula ORDER BY permitirá establecer la columna o columnas sobre las cuales las filas que se mostrarán de la consulta deberán ser ordenadas. Este orden puede ser ascendente si se agrega la palabra ASC y descendiente si se agrega la palabra DESC al final.

Esta cláusula puede omitirse.

```
SELECT nombre/s de la/s columna/s FROM tablaOrigen ORDER BY columna
ASC|DESC;
```

```
SELECT nombre, apellido FROM Alumno ORDER BY nombre ASC;
```

En este caso mostraría los resultados ordenados de manera ascendente, según el nombre de los alumnos.

Teniendo la siguiente tabla de alumnos:

Alumnos			
Id	Nombre	Apellido	Edad
1	Jerónimo	Wiunkhaus	24
2	Ana	Gadea	15
3	Mariela	Lima	20

El resultado que mostraría sería:

Alumnos			
Id	Nombre	Apellido	Edad

2	Ana	Gadea	15
1	Jerónimo	Wiunkhaus	24
3	Mariela	Lima	20

10. GROUP BY

Especifica una consulta sumaria. En vez de producir una fila de resultados por cada fila de datos de la base de datos, una consulta sumaria agrupa todas las filas similares y luego produce una fila sumaria de resultados para cada grupo de los nombres de columnas enunciado en esta cláusula.

En otras palabras, esta cláusula permitirá agrupar un conjunto de columnas con valores repetidos y utilizar las funciones de agregación sobre las columnas con valores no repetidas. Esta cláusula puede omitirse.

```
SELECT nombre/s de la/s columna/s FROM tablaOrigen GROUP BY nombres de
columna/s por la cual Agrupar;
```

¿QUE SON LAS FUNCIONES DE AGREGACIÓN?

En la gestión de bases de datos, una función de agregación es una función en la que los valores de varias filas se agrupan bajo un criterio para formar un valor único más significativo.

Estas funciones se ponen el SELECT.

Existen 5 tipos de funciones de agregación, MAX(), MIN(), COUNT(), SUM(), AVG(). a) MAX

Esta función retorna el valor más grande de una columna.

```
SELECT MAX(nombre de la columna) FROM tablaOrigen;
```

SELECT MAX(salario) FROM Empleados; Teniendo la siguiente tabla de empleados:

Id	Nombre	Apellido	Salario
1	Franco	Medina	1000
2	Agustina	Koch	2000
3	Ignacio	Pérez	1500

El resultado que mostraría sería:

MAX(Salario)
2000

b) MIN

Esta función retorna el valor más chico de una columna.

```
SELECT MIN(nombre de la columna) FROM tablaOrigen;
```

```
SELECT MIN(salario) FROM Empleados;
```

Teniendo la siguiente tabla de empleados:

Id	Nombre	Apellido	Salario
1	Franco	Medina	1000
2	Agustina	Koch	2000
3	Ignacio	Pérez	1500
4	Martin	Bruno	3000

El resultado que mostraría sería:

MIN(Salario)
1000

c) AVG

Esta función retorna el promedio de una columna.

```
SELECT AVG(nombre de la columna) FROM tablaOrigen;
```

SELECT AVG(salario) FROM Empleados; Teniendo la siguiente tabla de empleados:

Id	Nombre	Apellido	Salario
1	Franco	Medina	1000
2	Agustina	Koch	2000
3	Ignacio	Pérez	1600
4	Valentín	Mazuran	700

El resultado que mostraría sería:

AVG(Salario)
1325

d) COUNT

Esta función retorna el numero de filas de una columna.

```
SELECT COUNT(nombre de la columna) FROM tablaOrigen;
```

```
SELECT COUNT(Id) FROM Empleados;
```

Teniendo la siguiente tabla de empleados:

Id	Nombre	Apellido	Salario
1	Franco	Medina	1000
2	Agustina	Koch	2000
3	Ignacio	Pérez	1600

El resultado que mostraría sería:

COUNT(Id)
3

En este caso ponemos el id, para saber cuantos empleados tenemos en la tabla empleados.

También podemos usar el COUNT(*), este no requiere que le pasamos una columna concreta y cuenta todas filas de una tabla, mostrando tanto los valores repetidos como los valores en null.

Id	Nombre	Apellido	Salario
1	Franco	Medina	1000
2	Agustina	Koch	2000
3	Franco	Medina	1600
4	Martin	Santiago	null

El resultado que mostraría sería:

COUNT(Id)
4

Entonces, volviendo al Group By, vamos a utilizar esta sentencia junto con las funciones de agregación para agrupar los valores que devuelva dicha función. Existen dos tipos de GROUP BY.

`SELECT nombre, SUM(salario) FROM Empleados GROUP BY nombre;`

Teniendo la siguiente tabla de empleados:

Empleados			
Id	Nombre	Salario	Comisión
1	Franco	1000	500
2	Mariela	2000	200
3	Franco	1000	800

4	Mariela	2000	350
---	---------	------	-----

El resultado que mostraría sería:

Empleados	
Nombre	SUM(Salario)
Franco	2000
Mariela	4000

El resultado de la consulta, muestra que agrupa todos los nombre repetidos bajo un solo nombre y el salario es la suma de los salarios de las filas que fueron agrupadas.

Otro ejemplo de un group by sería:

`SELECT COUNT(ID), pais FROM Personas GROUP BY pais;` Teniendo la siguiente tabla de personas:

Personas		
Id	Nombre	País
1	Franco	Argentina
2	Juliana	Alemania
3	Agustín	Argentina

El resultado que mostraría sería:

Personas	
COUNT(Id)	País
2	Argentina
1	Alemania

En la consulta hacemos un count del id de personas para saber cuantos hay, pero al agrupar el resultado por países, nos muestra cuantas personas hay en cada país.

11. HAVING

Esta cláusula le dice al SQL que incluya sólo ciertos grupos producidos por la cláusula GROUP BY en los resultados de la consulta. Al igual que la cláusula WHERE, utiliza una condición de búsqueda para especificar los grupos deseados. La cláusula HAVING es la encargada de condicionar la selección de los grupos en base a los valores resultantes en las funciones agregadas utilizadas debidas que la cláusula WHERE condiciona solo para la selección de filas individuales. Esta cláusula puede omitirse.

`SELECT nombre/s de la/s columna/s FROM tablaOrigen GROUP BY nombres de columnas por la cual Agrupar HAVING condiciónBúsqueda para Group By;`

`SELECT COUNT(ID), pais FROM Personas GROUP BY Personas GROUP BY pais`

HAVING COUNT(ID) > 1;

Teniendo la siguiente tabla de personas:

Personas		
Id	Nombre	País
1	Franco	Argentina
2	Juliana	Alemania
3	Agustín	Argentina
4	Gastón	Alemania
5	Mariela	Uruguay

El resultado que mostraría sería:

Personas	
COUNT(Id)	País
2	Argentina
2	Alemania

En la consulta hacemos un COUNT del ID de personas para saber cuantos hay, las agrupamos por países para que nos muestre cuantas personas hay en cada país. Pero, con el HAVING le decimos que nos muestre solo los resultados donde el COUNT sea mayor a 1, o en otras palabras, mostramos los países que tienen más de una persona.

12. AS

La sentencia AS, le da un alias a una o la columna de una tabla, un nombre temporal. El alias existe solo por la duración de la consulta.

El alias se usa para darle a una columna un nombre más legible

`SELECT nombre/s de la/s columna/s AS alias FROM tablaOrigen;`

`SELECT nombre AS Nombre_Alumno, apellido As Apellido_Alumno FROM Alumnos;`

Teniendo la siguiente tabla de alumnos:

Alumnos			
Id	Nombre	Apellido	Edad
1	Jerónimo	Wiunkhaus	24
2	Ana	Gadea	15
3	Mariela	Lima	20

El resultado que mostraría sería:

Alumnos	
Nombre Alumno	Apellido Alumno
Ana	Gadea
Jerónimo	Wiunkhaus
Mariela	Lima

Todas estas cláusulas / sentencias pueden ser usadas juntas, no es necesario que las usen separadas.

```
SELECT nombres de las columnas AS Alias FROM tablaOrigen  
WHERE condición de Búsqueda  
GROUP BY nombres de columnas por la cual Agrupar  
HAVING condiciónBúsqueda para Group By  
ORDER BY nombre de columnas [ASC | DESC]
```

13. ROUND

La sentencia round sirve para redondear los decimales de un número que se pida en un select.

```
SELECT AVG(salario) FROM Empleados;
```

Empleados
AVG(Salario)
1325,55

```
SELECT ROUND(AVG(salario)) FROM Empleados;
```

Empleados
AVG(Salario)
1326

14. LIMIT

La cláusula LIMIT se utiliza para establecer un límite al número de resultados devueltos por SQL.

```
SELECT nombres de las columnas FROM tablaOrigen LIMIT numero x;
```

```
SELECT nombre, apellido FROM Alumnos LIMIT 1;
```

Teniendo la siguiente tabla de alumnos:

Alumnos			
Id	Nombre	Apellido	Edad
1	Jerónimo	Wiunkhaus	24
2	Ana	Gadea	15
3	Mariela	Lima	20

El resultado que mostraría sería:

Alumnos	
Nombre	Apellido
Jerónimo	Wiunkhaus

CONSULTAS MULTITABLAS

Como habíamos dicho previamente en la teoría, estamos trabajando con base de datos relacionales, esto significa que tenemos tablas relacionadas entre sí. Y dentro de esas tablas, tenemos fila relacionadas con filas de otras tablas.

Pero, ¿cómo hacemos para traer la información de una tabla y la información de la tabla con la que está relacionada?. Una sentencia muy útil para unificar información de tablas relacionales es el JOIN.

SQL JOIN

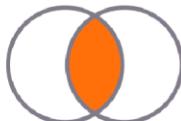
La sentencia JOIN, se usa para combinar data o filas de dos o más tablas que tengan un campo en común entre ellas. Usualmente es la llave foránea.

Los diferentes tipos de JOIN son:

1. INNER JOIN

El INNER JOIN selecciona todas las filas que tengan un valor en común con la/s tabla/s. Si hay una fila, que no tiene un valor en común con la otra tabla no la trae.

INNER JOIN



Solo los valores en común de la izquierda y la derecha

```
SELECT *
FROM TABLE_1
INNER JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

```
SELECT nombre/s de la/s columna/s FROM tabla1 INNER JOIN tabla2 ON
tabla1.nombre_columna = tabla2.nombre_columna;
```

```
SELECT Nombre, Nombre_curso FROM Profesores INNER JOIN Cursos ON
Profesores.Id = Cursos.Id_profesor;
```

Teniendo la siguiente tabla de profesores:

Profesores			
Id	Nombre	Apellido	Edad
1	Agustín	Oviedo	24
2	Ana	Gadea	15
3	Mariela	Lima	20
4	Francisco	Chirino	30

Teniendo la siguiente tabla de Cursos:

Cursos			
Id	Nombre_curso	Costo	Id_profesor
1	Curso de Programación	1000	1
2	Curso de Mecánica	2000	2
3	Curso de Cocina	500	3

El resultado que mostraría sería:

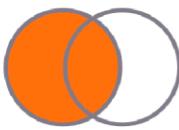
Resultado INNER JOIN	
Nombre	Nombre_curso
Agustín	Curso de Programación
Ana	Curso de Mecánica
Mariela	Curso de Cocina

Gracias al INNER JOIN, podemos mostrar todos los nombres de los profesores, junto al curso que dan, nosotros decimos que son el profesor de ese curso, porque tienen asignado su id en la tabla curso. Y como usamos el INNER JOIN, solo mostramos los profesores que tenían su id en la tabla curso. Esto se por la condición que pusimos arriba en el ON, donde decíamos que el valor a chequear por posible coincidencia era el id en la tabla profesor y el id_profesor en la tabla curso.

2. LEFT JOIN

La sentencia LEFT JOIN retorna todos los registros de la tabla de la izquierda (tabla1) y todos los registros con coincidencia de la tabla de la derecha (tabla2). Si no existe ninguna coincidencia para alguna de las filas de la tabla de la izquierda, de igual forma todos los resultados de la primera tabla se muestran.

LEFT JOIN



Todo lo de izquierda
+
valores en comun de la derecha

```
SELECT *
FROM TABLE_1
LEFT JOIN TABLE_2
ON TABLE_1.KEY = TABLE_2.KEY
```

```
SELECT nombre/s de la/s columna/s FROM tabla1 LEFT JOIN tabla2 ON
tabla1.nombre_columna = tabla2.nombre_columna;
```

```
SELECT Nombre, Nombre_curso FROM Profesores LEFT JOIN Cursos ON
Profesores.Id = Cursos.Id_profesor;
```

Teniendo la siguiente tabla de profesores:

Profesores			
Id	Nombre	Apellido	Edad
1	Agustín	Oviedo	24
2	Ana	Gadea	15
3	Mariela	Lima	20
4	Francisco	Chirino	30

Teniendo la siguiente tabla de Cursos:

Cursos			
Id	Nombre_curso	Costo	Id_profesor
1	Curso de Programación	1000	1
2	Curso de Mecánica	2000	2

El resultado que mostraría sería:

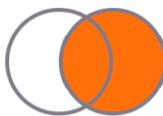
Resultado LEFT JOIN	
Nombre	Nombre_curso
Agustín	Curso de Programación
Ana	Curso de Mecánica
Mariela	NULL
Francisco Chirino	NULL

Si nos fijamos en el resultado de la consulta, podemos ver que trajo todas las filas de la tabla de la izquierda, sin importar si las filas tenían coincidencia o no.

3. RIGHT JOIN

Esta sentencia es parecida a la anterior pero le da prioridad al tabla de la derecha.

RIGHT JOIN



Todo lo de la derecha
+
valores en común de la izquierda

```
SELECT *  
FROM TABLE_1  
RIGHT JOIN TABLE_2  
ON TABLE_1.KEY = TABLE_2.KEY
```

`SELECT nombre/s de la/s columna/s FROM tabla1 RIGHT JOIN tabla2 ON
tabla1.nombre_columna = tabla2.nombre_columna;`

`SELECT Nombre, Nombre_curso FROM Profesores RIGHT JOIN Cursos ON
Profesores.Id = Cursos.Id_profesor;`

Teniendo la siguiente tabla de profesores:

Profesores			
Id	Nombre	Apellido	Edad
1	Agustín	Oviedo	24
2	Ana	Gadea	15
3	Mariela	Lima	20
4	Francisco	Chirino	30

Teniendo la siguiente tabla de Cursos:

Cursos			
Id	Nombre_curso	Costo	Id_profesor
1	Curso de Programación	1000	1
2	Curso de Mecánica	2000	2
3	Curso de Natación	600	NULL

El resultado que mostraría sería:

Resultado RIGHT JOIN	
Nombre	Nombre_curso
Agustín	Curso de Programación
Ana	Curso de Mecánica

NULL	Curso de Natación
------	-------------------

Si nos fijamos en el resultado de la consulta, podemos ver que trajo todas las filas de la tabla de la derecha, sin importar si las filas tenían coincidencia o no.

SUBCONSULTAS

Una subconsulta en SQL consiste en utilizar los resultados de una consulta dentro de otra, que se considera la principal. Esta posibilidad fue la razón original para la palabra “estructurada” en el nombre Lenguaje de Consultas Estructuradas (Structured Query Language, SQL).

Anteriormente hemos utilizado la cláusula WHERE para seleccionar los datos que deseábamos comparando un valor de una columna con una constante, o un grupo de ellas. Si los valores de dichas constantes son desconocidos, normalmente por proceder de la aplicación de funciones a determinadas columnas de la tabla, tendremos que utilizar subconsultas. Por ejemplo, queremos saber la lista de empleados cuyo salario supere el salario medio.

En primer lugar, tendríamos que averiguar el importe del salario medio:

```
SELECT AVG(salario) "Salario Medio" FROM Empleados;
```

Empleados
Salario Medio
256666,67

Ahora que sabemos el dato podríamos usarlo para la consulta:

```
SELECT nombre, salario FROM Empleados WHERE > 256666,67;
```

Empleados	
Nombre	Salario
Agustín	385000
Ana	608000

Esto estaría bien pero, es porque primero buscamos el dato en una consulta y una vez que conseguimos el dato, ahí hicimos la consulta. Pero, lo mejor sería que en vez de hacer dos consultas usemos una subconsulta, para que al mismo tiempo que averiguamos el salario medio, se calcule cuales son los empleados que tienen un sueldo mayor a ese salario medio.

```
SELECT nombre, salario FROM Empleados WHERE > (SELECT AVG(salario) FROM Empleados);
```

Empleados	
Nombre	Salario
Agustín	385000
Ana	608000

Esto nos daría el mismo resultado, pero sin la necesidad de hacer dos consultas para saber el dato. Estos son los casos donde usaríamos una subconsulta, donde no sabíamos el salario medio antes de hacer la consulta.

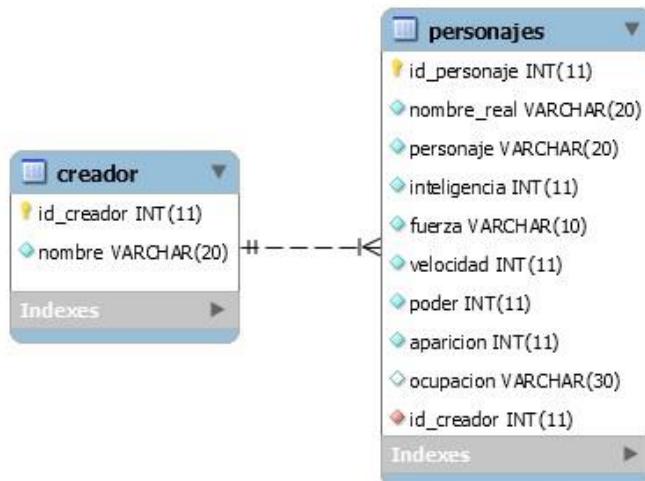
EJERCICIOS DE APRENDIZAJE

Para la realización de los ejercicios que se describen a continuación, es necesario descargar el archivo [scriptsBD.zip](#) que contiene algunos scripts con las bases de datos sobre las cuales se va a trabajar. En cada ejercicio se indica el nombre del script que se debe utilizar. Para abrir y ejecutar los scripts van a encontrar un pdf de como hacerlo en Moodle, con el nombre de [Tutorial Scripts SQL](#).



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Abrir el script llamado “superhéroes” y ejecutarlo de modo tal que se cree la base de datos y todas sus tablas. Posteriormente, crear las tablas que se muestran en el siguiente modelo de entidad relación:



- a) Insertar en las tablas creadas los siguientes datos:

Tabla creador

id_creador	creador
1	Marvel
2	DC Comics

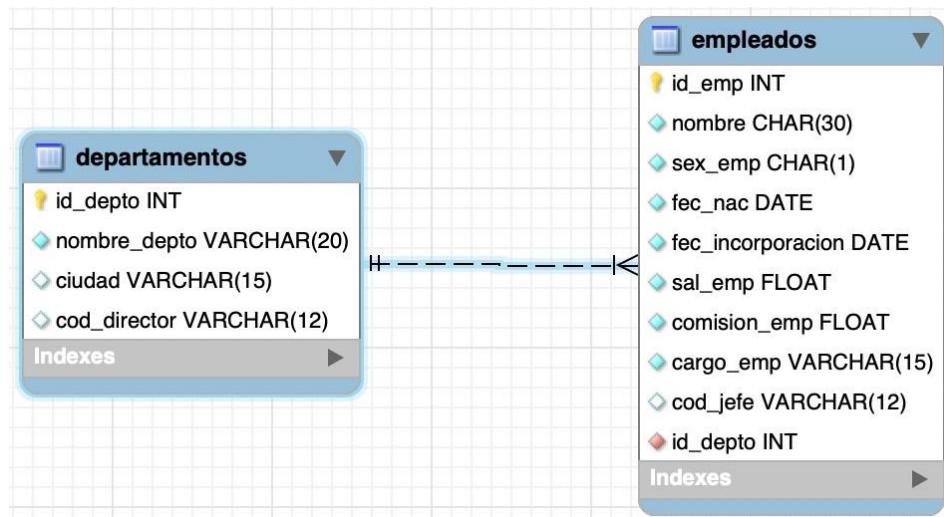
Tabla personajes

id_personaje	nombre_real	personaje	Inteligencia	fuerza	velocidad	poder	aparicion	ocupación	id_creador
1	Bruce Banner	Hulk	160	600 mil	75	98	1962	Fisico Nuclear	1
2	Tony Stark	Iron Man	170	200 mil	70	123	1963	Inventor Industrial	1
3	Thor Odinson	Thor	145	infinita	100	235	1962	Rey de Asgard	1
4	Wanda Maximoff	Bruja Escarlata	170	100 mil	90	345	1964	Bruja	1
5	Carol Danvers	Capitana Marvel	157	250 mil	85	128	1968	Oficial de inteligencia	1
6	Thanos	Thanos	170	infinita	40	306	1973	Adorador de la muerte	1
7	Peter Parker	Spiderman	165	25 mil	80	74	1962	Fotógrafo	1
8	Steve Rogers	Capitan America	145	600	45	60	1941	Oficial Federal	1
9	Bobby Drake	Ice Man	140	2 mil	64	122	1963	Contador	1
10	Barry Allen	Flash	160	10 mil	120	168	1956	Científico forense	2
11	Bruce Wayne	Batman	170	500	32	47	1939	Hombre de negocios	2
12	Clark Kent	Superman	165	infinita	120	182	1948	Reportero	2
13	Diana Prince	Mujer Maravilla	160	infinita	95	127	1949	Princesa guerrera	2

Una vez insertados todos los registros realizar una selección de todos los atributos para corroborar que las tablas se encuentren completas.

- b) Cambiar en la tabla personajes el año de aparición a 1938 del personaje Superman. A continuación, realizar un listado de toda la tabla para verificar que el personaje haya sido actualizado.
- c) El registro que contiene al personaje Flash. A continuación, mostrar toda la tabla para verificar que el registro haya sido eliminado.
- d) Eliminar la base de datos superhéroes.

2. Abrir el script llamado “personal-inserts” y ejecutarlo de modo tal que se cree la base de datos “personal”, se creen las tablas y se inserten todos los datos en las tablas para que quede de la siguiente manera:



a) A continuación, realizar las siguientes consultas sobre la base de datos personal:

1. Obtener los datos completos de los empleados.
2. Obtener los datos completos de los departamentos.
3. Listar el nombre de los departamentos.
4. Obtener el nombre y salario de todos los empleados.
5. Listar todas las comisiones.
6. Obtener los datos de los empleados cuyo cargo sea ‘Secretaria’.
7. Obtener los datos de los empleados vendedores, ordenados por nombre alfabéticamente.
8. Obtener el nombre y cargo de todos los empleados, ordenados por salario de menor a mayor.
9. Elabore un listado donde para cada fila, figure el alias ‘Nombre’ y ‘Cargo’ para las respectivas tablas de empleados.
10. Listar los salarios y comisiones de los empleados del departamento 2000, ordenado por comisión de menor a mayor.
11. Obtener el valor total a pagar que resulta de sumar el salario y la comisión de los empleados del departamento 3000 una bonificación de 500, en orden alfabético del empleado.
12. Muestra los empleados cuyo nombre empiece con la letra J.

13. Listar el salario, la comisión, el salario total (salario + comisión) y nombre, de aquellos empleados que tienen comisión superior a 1000.
14. Obtener un listado similar al anterior, pero de aquellos empleados que NO tienen comisión.
15. Obtener la lista de los empleados que ganan una comisión superior a su sueldo.
16. Listar los empleados cuya comisión es menor o igual que el 30% de su sueldo.
17. Hallar los empleados cuyo nombre no contiene la cadena “MA”
18. Obtener los nombres de los departamentos que sean “Ventas” ni “Investigación” ni ‘Mantenimiento’.
19. Ahora obtener los nombres de los departamentos que no sean “Ventas” ni “Investigación” ni ‘Mantenimiento’.
20. Mostrar el salario más alto de la empresa.
21. Mostrar el nombre del último empleado de la lista por orden alfabético.
22. Hallar el salario más alto, el más bajo y la diferencia entre ellos.
23. Hallar el salario promedio por departamento.

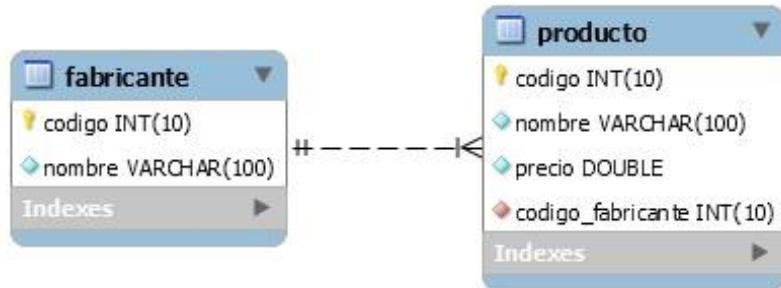
Consultas con Having

24. Hallar los departamentos que tienen más de tres empleados. Mostrar el número de empleados de esos departamentos.
25. Mostrar el código y nombre de cada jefe, junto al número de empleados que dirige. Solo los que tengan más de dos empleados (2 incluido).
26. Hallar los departamentos que no tienen empleados

Consulta con Subconsulta

27. Mostrar la lista de los empleados cuyo salario es mayor o igual que el promedio de la empresa. Ordenarlo por departamento.

3. Abrir el script de la base de datos llamada “tienda.sql” y ejecutarlo para crear sus tablas e insertar datos en las mismas. A continuación, generar el modelo de entidad relación. Deberá obtener un diagrama de entidad relación igual al que se muestra a continuación:



A continuación, se deben realizar las siguientes consultas sobre la base de datos:

1. Lista el nombre de todos los productos que hay en la tabla producto.
2. Lista los nombres y los precios de todos los productos de la tabla producto.
3. Lista todas las columnas de la tabla producto.

4. Lista los nombres y los precios de todos los productos de la tabla producto, redondeando el valor del precio.
5. Lista el código de los fabricantes que tienen productos en la tabla producto.
10. Lista el código de los fabricantes que tienen productos en la tabla producto, sin mostrar los repetidos.
11. Lista los nombres de los fabricantes ordenados de forma ascendente.
12. Lista los nombres de los productos ordenados en primer lugar por el nombre de forma ascendente y en segundo lugar por el precio de forma descendente.
13. Devuelve una lista con las 5 primeras filas de la tabla fabricante.
14. Lista el nombre y el precio del producto más barato. (Utilice solamente las cláusulas ORDER BY y LIMIT)
15. Lista el nombre y el precio del producto más caro. (Utilice solamente las cláusulas ORDER BY y LIMIT)
16. Lista el nombre de los productos que tienen un precio menor o igual a \$120.
17. Lista todos los productos que tengan un precio entre \$60 y \$200. Utilizando el operador BETWEEN.
18. Lista todos los productos donde el código de fabricante sea 1, 3 o 5. Utilizando el operador IN.
23. Devuelve una lista con el nombre de todos los productos que contienen la cadena Portátil en el nombre.

Consultas Multitable

1. Devuelve una lista con el código del producto, nombre del producto, código del fabricante y nombre del fabricante, de todos los productos de la base de datos.
2. Devuelve una lista con el nombre del producto, precio y nombre de fabricante de todos los productos de la base de datos. Ordene el resultado por el nombre del fabricante, por orden alfabetico.
3. Devuelve el nombre del producto, su precio y el nombre de su fabricante, del producto más barato.
4. Devuelve una lista de todos los productos del fabricante Lenovo.
5. Devuelve una lista de todos los productos del fabricante Crucial que tengan un precio mayor que \$200.
6. Devuelve un listado con todos los productos de los fabricantes Asus, HewlettPackard. Utilizando el operador IN.
7. Devuelve un listado con el nombre de producto, precio y nombre de fabricante, de todos los productos que tengan un precio mayor o igual a \$180. Ordene el resultado en primer lugar por el precio (en orden descendente) y en segundo lugar por el nombre (en orden ascendente)

Consultas Multitable

Resuelva todas las consultas utilizando las cláusulas LEFT JOIN y RIGHT JOIN.

1. Devuelve un listado de todos los fabricantes que existen en la base de datos, junto con los productos que tiene cada uno de ellos. El listado deberá mostrar también aquellos fabricantes que no tienen productos asociados.

- Devuelve un listado donde sólo aparezcan aquellos fabricantes que no tienen ningún producto asociado.

Subconsultas (En la cláusula WHERE)

Con operadores básicos de comparación

- Devuelve todos los productos del fabricante Lenovo. (Sin utilizar INNER JOIN).
- Devuelve todos los datos de los productos que tienen el mismo precio que el producto más caro del fabricante Lenovo. (Sin utilizar INNER JOIN).
- Lista el nombre del producto más caro del fabricante Lenovo.
- Lista todos los productos del fabricante Asus que tienen un precio superior al precio medio de todos sus productos.

Subconsultas con IN y NOT IN

- Devuelve los nombres de los fabricantes que tienen productos asociados. (Utilizando IN o NOT IN).
- Devuelve los nombres de los fabricantes que no tienen productos asociados. (Utilizando IN o NOT IN).

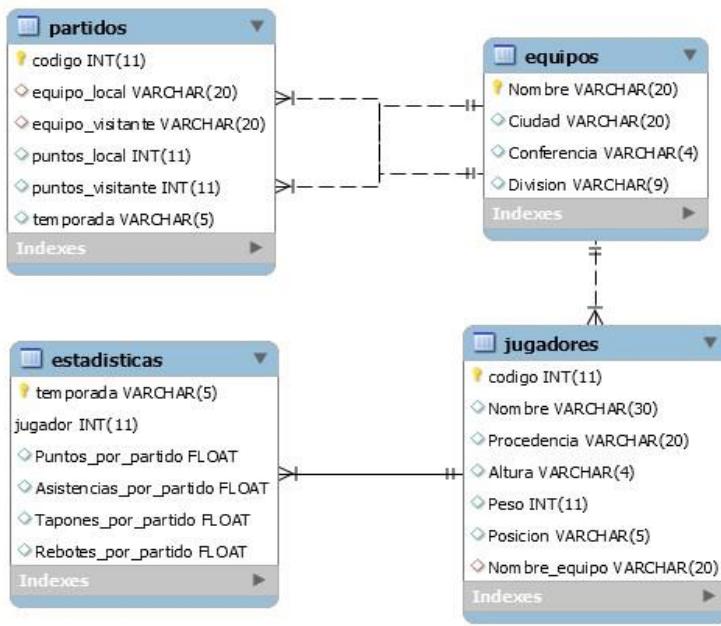
Subconsultas (En la cláusula HAVING)

- Devuelve un listado con todos los nombres de los fabricantes que tienen el mismo número de productos que el fabricante Lenovo.

EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

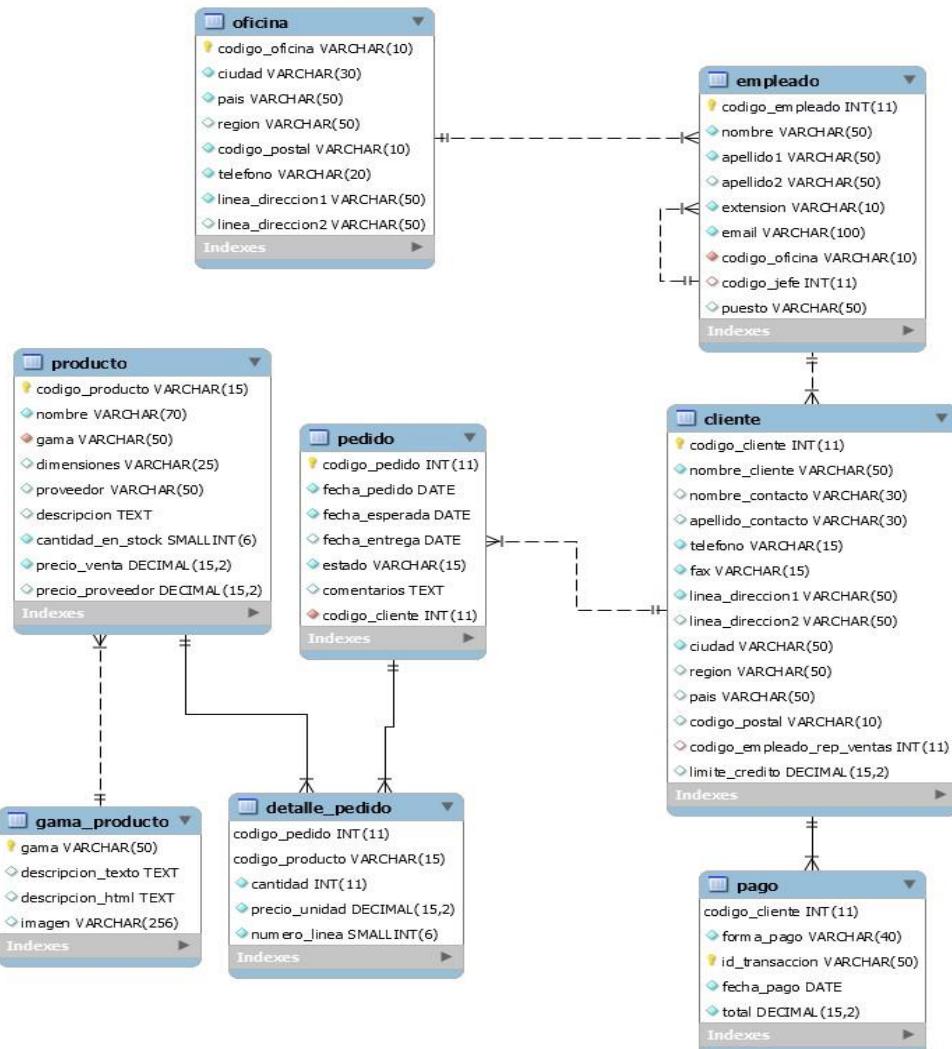
- Abrir el script de la base de datos llamada “nba.sql” y ejecutarlo para crear todas las tablas e insertar datos en las mismas. A continuación, generar el modelo de entidad relación. Deberá obtener un diagrama de entidad relación igual al que se muestra a continuación:



A continuación, se deben realizar las siguientes consultas sobre la base de datos:

1. Mostrar el nombre de todos los jugadores ordenados alfabéticamente.
2. Mostrar el nombre de los jugadores que sean pivots ('C') y que pesen más de 200 libras, ordenados por nombre alfabéticamente.
3. Mostrar el nombre de todos los equipos ordenados alfabéticamente.
4. Mostrar el nombre de los equipos del este (East).
5. Mostrar los equipos donde su ciudad empieza con la letra 'c', ordenados por nombre.
6. Mostrar todos los jugadores y su equipo ordenados por nombre del equipo.
7. Mostrar todos los jugadores del equipo "Raptors" ordenados por nombre.
8. Mostrar los puntos por partido del jugador 'Pau Gasol'.
9. Mostrar los puntos por partido del jugador 'Pau Gasol' en la temporada '04/05'.
10. Mostrar el número de puntos de cada jugador en toda su carrera.
11. Mostrar el número de jugadores de cada equipo.
12. Mostrar el jugador que más puntos ha realizado en toda su carrera.
13. Mostrar el nombre del equipo, conferencia y división del jugador más alto de la NBA.
14. Mostrar la media de puntos en partidos de los equipos de la división Pacific.
15. Mostrar el partido o partidos (equipo_local, equipo_visitante y diferencia) con mayor diferencia de puntos.
16. Mostrar la media de puntos en partidos de los equipos de la división Pacific.
17. Mostrar los puntos de cada equipo en los partidos, tanto de local como de visitante.
18. Mostrar quien gana en cada partido (codigo, equipo_local, equipo_visitante, equipo_ganador), en caso de empate sera null.

2. Abrir el script de la base de datos llamada “jardineria.sql” y ejecutarlo para crear todas las tablas e insertar datos en las mismas. Deberá obtener un diagrama de entidad relación igual al que se muestra a continuación:



A continuación, se deben realizar las siguientes consultas sobre la base de datos:

Consultas sobre una tabla

- Devuelve un listado con el código de oficina y la ciudad donde hay oficinas.
- Devuelve un listado con la ciudad y el teléfono de las oficinas de España.
- Devuelve un listado con el nombre, apellidos y email de los empleados cuyo jefe tiene un código de jefe igual a 7.
- Devuelve el nombre del puesto, nombre, apellidos y email del jefe de la empresa.
- Devuelve un listado con el nombre, apellidos y puesto de aquellos empleados que no sean representantes de ventas.
- Devuelve un listado con el nombre de todos los clientes españoles.
- Devuelve un listado con los distintos estados por los que puede pasar un pedido.

8. Devuelve un listado con el código de cliente de aquellos clientes que realizaron algún pago en 2008. Tenga en cuenta que deberá eliminar aquellos códigos de cliente que aparezcan repetidos. Resuelva la consulta:
 - Utilizando la función YEAR de MySQL.
 - Utilizando la función DATE_FORMAT de MySQL.
 - Sin utilizar ninguna de las funciones anteriores.
9. Devuelve un listado con el código de pedido, código de cliente, fecha esperada y fecha de entrega de los pedidos que no han sido entregados a tiempo.
10. Devuelve un listado con el código de pedido, código de cliente, fecha esperada y fecha de entrega de los pedidos cuya fecha de entrega ha sido al menos dos días antes de la fecha esperada.
 - Utilizando la función ADDDATE de MySQL.
 - Utilizando la función DATEDIFF de MySQL.
11. Devuelve un listado de todos los pedidos que fueron rechazados en 2009.
12. Devuelve un listado de todos los pedidos que han sido entregados en el mes de enero de cualquier año.
13. Devuelve un listado con todos los pagos que se realizaron en el año 2008 mediante Paypal. Ordene el resultado de mayor a menor.
14. Devuelve un listado con todas las formas de pago que aparecen en la tabla pago. Tenga en cuenta que no deben aparecer formas de pago repetidas.
15. Devuelve un listado con todos los productos que pertenecen a la gama Ornamentales y que tienen más de 100 unidades en stock. El listado deberá estar ordenado por su precio de venta, mostrando en primer lugar los de mayor precio.
16. Devuelve un listado con todos los clientes que sean de la ciudad de Madrid y cuyo representante de ventas tenga el código de empleado 11 o 30.

Consultas multitabla (Composición interna)

Las consultas se deben resolver con INNER JOIN.

1. Obtén un listado con el nombre de cada cliente y el nombre y apellido de su representante de ventas.
2. Muestra el nombre de los clientes que hayan realizado pagos junto con el nombre de sus representantes de ventas.
3. Muestra el nombre de los clientes que no hayan realizado pagos junto con el nombre de sus representantes de ventas.
4. Devuelve el nombre de los clientes que han hecho pagos y el nombre de sus representantes junto con la ciudad de la oficina a la que pertenece el representante.
5. Devuelve el nombre de los clientes que no hayan hecho pagos y el nombre de sus representantes junto con la ciudad de la oficina a la que pertenece el representante.
6. Lista la dirección de las oficinas que tengan clientes en Fuenlabrada.

7. Devuelve el nombre de los clientes y el nombre de sus representantes junto con la ciudad de la oficina a la que pertenece el representante.
8. Devuelve un listado con el nombre de los empleados junto con el nombre de sus jefes.
9. Devuelve el nombre de los clientes a los que no se les ha entregado a tiempo un pedido.
10. Devuelve un listado de las diferentes gamas de producto que ha comprado cada cliente.

Consultas multitable (Composición externa)

Resuelva todas las consultas utilizando las cláusulas LEFT JOIN, RIGHT JOIN, JOIN.

1. Devuelve un listado que muestre solamente los clientes que no han realizado ningún pago.
2. Devuelve un listado que muestre solamente los clientes que no han realizado ningún pedido.
3. Devuelve un listado que muestre los clientes que no han realizado ningún pago y los que no han realizado ningún pedido.
4. Devuelve un listado que muestre solamente los empleados que no tienen una oficina asociada.
5. Devuelve un listado que muestre solamente los empleados que no tienen un cliente asociado.
6. Devuelve un listado que muestre los empleados que no tienen una oficina asociada y los que no tienen un cliente asociado.
7. Devuelve un listado de los productos que nunca han aparecido en un pedido.
8. Devuelve las oficinas donde no trabajan ninguno de los empleados que hayan sido los representantes de ventas de algún cliente que haya realizado la compra de algún producto de la gama Frutales.
9. Devuelve un listado con los clientes que han realizado algún pedido, pero no han realizado ningún pago.
10. Devuelve un listado con los datos de los empleados que no tienen clientes asociados y el nombre de su jefe asociado.

Consultas resumen

1. ¿Cuántos empleados hay en la compañía?
2. ¿Cuántos clientes tiene cada país?
3. ¿Cuál fue el pago medio en 2009?
4. ¿Cuántos pedidos hay en cada estado? Ordena el resultado de forma descendente por el número de pedidos.
5. Calcula el precio de venta del producto más caro y más barato en una misma consulta.
6. Calcula el número de clientes que tiene la empresa.
7. ¿Cuántos clientes tiene la ciudad de Madrid?
8. ¿Calcula cuántos clientes tiene cada una de las ciudades que empiezan por M?
9. Devuelve el nombre de los representantes de ventas y el número de clientes al que atiende cada uno.
10. Calcula el número de clientes que no tiene asignado representante de ventas.

11. Calcula la fecha del primer y último pago realizado por cada uno de los clientes. El listado deberá mostrar el nombre y los apellidos de cada cliente.
12. Calcula el número de productos diferentes que hay en cada uno de los pedidos.
13. Calcula la suma de la cantidad total de todos los productos que aparecen en cada uno de los pedidos.
14. Devuelve un listado de los 20 productos más vendidos y el número total de unidades que se han vendido de cada uno. El listado deberá estar ordenado por el número total de unidades vendidas.
15. La facturación que ha tenido la empresa en toda la historia, indicando la base imponible, el IVA y el total facturado. La base imponible se calcula sumando el coste del producto por el número de unidades vendidas de la tabla detalle_pedido. El IVA es el 21 % de la base imponible, y el total la suma de los dos campos anteriores.
16. La misma información que en la pregunta anterior, pero agrupada por código de producto.
17. La misma información que en la pregunta anterior, pero agrupada por código de producto filtrada por los códigos que empiecen por OR.
18. Lista las ventas totales de los productos que hayan facturado más de 3000 euros. Se mostrará el nombre, unidades vendidas, total facturado y total facturado con impuestos (21% IVA)

Subconsultas con operadores básicos de comparación

1. Devuelve el nombre del cliente con mayor límite de crédito.
2. Devuelve el nombre del producto que tenga el precio de venta más caro.
3. Devuelve el nombre del producto del que se han vendido más unidades. (Tenga en cuenta que tendrá que calcular cuál es el número total de unidades que se han vendido de cada producto a partir de los datos de la tabla detalle_pedido. Una vez que sepa cuál es el código del producto, puede obtener su nombre fácilmente.)
4. Los clientes cuyo límite de crédito sea mayor que los pagos que haya realizado. (Sin utilizar INNER JOIN).
5. Devuelve el producto que más unidades tiene en stock.
6. Devuelve el producto que menos unidades tiene en stock.
7. Devuelve el nombre, los apellidos y el email de los empleados que están a cargo de Alberto Soria.

Subconsultas con ALL y ANY

1. Devuelve el nombre del cliente con mayor límite de crédito.
2. Devuelve el nombre del producto que tenga el precio de venta más caro.
3. Devuelve el producto que menos unidades tiene en stock.

Subconsultas con IN y NOT IN

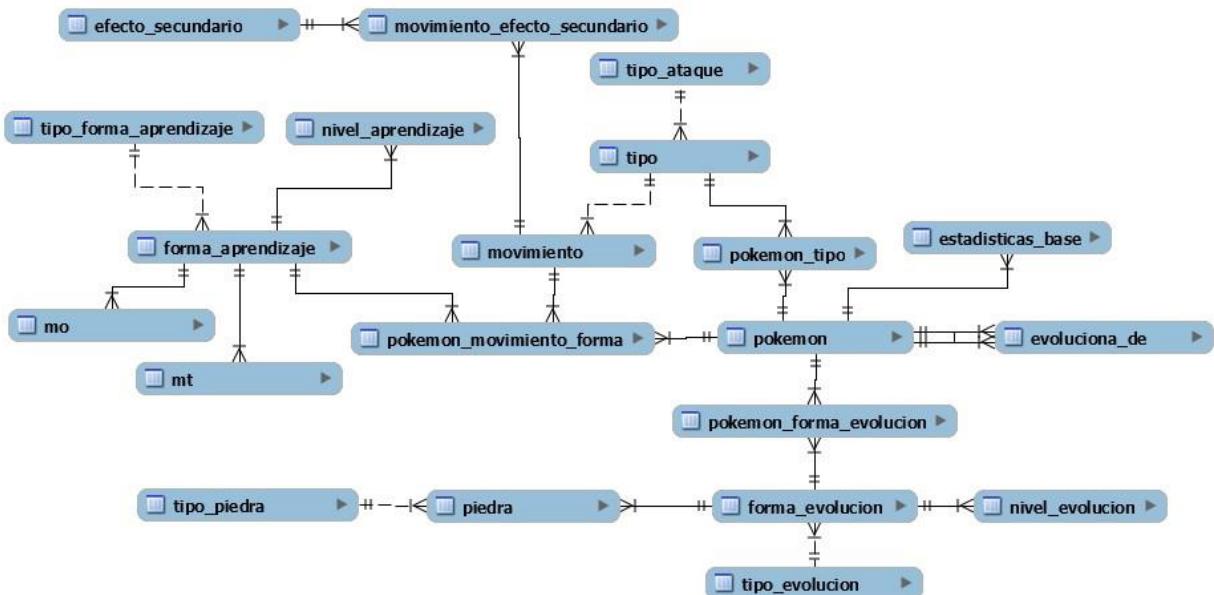
1. Devuelve el nombre, apellido1 y cargo de los empleados que no representen a ningún cliente.
2. Devuelve un listado que muestre solamente los clientes que no han realizado ningún pago.
3. Devuelve un listado que muestre solamente los clientes que sí han realizado ningún pago.
4. Devuelve un listado de los productos que nunca han aparecido en un pedido.

- Devuelve el nombre, apellidos, puesto y teléfono de la oficina de aquellos empleados que no sean representante de ventas de ningún cliente.

Subconsultas con EXISTS y NOT EXISTS

- Devuelve un listado que muestre solamente los clientes que no han realizado ningón pago.
- Devuelve un listado que muestre solamente los clientes que sí han realizado ningón pago.
- Devuelve un listado de los productos que nunca han aparecido en un pedido.
- Devuelve un listado de los productos que han aparecido en un pedido alguna vez.

- 3.** Importar el script de la base de datos llamada “pokemondb.sql” y ejecutarlo para crear todas las tablas e insertar los registros en las mismas. A continuación, generar el modelo de entidad relación y reorganizar las tablas para mayor claridad de sus relaciones. Deberá obtener un diagrama de entidad de relación similar al que se muestra a continuación:



A continuación, se deben realizar las siguientes consultas:

- Mostrar el nombre de todos los pokémon.
- Mostrar los pokémon que pesen menos de 10k.
- Mostrar los pokémon de tipo agua.
- Mostrar los pokémon de tipo agua, fuego o tierra ordenados por tipo.
- Mostrar los pokémon que son de tipo fuego y volador.
- Mostrar los pokémon con una estadística base de ps mayor que 200.
- Mostrar los datos (nombre, peso, altura) de la prevolución de Arbok.

8. Mostrar aquellos pokemon que evolucionan por intercambio.
9. Mostrar el nombre del movimiento con más prioridad.
10. Mostrar el pokemon más pesado.
11. Mostrar el nombre y tipo del ataque con más potencia.
12. Mostrar el número de movimientos de cada tipo.
13. Mostrar todos los movimientos que puedan envenenar.
14. Mostrar todos los movimientos que causan daño, ordenados alfabéticamente por nombre.
15. Mostrar todos los movimientos que aprende pikachu.
16. Mostrar todos los movimientos que aprende pikachu por MT (tipo de aprendizaje).
17. Mostrar todos los movimientos de tipo normal que aprende pikachu por nivel.
18. Mostrar todos los movimientos de efecto secundario cuya probabilidad sea mayor al 30%.
19. Mostrar todos los pokemon que evolucionan por piedra. 20. Mostrar todos los pokemon que no
pueden evolucionar.
21. Mostrar la cantidad de los pokemon de cada tipo.

Bibliografía

Información sacada de las paginas:

- <https://www.oracle.com/ar/database/what-is-a-relational-database/>
- <https://www.geeksforgeeks.org/sql-tutorial/>
- <https://count.co/blog/posts/take-your-sql-from-good-to-great-part-3>
- <https://bookdown.org/paranedagarcia/database/modelo-relacional.html>
- <https://styde.net/relaciones-entre-tablas-de-bases-de-datos/>

CURSO DE PROGRAMACIÓN FULL STACK

ACCESO A BASES DE DATOS DESDE JAVA: JDBC



GUÍA DE PERSISTENCIA CON JDBC Y JPA

¿QUE ES JDBC?

Java™ Database Connectivity (JDBC) es la especificación JavaSoft de una interfaz de programación de aplicaciones (API) estándar que permite que los programas Java accedan a sistemas de gestión de bases de datos. La API JDBC consiste en un conjunto de interfaces y clases escritas en el lenguaje de programación Java.

Con estas interfaces y clases estándar, los programadores pueden escribir aplicaciones que se conecten con bases de datos, envíen consultas escritas en el lenguaje de consulta estructurada (SQL) y procesen los resultados.

Puesto que JDBC es una especificación estándar, un programa Java que utilice la API JDBC puede conectar con cualquier sistema de gestión de bases de datos (DBMS), siempre y cuando haya un driver para dicho DBMS en concreto.

COMPONENTES DE JDBC

En general, hay dos componentes principales de JDBC a través de los cuales puede interactuar con una base de datos. Son los que se mencionan a continuación:

JDBC Driver Manager: carga el driver específico de la base de datos en una aplicación para establecer una conexión con una base de datos. Se utiliza para realizar una llamada específica de la base de datos a la base de datos para procesar la solicitud del usuario.

API JDBC: Es un conjunto de *interfaces* y *clases*, que proporciona varios métodos e interfaces para una fácil comunicación con la base de datos. Proporciona dos paquetes de la siguiente manera que contiene las plataformas java SE y java EE para exhibir capacidades WORA (*write once run everywhere*).

Estos paquetes son:

1. `java.sql.*;`
2. `javax.sql.*;`

Las clases e interfaces principales de JDBC son:

- `java.sql.DriverManager`
- `java.sql.Connection`
- `java.sql.Statement`
- `java.sql.ResultSet`
- `java.sql.PreparedStatement`
- `javax.sql.DataSource`

ACCESO A BASES DE DATOS CON JDBC

JDBC nos permitirá acceder a bases de datos desde Java. Para ello necesitaremos contar con un SGBD (sistema gestor de bases de datos) además de un driver específico para poder acceder a este SGBD. La ventaja de JDBC es que nos permitirá acceder a cualquier tipo de base de datos, siempre que contemos con un driver apropiado para ella.

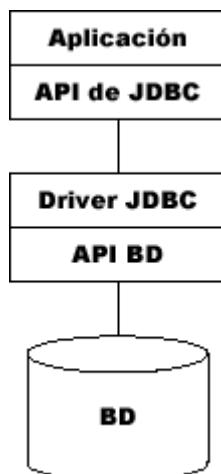


Figura 1: Arquitectura de JDBC

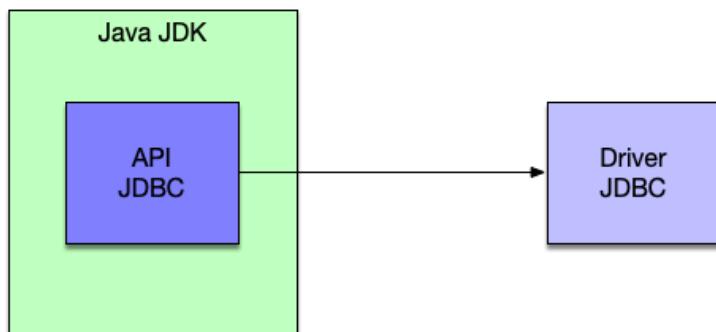
Como se observa en la Figura 1, cuando se construye una aplicación Java utilizando JDBC para el acceso a una base de datos, en la aplicación siempre se utiliza la API estándar de JDBC, y la implementación concreta de la base de datos será transparente para el usuario.

¿QUÉ ES UN DRIVER JDBC?

Vimos que dentro de los componentes de JDBC existe el Driver Manager que es el encargado de cargar el driver, pero que es el **driver** exactamente.

La API JDBC define las interfaces y clases Java™ que utilizan los programadores para conectarse con bases de datos y enviar consultas. Un driver JDBC implementa dichas interfaces y clases para un determinado proveedor de DBMS.

Un programa Java que utiliza la API JDBC carga el controlador especificado para el DBMS particular antes de conectar realmente con una base de datos. Luego la clase JDBC DriverManager envía todas las llamadas de la API JDBC al controlador cargado.



Cada base de datos debe aportar sus propias implementaciones y es ahí donde el Driver JDBC realiza sus aportes. El concepto de Driver hace referencia al conjunto de clases necesarias que implementa de forma nativa el protocolo de comunicación con la base de datos en un caso será Oracle y en otro caso será MySQL.

Por lo tanto para cada base de datos deberemos elegir su Driver .¿Cómo se encarga Java de saber cual tenemos que usar en cada caso?. Muy sencillo, Java realiza esta operación en dos pasos. En el primero registra el driver con la instrucción:

```
Class.forName("com.mysql.jdbc.Driver");
```

Una vez registrado el Driver , este es seleccionado a través de la propia cadena de conexión que incluye la información sobre cual queremos usar, en la siguiente línea podemos ver que una vez especificado el tipo de conexión define el Driver "MySQL"

```
String url= "jdbc:mysql://localhost:3306/biblioteca";
```

COMPONENTES DEL API DE JDBC

Nombramos cuales eran los componentes del API de JDBC, ahora los veremos en profundidad:

- **Driver:** Es el enlace de comunicaciones de la base de datos que maneja toda la comunicación con la base de datos. Normalmente, una vez que se carga el controlador, el desarrollador no necesita llamarlo explícitamente.
- **Connection:** Es una interfaz con todos los métodos para contactar una base de datos. El objeto de conexión representa el contexto de comunicación, es decir, toda la comunicación con la base de datos es solo a través del objeto de Connection.
- **Statement:** Encapsula una instrucción SQL que se pasa a la base de datos para ser analizada, compilada, planificada y ejecutada.
- **ResultSet:** Los ResultSet representan un conjunto de filas recuperadas debido a la ejecución de una consulta.

CONEXIÓN CON LA BASE DE DATOS

Para comunicarnos con una base de datos utilizando JDBC, se debe en primer lugar establecer una conexión con la base de datos a través del driver JDBC apropiado. El API JDBC especifica la conexión en la interfaz java.sql.Connection.

La clase DriverManager permite obtener objetos Connection con la base de datos.

Para conectarse es necesario proporcionar:

- **URL de conexión**, que incluye:
 - Nombre del host donde está la base de datos.
 - Nombre de la base de datos a usar.

- Nombre del usuario en la base de datos.
- Contraseña del usuario en la base de datos.

El siguiente código muestra un ejemplo de conexión y obtención de datos en JDBC a una base de datos MySQL:

```
Connection connection = null;
(...)

try {
    Class.forName("com.mysql.jdbc.Driver");
    String url = "jdbc:mysql://hostname/database-name";
    connection = DriverManager.getConnection(url, "user", "password");
}

} catch (SQLException ex) {
    connection = null;
    ex.printStackTrace();
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
```

En este ejemplo, primero se revisa el driver con la sentencia `Class.forName`. Después, la clase `DriverManager` intenta establecer una conexión con la base de datos `database-name` utilizando el driver `JDBC` que proporciona MySQL. Para poder acceder al RDBMS MySQL es necesario introducir un `username` y un `password` válidos. En el API JDBC, hay varios métodos que pueden lanzar la excepción `SQLException`.

Conexión a la Base de Datos (Objeto Connection)

Una vez cargado el driver apropiado para nuestro SGBD se debe establecer la conexión con la BD. Para ello se utiliza el siguiente método:

```
Connection con = DriverManager.getConnection(url, login, password);
```

La conexión a la BD está encapsulada en un objeto `Connection`, y para su creación se debe proporcionar la `url de la BD` y el `username` y `password` para acceder a ella. El formato de la url variará según el driver que se utilice.

El objeto `Connection` representa el contexto de una conexión con la base de datos, es decir:

- Permite obtener objetos `Statement` para realizar consultas SQL.
- Permite obtener metadatos acerca de la base de datos (nombres de tablas, etc.)
- Permite gestionar transacciones.

OBTENCIÓN DE DATOS DE LA BASE DE DATOS

Una vez que tengamos la conexión con el objeto Connection, la vamos a usar para crear un objeto **Statement**, este objeto recibe la consulta para ejecutarla y enviársela a la base de datos. La información que recibimos de la base de datos, va a ser capturada por el objeto **ResultSet** para después poder mostrar la información.

```
connection = DriverManager.getConnection(url, "user", "password");

String sql = "SELECT a, b, c FROM Table1";

Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery(sql);

while (rs.next()) {

    int x = rs.getInt("a");

    String s = rs.getString("b");

    double d = rs.getDouble("c");

    System.out.println("Fila = " + x + " " + s + " " + d);
}
```

Creación y ejecución de sentencias SQL (Objeto Statement)

Una vez obtenida la conexión a la BD, se puede utilizar para crear sentencias. Estas sentencias están encapsuladas en la clase Statement, y se pueden crear de la siguiente forma:

```
Statement stmt = con.createStatement();
```

- Los objetos Statement permiten realizar consultas SQL en la base de datos.
- Se obtienen a partir de un objeto Connection.

Tienen distintos métodos para hacer consultas:

- **executeQuery**: envía a la base de datos sentencias SQL para que recuperen datos y devuelvan un único objeto ResultSet. Es usado para leer datos (típicamente consultas SELECT).
- **executeUpdate**: para realizar actualizaciones que no devuelvan un ResultSet. Es usado para insertar, modificar o borrar datos (típicamente sentencias INSERT, UPDATE y DELETE).

Una vez obtenido este objeto se puede ejecutar sentencias utilizando su método executeQuery() al que se proporciona una cadena con la sentencia SQL que se quiere ejecutar:

```
stmt.executeQuery(sentenciaSQL);
```

Estas sentencias pueden utilizarse para consultas al base de datos.

Las sentencias SQL van a ser las mismas que veníamos trabajando en MySQL Workbench, se van a poner entre comillas dobles ya que el objeto Statement recibe String, como dato para las sentencias.

```
String sentenciaSQL = "SELECT nombre, apellido FROM alumnos";
```

Como podemos ver las sentencias van a tener la misma sintaxis que veníamos trabajando, la única diferencia, se va a presentar a la hora de trabajar con datos de tipo String y de tipo Date. Como estos datos suelen ir en comillas dobles en Java y en SQL, y nuestra sentencia ya está entre comillas dobles, deberemos poner los datos entre comillas simples para diferenciarlos.

String:

```
"SELECT nombre, apellido FROM Alumnos WHERE nombre = 'Agustín';
```

Date:

```
"SELECT nombre FROM Alumnos WHERE fechaNacimiento = '01-11-1990';
```

Obtención de datos (Objeto ResultSet)

Para obtener datos almacenados en la BD se utiliza una consulta SQL (query). La consulta se puede ejecutar utilizando el objeto Statement, con el método executeQuery() al que le se le pasa una cadena con la consulta SQL. Los datos resultantes se devuelven como un objeto ResultSet.

```
ResultSet result = stmt.executeQuery(sentenciaSQL);
```

La consulta SQL devolverá una tabla, que tendrá una serie de campos y un conjunto de registros, cada uno de los cuales consistirá en una tupla de valores correspondientes a los campos de la tabla.

El objeto ResultSet proporciona el acceso a los datos de estas filas mediante un conjunto de métodos *get* que permiten el acceso a las diferentes columnas de la filas. El método **ResultSet.next** se usa para moverse a la siguiente fila del ResultSet, convirtiendo a ésta en la fila actual.

El formato general de un ResultSet es una tabla con cabeceras de columna y los valores correspondientes devueltos por la “query”. Por ejemplo, si la “query” es *SELECT a, b, c FROM Table1*, el resultado tendrá una forma semejante a:

a	b	c
12345	Argentina	10,5
31245	Brasil	22,7
47899	Perú	56,7

El siguiente fragmento de código es un ejemplo de la ejecución de una sentencia SQL que devolverá una colección de filas, con la columna 1 como un int, la columna 2 como una String y la columna 3 como un real:

```
Statement stmt = connection.createStatement();
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (r.next()) {
    int i = r.getInt("a");
    String s = r.getString("b");
    double d = r.getDouble("c");
    // imprimimos los valores de la fila actual
    System.out.println("Fila = " + i + " " + s + " " + d);
}
```

Filas ResultSet

Un ResultSet mantiene un cursor que apunta a la fila actual de datos. El cursor se mueve una fila hacia abajo cada vez que se llama al método next. Inicialmente se sitúa antes de la primera fila, por lo que hay que llamar al método next para situarlo en la primera fila convirtiéndola en la fila actual. Las filas de ResultSet se recuperan en secuencia desde la fila más alta a la más baja.

Columnas ResultSet

Los métodos getX suministran los medios para recuperar los valores de las columnas de la fila actual. Dentro de cada fila, los valores de las columnas pueden recuperarse en cualquier orden, pero para asegurar la máxima portabilidad, deberían extraerse las columnas de izquierda a derecha y leer los valores de las columnas una única vez.

Puede usarse, o bien el nombre de la columna o el número de columna, para referirse a esta. Por ejemplo: si la columna **segunda** de un objeto ResultSet *rs* se denomina “**nombre**” y almacena valores de cadena, cualquiera de los dos ejemplos siguientes nos devolverá el valor almacenado en la columna.

```
String s = rs.getString("nombre");
String s = rs.getString(2);
```

Nótese que las columnas se numeran de izquierda a derecha comenzando con la columna 1. Además los nombres usados como input en los métodos getX son insensibles a las mayúsculas.

Algunos de los datos que podemos traer con el método get es:

Método	Explicación
getInt()	Sirve para obtener un numero entero de la base de datos
getLong()	Sirve para obtener un numero long de la base de datos
getDouble()	Sirve para obtener un numero real de la base de datos
getBoolean()	Sirve para obtener un booleano de la base de datos
getString()	Sirve para obtener una cadena de la base de datos
getDate()	Sirve para obtener una fecha de la base de datos

Optimización de sentencias

Cuando se quiere invocar una determinada sentencia repetidas veces, puede ser conveniente dejar esa sentencia preparada para que pueda ser ejecutada de forma más eficiente. Para hacer esto se utiliza la interfaz PreparedStatement, que podrá obtenerse a partir de la conexión a la BD de la siguiente forma:

```
PreparedStatement ps = con.prepareStatement("SELECT * FROM nombreTabla  
WHERE campo2 > 1200 AND campo2 < 1300");
```

Vemos que, a este objeto, a diferencia del objeto Statement visto anteriormente, se le proporciona la sentencia SQL en el momento de su creación, por lo que estará preparado y optimizado para la ejecución de dicha sentencia posteriormente.

Sin embargo, lo más común es que se necesite hacer variaciones sobre la sentencia, ya que normalmente no será necesario ejecutar repetidas veces la misma sentencia exactamente, sino variaciones de ella. Por ello, este objeto nos permite parametrizar la sentencia. Para ello se deben establecer las posiciones de los parámetros con el carácter '?' dentro de la cadena de la sentencia, tal como se muestra a continuación:

```
PreparedStatement ps = con.prepareStatement("UPDATE FROM nombreTabla  
SET campo1 = 'valor'  
WHERE campo2 > ? AND campo2 < ?");
```

En este caso se tienen dos parámetros, que representan un rango de valores en el cual se quiere actualizar. Cuando se ejecute esta sentencia, el campo1 de la tabla nombreTabla se establecerá a valor1 desde el límite inferior hasta límite superior indicado en el campo2.

Para dar valor a estos parámetros se utiliza el método setXXX() donde XXX será el tipo de los datos que asignamos al parámetro, indicando el número del parámetro (que empieza desde 1) y el valor que le queremos dar. Por ejemplo, para asignar valores enteros a los parámetros se debe hacer:

```
ps.setInt(1,1200);  
ps.setInt(2,1300);
```

Una vez asignados los parámetros, se puede ejecutar la sentencia llamando al método executeUpdate() del objeto PreparedStatement:

```
int n = ps.executeUpdate();
```

Igual que en el caso de los objetos Statement, se puede utilizar cualquier otro de los métodos para la ejecución de sentencias, executeQuery() o execute(), según el tipo de sentencia que se vaya a ejecutar.

PATRON DE DISEÑO DAO

Para la realización de los ejercicios y en los videos, vamos a trabajar JDBC usando el patrón de diseño DAO. Pero primero debemos saber que es un patrón de diseño.

¿QUE ES UN PATRON DE DISEÑO?

Un patrón de diseño es una solución probada que resuelve un tipo específico de problema en el desarrollo de software referente al diseño.

Las ventajas de usar un patrón de diseño son, que permiten tener el código bien organizado, legible y mantenible, además te permite reutilizar código y aumenta la escalabilidad en tu proyecto. En sí proporcionan una terminología estándar y un conjunto de buenas prácticas en cuanto a la solución en problemas de desarrollo de software.

PATRON DE DISEÑO DAO

A la hora de trabajar con JDBC y trabajar con base de datos, una de las grandes problemáticas al momento de acceder a los datos, es que la implementación y formato de la información puede variar según la fuente de los datos. Implementar la lógica de acceso a datos en la capa de lógica de negocio puede ser un gran problema, pues tendríamos que lidiar con la lógica de negocio en sí, más la implementación para acceder a los datos

Dado lo anterior, **el patrón DAO** propone separar por completo **la lógica de negocio de la lógica para acceder a los datos**, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.

CLASES

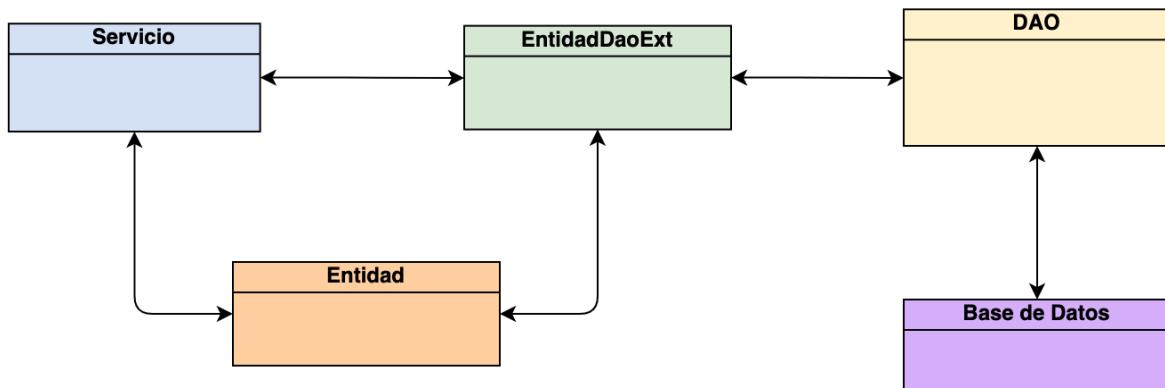
Esto lo vamos a lograr a través de cuatro clases:

Entidad: va a ser la clase que va a representar a la tabla que queremos trabajar de la base de datos. Va tener como atributos las columnas de la tabla de la base de datos.

Servicio o Business Service: va a tener toda la lógica de negocio del proyecto, usualmente se genera una para cada entidad. Es la que se encarga de obtener datos desde la base de datos y enviarla al cliente, o a su vez recibir la clase desde el cliente y enviar los datos al servidor, por lo general tiene todos los métodos CRUD (create, read, update y delete).

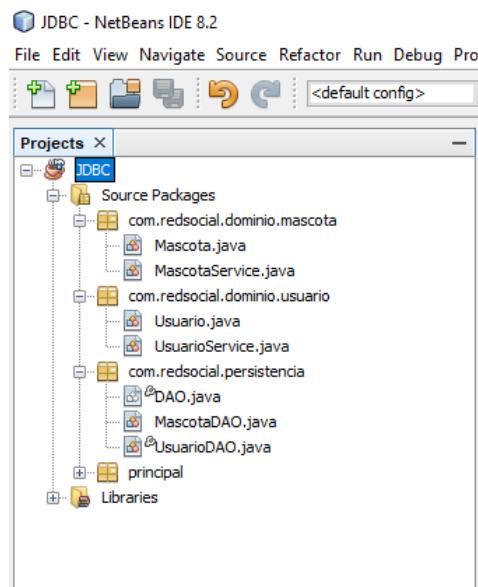
DAO: representa una capa de acceso a datos que oculta la fuente y los detalles técnicos para recuperar los datos. Esta clase va a ser la encargada de comunicarse con la base de datos, de conectarse con la base de datos, enviar las consultas y recuperar la información de la base de datos.

EntidadDaoExt: esta clase va a extender de la clase DAO y se va encargar de generar las sentencias para enviar a la clase DAO, como un insert, select, etc. Y si estuviéramos haciendo un select, sería también, la encargada de recibir la información, que recupera la clase DAO de la base de datos sobre una entidad, para después enviarla al servicio, que será la encargada de imprimir dicha información. Este es un objeto plano que implementa el patrón Data Transfer Object (DTO), el cual sirve para transmitir la información entre el DAO y el Business Service.



PAQUETES

Esto representado en un proyecto tendría las siguientes clases y paquetes:



Nota: estos conceptos van a poder verlos en más profundidad en los videos, donde verán clase por clase y tendrán un ejemplo para descargar y poder verlo ustedes mismos también.

EJERCICIOS DE APRENDIZAJE

Para la realización de los ejercicios que se describen a continuación, es necesario descargar el archivo persistencia.zip que contiene el material necesario para realizar esta práctica. Por otra parte, se recomienda consultar el [Instructivo Conexión Netbeans – MySql](#) para poder conectarnos correctamente a la base de datos

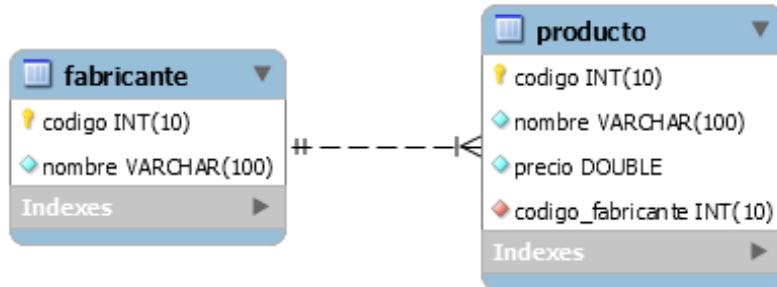


VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Tienda

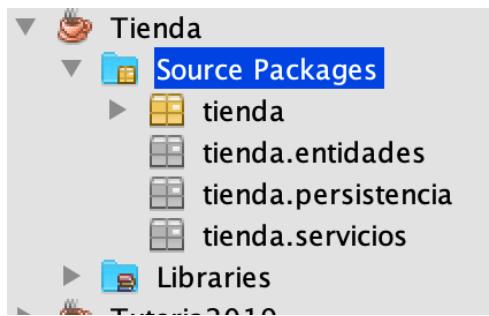
Nos han pedido que hagamos una aplicación Java para una tienda con sus productos. El objetivo es realizar consultas para saber el stock de ciertos productos o que productos hay, etc. Utilizando el lenguaje JAVA, una base de datos MySQL y JDBC para realizar la ejecución de operaciones sobre la base de datos (BD).

Para este ejercicio vamos a usar el script de la base de datos llamada "tienda.sql" que lo trabajamos en la guía de MySql, igualmente lo van a encontrar dentro del archivo *persistencia.zip*. Deberá obtener un diagrama de entidad relación igual al que se muestra a continuación:



Paquetes del Proyecto Java

Crear un nuevo proyecto en Netbeans del tipo "Java Application" con el nombre Tienda y agregar dentro 3 paquetes, a uno se lo llamará entidades, al otro se le llamará servicios y al otro persistencia:



Para crear los paquetes de esta manera, se deben crear desde el paquete principal, sería nos paramos en el paquete tienda -> Click derecho -> New Java Package y creamos los paquetes. También es importante agregar en “Libraries” la librería “MySQL JDBC Driver” para permitir conectar la aplicación de Java con la base de datos MySQL. Esto se explica en el **Instructivo**.

Paquete persistencia

En este paquete estará la clase DAO encarga de conectarse con la base de datos y de comunicarse con la base de datos para obtener sus datos. Además, estará las clases de EntidadDaoExt para cada entidad / tabla de nuestro proyecto.

Es importante tener la conexión creada a la base de datos, como lo explica el Instructivo en la pestaña de **Services** en Netbeans.

Paquete entidades:

Dentro de este paquete se deben crear todas las clases necesarias que vamos a usar de la base de datos. Por ejemplo, una de las clases a crear dentro de este paquete es la clase “Producto.java” con los siguientes atributos:

- private int codigo;
- private String nombre;
- private double precio;
- private int codigoFabricante;

Agregar a cada clase el/los constructores necesarios y los métodos públicos getters y setters para poder acceder a los atributos privados de la clase. La llave foránea se pondrá como dato nada más, no como objeto.

Paquete servicios:

En este paquete se almacenarán aquellas clases que llevarán adelante lógica del negocio. En general se crea un servicio para administrar cada una de las entidades y algunos servicios para manejar operaciones muy específicas como las estadísticas.

Realizar un menú en Java a través del cual se permita elegir qué consulta se desea realizar. Las consultas a realizar sobre la BD son las siguientes:

- a) Lista el nombre de todos los productos que hay en la tabla producto.
- b) Lista los nombres y los precios de todos los productos de la tabla producto.
- c) Listar aquellos productos que su precio esté entre 120 y 202.
- d) Buscar y listar todos los Portátiles de la tabla producto.
- e) Listar el nombre y el precio del producto más barato.
- f) Ingresar un producto a la base de datos.
- g) Ingresar un fabricante a la base de datos
- h) Editar un producto con datos a elección.

EJERCICIOS DE APRENDIZAJE EXTRAS

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Estancias en el extranjero

Nos han pedido que hagamos una aplicación Java de consola para una pequeña empresa que se dedica a organizar estancias en el extranjero dentro de una familia. El objetivo es el desarrollo del sistema de reserva de casas para realizar estancias en el exterior, utilizando el lenguaje JAVA, una base de datos MySQL y JDBC para realizar la ejecución de operaciones sobre la base de datos (BD).

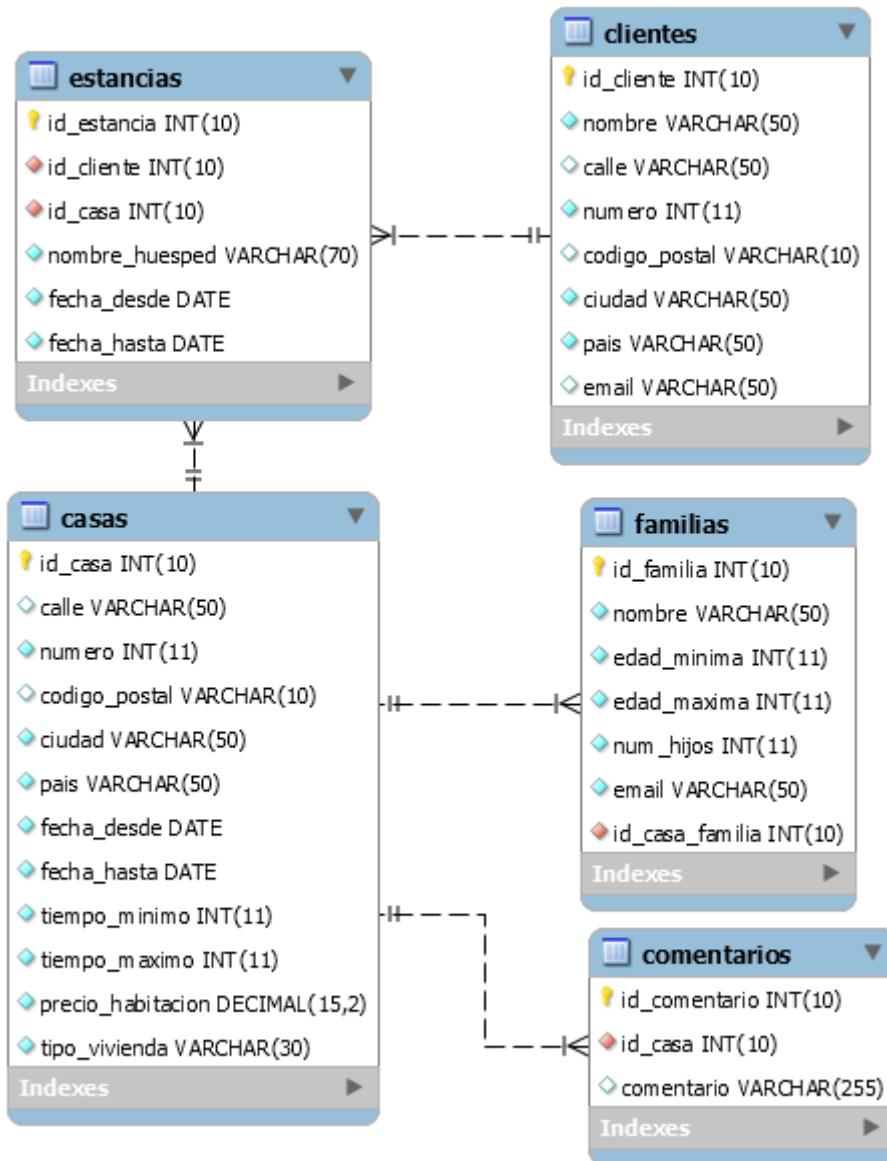
Creación de la Base de Datos MySQL

La información que se desea almacenar en la base de datos es la siguiente:

- Se tienen contactos con familias de diferentes países que ofrecen alguna de las habitaciones de su hogar para acoger algún chico (por un módico precio). De cada una de estas familias se conoce el nombre, la edad mínima y máxima de sus hijos, número de hijos y correo electrónico.
- Cada una de estas familias vive en una casa, de la que se conoce la dirección (calle, numero, código postal, ciudad y país), el periodo de disponibilidad de la casa (fecha_desde, fecha_hasta), la cantidad de días mínimo de estancia y la cantidad máxima de días, el precio de la habitación por día y el tipo de vivienda.
- Se dispone también de información de los clientes que desean mandar a sus hijos a alguna de estas familias: nombre, dirección (calle, numero, código postal, ciudad y país) y su correo electrónico.
- En la BD se almacena información de las reservas y estancias realizadas por alguno de los clientes. Cada estancia o reserva la realiza un cliente, y además, el cliente puede reservar varias habitaciones al mismo tiempo (por ejemplo para varios de sus hijos), para un periodo determinado (fecha_llegada, fecha_salida).
- El sistema debe también almacenar información brindada por los clientes sobre las casas en las que ya han estado (comentarios).

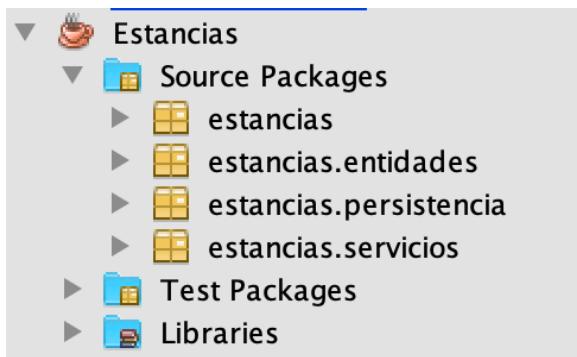
Según todas estas especificaciones se debe realizar:

Para este ejercicio vamos a usar el script de la base de datos llamada "estancias_exterior.sql" lo van a encontrar en el archivo persistencia.zip Deberá obtener un diagrama de entidad relación igual al que se muestra a continuación:



Paquetes del Proyecto Java

Crear un nuevo proyecto en Netbeans del tipo "Java Application" con el nombre Estancias y agregar dentro 3 paquetes, a uno se lo llamará entidades, al otro se lo llamará servicios y al otro persistencia:



Paquete persistencia

En este paquete estará la clase DAO encarga de conectarse con la base de datos y de comunicarse con la base de datos para obtener sus datos. Además, estará las clases de EntidadDaoExt para cada entidad / tabla de nuestro proyecto.

Es importante tener la conexión creada a la base de datos, como lo explica el Instructivo en la pestaña de **Services** en Netbeans.

Agregar en "Libraries" la librería "MySQL JDBC Driver" para permitir conectar la aplicación de Java con la base de datos MySQL.

Paquete entidades:

Dentro de este paquete se deben crear todas las clases necesarias que queremos persistir en la base de datos. Por ejemplo, una de las clases a crear dentro de este paquete es la clase "Familia.java" con los siguientes atributos:

- private int id;
- private String nombre;
- private int edad_minima;
- private int edad_maxima;
- private int num_hijos;
- private String email;

Agregar a cada clase el/los constructores necesarios y los métodos públicos getters y setters para poder acceder a los atributos privados de la clase.

Paquete servicios:

En este paquete se almacenarán aquellas clases que llevarán adelante lógica del negocio. En general se crea un servicio para administrar cada una de las entidades y algunos servicios para manejar operaciones muy específicas como las estadísticas.

Para realizar las consultas con la base de datos, dentro del paquete servicios, creamos las clases para cada una de las entidades con los métodos necesarios para realizar consultas a la base de datos. Una de las clases a crear en este paquete será: FamiliaServicio.java, y en esta clase se implementará, por ejemplo, un método para listar todas las familias que ofrecen alguna habitación para realizar estancias.

Realizar un menú en java a través del cual se permita elegir qué consulta se desea realizar. Las consultas a realizar sobre la BD son las siguientes:

- a) Listar aquellas familias que tienen al menos 3 hijos, y con edad máxima inferior a 10 años.
- b) Buscar y listar las casas disponibles para el periodo comprendido entre el 1 de agosto de 2020 y el 31 de agosto de 2020 en Reino Unido.
- c) Encuentra todas aquellas familias cuya dirección de mail sea de Hotmail.
- d) Consulta la BD para que te devuelva aquellas casas disponibles a partir de una fecha dada y un número de días específico.

- e) Listar los datos de todos los clientes que en algún momento realizaron una estancia y la descripción de la casa donde la realizaron.
- f) Listar todas las estancias que han sido reservadas por un cliente, mostrar el nombre, país y ciudad del cliente y además la información de la casa que reservó. La que reemplazaría a la anterior
- g) Debido a la devaluación de la libra esterlina con respecto al euro se desea incrementar el precio por día en un 5% de todas las casas del Reino Unido. Mostar los precios actualizados.
- h) Obtener el número de casas que existen para cada uno de los países diferentes.
- i) Busca y listar aquellas casas del Reino Unido de las que se ha dicho de ellas (comentarios) que están 'limpias'.
- j) Insertar nuevos datos en la tabla estancias verificando la disponibilidad de las fechas.

Para finalizar, pensar junto con un compañero cómo sería posible optimizar las tablas de la BD para tener un mejor rendimiento.

CURSO DE PROGRAMACIÓN FULL STACK

ACCESO A BASES DE DATOS DESDE JAVA: JPA



GUÍA DE PERSISTENCIA CON JPA

PERSISTENCIA EN JAVA CON JPA

JPA (Java Persistence API) es la propuesta estándar que ofrece Java para implementar un Framework Object Relational Mapping (ORM), que permite interactuar con la base de datos por medio de objetos, de esta forma, JPA es el encargado de convertir los objetos Java en instrucciones para el Manejador de Base de Datos (MDB). El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional).

Cuando empezamos a trabajar con bases de datos en Java utilizamos el API de JDBC el cual nos permite realizar consultas directas a la base de datos a través de consultas SQL nativas. JDBC por mucho tiempo fue la única forma de interactuar con las bases de datos, pero representaba un gran problema y es que Java es un lenguaje orientado a objetos y se tenían que convertir los atributos de las clases en una consulta SQL como SELECT, INSERT, UPDATE, DELETE, etc. Lo que ocasionaba un gran esfuerzo de trabajo y provocaba muchos errores en tiempo de ejecución, debido principalmente a que las consultas SQL se tenían que generar frecuentemente al vuelo.

JPA es una especificación, es decir, no es más que un documento en el cual se plasman las reglas que debe de cumplir cualquier proveedor que desee desarrollar una implementación de JPA, de tal forma que cualquier persona puede tomar la especificación y desarrollar su propia implementación de JPA. Existen varios proveedores como lo son los siguientes:

- Hibernate
- ObjectDB
- EclipseLink
- OpenJPA

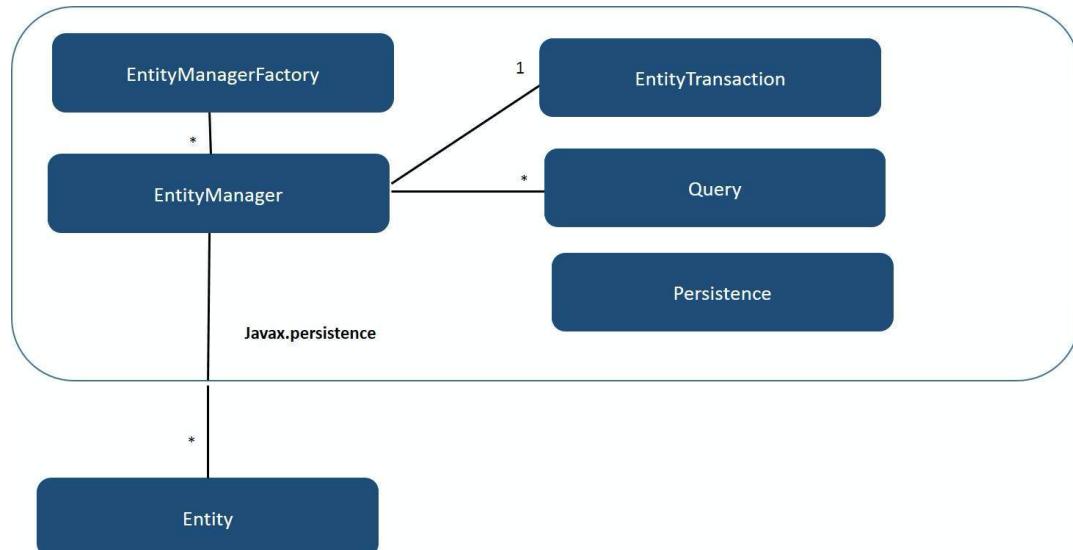
PERSISTENCIA DE OBJETOS

JPA representa una simplificación del modelo de programación de persistencia. La especificación JPA define explícitamente la correlación relacional de objetos, en lugar de basarse en implementaciones de correlación específicas del proveedor. JPA crea un estándar para la importante tarea de la correlación relacional de objetos mediante la utilización de anotaciones o XML para correlacionar objetos con una o más tablas de una base de datos. Para simplificar aún más el modelo de programación de persistencia:

- La API EntityManager puede actualizar, recuperar, eliminar o aplicar la persistencia de objetos de una base de datos.

- JPA proporciona un lenguaje de consulta, que amplía el lenguaje de consulta EJB independiente, conocido también como JPQL, el cual puede utilizar para recuperar objetos sin grabar consultas SQL específicas en la base de datos con la que está trabajando.
- El programador no necesita programar código JDBC ni consultas SQL.
- El entorno realiza la conversión entre tipos Java y tipos SQL.
- El entorno crea y ejecuta las consultas SQL necesarias.

ARQUITECTURA JPA



La arquitectura de JPA está diseñada para gestionar Entidades y las relaciones que hay entre ellas. A continuación, detallamos los principales componentes de la arquitectura

Entity: Clase Java simple que representa una fila en una tabla de base de datos con su formato más sencillo. Los objetos de entidades pueden ser clases concretas o clases abstractas. Podemos decir que cada Entidad corresponderá con una tabla de nuestra Base de Datos

Persistence: Clase con métodos estáticos que nos permiten obtener instancias de EntityManagerFactory.

EntityManagerFactory: Es una factoría de EntityManager. Se encarga crear y gestionar múltiples instancias de EntityManager

EntityManager: Es una interface que gestiona las operaciones de persistencia de las entidades, ya sea crear, editar, eliminar, traer de la base de datos una entidad, etc. Es la base de todo proyecto de JPA. A su vez trabaja como factoría de las Querys.

Query: Es una interface para obtener la relación de objetos que cumplen un criterio

EntityTransaction: Agrupa las operaciones realizadas sobre un EntityManager en una única transacción de Base de Datos.

MAPEO CON ANOTACIONES

Como sabemos las bases de datos relacionales almacenan la información mediante tablas, filas, y columnas, de manera que para almacenar un objeto en la base de datos, hay que realizar una correlación entre el sistema orientado a objetos de Java y el sistema relacional de nuestra base de datos. JPA nos permite realizar dicha correlación de forma sencilla, realizando nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos. Esta conversión se llama **ORM** (Object Relational Mapping - Mapeo Relacional de Objetos), y puede configurarse a través de metadatos (**anotaciones**). A estos objetos, las cuales son clases comunes y corrientes, los llamaremos desde ahora **entidades**.

Las anotaciones nos permiten configurar el mapeo de una entidad dentro del mismo archivo donde se declara la clase, de este modo, relaciona las clases contra las tablas y los atributos contra las columnas. Mediante las anotaciones vamos a explicarle al ORM, como transformar la entidad en una tabla de base de datos.

Las anotaciones comienzan con el símbolo "@" seguido de un identificador. Las anotaciones son utilizadas antes de la declaración de clase, propiedad o método. A continuación, se detallan las principales:

@Entity: Declara la clase como una Entidad

@Table: Declara el nombre de la Tabla con la que se mapea la Entidad

@Id: Declara un atributo como la clave primaria de la Tabla

@GeneratedValue: Declara como el atributo que va a ser la clave primaria va a ser inicializada. Manualmente, Automático o a partir de una secuencia.

@Column: Declara que un atributo se mapea con una columna de la tabla

@Enumerated: Declara que un atributo es de alguno de los valores definidos en un Enumerado (lista de valores constantes). Los valores de un tipo enumerado tienen asociado implícitamente un tipo ordinal que será asociada a la propiedad de este tipo.

@Temporal: Declara que se está tratando de un atributo que va a trabajar con fechas, entre paréntesis, debemos especificarle que estilo de fecha va a manejar en la base de datos:

@Temporal(TemporalType.DATE), @Temporal(TemporalType.TIME),
@Temporal(TemporalType.TIMESTAMP)

DECLARAR ENTIDADES CON @ENTITY

Como ya discutimos hace un momento, **las entidades son simples clases Java** como cualquier otra, sin embargo, JPA debe de ser capaz de identificar que clases son entidades para de esta forma poder administrarlas y convertirlas en tablas. Es aquí donde nace la importancia de la anotación **@Entity**, esta anotación se debe de definir a nivel de clase y sirve únicamente para indicarle a JPA que esa clase es un Entity, veamos el siguiente ejemplo:

```
public class Empleado {  
    private Long id;  
    private String nombre;  
}
```

En el ejemplo vemos una clase común y corriente la cual representa a un Empleado, hasta este momento la clase Empleado, no se puede considerar una entidad, pues a un no tiene la anotación `@Entity` que la señale como tal. Ahora bien, si a esta misma clase le agregamos la anotación `@Entity` le estaremos diciendo a JPA que esta clase es una entidad y deberá ser administrada por el EntityManager, veamos el siguiente ejemplo:

```
@Entity  
public class Empleado {  
    private Long id;  
    private String nombre;  
}
```

En este punto la clase ya se puede considerar una Entidad.

DEFINIR LLAVE PRIMARIA CON @ID

Al igual que en las tablas, las entidades también requieren un identificador o clave primaria(ID). Dicho identificador deberá de diferenciar a la entidad del resto. Como regla general, todas las entidades definir un ID, de lo contrario provocaremos que el EntityManager marque error a la hora de instanciarlo.

El ID es importante porque será utilizado por el EntityManager a la hora de persistir un objeto, y es por este que puede determinar sobre que registro hacer el select, update o delete.

```
@Entity  
public class Empleado {  
    @Id  
    private Long id;  
    private String nombre;  
}
```

Se ha agregado `@Id` sobre el atributo `id`, de esta manera, cuando el EntityManager inicie sabrá que el campo `id` es el Identificador de la clase Empleado.

ANOTACIÓN @GENERATEDVALUE

Esta anotación se utiliza cuando el ID es autogenerado (Identity) como en el caso de MySQL. JPA cuenta con la anotación `@GeneratedValue` para indicarle a JPA que regla de autogeneración de la lleva primaria vamos a utilizar.

Identity

Esta estrategia es la más fácil de utilizar pues solo hay que indicarle la estrategia y listo, no requiere nada más, JPA cuando persista la entidad **no enviará este valor**, pues asumirá que la columna **es auto generada**. Esto provoca que el contador de la columna **incremente en 1** cada vez que un nuevo objeto es insertado.

```
@Entity  
public class Empleado {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String nombre;  
  
}
```

MAPEO DE FECHAS CON @TEMPORAL

Mediante la anotación @Temporal es posible mapear las fechas con la base de datos de una forma simple. Una de las principales complicaciones cuando trabajamos con fecha y hora es determinar el formato empleado por el manejador de base de datos. Sin embargo, esto ya no será más problema con @Temporal.

Mediante el uso de @Temporal es posible determinar si el atributo almacena **Hora**, **Fecha** u **Hora y Fecha**. Para esto podemos utilizar la clase Date o Calendar. Se pueden establecer tres posibles valores para la anotación:

DATE: Acotara el campo solo a la Fecha, descartando la hora.

`@Temporal(TemporalType.DATE)`

TIME: Acotara el campo solo a la Hora, descartando a la fecha.

`@Temporal(TemporalType.TIME)`

TIMESTAMP: Toma la fecha y hora.

`@Temporal(TemporalType.TIMESTAMP)`

Ejemplo:

```
@Entity  
public class Persona {  
  
    @Id  
    private Long id;  
  
    private String nombre;  
  
    @Temporal(TemporalType.DATE)  
    private Date fechaNacimiento;  
  
}
```

LAS RELACIONES

Como sabemos en Java, los objetos pueden estar relacionados entre sí mediante las **relaciones entre clases** y sabemos que en MySQL las tablas tienen **4 tipos de relaciones** posibles. Entonces, supongamos que **tenemos dos objetos** que están relacionados entre sí y queremos que **esa relación también esté representada en las tablas**.

Es por esto que JPA, nos da cuatro anotaciones para cuando tenemos una relación entre dos clases en Java y **le queremos explicar a la base de datos**, que **tipo de relación** tendrán **las tablas entre sí**. Estas anotaciones solo van a afectar a las tablas, sirven para especificar como se van a relacionar los registros de una tabla, con los registros de otra tabla. Recordemos que las anotaciones cumplen el propósito de "traducir" nuestro código de Java para que lo entienda la base de datos, por lo que las anotaciones, no van a afectar nunca a nuestro código.

Las anotaciones que nos da JPA, son los tipos de relaciones entre tablas que vimos en la guía de MySQL. Estas anotaciones son:

- **@OneToOne**: relación entre tablas uno a uno
- **@OneToMany**: relación entre tablas uno a muchos
- **@ManyToOne**: relación entre tablas muchos a uno
- **@ManyToMany**: relación entre tablas muchos a muchos

@ONETOONE

Entonces, supongamos que tenemos dos clases, Curso y Profesor, entre las cuales existe una relación de **1 a 1** en Java. Un Curso por lo tanto tiene 1 Profesor y un Profesor pertenece a un Curso. Esto en nuestro código Java sería algo así:

```
@Entity
public class Profesor {

    @Id
    private Long id;

    private String nombre;

}

@Entity
public class Curso {

    @Id
    private Long id;

    private Integer precio;

    private String nombreCurso;

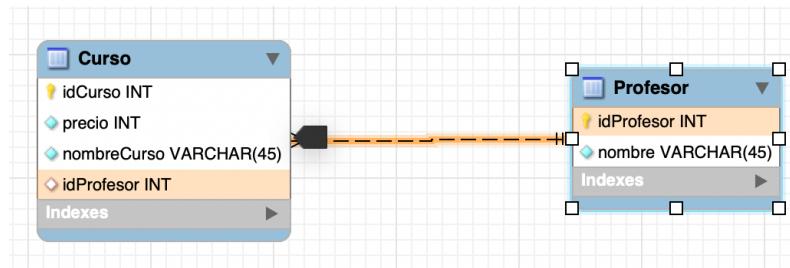
    private Profesor profesor;

}
```

Por ahora lo único que hemos creado es una referencia a la clase Profesor sin utilizar JPA para nada. El siguiente paso será anotar la clase con anotaciones de JPA para que se construya la relación a nivel de persistencia. Decidimos que la relación entre tablas también sea 1 a 1, por lo que pondremos la anotación @OneToOne

```
@Entity  
public class Curso {  
  
    @Id  
    private Long id;  
  
    private Integer precio;  
  
    private String nombreCurso;  
  
    @OneToOne  
    private Profesor profesor;  
}
```

Con esta anotación nos quedará unas tablas en MySQL de la siguiente manera:



Como podemos observar en la tabla Curso, existe una llave foránea de la tabla Profesor, de la misma manera que en nuestra clase Curso existe un objeto de tipo Profesor.

La relación OneToOne en nuestras tablas va a especificar que para un registro de un Curso, solo hay un registro de un Profesor. En otras palabras sería que un Curso no puede tener dos Profesores o que a cada Curso, solo podemos asignarle/persistir un Profesor.

A partir de estos conceptos definimos 4 tipos principales de relaciones entre entidades:

@MANYTOONE

Usamos esta anotación cuando queremos que entre nuestras tablas haya una relación de **muchos a uno**. Por ejemplo, muchos Álbumes pueden pertenecer a un Autor. Esta relación se representa en Java de la siguiente manera:

```
@ManyToOne  
private Autor autor;
```

La relación ManyToOne en nuestras tablas va a especificar que para uno o varios registros de Álbumes va a haber un Autor. En otras palabras sería que uno o más Álbumes van a tener el mismo Autor. Esto nos daría la posibilidad de que, a uno o muchos Álbumes asignarle el mismo Autor, esta posibilidad no existe con la OneToOne, ya que a cada registro solo le podemos asignar **un** registro.

Es importante entender que la manera que pensamos nuestras relaciones en Java es, la primera palabra de la anotación se aplica a la clase que tendrá la relación y la segunda palabra se aplica a la clase que será el atributo de la otra clase. En el ejemplo anterior, vimos que el Many se aplicaba a los Álbumes y el One al Autor.

Como el Autor solo va a ser 1, nosotros lo representamos en Java con un solo objeto, si fueran muchos autores, pondríamos una colección, esto es igual a las relaciones entre clases normales de Java

@ONETOMANY

Usamos esta anotación cuando tenemos una relación de **1 a n** entre tablas. Por ejemplo, un Curso tiene muchos Alumnos. Es importante recordar que cuando hablamos de que una clase tiene **muchos(Many)** o n de algo, usamos una colección para representar esa relación en Java, sino usaríamos un solo objeto.

@OneToMany

```
private List<Alumno> alumnos;
```

La relación OneToMany en nuestras tablas va a especificar que para un registro de un Curso, va a haber varios registros de Alumnos. En otras palabras sería que un Curso puede tener **uno o más** Alumnos. Esto nos daría la posibilidad de, al mismo registro de Curso asignarles varios Alumnos.

@ManyToMany: usamos esta anotación para tablas que están relacionadas con muchos elementos de un tipo determinado, pero al mismo tiempo, estos últimos registros no son exclusivos de un registro en particular, si no que pueden ser parte de varios. Por lo tanto, tenemos una Entidad A, la cual puede estar relacionada como **muchos** registros de la Entidad B, pero al mismo tiempo, la Entidad B puede pertenecer a **varias** instancias de la Entidad A.

Algo muy importante a tomar en cuenta cuando trabajamos con relaciones @ManyToMany o @OneToMany, es que en realidad este tipo de relaciones no existen físicamente en la base de datos, y en su lugar, es necesario crear una **tabla intermedia** que relate las dos entidades.

@ManyToMany

```
private List<Tarea> tareas;
```

Nota: que clase va a tener la referencia a la otra clase va a ser decisión del programador.

TABLA INTERMEDIA

El concepto de tabla intermedia se presenta cuando tenemos una entidad que va a pertenecer a varias instancias de otra entidad. Por ejemplo, un **Empleado** o varios **Empleados** pueden hacer *una o varias Tareas* y a su vez, *una o varias Tareas* pueden ser realizadas por *uno o varios Empleados*. Esto en SQL sería que un registro tiene varios registros asignados a él mismo.

El problema es que en SQL solo podemos poner un dato por columna, supongamos que el **Empleado** tiene tres **Tareas** con los identificadores(id) 1,2,3. Nosotros no podemos tener una columna que tenga tres valores separados. Lo que podríamos hacer es que se repita el registro de **Empleado** tres veces con los tres identificadores, pongamos un ejemplo de una tabla que cumpla ese requisito:

Empleado			
ID	Nombre	Apellido	IdTarea
1	Adriana	Cardello	1
1	Adriana	Cardello	2
1	Adriana	Cardello	3

Esto parecería estar bien, pero si pensamos en las reglas de SQL, no podemos tener dos identificadores iguales en el mismo registro y en nuestra tabla **Empleado** hay tres veces el mismo identificador para el **Empleado**, ya que necesitamos que se repita.

Por lo que, para este tipo de relación se crea una tabla intermedia conocida como tabla asociativa. Por convención, el nombre de esta tabla debe estar formado por el nombre de las tablas participantes (en singular y en orden alfabético) separados por un guion bajo (_).

Esta tabla está compuesta por las claves primarias de las tablas que se relacionan con ella, así se logra que la relación sea de uno a muchos, en los extremos, de modo tal que la relación se lea:

Un **empleado** tiene muchas **tareas**.

y

Una **tarea** puede ser hecha por muchos **empleados**.

Las tablas se verían así:

Empleado		
ID	Nombre	Apellido
1	Adriana	Cardello
2	Mariela	Arbona
3	Agustín	Santamarina

Tarea	
ID	Nombre
1	Limpiar Platos
2	Ordenar
3	Barrer

Tabla Intermedia:

Empleado_Tarea	
Id_Empleado	Id_Tarea
1	2
1	1
2	3
2	2

Como podemos observar la tabla intermedia solo tiene dos columnas, el id del empleado y el id de la tarea. Esta tabla no toma las columnas como llaves primarias, sino como llaves foráneas, esto nos permite repetir los valores para asignar las tareas al empleado.

Las tablas en MySQL se verían así:



Notemos que en la tabla Empleado no existe una columna que haga referencia a Tarea, ni en Tarea a Empleado, si no que es la tabla intermedia la encargada de hacer el cruce entre las dos tablas.

Nota: recordemos que esta tabla intermedia se va a generar con las anotaciones @OneToMany y @ManyToMany.

RELACIONES JPA Y UML

En Java, nosotros tenemos dos posibles relaciones entre clases, **uno a uno o de uno a muchos**, además son las dos que podemos representar en código. En cambio en MySQL tenemos cuatro tipo de relaciones, el problema es que hay relaciones de MySQL que no podemos representar en Java.

Por ejemplo, la relación muchos a uno, no podemos representarla en Java, ya que una clase no puede ser un List para representar el muchos. Por lo que Java al ver que hay una referencia a una clase de un solo objeto, en el caso, por ejemplo, de la ManyToOne, esto lo va a tomar como una relación de uno a uno.

Esto va a generar que cuando veamos un UML de nuestro proyecto JPA, no veamos las relaciones ManyToOne o ManyToMany, ya que, como dijimos, en Java solo existen las uno a uno o uno a muchos

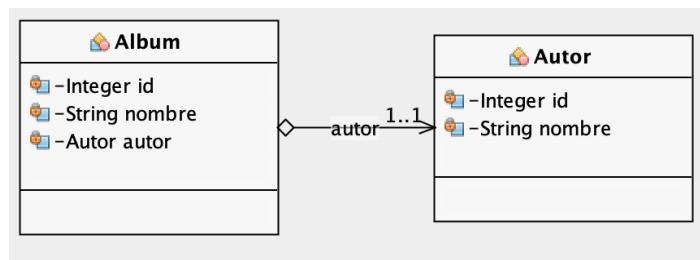
Pongamos un ejemplo, tenemos la Clase **Autor** y la clase **Álbum**, vamos a decir que para **muchos** Álbunes existe **el mismo** autor, por lo que sería una **ManyToOne**. Si la representásemos en código sería así:

```
@ManyToOne  
private Autor autor;
```

Ahora si tuviéramos una **OneToOne**, la representación en código sería esta:

```
@OneToOne  
private Autor autor;
```

Si miramos el código, podemos observar que para las dos relaciones escribimos lo mismo. Entonces, para Java la relación **ManyToOne** la va a representar como una **OneToOne** (1...1) en UML y lo mismo nos pasaría con la **OneToMany** y la **ManyToOne**.



Por lo que si nos encontramos con un UML que tiene una relación uno a uno (1...1) y nos piden decidir que relación entre tablas ponerle, deberíamos considerar, que puede ser una **OneToOne** o una **ManyToOne**; o que si tiene una relación uno a muchos (1...n), puede ser una **OneToMany** o una **ManyToOne**.

Esto es porque, como habíamos previamente las relaciones muchos a uno y muchos a muchos son propias de MySQL, no de Java. Entonces es importante, porque a la hora de pensar en que anotaciones le vamos a poner a nuestras entidades, pensemos en las tablas, ya que estamos trabajando sobre como va a ser la relación entre las tablas y no las clases.

PERSISTENCIA EN JPA CON ENTITYMANAGER

Ahora que entendemos como **a través de las anotaciones del ORM podemos unificar las tablas de la base de datos con los objetos, que pasan a llamarse entidades**. Ahora tenemos que ver como hacemos para poder guardar, editar, eliminar, etc. a esas entidades en la base de datos.

JPA tiene como interface medular al EntityManager, el cual es el componente que se encarga de controlar el ciclo de vida de todas las entidades definidas en la unidad de persistencia (como configurar la unidad de persistencia se va a encontrar en el Moodle en una PDF aparte).

El EntityManager nos dará la posibilidad de poder crear, borrar, actualizar y consultar todas estas entidades de la base de datos. También es la clase por medio de la cual se controlan las transacciones. Los EntityManager son configurados siempre a partir de las unidades de persistencia definidas en el archivo persistence.xml.

```
EntityManager em =  
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").create  
EntityManager();
```

En esta línea se puede ver que se obtiene una instancia de la **Interfaz EntityManagerFactory**, mediante la clase **Persistence**, esta última recibe como parámetro el nombre de la unidad de persistencia que definimos en el archivo persistence.xml. Una vez con el EntityManagerFactory se obtiene una instancia de EntityManager para finalmente ser retornada para ser utilizada.

OPERACIONES ENTITYMANAGER

Las entidades pueden ser cargadas, creadas, actualizadas y eliminadas a través del EntityManager. Vamos a mostrar los métodos del EntityManager que nos permiten lograr estas operaciones.

Persist()

Este método nos deja **persistir** una entidad en nuestra base de datos. Persistir es la acción de preservar la información de un objeto de forma permanente, en este caso en una base de datos, pero a su vez también se refiere a poder recuperar la información del mismo para que pueda ser nuevamente utilizado.

Antes de ver como persistimos un objeto, también tenemos que entender el concepto de transacciones, ya que para persistir un objeto en la base de datos, la operación debe estar marcada como una transacción.

Una transacción es un conjunto de operaciones sobre una base de datos, que suelen crear, editar o eliminar un registro de la base de datos, que se deben ejecutar como una unidad. Por lo que una consulta a la base de datos no se la considera una transacción.

Entendiendo esto veamos un ejemplo del método Persist():

```
// Creamos un EntityManager  
EntityManager em =  
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").create  
EntityManager();  
  
//Creamos un objeto Alumno y le asignamos un nombre  
Alumno alumno = new Alumno();  
a1.setNombre("Nahuel");  
  
//Iniciamos una transacción con el método getTransaction().begin();  
em.getTransaction().begin();  
  
//Persistimos el objeto  
em.persist(alumno);  
  
//Terminamos la transacción con el método commit. Commit en programación  
//significa confirmar un conjunto de cambios, en este caso persistir el  
//objeto  
em.getTransaction().commit();
```

Find()

Este método se encarga de buscar y devolver una Entidad en la base de datos, a través de su clave primaria(Id). Para ello necesita que le pasemos la clave y el tipo de Entidad a buscar.

```
// Creamos un EntityManager  
EntityManager em =  
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").create  
EntityManager();  
  
// Usamos el método find para buscar una persona con el id 123 en nuestra  
base de datos  
  
Persona persona = em.find(Persona.class, 123);
```

De esta manera podremos obtener una Persona de la base de datos para usar ese objeto como queramos.

Merge()

Este método funciona igual que el método persist pero, sirve para actualizar una entidad en la base de datos.

```
EntityManager em =  
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").create  
EntityManager();  
  
//Usamos el método find para buscar el alumno a editar  
Alumno alumno = em.find(Alumno.class,1234);  
  
//Le asignamos un nuevo nombre  
alumno.setNombre("Francisco");  
em.getTransaction().begin();  
  
//Actualizamos el alumno  
em.merge(alumno);  
em.getTransaction().commit();
```

Remove()

Este método se encarga de eliminar una entidad de la base de datos.

```
EntityManager em =  
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").create  
EntityManager();  
  
//Usamos el método find para buscar el alumno a borrar  
Alumno alumno = em.find(Alumno.class,1234);  
em.getTransaction().begin();  
  
//Borramos el alumno  
em.remove(alumno);  
em.getTransaction().commit();
```

JAVA PERSISTENCE QUERY LANGUAGE (JPQL)

Es un lenguaje de consulta orientado a objetos independiente de la plataforma definido como parte de la especificación Java Persistence API (JPA). JPQL es usado para hacer consultas contra las entidades almacenadas en una base de datos relacional. Está inspirado en gran medida por SQL, y sus consultas se asemejan a las consultas SQL en la sintaxis, pero opera con objetos entidad de JPA en lugar de hacerlo directamente con las tablas de la base de datos.

CLAUSULAS SELECT – FROM

La cláusula FROM define de qué entidades se seleccionan los datos. Cualquier implementación de JPA, mapea las entidades a las tablas de base de datos correspondientes. Esto significa que vamos a utilizar el nombre de las entidades en vez del nombre de las tablas y los atributos de las entidades en vez de las columnas de las tablas.

La sintaxis de una cláusula FROM de JPQL es similar a SQL pero usa el modelo de entidad en lugar de los nombres de tabla o columna. El siguiente fragmento de código muestra una consulta JPQL simple en la que selecciono todas las entidades Autor.

```
SELECT a FROM Autor a;
```

En la query se ve que, se hace referencia a la entidad Autor en lugar de la tabla de autor y se le asigna la variable de identificación a. La variable de identificación a menudo se llama alias y es similar a una variable en su código Java.

Se utiliza en todas las demás partes de la consulta para hacer referencia a esta entidad. Por ejemplo, si queremos seleccionar un atributo de la entidad Autor, en vez de todos, usaríamos el alias así:

```
SELECT a.nombre, a.apellido FROM Autor a;
```

CLAUSULA WHERE

La sintaxis es muy similar a SQL, pero JPQL admite solo un pequeño subconjunto de las características de SQL.

JPQL admite un conjunto de operadores básicos para definir expresiones de comparación. La mayoría de ellos son idénticos a los operadores de comparación admitidos por SQL, y puede combinarlos con los operadores lógicos AND, OR y NOT en expresiones más complejas.

Operadores:

- Igual: `author.id = 10`
- Distinto: `author.id <> 10`
- Mayor que: `author.id > 10`

- Mayor o Igual que: `author.id => 10`
- Menor que: `author.id < 10`
- Menor o igual que: `author.id <= 10`
- Between: `author.id BETWEEN 5 and 10`
- Like: `author.firstName LIKE :'%and%'`
- Is null: `author.firstName IS NULL`
Se lo puede negar con el operador NOT, para traer todos los que no son nulos
- In: `author.firstName IN ('John', 'Jane')`
Va a traer todos los autores con el nombre John o Jane.

UNIR ENTIDADES

Si necesitamos seleccionar datos de más de una entidad, por ejemplo, todos los libros que ha escrito un autor, debe unir las entidades en la cláusula FROM. La forma más sencilla de hacerlo es utilizar las asociaciones definidas de una entidad como en el siguiente fragmento de código.

```
SELECT a FROM Libro a JOIN a.autor b;
```

La definición de la entidad Libro proporciona toda la información necesaria para unirla a la entidad Autor, y no es necesario que proporcione una declaración ON adicional.

También podemos utilizar el operador “.”, para navegar a través del atributo de autor de la entidad Libro y traer los libros que tengan un autor con un nombre a elección. Esto generaría una relación implícita entre las dos entidades, sin la necesidad de usar un Join.

```
SELECT a FROM Libro a WHERE a.autor.nombre LIKE : "Homero";
```

CREAR CONSULTAS / QUERY'S CON ENTITYMANAGER

El EntityManager nos permite crear **query's dinámicas** con el lenguaje JPQL. Esto se logra mediante el método `createQuery(query)`. El método `createQuery`, recibe una query SQL, la envía a la base de datos, que está anclada con la Unidad de Persistencia y devuelve el resultado de la consulta.

Para obtener estos resultados vamos a usar dos métodos de la clase Query. Un método es `getResultSet()`, que nos deja atrapar el resultado de la query y guardarlo en una lista y el otro método es `getSingleResult()`, que sirve para cuando queremos traer un solo resultado de la consulta.

Igualmente, recomendamos **usar siempre `getResultSet()`**, ya que, si la consulta llega a traer más de un resultado nos va a generar una excepción, entonces, para evitar esto usamos el `getResultSet` por las dudas.

```

// Para esto vamos a tener que crear un EntityManager
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").create
EntityManager();
// Usamos el metodo createQuery y le ponemos la query de JPQL
List<Autor> autores = em.createQuery("SELECT a FROM Autor a")
.getResultList();
// Ponemos una lista del tipo de dato que vamos a traer en la query y
// usamos el getResultList() para atrapar todos los resultados de la query.

```

De esta manera ya tenemos todos los autores de la base de datos guardados en una lista, solo nos quedaría imprimir la lista de autores y mostrar todos los autores.

AGREGAR PARÁMETROS QUERY

Supongamos que tenemos la siguiente query:

`"SELECT p FROM Persona p WHERE p.edad LIKE :20"`

De esta manera la query no es dinámica, ya que siempre va a buscar personas con la edad de 20, si queremos hacer que los parámetros de las querys sean dinámicas, vamos a hacerlo mediante el método de la clase *Query*, `setParameter()`.

El método recibe un String y una variable para ingresar a la Query. El String tiene que tener el mismo nombre que el parámetro que vamos a poner en la query y el método `setParameter()`, va a asignarle esa variable al parámetro de la query. Pongamos un ejemplo

```

EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").create
EntityManager();
// Creamos una variable de tipo int
int edad = 20;
List<Autor> autores = em.createQuery("SELECT p FROM Persona p WHERE p.edad
LIKE :edad").getResultList().setParameter("edad", edad) ;

```



La query, utiliza un parámetro llamado `edad`, y en el `setParameter()`, le decimos que el parámetro `"edad"`, que es el que está en la query dentro del método `setParameter()`, va a ser igual al valor que está en la variable entera `edad`. Básicamente pone el valor 20, en el parámetro `edad` de la query.

Para hacerla dinámica, solo deberíamos pedirle al usuario un valor distinto para `edad`, cada vez que se haga la query.

EJERCICIOS DE APRENDIZAJE

Para la realización de los ejercicios que se describen a continuación, sigue siendo necesario el conector de MySQL y es necesario tener descargado el [Instructivo Unidad de Persistencia](#).



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Sistema de Guardado de una Librería

El objetivo de este ejercicio es el desarrollo de un sistema de guardado de libros en JAVA utilizando una base de datos MySQL y JPA como framework de persistencia.

Creación de la Base de Datos MySQL:

Lo primero que se debe hacer es crear la base de datos sobre el que operará el sistema de reservas de libros. Para ello, se debe abrir el IDE de base de datos que se está utilizando (Workbench) y ejecutar la siguiente sentencia:

```
CREATE DATABASE libreria;
```

De esta manera se habrá creado una base de datos vacía llamada librería.

Paquetes del Proyecto Java:

Los paquetes que se utilizarán para este proyecto son los siguientes:

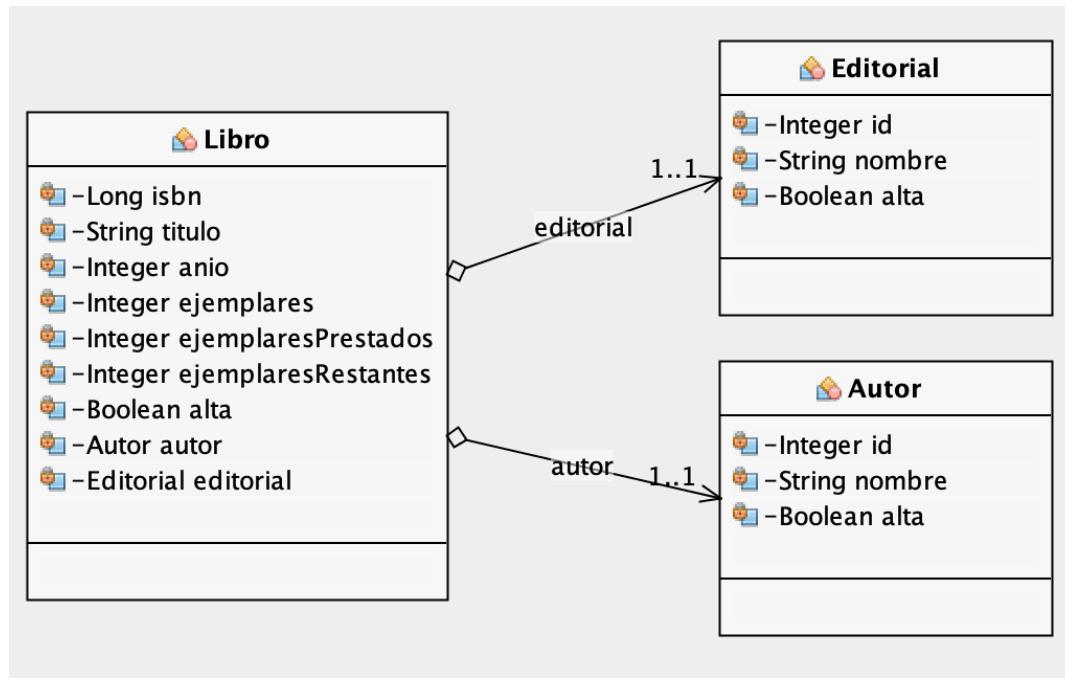
- **entidades:** en este paquete se almacenarán aquellas clases que se quiere persistir en la base de datos.
- **servicios:** en este paquete se almacenarán aquellas clases que llevarán adelante la lógica del negocio. En general se crea un servicio para administrar las operaciones **CRUD** (Create, Remove, Update, Delete) cada una de las entidades y las consultas de cada entidad.

Nota: En este proyecto vamos a eliminar entidades, pero no es considerado una buena práctica. Por esto, además de eliminar nuestras entidades, vamos a practicar que nuestras entidades estén dados de alta o de baja. Por lo que las entidades tendrán un atributo alta booleano, que estará en true al momento de crearlas y en false cuando las demos de baja, que sería cuando se quiere eliminar esa entidad.



a) Entidades

Crearemos el siguiente modelo de entidades:



Entidad Libro

La entidad libro modela los libros que están disponibles en la biblioteca para ser prestados. En esta entidad, el atributo “ejemplares” contiene la cantidad total de ejemplares de ese libro, mientras que el atributo “ejemplaresPrestados” contiene cuántos de esos ejemplares se encuentran prestados en este momento y el atributo “ejemplaresRestantes” contiene cuántos de esos ejemplares quedan para prestar.

Entidad Autor

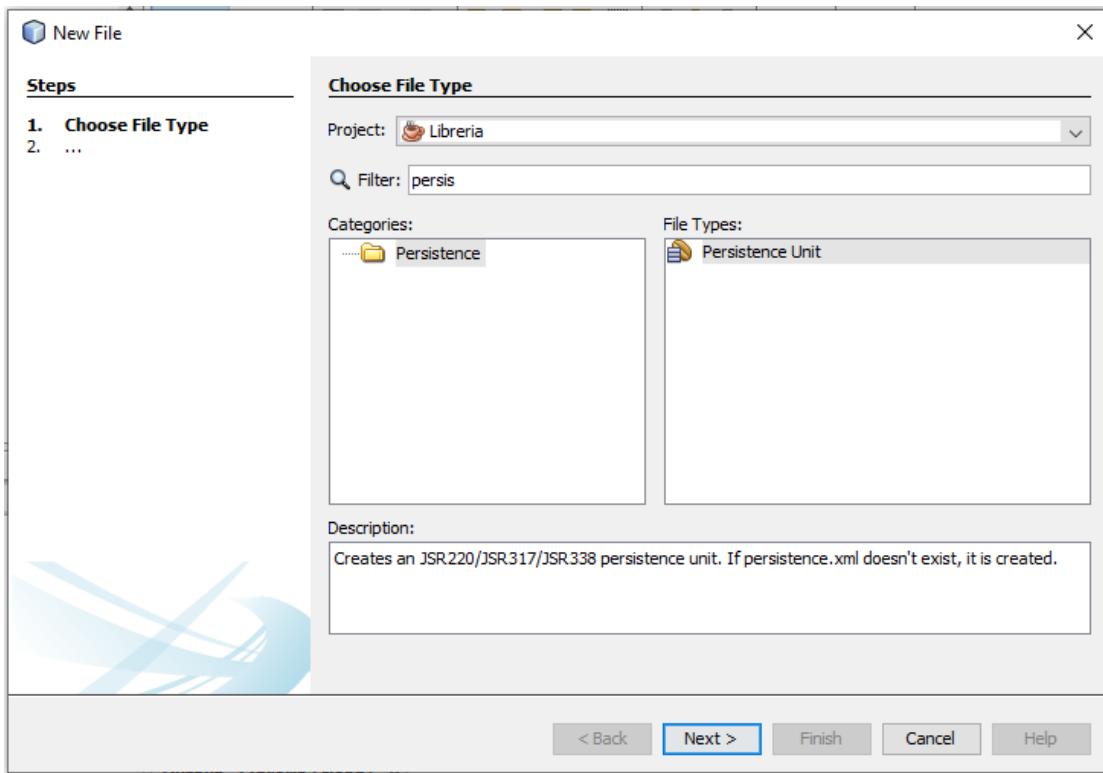
La entidad autor modela los autores de libros.

Entidad Editorial

La entidad editorial modela las editoriales que publican libros.

b) Unidad de Persistencia

Para configurar la unidad de persistencia del proyecto, se recomienda leer el **Instructivo Unidad de Persistencia** recuerde hacer click con el botón derecho sobre el proyecto y seleccionar nuevo. A continuación, se debe seleccionar la opción de Persistence Unit como se indica en la siguiente imagen.



Base de Datos

Para este proyecto nos vamos a conectar a la base de datos Librería, que creamos previamente.

Generación de Tablas

La estrategia de generación de tablas define lo que hará JPA en cada ejecución, si debe crear las tablas faltantes, si debe eliminar todas las tablas y volver a crearlas o no hacer nada. Recomendamos en este proyecto utilizar la opción: "Create"

Librería de Persistencia

Se debe seleccionar para este proyecto la librería "EclipseLink".

c) Servicios

AutorServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar autores (consulta, creación, modificación y eliminación).

EditorialServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar editoriales (consulta, creación, modificación y eliminación)

LibroServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar libros (consulta, creación, modificación y eliminación).

d) Main

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para interactuar con el usuario. En esta clase se muestra el menú de opciones con las operaciones disponibles que podrá realizar el usuario.

e) Tareas a Realizar

Al alumno le toca desarrollar, las siguientes funcionalidades:

- 1)** Crear base de datos Librería
- 2)** Crear unidad de persistencia
- 3)** Crear entidades previamente mencionadas (excepto Préstamo)
- 4)** Generar las tablas con JPA
- 5)** Crear servicios previamente mencionados.
- 6)** Crear los métodos para persistir entidades en la base de datos librería
- 7)** Crear los métodos para dar de alta/bajo o editar dichas entidades.
- 8)** Búsqueda de un Autor por nombre.
- 9)** Búsqueda de un libro por ISBN.
- 10)** Búsqueda de un libro por Título.
- 11)** Búsqueda de un libro/s por nombre de Autor.
- 12)** Búsqueda de un libro/s por nombre de Editorial.
- 13)** Agregar las siguientes validaciones a todas las funcionalidades de la aplicación:
 - Validar campos obligatorios.
 - No ingresar datos duplicados.

EJERCICIOS DE APRENDIZAJE EXTRAS

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

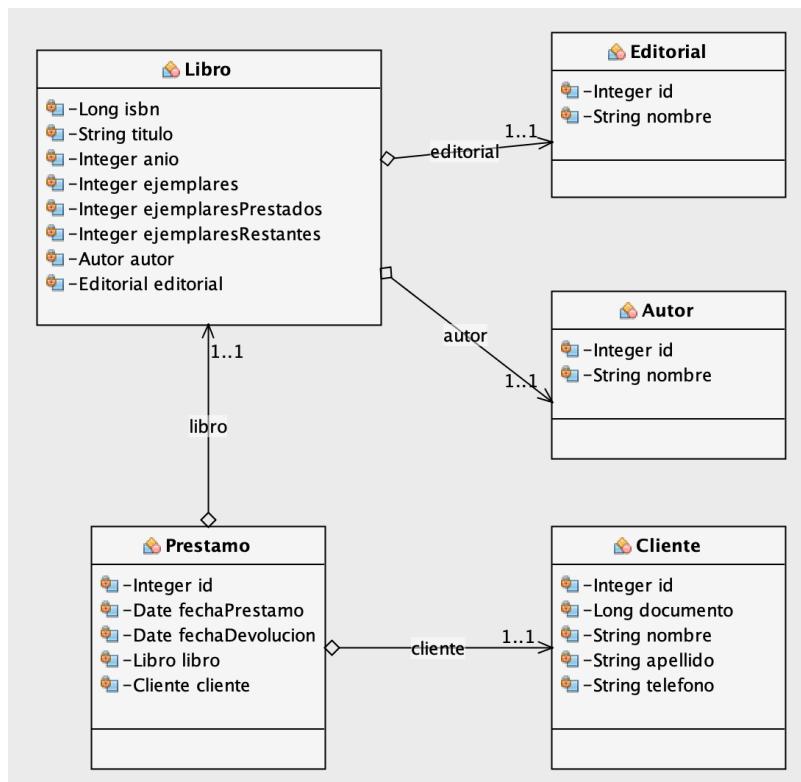
1. Sistema de Reservas de una Librería

Vamos a continuar con el ejercicio anterior. Ahora el objetivo de este ejercicio es el desarrollo de un sistema de reserva de libros en JAVA. Para esto vamos a tener que sumar nuevas entidades a nuestro proyecto en el paquete de entidades y crearemos los servicios de esas entidades.

Usaremos la misma base de datos y se van a crear las tablas que nos faltan. Deberemos agregar las entidades a la unidad de persistencia.

a) Entidades

Crearemos el siguiente modelo de entidades:



Entidad Cliente

La entidad cliente modela los clientes (a quienes se les presta libros) de la biblioteca. Se almacenan los datos personales y de contacto de ese cliente.

Entidad Préstamo

La entidad préstamo modela los datos de un préstamo de un libro. Esta entidad registra la fecha en la que se efectuó el préstamo y la fecha en la que se devolvió el libro. Esta entidad también registra el libro que se llevó en dicho préstamo y quien fue el cliente al cual se le prestaron.

b) Servicios

ClienteServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar clientes (consulta, creación, modificación y eliminación).

PrestamoServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para generar préstamos, va a guardar la información del cliente y del libro para persistirla en la base de datos. (consulta, creación, modificación y eliminación).

c) Tareas a Realizar

Al alumno le toca desarrollar, las siguientes funcionalidades:

1. Creación de un Cliente nuevo
2. Crear entidad Préstamo
3. Registrar el préstamo de un libro.
4. Devolución de un libro
5. Búsqueda de todos los préstamos de un Cliente.
6. Agregar validaciones a todas las funcionalidades de la aplicación:
 - Validar campos obligatorios.
 - No ingresar datos duplicados.
 - No generar condiciones inválidas. Por ejemplo, no se debe permitir prestar más ejemplares de los que hay, ni devolver más de los que se encuentran prestados. No se podrán prestar libros con fecha anterior a la fecha actual, etc.

CURSO DE PROGRAMACIÓN FULL STACK

HTML & CSS



GUIA HTML & CSS

INTRODUCCIÓN

El *World Wide Web* (WWW) es un sistema que contiene una cantidad de información casi infinita. Pero esta información debe estar ordenada de alguna forma de manera que sea posible encontrar lo que se busca. La unidad básica donde está almacenada esta información son las páginas Web. Estas páginas se caracterizan por contener texto, imágenes, animaciones... e incluso sonido y video.

Una de las características más importantes de las páginas Web es que son hipertexto. Esto quiere decir que las páginas no son elementos aislados, sino que están unidas a otras mediante los links o enlaces hipertexto. Gracias a estos enlaces el navegante de internet puede pulsar sobre un texto de una página para navegar hasta otra página. Será cuestión del programador de la página inicial decidir qué palabras o frases serán activas y a dónde nos conducirá pulsar sobre ellas.

HTML

Entendiendo que las páginas web son hipertexto, aquí es donde entra HTML. El **Lenguaje de Marcado de Hipertexto** o *Hyper Text Markup Language* (HTML) es el código que se utiliza para estructurar y desplegar una página web y sus contenidos. HTML es el lenguaje con el que se escribe el contenido de las páginas web. Las páginas web pueden ser vistas por el usuario mediante un tipo de aplicación llamada cliente web o más comúnmente "navegador". Podemos decir por lo tanto que el HTML es el lenguaje usado para especificar el contenido que los navegadores deben representar a la hora de mostrar una página web.

Este lenguaje nos permite aglutinar textos, imágenes, enlaces... y combinarlos a nuestro gusto. La ventaja del HTML a la hora de representar el contenido en un navegador, con respecto a otros formatos físicos como libros o revistas, es justamente la posibilidad de colocar referencias a otras páginas, por medio de los enlaces hipertexto.

Cuando nos referimos al contenido queremos indicar párrafos, imágenes, listas, tablas y todo aquello que forma parte de "el qué". Nunca debemos usar HTML para definir cómo se debe de ver un contenido, si el texto debe tener color rojo, con una fuente mayor, o si se debe alinear a la derecha. Para especificar el aspecto que debe tener una web se usa un lenguaje complementario, llamado CSS.

HTML LENGUAJE DE MARCADO

HTML no es un lenguaje de programación; es un lenguaje de marcado que define la estructura de tu contenido. Basa su sintaxis en un elemento base al que llamamos marca, tag o simplemente etiqueta. A través de las etiquetas vamos definiendo los elementos del documento, como enlaces, párrafos, imágenes, etc. Así pues, un documento HTML estará constituido por texto y un conjunto de etiquetas para definir la función que juega cada contenido dentro de la página. Todo eso le servirá al navegador para saber cómo se tendrá que presentar el texto y otros elementos en la página.

Existen etiquetas para crear negritas, párrafos, imágenes, tablas, listas, enlaces, etc. Así pues, aprender HTML es básicamente aprenderse una serie de etiquetas, sus funciones, sus usos y saber un poco sobre cómo debe de construirse un documento básico.

Es una tarea muy sencilla de afrontar, al alcance de cualquier persona, puesto que el lenguaje es muy entendible para los seres humanos.

Por ejemplo, toma la siguiente línea de texto:

Mi gato es muy gruñon

Si quieras especificar que se trata de un párrafo, podrías encerrar el texto con la etiqueta de párrafo (<p>):

<p> *Mi gato es muy gruñon* </p>

ANATOMIA DE UNA ETIQUETA HTML



Las partes principales de la etiqueta completa llamada elemento son:

1. **La etiqueta de apertura:** consiste en el nombre de la etiqueta (en este caso, p), encerrado por paréntesis angulares (<>) de apertura y cierre. Establece dónde comienza o empieza a tener efecto la etiqueta ,en este caso, dónde es el comienzo del párrafo.
2. **La etiqueta de cierre:** es igual que la etiqueta de apertura, excepto que incluye una barra de cierre (/) antes del nombre de la etiqueta. Establece dónde termina la etiqueta, en este caso dónde termina el párrafo.
3. **El contenido:** este es el contenido de la etiqueta, que en este caso es sólo texto.
4. **El elemento:** la etiqueta de apertura, más la etiqueta de cierre, más el contenido equivale al elemento.

ANIDAR ETIQUETAS

Puedes también colocar etiquetas dentro de otros etiquetas, esto se llama **anidamiento**. Si, por ejemplo, quieras resaltar una palabra del texto (en el ejemplo la palabra «muy»), podemos encerrarla en una etiqueta , que significa que dicha palabra se debe enfatizar:

<p> *Mi gato es muy gruñon* </p>

Debes asegurarte que las etiquetas estén correctamente anidadas: en el ejemplo, creaste la etiqueta de apertura del elemento <p> primero, luego la del elemento , por lo tanto, debes cerrar esta etiqueta primero, y luego la de <p>.

Las etiquetas deben abrirse y cerrarse ordenadamente, de forma tal que se encuentren claramente dentro o fuera el uno del otro. Si estos se encuentran solapados, el navegador web tratará de adivinar lo que intentas decirle, pero puede que obtengas resultados inesperados.

ANATOMIA DE UN DOCUMENTO HTML

Hasta ahora has visto lo básico de elementos HTML individuales, pero estos no son muy útiles por sí solos. Ahora verás cómo los elementos individuales son combinados para formar una página HTML entera.

Los documentos html van a ser archivos de texto con la extensión .html y tienen la siguiente anatomía:

```
<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Mi pagina de prueba</title>

</head>

<body>

<p> Cooperacion Humana </p>

</body>

</html>
```

Tienes:

- **<!DOCTYPE html>**: el tipo de documento. Es un preámbulo requerido. Anteriormente, cuando HTML era joven (cerca de 1991/2), los tipos de documento actuaban como vínculos a un conjunto de reglas que el código HTML de la página debía seguir para ser considerado bueno, lo que podía significar la verificación automática de errores y algunas otras cosas de utilidad. Sin embargo, hoy día es simplemente un artefacto antiguo que a nadie le importa, pero que debe ser incluido para que todo funcione correctamente. Por ahora, eso es todo lo que necesitas saber.
- **<html></html>**: la etiqueta <html>. Esta etiqueta encierra todo el contenido de la página entera y, a veces, se le conoce como la etiqueta raíz (*root element*).
- **<head></head>**: la etiqueta <head>. Esta etiqueta actúa como un contenedor de todo aquello que quieras incluir en la página HTML que no es contenido visible por los visitantes de la página. Incluye cosas como palabras clave (*keywords*), una descripción de la página que quieras que aparezca en resultados de búsquedas, código CSS para dar estilo al contenido, declaraciones del juego de caracteres, etc.
- **<meta charset="utf-8">**: esta etiqueta establece el juego de caracteres que tu documento usará en utf-8, que incluye casi todos los caracteres de todos los idiomas humanos. Básicamente, puede manejar cualquier contenido de texto que puedas incluir. No hay razón para no establecerlo, y puede evitar problemas en el futuro.

- **<title></title>**: la etiqueta `<title>` establece el título de tu página, que es el título que aparece en la pestaña o en la barra de título del navegador cuando la página es cargada, y se usa para describir la página cuando es añadida a los marcadores o como favorita.
- **<body></body>**: la etiqueta `<body>`. Encierra todo el contenido que deseas mostrar a los usuarios web que visiten tu página, ya sea texto, imágenes, videos, juegos, pistas de audio reproducibles, y demás. Estos, delimitados a su vez por otras etiquetas como las que hemos visto.

ELEMENTOS EN BLOQUE Y EN LÍNEA

El lenguaje HTML clasifica a todos los elementos en dos grupos: **elementos en línea** o *inline elements* y **elementos en bloque** o *block elements*. La diferencia entre ambos viene dada por el modelo de contenido, por el formato y la dirección.

Los elementos en bloque siempre empiezan en una nueva línea y ocupan todo el espacio disponible hasta el final de la línea, mientras que los elementos en línea sólo ocupan el espacio necesario para mostrar sus contenidos.

ATRIBUTOS ETIQUETAS

Las etiquetas son la estructura básica del HTML. Estas etiquetas se componen y contienen otras propiedades, como son los atributos y el contenido.

HTML define un total de 91 etiquetas, de las cuales 10 se consideran obsoletas. Sin embargo, una etiqueta por sí sola a veces no contiene la suficiente información para estar completamente definida. Para ello contamos con los **atributos**: pares nombre-valor separados por `=` y escritos en la etiqueta inicial de un elemento después del nombre del elemento. El valor puede estar encerrado entre "comillas dobles" o 'simples'. Existen, también, algunos atributos que afectan al elemento por su presencia en la etiqueta de inicio.

Esta sería la estructura general de una línea de código en lenguaje HTML:

```
<etiqueta atributo1="valor1" atributo2="valor2">contenido</etiqueta>
<a href="http://www.enlace.com" target="_blank">Ejemplo de enlace</a>
```

Donde:

- `<a>` es la etiqueta inicial y `` la etiqueta de cierre.
- `href` y `target` son los atributos.
- `http://www.enlace.com` y `_blank` son las variables.
- `Ejemplo de enlace` es el contenido.

Nota: las etiquetas `` y `<a>` las veremos en mayor profundidad más adelante.

TIPOS DE ATRIBUTOS

Aunque cada una de las etiquetas HTML define sus propios atributos, encontramos algunos comunes a muchas o casi todas las etiquetas, que se dividen en cuatro grupos según su funcionalidad:

- Atributos básicos
- Atributos de internacionalización
- Atributos de eventos
- Atributos de foco

En esta guía solo vamos a ver los básicos. Ya que el resto de atributos son para el uso de otro lenguaje.

ATRIBUTOS BASICOS

Los atributos básicos se utilizan en la mayoría de etiquetas HTML y XHTML, aunque adquieren mayor sentido cuando se utilizan hojas de estilo en cascada (CSS):

Atributo	Descripción
id="texto"	Establece un indicador único a cada elemento
class="texto"	Establece la clase CSS que se aplica a los estilos del elemento
style="texto"	Aplica de forma directa los estilos CSS de un elemento
title="texto"	Establece el título del elemento (Mejora la accesibilidad)

Nota: los atributos de id, class y style los veremos en mayor profundidad en la parte de CSS.

SINTAXIS HTML

LAS MAYÚSCULAS O MINÚSCULAS SON INDIFERENTES AL ESCRIBIR ETIQUETAS

En HTML las mayúsculas y minúsculas son indiferentes. Quiere decir que las etiquetas pueden ser escritas con cualquier tipo de combinación de mayúsculas y minúsculas. Resulta sin embargo aconsejable acostumbrarse a escribirlas en minúscula ya que otras tecnologías que pueden convivir con nuestro HTML (XML por ejemplo) no son tan permisivas y nunca viene mal hacernos a las buenas costumbres desde el principio, para evitar fallos triviales en un futuro.

COMENTARIOS EN HTML

En un documento HTML, los comentarios se escriben entre los caracteres "<!--" y "-->". Por ejemplo: <!--Esto es un comentario en HTML-->

SALTOS DE LÍNEA EN HTML

Otra de las cosas importantes de conocer sobre la sintaxis básica del HTML es que los saltos de línea no importan a la hora de interpretar una página. Un salto de línea será simplemente interpretado como un separador de palabras, un espacio en blanco. Es por ello que para separar líneas necesitamos usar la etiqueta de párrafo, o la etiqueta BR que significa un salto de línea simple.

Esto es una línea

Esto es otra línea

Esto en una pagina se vería así

Esto es una línea

Esto es otra línea

La etiqueta BR no tiene su correspondiente cierre. Es un detalle que quizás te haya llamado la atención.

FORMATO DE PARRAFOS HTML

Previamente en nuestra guía habíamos visto la etiqueta que nos permitía darle formato a nuestro texto, más concreto ponerlo en negrita. Ahora veremos con más detalle las más ampliamente utilizadas y exemplificaremos algunas de ellas posteriormente.

Formatear un texto pasa por tareas tan evidentes como definir los párrafos, justificarlos, introducir viñetas, numeraciones o bien poner en negrita, itálica, etc.

Hemos visto que para definir los párrafos nos servimos de la etiqueta P que introduce un salto y deja una línea en blanco antes de continuar con el resto del documento.

Podemos también usar la etiqueta
, de la cual no existe su cierre correspondiente, para realizar un simple salto de línea con lo que no dejamos una línea en blanco sino que solo cambiamos de línea. Cabe destacar que la etiqueta
, no es la única etiqueta sin cierre.

Podéis comprobar que cambiar de línea en nuestro documento HTML sin introducir alguna de estas u otras etiquetas no implica en absoluto un cambio de línea en la página visualizada. En realidad el navegador introducirá el texto y no cambiara de línea a no ser que esta llegue a su fin o bien lo especifiquemos con la etiqueta correspondiente.

ALINEAR TEXTO

Los párrafos delimitados por etiquetas P pueden ser fácilmente justificados a la izquierda, centro o derecha especificando dicha justificación en el interior de la etiqueta por medio de un atributo "align". Recordemos que los atributos no son más que un parámetro incluido en el interior de la etiqueta que ayudan a definir el funcionamiento de la etiqueta de una forma más personalizada.

Es importante tener muy en cuenta lo siguiente, que ya hemos comentado anteriormente. El HTML se usa para definir el contenido. Por lo tanto, los atributos align que vamos a conocer a continuación se estarán metiendo en una terreno que no le corresponde al HTML, porque están definiendo la forma en la que un párrafo debe de representarse, su estilo, y no el contenido. Es importante señalarlo para aprender que estas cosas se deben hacer mediante el lenguaje CSS, que sirve para definir el estilo, la forma. Usamos este ejemplo también para reforzar el uso de los atributos de una manera más práctica.

Así, si deseásemos introducir un texto alineado a la izquierda escribiríamos:

```
<p align="left">Texto alineado a la izquierda</p>
```

Para una justificación al centro:

```
<p align="center">Texto alineado al centro</p>
```

Para alinear a la derecha:

```
<p align="right">Texto alineado a la derecha</p>
```

Esto en una página se vería así:

Texto alineado a la izquierda

Texto alineado al centro

Texto alineado a la derecha

Como veis, en cada caso el atributo align toma determinados valores que son escritos entre comillas. En algunas ocasiones necesitamos especificar algunos atributos para el correcto funcionamiento de la etiqueta. En otros casos, el propio navegador toma un valor definido por defecto. Para el caso de align, el valor por defecto es left.

FORMATO DE LETRA

Además de todo lo relativo a la organización de los párrafos, uno de los aspectos primordiales del formato de un texto es el de la propia letra. Resulta muy común y práctico presentar texto resaltado en negrita, itálica y otros. Todo esto y mucho más es posible por medio del HTML a partir de multitud de etiquetas entre las cuales vamos a destacar algunas.

Pero antes de comenzar cabe hacer una reflexión sobre por qué son interesantes estas etiquetas y se siguen usando, a pesar que están entrando prácticamente en el terreno de CSS, ya que en la práctica están directamente formateando el aspecto de las fuentes. Son importantes porque las etiquetas en sí no están para definir un estilo en concreto, sino una función de ciertas palabras dentro de un contenido.

NEGRITA

Podemos escribir texto en negrita incluyéndolo dentro de las etiquetas strong y su cierre. Recordemos que ya la habíamos visto previamente.

```
<p><strong> Texto en negrita </strong> y texto normal</p>
```

Esto en una pagina se vería así:

Texto en negrita y texto normal

ITÁLICA

En este caso existen dos posibilidades, una corta: *i* y su cierre (italic) y otra un poco más larga: **EM** y su cierre. En esta guía vamos a usar, y en la mayoría de las páginas que veréis por ahí, os encontraréis con la primera forma sin duda más sencilla a escribir y a acordarse.

< p > < i > Texto en italicica < /i > y texto normal < /p >

Esto en una pagina se vería así:

Texto en italicica y texto normal

SUBRAYADO

El HTML nos propone también para el subrayado la etiqueta: **U** (underlined). Sin embargo, el uso de subrayados ha de ser aplicado con mucha precaución dado que los enlaces hipertexto van, a no ser que se indique lo contrario, subrayados con lo que podemos confundir al lector y apartarlo del verdadero interés de nuestro texto.

Además, cabe decir que la etiqueta **U** se ha quedado obsoleta, debido a que es algo que realmente se debe hacer del lado del CSS, al ser básicamente un estilo.

< p > < u > Texto subrayado < /u > y texto normal < /p >

Esto en una pagina se vería así:

Texto subrayado y texto normal

ENCABEZADOS

Existen otras etiquetas para definir párrafos especiales, que funcionaran como títulos de nuestra pagina. Son los encabezados o headings en inglés. Como decimos, son etiquetas que formatean el texto como un titulo, pero el hecho de que cambien el formato no es lo que nos tiene que preocupar, sino el significado en sí de la etiqueta. Es cierto que los navegadores asignan un tamaño mayor de letra y colocan el texto en negrita, pero lo importante es que sirven para definir la estructura del contenido de un documento HTML. Así los navegadores para ciegos podrán informar a los invidentes que esta es una división nueva de contenido y que su titulo es este o aquel. También motores de búsqueda sabrán interpretar mejor el contenido de una página en función de los títulos y subtítulos.

Hay varios tipos de encabezados, que se diferencian visualmente en el tamaño de la letra que utilizan. La etiqueta en concreto es la H1, para los encabezados más grandes, H2 para los de segundo nivel y así hasta H6 que es el encabezado más pequeño. Pero lo importante, insistimos es la estructura que denotan. Una página tendrá generalmente un encabezado de nivel 1 y dentro varios de nivel 2.

Luego, dentro de los H2 encontraremos si acaso H3, etc. Nunca debemos usar los encabezados porque nos formateen el texto de una manera dada, sino porque nuestro documento lo requiera según su estructura.

Los encabezados se verán de esta manera en la página:

Encabezado de nivel 1

Encabezado de nivel 2

Encabezado de nivel 3

Encabezado de nivel 4

Encabezado de nivel 5

Encabezado de nivel 6

Los encabezados implican también una separación en párrafos, así que todo lo que escribamos dentro de H1 y su cierre (o cualquier otro encabezado) se colocará en un párrafo independiente.

Podemos ver cómo se presentan algunos encabezados a continuación.

`<h1> Encabezado de nivel 1 </h1>`

Los encabezados, como otras etiquetas de HTML, soportan el atributo align. Veremos un ejemplo de encabezado de nivel 2 alineado al centro, aunque repetimos que esto debería hacerse en CSS.

`<h2 align="center"> Encabezado de nivel 2 </h2>`

LISTAS EN HTML

Las posibilidades que nos ofrece el HTML en cuestión de tratamiento de texto son realmente notables. No se limitan a lo visto hasta ahora, sino que van más lejos todavía. Varios ejemplos de ello son las listas, que sirven para enumerar y definir elementos.

Las listas originalmente están pensadas para citar, numerar y definir cosas a través de características, o al menos así lo hacemos en la escritura de textos. Sin embargo, las listas finalmente se utilizan para mucho más que enumerar una serie de puntos, en realidad son un recurso muy interesante para poder maquetar elementos diversos, como barras de navegación, pestañas etc.

Por ahora, trataremos las listas desde el punto de vista de su construcción y veremos los diferentes tipos que existen, y que podemos utilizar para resolver nuestras distintas necesidades a la hora de escribir textos en HTML.

Podemos distinguir dos tipos de listas HTML:

- Listas desordenadas
- Listas ordenadas

LISTAS DESORDENADAS

Son delimitadas por las etiquetas **UL** y su cierre (*unordered list*). Cada uno de los elementos de la lista es citado por medio de una etiqueta **LI** (La LI tiene su cierre, aunque si no lo colocas el navegador al ver el siguiente LI interpretará que estás cerrando el anterior). La cosa queda así:

```
<p>Países del mundo</p>  
<ul>  
    <li>Argentina</li>  
    <li>Perú</li>  
    <li>Chile</li>  
</ul>
```

Esto en una pagina se vería así:

Países del mundo

- Argentina
- Perú
- Chile

Podemos definir el tipo de viñeta empleada para cada elemento. Para ello debemos especificarlo por medio del **atributo type** incluido dentro de la etiqueta de apertura UL, si queremos que el estilo sea válido para toda la lista, o dentro de la etiqueta LI si queremos hacerlo específico de un solo elemento. La sintaxis es del siguiente tipo:

```
<ul type="tipo de viñeta">
```

Donde tipo de viñeta puede ser uno de los siguientes:

- circle
- disc
- square

Vamos a ver un ejemplo de lista con un cuadrado en lugar de un redondel, y en el último elemento colocaremos un círculo. Para ello vamos a colocar el atributo type en la etiqueta UL, con lo que afectará a todos los elementos de la lista.

```
<ul type="square">  
    <li>Elemento 1 </li>  
    <li>Elemento 2 </li>  
    <li>Elemento 3 </li>  
    <li type="circle">Elemento 4  
</ul>
```

Esto en una pagina se vería así:

- Elemento 1
- Elemento 2
- Elemento 3
- Elemento 4

LISTAS ORDENADAS

Las listas ordenadas sirven también para presentar información, en diversos elementos o items, con la particularidad que éstos estarán precedidos de un número o una letra para enumerarlos, siempre por un orden.

Para realizar las listas ordenadas usaremos las etiquetas OL (ordered list) y su cierre. Cada elemento sera igualmente indicado por la etiqueta LI, que ya vimos en las listas desordenadas.

Pongamos un ejemplo:

```
<p>Reglas de comportamiento en el trabajo</p>
<ol>
<li>El jefe siempre tiene la razón </li>
<li>En caso de duda aplicar regla 1 </li>
</ol>
```

Esto en una pagina se vería así:

Reglas de comportamiento en el trabajo

1. El jefe siempre tiene la razón
2. En caso de duda aplicar regla 1

Del mismo modo que para las listas desordenadas, las listas ordenadas ofrecen la posibilidad de modificar el estilo. En concreto nos es posible especificar el tipo de numeración empleado eligiendo entre números (1, 2, 3...), letras (a, b, c...) y sus mayúsculas (A, B, C,...) y números romanos en sus versiones mayúsculas (I, II, III,...) y minúsculas (i, ii, iii,...).

Para realizar dicha selección hemos de utilizar, como para el caso precedente, el atributo type, el cual será situado dentro de la etiqueta OL. Los valores que puede tomar el atributo en este caso son:

- 1 Para ordenar por números
- a Por letras del alfabeto
- A Por letras mayúsculas del alfabeto
- i Ordenación por números romanos en minúsculas
- I Ordenación por números romanos en mayúsculas

Puede que en algún caso deseemos comenzar nuestra enumeración por un número o letra que no tiene por qué ser necesariamente el primero de todos. Para solventar esta situación, podemos utilizar un segundo atributo, start, que tendrá como valor un número. Este número, que por defecto es 1, corresponde al valor a partir del cual comenzamos a definir nuestra lista. Para el caso de las letras o los números romanos, el navegador se encarga de hacer la traducción del número a la letra correspondiente.

Un ejemplo de todo esto sería:

```
<p>Ordenamos por números</p>  
<ol type="1">  
  <li>Elemento 1 </li>  
  <li> Elemento 2 </li>  
</ol>  
  
<p>Ordenamos por letras</p>  
<ol type="a">  
  <li>Elemento a </li>  
  <li> Elemento b </li>  
</ol>  
  
<p>Ordenamos por números romanos empezando por el 10</p>  
<ol type="i" start="10">  
  <li>Elemento x </li>  
  <li> Elemento xi </li>  
</ol>
```

Esto en una pagina se vería así:

Ordenamos por números

- 1. Elemento 1
- 2. Elemento 2

Ordenamos por letras

- a. Elemento a
- b. Elemento b

Ordenamos por números romanos empezando por el 10

- x. Elemento x
- xi. Elemento xi

ANIDANDO LISTAS

Nada nos impide utilizar todas estas etiquetas de forma anidada como hemos visto en otros casos. De esta forma, podemos conseguir listas mixtas como por ejemplo:

```
<p>Ciudades del mundo</p>
<ul>
    <li>Argentina </li>
    <ol>
        <li>Buenos Aires </li>
        <li>Bariloche </li>
    </ol>
    <li>Uruguay </li>
    <ol>
        <li>Montevideo </li>
        <li>Punta del Este </li>
    </ol>
</ul>
```

Esto en una pagina se vería así:

Ciudades del mundo

- Argentina
 - 1. Buenos Aires
 - 2. Bariloche
- Uruguay
 - 1. Montevideo
 - 2. Punta del Este

ENLACES EN HTML

Hasta aquí, hemos podido ver que una página web es un archivo HTML en el que podemos incluir, entre otras cosas, textos formateados a nuestro gusto e imágenes (las veremos con detalle enseguida). Del mismo modo, un sitio web podrá ser considerado como el conjunto de archivos, principalmente páginas HTML e imágenes, que constituyen el contenido al que el navegante tiene acceso.

Sin embargo, no podríamos hablar de navegante o de navegación si estos archivos HTML no estuviesen debidamente conectados entre ellos y con el exterior de nuestro sitio por medio de enlaces hipertexto. En efecto, el atractivo original del HTML radica en la posible puesta en relación de los contenidos de los archivos introduciendo referencias bajo forma de enlaces que permitan un acceso rápido a la información deseada. De poco serviría en la red tener páginas aisladas a las que la gente no puede acceder y desde las que la gente no puede saltar a otras.

Un enlace puede ser fácilmente detectado por el usuario en una página. Basta con deslizar el puntero del ratón sobre las imágenes o el texto y ver como cambia de su forma original transformándose por regla general en una mano con un dedo señalador.

Adicionalmente, estos enlaces suelen ir, en el caso de los textos, coloreados y subrayados para que el usuario no tenga dificultad en reconocerlos.

SINTAXIS DE UN ENLACE

Para colocar un enlace, nos serviremos de las etiquetas **a** y su cierre. Dentro de la etiqueta de apertura deberemos especificar asimismo el destino del enlace. Este destino será introducido bajo forma de atributo, el cual lleva por nombre "href".

La sintaxis general de un enlace es por tanto de la forma:

```
<a href="destino">contenido</a>
```

Siendo el "contenido" un texto o una imagen. Es la parte de la página que se colocará activa y donde deberemos pulsar para acceder al enlace. Por su parte, "destino" será una página, un correo electrónico o un archivo.

Por ejemplo, un enlace a la home de EggEducación, se vería así:

```
<a href="https://eggeducacion.com/es-AR/">Home de EggEducación.com</a>
```

Esto en una pagina se vería así:

[Home de EggEducación.com](https://eggeducacion.com/es-AR/)

Ahora, si queremos que el contenido del enlace sea una imagen y no un texto, podremos colocar la correspondiente etiqueta IMG dentro de la etiqueta a.

```
<a href="https://eggeducacion.com/es-AR/"></a>
```

Nota: veremos la etiqueta de imágenes más adelante.

EL ASPECTO DE LOS ENLACES

Nosotros mediante el HTML, y las hojas de estilo CSS, podemos definir el aspecto que tendrán los enlaces en una página. Sin embargo, ya de manera predeterminada el navegador los destaca para que los podamos distinguir. Generalmente encontraremos a los enlaces subrayados y coloreados en azul, aunque esta regla depende del navegador del usuario y de sus estilos definidos como predeterminados.

TIPOS DE ENLACES

Para estudiar en profundidad los enlaces tenemos que clasificarlos por su tipo, porque dependiendo ese tipo algunas cosas cambiarán a la hora de construirlos.

En función del destino los enlaces son clásicamente agrupados del siguiente modo:

- Enlaces locales: los que se dirigen a otras páginas del mismo sitio web.
- Enlaces remotos: los dirigidos hacia páginas de otros sitios web. Estos son los que vimos en el ejemplo anterior.

ENLACES LOCALES

Como hemos dicho, un sitio web esta constituido de páginas interconexas, que se relacionan mediante enlaces de hipertexto. Para cumplir con esto es que vamos a utilizar los enlaces locales.

Los enlaces locales se tratan de un tipo de enlace mucho más común en el día a día del desarrollo. De hecho, es el tipo de enlace que más se produce en lo general. Estos enlaces locales nos permiten relacionar distintos documentos HTML que componen un sitio web. Gracias a los enlaces locales podremos convertir varias páginas sueltas en un sitio web completo, compuesto de varios documentos.

Para crear este tipo de enlaces, hemos de usar la misma etiqueta A que ya conocemos, de la siguiente forma:

```
<a href="archivo.html">contenido</a>
```

RUTAS DE LOS ENLACES

Hacer un enlace en si no es para nada complejo. No requiere muchas explicaciones con lo que ya hemos visto en la guía alcanza, sin embargo hay que abordar con detalle un tema importante: las rutas de los enlaces. Como rutas nos referimos al destino del enlace, o sea, lo que ponemos en el atributo "href" y es importante que nos paremos aquí porque nos puede dar algunos problemas al desarrollar, sobre todo para las personas que puedan tener menos experiencia en el trabajo con el ordenador.

Por regla general, para una mejor organización, los sitios suelen estar ordenados por directorios. Estos directorios suelen contener diferentes secciones de la página, imágenes, scripts, estilos, etc. Es por ello que en muchos casos no nos valdrá con especificar el nombre del archivo, sino que tendremos que especificar además el directorio en el que nuestro archivo.html esta alojado.

Para aquellos que no saben como mostrar un camino de un archivo, aquí van una serie de indicaciones que los ayudaran a comprender la forma de expresarlos. No resulta difícil en absoluto y con un poco de practica lo haréis prácticamente sin pensar.

1. Hay que situarse mentalmente en el directorio en el que se encuentra la página donde vamos a crear el enlace.
2. Si la página destino está en el mismo directorio que el archivo desde donde vamos a enlazar podemos colocar simplemente el nombre del archivo de destino, ya que no hay necesidad de cambiar de directorio.
3. Si la página de destino está en una carpeta o subdirectorío interior al directorio donde está el archivo de origen, hemos de marcar la ruta enumerando cada uno de los directorios por los que pasamos hasta llegar al archivo de destino, separándolos por el símbolo barra "/". Al final obviamente, escribimos el nombre del archivo destino.
4. Si la página destino se encuentra en un directorio padre (superior al de la página del enlace), hemos de escribir dos puntos y una barra "../" tantas veces como niveles subamos en la arborescencia hasta dar con el directorio donde esta emplazado el archivo destino.

5. Si la página se encuentra en otro directorio no incluido ni incluyente del archivo origen, tendremos que subir como en la regla 3 por medio de ".." hasta encontrar un directorio que englobe el directorio que contiene a la página destino. A continuación haremos como en la regla 2. Escribiremos todos los directorios por los que pasamos hasta llegar al archivo.

Imagina que tienes la siguiente estructura de carpetas y archivos. La que aparece en la siguiente imagen.



- 1) Para hacer un enlace a index.html

```
<a href="index.html">Ir a index.html</a>
```

- 2) Para hacer un enlace desde index.html hacia pagina1.html:

```
<a href="seccion1/paginas/pagina1.html">Ir a pagina1.html</a>
```

- 3) Para hacer un enlace desde pagina2 hacia pagina1:

```
<a href="../seccion1/paginas/pagina1.html">Ir (también) a pagina1.html</a>
```

- 4) Para hacer un enlace desde pagina1 hacia pagina2:

```
<a href="../../seccion2/pagina2.html">Ir ahora a pagina2.html</a>
```

IMÁGENES EN HTML

Sin duda uno de los aspectos más vistosos y atractivos de las páginas web es el grafismo. La introducción en nuestro texto de imágenes puede ayudarnos a explicar más fácilmente nuestra información y darle un aire mucho más estético. El abuso no obstante, puede conducirnos a una sobrecarga que se traduce en una distracción para el navegante, quien tendrá más dificultad en encontrar la información necesaria.

El uso de imágenes también tiene que ser realizado con cuidado porque aumentan el tiempo de carga de la página, lo que puede ser de un efecto nefasto si nuestro visitante no tiene una buena conexión o si es un poco impaciente. Por ello es recomendable siempre optimizar las imágenes para Internet, haciendo que su tamaño en bytes sea lo mínimo posible, para facilitar la descarga, pero sin que ello comprometa mucho su calidad.

En esta guía no explicaremos como crear ni tratar las imágenes, únicamente diremos que para ello se utilizan aplicaciones como Paint Shop Pro, Photoshop o Gimp. Tampoco explicaremos las particularidades de cada tipo de archivo: GIF, JPG o PNG y la forma de optimizar nuestras imágenes.

La etiqueta que utilizaremos para insertar una imagen es **IMG** (*image*). Esta etiqueta no posee su cierre correspondiente y en ella hemos de especificar obligatoriamente el paradero de nuestro archivo gráfico mediante el atributo **src** (*source*).

La sintaxis queda entonces de la siguiente forma:

```

```

Para expresar el camino, lo haremos de la misma forma que vimos para los enlaces. Las reglas siguen siendo las mismas, lo único que cambia es que, en lugar de una página siendo el destino, el destino es un archivo gráfico. En el código anterior estamos enlazando un archivo con extensión .jpg, pero podrá ser otro tipo de archivo como .gif o .png.

A parte de este atributo, indispensable obviamente para la visualización de la imagen, la etiqueta IMG nos propone otra serie de atributos de mayor o menor utilidad, que listamos a continuación:

ATRIBUTO ALT

Dentro de las comillas de este atributo colocaremos una brevíssima descripción de la imagen. Esta etiqueta no es indispensable pero presenta varias utilidades. La sintaxis te quedaría de esta manera:

```

```

Primeramente, sirve para el posicionamiento de la página en buscadores. De los atributos alt el buscador puede extraer palabras clave y le ayuda a entender qué función o contenido tiene la imagen, y por lo tanto la página.

Otra utilidad importante la encontramos en determinadas aplicaciones, usadas por personas con discapacidad. Navegadores para ciegos, por ejemplo, no muestran las imágenes y por tanto los alt ofrecen la posibilidad de leerlas. Nunca esta de más pensar en crear páginas accesibles.

Por último, durante el proceso de carga de la página y cuando la imagen no ha sido todavía cargada, el navegador podría mostrar esta descripción, con lo que el navegante se puede hacer una idea de lo que va en ese lugar. Si hubo problemas de conexión y no se pudo mostrar la imagen, también podría usarse ese alt para mostrar al menos su descripción.

En general podemos considerar como aconsejable el uso de este atributo, salvo para imágenes de poca importancia. Si la imagen es usada como cuerpo de un enlace todavía se hace más indispensable.

ATRIBUTOS HEIGHT Y WIDTH

Estos atributos definen la altura y anchura respectivamente de la imagen en píxeles. Aunque estas dimensiones forman parte del estilo de la imagen, y por tanto podrían ir en el CSS, todavía puede ser interesante definirlas dentro del HTML. Esto, ya no es tan indispensable, puesto que muchos sitios creados con "Responsive Web Design" prefieren que las imágenes se adapten al tamaño de la pantalla donde se va a visualizar.

Todos los archivos gráficos poseen unas dimensiones de ancho y alto. Estas dimensiones pueden obtenerse a partir del propio diseñador gráfico o bien haciendo clic con el botón derecho sobre la imagen, vista desde el explorador de archivos de tu ordenador, para luego elegir "propiedades" o "información de la imagen" sobre el menú que se despliega.

Un ejemplo de etiqueta IMG con sus valores de anchura y altura declarados te quedaría así:

```

```

IMÁGENES QUE SON ENLACES Y EL ATRIBUTO BORDER

Si un texto puede servir de enlace, una imagen puede cumplir la misma función:

```
<a href="archivo.html"></a>
```

El problema de hacer esto en ciertos navegadores es que se crea un borde en la imagen, del mismo color que el color configurado para los enlaces, lo que suele ser un efecto poco deseado.

Sin embargo, en HTML podemos indicar que una imagen tenga borde. Mediante el atributo "border" se define el tamaño en píxeles del cuadro que rodea la imagen. De esta forma podemos recuadrar nuestra imagen si lo deseamos. No es algo que se use mucho, pero resulta particularmente útil cuando deseamos eliminar el borde que aparece cuando la imagen sirve de enlace. En dicho caso tendremos que especificar border="0".

TABLAS EN HTML

Una tabla es un conjunto de celdas organizadas dentro de las cuales podemos alojar distintos contenidos. HTML dispone de una gran variedad de etiquetas para crear tablas, con sus atributos.

En un principio nos podría parecer que las tablas son raramente útiles y que pueden ser utilizadas principalmente para listar datos como agendas, resultados y otros datos de una forma organizada. En general, sirven para representar información tabulada, en filas y columnas. Esto es una realidad en los últimos años, desde que las tablas se han descartado para fines relacionados con la maquetación.

Como veremos a continuación, existen diversas etiquetas que se deben utilizar en una forma determinada para la creación de tablas. Por ello, puede que en un principio nos resulte un poco complicado trabajar con estas estructuras pero, con un poco de práctica podremos crear tablas con absoluta soltura. Si deseamos mostrar datos de una manera sencilla de leer, dispuestos en filas y columnas, tarde o temprano observaremos que las tablas son la mejor solución y apreciaremos las posibilidades nos ofrecen.

ETIQUETAS BÁSICAS PARA TABLAS EN HTML

Para empezar, nada más sencillo que por el principio: las tablas son definidas por las etiquetas **TABLE** y su cierre.

Dentro de estas dos etiquetas colocaremos todas las otras etiquetas de las tablas, hasta llegar a las celdas. Dentro de las celdas ya es permitido colocar textos e imágenes que darán el contenido a la tabla.

Las tablas son descritas por líneas de arriba a abajo (y luego por columnas de izquierda a derecha). Cada una de estas líneas, llamada fila, es definida por otra etiqueta y su cierre: **TR** (*table row*).

Asimismo, dentro de cada línea, habrá diferentes celdas. Cada una de estas celdas será definida por otra etiqueta: **TD** (*table data*). Dentro de ésta y su cierre será donde coloquemos nuestro contenido, el contenido de cada celda.

Aquí tenéis un ejemplo de estructura de tabla:

```
<table>
<tr>
  <td>Celda 1, linea 1</td>
  <td> Celda 2, linea 1</td>
</tr>
<tr>
  <td> Celda 1, linea 2</td>
  <td> Celda 2, linea 2</td>
</tr>
</table>
```

Esto en una pagina se vería así:

Celda 1, linea 1 Celda 2, linea 1
Celda 1, linea 2 Celda 2, linea 2

También existe la etiqueta **TH** (*table header*), que sirve para crear una celda cuyo contenido esté formateado como un título o cabecera de la tabla. En la práctica, lo que hace es poner en negrita y centrado el contenido de esa celda, lo que se puede conseguir aplicando las correspondientes etiquetas dentro de la celda.

Aquí tenéis un ejemplo de estructura de tabla con la etiqueta th:

```
<table>
<tr>
  <th>Titulo Celda 1</th>
  <th> Titulo Celda 2</th>
</tr>
<tr>
  <td>Celda 1, linea 1</td>
  <td> Celda 2, linea 1</td>
</tr>
<tr>
  <td> Celda 1, linea 2</td>
  <td> Celda 2, linea 2</td>
</tr>
</table>
```

Esto en una pagina se vería así:

Titulo Celda 1 Titulo Celda 2

Celda 1, linea 1 Celda 2, linea 1

Celda 1, linea 2 Celda 2, linea 2

ATRIBUTOS PARA TABLAS, FILAS Y CELDAS

A partir de esta idea simple y sencilla, las tablas adquieren otra magnitud cuando les incorporamos toda una cantidad de atributos aplicados sobre cada tipo de etiquetas que las componen.

En cuanto a atributos para tabla hay unos cuantos. Muchos los conoces ya de otras etiquetas, como width, height, align, etc. Hay otros que son especialmente creados para las etiquetas TABLE.

- **cellspacing**: es el espacio entre celdas de la tabla.
- **cellpadding**: es el espacio entre el borde de la celda y su contenido.
- **border**: es el número de píxeles que tendrá el borde de la tabla.
- **bordercolor**: es el rbg que le vas a asignar al borde de la tabla.

En cuanto a las etiquetas "interiores" de una tabla, nos referimos a TR y TD, ten en cuenta:

- Podemos usar prácticamente cualquier tipo de etiqueta dentro de la etiqueta TD para, de esta forma, escribir su contenido.
- Las etiquetas situadas en el interior de la celda no modifican el resto del documento.
- Las etiquetas de fuera de la celda no son tenidas en cuenta por ésta.

Así pues, podemos especificar el formato de nuestras celdas a partir de etiquetas introducidas en su interior o mediante atributos colocados dentro de la etiqueta de celda TD o bien, en algunos casos, dentro de la etiqueta TR, si deseamos que el atributo sea valido para toda la línea. La forma más útil y actual de dar forma a las celdas es a partir de las hojas de estilo en cascada que ya tendrás la oportunidad de abordar más adelante.

Veamos a continuación algunos atributos útiles para la construcción de nuestras tablas. Empecemos viendo atributos que nos permiten modificar una celda en concreto o toda una línea:

- **align**: Justifica el texto de la celda del mismo modo que si fuese el de un párrafo.
- **valign**: Podemos elegir si queremos que el texto aparezca arriba (top), en el centro (middle) o abajo (bottom) de la celda.
- **bgcolor**: Da color a la celda o línea elegida.
- **bordercolor**: Define el color del borde.

Otros atributos que pueden ser únicamente asignados a una celda y no al conjunto de celdas de una línea son:

- **background**: Nos permite colocar un fondo para la celda a partir de un enlace o una imagen.

- **height:** Define la altura de la celda en pixeles o porcentaje.
- **width:** Define la anchura de la celda en pixeles o porcentaje.
- **colspan:** Expande una celda horizontalmente.
- **rowspan:** Expande una celda verticalmente.

Estos últimos cuatro atributos descritos son de gran utilidad. Concretamente, height y width nos ayudan a definir las dimensiones de nuestras celdas de una forma absoluta (en pixeles o puntos de pantalla) o de una forma relativa, es decir por porcentajes referidos al tamaño total de la tabla.

Los atributos rowspan y colspan son también utilizados frecuentemente. Gracias a ellos es posible expandir celdas fusionando éstas con sus vecinas. El valor que pueden tomar estas etiquetas es numérico. El número representa la cantidad de celdas fusionadas.

Así:

```
<td colspan="2">
```

Esta celda tiene un colspan 2
Celda 1, linea 2 Celda 2, linea 2

Por otro lado:

```
<td rowspan="2">
```

Esta celda tiene rowspan="2", Celda por eso tiene fusionada la celda de abajo.	Normal Otra celda normal
---	---

El resto de los atributos presentados presentan una utilidad y uso bastante obvios. Los dejamos a vuestra propia investigación.

FORMULARIOS HTML

Hasta ahora hemos visto la forma en la que el HTML gestiona y muestra la información, esencialmente mediante texto, imágenes y enlaces. Nos queda por ver de qué forma podemos intercambiar información con nuestro visitante. Desde luego, este nuevo aspecto resulta primordial para gran cantidad de acciones que se pueden llevar a cabo mediante la Web: comprar un artículo, llenar una encuesta, enviar un comentario al autor, registrar un usuario, etc.

Los formularios son esas famosas cajas de texto y botones que podemos encontrar en muchas páginas web. Son muy utilizados para realizar búsquedas o bien para introducir datos personales por ejemplo en sitios de comercio electrónico. Los datos que el usuario introduce en estos campos son enviados al correo electrónico del administrador del formulario o bien a un programa que se encarga de procesarlo automáticamente. Nosotros en esta guía no vamos a mostrar como enviar la información al mail, ya que nos interesa, más adelante poder manejar esa información.

QUÉ SE PUEDE HACER CON UN FORMULARIO

Usando HTML podemos únicamente enviar el contenido del formulario a un correo electrónico, es decir, construir un formulario con diversos campos y, a la hora pulsar el botón de enviar, generar un mensaje de que se ha registrado con éxito la información.

Pero para todo lo que sea manejar esa información y guardarla, por ejemplo, en una base de datos vamos a tener que utilizar Java. Como lo haremos lo veremos más adelante en el curso.

Así pues, en resumen, con HTML podremos construir los formularios, con diversos tipos de campos, como cajas de texto, botones de radio, cajas de selección, menús desplegables, etc. Sin embargo, debe quedar claro que desde HTML no se puede manejar esta información para guardarla o enviarla a algún correo, etc. Eso será trabajo de Java.

CÓMO HACER UN FORMULARIO EN HTML

Los formularios son definidos por medio de las etiquetas **FORM** y su cierre. Entre estas dos etiquetas colocaremos todos los campos y botones que componen el formulario. Dentro de esta etiqueta FORM debemos especificar algunos atributos:

action: define el tipo de acción a llevar a cabo con el formulario. Como ya hemos dicho, existen dos posibilidades:

- El formulario es enviado a una dirección de correo electrónico. Para esto hay que poner el mail en el action.
- El formulario es enviado a un programa o script que procesa su contenido. Esta es la posibilidad que más no interesa.

```
<form action="ruta del método que va a manejar la información"></form>
```

method: Este atributo se encarga de especificar la forma en la que el formulario es enviado. Los dos valores posibles que puede tomar esta atributo son POST y GET. A efectos prácticos y, salvo que se os diga lo contrario, daremos siempre el valor POST. Estos conceptos de POST y GET, lo veremos más adelante en el curso.

enctype: Se utiliza para indicar la forma en la que viajará la información que se mande por el formulario. En el caso más corriente, enviar el formulario por correo electrónico, el valor de este atributo debe de ser "text/plain". Así conseguimos que se envíe el contenido del formulario como texto plano dentro del email. Si fuéramos a enviar una imagen dentro del formulario, este atributo debería ser "multipart/form-data". También todos estos conceptos vamos a verlos más adelante.

Este ultimo atributo puede que esté como que no esté, las otras dos si vamos a guardar la información de nuestro formulario en Java, van a estar siempre.

EJEMPLO DE ETIQUETA FORM COMPLETA

Entonces con todo lo anterior ya explicado, la etiqueta completa nos quedaría así:

```
<form action="ruta del metodo que va a manejar la información" method="POST"
      enctype="multipart/form-data"></form>
```

Entre esta etiqueta y su cierre colocaremos el resto de etiquetas que darán forma a nuestro formulario.

CAMPOS DE TEXTO

El lenguaje HTML nos propone una gran diversidad de alternativas a la hora de crear nuestros formularios, es decir, una gran variedad de elementos para diferentes propósitos. Estas van desde la clásica caja de texto, hasta la lista de opciones en un menú desplegable, pasando por las cajas de validación, etc.

Las etiquetas que tenemos que utilizar para crear campos de texto, pueden ser de dos tipos. Veamos en qué consiste cada una de estas modalidades y como podemos implementarlas en nuestro formulario.

ETIQUETA INPUT

Las cajas de texto son colocadas por medio de la etiqueta **INPUT**. Dentro de esta etiqueta hemos de especificar el valor de dos atributos: **type** y **name**.

```
<input type="text" name="nombre">
```

Como todos sabrán un input se ve así:



De este modo expresamos nuestro deseo de crear una caja de texto cuyo contenido será llamado "nombre" (por ejemplo, en el caso de la etiqueta anterior, pero podemos poner distintos nombres a cada uno de los campos de texto que habrán en los formularios).

ATRIBUTO TYPE

Como hemos visto el atributo type nos sirve para especificar el tipo de dato que se va a ingresar en nuestro input, en el ejemplo anterior lo habíamos puesto como tipo text, para que sea una caja de texto y poder ingresar texto. Pero existen otros tipos de valores para el atributo type

NUMBER

Este tipo permite al usuario ingresar números. Los navegadores vienen con validaciones para evitar que el usuario ingrese algo que no sea números. Además, en los navegadores modernos, los campos numéricos suelen venir con controles que permiten a los usuarios cambiar su valor de forma gráfica.

```
<input type="number">
```

Se vería así:



DATE

Este le permite al usuario ingresar una fecha, ya sea mediante una caja de texto o una interfaz gráfica con selector de fecha.

```
<input type="date">
```

Se vería así:



EMAIL

Este tipo permite al usuario ingresar un mail. Los navegadores vienen con validaciones para validar que se esté ingresando con el formato correcto de un mail. Este input se va a ver como un input de texto común y corriente.

```
<input type="email">
```

TEXTO OCULTO

Hay determinados casos en los que podemos desear esconder el texto escrito en el campo input, por medio de círculos negros, de manera que aporte una cierta confidencialidad. Para esto vamos a usar el type **password**.

```
<input type="password">
```

Se vería así:



Más adelante veremos otros valores para el atributo type con otras utilidades

ATRIBUTO NAME

Si vemos de nuevo el ejemplo del principio:

```
<input type="text" name="nombre">
```

En este ejemplo creamos una caja de texto cuyo contenido será llamado "nombre", elegimos nombre, pero podemos ponerles el nombre que queramos.

El nombre del elemento del formulario es de gran importancia para poder identificarlo en nuestro programa de procesamiento (Java).

Además de estos dos atributos, esenciales para el correcto funcionamiento de nuestra etiqueta, existen otra serie de atributos que pueden resultarnos de utilidad pero que no son imprescindibles:

size: define el tamaño de la caja de texto, en número de caracteres visibles. Si al escribir el usuario llega al final de la caja, el texto que escriba a continuación también cabrá dentro del campo pero irá desfilando, a medida que se escribe, haciendo desaparecer la parte de texto que queda a la izquierda.

maxlength: indica el tamaño máximo del texto, en número de caracteres, que puede ser escrito en el campo. En caso que el campo de texto tenga definido el atributo maxlength, el navegador no permitirá escribir más caracteres en ese campo que los que hayamos indicado.

value: en algunos casos puede resultarnos interesante asignar un valor definido al campo en cuestión. Esto puede ayudar al usuario a llenar más rápidamente el formulario o darle alguna idea sobre la naturaleza de datos que se requieren. Este valor inicial del campo puede ser expresado mediante el atributo value. Veamos su efecto con un ejemplo sencillo:

```
<input type="text" name="instituto" value="Egg Educación">
```

Genera un campo de este estilo:

Egg Educación

placeholder: este atributo especifica una pequeña pista que describe el valor esperado de para el campo (input).

La pequeña sugerencia se muestra en el campo de entrada antes de que el usuario ingrese un valor. Una vez que escriba, esa pista va a desaparecer.

```
<input type="text" name="nombre" placeholder="Nombre del usuario">
```

Genera un campo de este estilo:

Nombre del usuario

Nota: recordemos que todos estos ejemplos de input deben ir entre las etiquetas de apertura y de cierre form.

```
<form>  
<input type="text" name="instituto" value="Egg Educación">  
</form>
```

ETIQUETA TEXTAREA PARA TEXTO LARGO

Si deseamos poner a la disposición de usuario un campo de texto donde pueda escribir cómodamente sobre un espacio compuesto de varias líneas, hemos de invocar una nueva etiqueta: TEXTAREA y su cierre correspondiente.

Este tipo de campos son prácticos cuando el contenido a enviar no es un nombre, teléfono, edad o cualquier otro dato breve, sino más bien, un comentario, opinión, etc. en los que existe la posibilidad que el usuario desee llenar varias líneas.

Dentro de la etiqueta textarea deberemos indicar, como para el caso visto anteriormente, el atributo name para asociar el contenido a un nombre que será asemejado a una variable en un lenguaje de programación. Además, podemos definir las dimensiones del campo a partir de los atributos siguientes:

- **rows:** define el número de líneas del campo de texto.
- **cols:** define el número de columnas del campo de texto.

La etiqueta queda por tanto de esta forma:

```
<textarea name="comentario" rows="10" cols="40"></textarea>
```

El resultado es el siguiente:

Asimismo, es posible predefinir el contenido del campo. Para ello, no usaremos el atributo value, sino que escribiremos dentro de la etiqueta el contenido que deseamos atribuirle. Veámoslo:

```
<textarea name="comentario" rows="10" cols="40">Escribe tu comentario...</textarea>
```

Esta etiqueta al igual que el input debe ir dentro de la etiqueta form.

ETIQUETA LABEL

El elemento **LABEL** y su etiqueta de cierre, provee una descripción corta para el campo de texto y que puede ser asociada a un campo de texto. Podemos asociar una etiqueta label a un campo de texto para que el usuario pueda acceder al campo de texto con solo clickear el label. También, como veremos más adelante, cuando veamos las cajas de opciones, clickear en el nombre de la opción para acceder a ella, "tickear" esa opción.

La etiqueta se ve de esta forma:

```
<label>Nombre del Usuario</label>  
<input type="text" name="nombre">
```

Esto en una pagina se vería así:

Nombre del Usuario

Podríamos poner un salto de línea para que el label quede arriba del input, si lo quisieramos.

Nombre del Usuario



ATRIBUTO LABEL

La etiqueta label solo consta del atributo for. Mediante la utilización del atributo for podemos asociar el label con el input. Para lograr esto vamos a tener que utilizar también el atributo ID, este atributo lo explicamos previamente y lo vamos a ver más en detalle en la parte de CSS.

La manera que anclamos un label a un input es, al label le vamos a dar un valor en su atributo for, este va a representar el dato que se va a ingresar en el input y en el input vamos a poner el mismo valor pero en el atributo ID. Entonces, el primer elemento input en el documento con un ID que coincida con el dispuesto en el atributo for puesto en el label, será el control etiquetado para este elemento.

Esto se vería así:

```
<label for="nombre">Nombre del Usuario</label>  
<input type="text" id="nombre" name="nombre">
```

El label y el input se verán igual pero ahora cuando el usuario cliquee el label se va a activar el campo de texto del input para poder ingresar el valor que el usuario necesite. Después vamos a ver un ejemplo más útil con las cajas de opciones.

OTROS ELEMENTOS DE FORMULARIOS

Seguramente hayan notado que los input son un manera muy practica de hacernos llegar la información del navegante. No obstante, en muchos casos, permitir al usuario que escriba cualquier texto permite demasiada libertad y puede que la información que éste escriba no sea la que nosotros estamos necesitando.

Por ejemplo, pensemos que queremos que el usuario indique su país de residencia. En ese caso podríamos ofrecer una lista de países para que seleccione el que sea. Este mismo caso se puede aplicar a gran variedad de informaciones, como el tipo de tarjeta de crédito para un pago, la puntuación que da a un recurso, si quiere recibir o no un boletín de novedades, etc...

Este tipo de opciones predefinidas por nosotros pueden ser expresadas por medio de diferentes campos de formulario. Veamos a continuación cuales son:

LISTAS DE OPCIONES

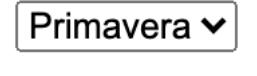
Las listas de opciones son ese tipo de menús desplegables que nos permiten elegir una (o varias) de las múltiples opciones que nos proponen. Para construirlas emplearemos una etiqueta **SELECT**, con su respectivo cierre.

Como para los casos ya vistos, dentro de esta etiqueta definiremos su nombre por medio del atributo name. Cada opción será incluida en una línea precedida de la etiqueta **OPTION**.

Podemos ver, a partir de estas explicaciones, la forma más típica y sencilla de esta etiqueta:

```
<select name="estacion">  
  <option>Primavera</option>  
  <option>Verano</option>  
  <option>Otoño</option>  
  <option>Invierno</option>  
</select>
```

Esto en una pagina se vería así:



Primavera ▾

Y cuando el usuario clickea en el select muestra las opciones así:



✓ Primavera
Verano
Otoño
Invierno

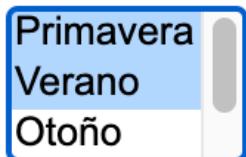
Esta estructura puede verse modificada principalmente a partir de otros dos atributos:

size: indica el número de valores mostrados a la vez en la lista. Lo típico es que no se incluya ningún valor en el atributo size, en ese caso tendremos un campo de opciones desplegable, pero si indicamos un valor para el atributo size aparecerá un campo donde veremos las opciones definidas por size y el resto podrán ser vistos por medio de la barra lateral de desplazamiento.

multiple: permite la selección de más elementos de la lista. Este atributo se expresa sin valor alguno, es decir, no se utiliza con el igual, simplemente se pone para conseguir el efecto, o no se pone si queremos una lista desplegable común.

```
<select name="estacion" size="3" multiple>
```

Esto en una pagina se vería así:



La etiqueta OPTION puede asimismo ser modificada por medio de otros atributos.

selected: del mismo modo que multiple, este atributo no toma ningún valor sino que simplemente indica que la opción que lo presenta esta elegida por defecto.

```
<option selected>Otoño</option>
```

BOTONES DE RADIO

Existe otra alternativa para plantear una elección, en este caso, obligamos al usuario a elegir únicamente una de las opciones que se le proponen.

La etiqueta empleada en este caso es **INPUT** en la cual usaremos el atributo type con el de valor **radio**. Este atributo colocara una casilla pinchable al lado del valor del input. Veamos un ejemplo:

```
<input type="radio" name="estacion" value="1">Primavera  
<br>  
<input type="radio" name="estacion" value="2">Verano  
<br>  
<input type="radio" name="estacion" value="3">Otoño  
<br>  
<input type="radio" name="estacion" value="4">Invierno
```

Esto en una pagina se vería así:

- Primavera
- Verano
- Otoño
- Invierno

En este tipo de input para elegir una opción debemos tocar en la casilla clickeable, pero habíamos explicado previamente en la etiqueta label, que podíamos hacer que la etiqueta label al clickearla se active la caja de texto del input. Ahora, podemos usar eso para que cuando el usuario cliquee la palabra primavera se seleccione esa opción. Esto se vería así:

```
<input type="radio" id="primavera" name="estacion" value="1">  
<label for="primavera">Primavera</label>  
<br>  
<input type="radio" id="verano" name="estacion" value="2">  
<label for="verano">Verano</label>  
<br>
```

```
<input type="radio" id="otono" name="estacion" value="3">
<label for="otono">Otoño</label>
<br>
<input type="radio" id="invierno" name="estacion" value="4">
<label for="invierno">Invierno</label>
```

Esto en la pagina se verá igual que el anterior:

- Primavera
- Verano
- Otoño
- Invierno

La única diferencia va a ser que el usuario va a poder clickear el nombre de la estación que quiere para seleccionar esa opción, además de poder clickear la casilla.

CAJAS DE VALIDACIÓN

Este tipo de elementos pueden ser activados o desactivados por el visitante por un simple click sobre la caja en cuestión. Para esto vamos a usar la etiqueta INPUT con el valor **checkbox** en el atributo type.

```
<input type="checkbox" name="estacion" value="1">Primavera
```

Esto se vera así:

- Primavera
- Verano
- Otoño
- Invierno

ENVIO, BORRADO Y DEMÁS

Ha llegado el momento de explicar cómo podemos hacer un botón para provocar el envío del formulario, entre otras cosas.

Como podremos imaginarnos, en formularios no solamente habrá elementos o campos donde solicitar información del usuario, sino también habrá que implementar otra serie de funciones. Concretamente, han de permitirnos su envío mediante un botón. También puede resultar práctico poder proponer un botón de borrado o bien acompañar el formulario de datos ocultos que puedan ayudarnos en su procesamiento.

BOTÓN DE ENVÍO DE FORMULARIO (BOTÓN DE SUBMIT)

Para dar por finalizado el proceso de relleno del formulario y hacerlo llegar a su gestor, el usuario ha de enviarlo por medio de un botón previsto a tal efecto. Para esto vamos a utilizar la etiqueta **BUTTON** y su respectivo cierre. Dentro el elemento button se puede poner texto (y etiquetas como *<i>*, **, **, *
*, **, etc.). Se vería así:

```
<button type="submit">Enviar</button>
```

Esto en la pagina se verá así:

Enviar

Como puede verse, tan solo hemos de especificar que se trata de un botón de envío (`type="submit"`) y hemos de definir el mensaje que queremos que aparezca escrito en el botón.

BOTÓN DE BORRADO (BOTÓN DE RESET)

Este botón nos permitirá borrar el formulario por completo, en el caso de que el usuario desee rehacerlo desde el principio. Su estructura sintáctica es parecida a la anterior:

```
<button type="reset">Borrar</button>
```

A diferencia del botón de envío, indispensable en cualquier formulario, el botón de borrado resulta meramente optativo y no es utilizado frecuentemente. Hay que tener cuidado de no ponerlo muy cerca del botón de envío y de distinguir claramente el uno del otro, para que ningún usuario borre el contenido del formulario que acaba de llenar por error.

BOTONES NORMALES

Dentro de los formularios también podemos colocar botones normales, pulsables como cualquier otro botón. Estos botones por si solos no tienen mucha utilidad pero podremos necesitarlos para realizar acciones en el futuro. Su sintaxis es la siguiente:

```
<button type="button">Borrar</button>
```

DATOS OCULTOS (CAMPOS HIDDEN)

En algunos casos, aparte de los propios datos rellenados por el usuario, puede resultar práctico enviar datos definidos por nosotros mismos que ayuden al programa en su procesamiento del formulario. Este tipo de datos, que no se muestran en la página pero si pueden ser detectados solicitando el código fuente, no son frecuentemente utilizados por páginas construidas en HTML, son más bien usados por páginas que emplean tecnologías de servidor. No se asusten, veremos más adelante qué quiere decir esto. Tan solo queremos dar constancia de su existencia y de su modo creación. He aquí un ejemplo:

```
<input type="hidden" name="instituto" value="Egg Educación">
```

Esta etiqueta, incluida dentro de nuestro formulario, enviará un dato adicional al programa encargado de la gestión del formulario.

EJEMPLO COMPLETO DE FORMULARIO

Con esto último finalizamos el tema de formularios. Pasemos ahora a ejemplificar todo lo aprendido a partir de la creación de un formulario.

```
<form action="ruta del metodo que va a manejar la información" method="POST" enctype="multipart/form-data"></form>
```

```
<label>Nombre del usuario</label> <br>
```

```
<input type="text" name="nombre"> <br>
```

```

<label>Edad del usuario</label> <br>
<input type="number" name="edad "> <br>
<label>Fecha de nacimiento del usuario</label> <br>
<input type="date" name="fechanac"> <br>
<label>Sexo del usuario</label> <br>
<input type="radio" name="sexo" value="Hombre"> Hombre <br>
<input type="radio" name="sexo" value="Mujer"> Mujer <br>
<label>Pais nacimiento del usuario</label> <br>
<select name="pais">
    <option>Argentina</option>
    <option>Brasil</option>
    <option>Chile</option>
    <option>Uruguay</option>
</select>
<br>
<button type="submit">Enviar</button>
<button type="reset">Borrar</button>

```

Este formulario se verá así:

Nombre del usuario

Edad del usuario

Fecha de nacimiento del usuario

 dd / mm / aaaa

Sexo del usuario

Hombre

Mujer

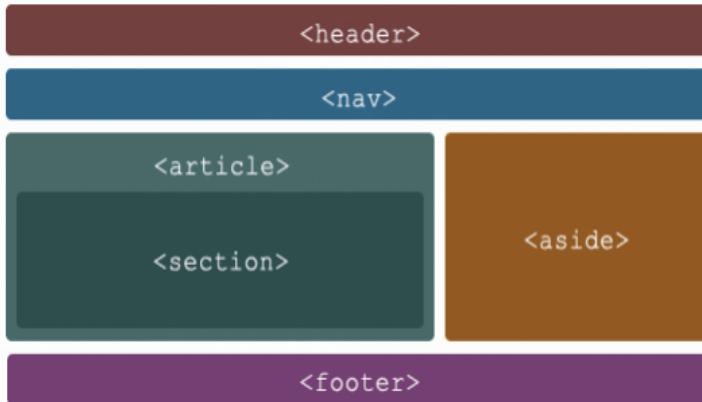
Pais nacimiento del usuario

SECCIONES EN HTML

Las páginas web se trabajan con lo que se conoce como un esquema. El esquema (outline) de una página web es un índice de los apartados de una página web que muestra la relación jerárquica entre los diferentes apartados y subapartados. El concepto de esquema se formalizó en HTML 5 con más precisión que en HTML 4 / XHTML 1 y explica algunas características y formas de utilización de las etiquetas de secciones y bloques de contenido.

En relación a esto se pensó que las páginas de HTML se pueden dividir en secciones y en HTML 5 se introdujo una serie de etiquetas que nos van a ayudar con la división de nuestra página en secciones. Dentro de cada sección van a haber más etiquetas, esto es simplemente para que podemos tener un índice de los apartados de la página web.

La página dividida en secciones con sus respectivas etiquetas se ve así:



```

<body>
  <header>
    <a href="/"><img src=logo.png alt="home"></a>
    <hgroup>
      <h1>Title</h1>
      <h2 class="tagline">
        A lot of effort went into making this effortless.
      </h2>
    </hgroup>
  </header>
  <nav>
    <ul>
      <li><a href="#">home</a></li>
      <li><a href="#">blog</a></li>
      <li><a href="#">gallery</a></li>
      <li><a href="#">about</a></li>
    </ul>
  </nav>
  <section class="articles">
    <article>
      <time datetime="2009-10-22">October 22, 2009</time>
      <h2>
        <a href="#" title="link to this post">Travel day</a>
      </h2>
      <div class="content">
        Content goes here...
      </div>
      <section class="comments">
        <p><a href="#">3 comments</a></p>
      </section>
    </article>
  </section>
  <aside>
    <div class="related"></div>
  </aside>

```

<section>: se utiliza para representar una sección "general" dentro de un documento o aplicación, como un capítulo de un libro. Puede contener subsecciones y si lo acompañamos de h1-h6 podemos estructurar mejor toda la página creando jerarquías del contenido, algo muy favorable para el buen posicionamiento web.

<article>: representa un componente de una página que consiste en una composición autónoma en un documento, página, aplicación, o sitio web con la intención de que pueda ser reutilizado y repetido.

<aside>: representa una sección de la página que abarca un contenido relacionado con el contenido que lo rodea, por lo que se le puede considerar un contenido independiente. Este elemento puede utilizarse para efectos tipográficos, barras laterales, elementos publicitarios u otro contenido que se considere separado del contenido principal de la página.

<header>: representa un grupo de artículos introductorios o de navegación. Está destinado a contener por lo general la cabecera de la sección (un elemento h1-h6 o un elemento hgroup).

<nav>: representa una sección de una página que enlaza a otras páginas o a otras partes dentro de la página. No todos los grupos de enlaces en una página necesita estar en un elemento nav, sólo las secciones que constan de bloques de navegación principales son apropiadas para el elemento de navegación.

<footer>: representa el pie de una sección, con información acerca de la página/sección que poco tiene que ver con el contenido de la página, como el autor, el copyright o el año.

<hgroup>: representa el encabezado de una sección. El elemento se utiliza para agrupar un conjunto de elementos h1-h6 cuando el título tiene varios niveles, tales como subtítulos o títulos alternativos.

ETIQUETAS EXTRAS

En este apartado que va a ser el ultimo de nuestra parte de html vamos a ver unas etiquetas que no hemos visto todavía y que son importantes.

ETIQUETA DIV

La etiqueta div se conoce como etiqueta de división. La etiqueta div se usa en HTML para hacer divisiones de contenido en la página web como (texto, imágenes, encabezado, pie de página, barra de navegación, etc.). La etiqueta Div tiene etiquetas de apertura (`<div>`) y de cierre (`</div>`) y es obligatorio cerrar la etiqueta. Div es la etiqueta más útil en el desarrollo web porque nos ayuda a separar datos en la página web y podemos crear una sección particular para datos o funciones particulares en las páginas web. Cabe aclarar que la etiqueta div genera un salto de linea.

- La etiqueta Div es una etiqueta de nivel de bloque
- Es una etiqueta de contenedor genérica
- Se utiliza para agrupar varias etiquetas de HTML para que se puedan crear secciones y aplicarles estilo.

Un ejemplo que podríamos usar para la etiqueta div es, supongamos que tenemos 3 párrafos que queremos alinear a la izquierda, esto recordemos lo haríamos con el atributo align. Nosotros haríamos algo así:

```
<p align="left">Parrafo 1</p>
<p align="left">Parrafo 2</p>
<p align="left">Parrafo 3</p>
```

Una forma de simplificar nuestro código anterior y de evitar introducir continuamente el atributo align sobre cada una de nuestras etiquetas es utilizando la etiqueta DIV. Vamos a usar un div para generar una sección de todos los párrafos y le pones el atributo align al div.

Esto se vería así:

```
<div align="left">  
<p>Parrafo 1</p>  
<p>Parrafo 2</p>  
<p>Parrafo 3</p>  
</div>
```

Como hemos visto, la etiqueta DIV marca divisiones en las que definimos un bloque de contenido, y a los que podríamos aplicar estilo de manera global, aunque lo correcto sería aplicar ese estilo del lado del CSS.

ETIQUETA SPAN

El elemento span HTML es un contenedor en línea genérico para elementos y contenido en línea. Solía agrupar elementos con fines de estilo (mediante el uso de los atributos de clase o id). La mejor manera de usarlo es cuando no hay ningún otro elemento semántico disponible. span es muy similar a la etiqueta div, pero div es una etiqueta a nivel de bloque y span es una etiqueta en línea. La etiqueta Span es una etiqueta emparejada, lo que significa que tiene una etiqueta de apertura (<) y de cierre (>), y es obligatorio cerrar la etiqueta.

- La etiqueta span se utiliza para agrupar elementos en línea.
- La etiqueta span no realiza ningún cambio visual por sí misma.
- span es muy similar a la etiqueta div, pero div es una etiqueta a nivel de bloque y span es una etiqueta en línea.

Un ejemplo de la etiqueta span, es poner una parte de un párrafo de un color concreto, ya que es una etiqueta en linea, la podemos meter dentro de una etiqueta p.

```
<p>My mother has <span style="color: blue">blue</span> eyes.</p>
```

Esto en una pagina se vería así:

My mother has blue eyes.

La etiqueta span no crea un salto de línea similar a una etiqueta div, sino que permite al usuario separar cosas de otros elementos a su alrededor en una página dentro de la misma línea. Al evitar el salto de línea, solo da como resultado el texto seleccionado para cambiar, manteniendo todos los demás elementos a su alrededor iguales.

Nota: en el apartado de CSS vamos a ver mejor el atributo style y el atributo color. Ahora los estamos usando para el ejemplo.

CSS

INTRODUCCIÓN

CSS es el acrónimo de **Cascading Style Sheets**, o lo que sería en español Hojas de Estilo en Cascada. Es un lenguaje que sirve para especificar el estilo o aspecto de las páginas web. CSS se define en base a un estándar publicado por una organización llamada W3C, que también se encarga de estandarizar el propio lenguaje HTML.

POR QUÉ EXISTE CSS

El lenguaje HTML está limitado a la hora de aplicar forma a un documento. Sirve de manera excelente para especificar el contenido que debe tener una página web, pero no permite definir cómo ese documento se debe presentar al usuario.

Otro motivo que ha hecho necesaria la creación de CSS ha sido la separación del contenido de la presentación. Al inicio las páginas web tenían mezclados en su código HTML el contenido con las etiquetas necesarias para darle forma. Esto tiene sus inconvenientes, ya que la lectura del código HTML se hace pesada y difícil a la hora de buscar errores o depurar las páginas. Además, desde el punto de vista de la riqueza de la información y la utilidad de las páginas a la hora de almacenar su contenido, es un gran problema que los textos están mezclados con etiquetas incrustadas para dar forma a éstos, pues se degrada su utilidad.

CSS SOLVENTA ESTOS PROBLEMAS

Como hemos visto, para facilitar un correcto mantenimiento de las páginas web y para permitir que los diseñadores pudieran trabajar como sería deseable, había que introducir un nuevo elemento en los estándares y éste fue el lenguaje CSS.

CSS se ideó para aplicar el formato en las páginas, de una manera mucho más detallada, con nuevas posibilidades que no estaban al alcance de HTML. Al mismo tiempo, gracias a la posibilidad de aplicar el estilo de manera externa al propio documento HTML, se consiguió que el mantenimiento de las páginas fuese mucho más sencillo.

CARACTERÍSTICAS Y VENTAJAS DE CSS

El modo de funcionamiento de CSS consiste en definir, mediante una sintaxis especial, la forma de presentación que le aplicaremos a los elementos de la página.

Podemos aplicar CSS a muchos niveles, desde un sitio web entero hasta una pequeña etiqueta. Estos son los principales bloques de acción.

- **Una web entera:** de modo que se puede definir en un único lugar el estilo de toda una web, de una sola vez.
- **Un documento HTML o página en particular:** se puede definir la forma de cada uno de los bloques de contenido de una página, en una declaración que afectará a un solo documento de un sitio web.
- **Una porción del documento:** aplicando estilos visibles en un trozo de la página, como podría ser la cabecera.

- **Una etiqueta en concreto:** llegando incluso a poder definir varios estilos diferentes para una sola etiqueta. Esto es muy importante ya que ofrece potencia en nuestra programación. Podemos definir, por ejemplo, varios tipos de párrafos: en rojo, en azul, con márgenes, sin ellos...

La potencia de la tecnología salta a la vista. Pero no solo se queda ahí, ya que además esta sintaxis CSS permite aplicar al documento formato de modo mucho más exacto. Si antes el HTML se nos quedaba corto para maquetar las páginas y teníamos que utilizar trucos para conseguir nuestros efectos, ahora tenemos muchas más herramientas que nos permiten definir esta forma:

- Podemos definir la distancia entre líneas del documento.
- Se puede aplicar identado (sangrado) a las primeras líneas del párrafo.
- Podemos colocar elementos en la página con mayor precisión, y sin lugar a errores.
- Y mucho más, como definir la visibilidad de los elementos, márgenes, subrayados, tachados, etc.

Otra ventaja importante de CSS es la capacidad de especificar las medidas con diversas unidades. Si con HTML tan sólo podíamos definir atributos en las páginas con píxeles y porcentajes, ahora podemos definir utilizando muchas más unidades como:

- Píxeles (px) y porcentaje (%), como antes.
- Pulgadas (in).
- Puntos (pt).
- Centímetros (cm).
- Y otras que veremos más adelante

SINTAXIS HTML

La meta básica del lenguaje Cascading Stylesheet (CSS) es permitir al motor del navegador pintar elementos de la página con características específicas, como colores, posición o decoración. La sintaxis CSS refleja estas metas y estos son los bloques básicos de construcción.

- La propiedad que es un identificador, un nombre leíble por humanos, que define qué característica es considerada.
- El valor que describe como las características deben ser manejadas por el motor. Cada propiedad tiene un conjunto de valores válidos, definido por una gramática formal, así como un significado semántico, implementados por el motor del navegador.

DECLARACIONES DE CSS

Configurando propiedades CSS a valores específicos es la función principal del lenguaje del CSS. Una propiedad y su valor son llamados una declaración, y cualquier motor de CSS calcula qué declaraciones aplican a cada uno de los elementos de una página para mostrarlos apropiadamente y estilizarlos.

Ambos propiedades y valores son sensibles a mayúsculas y minúsculas en CSS. El par se separa por dos puntos, “：“, y los espacios en blanco antes, entre ellos y después, pero no necesariamente dentro de ellos, son ignorados.

Declaración CSS:



Hay más de 100 propiedades diferentes en CSS y cerca de un número infinito de diferentes valores. No todos los pares de propiedades y valores son permitidos, cada propiedad define que valores son válidos. Cuando un valor no es válido para una propiedad específica, la declaración es considerada inválida y es completamente ignorada por el motor del CSS.

BLOQUES DE DECLARACIONES EN CSS

Las declaraciones son agrupadas en bloques, que es una estructura delimitada por una llave de apertura, '{', y una de cierre, '}'. Los bloques en ocasiones puedes anidarse, por lo que las llaves de apertura y cierre deben de coincidir.

Bloque css:

```
{  
    Aquí puede ir cualquier  
    contenido, incluso  
    ningún contenido.  
}
```

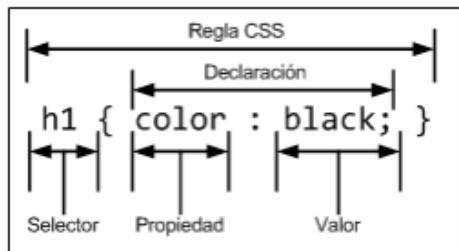
Las llaves delimitan el inicio y el final del bloque.

Esos bloques son naturalmente llamados bloques de declaraciones y las declaraciones dentro de ellos están separadas por un punto y coma, “；”. Un bloque de declaración puede estar vacío, que es contener una declaración nula. Los espacios en blanco alrededor de las declaraciones son ignorados. En cuanto a la última declaración de un bloque, esta no necesita terminar en un punto y coma, aunque es usualmente considerado una buena práctica porque previene el olvidar agregarlo cuando se extienda el bloque con otra declaración.

Bloque css con declaraciones:

```
{  
    background-color : red ;  
    background-style: none;  
}
```

CSS define una serie de términos que permiten describir cada una de las partes que componen los estilos CSS. El siguiente esquema muestra las partes que forman un estilo CSS muy básico:



Los diferentes términos se definen a continuación:

Regla: cada uno de los estilos que componen una hoja de estilos CSS.

Selector: indica el elemento o elementos HTML a los que se aplica la regla CSS.

Declaración: especifica los estilos que se aplican a los elementos. Está compuesta por una o más propiedades CSS.

Propiedad: permite modificar el aspecto de una característica del elemento.

Valor: indica el nuevo valor de la característica modificada en el elemento.

INCLUIR CSS A NUESTRO HTML

CSS sirve para definir el aspecto de las páginas web, eso ya debe haber quedado claro. No obstante, hay diferentes niveles a los que podemos aplicar los estilos. Vamos a ir por orden, describiendo los puntos desde el más específico al más general, de manera que también iremos aumentando la dificultad e importancia de los distintos usos.

PEQUEÑAS PARTES DE LA PÁGINA

Para definir estilos en secciones reducidas de una página se puede utilizar el atributo **style** en la etiqueta sobre la que queremos aplicar estilos. Como valor de ese atributo indicamos en sintaxis CSS las características de estilos. Lo vemos con un ejemplo, pondremos un párrafo en el que determinadas palabras las vamos a visualizar en color verde.

```
<p>My mother has <span style="color: green">blue</span> eyes.</p>
```

ESTILO DEFINIDO PARA UNA ETIQUETA

De este modo podemos hacer que toda una etiqueta muestre un estilo determinado. Por ejemplo, podemos definir un párrafo entero en color rojo y otro en color azul. Para ello utilizamos el atributo **style**, que es admitido por todas las etiquetas del HTML.

```
<p style="color: #990000">
```

Esto es un párrafo de color rojo.

```
</p>
```

```
<p style="color: #000099">
```

Esto es un párrafo de color azul.

```
</p>
```

ESTILO DEFINIDO EN UNA PARTE DE LA PÁGINA

Con la etiqueta `<div>` podemos definir secciones de una página y aplicarle estilos con el atributo **style**, es decir, podemos definir estilos de una vez a todo un bloque de la página.

```
<div style="color: #000099; font-weight: bold">
```

```
<h3>Estas etiquetas van en <strong>azul y negrita</strong></h3>
```

```
<p>
```

Seguimos dentro del DIV, luego permanecen los estilos

```
</p>
```

```
</div>
```

Hasta aquí hemos visto los usos de las CSS más específicos, que se consiguen usando el atributo `style` en las etiquetas. Sin embargo, hay otras formas más avanzadas de usar las CSS, que deberías tener en cuenta porque son todavía más versátiles y recomendadas.

ESTILO DEFINIDO PARA TODA UNA PÁGINA

Podemos definir, en la cabecera del documento, estilos para que sean aplicados a toda la página. Es una manera muy cómoda de darle forma al documento y muy potente, ya que estos estilos serán seguidos en toda la página y nos ahorraremos así "ensuciar" las etiquetas HTML colocando el atributo `style`.

Además, es común que los estilos declarados se quieran aplicar a distintas etiquetas dentro del mismo documento. Gracias a la aplicación de estilos para toda la página, podemos escribir los estilos una vez y usarlos para un número indefinido de etiquetas. Por ejemplo podremos definir el estilo a todos los párrafos una vez y que se aplique igualmente, sea cual sea el número de párrafos del documento. Por último, también tendremos la ventaja que, si más adelante deseamos cambiar los estilos de todas las etiquetas, lo haremos de una sola vez, ya que el estilo fue definido una única vez de manera global.

A grandes rasgos, entre `<style>` y `</style>`, se coloca el nombre de la etiqueta (o selector) para la que queremos definir los estilos y entre llaves `-{ }-` colocamos en sintaxis CSS las características de estilos. El concepto de selectores lo veremos más adelante.

```
<html>
```

```
<head>
```

```
<title>Ejemplo de estilos para toda una pagina</title>
```

```
<style>
```

```
h1 { text-decoration: underline; text-align: center }
```

```
p { font-family: arial,verdana; color: white; background-color: black }
```

```
body { color: black; background-color: #cccccc; text-indent: 1cm }
```

```
</style>
```

```
</head>
<body>
<h1>Pagina con estilos</h1>
<p>Pagina con estilos de ejemplo</p>
</body>
</html>
```

Como se puede apreciar en el código, hemos definido que la etiqueta `<h1>` se presentará

- Subrayado
- Centrada

También, por ejemplo, hemos definido que el cuerpo entero de la página (etiqueta `<body>`) se le apliquen los estilos siguientes:

- Color del texto negro
- Color del fondo grisáceo
- Margen lateral de 1 centímetro

Cabe destacar que muchos de los estilos aplicados a la etiqueta `<body>` son heredados por el resto de las etiquetas del documento, como el color del texto o su tamaño. Esto es así, siempre y cuando no se vuelvan a definir esos estilos en las etiquetas hijas, en cuyo caso el estilo de la etiqueta más concreta será el que mande. Puede verse este detalle en la etiqueta `<p>`, que tiene definidos estilos que ya fueron definidos para `<body>`. Los estilos que se tienen en cuenta son los de la etiqueta `<p>`, que es más concreta.

ESTILO DEFINIDO PARA TODO UN SITIO WEB

Una de las características más potentes del desarrollo con hojas de estilos es la posibilidad de **definir los estilos de todo un sitio web en una única declaración**.

Esto se consigue creando un archivo de extensión `.css` donde tan sólo colocamos las declaraciones de estilos de la página y enlazando todas las páginas del sitio con ese archivo. De este modo, todas las páginas comparten una misma declaración de estilos, reutilizando el código CSS de una manera mucho más potente.

Este es el modelo más ventajoso de aplicar estilos al documento HTML y por lo tanto el más recomendable. De hecho, cualquier otro modo de definir estilos no es considerado una buena práctica y lo tenemos que evitar siempre que se pueda.

Algunas de las ventajas de este modelo de definición de estilos son las siguientes:

- Se ahorra en líneas de código HTML, ya que no tenemos que escribir el CSS en la propia página (lo que reduce el peso del documento y mejora la velocidad de descarga).
- Se mantiene separado correctamente lo que es el contenido (HTML) de la presentación (CSS), que es uno de los objetivos de las hojas de estilo y una de las máximas de todo desarrollador: cada cosa en su sitio.

- Se evita la molestia de definir una y otra vez los estilos con el HTML y lo que es más importante, si cambiamos la declaración de estilos, cambiarán automáticamente todas las páginas del sitio web. Esto es una característica muy deseable, porque aumenta considerablemente la facilidad de mantenimiento del sitio web.

Veamos ahora cómo el proceso para incluir estilos con un fichero externo.

1- Creamos el fichero con la declaración de estilos

Es un fichero de texto normal con la extensión **.css** para aclararnos qué tipo de archivo es. El texto que debemos incluir debe ser escrito exclusivamente en sintaxis CSS, es decir, sería erróneo incluir código HTML en él: etiquetas y demás. Podemos ver un ejemplo a continuación.

El nombre de este archivo va a ser estilos.css.

```
p {
    font-size: 12cm;
    font-family: arial, helvetica;
    font-weight: normal;
}

h1 {
    font-size: 36cm;
    font-family: verdana, arial;
    text-decoration: underline;
    text-align: center;
    background-color: Teal;
}

body {
    background-color: #006600;
    font-family: arial;
    color: White;
}
```

2- Enlazamos la página web con la hoja de estilos

Para ello, vamos a colocar la etiqueta **<link>** dentro de la etiqueta **<head></head>** con los atributos siguientes:

- **rel**: indica el tipo de relación que tiene el recurso enlazado y la página HTML. Para los archivos CSS, siempre se utiliza el valor **stylesheet**.
- **href**: indica la URL del archivo CSS que contiene los estilos. La URL indicada puede ser relativa o absoluta y puede apuntar a un recurso interno o externo al sitio web.

Veamos una página web entera que enlaza con la declaración de estilos anterior:

```

<html>
<head>
<link rel="stylesheet" href="estilos.css">
<title>Ejemplo de pagina que lee estilos </title>
</head>
<body>
<h1>Pagina con estilos</h1>
<p>Pagina con estilos de ejemplo</p>
</body>
</html>

```

SELECTORES CSS

Teniendo en cuenta que ya podemos asignarle estilos a todo un sitio web, mediante un archivo css que usa selectores para elegir las etiquetas a las que asignarles los estilos, también tenemos que entender que existen varios tipos de selectores

Selector Universal

Se utiliza para seleccionar todos los elementos de la página. El siguiente ejemplo elimina el margen y el relleno de todos los elementos HTML (por ahora no es importante fijarse en la parte de la declaración de la regla CSS):

```

* {
    margin: 0;
    padding: 0;
}

```

Selector de Etiqueta

Selecciona todos los elementos de la página cuya etiqueta HTML coincide con el valor del selector. El siguiente ejemplo selecciona todos los párrafos de la página:

```

p {
    text-align: justify;
    font-family: Verdana;
}

```

El siguiente ejemplo selecciona todas las tablas y div de la página:

```

table, div {
    border: 1px solid red;
}

```

Selector Descendente

Selecciona los elementos que se encuentran dentro de otros elementos. Un elemento es descendiente de otro cuando se encuentra entre las etiquetas de apertura y de cierre del otro elemento.

El selector del siguiente ejemplo selecciona todos los elementos `` de la página que se encuentren dentro de un elemento `<p>`.

```
p span { color: red; }
```

Selector de Clase

¿Como hago para aplicarle estilos solo al primer párrafo?

Una de las soluciones más sencillas para aplicar estilos a un solo elemento de la página consiste en utilizar el atributo class de HTML sobre ese elemento para indicar directamente la regla CSS que se le debe aplicar. Ejemplo:

HTML:

```
<body>
<p class="destacado">Parrafo 1</p>
<p class="error">Parrafo 2</p>
<p>Parrafo 3</p>
</body>
```

CSS:

```
.destacado {
font-size: 15px;
}

.error {
color: red;
}
```

En nuestro archivo CSS para especificar una clase, vamos a poner punto(.) y el nombre de la clase que queremos que coincida con valor que pongamos en nuestro atributo class en el html.

Entonces, en el ejemplo podemos ver como el primer párrafo tiene el valor **destacado** y el segundo párrafo el valor **error** para el atributo class y en nuestro archivo CSS, hemos definido un estilo para esas clases.

El beneficio del atributo class, además de dejarnos asignar estilos a un solo elemento, es que después podemos reutilizar esa class para asignarle ese estilo a otros párrafos concretos o a otras etiquetas, solo deberemos ponerle el valor de un estilo que ya existe en el atributo class.

Selector de Id

En un documento HTML, los selectores de ID de CSS buscan un elemento basado en el contenido del atributo id. El atributo ID del elemento seleccionado debe coincidir exactamente con el valor dado en el selector. Este tipo de selectores sólo seleccionan un elemento de la página porque el valor del atributo id no se puede repetir en dos elementos diferentes de una misma página.

Ejemplo:

HTML:

```
<div id="identificador">jEste div tiene un ID especial!</div>
<div>Este solo es un div regular.</div>
```

CSS:

```
#identificador{
background-color: blue;
}
```

En nuestro archivo CSS para especificar un ID, vamos a poner el numeral ('#') y el nombre del ID que queremos que coincida con valor que pongamos en nuestro atributo ID en el html.

Como podemos ver el ID, es muy parecido al atributo class pero la diferencia es que el ID se puede usar para identificar un solo elemento, mientras que una clase se puede usar para agrupar más de uno.

PRIORIDAD EN APLICACIÓN DE ESTILO

Ahora que entendemos los selectores en CSS, tenemos que entender como prioriza los estilos CSS.

Herencia

Los hijos heredan los estilos de sus elementos padres, no es necesario declarar sus estilos si estos se mantienen igual.

```
body{
    color: yellow;
}
h2{
    color: yellow; /*No es necesario*/
}
```

Cascada

Todo estilo sobrescribe a uno anterior.

```
h2{
    color: yellow;
}
h2{
    color: red;
}
```

Especificidad

Cuando hay conflictos de estilos el navegador aplica sólo el de mayor especificidad.

```
h2{  
    color: red  
}  
h2.subtitle{  
    color: purple;  
}
```

UNIDADES DE MEDIDA CSS

Los valores que se pueden asignar a los atributos de estilo se pueden ver en una tabla más adelante en la guía. Muchos de los valores que podemos asignarle son unidades de medida, por ejemplo, el valor del tamaño de un margen o el tamaño de la fuente. Las unidades de medida CSS se pueden clasificar en dos grupos, las relativas y las absolutas. Más la posibilidad de expresar valores en porcentaje.

Absolutas: las unidades absolutas son medidas fijas, que deberían verse igual en todos los dispositivos. Como los centímetros, que son una convención de medida internacional. Pese a que en principio pueden parecer más útiles, puesto que se verán en todos los sistemas igual, tienen el problema de adaptarse menos a las distintas particularidades de los dispositivos que pueden acceder a una web y restan accesibilidad a nuestro web. Puede que en tu ordenador 1 centímetro sea una medida razonable, pero en un móvil puede ser un espacio exageradamente grande, puesto que la pantalla es mucho menor. Se aconseja utilizar, por tanto, medidas relativas.

- **pt** (puntos): Un punto es 1/72 pulgadas.
- **in** (pulgadas)
- **cm** (centímetros)
- **mm** (milímetros)
- **px** (pixeles): Es la unidad mínima de resolución de la pantalla. En realidad suele considerársela una unidad absoluta, relativa o híbrida dependiendo del criterio que se analice. Un pixel equivale a 0.26 milímetros.

Relativas: se llaman así porque son unidades relativas al medio o soporte sobre el que se está viendo la página web, que dependiendo de cada usuario puede ser distinto, puesto que existen muchos dispositivos que pueden acceder a la web, como ordenadores o teléfonos móviles. En principio las unidades relativas son más aconsejables, porque se ajustarán mejor al medio con el que el usuario está accediendo a nuestra web. Son las siguientes:

Unidad em

La unidad em se utiliza para hacer referencia al tamaño actual de la fuente que ha sido establecida en el navegador, que habitualmente es un valor aproximado a 16px (salvo que se modifique por el usuario). De esta forma, podemos trabajar simplificando las unidades a medidas en base a ese tamaño.

Por ejemplo, imaginemos que el tamaño de la fuente establecida en el navegador del usuario es exactamente 16px. Una cantidad 1em equivaldría a 16px, mientras que una cantidad de 2em sería justo el doble: 32px. Por otro lado, una cantidad de 0.5em sería justo la mitad: 8px.



Unidad porcentaje

Porcentaje (%), es una de las unidades relativas más utilizadas. Su valor está calculado siempre en base a otro elemento. Si lo aplicamos sobre una fuente es relativo al tamaño de la fuente declarada en el contexto, pero si lo aplicamos al width de un elemento entonces es relativo al ancho de su contenedor.

El porcentaje se utiliza para definir una unidad en función de la que esté definida en un momento dado. Imaginemos que estamos trabajando en 12pt y definimos una unidad como 150%. Esto sería igual al 150% de los 12pt actuales, que equivale a 18pt.

COLORES EN CSS

Con CSS se puede especificar colores para cada elemento HTML de la página, incluso hay elementos que podrían admitir varios colores, como el color de fondo o el color del borde. Pero bueno, vamos a ver ahora es las distintas maneras de escribir un color en una declaración CSS.

Porque lo más habitual es que especifiquemos un color con su valor RGB. Pero en CSS tenemos otras maneras de declarar colores que pueden interesarnos, como mínimo para poder entender el código CSS cuando lo veamos escrito.

NOTACIÓN HEXADECIMAL RGB

Se especifican los tres valores de color (rojo, verde y azul) con valores en hexadecimal entre 00 y FF.

```
background-color: #ff8800;
```

NOMBRE DEL COLOR

También podemos definir un color por su nombre. Los nombres de colores son en inglés, los mismos que sirven para especificar colores con HTML.

```
color: red;
```

```
border-color: Lime;
```

NOTACIÓN DE COLOR CON PORCENTAJES DE RGB

Se puede definir un color por los distintos porcentajes de valores RGB. Si todos los valores están al 100% el color es blanco. Si todos están al 0% obtendríamos el negro y con combinaciones de distintos porcentajes de RGB obtendríamos cualquier matiz de color.

```
color: rgb(33%, 0%, 0%);
```

NOTACIÓN POR VALORES DECIMALES DE RGB, DE 0 A 255

De una manera similar a la notación por porcentajes de RGB se puede definir un color directamente con valores decimales en un rango desde 0 a 255.

```
color: rgb(200,255,0);
```

De entre todas estas notaciones podemos utilizar la que más nos interese o con la que nos sintamos más a gusto. Nosotros en nuestros ejemplos venimos utilizando la notación hexadecimal RGB por habernos acostumbrado a ella en HTML.

COLOR TRANSPARENTE

Para finalizar, podemos comentar que también existe el color transparente, que no es ningún color, sino que especifica que el elemento debe tener el mismo color que el fondo donde está. Este valor, transparent, sustituye al color. Podemos indicarlo en principio sólo para fondos de elementos, es decir, para el atributo background-color.

```
background-color: transparent;
```

PROPIEDADES CSS

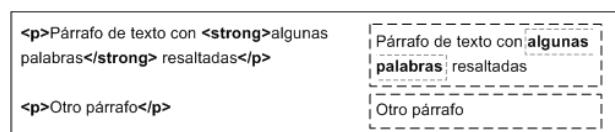
Tanto para practicar en tu aprendizaje como para trabajar con las CSS lo mejor es disponer de las distintas propiedades y valores de estilos que podemos aplicarle a las páginas web.

Aquí puedes ver las propiedades CSS más fundamentales para aplicar estilos a elementos básicos, que te vendrá perfectamente para comenzar con las CSS. Pero antes debemos explicar el concepto de **el modelo de caja** para poder entender algunas de las propiedades de css.

EL MODELO DE CAJA

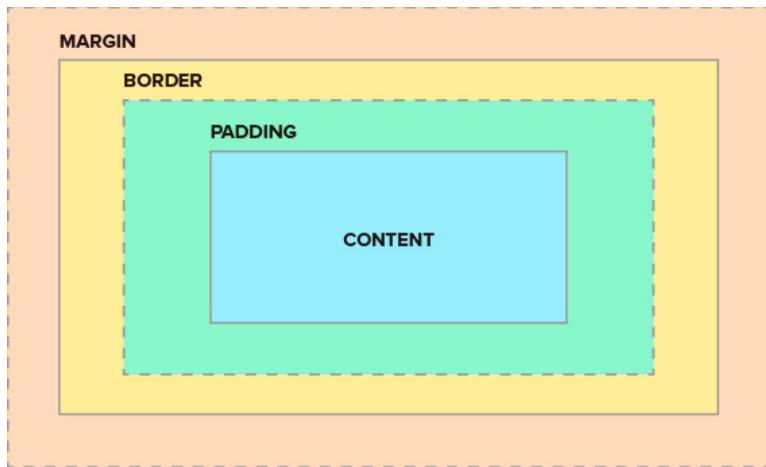
El **modelo de cajas** o "box model" es seguramente la característica más importante del lenguaje de hojas de estilos CSS, ya que condiciona el diseño de todas las páginas web. El modelo de cajas es el comportamiento de CSS que hace que todos los elementos de las páginas se representen mediante cajas rectangulares.

Las cajas de una página se crean automáticamente. Cada vez que se inserta una etiqueta HTML, se crea una nueva caja rectangular que encierra los contenidos de ese elemento. La siguiente imagen muestra las tres cajas rectangulares que crean las tres etiquetas HTML que incluye la página:



Las cajas de las páginas no son visibles a simple vista porque inicialmente no muestran ningún color de fondo ni ningún borde.

Los navegadores crean y colocan las cajas de forma automática, pero CSS permite modificar todas sus características. Cada una de las cajas está formada por cuatro partes, tal y como muestra la siguiente imagen:



- **Contenido** (content): se trata del contenido HTML del elemento (las palabras de un párrafo, una imagen, el texto de una lista de elementos, etc.)
- **Relleno** (padding): espacio libre opcional existente entre el contenido y el borde.
- **Borde** (border): línea que encierra completamente el contenido y su relleno.
- **Margen** (margin): separación opcional existente entre la caja y el resto de cajas adyacentes.

Existen otras dos partes de una caja que son:

- **Imagen de fondo** (background image): imagen que se muestra por detrás del contenido y el espacio de relleno.
- **Color de fondo** (background color): color que se muestra por detrás del contenido y el espacio de relleno.

El relleno y el margen son transparentes, por lo que en el espacio ocupado por el relleno se muestra el color o imagen de fondo (si están definidos) y en el espacio ocupado por el margen se muestra el color o imagen de fondo de su elemento padre (si están definidos). Si ningún elemento padre tiene definido un color o imagen de fondo, se muestra el color o imagen de fondo de la propia página (si están definidos).

Si una caja define tanto un color como una imagen de fondo, la imagen tiene más prioridad y es la que se visualiza. No obstante, si la imagen de fondo no cubre totalmente la caja del elemento o si la imagen tiene zonas transparentes, también se visualiza el color de fondo. Combinando imágenes transparentes y colores de fondo se pueden lograr efectos gráficos muy interesantes.

Teniendo esto en cuenta podemos ver las siguientes propiedades de css:

Ancho

La propiedad CSS que controla la anchura de la caja de los elementos se denomina **width**.

width	Anchura
Valores	<medida> <porcentaje> auto inherit
Se aplica a	Todos los elementos, salvo los elementos en línea que no sean imágenes, las filas de tabla y los grupos de filas de tabla
Valor inicial	auto
Descripción	Establece la anchura de un elemento

Alto

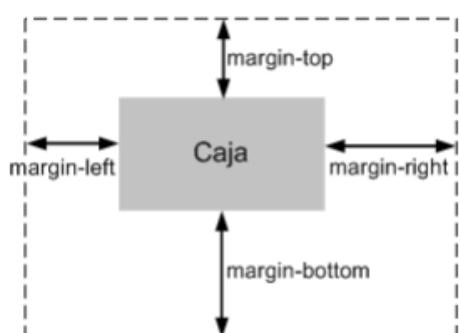
La propiedad CSS que controla la anchura de la caja de los elementos se denomina **height**.

height	Altura
Valores	<medida> <porcentaje> auto inherit
Se aplica a	Todos los elementos, salvo los elementos en línea que no sean imágenes, las columnas de tabla y los grupos de columnas de tabla
Valor inicial	auto
Descripción	Establece la altura de un elemento

Margen

CSS define cuatro propiedades para controlar cada uno de los márgenes horizontales y verticales de un elemento.

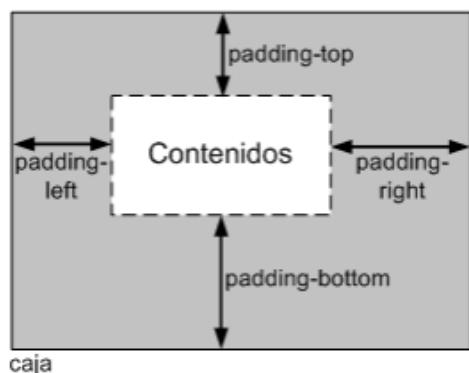
margin-top margin-right margin-bottom margin-left	Margen superior Margen derecho Margen inferior Margen izquierdo
Valores	<medida> <porcentaje> auto inherit
Se aplica a	Todos los elementos, salvo margin-top y margin-bottom que sólo se aplican a los elementos de bloque y a las imágenes
Valor inicial	0
Descripción	Establece cada uno de los márgenes horizontales y verticales de un elemento



Relleno

CSS define cuatro propiedades para controlar cada uno de los espacios de relleno horizontales y verticales de un elemento.

padding-top padding-right padding-bottom padding-left	Relleno superior Relleno derecho Relleno inferior Relleno izquierdo
Valores	<medida> <porcentaje> inherit
Se aplica a	Todos los elementos excepto algunos elementos de tablas como grupos de cabeceras y grupos de pies de tabla
Valor inicial	0
Descripción	Establece cada uno de los rellenos horizontales y verticales de un elemento



Bordes - Tamaño

CSS permite definir el aspecto de cada uno de los cuatro bordes horizontales y verticales de los elementos. Para cada borde se puede establecer su anchura, su color y su estilo.

border-top-width border-right-width border-bottom-width border-left-width	Anchura del borde superior Anchura del borde derecho Anchura del borde inferior Anchura del borde izquierdo
Valores	(<medida> thin medium thick) inherit
Se aplica a	Todos los elementos
Valor inicial	Medium
Descripción	Establece la anchura de cada uno de los cuatro bordes de los elementos

Bordes - Color

El color de los bordes se controla con las cuatro propiedades siguientes:

border-top-color border-right-color border-bottom-color border-left-color	Color del borde superior Color del borde derecho Color del borde inferior Color del borde izquierdo
Valores	<color> transparent inherit
Se aplica a	Todos los elementos
Valor inicial	-
Descripción	Establece el color de cada uno de los cuatro bordes de los elementos

Bordes - Estilo

El color de los bordes se controla con las cuatro propiedades siguientes:

border-top-color border-right-color border-bottom-color border-left-color	Color del borde superior Color del borde derecho Color del borde inferior Color del borde izquierdo
Valores	<color> transparent inherit
Se aplica a	Todos los elementos
Valor inicial	-
Descripción	Establece el color de cada uno de los cuatro bordes de los elementos

Bordes - Forma Resumida

Todos los estilos de los bordes se controlan con la siguiente siguientes:

border	Estilo completo de todos los bordes
Valores	(<medida_borde> <color_borde> <estilo_borde>) inherit
Se aplica a	Todos los elementos
Valor inicial	-
Descripción	Establece el estilo completo de todos los bordes de los elementos

Fondo - Color

El color de fondo se establece con esta propiedad:

background-color	Color de fondo
Valores	<color> transparent inherit
Se aplica a	Todos los elementos
Valor inicial	transparent
Descripción	Establece un color de fondo para los elementos

La imagen de fondo se establece con esta propiedad:

background-image	Imagen de fondo
Valores	<url> none inherit
Se aplica a	Todos los elementos
Valor inicial	none
Descripción	Establece una imagen como fondo para los elementos

Fondo - Repetición

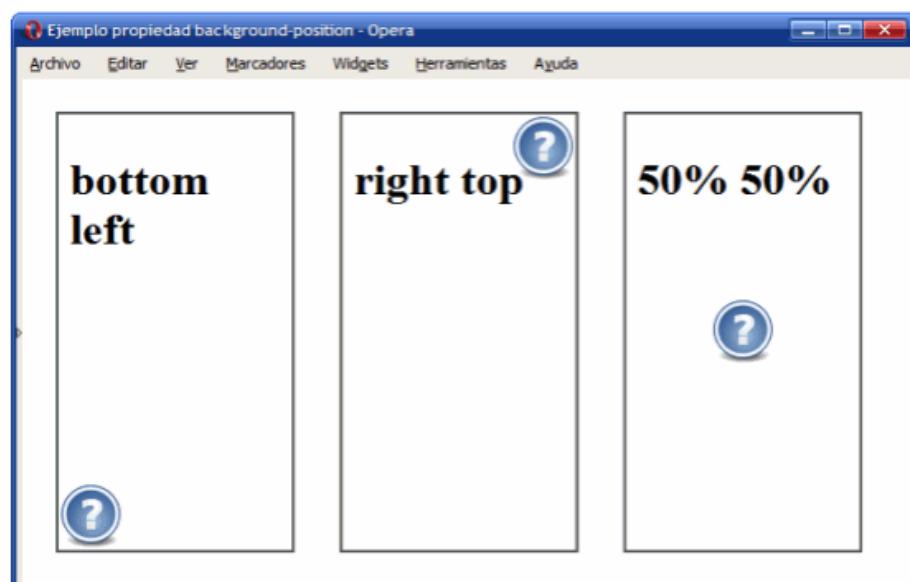
La repetición de la imagen de fondo se configura con esta propiedad:

background-repeat	Repetición de la imagen de fondo
Valores	repeat repeat-x repeat-y no-repeat inherit
Se aplica a	Todos los elementos
Valor inicial	repeat
Descripción	Controla la forma en la que se repiten las imágenes de fondo

Fondo - Posición

La posición de la imagen de fondo se configura con esta propiedad:

background-position	Posición de la imagen de fondo
Valores	((<porcentaje> <medida> left center right) (<porcentaje> <medida> top center bottom)?) ((left center right) (top center bottom)) inherit
Se aplica a	Todos los elementos
Valor inicial	0% 0%
Descripción	Controla la posición en la que se muestra la imagen en el fondo del elemento



Fondo - Imagen de Fondo

Para controlar la manera de visualizar la imagen de fondo:

background-attachment	Comportamiento de la imagen de fondo
Valores	<code>scroll fixed inherit</code>
Se aplica a	Todos los elementos
Valor inicial	<code>scroll</code>
Descripción	Controla la forma en la que se visualiza la imagen de fondo: permanece fija cuando se hace scroll en la ventana del navegador o se desplaza junto con la ventana

Fondo - Resumida

Establecer todas las propiedades de fondo:

background	Fondo de un elemento
Valores	<code>(<background-color> <background-image> <background-repeat> <background-attachment> <background-position>) inherit</code>
Se aplica a	Todos los elementos
Valor inicial	-
Descripción	Establece todas las propiedades del fondo de un elemento

Tipografía - Resumida

CSS define numerosas propiedades para modificar la apariencia del texto. A pesar de que no dispone de tantas posibilidades como los lenguajes y programas específicos para crear documentos impresos, CSS permite aplicar estilos complejos y muy variados al texto de las páginas web. La propiedad básica que define CSS relacionada con la tipografía se denomina color y se utiliza para establecer el color de la letra.

color	Color del texto
Valores	<code><color> inherit</code>
Se aplica a	Todos los elementos
Valor inicial	Depende del navegador
Descripción	Establece el color de letra utilizado para el texto

Tipografía - Fuente

La otra propiedad básica que define CSS relacionada con la tipografía se denomina font-family y se utiliza para indicar el tipo de letra con el que se muestra el texto.

font-family	Tipo de letra
Valores	<code>((<nombre_familia> <familia_generica>) (,<nombre_familia> <familia_generica>)*) inherit</code>
Se aplica a	Todos los elementos
Valor inicial	Depende del navegador
Descripción	Establece el tipo de letra utilizado para el texto

Tipografía – Tamaño

Una vez seleccionado el tipo de letra, se puede modificar su tamaño mediante la propiedad font-size.

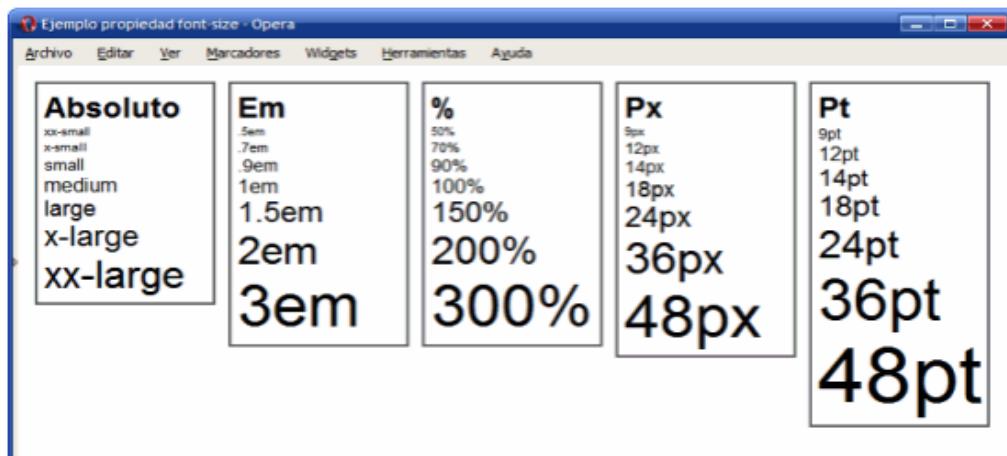
font-size	Tamaño de letra
Valores	<tamaño_absoluto> <tamaño_relativo> <medida> <porcentaje> inherit
Se aplica a	Todos los elementos
Valor inicial	medium
Descripción	Establece el tamaño de letra utilizado para el texto

Tipografía – Tamaño

Además de todas las unidades de medida relativas y absolutas y el uso de porcentajes, CSS permite utilizar una serie de palabras clave para indicar el tamaño de letra del texto:

Tamaño Absoluto: indica el tamaño de letra de forma absoluta mediante alguna de las siguientes palabras clave: xx-small, x-small, small, medium, large, x-large, xx-large.

Tamaño Relativo: indica de forma relativa el tamaño de letra del texto mediante dos palabras clave (larger, smaller) que toman como referencia el tamaño de letra del elemento padre.



Tipografía – Grosor

Una vez indicado el tipo y el tamaño de letra, es habitual modificar otras características como su grosor (texto en negrita) y su estilo (texto en cursiva). La propiedad que controla la anchura de la letra es font-weight.

font-weight	Anchura de la letra
Valores	normal bold bolder lighter 100 200 300 400 500 600 700 800 900 inherit
Se aplica a	Todos los elementos
Valor inicial	normal
Descripción	Establece la anchura de la letra utilizada para el texto

Una vez indicado el tipo y el tamaño de letra, es habitual modificar otras características como su grosor (texto en negrita) y su estilo (texto en cursiva). La propiedad que controla la anchura de la letra es font-weight.

font-style	Estilo de la letra
Valores	<code>normal italic oblique inherit</code>
Se aplica a	Todos los elementos
Valor inicial	<code>normal</code>
Descripción	Establece el estilo de la letra utilizada para el texto

Texto - Alineación

Para establecer la alineación del contenido del elemento. La propiedad text-align no sólo alinea el texto que contiene un elemento, sino que también alinea todos sus contenidos, como por ejemplo las imágenes.

text-align	Alineación del texto
Valores	<code>left right center justify inherit</code>
Se aplica a	Elementos de bloque y celdas de tabla
Valor inicial	<code>left</code>
Descripción	Establece la alineación del contenido del elemento

Texto - Interlineado

El interlineado de un texto se controla mediante la propiedad line-height, que permite controlar la altura ocupada por cada línea de texto:

line-height	Interlineado
Valores	<code>normal <numero> <medida> <porcentaje> inherit</code>
Se aplica a	Todos los elementos
Valor inicial	<code>normal</code>
Descripción	Permite establecer la altura de línea de los elementos

Texto - Decoración

El valor underline subraya el texto.

El valor overline añade una línea en la parte superior del texto.

El valor line-through muestra el texto tachado con una línea continua, por lo que su uso tampoco es muy habitual.

El valor blink muestra el texto parpadeante.

text-decoration	Decoración del texto
Valores	<code>none (underline overline line-through blink) inherit</code>
Se aplica a	Todos los elementos
Valor inicial	<code>none</code>
Descripción	Establece la decoración del texto (subrayado, tachado, parpadeante, etc.)

Texto - Transformación

El valor capitalize transforma a mayúsculas la primera letra de las palabras del texto.

El valor uppercase transforma a mayúsculas todo el texto.

El valor lowercase transforma a minúsculas todo el texto.

text-transform	Transformación del texto
Valores	<code>capitalize uppercase lowercase none inherit</code>
Se aplica a	Todos los elementos
Valor inicial	<code>none</code>
Descripción	Transforma el texto original (lo transforma a mayúsculas, a minúsculas, etc.)

Imágenes – Ancho / Altura

Utilizando las propiedades width y height, es posible mostrar las imágenes con cualquier altura/anchura, independientemente de su altura/anchura real:

```
#destacada {
    width: 120px;
    height: 250px;
}



```

No obstante, si se utilizan alturas/anchuras diferentes de las reales, el navegador deforma las imágenes y el resultado estético es muy desagradable.

Imágenes – Bordes

Cuando una imagen forma parte de un enlace, los navegadores muestran por defecto un borde azul grueso alrededor de las imágenes. Por tanto, una de las reglas más utilizadas en los archivos CSS es la que elimina los bordes de las imágenes con enlaces:

```
img {
    border: none;
}
```

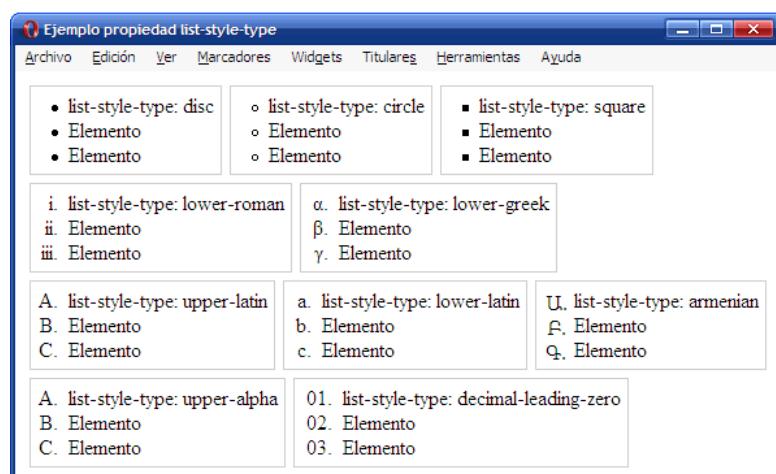
Listas – Viñetas Personalizadas

Por defecto, los navegadores muestran los elementos de las listas no ordenadas con una viñeta formada por un pequeño círculo de color negro. Los elementos de las listas ordenadas se muestran por defecto con la numeración decimal utilizada en la mayoría de países.

No obstante, CSS define varias propiedades para controlar el tipo de viñeta que muestran las listas, además de poder controlar la posición de la propia viñeta. La propiedad básica es la que controla el tipo de viñeta que se muestra y que se denomina `list-style-type`.

Propiedad	list-style-type
Valores	disc circle square decimal decimal-leading-zero lower-roman upper-roman lower-greek lower-latin upper-latin armenian georgian lower-alpha upper-alpha none inherit
Se aplica a	Elementos de una lista
Valor inicial	disc
Descripción	Permite establecer el tipo de viñeta mostrada para una lista

La siguiente imagen muestra algunos de los valores definidos por la propiedad `list-style-type`:

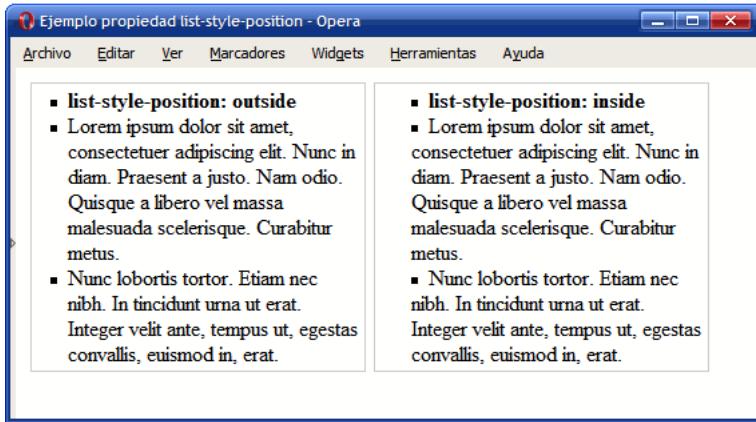


Listas – Viñetas Posición

La propiedad `list-style-position` permite controlar la colocación de las viñetas.

Propiedad	list-style-position
Valores	inside outside inherit
Se aplica a	Elementos de una lista
Valor inicial	outside
Descripción	Permite establecer la posición de la viñeta de cada elemento de una lista

La diferencia entre los valores `outside` y `inside` se hace evidente cuando los elementos contienen mucho texto, como en la siguiente imagen:



Utilizando las propiedades anteriores (list-style-type y list-style-position), se puede seleccionar el tipo de viñeta y su posición, pero no es posible personalizar algunas de sus características básicas como su color y tamaño.

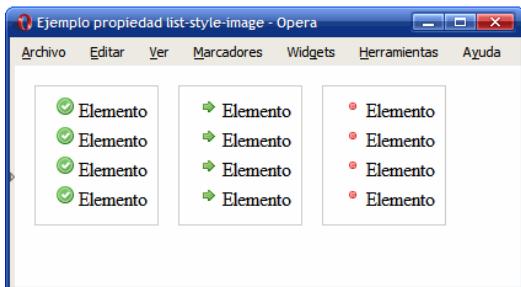
Listas – Viñetas Aspecto

Cuando se requiere personalizar el aspecto de las viñetas, se debe emplear la propiedad list-style-image, que permite mostrar una imagen propia en vez de una viñeta automática.

Propiedad	list-style-image
Valores	url none inherit
Se aplica a	Elementos de una lista
Valor inicial	none
Descripción	Permite reemplazar las viñetas automáticas por una imagen personalizada

Las imágenes personalizadas se indican mediante la URL de la imagen. Si no se encuentra la imagen o no se puede cargar, se muestra la viñeta automática correspondiente (salvo que explícitamente se haya eliminado mediante la propiedad list-style-type).

La siguiente imagen muestra el uso de la propiedad list-style-image mediante tres ejemplos sencillos de listas con viñetas personalizadas:



Las reglas CSS correspondientes al ejemplo anterior se muestran a continuación:

```
ul.ok { list-style-image: url("imagenes/ok.png"); }

ul.flecha { list-style-image: url("imagenes/flecha.png"); }

ul.circulo { list-style-image: url("imagenes/circulo_rojo.png"); }
```

Listas – Menú Vertical

Los sitios web correctamente diseñados emplean las listas de elementos para crear todos sus menús de navegación. Utilizando la etiqueta `` de HTML se agrupan todas las opciones del menú y haciendo uso de CSS se modifica su aspecto para mostrar un menú horizontal o vertical.

A continuación se muestra la transformación de una lista sencilla de enlaces en un menú vertical de navegación.

```
<ul>
- <a href="#">Elemento 1</a></li>
- <a href="#">Elemento 2</a></li>
- <a href="#">Elemento 3</a></li>
- <a href="#">Elemento 4</a></li>
- <a href="#">Elemento 5</a></li>
- <a href="#">Elemento 6</a></li>
</ul>

```

Aspecto final del menú vertical:



El proceso de transformación de la lista en un menú requiere de los siguientes pasos:

1) Definir la anchura del menú:

```
ul.menu { width: 180px; }
```

2) Eliminar las viñetas automáticas y todos los márgenes y espaciados aplicados por defecto:

```
ul.menu {
list-style: none;
margin: 0;
padding: 0;
width: 180px;
}
```

3) Añadir un borde al menú de navegación y establecer el color de fondo y los bordes de cada elemento del menú:

```
ul.menu {  
    border: 1px solid #7C7C7C;  
    border-bottom: none;  
    list-style: none;  
    margin: 0;  
    padding: 0;  
    width: 180px;  
}  
  
ul.menu li {  
    background: #F4F4F4;  
    border-bottom: 1px solid #7C7C7C;  
    border-top: 1px solid #FFF;  
}
```

4) Aplicar estilos a los enlaces: mostrarlos como un elemento de bloque para que ocupen todo el espacio de cada del menú, añadir un espacio de relleno y modificar los colores y la decoración por defecto:

```
ul.menu li a {  
    color: #333;  
    display: block;  
    padding: .2em 0 .2em .5em;  
    text-decoration: none;  
}
```

Tablas – Bordes Celdas

Cuando se aplican bordes a las celdas de una tabla, el aspecto por defecto con el que se muestra en un navegador es el siguiente:



The screenshot shows a table with 20 cells arranged in 4 rows and 5 columns. The columns are labeled A through E at the top. The rows are labeled a through d on the left. Each cell contains a number from 1 to 4. The table has a border around each individual cell, creating a distinct frame for each value. The entire table is contained within a window titled "Ejemplo propiedad border-spacing - Opera".

A	B	C	D	E
a	1	2	3	4
b	1	2	3	4
c	1	2	3	4
d	1	2	3	4

El código HTML y CSS del ejemplo anterior se muestra a continuación:

```

.normal {
    width: 250px;
    border: 1px solid #000;
}

.normal th, .normal td {
    border: 1px solid #000;
}

```

```

<table class="normal" summary="Tabla genérica">
    <tr>
        <th scope="col">A</th>
        <th scope="col">B</th>
        <th scope="col">C</th>
        <th scope="col">D</th>
        <th scope="col">E</th>
    </tr>
</table>

```

El estándar CSS 2.1 define dos modelos diferentes para el tratamiento de los bordes de las celdas. La propiedad que permite seleccionar el modelo de bordes es border-collapse:

Propiedad	border-collapse
Valores	collapse separate inherit
Se aplica a	Todas las tablas
Valor inicial	separate
Descripción	Define el mecanismo de fusión de los bordes de las celdas adyacentes de una tabla

El modelo collapse fusiona de forma automática los bordes de las celdas adyacentes, mientras que el modelo separate fuerza a que cada celda muestre sus cuatro bordes. Por defecto, los navegadores utilizan el modelo separate, tal y como se puede comprobar en el ejemplo anterior. Ejemplo collapse:



El código CSS completo del ejemplo anterior se muestra a continuación:

```

.normal {
    width: 250px;
    border: 1px solid #000;
    border-collapse: collapse;
}

.normal th, .normal td {
    border: 1px solid #000;
}

<table class="normal" summary="Tabla genérica">
<tr>

    <th scope="col">A</th>
    <th scope="col">B</th>
    <th scope="col">C</th>
    <th scope="col">D</th>
    <th scope="col">E</th>

</tr>

</table>

```

Si se opta por el modelo separate (que es el que se aplica si no se indica lo contrario) se puede utilizar la propiedad border-spacing para controlar la separación entre los bordes de cada celda.

Propiedad	border-spacing
Valores	unidad de medida unidad de medida? inherit
Se aplica a	Todas las tablas
Valor inicial	0
Descripción	Establece la separación entre los bordes de las celdas adyacentes de una tabla

Si solamente se indica como valor una medida, se asigna ese valor como separación horizontal y vertical. Si se indican dos medidas, la primera es la separación horizontal y la segunda es la separación vertical entre celdas.

La propiedad border-spacing sólo controla la separación entre celdas y por tanto, no se puede utilizar para modificar el tipo de modelo de bordes que se utiliza. En concreto, si se establece un valor igual a 0 para la separación entre los bordes de las celdas, el resultado es muy diferente al modelo collapse:

The screenshot shows a table with five columns labeled A through E. Each column contains four rows labeled a through d. The table has a border and a border-spacing of 10px between the cells.

A	B	C	D	E
a	1	2	3	4
b	1	2	3	4
c	1	2	3	4
d	1	2	3	4

Formularios – Campos de Texto

Por defecto, los campos de texto de los formularios no incluyen ningún espacio de relleno, por lo que el texto introducido por el usuario aparece pegado a los bordes del cuadro de texto.

Añadiendo un pequeño padding a cada elemento <input>, se mejora notablemente el aspecto del formulario:

The screenshot shows two examples of a form. The top section is titled "Formulario sin padding en los input" and the bottom section is titled "Formulario con padding en los input". Both sections contain two input fields: "Nombre" with value "lorem ipsum" and "Contraseña" with value "*****". In the "Formulario con padding en los input" section, there is visible padding around the input fields, making the text easier to read.

La regla CSS necesaria para mejorar el formulario es muy sencilla:

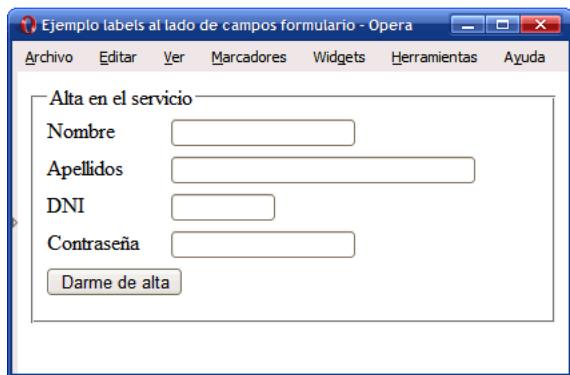
```
form.elegante input {
    padding: .2em;
}
```

Formularios – Labels Alineadas y Formateadas

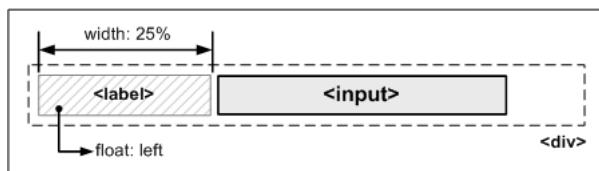
Los elementos <input> y <label> de los formularios son elementos en línea, por lo que el aspecto que muestran los formularios por defecto, es similar al de la siguiente imagen:

The screenshot shows a form titled "Alta en el servicio". It contains three input fields: "Nombre" (with placeholder "Nombre"), "Apellidos" (with placeholder "Apellidos"), and "DNI" (with placeholder "DNI"). Below these fields is a label "Contraseña" followed by an input field. At the bottom is a button labeled "Darme de alta". The labels are positioned directly above their respective input fields.

Aprovechando los elementos <label>, se pueden aplicar unos estilos CSS sencillos que permitan mostrar el formulario con el aspecto de la siguiente imagen:



Para mostrar un formulario tal y como aparece en la imagen anterior no es necesario crear una tabla y controlar la anchura de sus columnas para conseguir una alineación perfecta. Sin embargo, sí que es necesario añadir un nuevo elemento (por ejemplo un <div>) que encierre a cada uno de los campos del formulario (<label> y <input>). El esquema de la solución propuesta es el siguiente:



Por tanto, en el código HTML del formulario se añaden los elementos <div>:

```
<form>
  <fieldset>
    <legend>Alta en el servicio</legend>
    <div>
      <label for="nombre">Nombre</label>
      <input type="text" id="nombre" />
    </div>
    <div>
      <label for="apellidos">Apellidos</label>
      <input type="text" id="apellidos" size="35" />
    </div>
  </fieldset>
</form>
```

Y en el código CSS se añaden las reglas necesarias para alinear los campos del formulario:

```
div {
  margin: .4em 0;
```

```
div label {  
    width: 25%;  
    float: left;  
}
```

PSEUDO-CLASES

CSS también permite aplicar diferentes estilos a un mismo enlace en función de su estado. De esta forma, es posible cambiar el aspecto de un enlace cuando por ejemplo el usuario pasa el ratón por encima o cuando el usuario pincha sobre ese enlace.

Como con los atributos id o class no es posible aplicar diferentes estilos a un mismo elemento en función de su estado, CSS introduce un nuevo concepto llamado pseudo-clases. En concreto, CSS define las siguientes cuatro pseudo-clases:

- **:link**, aplica estilos a los enlaces que apuntan a páginas o recursos que aún no han sido visitados por el usuario.
- **:visited**, aplica estilos a los enlaces que apuntan a recursos que han sido visitados anteriormente por el usuario. El historial de enlaces visitados se borra automáticamente cada cierto tiempo y el usuario también puede borrarlo manualmente.
- **:hover**, aplica estilos al enlace sobre el que el usuario ha posicionado el puntero del ratón.
- **:active**, aplica estilos al enlace que está clickeado el usuario.

Esto se vería así:

```
/* link sin visitar */  
a:link {  
    color: red;  
}  
/* link visitado */  
a:visited {  
    color: green;  
}  
  
/* mouse sobre el link */  
a:hover {  
    color: pink;  
}  
  
/* link clickeado */  
a:active {  
    color: blue;  
}
```

BOOTSTRAP

Es un framework de interfaz de usuario, de código abierto, creado para un desarrollo web más rápido y sencillo. Mark Otto y Jacob Thornton fueron los creadores iniciales. El framework combina CSS y JavaScript para estilizar los elementos de una página HTML.

Contiene todo tipo de plantillas de diseño basadas en HTML y CSS para diversas funciones y componentes, como navegación, sistema de cuadrícula, carruseles de imágenes y botones.

Si bien Bootstrap ahorra tiempo al desarrollador de tener que administrar las plantillas repetidamente, su objetivo principal es crear sitios responsive. Permite que la interfaz de usuario de un sitio web funcione de manera óptima en todos los tamaños de pantalla, ya sea en teléfonos de pantalla pequeña o en dispositivos de escritorio de pantalla grande.

Por lo tanto, los desarrolladores no necesitan crear sitios específicos para dispositivos y limitar su rango de audiencia.

ARCHIVOS PRIMARIOS DE BOOTSTRAP

Ya sabemos qué es Bootstrap; consiste en una colección de sintaxis que realizan funciones específicas. Debido a esto, tiene sentido que el marco tenga solo tres diferentes tipos de archivos. A continuación, detallamos los tres archivos principales que administran esta interfaz de usuario y la funcionalidad de un sitio web.

BOOTSTRAP.CSS

Esta es la hoja de estilos de bootstrap, gracias a esta podremos implementar estilos ya definidos y así estilizar nuestra página de una manera sencilla. Además las plantillas que contiene bootstrap, usan esta hoja de estilos.

BOOTSTRAP.JS

Este archivo es la parte principal de Bootstrap. Consiste en archivos JavaScript que son responsables de la interactividad del sitio web.

CÓMO USAR BOOTSTRAP

Para utilizar bootstrap lo único que vamos a tener que hacer es ir a estas dos páginas:

<https://getbootstrap.com/docs/4.5/getting-started/introduction/#css>

<https://getbootstrap.com/docs/4.5/getting-started/introduction/#js>

Dentro de estas dos páginas vamos a encontrar una etiqueta link para el CSS de Bootstrap y unas etiquetas script para el Javascript de Bootstrap.

Para poder usar Bootstrap lo que haremos es pegar el link con la hoja de estilos de Bootstrap en la etiqueta <head> de nuestro html y las etiquetas script antes de la etiqueta de cierre </body>.

Esto se vería así:

```

<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@4.5.3/dist/css/bootstrap.min.css"
      integrity="sha384-TX8t27EcRE3e/ihU7zmQxVncDAy5ulKz4rEkgIXeMed4M0jlfDPvg6uqKI2xXr2"
      crossorigin="anonymous">
    <title>Pagina con bootstrap</title>
  </head>
  <body>
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+lbbVYUew+OrCXaRkfj"
      crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.5.3/dist/js/bootstrap.bundle.min.js"
      integrity="sha384-ho+j7jyWK8fNQe+A12Hb8AhRq26LrZ/JpcUGGOn+Y7RsweNrtN/tE3MoK7ZeZDyx"
      crossorigin="anonymous"></script>
  </body>
</html>

```

Una vez que hemos hecho esto, ya podemos usar Bootstrap y sus plantillas.

PLANTILLAS

Dado que es uno de los framework más utilizados, podemos encontrar un amplio abanico de marcos de trabajo pensados y diseñados a partir de los componentes y estilos que presenta Bootstrap, de modo que existen variables y ejemplos listos para utilizar en proyectos específicos.

<https://themes.getbootstrap.com/official-themes/>

EJERCICIOS DE APRENDIZAJE

Para la realización de los ejercicios que se describen a continuación, es necesario tener instalado Visual Studio Code o alguna aplicación parecida. Si no lo tienen instalado aquí les dejamos el link: [Visual Studio Code](#).



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

Recomendamos también ir al apartado de bibliografía al final de la guía por si necesitamos reforzar o queremos saber más sobre los temas vistos.

1. Crear un archivo HTML, que contenga un encabezado `<h1>` seguido de un párrafo y un encabezado `<h2>` seguido de otro párrafo. Después, hacer que un párrafo se muestre en negrita y el otro en itálica. Utilizar saltos de línea si los consideran necesarios.
2. Ahora vamos a tener que centrar nuestros encabezados y nuestros párrafos alinearlos a la izquierda. A continuación, después del último párrafo vamos a tener que crear una lista ordenada de lo que queramos. Tendremos que mostrar la lista donde la enumeración sean letras del alfabeto. **Sin usar CSS**
3. Ahora crearemos otro archivo HTML en que crearemos una lista anidada de enlaces, deberá verse así:

- **Buscadores**
 - **Google**
 - **Bing**
- **Redes sociales**
 - **Instagram**
 - **Twitter**

Cada buscador y red social que sale en la lista deben ser links a las respectivas páginas.

Recordemos que no debemos utilizar CSS para lograr ninguna de estas tareas, la idea es practicar HTML por su cuenta y después sumaremos CSS.

4. Crear un nuevo archivo HTML en el que explicaremos la receta para hacer papas fritas. La pagina debería verse así:

Papas fritas

Receta de papas fritas caseras



Ingredientes

- 3 o 4 papas (300gr)
- Aceite
- Sal

Elaboración (Pasos)

- Pelar las papas
- Cortalas en baston
- Calentar aceite en una sartén
- Cocinar hasta que estén doradas
- Removerlas del aceite y salar al gusto

5. Ahora vamos a crear una pagina web de nuestra banda favorita, vamos a mostrar un ejemplo con los Beatles:

Los Beatles

Es una banda de rock formada en el año 1960 en Liverpool.



Ingrediantes

- Paul McCartney
- John Lennon
- Ringo Star
- George Harrison

Año	Disco
1965	Help!
1968	The Beatles
1969	Abbey Road

"Abbey Road fue su ultimo disco".

La tabla tendrá bordes que se lo debemos agregar sin css. Investigar atributo border.

6. Por ultimo vamos a crear un formulario para registrar un usuario que se vea de la siguiente manera:

Registrar un usuario

Nombre del usuario

Contraseña del usuario

Edad del usuario

Fecha de nacimiento del usuario

Sexo del usuario
 Hombre
 Mujer
 Prefiero no decir

País nacimiento del usuario

Es importante que en las casillas de sexo del usuario se puede clickear la/s palabra/s hombre, mujer o prefiero no decir para seleccionar la opción. Recordemos que eso lo podemos hacer con la etiqueta label.

7. Crear un archivo HTML y un archivo CSS, vamos a linkear el archivo CSS al archivo HTML y vamos hacer lo mismo que el primer ejercicio, pero ahora la pagina va a tener un color de fondo a elección, el encabezado H1 tiene que tener una fuente a elección y estar centrado, lo mismo para el encabezado H2, y por ultimo los párrafos deben tener un fuente a elección, deben tener un color a elección y deben estar centrados a la izquierda.
8. Definir las reglas CSS que permiten mostrar los enlaces con los siguientes estilos:
- En su estado normal, los enlaces se muestran de color rojo #CC0000.
 - Cuando el usuario pasa su ratón sobre el enlace, se muestra con un color de fondo rojo #CC0000 y la letra de color blanco #FFF.
 - Los enlaces visitados se muestran en color gris claro #CCC.

9. A partir del siguiente código HTML proporcionado, añadir las reglas CSS necesarias para que la página resultante tenga el mismo aspecto que el de la siguiente imagen:

****Lore ipsum dolor sit amet****

Nulla pretium. Sed tempus nunc vitae neque. **Suspendisse gravida**, metus a scelerisque sollicitudin, lacus velit ultricies nisl, nonummy tempus neque diam quis felis. **Eti sagittis tortor** sed arcu sagittis tristique.

Aliquam tincidunt, sem eget volutpat porta

Vivamus velit dui, placerat vel, feugiat in, ornare et, urna. **Aenean turpis metus**, **aliquam non**, **tristique in**, pretium varius, sapien. Proin vitae nisi. Suspendisse **porttitor purus ac elit**. Suspendisse eleifend odio at dui. In in elit sed metus pretium elementum.

	Título columna 1	Título columna 2
Título fila 1	Donec purus ipsum	Curabitur blandit
Título fila 2	Donec purus ipsum	Curabitur blandit
	Título columna 1	Título columna 2

Donec purus ipsum, posuere id, venenatis at, placerat ac, lorem. Curabitur blandit, eros sed gravida aliquet, risus justo porta lorem, ut mollis lectus tortor in orci. Pellentesque nec augue.

Fusce nec felis eu diam pretium adipiscing. **Nunc elit elit**, **vehicula vulputate**, venenatis in, posuere id, lorem. Eti sagittis, tellus in ultrices accumsan, diam nisi feugiat ante, eu congue magna mi non nisl.

Vivamus ultrices aliquet augue. **Donec arcu pede**, **pretium vitae**, rutrum aliquet, tincidunt blandit, pede. Aliquam in nisi. Suspendisse volutpat. Nulla facilisi. Ut ullamcorper nisi quis mi.

Nota: el código para este ejercicio se encuentra en GitHub o Moodle.

10. Ahora vamos a utilizar Bootstrap. Deberemos crear una pagina que tenga un encabezado H1 con un párrafo y un encabezado H2 con un párrafo. Tenemos que lograr que se vean con los estilos de Bootstrap.

11. Usar Bootstrap para mostrar un tabla de productos así:

Nombre	Precio
Producto1	10000
Producto2	10000
Producto3	10000

12. Una vez que tenemos esa tabla vamos a sumarle una columna más que se vea así:

Nombre	Precio	Detalle
Producto1	10000	Ver Producto
Producto2	10000	Ver Producto
Producto3	10000	Ver Producto

Para esto investigar la clase **button** de Booksrap para las etiquetas **<a>** e investigar los colores de Bootstrap y como sumarlos.

13. Un formulario para que el usuario se pueda registrar en nuestra página usando Bootstrap.

Formulario Registro

Email:

Contraseña:

Recuerdame

EJERCICIOS EXTRAS

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por último, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Para este apartado vamos a crear 2 listas:
 - 1) Lista con viñetas con aspectos personalizados a elección.
 - 2) Lista vertical con links a páginas a elección.
2. Determinar las reglas CSS necesarias para mostrar la siguiente tabla con el aspecto final mostrado en la imagen (modificar el código HTML que se considere necesario añadiendo los atributos class oportunos).

Tabla original:

The screenshot shows a table titled "Ejercicio formatear tabla - Opera" with the following data:

Cambio	Compra	Venta	Máximo	Mínimo
Euro/Dolar	1.2524	1.2527	1.2539	1.2488
Dolar/Yen	119.01	119.05	119.82	119.82
Libra/Dolar	1.8606	1.8611	1.8651	1.8522
Euro/Yen	149.09	149.13	149.79	148.96

Tabla final:

The screenshot shows the same table as above, but with specific cells highlighted in different colors:

Cambio	Compra	Venta	Máximo	Mínimo
€ Euro/Dolar	1.2524	1.2527	1.2539	1.2488
\$ Dolar/Yen	119.01	119.05	119.82	119.82
£ Libra/Dolar	1.8606	1.8611	1.8651	1.8522
¥ Yen/Euro	0.6711	0.6705	0.6676	0.6713

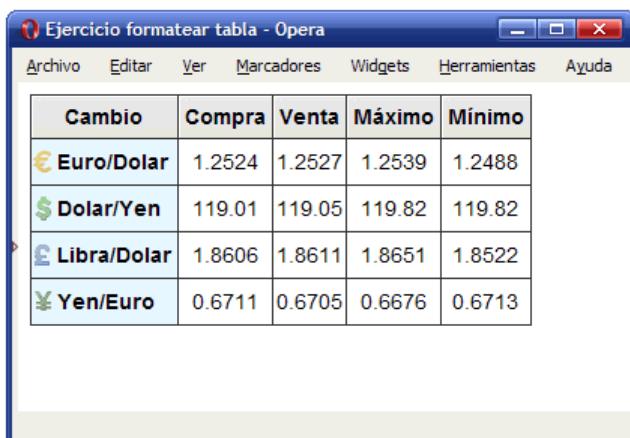
Pasos a hacer:

- 1) Alinear el texto de las celdas, cabeceras y título. Definir los bordes de la tabla, celdas y cabeceras (color gris oscuro #333).



Cambio	Compra	Venta	Máximo	Mínimo
Euro/Dolar	1.2524	1.2527	1.2539	1.2488
Dolar/Yen	119.01	119.05	119.82	119.82
Libra/Dolar	1.8606	1.8611	1.8651	1.8522
Yen/Euro	0.6711	0.6705	0.6676	0.6713

- 2) Formatear las cabeceras de fila y columna con la imagen de fondo correspondiente en cada caso (fondo_gris.gif, euro.png, dolar.png, yen.png, libra.png). Modificar el tipo de letra de la tabla y utilizar Arial. El color azul claro es #E6F3FF.



Cambio	Compra	Venta	Máximo	Mínimo
€ Euro/Dolar	1.2524	1.2527	1.2539	1.2488
\$ Dolar/Yen	119.01	119.05	119.82	119.82
£ Libra/Dolar	1.8606	1.8611	1.8651	1.8522
¥ Yen/Euro	0.6711	0.6705	0.6676	0.6713

- 3) Mostrar un color alterno en las filas de datos (color amarillo claro #FFFFCC).



Cambio	Compra	Venta	Máximo	Mínimo
€ Euro/Dolar	1.2524	1.2527	1.2539	1.2488
\$ Dolar/Yen	119.01	119.05	119.82	119.82
£ Libra/Dolar	1.8606	1.8611	1.8651	1.8522
¥ Yen/Euro	0.6711	0.6705	0.6676	0.6713

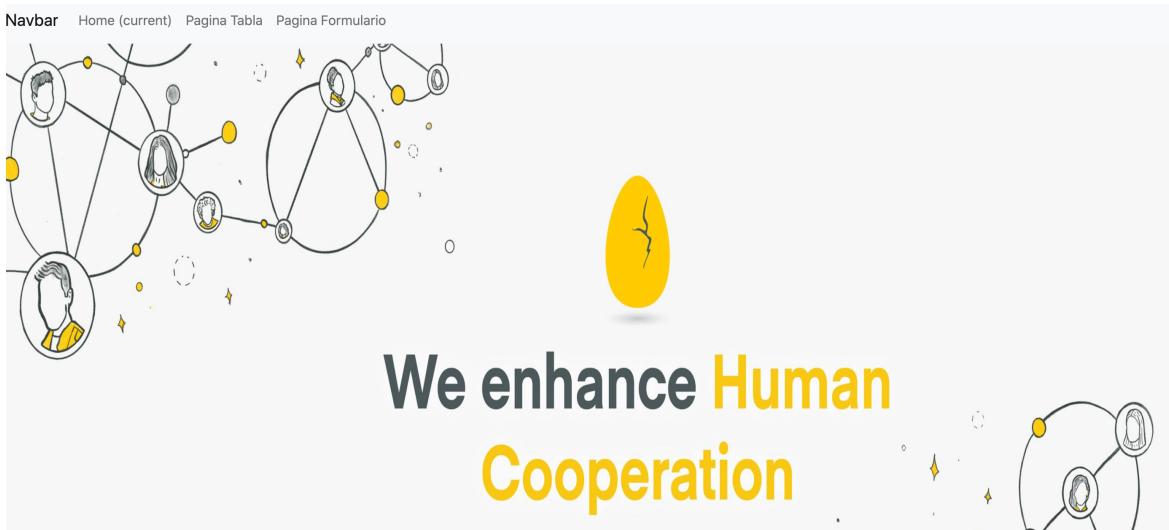
Nota: el código para este ejercicio se encuentra en GitHub o Moodle.

3. Siguiendo el ejercicio de Bootstrap, haremos una landing page que contenga un navbar, investigar el navbar en Bootstrap, con botones a todas las páginas previamente mencionadas.

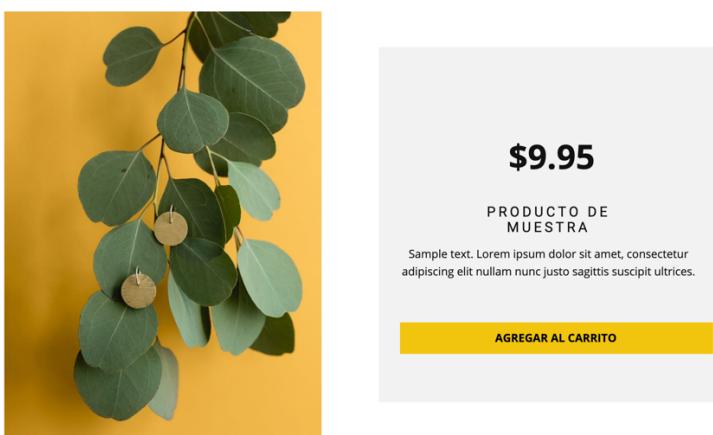
Navbar Home (current) Pagina Tabla Pagina Formulario

Pueden también hacer que ese navbar salga en el resto de las páginas.

4. Además, le sumaremos una imagen a la landing page usando las clases para la etiqueta que nos provee Bootstrap. Ejemplo:



5. Ahora deberemos hacer una página para el detalle del producto. Una pagina que muestre el nombre, el precio del producto y un botón para comprar y otro para agregar al carrito. A esta pagina se accederá con el botón en a la tabla de ver detalle.



6. Por último, sumarle a la landing cartas que muestren productos de esta manera:

Ejemplo Producto Descripción del producto Ver Producto	Ejemplo Producto Descripción del producto Ver Producto
Ejemplo Producto Descripción del producto Ver Producto	Ejemplo Producto Descripción del producto Ver Producto
Ejemplo Producto Descripción del producto Ver Producto	Ejemplo Producto Descripción del producto Ver Producto

Bibliografía

Información sacada de las páginas:

HTML:

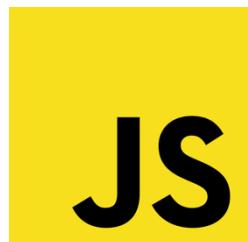
- <https://desarrolloweb.com/manuales/manual-html.html>
- https://developer.mozilla.org/es/docs/Learn/Getting_started_with_the_web/HTML_basics
- <https://www.arkaitzgarro.com/xhtml/capitulo-2.html#que-es-html>
- <https://www.geeksforgeeks.org/span-tag-html/>
- <https://www.geeksforgeeks.org/div-tag-html/>

CSS:

- <https://desarrolloweb.com/manuales/manual-css-hojas-de-estilo.html>
- <https://uniwebsidad.com/libros/css/capitulo-1>
- <https://lenguajecss.com/css/modelo-de-cajas/unidades-css/>
- https://developer.mozilla.org/es/docs/Learn/CSS/First_steps
- <https://web.dev/learn/css/>

CURSO DE PROGRAMACIÓN FULL STACK

GUIA DE JAVASCRIPT



EGG

GUIA DE JAVASCRIPT

¿QUÉ ES JAVASCRIPT?

JavaScript (abreviado comúnmente JS) es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. Desde 2012, todos los navegadores modernos soportan completamente ECMAScript 5.1, una versión de JavaScript.

Se utiliza principalmente del lado del cliente, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas y JavaScript del lado del servidor (Server-side JavaScript o SSJS). Su uso en aplicaciones externas a la web, por ejemplo en documentos PDF, aplicaciones de escritorio (mayoritariamente widgets) es también significativo.

JavaScript se diseñó con una sintaxis similar al lenguaje de programación C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo, Java y JavaScript tienen semánticas y propósitos diferentes.

Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Para interactuar con una página web se provee al lenguaje JavaScript de una implementación del *Document Object Model* (DOM).

Tradicionalmente se venía utilizando en páginas web HTML para realizar operaciones y únicamente en el marco de la aplicación cliente, sin acceso a funciones del servidor. Actualmente es ampliamente utilizado para enviar y recibir información del servidor junto con ayuda de otras tecnologías como AJAX. JavaScript se interpreta en el agente de usuario al mismo tiempo que las sentencias van descargándose junto con el código HTML.

Hay 3 formas de sumar JavaScript a nuestro HTML:

```
<!DOCTYPE html>
<html>

<head>
<script>
function miFuncion() {
  document.getElementById("demo").innerHTML = "Parrafo
cambiado!";
}
</script>
</head>
<body>

<h1>PerroMania</h1>
<p id="demo">Parrafo Inicial</p>
<button type="button" onclick="myFunction()">Intentalo</button>

</body>
</html>
```

```

<!DOCTYPE html>
<html>

<head>
</head>
<body>

<h1>PerroMania</h1>
<p id="demo">Parrafo Inicial</p>
<button type="button" onclick="myFunction()">Intentalo</button>

<script>
function miFuncion() {
    document.getElementById("demo").innerHTML = "Parrafo
cambiado!";
}
</script>

</body>
</html>

```

```

<!DOCTYPE html>
<html>
<body>

<h1>PerroMania</h1>
<p id="demo">Parrafo</p>

<button id="pinchable" type="button">Intentalo</button>

<script src="script1.js"></script>
<script src="script2.js"></script>

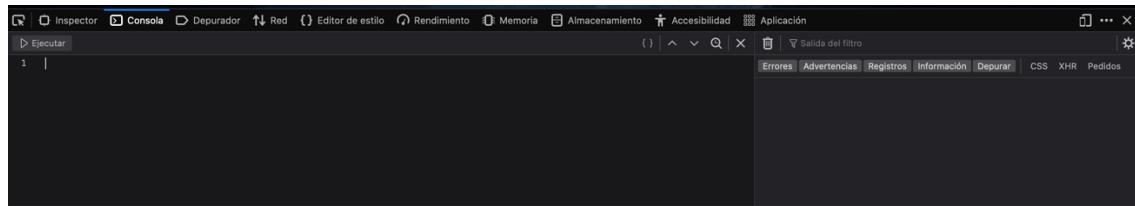
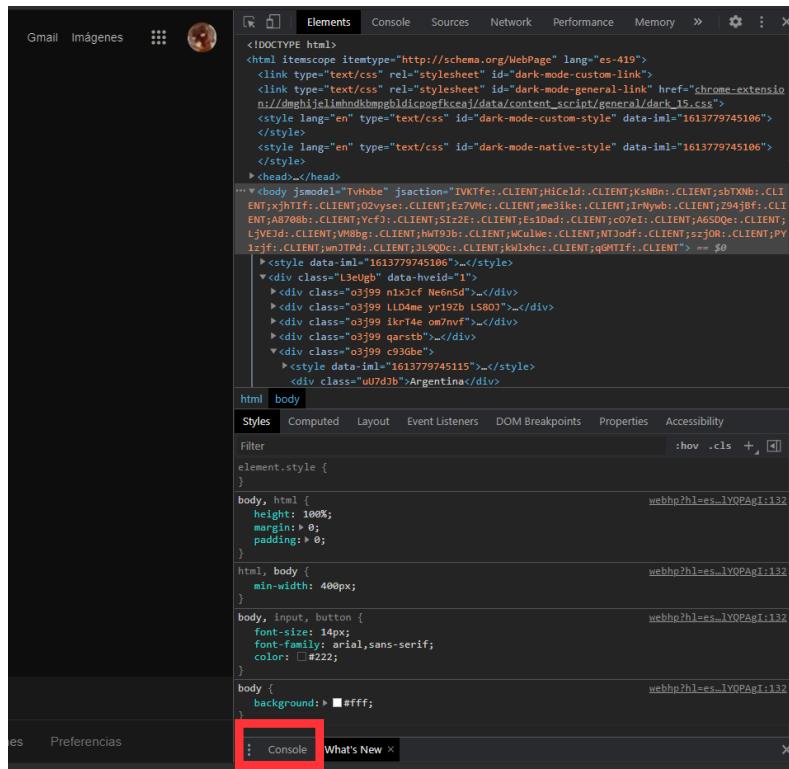
</body>
</html>

```

SALIDA Y ENTRADA DE DATOS

Una **consola web** es una herramienta que se utiliza principalmente para registrar información asociada a una página web como: solicitudes de red, JavaScript, errores de seguridad, advertencias, CSS, etc. Esta, nos permite interactuar con una página web ejecutando una expresión JavaScript en el contenido de la página.

En JavaScript, **Console**, es un objeto que proporciona acceso a la consola de depuración del navegador. Podemos abrir una consola en el navegador web Chrome, usando: Ctrl + Shift + J para Windows/Linux y Option + Command ⌘ + J para Mac.



El objeto “console” nos proporciona varios métodos diferentes, como:

- log()
- error()
- warn()
- clear()
- time() y timeEnd()
- table()
- count()
- group() y groupEnd()
- custom console logs

Método	Descripción	Ejemplo
console.log()	Se utiliza principalmente para registrar (imprimir) la salida en la consola. Podemos poner cualquier tipo dentro del log (), ya sea una cadena, matriz, objeto, booleano, etc.	<pre>console.log("abc"); console.log(123); console.log([1,2,3,4]);</pre>

console.error()	Error: se utiliza para registrar mensajes de error en la consola.	<code>console.error("Mensaje de error");</code>
console.warn	Warn: se usa para registrar mensajes de advertencia	<code>console.warn("Mensaje de advertencia");</code>
console.clear()	Se usa para limpiar la consola.	<code>console.clear();</code>
console.time() console.timeEnd()	Siempre que queramos saber la cantidad de tiempo empleado por un bloque o una función, podemos hacer uso de los métodos time() y timeEnd().	<code>console.time('abc');</code> <code>console.timeEnd('abc');</code>
console.table()	Este método nos permite generar una tabla dentro de una consola. La entrada debe ser una matriz o un objeto que se mostrará como una tabla.	<code>console.table({ 'a':1, 'b':2});</code>
console.count()	Este método se usa para contar el número que la función alcanza con este método de conteo.	<code>for(let i=0;i<5;i++){ console.count(i); }</code>

```
//console custom

const spacing = '10px';

const styles = `padding: ${spacing}; background-color: black; color: yellow; font-style: italic; border: 1px solid black; font-size: 2em;`;

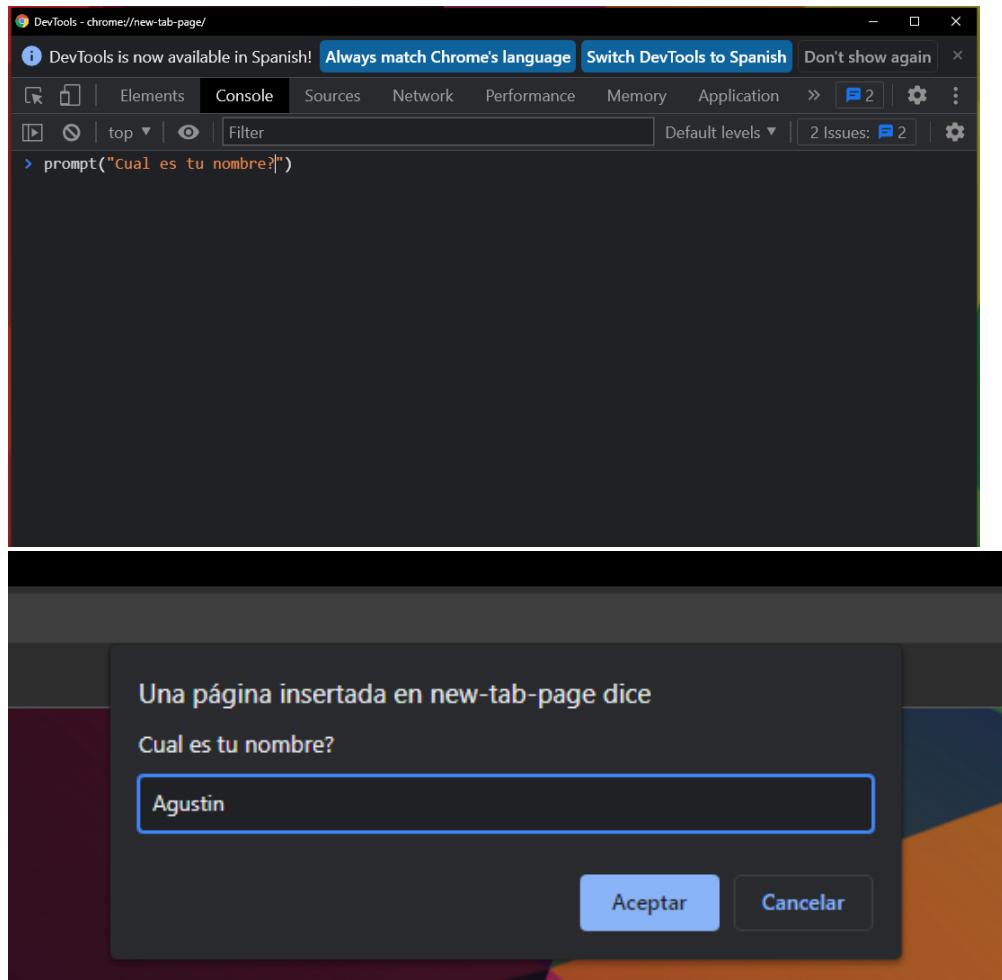
console.log('%cEGG', styles);
```

OBJETO WINDOW

El objeto window de Javascript nos sirve para controlar la ventana del navegador. Es el objeto principal en la jerarquía y contiene las propiedades y métodos para controlar la ventana del navegador. Tiene algunos métodos como:

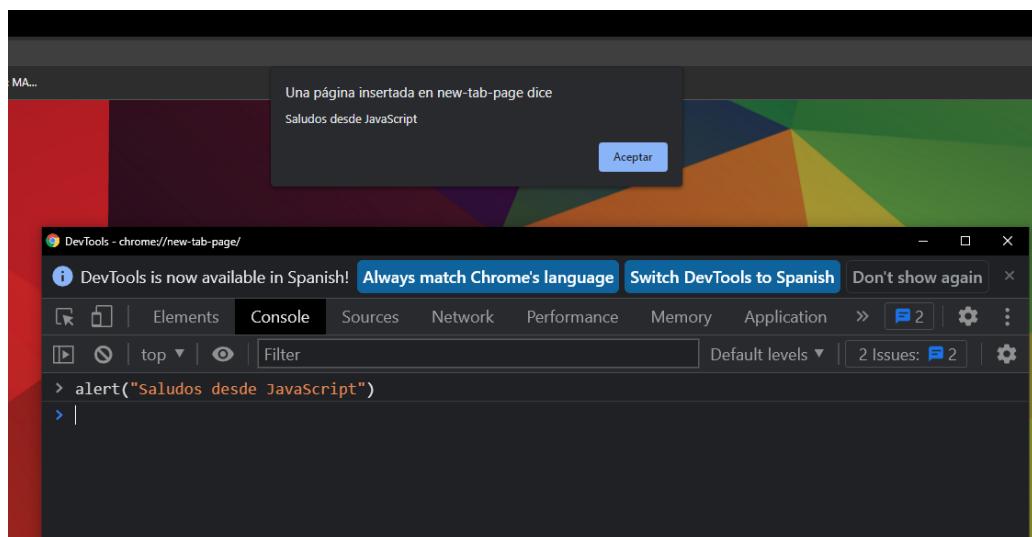
PROMPT

El método `Window.prompt()` muestra un diálogo con mensaje opcional, que solicita al usuario que introduzca un texto.



ALERT

El método `Window.alert()` muestra un diálogo de alerta con contenido opcional especificado y un botón OK (Aceptar).



VARIABLES EN JAVASCRIPT

En muchos lenguajes de programación hay unas reglas estrictas a la hora de declarar las variables, pero lo cierto es que Javascript es bastante permisivo.

Javascript se salta muchas reglas por ser un lenguaje un tanto libre a la hora de programar y uno de los casos en los que otorga un poco de libertad es a la hora de declarar las variables, ya que JavaScript, no nos obliga a declarar las variables, al contrario de lo que pasa en otros lenguajes de programación como Java, C, C# y muchos otros.

Otra cosa en la que JavaScript es más permisivo, es que, al ser de tipado suave, no tenemos que poner el tipo de dato de la variable a la hora de declararla, sino que al darle un valor, ahí toma el tipo de dato.

Aunque no es obligatorio declarar variables, recomendamos siempre declararlas antes de usarlas.

VAR

Javascript cuenta con la palabra “**var**” que utilizaremos cuando queramos declarar una o varias variables. Como es lógico, se utiliza esa palabra para definir la variable antes de utilizarla.

Esta variable se convertirá en una propiedad del objeto global, es decir sin importar donde se declare, todos tendrán la oportunidad de llamarla y utilizarla.

```
var variable;
```

También se puede asignar un valor a la variable cuando se está declarando

```
var precio1 = 5;  
var precio2 = 6;  
var total = precio1 + precio2;
```

A la hora de escribir nuestras variables mantenemos algunas buenas prácticas de Java como:

- Siempre se escriben en minúsculas
- Usamos el CamelCase cuando queremos que sean dos palabras
- Podemos usar guion bajo, para dos palabras también
- Nunca usamos la Ñ, ni tildes, ni ningún carácter especial

AMBITO DE VARIABLES

Antes de explicar la declaración de variables con la palabra let, tenemos que explicar que es el ámbito de variables

Se le llama **ámbito de las variables** al lugar donde estas están disponibles. Por lo general, cuando declaramos una variable hacemos que esté disponible en el lugar donde se ha declarado. Esto es igual a lo que hemos visto en Java, las variables globales pueden ser accedidas en cualquier lugar y las variables locales pueden ser accedidas solo donde se declararon, por ejemplo una función.

LET

Desde ECMAScript 2015 existe la declaración **let**. La sintaxis es la misma que var a la hora de declarar las variables, pero en el caso de **let** la declaración afecta al bloque.

Bloque significa cualquier espacio acotado por unas llaves, como podría ser las sentencias que hay dentro de las llaves de un bucle for.

Al declarar una variable con la palabra clave **let**, dentro de un bloque, hacemos que esa variable solo pueda ser accedida dentro de ese bloque concreto, por lo que solo existe dentro del bloque que la contiene. Lo que la haría una **variable local**.

A diferencia de la palabra clave **var**, la cual define una variable global o local en una función, sin importar el ámbito del bloque.

```
function varPrueba() {  
  var x = 31;  
  if (true) {  
    var x = 71; // Misma variable!  
    console.log(x); // Imprime el valor 71  
  }  
  console.log(x); // Imprime el valor 71  
}  
  
function letPrueba() {  
  let x = 31;  
  if (true) {  
    let x = 71; // variable diferente  
    console.log(x); // Imprime el valor 71  
  }  
  console.log(x); // Imprime el valor 31  
}
```

CONST

La sentencia const sirve para declarar una constante cuyo alcance puede ser global o local para el bloque en el que se declara. Es necesario inicializar la constante, es decir, se debe especificar su valor en la misma sentencia en la que se declara, lo que tiene sentido, dado que no se puede cambiar posteriormente.

```
const PI = 3.141592653589793;  
PI = 3.14; // Esto dará un error  
PI = PI + 10; // Esto también dará un error
```

TEMPLATE STRINGS

Las **template strings**, o **cadenas de texto de plantilla**, son una de las herramientas de ES6 para trabajo con cadenas de caracteres que nos pueden venir muy bien para producir un código Javascript más claro. Usarlas es por tanto una recomendación dado que facilitará el mantenimiento de los programas, gracias a que su lectura será más sencilla con un simple vistazo del ojo humano.

CONCATENACIÓN DE VARIABLES

En un programa realizado en JavaScript, y en cualquier lenguaje de programación en general, es normal crear cadenas en las que tenemos que juntar cadenas con los valores tomados desde las variables.

```
var sitioWeb = "DesarrolloWeb.com";  
var mensaje = 'Bienvenido a ' + sitioWeb;
```

Eso es muy fácil de leer, pero a medida que el código se complica y en una cadena tenemos que concatenar el contenido de varias variables, el código comienza a ser más enrevesado.

```
var nombre = 'Miguel Angel';  
var apellidos = 'Alvarez'  
var profesion = 'desarrollador';  
var perfil = ' ' + nombre + ' ' + apellidos + ' es ' + profesion;
```

Quizás estás acostumbrado a ver esto así. El código está bien y no tiene ningún problema, pero podría ser mucho más bonito si usas los template strings.

CREAR UN TEMPLATE STRING

Para crear un template string simplemente tienes que usar un carácter que se usa poco, como apertura y cierre de la cadena. Es el símbolo del acento grave. (`)

```
var cadena = `Esto es un template String`;
```

USOS DE LOS TEMPLATE STRINGS

Los template strings tienen varias características interesantes que, como decíamos, facilitan la sintaxis. Veremos a continuación algunos de ellos con código de ejemplo.

CONCATENACION DE VALORES

Creo que lo más interesante es el caso de la concatenación que genera un código poco claro hasta el momento. Echa un vistazo al código siguiente que haría lo mismo que el que hemos visto anteriormente del perfil.

```
var nombre = 'Miguel Angel';
var apellidos = 'Alvarez'
var profesion = 'desarrollador';
var perfil = `<b>${nombre} ${apellidos}</b> es ${profesion}`;
```

Como puedes comprobar, dentro de un template string es posible colocar expresiones encerradas entre llaves y precediendo de un símbolo "\$". Algo como \${expresion}.

En las expresiones podemos colocar código que queramos volcar, dentro de la cadena. Las usamos generalmente para colocar valores de variables, pero también servirían para colocar operaciones matemáticas, por ejemplo.

```
var suma = `45 + 832 = ${45 + 832}`;
```

O bien algo como esto:

```
var operando1 = 7;
var operando2 = 98;
var multiplicacion = `La multiplicación entre ${operando1} y ${operando2}
equivale a ${operando1 * operando2}`;
```

SALTOS DE LÍNEA DENTRO DE CADENAS

Hasta ahora, si queremos hacer una cadena con un salto de línea teníamos que usar el carácter de escape "contrabarra n".

```
var textoLargo = "esto es un texto\ncon varias líneas";
```

Con un template string tenemos la alternativa de colocar el salto de línea tal cual en el código y no producirá ningún problema.

```
var textoLargo = `esto es un texto
con varias líneas`;
```

TIPOS DE DATOS EN JAVASCRIPT

Los tipos de datos JavaScript se dividen en dos grupos: tipos primitivos y tipos objeto.

PRIMITIVOS	
Dato	Descripción.
Numérico	Números, ya sea enteros o reales.
String	Cadenas de texto.
Boolean	Valores lógicos como true o false.
null	Cuando un dato no existe.
undefined	Cuando no se le asigna un valor a la variable.

Los tipos objeto tienen sus propias subdivisiones

OBJETOS	
Tipo De Objeto	Descripción
Tipo predefinido de JavaScript	Date: fechas
Tipo predefinido de JavaScript	RegExp: expresiones regulares
Tipo predefinido de JavaScript	Error: datos de errores
Tipos definidos por el programador / usuario	Funciones Simples y Clases
Arrays	Serie de elementos o formación tipo vector o matriz. Lo consideramos un objeto especial

Objetos Especiales	Objeto Global
Objetos Especiales	Objeto prototipo
Objetos Especiales	Otros

OPERADORES EN JAVASCRIPT

Al igual que Java tendremos operadores para trabajar con datos, aunque hay algunos operadores que son distintos a los que conocemos en Java.

OPERADOR DE ASIGNACIÓN

= Operador de Asignación Simple

OPERADORES ARITMÉTICOS

+ Operador de Suma

- Operador de Resta

* Operador de Multiplicación

** Exponenciación

/ Operador de División

% Operador de Módulo

OPERADORES UNARIOS

++ Operador de Incremento.

-- Operador de Decremento.

+= y += x

`-=` $y -= x$

`*=` $y *= x$

`/=` $y /= x$

`%=` $y \%= x$

`**=` $y **= x$

OPERADORES LOGICOS Y RELACIONALES

`==` Es igual Ejemplo: `3 == "3"`

`==>` Es estrictamente igual Ejemplo: `3 ==> 3`

`!=` Distinto

`!==` Estrictamente Distinto

`>` Mayor que

`>=` Mayor o igual que

`<` Menor que

`<=` Menor o igual que

OPERADORES CONDICIONALES

`&&` AND

`||` OR

`!` Operador Lógico de Negación.

OPERADORES DE COMPARACIÓN DE TIPO

typeof	Devuelve el tipo de dato de una variable
Instanceof	Devuelve true si el objeto es una instancia de.

TYPEOF

La función typeof se utiliza para obtener el tipo de dato que tiene una variable.

```
console.log(typeof 42);
// expected output: "number"

console.log(typeof 'blubber');
// expected output: "string"

console.log(typeof true);
// expected output: "boolean"
```

CONDICIONALES EN JAVASCRIPT

Al igual que en Java, existen los condicionales que nos van a ayudar a modificar el flujo de ejecución del programa.

IF

El condicional if es un condicional lógico que evalúa el camino a tomar en base a la evaluación de una condición. Supongamos el siguiente ejemplo, mi sobrino quiere subirse a una montaña rusa, pero para ello tiene que aprobar las dos siguientes condiciones: tener más de 18 años y medir más de 160 cm. La evaluación de esas dos condiciones da por verdadero se podrá subir de lo contrario no podrá.

```
let edad = 15;
let altura = 166;
if(edad>18 && altura>160){
    console.log("Puedes subirte :D");
} else{
    console.log("No te puedes subir");
}
```

Como se puede ver, si la condición a evaluar se cumple, es decir, da verdadero, mostrará el mensaje "Puedes subirte :D", en caso que sea falso mostrará "No te puedes subir". Por otra parte, JavaScript permite también agregar la condición else if

```
if(a == 2){  
    console.log("a es igual a 2");  
}else if(a < 2){  
    console.log("a es menor que 2");  
}else{  
    console.log("a es mayor que 2");  
}
```

IF TERNARIO

El if ternario nos permite resolver en una línea una expresión lógica asignando un valor. Proviene del lenguaje C, donde se escriben muy pocas líneas de código y donde cuanto menos escribamos más elegantes seremos. Este operador es un claro ejemplo de ahorro de líneas y caracteres al escribir los scripts. Lo veremos rápidamente, pues la única razón que lo veamos es para que sepan que existe y si lo encuentran en alguna ocasión sepan identificarlo y cómo funciona.

```
Variable = (condición) ? valor1 : valor2
```

Este ejemplo no sólo realiza una comparación de valores, además asigna un valor a una variable. Lo que hace es evaluar la condición (colocada entre paréntesis) y si es positiva asigna el valor1 a la variable y en caso contrario le asigna el valor2. Veamos un ejemplo:

```
momento = (hora_actual < 12) ? "Antes del mediodía" : "Después del  
mediódia"
```

SWITCH

La declaración **switch** evalúa una expresión, comparando el valor de esa expresión con una instancia **case**, y ejecuta declaraciones asociadas a ese case, así como las declaraciones en los case que siguen.

El programa primero busca la primer instancia case cuya expresión se evalúa con el mismo valor de la expresión de entrada (usando comparación estricta, `==`) y luego transfiere el control a esa cláusula, ejecutando las declaraciones asociadas. Si no se encuentra una cláusula de case coincidente, el programa busca la cláusula **default opcional**, y si se encuentra, transfiere el control a esa instancia, ejecutando las declaraciones asociadas.

Al igual que Java, la declaración **break** es opcional y está asociada con cada etiqueta de case y asegura que el programa salga del switch una vez que se ejecute la instrucción coincidente y continúe la ejecución en la instrucción siguiente. Si se omite el `break` el programa continúa la ejecución en la siguiente instrucción en la declaración de switch .

```

switch (expr) {
    case 'Naranjas':
        console.log('El kilogramo de naranjas cuesta $0.59.');
        break;
    case 'Mangos':
    case 'Papayas':
        console.log('El kilogramo de mangos y papayas cuesta $2.79.');
        break;
    default:
        console.log('Lo lamentamos, por el momento no disponemos de ' + expr +
        '.');
}

```

ESTRUCTURAS REPETITIVAS

Veamos como son las estructuras repetitivas en JavaScript

WHILE

Crea un bucle que ejecuta una sentencia especificada mientras cierta condición se evalúe como verdadera. Dicha condición es evaluada antes de ejecutar la sentencia

```

let a = 0;
while(a != 10){
    console.log(++a);
}

```

DO WHILE

La sentencia (hacer mientras) crea un bucle que ejecuta una sentencia especificada, hasta que la condición de comprobación se evalúa como falsa. La condición se evalúa después de ejecutar la sentencia, dando como resultado que la sentencia especificada se ejecute al menos una vez.

```

let a = 0;
do{
    console.log(++a);
}while(a!=10);

```

FOR

El bucle FOR se utiliza para repetir una o más instrucciones un determinado número de veces. De entre todos los bucles, el FOR se suele utilizar cuando sabemos seguro el número de veces que queremos que se ejecute. La sintaxis del bucle for va a ser la misma que en Java:

```
for ([expresion-inicial]; [condicion]; [expresion-final]){
}
for(let i = 0; i < 10; i++){
    console.log("El valor de i es " + i);
}
```

BREAK

Termina el bucle actual, sentencia switch o label y transfiere el control del programa a la siguiente sentencia.

```
for (let i = 0; i < 10; i++) {
    if(i == 5){
        break;
    }
    console.log("Estamos por la vuelta "+i);
}
```

CONTINUE

Termina la ejecución de las sentencias de la iteración actual del bucle actual o la etiqueta y continua la ejecución del bucle con la próxima iteración.

En contraste con la sentencia break, continue no termina la ejecución del bucle por completo; en cambio,

- En un bucle while, salta de regreso a la condición.
- En un bucle for, salta a la expresión actualizada.

La sentencia continue puede incluir una etiqueta opcional que permite al programa saltar a la siguiente iteración del bucle etiquetado en vez del bucle actual. En este caso, la sentencia continue necesita estar anidada dentro de esta sentencia etiquetada.

```
for (let i = 0; i < 10; i++) {
    if(i == 5){
        continue;
    }
    console.log("Estamos por la vuelta " + i);
}
```

LABEL

Proporciona a una sentencia con un identificador al que se puede referir al usar las sentencias break o continue. Por ejemplo, puede usar una etiqueta para identificar un bucle, y entonces usar las sentencias break o continue para indicar si un programa debería interrumpir el bucle o continuar su ejecución.

`label o etiqueta : sentencia`

Ejemplo:

```
exterior: for (let i = 0; i < 10; i++) {  
    for (let j = 0; j < 10; j++) {  
        if(i == 4 && j == 4){  
            console.log("Vamos a cortar ambos for");  
            break exterior;  
        }  
        console.log(i+j+10*i);  
    }  
}
```

FUNCIONES EN JAVASCRIPT

Una función, en JavaScript es, al igual que Java, como una serie de instrucciones que englobamos dentro de un mismo proceso. Este proceso se podrá luego ejecutar desde cualquier otro sitio con solo llamarlo. Por ejemplo, en una página web puede haber una función para cambiar el color del fondo y desde cualquier punto de la página podríamos llamarla para que nos cambie el color cuando lo deseemos.

La sintaxis de una función en JavaScript es la siguiente:

```
function nomrefuncion (){  
    instrucciones de la función  
    ...  
}
```

Primero se escribe la palabra **function**, reservada para este uso. Seguidamente se escribe el nombre de la función, que como los nombres de variables puede tener números, letras y algún carácter adicional como en guión bajo y los paréntesis donde irán nuestros parámetros. A continuación se colocan entre llaves las distintas instrucciones de la función. Las llaves en el caso de las funciones no son opcionales, además es útil colocarlas siempre como se ve en el ejemplo, para que se reconozca fácilmente la estructura de instrucciones que engloba la función.

Un ejemplo de función que escribe hola mundo en la consola sería:

```
function holaMundo(){  
    console.log('Hola Mundo');  
}
```

Después para llamar a esta función, hay que invocarla mediante su nombre

```
holaMundo();
```

DÓNDE COLOCAMOS LAS FUNCIONES JAVASCRIPT

En principio, podemos colocar las funciones en cualquier parte de la página, siempre entre etiquetas `<script>`. No obstante existe una limitación a la hora de colocarla con relación a los lugares desde donde se la llame. Usualmente lo más fácil es colocar la función antes de cualquier llamada a la misma y así seguro que nunca nos equivocaremos.

Existen dos opciones posibles para colocar el código de una función:

- Colocar la función en el mismo bloque de script: En concreto, la función se puede definir en el bloque `<script>` donde esté la llamada a la función, aunque es indiferente si la llamada se encuentra antes o después del código de la función, dentro del mismo bloque `<script>`.

```
<script>
miFuncion();
function miFuncion(){
    //hago algo...
    console.log("Esto va bien");
}
</script>
```

- Colocar la función en otro bloque de script: También es válido que la función se encuentre en un bloque `<SCRIPT>` anterior al bloque donde está la llamada.

```
<html>
<head>
    <title>MI PÁGINA</title>
<script>
function miFuncion(){
    //hago algo...
    console.log("Esto va bien");
}
</script>
</head>
<body>
<script>
miFuncion()
</script>
</body>
</html>
```

PARAMETROS DE LAS FUNCIONES

Los parámetros se usan para mandar valores a las funciones. Una función trabajará con los parámetros para realizar las acciones. Por decirlo de otra manera, los parámetros son los valores de entrada que recibe una función.

Los parámetros en JavaScript, son iguales que en Java, la única diferencia es que, a diferencia de Java, no tenemos que especificar el tipo de dato que recibe la función, sino que va a ser el tipo de dato que enviamos como parámetro.

```
function escribirBienvenida(nombre){  
    console.log('Hola ' + nombre);  
}  
  
escribirBienvenida('Agustín');  
// O podemos hacerlo con una variable  
let nombre = 'Agustín';  
escribirBienvenida(nombre);
```

DEVOLVER VALORES EN FUNCIONES

Las funciones en Javascript también pueden **retornar valores**. De hecho, ésta es una de las utilidades más esenciales de las funciones.

Veamos un ejemplo de función que calcula la media de dos números. La función recibirá los dos números y retornará el valor de la media.

```
function media(valor1,valor2){  
    let resultado;  
    resultado = (valor1 + valor2) / 2;  
    return resultado;  
}
```

Para especificar el valor que retornará la función se utiliza la palabra **return** seguida de el valor que se desea devolver. En este caso se devuelve el contenido de la variable resultado, que contiene la media calculada de los dos números.

Pero, cómo podemos recibir un dato que devuelve una función. Cuando una función devuelve un valor simplemente se sustituye la llamada a la función por ese valor que devuelve. Así pues, para almacenar un valor de devolución de una función, tenemos que asignar la llamada a esa función como contenido en una variable, y eso lo haríamos con el operador de asignación **=**.

```
let miMedia  
miMedia = media(12,8);  
console.log(miMedia);
```

FUNCIONES FLECHA

Hay otra sintaxis muy simple y concisa para crear funciones, que a menudo es mejor que las Expresiones de funciones.

Se llama “funciones de flecha”, porque se ve así:

```
let func = (arg1, arg2, ..., argN) => expresion
```

Esto crea una función **func** (nombre de la función) que acepta parámetros arg1..argN, luego evalúa la **expresión** de la derecha con su uso y devuelve su resultado.

En otras palabras, es la versión más corta de:

```
let func = function(arg1, arg2, ..., argN) {  
    return expresion;  
};
```

Veamos un ejemplo completo:

```
let sum = (a, b) => a + b;  
/* Esta función de flecha es una forma más corta de:  
let sum = function(a, b) {  
    return a + b;  
};  
*/  
console.log(sum(1, 2)); // 3
```

Como puedes ver (a, b) => a + b significa una función que acepta dos parámetros llamados a y b. Tras la ejecución, evalúa la expresión a + b y devuelve el resultado.

Si solo tenemos un argumento, se pueden omitir paréntesis alrededor de los parámetros, lo que lo hace aún más corto.

```
let double = n => n * 2;  
// Más o menos lo mismo que: let double = function(n) { return n * 2 }  
console.log(double(3)); // 6
```

Si no hay parámetros, los paréntesis estarán vacíos (pero deben estar presentes):

```
let saludar = () => console.log("Hola!");  
saludar();
```

Este tema lo hemos visto para que sepan que existe y si lo encuentran en alguna ocasión sepan identificarlo y cómo funciona. Recomendamos que antes de pasar a las funciones flecha, aprendamos a trabajar las funciones “normales”.

OBJETOS

Vamos a introducirnos en un tema muy importante de Javascript como son los objetos. Javascript no es un lenguaje de programación orientado a objetos **puro** porque, aunque utiliza objetos en muchas ocasiones, no necesitamos programar todos nuestros programas en base a ellos. De hecho, lo que vamos a hacer generalmente con Javascript es **usar objetos** y no tanto programar orientado a objetos. Por ello, lo que hemos visto hasta aquí relativo a sintaxis, sigue siendo perfectamente válido y puede ser utilizado igual que se ha indicado. Solo vamos a **aprender** una especie de estructura nueva como son los **objetos**.

Un objeto es una colección de propiedades y una propiedad es una asociación entre un nombre (o clave) y un valor. El valor de una propiedad puede ser una función, en cuyo caso la propiedad se conoce como método.

Las claves de un objeto, solo pueden ser de tipo String, no podemos ponerle de nombre a una propiedad un numero.

En lenguajes de programación orientados a objetos puros, como puede ser Java, tienes que programar siempre en **base a objetos**. Para programar tendrías que crear "clases", que son a partir de los cuales se crean objetos. El programa resolvería cualquier necesidad mediante la creación de objetos en base de nuestras clases, existiendo varios objetos de diversas clases. Los objetos tendrían que colaborar entre si para resolver cualquier tipo de acción, igual que en sistemas como un avión existen diversos objetos (el motor, hélices, mandos...) que colaboran entre sí para resolver la necesidad de llevar pasajeros.

Sin embargo, como dijimos previamente, en JavaScript no es tanto programar orientado a objetos, **sino usar objetos**. Muchas veces serán objetos ya creados por el propio navegador (la ventana del navegador, una imagen o un formulario HTML, etc), y otras veces serán objetos creados por ti mismo o por otros desarrolladores. Por tanto, lo que nos interesa saber para comenzar es la sintaxis que necesitas para usar los objetos, básicamente acceder a sus propiedades y ejecutar sus métodos.

OBJETOS Y PROPIEDADES

Una propiedad de un objeto se puede explicar como una **variable adjunta al objeto**. Las propiedades de un objeto básicamente son lo mismo que las variables comunes de JavaScript, excepto por el nexo con el objeto. Las propiedades de un objeto definen las características del objeto. Accedes a las propiedades de un objeto con una simple notación de puntos:

nombreObjeto.nombrePropiedad;

Como todas las variables de JavaScript, tanto el nombre del objeto (que puede ser una variable normal) como el nombre de la propiedad son **sensibles a mayúsculas y minúsculas**. Puedes definir propiedades asignándoles un valor. Por ejemplo, vamos a crear un objeto llamado miAuto y le vamos a asignar propiedades denominadas marca, modelo, y año de la siguiente manera:

```
var miAuto = new Object();
miAuto.marca = 'Ford';
miAuto.modelo = 'Mustang';
miAuto.anio = 1969;
```

El ejemplo anterior también se podría escribir usando un **iniciador de objeto**, que es una lista delimitada por comas de cero o más pares de nombres de propiedad y valores asociados de un objeto, encerrados entre llaves ({}):

```
var miAuto = {
  marca : 'Ford',
  modelo : 'Mustang',
  anio : 1969
}
```

Las propiedades se separan por comas y se coloca siempre el nombre de la propiedad, el carácter ":" y luego el valor de la propiedad.

Las propiedades no asignadas de un objeto son **undefined** (y no null).

```
miAuto.color; // undefined
```

USAR UNA FUNCIÓN CONSTRUCTORA

Como alternativa, puedes crear un objeto con estos dos pasos:

1. Definir el tipo de objeto escribiendo una **función constructora**. Existe una fuerte convención, con buena razón, para utilizar en mayúscula la letra inicial.
2. Crear una instancia del objeto con el operador **new**.

Para definir un tipo de objeto, crea una **función para el objeto** que especifique su nombre, propiedades y métodos. Por ejemplo, supongamos que deseas crear un tipo de objeto para autos. Quieres llamar Auto a este tipo de objeto, y deseas que tenga las siguientes propiedades: marca, modelo y año. Para ello, podrías escribir la siguiente función:

```
function Auto(marca,modelo,anio){
  this.marca = marca;
  this.modelo = modelo;
  this.anio = anio;
}
```

Observa el uso de **this** para asignar valores a las propiedades del objeto en función de los valores pasados a la función.

Ahora puedes crear un objeto llamado **miAuto** de la siguiente manera:

```
var miAuto = new Auto('Ford', 'Mustang', 1969);
```

Esta declaración crea `miAuto` y le asigna los valores especificados a sus propiedades. Entonces el valor de `miAuto.marca` es la cadena "Ford", para `miAuto.anio` es el número entero 1969, y así sucesivamente.

Puedes crear cualquier número de objetos Auto con las llamadas a new. Por ejemplo,

```
var auto1 = new Auto('Nissan', '300ZX', 1992);
var auto2 = new Auto('Mazda', 'Miata', 1990);
```

Un objeto puede tener una propiedad que en sí misma es otro objeto. Por ejemplo, supongamos que defines un objeto llamado `Persona` de la siguiente manera:

```
function Persona(nombre, edad, sexo) {
    this.nombre = nombre;
    this.edad = edad;
    this.sexo = sexo;
}
```

y luego instancias dos nuevos objetos persona de la siguiente manera:

```
var agus = new Persona('Agustina Gomez', 33, 'F');
var valen = new Persona('Valentin Perez', 39, 'M');
```

Entonces, puedes volver a escribir la definición de `Auto` para incluir una propiedad `propietario` que tomará el objeto persona, de la siguiente manera:

```
function Auto(marca, modelo, anio, propietario){
    this.marca = marca;
    this.modelo = modelo;
    this.anio = anio;
    this.propietario = propietario;
}
```

Para crear instancias de los nuevos objetos, utiliza lo siguiente:

```
var auto1 = new Auto('Nissan', '300ZX', 1992, agus);
var auto2 = new Auto('Mazda', 'Miata', 1990, valen);
```

Nota que en lugar de pasar un valor de cadena o entero cuando se crean los nuevos objetos, las declaraciones anteriores pasan al objetos `agus` y `valen` como **argumentos** para los propietarios. Si luego quieras averiguar el nombre del propietario del `auto2`, puedes **acceder a la propiedad** de la siguiente manera:

```
auto2.propietario.nombre;
```

Ten en cuenta que siempre se puede añadir una propiedad a un objeto previamente definido. Por ejemplo, la declaración:

```
auto1.color = 'negro';
```

DEFINICION DE METODOS

Un método es una función asociada a un objeto, o, simplemente, un método es una propiedad de un objeto que es una función. Los métodos se definen normalmente como una función, con excepción de que tienen que ser asignados como la propiedad de un objeto. Un ejemplo puede ser:

```
nombreObjeto.nombreMetodo = nombreFuncion;
```

Ejemplo:

```
var miObj = {  
    miMetodo: function(parametros) {  
        // ...hacer algo  
    }  
};
```

Entonces puedes llamar al método en el contexto del objeto de la siguiente manera:

```
objeto.nombreMetodo(parametros);
```

Puedes definir métodos para un tipo de objeto incluyendo una definición del método en la función constructora del objeto. Podrías definir una función que muestre las propiedades de los objetos del tipo Car previamente definidas, por ejemplo:

```
function mostrarAuto() {  
    var resultado = `Un hermoso ${this.year} ${this.make} ${this.model}`;  
    console.log(resultado);  
}
```

Puedes hacer de esta función un método de Car agregando la declaración a la definición del objeto:

```
this.mostrarAuto = mostrarAuto;
```

Por lo tanto, la definición completa de Car ahora se verá así:

```
function Auto(marca,modelo,anio, propietario){  
    this.marca = marca;  
    this.modelo = modelo;  
    this.anio = anio;  
    this.propietario = propietario;  
    this.mostrarAuto = mostrarAuto;  
}
```

Entonces, podemos llamar al método mostrarAuto, de la siguiente manera:

```
auto1.mostrarAuto;  
auto2.mostrarAuto;
```

OBJETOS INCORPORADOS EN JAVASCRIPT

JavaScript define algunos objetos de forma nativa, por lo que pueden ser utilizados directamente por las aplicaciones sin tener que declararlos. Las clases que se encuentran disponibles de manera nativa en Javascript, y que vamos a ver a continuación, son las siguientes:

- **String**, para el trabajo con cadenas de caracteres.
- **Date**, para el trabajo con fechas.
- **Math**, para realizar funciones matemáticas.
- **Number**, para realizar algunas cosas con números
- **Boolean**, trabajo con booleanos.
- **Array**, trabajo con listas

STRING

En JavaScript las variables de tipo texto son objetos de la clase String. Esto quiere decir que cada una de las variables que creamos de tipo texto tienen una serie de propiedades y métodos. Para crear un objeto de la clase String lo único que hay que hacer es asignar un texto a una variable.

PROPIEDADES DE STRING

Length

La clase String sólo tiene una propiedad: length, que guarda el número de caracteres del String. Por ejemplo:

```
let cadena = "Hola";  
console.log(cadena.length); // 3
```

MÉTODOS DE STRING

Los objetos de la clase String tienen una buena cantidad de métodos para realizar muchas cosas interesantes.

Método	Descripción.
charAt(indice)	Devuelve el carácter que hay en la posición indicada como índice. Las posiciones de un String empiezan en 0.
toString()	Este método lo tienen todos los objetos y se usa para convertirlos en cadenas.

<code>indexOf(carácter,desde)</code>	Devuelve la posición de la primera vez que aparece el carácter indicado por parámetro en un String. Si no encuentra el carácter en el String devuelve -1. El segundo parámetro es opcional y sirve para indicar a partir de qué posición se desea que empiece la búsqueda.
<code>lastIndexOf(carácter,desde)</code>	Busca la posición de un carácter exáctamente igual a como lo hace la función indexOf pero desde el final en lugar del principio. El segundo parámetro indica el número de caracteres desde donde se busca, igual que en indexOf.
<code>toLowerCase()</code>	Pone todas los caracteres de un string en minúsculas.
<code>toUpperCase()</code>	Pone todas los caracteres de un string en mayúsculas.
<code>replace(substring_a_buscar,nuevoStr)</code>	Sirve para reemplazar porciones del texto de un string por otro texto, por ejemplo, podríamos utilizarlo para reemplazar todas las apariciones del substring "xxx" por "yyy". El método no reemplaza en el string, sino que devuelve un resultante de hacer ese reemplazo.
<code>substring(inicio,fin)</code>	Devuelve el substring que empieza en el carácter de inicio y termina en el carácter de fin. Si intercambiamos los parámetros de inicio y fin también funciona.

MATH

La clase **Math** es una de las clases nativas de JavaScript. Proporciona los mecanismos para realizar operaciones matemáticas en JavaScript. Algunas operaciones se resuelven rápidamente con los operadores aritméticos que ya conocemos, como la multiplicación o la suma, pero hay una serie de operaciones matemáticas adicionales que se tienen que realizar usando la clase Math como pueden ser calcular un seno o hacer una raíz cuadrada.

MÉTODOS DE MATH

Tenemos una serie de métodos para realizar operaciones matemáticas típicas, aunque un poco complejas. Para utilizar los métodos de la clase Math, la sintaxis será la siguiente:

`Math.metodo;`

Método	Descripción.
<code>abs()</code>	Devuelve el valor absoluto de un número. El valor después de quitarle el signo.
<code>ceil()</code>	Devuelve el entero igual o inmediatamente siguiente de un número. Por ejemplo, <code>ceil(3)</code> vale 3, <code>ceil(3.4)</code> es 4.
<code>exp()</code>	Retorna el resultado de elevar el número E por un número.
<code>floor()</code>	Lo contrario de <code>ceil()</code> , pues devuelve un número igual o inmediatamente inferior.
<code>max()</code>	Retorna el mayor de 2 números.
<code>min()</code>	Retorna el menor de 2 números.
<code>pow()</code>	Recibe dos números como parámetros y devuelve el primer número elevado al segundo número.
<code>random()</code>	Devuelve un número aleatorio entre 0 y 1.
<code>round()</code>	Redondea al entero más próximo.
<code>PI</code>	No es método, es una propiedad, que nos permite tener el valor de PI

DATE

Sobre la clase **Date** recae todo el trabajo con fechas en JavaScript, como obtener una **fecha**, el **día**, la **hora actuales** y otras cosas. Para trabajar con fechas necesitamos instanciar un objeto de la **clase Date** y con él ya podemos realizar las operaciones que necesitemos.

Un objeto de la clase Date se puede crear de dos maneras distintas. Por un lado podemos crear el objeto con el día y hora actuales y por otro podemos crearlo con un día y hora distintos a los actuales. Esto depende de los parámetros que pasemos al construir los objetos.

Para crear un objeto fecha con el día y hora actuales colocamos los paréntesis vacíos al llamar al constructor de la clase Date.

```
miFecha = new Date()
```

Para crear un objeto fecha con un día y hora distintos de los actuales tenemos que indicar entre paréntesis el momento con que inicializar el objeto.

```
miFecha = new Date(año,mes,dia)
```

Los objetos de la clase Date no tienen propiedades pero si un montón de métodos, que vamos a detallar a continuación.

Método	Descripción.
getDate()	Devuelve el día del mes.
getDay()	Devuelve el día de la semana.
getHours()	Retorna la hora.
getMinutes()	Devuelve los minutos.
getMonth()	Devuelve el mes (el mes que empieza en 0).
getFullYear()	Retorna el año con todos los dígitos.
setDate()	Actualiza el día del mes.
setMonth()	Cambia el mes (el mes empieza en 0).
setHours()	Actualiza la hora.
setMinutes()	Cambia los minutos.
setFullYear()	Cambia el año de la fecha al número que recibe por parámetro. El número se indica completo ej: 2005 o 1995.

ARRAYS

El objeto **Array** de JavaScript es un objeto global que es usado en la construcción de arrays. Los **arrays** son objetos similares a una lista cuyo prototipo proporciona métodos para efectuar operaciones de recorrido y de mutación. Tanto la longitud como el tipo de los elementos de un **array** son variables. Dado que la longitud de un array puede cambiar en cualquier momento, y los datos se pueden almacenar en ubicaciones no contiguas, no hay garantía de que los arrays de JavaScript sean densos; esto depende de cómo el programador elija usarlos. En general estas características son cómodas, pero si, en su caso particular, no resultan deseables, puede considerar el uso de arrays con tipo.

DECLARACIÓN

Hay dos sintaxis para crear un array vacío:

```
let arr = new Array();  
let arr = [];
```

Casi siempre se usa la segunda. Podemos suministrar elementos iniciales entre los corchetes:

```
let frutas = ["Manzana", "Naranja", "Uva"];
```

Los elementos del array están numerados comenzando desde cero.

Podemos obtener un elemento por su número entre corchetes:

```
let frutas = ["Manzana", "Naranja", "Uva"];  
console.log(frutas[0]);  
console.log(frutas[1]);  
console.log(frutas[2]);
```

Podemos reemplazar un elemento:

```
frutas[2] = "Pera" // ["Manzana", "Naranja", "Pera"];
```

...o agregar uno nuevo al array:

```
frutas[3] = "Limon" // ["Manzana", "Naranja", "Pera", "Limon"];
```

Un array puede almacenar elementos de cualquier tipo.

```
let frutas = ["Manzana", 1, true, function()];
```

La cuenta total de elementos en el array es su longitud `length`:

```
let frutas = ["Manzana", "Naranja", "Uva"];  
console.log(frutas.length); // 3
```

BUCLAS

Una de las formas más viejas de iterar los items de un array es el bucle for sobre sus índices:

```
let frutas = ["Manzana", "Naranja", "Uva"];
for (let i = 0; i < frutas.length; i++) {
    console.log( frutas[i] );
}
```

Pero existen otros bucles, para iterar Arrays que nos van a servir para distintas situaciones y es importante conocerlos todos.

.FOREACH

El **for each** es un método incluido dentro de los datos del tipo array. Este método nos permite recorrer el array de principio a fin y ejecutar una función o sentencia sobre cada elemento dentro del array. La sintaxis es la siguiente:

```
array.forEach(function(valorActual, indice, array){}, thisValor);
```

Los parámetros con los que se manejará el método son los siguientes:

- **function (callback)**: Función a ejecutar por cada elemento, que recibe tres argumentos:
 - **valorActual**: El elemento actual siendo procesado en el array.
 - **index (opcional)**: El índice del elemento actual siendo procesado en el array.
 - **array (opcional)**: El vector en el que forEach() está siendo aplicado.
- **thisValor (opcional)**: Valor que se usará como this cuando se ejecute el callback.

forEach() ejecuta la función callback una vez por cada elemento presente en el array en orden ascendente.

IMPRIMIR EL CONTENIDO DE UN ARRAY

Veamos un forEach que nos va a servir para mostrar los elementos de un array.

```
function mostrarElementosArray(elemento, indice, array) {
    console.log("a[" + indice + "] = " + elemento);
}

// Nótese que se evita el 2º índice ya que no hay ningún elemento en esa
// posición del array

[2, 5, , 9].forEach(mostrarElementosArray);

// salida:
// a[0] = 2
// a[1] = 5
// a[2] = 9
```

Es este ejemplo hacemos la función por separado y después se la pasamos al forEach, veamos como sería todo juntos

```
let array = [2, 5, 9].  
array.forEach(function mostrarElementosArray(elemento, indice, array) {  
    console.log("a[" + indice + "] = " + elemento);  
});  
// salida:  
// a[0] = 2  
// a[1] = 5  
// a[2] = 9
```

FOR OF

El **for of** es un bucle que itera sobre un elemento, como por ejemplo un array, desde su inicio a fin, la particularidad del for of es que tomara cada uno de los elementos, y los almacenará en una variable temporal, nosotros usaremos esa variable temporal, por ejemplo para mostrar todos los elementos de nuestro array.

```
for (variable of objeto) {  
    // sentencias  
}
```

En el siguiente ejemplo que veremos, tendremos un array de frutas y los almacenará en la variable temporal “fruta”, posterior a esto podremos hacer lo que se desee.

```
let frutas = ["Manzana", "Naranja", "Uva"];  
for(let fruta of frutas){  
    console.log(fruta); // ["Manzana", "Naranja", "Uva"];  
}
```

Ahora supongamos que tenemos un array de objetos de empleados:

```
for (let empleado of empleados){  
    console.log(empleado.nombre);  
    console.log(empleado.apellido);  
}
```

Nosotros usamos la variable temporal para acceder a cada propiedad de los objetos dentro del array.

FOR IN

El **for in** a diferencia del **for of** iterara sobre los elementos dentro de un dato. Por ejemplo, si utilizamos un **for in** sobre un array, lo que hara será darnos los índices de los elementos dentro del array, pero si usamos un **for in sobre un objeto**, nos dara cada uno de los atributos dentro del mismo. En el siguiente ejemplo, lo que se hace es usar un **for of** para recorrer la lista de empleados, y luego se toma un empleado con el **for in** para recorrer cada una de las propiedades del empleado.

```
for (let empleado of empleados){  
    for(let dato in empleado){  
        console.log(empleado[dato]);  
    }  
}
```

Por lo que este bucle, por más que podríamos usarlo para arrays, está mayormente pensado para recorrer objetos.

METODOS OBJETO ARRAY

El objeto Array de JavaScript cuenta con muchos métodos. Para hacer las cosas más sencillas, a la hora de trabajar con arrays, vamos a ver algunos en mayor detalle.

splice()

¿Cómo podemos borrar un elemento de un array?

Los arrays son objetos, por lo que podemos intentar con **delete**:

```
let arr = ["voy", "a", "casa"];  
  
delete arr[1]; // remueve "a"  
  
console.log(arr[1]); // undefined  
  
// ahora arr = ["voy", , "casa"];  
  
console.log(arr.length); // 3
```

El elemento fue borrado, pero el array todavía tiene 3 elementos, podemos ver que **arr.length == 3**.

Es natural, porque **delete obj.key** borra el valor de **key**, pero es todo lo que hace. Esto está **bien en los objetos**, pero en general lo que buscamos en los arrays es que el **resto de los elementos se desplace y se ocupe el lugar libre**. Lo que esperamos es un array más corto.

Por lo tanto, necesitamos utilizar métodos especiales.

El método **arr.splice** funciona como una navaja suiza para arrays. Puede hacer todo: insertar, remover y remplazar elementos.

```
arr.splice(inicio[, cantEliminar, elem1, ..., elemN])
```

Esto modifica arr comenzando en el índice `inicio`: remueve la cantidad `cantEliminar` de elementos y luego inserta `elem1, ..., elemN` en su lugar. Lo que devuelve es un array de los elementos removidos.

```
let arr = ["Yo", "estudio", "JavaScript"];
arr.splice(1, 1); // desde el índice 1, remover 1 elemento
console.log(arr); // ["Yo", "JavaScript"]
```

Empezando desde el índice 1 removió 1 elemento.

En el próximo ejemplo removemos 3 elementos y los reemplazamos con otros 2:

```
let arr = ["Yo", "estudio", "JavaScript", "ahora", "mismo"];
// remueve los primeros 3 elementos y los reemplaza con otros
arr.splice(0, 3, "a", "bailar");
console.log(arr) // ahora ["a", "bailar", "ahora", "mismo"]
```

slice()

El método `slice()` devuelve una copia de una parte del array dentro de un nuevo array empezando desde `inicio` hasta `fin` (fin no incluido). El array original no se modificará. La sintaxis es la siguiente:

```
arr.slice([inicio], [fin]);
```

Por ejemplo:

```
let arr = ["t", "e", "s", "t"];
console.log( arr.slice(1, 3) ); // e,s (copia desde 1 hasta 3)
```

split()

Analicemos una situación de la vida real. Estamos programando una app de mensajería y el usuario ingresa una lista de receptores delimitada por comas: Celina, David, Federico. Pero para nosotros un array sería mucho más práctico que una simple string. ¿Cómo podemos hacer para obtener un array?

El método `str.split(delim)` hace precisamente eso. Separa la string en elementos según el delimitante `delim` dado y los devuelve como un array.

En el ejemplo de abajo, separamos por “coma seguida de espacio”:

```
let nombres = 'Bilbo, Gandalf, Nazgul';
let arr = nombres.split(',');
for (let name of arr) {
  console.log(`Un mensaje para ${name}.`); // Un mensaje para Bilbo y los
                                              // otros nombres
}
```

reverse()

El método arr.reverse() revierte el orden de los elementos en arr.

Por ejemplo:

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();
console.log( arr ); // 5,4,3,2,1
```

sort(fn)

Cuando usamos arr.sort(), este ordena el propio array cambiando el orden de los elementos.

También devuelve un nuevo array ordenado pero éste usualmente se descarta ya que arr en sí mismo es modificado.

Por ejemplo:

```
let arr = [ 1, 2, 15 ];
// el método reordena el contenido de arr
arr.sort();
console.log( arr ); // 1, 15, 2
```

Los elementos fueron reordenados a 1, 15, 2. Pero ¿por qué pasa esto?

Los elementos son ordenados como Strings (cadenas de caracteres) por defecto

Todos los elementos son convertidos a String para ser comparados. En el caso de Strings se aplica el orden lexicográfico, por lo que efectivamente "2" > "15".

Para usar nuestro propio criterio de reordenamiento, necesitamos proporcionar una función como argumento de arr.sort().

La función debe comparar dos valores arbitrarios y devolver el resultado, sería parecido al comparator que conocemos en Java:

```
function compare(a, b) {
  if (a > b) return 1; // si el primer valor es mayor que el segundo
  if (a == b) return 0; // si ambos valores son iguales
  if (a < b) return -1; // si el primer valor es menor que el segundo
}
```

Por ejemplo, para ordenar como números:

```
function compareNumeric(a, b) {  
  if (a > b) return 1;  
  if (a == b) return 0;  
  if (a < b) return -1;  
}  
  
let arr = [ 1, 15, 2 ];  
arr.sort(compareNumeric);  
console.log(arr); // 1, 2, 15
```

map()

El método **map()** crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos. La sintaxis es:

```
let result = arr.map(function(elemento, indice, array) {  
  // devuelve el nuevo valor en lugar de item  
});
```

Por ejemplo, acá transformamos cada elemento en el valor de su respectivo largo (length):

```
let longitudes = ["Bilbo", "Gandalf", "Nazgul"].map(function(elemento){  
  elemento.length();  
});  
console.log(longitudes); // 5,7,6
```

flat()

El método **flat()** crea una nueva matriz con todos los elementos de sub-array concatenados recursivamente hasta la profundidad especificada.

```
var nuevoArray = arr.flat([profundidad]);
```

El método tiene un solo parámetro que es, **depth**, este parámetro es opcional y especifica qué tan profunda debe aplanarse una estructura de matriz anidada. El valor predeterminado es 1.

Ejemplos:

```
var arr1 = [1, 2, [3, 4]];  
arr1.flat();  
// [1, 2, 3, 4]  
  
var arr2 = [1, 2, [3, 4, [5, 6]]];  
arr2.flat();  
// [1, 2, 3, 4, [5, 6]]
```

flatMap()

El método `flatMap()` devuelve un nuevo array formado al aplicar una función de devolución de llamada determinada a cada elemento de la matriz y luego aplanar el resultado en un nivel. Es idéntico a un `map()` seguido de un `flat()` de profundidad 1, pero un poco más eficiente que llamar a esos dos métodos por separado.

```
var arrayNuevo = array.flatMap(function(elemento, indice, array){  
    // retorna elementos para el nuevo array  
});
```

Ejemplos:

```
var arr1 = [1, 2, 3, 4];  
var arr2 = arr1.map(function(x){  
    x = x * 2;  
});  
console.log(arr2) // [[2], [4], [6], [8]]  
var arr2 = arr1.flatMap(function(x){  
    x = x * 2;  
});  
console.log(arr2) // [2, 4, 6, 8]
```

METODOS EXTRAS DE ARRAY

Vamos a mostrar otros métodos más de la clase Array

Método	Descripción	Ejemplo
<code>concat()</code>	Une dos o más arrays	<code>var arraytotal = array1.concat(array2);</code>
<code>join()</code>	Junta los elementos de un array en una cadena con un separador – opcional.	<code>var fruta = ["Kiwi", "Limon", "Otra"];</code> <code>var ej = frutas.join();</code> <code>// Kiwi, Limon, Otra</code>
<code>pop()</code>	Borra el último elemento del array y devuelve su contenido	<code>var a = frutas.pop();</code>
<code>push()</code>	Añade nuevos elementos al array y devuelve su nueva longitud	<code>var a = frutas.push("Uva");</code>

shift()	Elimina el primer elemento del array y devuelve el elemento	<code>var a = frutas.shift();</code>
find()	El método find() devuelve el valor del primer elemento del array que cumple la función de prueba proporcionada.	<code>const array1 = [5, 12, 8, 130, 44]; const encontrado = array1.find(elemento => elemento > 10); console.log(encontrado); // resultado: 12</code>
unshift()	Añade elementos al inicio del array y devuelve la nueva longitud	<code>var frutas = ["Banana", "Naranja", "Manzana"]; frutas.unshift("Limon", "Anana"); //Limon, Anana, Banana, "Naranja, Manzana</code>

OBJETOS MAP Y SET

Hasta este momento, hemos aprendido sobre las siguientes estructuras de datos:

- Objetos para almacenar colecciones de datos ordenadas mediante una clave.
- Arrays para almacenar colecciones ordenadas de datos.

Pero eso no es suficiente para la vida real. Por eso también existen Map y Set.

MAP

Map es, al igual que Object, una colección de datos identificados por claves. Pero la principal diferencia es que Map, permite claves de cualquier tipo.

Los métodos y propiedades son:

- `new Map()`: crea el mapa.
- `map.set(clave, valor)`: almacena el valor asociado a la clave.
- `map.get(clave)`: devuelve el valor de la clave. Será undefined si la clave no existe en map.
- `map.has(clave)`: devuelve true si la clave existe en map, false si no existe.
- `map.delete(clave)`: elimina el valor de la clave.
- `map.clear()`: elimina todo de map.
- `map.size`: tamaño, devuelve la cantidad actual de elementos.

Por ejemplo:

```
let map = new Map();

map.set('1', 'str1'); // un string como clave
map.set(1, 'num1'); // un número como clave
map.set(true, 'bool1'); // un booleano como clave
// ¿recuerda el objeto regular? convertiría las claves a string.

// Map mantiene el tipo de dato en las claves, por lo que estas dos son
// diferentes:

console.log( map.get(1) ); // 'num1'
console.log( map.get('1') ); // 'str1'
console.log( map.size ); // 3
```

Podemos ver que, a diferencia de los objetos, las claves no se convierten en strings. Cualquier tipo de clave es posible en un Map.

También podemos usar objetos como claves.

Por ejemplo:

```
let john = { name: "John" };

// para cada usuario, almacenemos el recuento de visitas
let contadorVisitasMap = new Map();

// john es la clave para el Map
contadorVisitasMap.set(john, 123);
console.log(contadorVisitasMap.get(john)); // 123
```

El uso de objetos como claves es una de las características de Map más notables e importantes. Esto no se aplica a los objetos: una clave de tipo String está bien en un Object, pero no podemos usar otro Object como clave.

ITERACIÓN SOBRE MAP

Para recorrer un map, hay 3 métodos:

- `map.keys()`: devuelve un iterable para las claves.
- `map.values()`: devuelve un iterable para los valores.
- `map.entries()`: devuelve un iterable para las entradas [clave, valor]. Es el que usa por defecto en `for..of`.

Por ejemplo:

```
let recetaMap = new Map([
  ['pepino', 500],
  ['tomates', 350],
  ['cebollas', 50]
]);
// iterando sobre las claves (verduras)
for (let vegetales of recetaMap.keys()) {
  console.log(vegetales); // pepino, tomates, cebollas
}
// iterando sobre los valores (precios)
for (let cantidad of recetaMap.values()) {
  console.log(cantidad); // 500, 350, 50
}
// iterando sobre las entradas [clave, valor]
for (let entry of recetaMap) { // lo mismo que recipeMap.entries()
  console.log(entry); // pepino,500 (etc)
}
```

SET

Un **Set** es una colección de tipo especial: “conjunto de valores” (sin claves), donde cada valor puede aparecer solo una vez.

Sus principales métodos son:

- **new Set(iterable)**: crea el set. El argumento opcional es un objeto iterable (generalmente un array) con valores para inicializarlo.
- **set.add(valor)**: agrega un valor, y devuelve el set en sí.
- **set.delete(valor)**: elimina el valor, y devuelve true si el valor existía al momento de la llamada; si no, devuelve false.
- **set.has(valor)**:devuelve true si el valor existe en el set, si no, devuelve false.
- **set.clear()**: elimina todo el contenido del set.
- **set.size**: es la cantidad de elementos.

La característica principal es que llamadas repetidas de `set.add(valor)` con el mismo valor no hacen nada. Esa es la razón por la cual cada valor aparece en Set solo una vez.

```

let setNombres = new Set();
let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };
// visitas, algunos usuarios lo hacen varias veces
setNombres.add(john);
setNombres.add(pete);
setNombres.add(mary);
setNombres.add(john);
setNombres.add(mary);
// set solo guarda valores únicos
console.log(set.size); // 3
for (let usuario of setNombres) {
  console.log(usuario.name); // John (luego Pete y Mary)
}

```

ITERACIÓN SOBRE SET

Podemos recorrer Set con `for..of` o usando `forEach`:

```

let setFrutas = new Set(["naranjas", "manzanas", "uvas"]);
for (let valor of setFrutas){
  console.log(valor); // "naranjas", "manzanas", "uvas"
}
// lo mismo que forEach:
setFrutas.forEach((valor, valorDeNuevo, setFrutas) => {
  console.log(valor); // "naranjas", "manzanas", "uvas"
});

```

JSON

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es un formato ligero de **intercambio de datos**. JSON es un formato que almacena información estructurada y se utiliza principalmente para transferir datos entre un servidor y un cliente.

Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript, **Standard ECMA-262**. JSON es un **formato de texto** que es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, Java, JavaScript, y muchos otros. Este formato funcionan bien para lograr la carga asincrónica de los datos almacenados, lo que significa que un sitio web puede actualizar su información sin actualizar la página.

SINTAXIS JSON

Para crear correctamente un archivo **.json**, debes seguir la sintaxis correcta.

JSON está constituido por dos estructuras:

- Una colección de pares de nombre/valor. En varios lenguajes esto es conocido como un objeto.
- Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o secuencias.

Objetos

Hay dos elementos centrales en un objeto JSON: **claves** (Keys) y **valores** (Values).

- Las **Keys** deben ser cadenas de caracteres (strings). Como su nombre en español lo indica, estas contienen una secuencia de caracteres rodeados de comillas.
- Los **Values** son un tipo de datos JSON válido. Puede tener la forma de un arreglo (array), objeto, cadena (string), booleano, número o nulo.

Un **objeto JSON** comienza y termina con **llaves {}**. Puede tener dos o más pares de claves/valor dentro, con **una coma para separarlos**. Así mismo, cada **key** es seguida por dos **puntos** para distinguirla del valor.

Ejemplo:

```
{"ciudad": "Nueva York", "país": "Estados Unidos"}
```

Aquí tenemos dos pares de clave/valor: **ciudad** y **país** son las claves, **Nueva York** y **Estados Unidos** son los **valores**. Los valores en este ejemplo, son Strings. Por eso también están entre comillas, similares a las claves.

Array

Un valor de un array puede contener objetos JSON, lo que significa que utiliza el mismo concepto de par clave/valor. Por ejemplo:

```
"estudiantes": [
    {"primerNombre": "Tom", "Apellido": "Jackson"},
    {"primerNombre": "Linda", "Apellido": "Garner"},
    {"primerNombre": "Adam", "Apellido": "Cooper"}]
```

En este caso, la información entre corchetes es un array, que tiene tres objetos.

METODOS JSON

Si analizamos bien la sintaxis de un JSON, nos daremos cuenta que es muy similar a un objeto de Javascript y que no debería ser muy difícil pasar de JSON a JavaScript y viceversa.

En Javascript tenemos una serie de métodos que nos facilitan esa tarea, pudiendo trabajar con Strings que contengan JSON y objetos Javascript de forma indiferente:

Convertir JSON a Objeto

La acción de convertir JSON a objeto JavaScript se le suele denominar parsear. Es una acción que analiza un **String que contiene un JSON** válido y devuelve un **objeto JavaScript** con dicha información correctamente estructurada. Para ello, utilizaremos el método **JSON.parse()**:

```
const str = '{ "name": "Manz", "life": 99 }';
const obj = JSON.parse(str);
obj.name; // 'Manz'
obj.life; // 99
```

Como se puede ver, **obj** es un **objeto** generado a partir del **JSON** recogido en la variable **str** y podemos consultar sus propiedades y trabajar con ellas sin problemas.

Convertir Objeto a JSON

La acción inversa, convertir un **objeto Javascript a JSON** también se puede realizar fácilmente haciendo uso del método **JSON.stringify()**. Este método difícil de pronunciar viene a ser algo así como **"convertir a texto"**, y lo podemos utilizar para transformar un **objeto de Javascript a JSON** rápidamente:

```
const obj = {
  name: "Manz",
  life: 99,
  saludar: function () {
    return "Hola!";
  },
};

const str = JSON.stringify(obj);
console.log(str); // '{"name":"Manz","life":99}'
```

Las **funciones no están soportadas por JSON**, por lo que si intentamos convertir un objeto que contiene métodos o funciones, **JSON.stringify()** no fallará, pero simplemente devolverá un String **omitiendo las propiedades que contengan funciones**.

MANEJO DE ERRORES

La sentencia try consiste en un bloque “try” que contiene una o más sentencias. Las llaves {} se deben utilizar siempre, incluso para una bloques de una sola sentencia. Al menos un bloque “catch” o un bloque “finally” debe estar presente. Esto nos da tres formas posibles para la sentencia “try”:

1. try...catch
2. try...finally
3. try...catch...finally

Un bloque “catch” contiene sentencias que especifican que hacer si una excepción es lanzada en el bloque “try”. Si cualquier sentencia dentro del bloque “try” (o en una función llamada desde dentro del bloque “try”) lanza una excepción, el control cambia inmediatamente al bloque “catch”. Si no se lanza ninguna excepción en el bloque “try”, el bloque “catch” se omite.

La bloque “finally” se ejecuta después del bloque “try” y el/los bloque(s) “catch” hayan finalizado su ejecución. Éste bloque siempre se ejecuta, independientemente de si una excepción fue lanzada o capturada.

Puede anidar una o más sentencias “try”. Si una sentencia “try” interna no tiene una bloque “catch”, se ejecuta el bloque “catch” de la sentencia “try” que la encierra.

Usted también puede usar la declaración “try” para manejar excepciones de JavaScript.

Cuando solo se utiliza un bloque “catch”, el bloque “catch” es ejecutado cuando cualquier excepción es lanzada. Por ejemplo, cuando la excepción ocurre en el siguiente código, el control se transfiere a la cláusula “catch”.

```
try {
    throw "miExcepcion"; // genera una excepción
} catch(e) {
    // sentencias para manejar cualquier excepción
    logMyErrors(e); // pasa el objeto de la excepción al manejador de
                    //errores
}
function isValidJSON(text) {
    try {
        JSON.parse(text);
        return true;
    } catch {
        return false;
    }
}
```

También se pueden crear "bloques "catch" condicionales", combinando bloques try...catch con estructuras "if...else" como estas:

```
try {
    miRutina(); // puede lanzar tres tipos de excepciones
} catch (e) {
    if (e instanceof TypeError) {
        // sentencias para manejar excepciones TypeError
    } else if (e instanceof RangeError) {
        // sentencias para manejar excepciones RangeError
    } else if (e instanceof EvalError) {
        // sentencias para manejar excepciones EvalError
    } else {
        // sentencias para manejar cualquier excepción no especificada
        logMyErrors(e); // pasa el objeto de la excepción al manejador de
                        // errores
    }
}
```

El siguiente ejemplo **abre** un archivo y **después ejecuta** sentencias que usan el archivo (JavaScript del lado del servidor permite acceder a archivos). Si una excepción es lanzada mientras el archivo está abierto, la cláusula "**finally**" **cierra el archivo** antes de que el script falle. El código en "**finally**" también se ejecuta después de un retorno explícito de los bloques "**try**" o "**catch**".

```
openMyFile(){
    try {
        // retiene un recurso
        escribirMyArchivo(informacion);
    } catch (e) {
        // sentencias para manejar cualquier excepción
        logMyErrors(e); // pasa el objeto de la excepción al manejador de
                        // errores
    } finally {
        cerrarMiArchivo(); // siempre cierra el recurso
    }
}
```

INTRODUCCIÓN A DOM

Cuando aprendimos HTML/CSS, nos dimos cuenta que, sólo podremos crear páginas "estáticas" (sin demasiada personalización por parte del usuario), pero si añadimos Javascript, podremos crear páginas "dinámicas". Cuando hablamos de páginas dinámicas, nos referimos a que podemos dotar de la potencia y flexibilidad que nos da un lenguaje de programación para crear documentos y páginas mucho más ricas, que brinden una experiencia más completa y con el que se puedan automatizar un gran abanico de tareas y acciones.

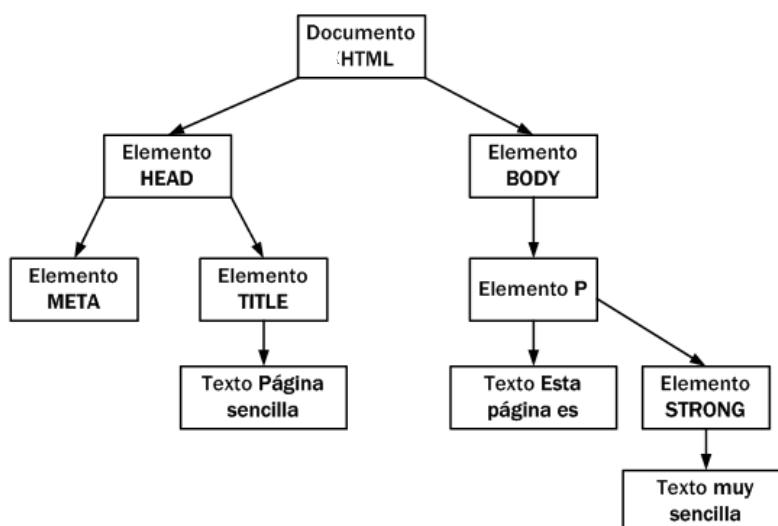
¿QUÉ ES EL DOM?

Las siglas DOM significan **Document Object Model**, o lo que es lo mismo, la estructura del documento HTML. Una página HTML está formada por múltiples etiquetas HTML, anidadas una dentro de otra, formando un árbol de etiquetas relacionadas entre sí, que se denomina **árbol DOM** (o simplemente DOM).

Supongamos que tenemos la siguiente página:

```
<!DOCTYPE>
<html>
  <head>
    <meta/>
    <title>Página sencilla</title>
  </head>
  <body>
    <p>Esta página es <strong>muy sencilla</strong></p>
  </body>
</html>
```

El árbol de etiquetas que nos mostraría el DOM, sería el siguiente:



En Javascript, cuando nos referimos al **DOM** nos referimos a esta estructura, que podemos modificar de forma dinámica desde Javascript, añadiendo nuevas etiquetas, modificando o eliminando otras, cambiando sus atributos HTML, añadiendo clases, cambiando el contenido de texto, etc..

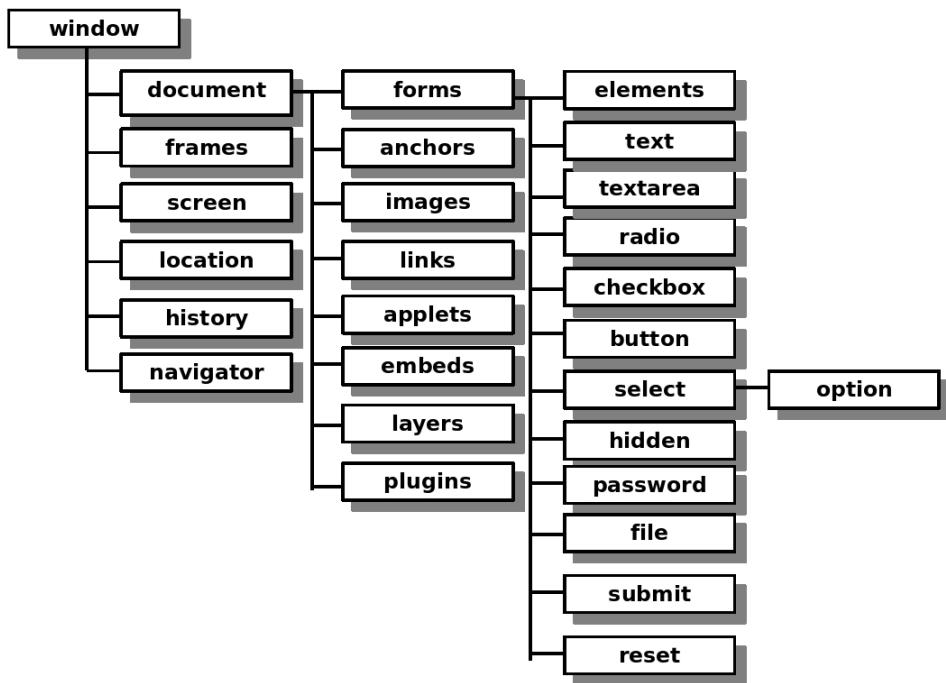
Al estar "amparado" por un **lenguaje de programación**, todas estas tareas se pueden automatizar, incluso indicando que se realicen cuando el usuario haga acciones determinadas, como por ejemplo: pulsar un botón, mover el ratón, hacer click en una parte del documento, escribir un texto, etc...

¿COMO PODEMOS ACCEDER A LAS ETIQUETAS?

Cuando se carga una página, el navegador crea una jerarquía de objetos en memoria que sirven para controlar los distintos elementos de dicha página. Con Javascript y la nomenclatura de objetos que hemos aprendido, podemos trabajar con esa jerarquía de objetos, acceder a sus propiedades e invocar sus métodos.

Cualquier elemento de la página se puede controlar de una manera u otra accediendo a esa jerarquía. Es crucial conocerla bien para poder controlar perfectamente las páginas web con Javascript o cualquier otro lenguaje de programación del lado del cliente.

Esta jerarquía de objetos está compuesta de la siguiente manera:



Como se puede apreciar, todos los objetos comienzan en un objeto que se llama **window**. Este objeto ofrece una serie de métodos y propiedades para controlar la ventana del navegador. Con ellos podemos controlar el aspecto de la ventana, la barra de estado, abrir ventanas secundarias y otras cosas.

Además de ofrecer control, el objeto window da acceso a otros objetos como el **document** (La página web que se está visualizando), el historial de páginas visitadas o los distintos frames de la ventana. De modo que para acceder a cualquier otro objeto de la jerarquía deberíamos empezar por el objeto window. Tanto es así que JavaScript **entiende** perfectamente que la jerarquía empieza en window aunque no lo señalemos.

OBJETO WINDOW

Vamos a ver algunos de los métodos de la clase window:

Método	Descripción.
<code>alert(texto)</code>	Presenta una ventana de alerta donde se puede leer el texto que recibe por parámetro
<code>back()</code>	Ir una página atrás en el historial de páginas visitadas.
<code>captureEvents(eventos)</code>	Captura los eventos que se indiquen por parámetro
<code>close()</code>	Cierra la ventana
<code>confirm(texto)</code>	Muestra una ventana de confirmación y permite aceptar o rechazar.
<code>find()</code>	Muestra una ventanita de búsqueda.
<code>home()</code>	Ir a la página de inicio que haya configurada en el explorador.
<code>prompt(pregunta,inicializacionDeLaRespuesta)</code>	Muestra una caja de diálogo para pedir un dato. Devuelve el dato que se ha escrito.
<code>setInterval()</code>	Define un script para que sea ejecutado indefinidamente en cada intervalo de tiempo.

<code>setTimeout(sentencia,milisegundos)</code>	Define un script para que sea ejecutado una vez después de un tiempo de espera determinado.
<code>clearInterval()</code>	Elimina la ejecución de sentencias asociadas a un intervalo indicadas con el método <code>setInterval()</code> .
<code>clearTimeout()</code>	Elimina la ejecución de sentencias asociadas a un tiempo de espera indicadas con el método <code>setTimeout()</code> .

Estos son solo algunos de los métodos del objeto Window, para conocer más sobre este objeto, sus propiedades y sus métodos. Les recomendamos entrar al siguiente link: [ObjetoWindow](#).

EL OBJETO DOCUMENT

En Javascript, la forma de acceder al DOM es a través de un objeto llamado **document**, que representa el árbol DOM de la página o pestaña del navegador donde nos encontramos. En su interior pueden existir varios tipos de elementos, pero principalmente serán o **Element** o **Node**:

- **Element** no es más que la representación genérica de una **etiqueta**: **HTMLElement**.
- **Node** es una unidad más básica, la cuál puede ser **Element** o un **nodo de texto**.

SELECCIONAR ELEMENTOS DEL DOM

Si nos encontramos en nuestro código Javascript y queremos hacer modificaciones en un elemento de la página HTML, lo primero que debemos hacer es buscar dicho elemento. Para ello, se suele intentar identificar el elemento a través de el **nombre de la etiqueta** o de alguno de sus atributos más utilizados, generalmente el **id** o la **clase**.

MÉTODOS TRADICIONALES

Existen varios métodos, los más clásicos y tradicionales para realizar búsquedas de elementos en el documento. Observa que si lo que buscas es un elemento específico, lo mejor sería utilizar `getElementById()`, en caso contrario, si utilizamos uno de los 3 siguientes métodos, nos devolverá un Array donde tendremos que elegir el elemento en cuestión posteriormente:

Método	Descripción.
.getElementById(id)	Busca el elemento HTML con el id id. Si no, devuelve null.
.getElementsByClassName(class)	Busca elementos con la clase class. Si no, devuelve [].
.getElementsByName(name)	Busca elementos con atributo name name. Si no, devuelve [].
.getElementsByTagName(tag)	Cierra la ventana

Estos son los **4 métodos tradicionales** de Javascript para manipular el DOM. Se denominan tradicionales porque son los que existen en Javascript desde versiones más antiguas. Dichos métodos te permiten buscar elementos en la página dependiendo de los atributos **id**, **class**, **name** o **de la propia etiqueta**, respectivamente.

getElementById()

El primer método, `.getElementById(id)` busca un elemento HTML con el **id** especificado en **id por parámetro**. En principio, un documento HTML bien construído **no debería** tener más de un elemento con el mismo id, por lo tanto, este método devolverá siempre un solo elemento:

```
const page = document.getElementById("page"); // <div id="page"></div>
```

getElementsByClassName()

Por otro lado, el método `.getElementsByClassName(class)` permite buscar los elementos con la clase especificada en **class**. Es importante darse cuenta del matiz de que el método tiene `getElements` en plural, y esto es porque **al devolver clases** (al contrario que los **id**) **se pueden repetir**, y por lo tanto, **devolvernos varios elementos, no sólo uno**.

```
const items = document.getElementsByClassName("item"); // [div, div, div]
console.log(items[0]); // Primer item encontrado:
// class="item"></div>
console.log(items.length); // 3
```

getElementsByTagName()

Esta función nos permite obtener todos los elementos **cuya etiqueta sea igual al parámetro** que le pasamos a la función. Por ejemplo, para seleccionar todos los **h1** de una página html.

```

const items = document.getElementsByTagName("h1"); // [h1, h1]
console.log(items[0]); // Primer item encontrado:
// <h1></h1>

console.log(items.length); // 2

```

Una manera menos tradicional pero que pueden llegar a ver es, seleccionar la etiqueta por su nombre sin usar ningún método, por ejemplo:

```
const items = document.h1;
```

Esto no puede traer muchas etiquetas o una sola como en el caso de body.

ATRIBUTOS DE LOS ELEMENTOS

De la misma manera que podemos acceder a los elementos del HTML, también podemos acceder a sus atributos, podemos ponerle nuevos valores a sus atributos, removerlos o validar si están. Esto se hace con los siguientes métodos:

Método	Descripción.
.getAttribute(atributo)	Devuelve el valor del atributo especificado en el elemento
getAttributeNames()	Devuelve un Array con los atributos del elemento
.setAttribute(atributo, valor)	Establece el valor de un atributo en el elemento indicado. Si el atributo ya existe, el valor es actualizado, en caso contrario, el nuevo atributo es añadido con el nombre y valor indicado.
hasAttributes()	Indica si el elemento tiene atributos HTML.
.hasAttribute(atributo)	El método hasAttribute() devuelve un valor Booleano indicando si el elemento tiene el atributo especificado o no.
.removeAttribute(atributo)	removeAttribute elimina un atributo del elemento especificado.

Estos métodos son bastante autoexplicativos y fáciles de entender, aún así, vamos a ver unos ejemplos de uso donde podemos ver como funcionan:

```
// Obtenemos <div id="page" class="info data dark" data-number="5"></div>
const div = document.getElementById("page");

div.hasAttribute("data-number"); // true (data-number existe)
div.hasAttributes();           // true (tiene 3 atributos)

div.getAttributeNames();        // ["id", "data-number", "class"]
div.getAttribute("id");         // "page"

div.removeAttribute("id");     // class="info data dark" y data-
                                // number="5"

div.setAttribute("id", "page"); // Vuelve a añadir id="page"
```

getAttribute()

Vamos a ver en mayor profundidad este método, recordemos que el método `.getAttribute()` devuelve el valor del atributo especificado en el elemento. Si el atributo especificado no existe, el valor returned puede ser tanto null como "" (una cadena vacía).

```
var div1 = document.getElementById("div1");
var align = div1.getAttribute("align");
alert(align); // Muestra el valor de la alineación(align) del elemento con
              // id="div1"
```

Otro ejemplo sería si queremos validar que el input de un formulario no está vacío, usaríamos el atributo value:

```
<input type="text" id="inputNombre" value=" ">
var nombre = document.getElementById("inputNombre").getAttribute("value");
if(nombre == " "){
  alert("El nombre está vacío");
}
```

De la misma manera que vimos que podemos acceder a los elementos por los nombres, también podemos hacerlo con los atributos, por ejemplo:

```
var nombre = input.value;
```

Siempre es mejor hacer uso de los métodos que nos da el DOM, ya que facilita la lectura.

MODIFICAR ELEMENTOS DEL DOM

En este artículo vamos a centrarnos en tres categorías:

- Reemplazar contenido de elementos en el DOM
- Insertar elementos en el DOM
- Eliminar elementos del DOM

REEMPLAZAR CONTENIDO

Comenzaremos por la familia de propiedades siguientes, que enmarcamos dentro de la categoría de **reemplazar contenido** de elementos HTML. Se trata de una vía rápida con la cuál podemos añadir (*o más bien, reemplazar*) el contenido de una etiqueta HTML.

Método	Descripción.
.nodeName	Devuelve el nombre del nodo (etiqueta si es un elemento HTML). Sólo lectura.
.textContent	Devuelve el contenido de texto del elemento. Se puede asignar para modificar.
.innerHTML	Devuelve el contenido HTML del elemento. Se puede usar asignar para modificar.
.outerHTML	Idem a .innerHTML pero incluyendo el HTML del propio elemento HTML.
.innerText	Versión no estándar de .textContent de Internet Explorer con diferencias. Evitar.
.outerText	Versión no estándar de .textContent/.outerHTML de Internet Explorer. Evitar.

LA PROPIEDAD TEXTCONTENT

La propiedad .textContent nos devuelve el **contenido de texto** de un elemento HTML. Es útil para obtener (*o modificar*) **sólo el texto** dentro de un elemento, obviando el etiquetado HTML:

```
const div = document.getElementById("div"); // <div></div>
div.textContent = "Hola a todos"; // <div>Hola a todos</div>
div.textContent; // "Hola a todos"
```

Observa que también podemos utilizarlo para **reemplazar el contenido de texto**, asignándolo como si fuera una variable o constante.

LA PROPIEDAD INNERHTML

Por otro lado, la propiedad .innerHTML nos permite hacer lo mismo, pero interpretando el código HTML indicado y renderizando sus elementos:

```

const div = document.getElementById("info"); // <div class="info"></div>
div.innerHTML = "<strong>Importante</strong>"; // Interpreta el HTML
div.innerHTML; // "<strong>Importante</strong>"
div.textContent; // "Importante"
div.textContent = "<strong>Importante</strong>"; // No interpreta el HTML

```

Observa que la diferencia principal entre `.innerHTML` y `.textContent` es que el primero renderiza e interpreta el marcado HTML, mientras que el segundo lo inserta como contenido de texto literalmente.

INSERTAR ELEMENTOS

Hemos visto, como reemplazar contenido y como seleccionar elementos, pero no hemos visto como añadir los elementos al documento HTML actual (conectarlos al DOM), operación que se puede realizar de diferentes formas mediante los siguientes métodos disponibles:

Método	Descripción.
<code>.appendChild(node)</code>	Añade como hijo el nodo node. Devuelve el nodo insertado.
<code>.insertAdjacentElement(pos, elem)</code>	Inserta el elemento elem en la posición pos. Si falla, null.
<code>.insertAdjacentHTML(pos, str)</code>	Inserta el código HTML str en la posición pos.
<code>.insertAdjacentText(pos, text)</code>	Inserta el texto text en la posición pos.
<code>.insertBefore(new, node)</code>	Inserta el nodo new antes de node y como hijo del nodo actual.
<code>.appendChild(node)</code>	Añade como hijo el nodo node. Devuelve el nodo insertado.

EL MÉTODO APPENDCHILD()

Uno de los métodos más comunes para añadir un elemento HTML creado con Javascript es `appendChild()`. Como su propio nombre indica, este método realiza un “append”, es decir, **inserta el elemento como un hijo al final de todos los elementos hijos que existan**.

Es importante tener clara esta particularidad, porque aunque es lo más común, no siempre querremos insertar el elemento en esa posición:

```

const img = document.getElementByTagName("img");
img.src = "https://lenguajejs.com/assets/logo.svg";
img.alt = "Logo Javascript";
document.body.appendChild(img);

```

En este ejemplo podemos ver como añadimos los atributos src y alt, obligatorios en una etiqueta de imagen. Por último, conectamos al DOM el elemento, utilizando el método .appendChild() sobre document.body que no es más que una referencia a la etiqueta <body> del documento HTML.

ELIMINAR ELEMENTOS

Al igual que podemos insertar o reemplazar elementos, también podemos eliminarlos. Ten en cuenta que al «eliminar» un nodo o elemento HTML, lo que hacemos realmente no es borrarlo, sino **desconectarlo del DOM o documento HTML**, de modo que no están conectados, pero siguen existiendo. Estas operaciones que se puede realizar de diferentes formas mediante los siguientes métodos disponibles:

Método	Descripción.
.remove()	Elimina el propio nodo de su elemento padre.
.removeChild(node)	Elimina y devuelve el nodo hijo node.
.replaceChild(new, old)	Reemplaza el nodo hijo old por new. Devuelve old.

El método `.remove()` se encarga de desconectarse del DOM a sí mismo, mientras que el segundo método, `.removeChild()`, desconecta el nodo o elemento HTML proporcionado. Por último, con el método `.replaceChild()` se nos permite cambiar un nodo por otro.

EL MÉTODO REMOVE()

Probablemente, la forma más sencilla de eliminar nodos o elementos HTML es utilizando el método `.remove()` sobre el nodo o etiqueta a eliminar:

```

const div = document.getElementById("deleteme");

div.isConnected; // true
div.remove();
div.isConnected; // false

```

EVENTOS

En la programación tradicional, las aplicaciones se ejecutan secuencialmente de principio a fin para producir sus resultados. Sin embargo, en la actualidad el modelo predominante es el de la programación basada en eventos. Los scripts y programas esperan sin realizar ninguna tarea hasta que se produzca un evento. Una vez producido, ejecutan alguna tarea asociada a la aparición de ese evento y cuando concluye, el script o programa vuelve al estado de espera.

JavaScript permite realizar scripts con ambos métodos de programación: secuencial y basada en eventos. Los eventos de JavaScript permiten la interacción entre las aplicaciones JavaScript y los usuarios. Cada vez que se pulsa un botón, se produce un evento. Cada vez que se pulsa una tecla, también se produce un evento. No obstante, para que se produzca un evento no es obligatorio que intervenga el usuario, ya que por ejemplo, cada vez que se carga una página, también se produce un evento.

TIPOS DE EVENTOS

Cada **elemento HTML** tiene definida su propia lista de posibles **eventos** que se le pueden asignar. Un mismo tipo de evento (por ejemplo, pinchar el botón izquierdo del ratón) puede estar definido para varios elementos HTML y un mismo elemento HTML puede tener asociados diferentes eventos.

El nombre de los eventos se construye mediante **el prefijo on**, seguido del **nombre** en inglés de la **acción asociada al evento**. Así, el evento de pinchar un elemento con el ratón se denomina **onclick** y el evento asociado a la acción de mover el ratón se denomina **onmousemove**.

La siguiente tabla resume los eventos más importantes definidos por JavaScript:

Evento	Descripción	Elementos para los que está definido
onblur	Un elemento pierde el foco	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Un elemento ha sido modificado	<input>, <select>, <textarea>
onclick	Pulsar y soltar el ratón	Todos los elementos
ondblclick	Pulsar dos veces seguidas con el ratón	Todos los elementos

onfocus	Un elemento obtiene el foco	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla y no soltarla	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	Página cargada completamente	<body>
onmousedown	Pulsar un botón del ratón y no soltarlo	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón "sale" del elemento	Todos los elementos
onmouseover	El ratón "entra" en el elemento	Todos los elementos
onmouseup	Soltar el botón del ratón	Todos los elementos
onreset	Inicializar el formulario	<form>
onresize	Modificar el tamaño de la ventana	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página, por ejemplo al cerrar el navegador	<body>

Los eventos más utilizados en las aplicaciones web tradicionales son `onload` para esperar a que se cargue la página por completo, los eventos `onclick`, `onmouseover`, `onmouseout` para controlar el ratón y `onsubmit` para controlar el envío de los formularios.

MANEJADORES DE EVENTOS

Un evento de JavaScript por sí mismo carece de utilidad. Para que los eventos resulten útiles, se deben **asociar funciones o código JavaScript a cada evento**. De esta forma, cuando se produce un evento se ejecuta el código indicado, por lo que la aplicación puede responder ante cualquier evento que se produzca durante su ejecución.

Las funciones o código JavaScript que se definen para cada evento se denominan *manejador de eventos* (*event handlers* en inglés) y como JavaScript es un lenguaje muy flexible, existen varias formas diferentes de indicar los manejadores:

- Manejadores como **atributos de los elementos XHTML**.
- Manejadores como **funciones JavaScript externas**.
- Manejadores "**semánticos**".

MANEJADORES COMO ATRIBUTOS HTML

Se trata del método más sencillo y a la vez *menos profesional* de indicar el código JavaScript que se debe ejecutar cuando se produzca un evento. En este caso, el código se incluye en un atributo del propio elemento HTML. En el siguiente ejemplo, se quiere mostrar un mensaje cuando el usuario pinche con el ratón sobre un botón:

```
<input type="button" value="Pinchame y verás" onclick="console.log('Gracias por pinchar');" />
```

El ejemplo anterior sólo quiere controlar el evento de pinchar con el ratón, cuyo nombre es `onclick`. Así, el elemento HTML para el que se quiere definir este evento, debe incluir un atributo llamado `onclick`.

El contenido del atributo es una cadena de texto que contiene todas las instrucciones JavaScript que se ejecutan cuando se produce el evento. En este caso, el código JavaScript es muy sencillo (`console.log('Gracias por pinchar');`), ya que solamente se trata de mostrar un mensaje.

MANEJADORES DE EVENTOS COMO FUNCIONES EXTERNAS

La definición de manejadores de eventos en los atributos HTML es un método sencillo pero **poco aconsejable para tratar con los eventos en JavaScript**. El principal inconveniente es que se complica en exceso en cuanto se añaden algunas pocas instrucciones, por lo que solamente es recomendable para los casos más sencillos.

Cuando el código de la función manejadora es más complejo, como por ejemplo la validación de un formulario, **es aconsejable agrupar todo el código JavaScript en una función externa** que se invoca desde el código HTML cuando se produce el evento.

De esta forma, el siguiente ejemplo:

```
<input type="button" value="Pinchame y verás" onclick="console.log('Gracias por pinchar');" />
```

Se puede transformar en:

```
function muestraMensaje() {  
    console.log('Gracias por pinchar');  
}  
  
<input type="button" value="Pinchame y verás" onclick="muestraMensaje()" />
```

MANEJADORES DE EVENTOS SEMÁNTICOS

Utilizar los atributos HTML o las funciones externas para añadir manejadores de eventos tiene un grave inconveniente: **"ensucian"** el código HTML de la página.

Como es conocido, al crear páginas web se recomienda separar los contenidos (HTML) de la presentación (CSS). En lo posible, también se recomienda separar los contenidos (HTML) de la programación (JavaScript). Mezclar JavaScript y HTML complica excesivamente el código fuente de la página, dificulta su mantenimiento y reduce la semántica del documento final producido.

Afortunadamente, existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica consiste en asignar las **funciones externas** mediante las propiedades DOM de los elementos HTML. Así, el siguiente ejemplo:

```
<input type="button" value="Pinchame y verás" onclick="console.log('Gracias por pinchar');" />
```

Se puede transformar en:

```
function muestraMensaje() {  
    console.log('Gracias por pinchar');  
}  
  
document.getElementById("pinchable").onclick = muestraMensaje;  
  
<input id="pinchable" type="button" value="Pinchame y verás"/>
```

El código HTML resultante es muy "limpio", ya que no se mezcla con el código JavaScript. La técnica de los manejadores semánticos consiste en:

1. Asignar un **identificador único** al elemento HTML mediante el **atributo id**.
2. Crear una **función** de JavaScript encargada de manejar el evento.
3. **Asignar la función a un evento** concreto del elemento XHTML mediante DOM.

Asignar la función manejadora mediante DOM es un proceso que requiere una explicación detallada. En primer lugar, se obtiene la referencia del elemento al que se va a asignar el manejador:

```
document.getElementById("pinchable");
```

A continuación, se asigna el evento deseado para el elemento:

```
document.getElementById("pinchable").onclick = ...
```

Por último, se asigna la función externa. Como ya se ha comentado en capítulos anteriores, lo más importante (y la causa más común de errores) es indicar solamente el nombre de la función, es decir, prescindir de los paréntesis al asignar la función:

```
document.getElementById("pinchable").onclick = muestraMensaje;
```

Si se añaden los paréntesis al final, en realidad se está invocando la función y asignando el valor devuelto por la función al evento onclick de elemento.

ARCHIVO JAVASCRIPT

Para los ejemplo previamente vistos, habremos creado un archivo propio de JavaScript, que sería un **archivo con extensión .js** y llamaríamos este archivo mediante la etiqueta <script>. Recordemos que esto nos va a ayudar a separar el HTML de JavaScript y tener un HTML más claro.

El archivo js se vería así:

```
function muestraMensaje() {  
    console.log('Gracias por pinchar');  
}  
  
document.getElementById("pinchable").onclick = muestraMensaje;
```

Y el html se vería así:

```
<!DOCTYPE html>  
<html>  
<body>  
  
<h1>PerroMania</h1>  
<p id="demo">Parrafo</p>  
  
<input id="pinchable" type="button" value="Pinchame y verás"/>  
  
<script src="script1.js"></script>  
  
</body>  
</html>
```

ASINCRONIA

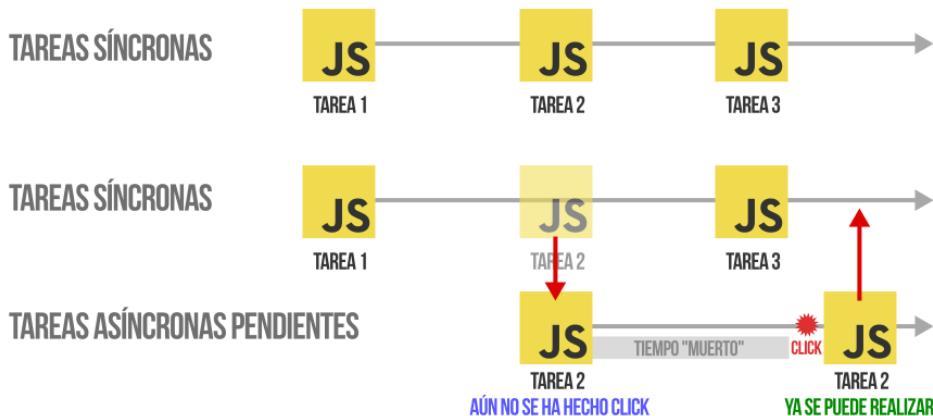
La asincronía es uno de los pilares fundamentales de Javascript, ya que es un lenguaje de programación de un **sólo subprocesso o hilo (single thread)**, lo que significa que **sólo puede ejecutar una cosa a la vez**.

Si bien los idiomas de un sólo hilo simplifican la escritura de código porque no tiene que preocuparse por los problemas de concurrencia, esto también significa que no puede realizar operaciones largas como el acceso a la red sin bloquear el hilo principal.

Imagina que solicitas datos de una *API*. Dependiendo de la situación, el servidor puede tardar un tiempo en procesar la solicitud mientras bloquea el hilo principal y hace que la página web no responda.

Ahí es donde entra en juego la asincronía que permite realizar largas solicitudes de red sin bloquear el hilo principal.

Cuando hablamos de Javascript, habitualmente nos referimos a él como un lenguaje **no bloqueante**. Con esto queremos decir que las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas, y por consiguiente, evitando proseguir con el resto de tareas.



Por lo que Javascript usa un **modelo asíncrono y no bloqueante**, con un *loop* de **eventos** implementado en un sólo hilo, (*single thread*) para operaciones de entrada y salida (*input/output*).

¿Pero qué significan todos estos conceptos? Ahora los vamos a explicar de manera más detallada.

SINGLE THREAD Y MULTI THREAD

Un hilo la unidad básica de ejecución de un proceso, cada vez que abres un programa como el navegador o tu editor de código, se levanta un proceso en tu computadora e internamente este puede tener uno o varios hilos (*threads*) ejecutándose para que el proceso funcione.

OPERACIONES DE CPU Y DE ENTRADA Y SALIDA

- **Operaciones CPU:** Aquellas que pasan el mayor tiempo consumiendo Procesos del *CPU*, por ejemplo, la escritura de ficheros.
- **Operaciones de Entrada y Salida:** Aquellas que pasan la mayor parte del tiempo esperando la respuesta de una petición o recurso, como la solicitud a una *API* o *DB*.

CONCURRENCIA Y PARALELISMO

- **Concurrencia:** cuando dos o más tareas progresan simultáneamente.
- **Paralelismo:** cuando dos o más tareas se ejecutan, al mismo tiempo.

BLOQUEANTE Y NO BLOQUEANTE

Se refiere a como la fase de espera de las operaciones afectan a nuestra aplicación:

- **Bloqueante:** Son operaciones que no devuelven el control a nuestra aplicación hasta que se ha completado. Por tanto el *thread* queda bloqueado en estado de espera.
- **No Bloqueante:** Son operaciones que devuelven inmediatamente el control a nuestra aplicación, independientemente del resultado de esta. En caso de que se haya completado, devolverá los datos solicitados. En caso contrario (si la operación no ha podido ser satisfecha) podría devolver un código de error.

SÍNCRONO Y ASÍNCRONO

Se refiere a ¿cuándo tendrá lugar la respuesta?:

- **Síncrono:** La respuesta sucede en el presente, una operación síncrona esperará el resultado.
- **Asíncrono:** La respuesta sucede a futuro, una operación asíncrona no esperará el resultado.

MECANISMOS ASÍNCRONOS EN JAVASCRIPT

Para controlar la asincronía, JavaScript cuenta con algunos mecanismos:

- Callback.
- Promises
- Async / Await.

FUNCIONES CALLBACK

Los **callbacks** (*a veces denominados funciones de retrollamada o funciones callback*) no son más que un tipo de **funciones** que se pasan por parámetro a otras funciones. El objetivo de esto es tener una forma más legible de escribir funciones, más cómoda y flexible para reutilizarlas, y además entra bastante en consonancia con el concepto de asincronía de Javascript.

CALLBACKS EN JAVASCRIPT

Vamos a ver un poco las **funciones callbacks** utilizadas para realizar tareas asíncronas. Probablemente, el caso más fácil de entender es utilizar un temporizador mediante la función `setTimeout(callback, time)`.

Dicha función nos exige dos parámetros:

- La función **callback** a ejecutar
- El **tiempo time** que esperará antes de ejecutarla

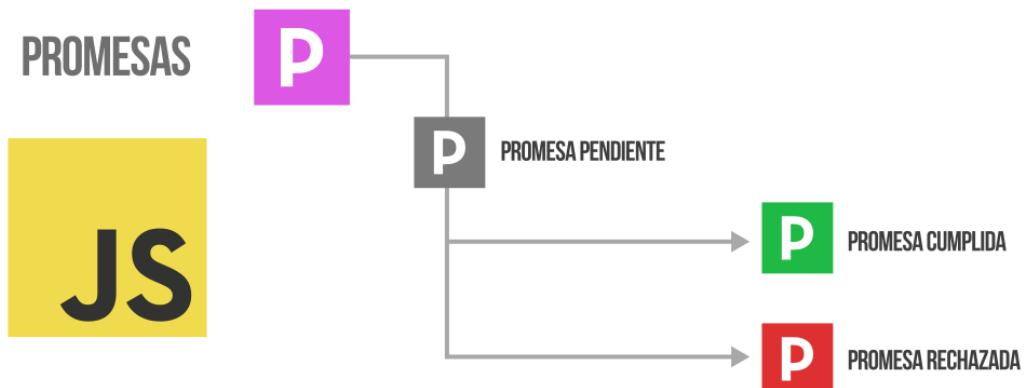
Así pues, el ejemplo sería el siguiente:

```
setTimeout(function() {  
    console.log("He ejecutado la función");  
}, 2000);
```

Simplemente, le decimos a `setTimeout()` que ejecute la función **callback** que le hemos pasado por primer parámetro cuando transcurran **2000 milisegundos** (*es decir, 2 segundos*).

PROMESAS

Como su propio nombre indica, una **promesa** es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:



- La promesa **se cumple** (*promesa resuelta*)
- La promesa **no se cumple** (*promesa se rechaza*)
- La promesa se queda en un **estado incierto** indefinidamente (*promesa pendiente*)

Con estas sencillas bases, podemos entender el funcionamiento de una promesa en JavaScript. Antes de empezar, también debemos tener claro que existen dos partes importantes de las promesas: **como consumirlas** (*utilizar promesas*) y **como crearlas** (*preparar una función para que use promesas y se puedan consumir*).

PROMESAS EN JAVASCRIPT

Las **promesas** en Javascript se representan a través de un object, y cada **promesa** estará en un estado concreto: **pendiente**, **aceptada** o **rechazada**. Además, cada **promesa** tiene los siguientes métodos, que podremos utilizar para utilizarla:

Método	Descripción.
.then(function resolve)	Ejecuta la función callback resolve cuando la promesa se cumple.
.catch(function reject)	Ejecuta la función callback reject cuando la promesa se rechaza.
.then(function resolve, function reject)	Método equivalente a las dos anteriores en el mismo .then().
.finally(function end)	Ejecuta la función callback end tanto si se cumple como si se rechaza.

CONSUMIR UNA PROMESA

La forma general de consumir una promesa es utilizando el .then() con un sólo parámetro, puesto que muchas veces lo único que nos interesa es realizar una acción cuando la promesa se cumpla:

```
fetch("/robots.txt").then(function(response) {
  /* Código a realizar cuando se cumpla la promesa */
});
```

Lo que vemos en el ejemplo anterior es el uso de la función `fetch()`, la cuál devuelve una promesa que se cumple cuando obtiene respuesta de la petición realizada. De esta forma, estaríamos preparando (de una forma legible) la forma de actuar de nuestro código a la respuesta de la petición realizada, todo ello de forma asíncrona.

API DE LAS PROMESAS

Ahora que sabemos ¿Qué son las promesas?, para qué y como se usan, podemos profundizar y aprender más sobre la **API Promise** nativa de Javascript, mediante la cuál podemos realizar operaciones con grupos de promesas, tanto independientes como dependientes entre sí.

OBJETO PROMISE

El objeto **Promise** de Javascript tiene varios **métodos estáticos** que podemos utilizar en nuestro código. Todos devuelven una promesa y son los que veremos en la siguiente tabla:

Método	Descripción.
Promise.all(Array list)	Acepta sólo si todas las promesas del Array se cumplen.
Promise.allSettled(Array list)	Acepta sólo si todas las promesas del Array se cumplen o rechazan.
Promise.any(Objet value)	Acepta con el valor de la primera promesa del Array que se cumpla.
Promise.race(Object value)	Acepta o rechaza dependiendo de la primera promesa que se procese.
Promise.resolve(Object value)	Devuelve un valor envuelto en una promesa que se cumple directamente.
Promise.reject(Object value)	Devuelve un valor envuelto en una promesa que se rechaza directamente.

PROMISE.RESOLVE() Y PROMISE.REJECT()

Mediante los métodos estáticos `Promise.resolve()` y `Promise.reject()` podemos devolver una promesa cumplida o rechazada.

```
//Promise
function resolverEn3seg() {
    return new Promise(function (resolve, reject) {
        // setTimeout(() => {
        //     resolve('2-Resuelto');
        // }, 3000);
        setTimeout(() => {
            reject(new Error("2-Ops i did it again"));
        }, 3000);
    });
}
```

ASYNC / AWAIT

Las promesas fueron una gran mejora respecto a las callbacks para controlar la asincronía en JavaScript, sin embargo pueden llegar a ser muy verbosas a medida que se requieran más y más métodos `.then()`.

Las funciones asíncronas (`async / await`) surgen para simplificar el manejo de las promesas.

La palabra **async** declara una función como asíncrona e indica que una promesa será automáticamente devuelta.

Podemos declarar como **async** funciones con nombre, anónimas o funciones flecha.

La palabra **await** debe ser usado siempre dentro de una función declarada como **async** y esperará de forma asíncrona y no bloqueante a que una promesa se resuelva o rechace.

LA PALABRA CLAVE ASYNC

En primer lugar, tenemos la palabra clave **async**. Esta palabra clave se colocará previamente a function, para definirla así como una función asíncrona, el resto de la función no cambia:

```
async function funcion_asincrona() {  
    return 42;  
}
```

En el caso de que utilicemos **arrow function**, se definiría como vemos a continuación, colocando el **async** justo antes de los parámetros de la arrow function:

```
const funcion_asincrona = async () => 42;
```

Al ejecutar la función veremos que ya nos devuelve una **promise** que ha sido cumplida, con el valor devuelto en la función (*en este caso, 42*). De hecho, podríamos utilizar un **.then()** para manejar la promesa:

```
funcion_asincrona().then(valor => {  
    console.log("El resultado devuelto es: ", valor);  
});
```

Sin embargo, veremos que lo que se suele hacer junto a **async** es utilizar la palabra clave **await**, que es donde reside lo interesante de utilizar este enfoque

LA PALABRA CLAVE AWAIT

Cualquier función definida con **async**, o lo que es lo mismo, cualquier promise puede utilizarse junto a la palabra clave **await** para manejarla. Lo que hace **await** es esperar a que se resuelva la promesa, mientras permite continuar ejecutando otras tareas que puedan realizarse:

```
const funcion_asincrona = async () => 42;  
  
const valor = funcion_asincrona();           // Promise { <fulfilled>:  
                                                // 42 }  
  
const asyncValue = await funcion_asincrona(); // 42
```

Observa que en el caso de **valor**, que se ejecuta sin **await**, lo que obtenemos es el valor devuelto por la función, **pero “envuelto” en una promesa que deberá utilizarse con .then() para manejarse**. Sin embargo, en **asyncValue** estamos obteniendo un tipo de dato numérico, guardando el valor directamente **ya procesado**, ya que **await espera a que se resuelva la promesa de forma asíncrona y guarda el valor**

API

¿QUÉ ES UNA API?

El término API es una abreviatura de **Application Programming Interfaces**, que en español significa **interfaz de programación de aplicaciones**. Se trata de un **conjunto de definiciones y protocolos** que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

Así pues, podemos hablar de una API como una especificación formal que establece cómo un módulo de un software se comunica o interactúa con otro para cumplir una o muchas funciones. Todo dependiendo de las aplicaciones que las vayan a utilizar, y de los permisos que les dé el propietario de la API a los desarrolladores de terceros.

PARA QUÉ SIRVE UNA API

Una de las principales funciones de las API es poder facilitarle el trabajo a los desarrolladores y ahorrarles tiempo y dinero. Por ejemplo, si estás creando una aplicación que es una tienda online, no necesitarás crear desde cero un sistema de pagos u otro para verificar si hay stock disponible de un producto. Podrás utilizar la API de un servicio de pago ya existente, por ejemplo PayPal, y pedirle a tu distribuidor una API que te permita saber el stock que ellos tienen.

- <https://developers.mercadolibre.com.ar/>
- <https://developer.paypal.com/docs/api/overview/>
- https://developers.facebook.com/docs/apis-and-sdks?locale=es_ES
- <https://datosgobar.github.io/georef-ar-api/>
- <https://developers.google.com/maps/documentation/javascript/overview>
- <https://rickandmortyapi.com/>
- <https://dog.ceo/dog-api/>
- <https://developer.spotify.com/documentation/web-api/>

Etc...

FETCH

¿COMO COMUNICARME CON UNA API USANDO JAVASCRIPT?

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global `fetch()` (en-US) que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de XMLHttpRequest. Fetch proporciona una alternativa mejor que puede ser empleada fácilmente por otras tecnologías como Service Workers (en-US). Fetch también aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como CORS y extensiones para HTTP.

La especificación `fetch` difiere de JQuery.ajax() en dos formas principales:

- El objeto Promise devuelto desde `fetch()` no será rechazado con un estado de error HTTP incluso si la respuesta es un error HTTP 404 o 500. En cambio, este se resolverá normalmente (con un estado `ok` configurado a `false`), y este solo sera rechazado ante un fallo de red o si algo impidió completar la solicitud.
- Por defecto, `fetch` no enviará ni recibirá cookies del servidor, resultando en peticiones no autenticadas si el sitio permite mantener una sesión de usuario (para mandar cookies, credentials de la opción `init` deberan ser configuradas). Desde el 25 de agosto de 2017. La especificación cambió la politica por defecto de las credenciales a `same-origin`. Firefox cambió desde la versión 61.0b13.

Una petición básica de `fetch` es realmente simple de realizar. Eche un vistazo al siguiente código:

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

Aquí estamos recuperando un archivo JSON a través de red e imprimiendo en la consola. El uso de `fetch()` más simple toma un argumento (la ruta del recurso que quieras obtener) y devuelve un objeto Promise conteniendo la respuesta, un objeto Response.

Esto es, por supuesto, una respuesta HTTP no el archivo JSON. Para extraer el contenido en el cuerpo del JSON desde la respuesta, **usamos el método `json()`** (definido en el mixin de Body, el cual está implementado por los objetos Request y Response).

```
// Fetch GET
async function getAllCharacters() {
  let response = await fetch("https://rickandmortyapi.com/api/character");
  let data = await response.json();
  return data;
}
```

```
//Fetch POST
async function postData(url = '', data = {}) {
  const response = await fetch(url, {
    method: 'POST', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, *cors, same-origin
    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, *same-origin, omit
    headers: {
      'Content-Type': 'application/json'
      // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade, origin, origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-cross-origin, unsafe-url
    body: JSON.stringify(data) // el body debe ser igual al "Content-Type"
  })
  return response.json();
}

postData('https://localhost:8080', { mascota: "Chiquito" })
  .then(data => {
    console.log(data);
  });
}
```

OBJETO STORAGE

El objeto Storage (API de almacenamiento web) nos permite almacenar datos de manera local en el navegador y sin necesidad de realizar alguna conexión a una base de datos.

LOCALSTORAGE Y SESSIONSTORAGE: ¿QUÉ SON?

`localStorage` y `sessionStorage` son propiedades que acceden al objeto `Storage` y tienen la función de `almacenar datos de manera local`, la diferencia entre éstas dos es que `localStorage` almacena la información de `forma indefinida o hasta que se decida limpiar los datos del navegador` y `sessionStorage` almacena información mientras la pestaña donde se esté utilizando siga abierta, una vez cerrada, la información se elimina.

GUARDAR DATOS EN STORAGE

Para guardar datos usamos el método `setItem(String "key", item)`. En el primer parámetro “key” ingresamos el nombre de nuestro elemento, y en el parámetro “item” ingresamos el valor de éste.

```
localStorage.setItem('mascota', 'Chiquito');//guardo un elemento
//SessionStorage:
sessionStorage.setItem('mascota', 'Filomena');//guardo un elemento
```

RECUPERAR DATOS DE STORAGE

Con éste método obtenemos desde el Local o SessionStorage el valor de nuestro elemento, donde "key" es el nombre de éste.

```
let miMascota1 = localStorage.getItem('mascota');//obtengo un elemento  
//SessionStorage:  
let miMascota2 = sessionStorage.getItem('mascota');//obtengo un elemento
```

ELIMINAR DATOS DE STORAGE

Para eliminar un elemento dentro de Storage, usaremos el método remove("key"), que recibe el nombre del contenido a borrar.

```
localStorage.removeItem('mascota');//solo elimino este ítem  
//SessionStorage:  
sessionStorage.removeItem('mascota');//solo elimino este ítem
```

NUMERO DE ELEMENTOS EN EL STORAGE

Si queremos averiguar la cantidad de elementos que tenemos guardados en el storage, vamos a usar el método length.

```
localStorage.length;//numero de elementos en local storage  
//SessionStorage:  
sessionStorage.length;//numero de elementos en local storage
```

LIMPIAR TODO EL STORAGE

Ya para finalizar veremos la forma para eliminar todos los datos del Storage y dejarlo completamente limpio

```
localStorage.clear();//borro todos los items  
//SessionStorage:  
sessionStorage.clear();//borro todos los items
```

¿QUE SON LAS COOKIES?

Las cookies, de nombre más exacto HTTP cookies, es una tecnología que en su día inventó el navegador Netscape, y que consiste básicamente en información enviada o recibida en las cabeceras HTTP y que queda almacenada localmente client-side durante un tiempo determinado. En otras palabras, es información que queda almacenada en el dispositivo del usuario y que se envía hacia y desde el servidor web en las cabeceras HTTP.

Cuando un usuario solicita una página web (o cualquier otro recurso), **el servidor envía el documento, cierra la conexión y se olvida del usuario**. Si el mismo usuario vuelve a solicitar la misma u otra página al servidor, será tratado como si fuera la primera solicitud que realiza. Esta situación puede suponer un problema en muchas situaciones y **las cookies son una técnica que permite solucionarlo**.

Con las cookies, el servidor puede enviar información al usuario en las cabeceras HTTP de respuesta y esta información queda almacenada en el dispositivo del usuario. En la siguiente solicitud que realice el usuario la cookie es enviada de vuelta al servidor en las cabeceras HTTP de solicitud. En **el servidor podemos leer esta información y así "recordar" al usuario e información asociada a él**.

DOCUMENT.COOKIE

Mediante esta propiedad de pueden crear, modificar, eliminar y leer cookies en Javascript. Dentro de ella se incluyen diversos parámetros, como los que vemos a continuación.

PARAMETROS

Una cookie consiste en una cadena de texto (string) con varios pares key=value cada uno separado por ";" :

```
<nombre>=<valor>; expires=<fecha>; max-age=<segundos>; path=<ruta>;  
domain=<dominio>; secure; httponly;
```

Veamos cada uno de los parámetros de una cookie en más detalle.

Nombre-Valor

Es un parámetro **obligatorio** a la hora de crear la cookies. **"Nombre"** se refiere al nombre que se adjudica a la cookies, mientras que **"valor"** representa su valor. Por ejemplo, "nombre" podría ser "color_favorito" y "valor" podría ser "azul".

Expire date

Este parámetro **es opcional** y establece una fecha de final de **validez de la cookie**. La fecha se ha de establecer en formato UTC.

Otro parámetro temporal es «*max-age*», que establece la duración en segundos de la cookie.

En caso de no asignar ningún valor para estos parámetros, se creará una cookie de sesión que espirará cuando el usuario finalice la sesión.

Si se establece una fecha de validez anterior a la fecha actual, o se asigna un valor negativo en «*max-age*», lo que se conseguirá es eliminar la cookie.

Domain & Path

Este parámetro **es opcional** y básicamente se trata de la URL para la cual la cookie es válida. En el caso de "Domain" se refiere al **dominio**, mientras que "Path" es el **subdominio**.

Debes tener en cuenta que la directiva same-origin policy no permite crear cookies para un dominio diferente al que crea la propia cookie. En el caso de los subdominios, se debe indicar para cuál se desea asignar la cookie. En caso de no asignar ninguna ruta, se creará automáticamente para la ruta de la página actual.

secure

Parámetro **opcional**, sin valor. Si está presente la cookie sólo es válida para conexiones encriptadas (por ejemplo mediante protocolo HTTPS).

HttpOnly

Parámetro **opcional**, no disponible en JavaScript ya que, crea cookies válidas sólo para protocolo HTTP/HTTPS y no para otras APIs, incluyendo JavaScript.

¿CÓMO CREAR UNA COOKIE CON JAVASCRIPT?

Para **guardar cookies** en JavaScript hay que definir el código y los parámetros de dicha cookie, y asignarlos a `document.cookie`.

```
document.cookie = "mascota=Malva";//guardo un elemento
```

Para crear más cookies es necesario seguir este mismo proceso para cada una de ellas. Ten en cuenta que si creas una cookie con el mismo nombre y para la misma ruta que una ya existente ésta sustituirá a la anterior.

¿CÓMO LEER UNA COOKIE CON JAVASCRIPT?

Uno de los puntos negativos de las cookies en JavaScript es que no hay una manera de leer o encontrar cookies de manera individual. Para **leer cookies en JavaScript** hay que crear un String que incluya todas las cookies válidas del documento, y manipularlo de manera que se encuentre el nombre y valor de la cookie que buscas.

El código para hacerlo es el siguiente

```
let cookies = document.cookie;//obtengo todas las cookies
```

Y los resultados se mostrarán de la siguiente manera:

```
"cookie1=valor1;cookie2=valor2;cookie3=valor3;cookie4=valor4;....
```

¿COMO ELIMINAR UN DATO ASOCIADO A UNA COOKIE?

Para esto, pisaremos la cookie ya creada y le dejaremos el dato como una cadena vacía.

```
document.cookie = "mascota= "//elimino el dato asociado al elemento
```

DIFERENCIA ENTRE COOKIES Y SESIONES

Principalmente, una de las mayores diferencias es que la información cuando la almacenas con una session se guarda en el **lado del servidor** y la información cuando la guardas con una cookie se guarda en el **lado del cliente**.

Además, las sesiones se destruyen cuando cierras el navegador (o cuando las destruyes manualmente) mientras que las cookies permanecen por un tiempo determinado en el navegador (que pueden ser varias semanas o incluso meses).

En relación con la seguridad, la cookie se guarda en el cliente, el usuario puede ver el archivo de la cookie y puede realizar operaciones similares de modificación y eliminación en el archivo de la cookie. La seguridad de los datos de la cookie es difícil de garantizar, mientras que los datos de la sesión se almacenan en el lado del servidor, tiene mejor seguridad. Si se usa junto con la base de datos, puede mantener los datos de la sesión durante mucho tiempo y obtener una buena seguridad. Por lo tanto, se puede decir que las sesiones son más seguras que las cookies.

Esto hace, **que no sea usar una o la otra**, sino según que situaciones decidir que es lo mejor para utilizar en cada caso.

EJERCICIOS DE APRENDIZAJE

¡¡Llegó el momento de poner nuestro conocimiento a prueba!! Vamos a trabajar todo lo que hemos visto en la guía de JavaScript y todo lo que veremos ahora con los videos de Youtube.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Escribir un algoritmo en el cual se consulte al usuario que ingrese ¿cómo está el día de hoy? (soleado, nublado, lloviendo). A continuación, mostrar por pantalla un mensaje que indique "El día de hoy está ...", completando el mensaje con el dato que ingresó el usuario.
2. Conocido el número en matemática PI π , pedir al usuario que ingrese el valor del radio de una circunferencia y calcular y mostrar por pantalla el área y perímetro. Recuerde que para calcular el área y el perímetro se utilizan las siguientes fórmulas:
$$\text{área} = \pi * \text{radio}^2$$
$$\text{perímetro} = 2 * \pi * \text{radio}$$
3. Escriba un programa en donde se pida la edad del usuario. Si el usuario es mayor de edad se debe mostrar un mensaje indicándolo.
4. Realiza un programa que sólo permita introducir los caracteres 'S' y 'N'. Si el usuario ingresa alguno de esos dos caracteres se deberá de imprimir un mensaje por pantalla que diga "CORRECTO", en caso contrario, se deberá imprimir "INCORRECTO".
5. Construir un programa que simule un menú de opciones para realizar las cuatro operaciones aritméticas básicas (suma, resta, multiplicación y división) con dos valores numéricos enteros. El usuario, además, debe especificar la operación con el primer carácter de la operación que desea realizar: 'S' o 's' para la suma, 'R' o 'r' para la resta, 'M' o 'm' para la multiplicación y 'D' o 'd' para la división.
6. Realizar un programa que, dado un número entero, visualice en pantalla si es par o impar. En caso de que el valor ingresado sea 0, se debe mostrar "el número no es par ni impar".
7. Escriba un programa en el cual se ingrese un valor límite positivo, y a continuación solicite números al usuario hasta que la suma de los números introducidos supere el límite inicial.
8. Escribir un programa que lea números enteros hasta teclear 0 (cero). Al finalizar el programa se debe mostrar el máximo número ingresado, el mínimo, y el promedio de todos ellos.

9. Realizar un programa que pida una frase y el programa deberá mostrar la frase con un espacio entre cada letra. La frase se mostrara así: H o l a. Nota: recordar el funcionamiento de la función Substring().
10. Escribir una función flecha que reciba una palabra y la devuelva al revés.
11. Escribir una función que reciba un String y devuelva la palabra más larga.
String Ejemplo: "Guia de JavaScript"
Resultado esperado : "JavaScript"
12. Escribir una función flecha de JavaScript que reciba un argumento y retorne el tipo de dato.
13. Crear un objeto persona, con las propiedades nombre, edad, sexo ('H' hombre, 'M' mujer, 'O' otro), peso y altura. A continuación, muestre las propiedades del objeto JavaScript.
14. Crear un objeto libro que contenga las siguientes propiedades: ISBN, Título, Autor, Número de páginas. Crear un método para cargar un libro pidiendo los datos al usuario y luego informar mediante otro método el número de ISBN, el título, el autor del libro y el numero de páginas.
15. Escribe un programa JavaScript para calcular el área y el perímetro de un objeto Círculo con la propiedad radio. **Nota:** Cree dos métodos para calcular el área y el perímetro. El radio del círculo lo proporcionará el usuario.
16. Realizar un programa que rellene dos vectores al mismo tiempo, con 5 valores aleatorios y los muestre por pantalla.
17. Realizar un programa que elimine los dos últimos elementos de un array. Mostrar el resultado
18. A partir del siguiente array: var valores = [true, 5, false, "hola", "adios", 2]:
a) Determinar cual de los dos elementos de texto es mayor
b) Utilizando exclusivamente los dos valores booleanos del array, determinar los operadores necesarios para obtener un resultado true y otro resultado false
c) Determinar el resultado de las cinco operaciones matemáticas realizadas con los dos elementos numéricos
19. Realizar un programa en Java donde se creen dos arreglos: el primero será un arreglo A de 50 números reales, y el segundo B, un arreglo de 20 números, también reales. El programa deberá inicializar el arreglo A con números aleatorios y mostrarlo por pantalla. Luego, el arreglo A se debe ordenar de menor a mayor y copiar los primeros 10 números ordenados al arreglo B de 20 elementos, y llenar los 10 últimos elementos con el valor 0.5. Mostrar los dos arreglos resultantes: el ordenado de 50 elementos y el combinado de 20.

20. Realizar un programa que obtenga la siguiente matriz [[3], [6], [9], [12], [15]] y devuelva y muestre el siguiente array [6, 9, 12, 15, 18].
21. Escribir un programa para obtener un array de las propiedades de un objeto Persona. Las propiedades son nombre, edad, sexo ('H' hombre, 'M' mujer, 'O' otro), peso y altura.
22. Escribir un programa de JavaScript que al clickear un botón muestre un mensaje a elección.
23. Resalte todas las palabras de más de 8 caracteres en el texto del párrafo (con un fondo amarillo, por ejemplo)

Hey, you're not **permitted** in there. It's **restricted**. You'll be **deactivated** for sure.. Don't call me a mindless **philosopher**, you **overweight** glob of grease! Now come out before somebody sees you. Secret mission? What plans? What are you talking about? I'm not getting in there! I'm going to regret this. There goes another one. Hold your fire. There are no life forms. It must have been **short-circuited**. That's funny, the damage doesn't look as bad from out here. Are you sure this things safe? Close up **formation**. You'd better let her loose. Almost there! I can't hold them! It's away! It's a hit! **Negative, Negative!** It didn't go in. It just impacted on the surface. Red Leader, we're right above you. Turn to point... oh-five, we'll cover for you. Stay there... I just lost my starboard engine. Get set to make your attack run. The Death Star plans are not in the main **computer**. Where are those **transmissions** you **intercepted**? What have you done with those plans? We **intercepted** no **transmissions**. Aaaah....This is a consular ship. Were on a **diplomatic** mission. If this is a consular **ship...were** is the **Ambassador?** Commander, tear this ship apart until you've found those plans and bring me the **Ambassador**. I want her alive! There she is! Set for stun! She'll be all right. Inform Lord Vader we have a **prisoner**. What a piece of junk. She'll make point five beyond the speed of light. She may not look like much, but she's got it where it counts, kid. I've added some special **modifications** myself. We're a little rushed, so if you'll hurry aboard we'll get out of here. Hello, sir.

24. Escribir un programa de JavaScript que a través de un formulario calcule el radio de un círculo y lo muestre en el HTML.
25. Escriba una función de JavaScript para obtener los valores de Nombre y Apellido del siguiente formulario.

```
<!DOCTYPE html>
<html><head>
<meta charset=utf-8 />
<title>Obtener nombre y apellido de form </title>
</head><body>
<form id="form1" onsubmit="getFormValores()">
Nombre: <input type="text" name="nombre" value="David"><br>
Apellido: <input type="text" name="apellido" value="Beckham"><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

CURSO DE PROGRAMACIÓN FULL STACK

SPRING FRAMEWORK

GUIA 1

VISTA - CONTROLADOR



Thymeleaf



EGG

INTRODUCCION

¡Entramos en la última etapa del curso! ¡Ahora podremos incorporar todo lo que venimos aprendiendo, y darle formato a una aplicación web funcional!

Veniamos trabajando con JAVA dentro del paradigma de programación orientado a objetos (POO) que es usado por miles de millones de dispositivos, desde computadoras hasta parquímetros, pero hoy en día, uno de sus mayores usos es para el **creciente mundo de la web**.

FUNDAMENTOS WEB

El éxito de la web se basa en dos factores fundamentales: el **protocolo HTTP** y el lenguaje de marcado HTML. El primero permite una implementación sencilla de un sistema de comunicaciones que permite enviar cualquier archivo de forma fácil, simplificando el funcionamiento del servidor y posibilitando que servidores poco potentes atiendan cientos o miles de peticiones y reduzcan de este modo los costes de despliegue. El segundo, el lenguaje HTML, proporciona un mecanismo sencillo y muy eficiente de creación de páginas enlazadas.

¿QUÉ ES EL PROTOCOLO HTTP?

Un protocolo es una PETICIÓN o SOLICITUD desde el cliente hacia un servidor.

El protocolo HTTP (Hypertext Transfer Protocol) es el protocolo principal de la World Wide Web. Es un protocolo simple, orientado a conexión y sin estado. Está orientado a conexión porque emplea para su funcionamiento un protocolo de comunicaciones (TCP, o Transport Control Protocol) de modo conectado, que establece un canal de comunicaciones entre el cliente y el servidor, por el cual pasan los bytes que constituyen los datos de la transferencia, en contraposición a los protocolos denominados de datagrama (o no orientados a conexión) que dividen la serie de datos en pequeños paquetes (o datagramas) antes de enviarlos, pudiendo llegar por diversas vías del servidor al cliente.

Explicado de manera más simple, cuando escribes una dirección web en tu navegador y se abre la página que deseas, es porque tu navegador se ha comunicado con el servidor web por HTTP. Dicho de otra manera, el protocolo HTTP es el código o lenguaje en el que el navegador le comunica al servidor qué página quiere visualizar.

HTTPS

Existe una variante de HTTP denominada HTTPS (S significa "secure", o "seguro") que utiliza el protocolo de seguridad SSL (o "Secure Socket Layer") para cifrar y autenticar el tráfico de datos, muy utilizada por los servidores web orientados al comercio electrónico o por aquellos que albergan información de tipo personal o confidencial.

Arquitectura Cliente - Servidor



¿CÓMO FUNCIONA HTTP?

De forma esquemática, el funcionamiento de HTTP es como sigue: el cliente establece una conexión TCP con el servidor, hacia el puerto por defecto para el protocolo HTTP (o el indicado expresamente en la conexión), envía una orden HTTP de solicitud de un recurso (añadiendo algunas cabeceras con información) y, utilizando la misma conexión, el servidor responde enviando los datos solicitados y, además, añadiendo algunas cabeceras con información.

La manera más fácil de explicar cómo funciona HTTP es describiendo cómo se abre una página web:

1. En la barra de direcciones del navegador, el usuario teclea example.com.
2. El navegador envía esa *solicitud*, es decir, **la petición HTTP**, al servidor web que administre el dominio example.com. Normalmente, la solicitud del cliente dice algo así como “Envíame este archivo”, pero también puede ser simplemente “¿Tienes este archivo?”.
3. El servidor web recibe la *solicitud HTTP*, busca el archivo en cuestión (en nuestro ejemplo, la página de inicio de example.com, que corresponde al archivo index.html) y el servidor envía una *respuesta*. En primer lugar envía una cabecera o header. Esta cabecera le comunica al cliente, mediante un *código de estado*, el resultado de la búsqueda.
4. Si se ha encontrado el archivo solicitado y el cliente ha solicitado recibirla (y no solo saber si existe), el servidor envía, tras el header, el message body o cuerpo del mensaje, es decir, el contenido solicitado: en nuestro ejemplo, el **archivo** index.html.
5. El navegador recibe el archivo y lo abre en forma de página web.

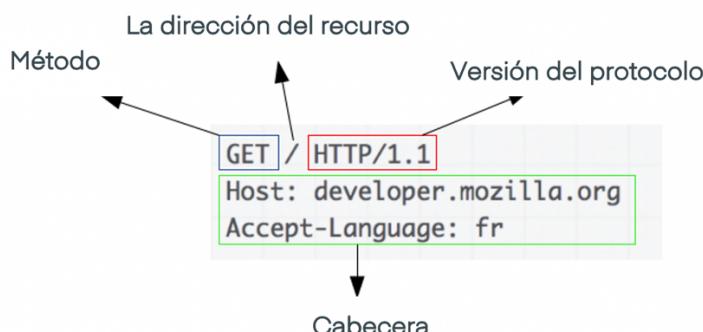
MENSAJES HTTP

Como vimos el servidor recibe un mensaje que es la petición HTTP del usuario y después envía una respuesta HTTP al cliente/navegador, en base a la petición.

Existen dos tipos de mensajes HTTP: peticiones y respuestas, cada uno sigue su propio formato.

Peticiones

Una petición de HTTP se ve de la siguiente manera:

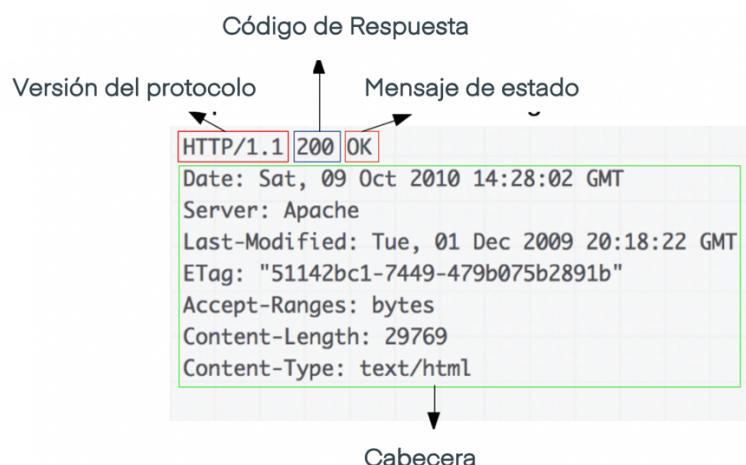


Una petición de HTTP está formada por los siguientes campos:

- **Un método HTTP**, normalmente pueden ser un verbo, como: **GET, POST** o un nombre como: **OPTIONS** (en-US) o **HEAD** (en-US), que defina la operación que el cliente quiera realizar. El objetivo de un cliente, suele ser una petición de recursos, usando **GET**, o presentar un valor de un formulario HTML, usando **POST**, aunque en otras ocasiones puede hacer otros tipos de peticiones.
- **La dirección del recurso pedido**; la URL del recurso, sin los elementos obvios por el contexto, como pueden ser: sin el protocolo (`http://`), el dominio (aquí `developer.mozilla.org`), o el puerto TCP (aquí el 80).
- **La versión del protocolo HTTP**.
- **Cabeceras HTTP**, opcionales, que pueden aportar información adicional a los servidores.

Respuestas

Un ejemplo de respuesta:



Las respuestas están formadas por los siguientes campos:

- La **versión del protocolo HTTP** que están usando.
- Un **código de respuesta**, indicando si la petición ha sido exitosa, o no, y debido a qué.
- Un **mensaje de estado**, una breve descripción del código de estado.
- **Cabeceras HTTP**, como las de las peticiones.

¿CUÁLES SON LOS MÉTODOS DE PETICIÓN?

En la web, los clientes, como un navegador, por ejemplo, se comunican con los distintos servidores web con ayuda del protocolo HTTP, el cual regula cómo ha de formular sus peticiones el cliente y cómo ha de responder el servidor. El protocolo HTTP emplea varios métodos de petición diferentes.

GET

GET es la madre de todas las peticiones de HTTP. Este método de petición existía ya en los inicios de la *world wide web* y se utiliza para **solicitar un recurso**, como un archivo HTML, **del servidor web**. Esto podría ser cuando un usuario clickea un link para ir a una pagina concreta.

Cuando escribes la dirección URL `www.ejemplo.com/test.html` en tu navegador, este se conecta con el servidor web y le envía una petición GET:

`GET/test.html`

El servidor enviaría el archivo `test.html` como respuesta.

PARTES DE UNA URL

A la petición GET puede añadirse **más información**, con la intención de que el servidor web también la procese. Estos llamados parámetros de URL se adjuntan a la dirección URL, la URL puede estar compuesta de varias partes, y las vamos a ver a continuación:

Ruta (Path)

Es lo que viene **después de la barra /**.

Normalmente indica páginas y subpáginas que podemos encontrar en un sitio web.

`www.ejemplo.com/otraPagina.html`

Parámetro (Query String)

Es lo que viene **después del signo de interrogación ?**. Todos los parámetros se componen de un nombre y un valor: “**Nombre=Valor**”.

En una URL puede haber varios parámetros. Y cuando es el caso, éstos se separan con el símbolo de **ampersand &**.

Los parámetros pueden indicar diferentes cosas. A veces tienen que ver con una **búsqueda en el sitio**, a veces son parámetros de campañas publicitarias, etc.

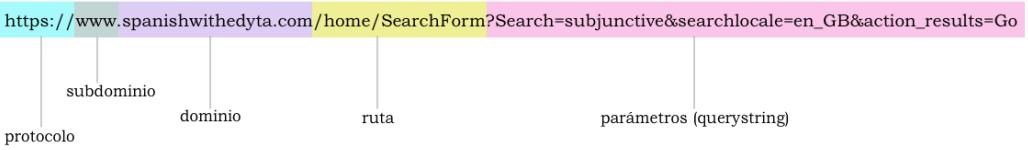
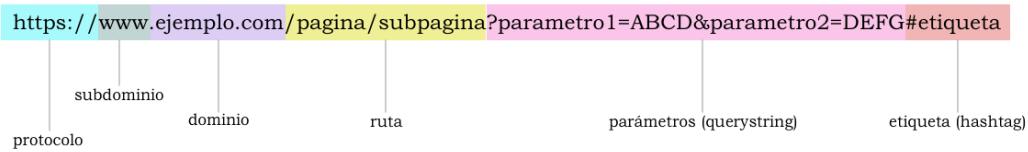
Veámoslo con este ejemplo: para buscar ciertas ofertas en la página web de una empresa de software, en la petición GET se indicará “Windows” como plataforma y “Office” como categoría:

`GET /search?platform=Windows&category=office`

Etiqueta

Las etiquetas en una URL aparecen después del hashtag #.

Su función, entre otras cosas, consiste en permitir hacer scroll hasta un elemento en concreto. Por ejemplo, si mandamos a alguien una URL que contenga una etiqueta, ésta le dirigirá a la parte exacta de la página en cuestión. Veamos una URL completa:



POST

Cuando se tienen que enviar al servidor web paquetes grandes de datos, como imágenes o datos de formulario de carácter privado, por ejemplo. El método GET se queda corto, porque todos los datos que se transmiten se escriben abiertos en la barra de direcciones del navegador.

En estos casos, se recurre al método POST. Este método no escribe el parámetro del URL en la dirección URL, sino que lo adjunta al encabezado HTTP.

Las peticiones POST suelen emplearse con **formularios digitales**. Abajo encontrarás el ejemplo de un formulario que recoge un nombre y una dirección de correo electrónico y lo envía al servidor por medio de POST:

```
<html>
<body>
<form action="/prueba" method="post">
Name: <input type="text" name="name"><br>
E-mail: <input type="text" name="email"><br>
<button type="submit">
</form>
</body>
</html>
```

¿CUÁNDO USAR UNO U OTRO?

El método POST es aconsejable cuando el usuario debe enviar datos o archivos al servidor, como, por ejemplo, cuando se llenan formularios o se suben fotos.

El método GET es adecuado para la personalización de páginas web: el usuario puede guardar búsquedas, configuraciones de filtros y ordenaciones de listas junto al URL como marcadores, de manera que en su próxima visita la página web se mostrará según sus preferencias.

A modo de resumen:

GET – Utilizado para obtener un recurso del servidor, identificado por una url. Para la configuración de páginas web (filtros, ordenación, búsquedas, links, etc.).

POST – Utilizado para la transferencia de información y datos al servidor. Puede utilizarse para enviar parámetros y su longitud es ilimitada.

CÓDIGOS DE RESPUESTA

Al iniciar el navegador (llamado cliente en este caso) se realiza una petición al servidor web, quien responde, a su vez, con un código de estado HTTP en forma de cadena de tres dígitos.

Con este mensaje, el servidor web le indica al navegador si su solicitud ha sido procesada correctamente, si ha ocurrido un error o si se necesita una autenticación. Como consecuencia, el código de estado HTTP se convierte en una parte esencial en la transmisión de mensajes de respuesta por parte del servidor web, que es insertado automáticamente en su encabezado. Por lo general, los usuarios se encuentran con páginas en formato HTML en vez de códigos de estado HTTP, cuando el servidor web no puede o no tiene permitido procesar la solicitud del cliente o no es posible realizar la transmisión de datos.

TIPOS DE RESPUESTA DE LOS CÓDIGOS DE ESTADO HTTP

En principio, los códigos de estado HTTP se dividen en cinco categorías diferentes, identificadas a su vez, por el primer dígito del código. Por ejemplo, el código de estado HTTP 200 forma parte del tipo de respuesta 2xx, el código 404 del tipo de respuesta 4xx. Esta clasificación se deriva principalmente de la importancia y la función de los códigos de estado, divididos principalmente en 5 tipos:

Códigos de estado 1xx – Información: Cuando se envía un código de estado HTTP 1xx, el servidor le notifica al cliente que la petición actual aún continúa. Esta clase reúne y proporciona información sobre el procesamiento y envío de una solicitud.

Códigos de estado 2xx – Éxito: Los códigos que comienzan con un 2 informan sobre una operación exitosa. Cuando se reciben este tipo de respuestas quiere decir que la solicitud del cliente fue recibida, comprendida y aceptada. Por lo general, el usuario solo percibe la web solicitada.

Códigos de estado 3xx – Redirecciones: Aquellos códigos que comienzan con 3 indican que la solicitud ha sido recibida por el servidor. Sin embargo, para asegurar un procesamiento exitoso es necesario que el cliente tome una acción adicional. Este tipo de códigos aparecen principalmente cuando hay redirecciones.

Códigos de estado 4xx – Errores del cliente: Cuando aparece un código 4xx quiere decir que se ha presentado un error de cliente. Esto quiere decir que el servidor ha recibido la solicitud, pero esta no se puede llevar a cabo. Una de las principales causas de este tipo de respuestas son las solicitudes defectuosas. Los usuarios de Internet son informados de este error por medio de una página HTML generada automáticamente.

Códigos de estado 5xx – Errores del servidor: El servidor indica un error propio cuando usa un código 5xx. Este tipo de respuestas indican que la solicitud correspondiente está temporalmente deshabilitada o es imposible de llevar a cabo. De nuevo, se generará automáticamente una página en formato HTML.

CÓDIGOS DE ESTADO HTTP MÁS IMPORTANTES

Los únicos códigos visibles para los visitantes son principalmente los códigos de error del cliente, como el 404 (Not Found), o de error del servidor como el 503 (Service Unavailable), ya que estos siempre se muestran automáticamente como páginas en formato HTML.

Pero ahora que vamos a trabajar sobre la creación de estas páginas web, va a ver códigos que nos van a informar de cosas a arreglar dentro de nuestro programa.

A continuación, presentamos una pequeña selección de los códigos de respuesta más comunes:

200 – OK, petición procesada correctamente.

301 – Indica al browser que visite otra dirección.

403 – Acceso prohibido, por falta de permisos.

404 – No encontrado, cuando el documento no existe.

500 – Error interno en el servidor.

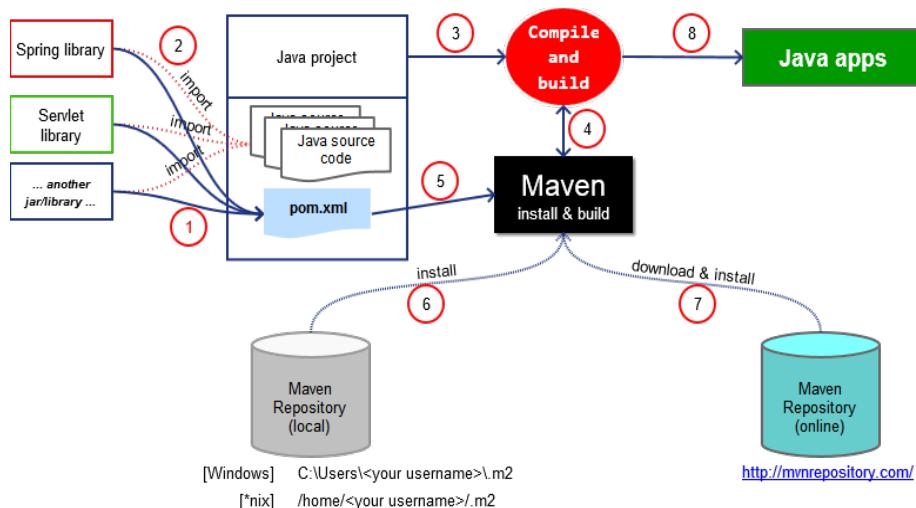
En el siguiente link podés ver todos los códigos de estado:

<https://uniwebsidad.com/tutoriales/los-codigos-de-estado-de-http?from=librosweb>

Hasta aquí vimos algunos conceptos generales del funcionamiento de los navegadores y servidores. Ahora debemos aprender a programar de tal manera de ser capaces de elaborar programas que utilicen el protocolo http y para ello debemos aprender MAVEN y SPRING:

¿QUÉ ES MAVEN?

Maven es una herramienta de software para la gestión y construcción de proyectos Java. Utiliza un Project Object Model (**POM**) para describir el proyecto de software a construir, sus **dependencias** de otros módulos y componentes externos, y el orden de construcción de los elementos. El modelo de configuración es simple y está basado en un formato XML (pom.xml). Además, Maven tiene objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado. La siguiente figura ilustra los pasos que lleva a cabo esta herramienta desde la importación de librerías hasta la generación de la aplicación Java.



DEPENDENCIAS

Uno de los puntos fuertes de Maven son las dependencias. En nuestro proyecto podemos decirle a Maven que necesitamos un jar (por ejemplo, *log4j* o el **conector de MySQL**) y maven es capaz de ir a internet, buscar esos jar y bajárselos automáticamente. Es más, si alguno de esos jar necesitara otros jar para funcionar, maven "tira del hilo" y va bajándose todos los jar que sean necesarios. Vamos a ver todo esto con un poco de detalle. Las dependencias se recopilan en el archivo **pom.xml**, dentro de una etiqueta **<dependencies>**.

Cuando ejecuta una compilación o ejecutamos un proyecto Maven, estas dependencias se resuelven y luego se cargan desde el repositorio local. Si no están presentes allí, Maven los descargará de un repositorio remoto y los almacenará en el repositorio local. También se le permite instalar manualmente las dependencias.

AÑADIR DEPENDENCIAS EN NUESTRO PROYECTO

Para indicarle a maven que necesitamos un jar determinado, debemos editar el fichero *pom.xml* que tenemos en el directorio raíz de nuestro proyecto. En el *pom.xml* generado por defecto por maven veremos un trozo como el siguiente:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Esta es una dependencia que pone maven automáticamente cuando creamos nuestro proyecto. Presupone que vamos a usar *junit* y en concreto, la versión 3.8.1. Si nosotros queremos añadir más dependencias, debemos poner más trozos como este. Por ejemplo, si añadimos la dependencia del conector de *mysql* versión 5.1.12, debemos poner lo siguiente:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId> // CAMBIAR POR LA DEPEN TH
    <version>5.1.12</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

MAVEN VS GRADLE

¿Qué es Gradle?

Gradle es una **herramienta de automatización de compilación del código abierto**, que obtuvo una rápida popularidad ya que fue diseñada fundamentalmente para construir multiproyectos, utilizando conceptos provenientes de Apache Maven.

Diferencias Principales

A continuación, destacamos las diferencias más importantes que existen entre Maven y Gradle.

Gradle	Maven
El tiempo de construcción de Gradle es corto y rápido	El rendimiento de Maven es lento en comparación con Gradle
Los scripts de Gradle son mucho más cortos y limpios	Los scripts de Maven son un poco largos en comparación con Gradle
Utiliza lenguaje específico de dominio (DSL)	Utiliza XML
Se basa en la tarea mediante la cual se realiza el trabajo	En Maven se definen objetivos vinculados al proyecto
Admite compilaciones incrementales de la clase java	No admite compilaciones incrementales
Soporte en la mayoría de las herramientas de Integración continua	Soporte en la mayoría de las herramientas de Integración continua

¿QUÉ ES SPRING FRAMEWORK?

Spring es un framework alternativo al stack de tecnologías estándar en aplicaciones JavaEE. Spring popularizó ideas como la inyección de dependencias o el uso de objetos convencionales (POJOs) como objetos de negocio.

Spring es el framework más popular para el desarrollo de aplicaciones empresariales en Java, para crear código de alto rendimiento, liviano y reutilizable. Su finalidad es estandarizar, agilizar, manejar y resolver los problemas que puedan ir surgiendo en el trayecto de la programación.

¿QUÉ ES UN FRAMEWORK?

Un **framework** es un entorno de trabajo que tiene como **objetivo facilitar la labor de programación** ofreciendo una serie de **características y funciones** que aceleran el proceso, reducen los errores, favorecen el trabajo colaborativo y consiguen obtener un producto de mayor calidad.

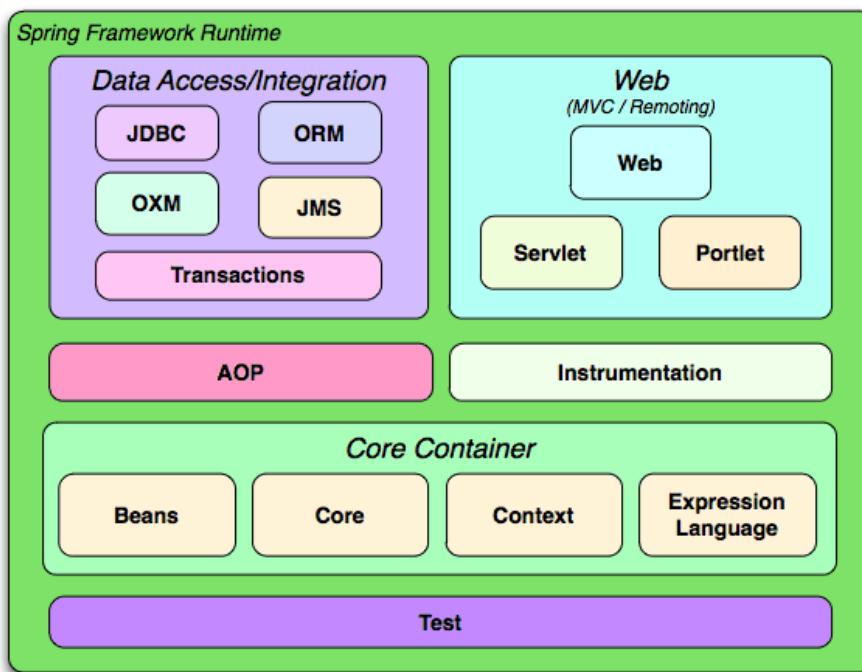
Los framework **ofrecen una estructura para el desarrollo** y no tienen que estar sujetos a un único lenguaje de programación, aunque es habitual encontrar en el mercado, distintos frameworks específicos para un lenguaje concreto.

SPRING FUNCIONALIDADES

Spring, ofrece como elemento clave la inyección de dependencias a nuestro proyecto, pero existen otras funcionalidades también muy útiles:

- **Core container:** proporciona inyección de dependencias e inversión de control.
- **Web:** nos permite crear controladores Web, tanto de vistas **MVC** como aplicaciones REST. Esto facilita en gran medida la programación basada en **MVC (Modelo Vista Controlador)**
- **Acceso a datos:** abstracciones sobre JDBC, ORMs como Hibernate, sistemas OXM (Object XML Mappers), JSM y transacciones.
- **Instrumentación:** proporciona soporte para la instrumentación de clases.
- **Pruebas de código:** contiene un framework de testing, con soporte para JUnit y TestNG y todo lo necesario para probar los mecanismos de Spring.

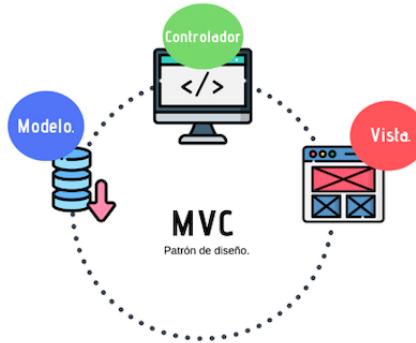
Estos módulos son opcionales, por lo que podemos utilizar los que necesitemos.



SPRING MVC

Antes de pasar a ver la inyección de dependencias, veremos otra funcionalidad, que es el Spring MVC.

Spring Web MVC es un sub-proyecto Spring que está dirigido a facilitar y optimizar el proceso creación de aplicaciones web utilizando el patrón **Modelo Vista Controlador**.



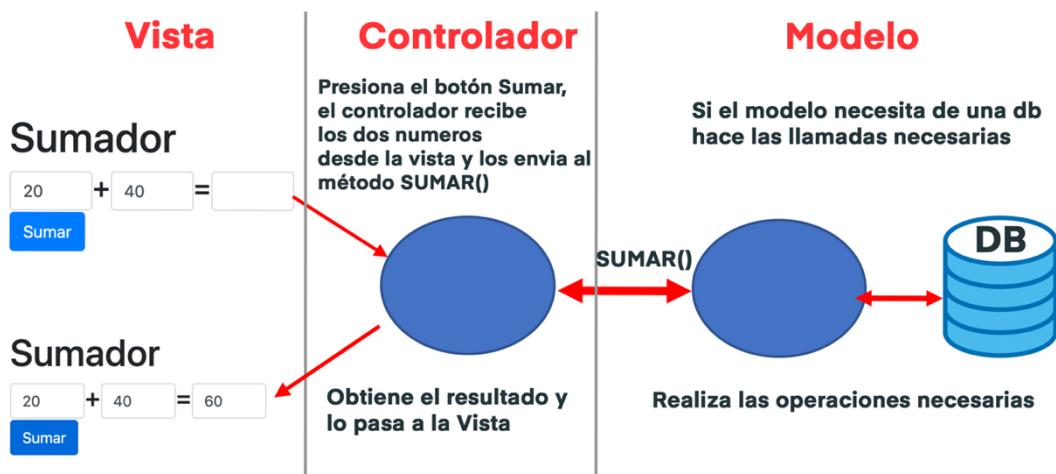
¿QUE ES EL PATRÓN DE DISEÑO MVC?

MVC es un **patrón de diseño** que se estructura mediante tres componentes: **modelo, vista y controlador**. Este patrón tiene como principio que cada uno de los componentes esté separado en diferentes objetos, esto significa que los componentes no se pueden combinar dentro de una misma clase. Sirve para clasificar la información, la lógica del sistema y la interfaz que se le presenta al usuario.

Modelo: Esta capa representa todo lo que tiene que ver con el acceso a datos: **guardar, actualizar, obtener datos**, además todo el código de la **lógica del negocio**, básicamente son las clases Java y parte de la lógica de negocio. No contiene ninguna lógica que describa como presentar los datos a un usuario.

Vista: este componente presenta los **datos del modelo al usuario**. La vista sabe cómo acceder a los datos del modelo, pero no sabe qué significa esta información o qué puede hacer el usuario para manipularla.

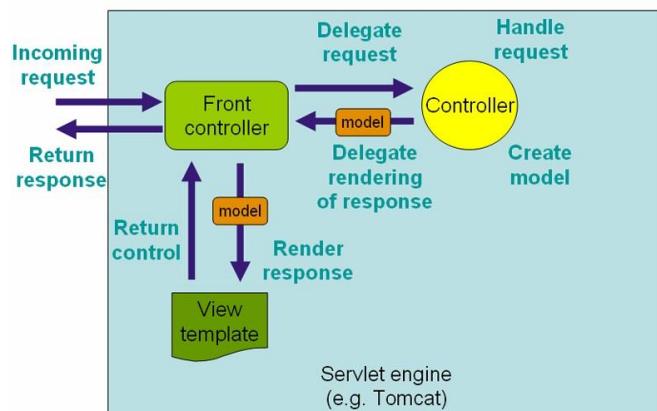
Controlador: este componente se encarga de gestionar las **instrucciones que se reciben, atenderlas y procesarlas**. El controlador es el encargado de **conectar el modelo con las vistas**, funciona como un puente entre la vista y el modelo, el controlador recibe eventos generados por el usuario desde las vistas y se encarga de dirigir al modelo la petición, recibir los resultados y entregarlos a la vista para que pueda mostrarlos.



PROCESAMIENTO DE UNA PETICIÓN EN SPRING MVC

Spring MVC se basa en este patrón de diseño para el manejo de las peticiones http y sus respuestas.

A continuación, se describe el flujo de procesamiento típico para una petición HTTP en Spring MVC. Spring es una implementación del patrón de diseño "front controller".



- Todas las peticiones HTTP se canalizan a través del *front controller*. En casi todos los frameworks MVC que siguen este patrón, el *front controller* no es más que un servlet cuya implementación es propia del framework. En el caso de Spring, la clase *DispatcherServlet*.
- El *front controller* averigua, normalmente a partir de la URL, a qué Controller hay que llamar para servir la petición. Para esto se usa un *HandlerMapping*.
- Se llama al *Controller*, que ejecuta la lógica de negocio, obtiene los resultados y los devuelve al *servlet*, encapsulados en un objeto del tipo *Model*. Además, se devolverá el nombre lógico de la vista a mostrar (normalmente devolviendo un String, como en JSF).
- Un *ViewResolver* se encarga de averiguar el nombre físico de la vista que se corresponde con el nombre lógico del paso anterior.
- Finalmente, el *front controller* (el *DispatcherServlet*) redirige la petición hacia la vista, que muestra los resultados de la operación realizada.

INYECCIÓN DE DEPENDENCIAS

La inyección de dependencias es quizás la característica más destacable del core de Spring Framework, que consiste que en lugar de que cada clase tenga que instanciar los objetos que necesita, **sea Spring el que inyecte esos objetos**, lo que quiere decir que es Spring el que creará los objetos y cuando una clase necesite usarlos se le pasarán (como cuando le pasas un parámetro a un método).

La **DI** (*Dependency Injector* o *Injector de Dependencias*) consiste en que en lugar de que sean las clases las encargadas de crear (instanciar) los objetos que van a usar (sus atributos), los objetos se inyectarán mediante los métodos setters o mediante el constructor en el momento en el que se cree la clase y cuando se quiera usar la clase en cuestión ya estará lista, en cambio sin usar DI la clase necesita crear los objetos que necesita cada vez que se use.

En Spring hay un Contenedor DI que es el encargado de inyectar a cada objeto los objetos que necesita (de los que depende) según se le indique ya sea en un archivo de configuración XML o mediante anotaciones.

Spring a estas clases que van a ser inyectadas por el contenedor, las llama **Spring Beans**.

¿QUE ES UN BEAN?

Un **Bean** es una clase de Java que debe cumplir los siguientes **requisitos**:

- Tener todos sus atributos privados (private).
- Tener métodos set() y get() públicos de los atributos privados que nos interese.
- Tener un constructor público por defecto.

A diferencia de los Bean convencionales que representan una clase, la particularidad de los Beans de Spring es que son objetos creados y manejados por el contenedor Spring.

CONTENEDOR SPRING

En Spring hay un Contenedor DI que es el encargado de inyectar a cada objeto los objetos que necesita (de los que depende) según se le indique ya sea en un archivo de **configuración XML o mediante anotaciones**. En el caso de Spring ese objeto es el contenedor IoC el cual es provisto por los módulos spring-core y spring-beans.

Spring se basa en el principio de **Inversión de Control (IoC)** o **Patrón Hollywood** («No nos llames, nosotros le llamaremos») consiste en:

- Un Contenedor que maneja objetos por vos, este contenedor es un archivo XML. Este archivo se llama **application-context.xml**.
- El contenedor generalmente controla la creación de estos objetos. Por decirlo de alguna manera, el contenedor hace los “new” de las clases java para que no los realices vos.
- El contenedor resuelve dependencias entre los objetos que contiene.

Un ejemplo típico para ver su utilidad es el de una clase que necesita una conexión a base de datos, sin DI si varios usuarios necesitan usar esta clase se tendrán que crear múltiples conexiones a la base de datos con la consiguiente posible pérdida de rendimiento, pero usando la inyección de dependencia, las dependencias de la clase (sus atributos), son instanciados una única vez cuando se despliega la aplicación y se comparten por todas las instancias de modo que una única conexión a base de datos es compartida por múltiples peticiones.

SPRING @CONFIGURATION

La anotación @Configuration se utiliza para la configuración basada en anotaciones de Spring. @Configuration es una anotación de marcador que indica que una clase declara uno o más beans, y puede ser procesada por el contenedor Spring para generar definiciones de beans y solicitudes de servicio para esos beans en tiempo de ejecución.

Por lo general, la clase que define el método Main es un buen candidato como principal @Configuration.

De todas formas, recomendamos utilizar la anotación @Configuration en una clase exclusiva de configuraciones, donde también tendremos las configuraciones de seguridad de nuestro proyecto.

```
import org.springframework.context.annotation.Configuration;
@Configuration
public class Configuraciones {  
}
```

ANOTACIONES

Ahora, cuando una clase está anotada con una de las siguientes anotaciones, Spring las registrará automáticamente en el application-context. Esto hace que la clase esté disponible para la inyección de dependencias en otras clases y esto se vuelve vital para construir nuestras aplicaciones. Estas anotaciones se conocen como **Spring Stereotypes** y se pueden encontrar en el paquete **org.springframework.stereotype**.

SPRING STEREOTYPES

Existen varios tipos de anotaciones Spring Stereotypes. Nos centraremos en las dos principales.

@Controller: Este estereotipo realiza las tareas de controlador y gestión de la comunicación entre el usuario y la aplicación. Para ello se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas. Donde se realiza la asignación de solicitudes desde la página de presentación, es decir, la capa de presentación (o Interface) no va a ningún otro archivo, va directamente a la clase **@Controller** y comprueba la ruta solicitada en la anotación **@RequestMapping** escrita antes de las llamadas al método si es necesario.

```
@Controller  
public class Controlador{}
```

Esta es una clase de controlador simple que contiene métodos para manejar peticiones HTTP para diferentes URLs.

@Autowired Esta anotación le indica a Spring dónde debe ocurrir una inyección. Si se lo coloca en un método, por ejemplo: setMovie, entiende (por el prefijo que establece la anotación @Autowired) que se necesita injectar un bean. Spring busca un bean de tipo Movie y, si lo encuentra, lo inyecta a este método. Sustituye la declaración de los atributos del bean en el xml. @Autowired se emplea para generar la inyección de dependencia de un tipo de Objeto que pertenece a una clase con la @Component(@Controller, @Service, @Repository)

```
@Autowired  
private final PeliculaServicio peliculaServicio;
```

SPRING MVC ANOTACIONES

También existen otras anotaciones que nos ayudarán con el manejo del patrón de diseño Spring MVC. Nos darán facilidades para la comunicación entre las vistas, el controlador y los modelos.

@Controller: esta anotación se repite en este apartado, ya que nos da la posibilidad de marcar a una clase como un controlador. Esta anotación se utiliza para crear una clase como controlador web, que puede manejar las solicitudes del cliente y enviar una respuesta al cliente.

@RequestMapping: La clase Controller contiene varios métodos para manejar diferentes peticiones HTTP, pero ¿cómo asigna Spring una petición en particular a un método del controlador en particular? Bueno, eso se hace con la ayuda de la anotación **@RequestMapping**. Es una anotación que se especifica sobre un método del controlador.

Proporciona el mapeo entre la **ruta de la petición** y el **método del controlador**. También admite alguna opción avanzada que se puede usar para especificar métodos de controlador separados para diferentes tipos de petición en el mismo URI como puede especificar un método para manejar una petición GET y otro para manejar la petición POST.

```

@Controller
public class Controlador{

@RequestMapping("/")
public String hola(){
    return "Hola Spring MVC";
}

}

```

En este ejemplo, la página de inicio se asignará a este método de controlador. Entonces, cualquier petición sobre la ruta localhost:8080 "/", irá a este método que devolverá "Hola Spring MVC".

@GetMapping: esta anotación se utiliza para asignar solicitudes HTTP GET a métodos de controlador específicos. **@GetMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.GET).

```

@Controller
public class Controlador{

@GetMapping("/")
public String hola(){
    return "Hola Spring MVC";
}

}

```

@PostMapping: esta anotación se utiliza para asignar solicitudes HTTP POST a métodos de controlador específicos. **@PostMapping** es una anotación compuesta que actúa como un acceso directo para **@RequestMapping** (method = RequestMethod.POST).

```

@Controller
public class Controlador{

@PostMapping("/guardar")
public String guardarUsuario(){
    return "Usuario Guardado ";
}

}

```

@RequestParam: esta es otra anotación Spring MVC útil que se usa para vincular los parámetros de una petición HTTP a los argumentos de un método controlador. Por ejemplo, si envía parámetros de un formulario junto con URL para guardar un usuario, el método puede obtenerlos como argumentos propios.

```

@GetMapping("/libro"){
public void mostrarDetalleLibro(@RequestParam("ISBN") String ISBN){
    System.out.println(ISBN);
}
}

```

Si accedes a tu aplicación web que proporciona detalles del libro con un parámetro de consulta(query string) como el siguiente:

`http://localhost:8080/libro?ISBN=900848893`

Entonces se llamará al método del controlador porque está vinculado a la URL "/libro" y el **parámetro de consulta ISBN** se usará para completar el **argumento del método** con el mismo nombre "ISBN" dentro del método **mostrarDetalleLibro()**.

De esa manera podemos obtener en nuestro controlador un dato que viaja a través de una URL, que va a venir de una petición HTTP.

@PathVariable: esta es otra anotación que se utiliza para recuperar datos de la URL. A diferencia de la anotación @RequestParam que se usa para extraer parámetros de consulta, esta anotación permite al controlador manejar una petición HTTP con URLs parametrizadas, estas serían URLs que tiene parámetros como parte de su ruta, por ejemplo:

```
http://localhost:8080/libro/900848893
```

Entonces para poder acceder a este detalle que se encuentra en la ruta de la URL, usariamos la anotación @PathVariable de la siguiente manera:

```
@GetMapping("/libro{ISBN}")  
public void mostrarDetalleLibro(@PathVariable("ISBN") String ISBN){  
    System.out.println(ISBN);  
}
```

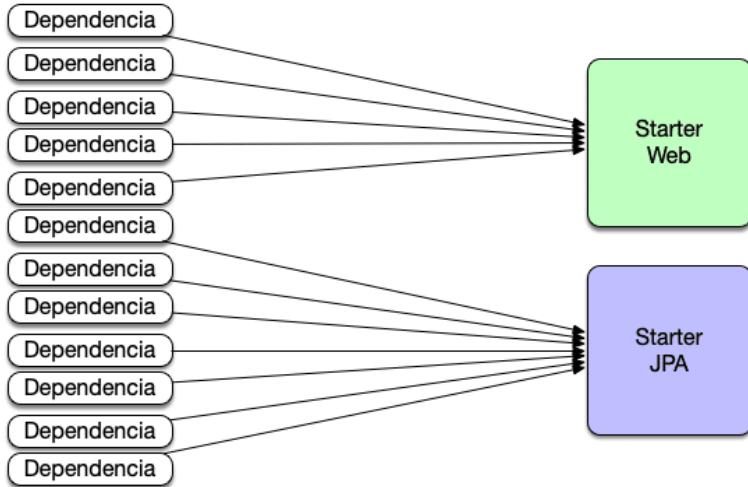
La variable Path o variable de ruta se representa entre llaves como {ISBN} en nuestra ruta de petición, lo que significa que la parte después de /libro se extrae y se completa en el ISBN del argumento del método, que está anotado con @PathVariable.

SPRING BOOT

Spring Boot es una de las tecnologías dentro del mundo de Spring de las que más se está hablando últimamente. **¿Qué es y cómo funciona Spring Boot?**. Para entender el concepto primero debemos reflexionar sobre cómo construimos aplicaciones con Spring Framework

- 1 seleccionar jars con maven
- 2 crear la aplicación
- 3 desplegar en servidor

Fundamentalmente existen tres pasos a realizar. El primero es crear un proyecto Maven/Gradle y descargar las dependencias necesarias. En segundo lugar desarrollamos la aplicación y en tercer lugar la desplegamos en un servidor. Si nos ponemos a pensar un poco a detalle en el tema, **únicamente el paso dos es una tarea de desarrollo**. Los otros pasos están más orientados a infraestructura. No deberíamos tener que estar eligiendo continuamente las dependencias y el servidor de despliegue.



SPRING INITIALIZER

SpringBoot nace con la intención de simplificar los pasos 1 y 3 y que nos podamos centrar en el desarrollo de nuestra aplicación. ¿Cómo funciona?. El enfoque es sencillo y lo entenderemos realizando un ejemplo. Para ello nos vamos a conectar a nuestro asistente de Boot que se denomina Spring Initializer.

The screenshot shows the Spring Initializer web interface. On the left, there's a sidebar for 'Project' (Maven Project selected), 'Language' (Java selected), and 'Spring Boot' (version 2.5.5 selected). Below that is 'Project Metadata' with fields for Group (com.ejemplospring), Artifact (EjemploGuia), Name (EjemploGuia), Description (Proyecto de ejemplo), Package name (com.ejemplospring.EjemploGuia), Packaging (Jar selected), Java version (8 selected), and Java 17 checkbox. On the right, under 'Dependencies', there are several sections: 'Spring Boot DevTools' (selected), 'Spring Web' (selected), 'Thymeleaf' (selected), 'MySQL Driver' (selected), and 'Spring Data JPA' (selected). At the bottom are buttons for 'GENERATE' (⌘ + ↵), 'EXPLORE' (CTRL + SPACE), and 'SHARE...'.

En este caso voy a construir una aplicación **Spring MVC** y elijo la dependencia web o **Spring Web**. Pulsamos generar proyecto y nos descargará un proyecto Maven en formato zip. Lo descomprimimos y lo abrimos en nuestro IDE, cuando lo vayamos a compilar, Maven se encargará de descargar todas las dependencias y sumarlas a nuestro proyecto.

Una vez que se termine de descargar nuestro proyecto Maven, se convertirá en proyecto Spring para poder trabajar, dentro encontraremos la clase **EjemploGuiaApplication**, se verá de la siguiente manera:

```

1. package com.ejemplospring;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5.
6. @SpringBootApplication
7. public class EjemploGuiaApplication{
8.
9.     public static void main(String[] args) {
10.         SpringApplication.run(EjemploGuiaApplication.class, args);
11.     }
12. }

```

Esta clase es la encargada de arrancar nuestra aplicación de Spring a diferencia de un enfoque clásico no hace falta desplegarla en un servidor web ya que Spring Boot provee de uno. Vamos a modificarla y añadir una anotación.

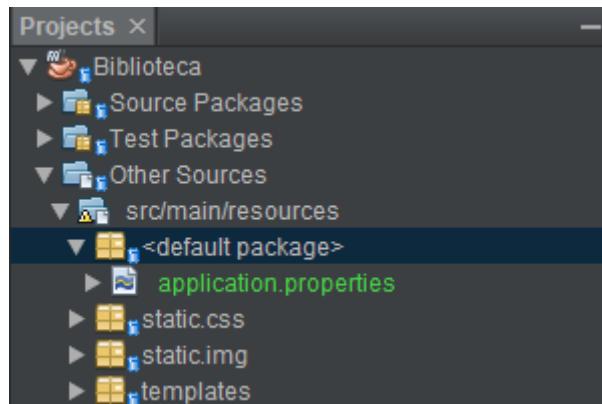
MANOS A LA OBRA

A continuación, invitamos a que creen su **primer proyecto Spring** desde Spring Initializr!

APPLICATION.PROPERTIES

Spring Framework trae incorporado un mecanismo para la configuración de aplicaciones usando un archivo llamado **application.properties**.

Este archivo se encuentra dentro de la carpeta `src/main/resources/<default package>`, como se muestra en la siguiente figura.



Este archivo nos permite ejecutar una aplicación en un entorno diferente.

En resumen, podemos usar el archivo `application.properties` principalmente con dos objetivos:

- Configurar el marco Spring Boot.
- Definir propiedades de configuración personalizadas para nuestra aplicación.

COMO EDITAR APPLICATION.PROPERTIES

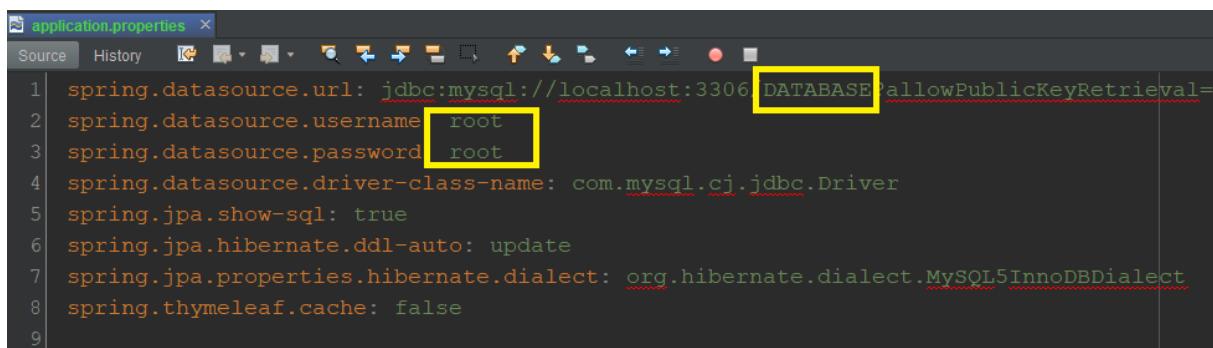
Abrimos el archivo application.properties, que inicialmente lo encontraremos vacío, y pegamos el siguiente texto dentro del mismo.

```
spring.datasource.url:jdbc:mysql://localhost:3306/DATABASE?allowPublicKeyRetrieval=true&useSSL=false&useTimezone=true&serverTimezone=GMT&characterEncoding=UTF-8
spring.datasource.username: root
spring.datasource.password: root
spring.datasource.driver-class-name: com.mysql.cj.jdbc.Driver
spring.jpa.show-sql: true
spring.jpa.hibernate.ddl-auto: update
spring.jpa.properties.hibernate.dialect: org.hibernate.dialect.MySQL5InnoDBDialect
spring.thymeleaf.cache: false
```



En la primer linea, podemos ver el nombre de la Base de Datos a la que nos vamos a conectar.

En las líneas 2 y 3 encontramos las credenciales de MySQL (usuario y contraseña).



Para más información sobre el resto de las configuraciones iniciales, te invitamos a investigar este link: <https://www.javatpoint.com/spring-boot-properties>.

SERVIDOR LOCAL

Levantar un servidor o tener un servidor a nuestra disposición no es algo fácil, ni barato. Por suerte Spring Boot nos deja, a través de **Apache Tomcat** y la clase que vimos previamente levantar un servidor local.

Tomcat nos permite hacer una conexión por red a si mismo, o escuchar a la espera de conexiones entrantes que se vayan a originar en el mismo dispositivo.

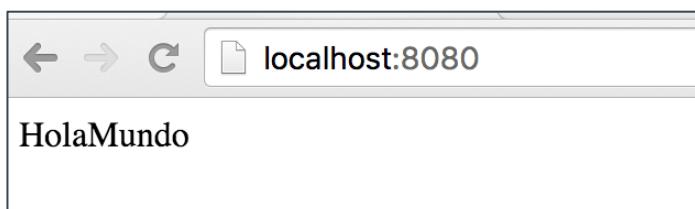
Se usa para desarrollo y pruebas: normalmente como desarrollador montas un servidor web (apache) y este escucha en el puerto 8080. Entonces lo que hace el desarrollador para probar las páginas web que está creando, o las aplicaciones web, o los APIs o servicios, es apuntar su navegador a <http://localhost/> o <http://localhost:8080> para hacer sus pruebas, cuando el puerto 80 se usa no se requiere especificar, solo cuando es un puerto diferente se tiene que poner con ":" después del nombre

Por lo que si tenemos la siguiente clase:

```
@Controller  
public class ControladorHola {  
    @GetMapping("/")  
    public String home() {  
        return "holaMundo";  
    }  
}
```

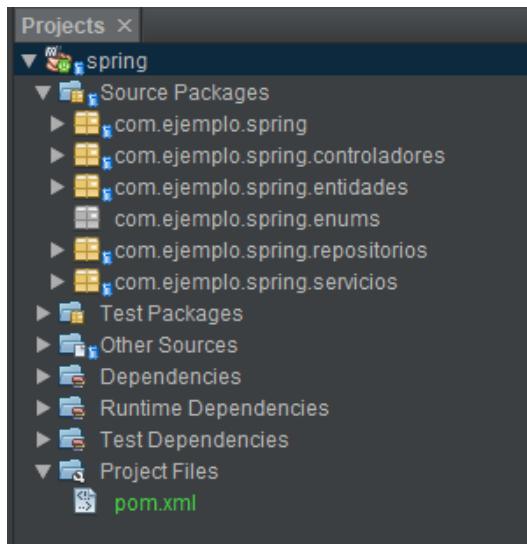
El controlador recibe la petición GET de HTTP con el GetMapping y usando el **return**, retorna como respuesta HTTP una pagina HTML como String, con el nombre holaMundo, que dentro tiene un <p>HolaMundo</p>, también por eso ponemos el método como String.

Entendido esto, vamos a nuestra clase EjemploGuiaApplication y corremos nuestro proyecto, se nos va a levantar un servidor local, por lo que si vamos a **localhost:8080**, nos encontraremos con la siguiente página:



PROGRAMACIÓN EN CAPAS

La programación por capas es un estilo de programación en el que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño.



A continuación, tendremos un modelo de Entidad, en base a la cual crearemos las capas del proyecto:

```
@Entity
public class Entidad {

    @Id
    @GeneratedValue(generator = "uuid")
    private String id;
    private String nombre;
    private Integer edad;

    @Temporal(TemporalType.DATE)
    private Date fecha;

    @Enumerated(EnumType.STRING)
    private Rol rol;
```

CLASE ENTITY

En esta clase podemos ver distintas anotaciones correspondientes a Spring y a JPA.

1. **@GeneratedValue** nos permite generar un id de forma automática.

La estrategia de generación generator="uuid" crea un valor String alfanumérico aleatorio. Esta estrategia es la recomendada a la hora de generar valores de identificadores únicos formato String.

En caso de querer generar un id tipo de dato numérico y autoincremental, podemos utilizar la anotación `@GeneratedValue(strategy = GenerationType.IDENTITY)`.

2. **@Temporal** nos permite mapear las fechas con la base de datos de una forma simple. (TemporalType.DATE) hace referencia al tipo de dato que trabajaremos desde la base de datos.
3. **@Enumerated** nos permite mapear un objeto de tipo Enum. La aclaración (EnumType.STRING) indica el tipo de dato que compone a ese Enum.

CAPA DE INTERFAZ (FRONT)

Esta capa resuelve la presentación de datos al usuario. Esta capa se encarga de “dibujar” las pantallas de la aplicación al usuario, y tomar los eventos que el cliente genere (por ejemplo, el hacer click en un botón).

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo HTML</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <div>Capa de Interfaz</div>
  </body>
</html>
```

CAPA DE COMUNICACIÓN

En esta capa están los controladores y es la capa responsable de mediar entre la interfaz de usuario y las capas inferiores. En esta capa contiene el dispatcher encargado de enrutar las peticiones, así como los controladores de acceso a los servicios web.

```
@Controller
@RequestMapping("/")
public class EntidadControlador {

    @GetMapping("/home")
    public String home (){
        return "index.html";
    }
}
```

CAPA DE SERVICIOS

Esta capa resuelve la lógica de la aplicación. Contiene los algoritmos, validaciones y coordinación necesaria para resolver la problemática. Los elementos fundamentales de esta capa son los objetos de dominio. Estos objetos representan los objetos principales del negocio. La lógica para manipular los objetos que representan los datos se encuentra en los llamados objetos de negocio (Service Object).

```

@Service
public class EntidadServicio {

    @Autowired
    private EntidadRepository entidadRepository;

    @Transactional()
    public Entidad guardar(String nombre,int edad, Rol rol) throws Exception {

        Entidad entidad = new Entidad();

        entidad.setNombre(nombre);
        entidad.setEdad(edad);
        entidad.setFecha(new Date());
        entidad.setRol(rol);

        // persisto el objeto en la base de datos con la funcion save de la clase Repository.
        return entidadRepository.save(entidad);
    }

}

```

- **@Transactional** es una anotación que debemos utilizar cada vez que vayamos a hacer una "transacción" en la DB. (una modificación persistente).
En caso de querer realizar simplemente una consulta, podemos mapear a través de la misma anotación, pero indicandole que será solo de lectura:
`@Transactional(readOnly=true)`.

CAPA DE ACCESO A DATOS (REPOSITORIOS)

Esta capa resuelve el acceso a datos, abstrayendo a su capa superior de la complejidad del acceso e interacción con los diferentes orígenes de datos. Esta capa se encarga de proveer un API simple de usar, orientado al negocio, sin exponer complejidades propias de un repositorio de datos.

En esta capa se resuelven:

- cualquier acceso a la base de datos
- cualquier acceso a filesystem
- cualquier acceso a otros sistemas
- cualquier acceso a un repositorio de datos en cualquier forma.

Con el soporte de Spring Data, la tarea repetitiva de crear las implementaciones concretas de DAO para nuestras clases de negocio se simplifica porque solo vamos a necesitar definir la interfaz.

Necesariamente hay que declarar la clase **Entidad** con la que trabajará el repositorio y el **tipo de dato del id** de la entidad nombrada.

```

@Repository
public interface EntidadRepository extends JpaRepository<Entidad, String>{

}

```

Spring Data nos provee de muchos métodos de consulta con sólo declarar esta clase. Algunos ejemplos son: count, delete, deleteAll, deleteAll, deleteAllById, deleteById, existsById, findById, save.

Recomendamos Investigar el siguiente link donde podrán encontrar las características de cada método según la documentacion oficial de Spring:

- <https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html>

Si los métodos anteriores no nos bastan, podemos declarar métodos de búsqueda propios en la interfaz, siguiendo una convención de nomenclatura jpql podremos realizar el siguiente tipo de consultas:

```
@Repository
public interface EntidadRepository extends JpaRepository<Entidad, String> {
    @Query("SELECT e FROM Entidad e WHERE e.nombre = :nombre")
    public Entidad buscarPorNombre(@Param("nombre") String nombre);
}
```

En esta @Query personalizada, podemos realizar una búsqueda a través de los parámetros señalados.

La anotación @Query es necesaria para que Spring sepa que tiene que evaluarlo como una consulta a base de datos.

El parámetro "**:nombre**" hace referencia al señalado en la anotación **@Param("nombre")**, seguido por el tipo de dato a evaluar.

THYMELEAF

Thymeleaf es un motor de plantillas, es decir, es una tecnología que nos va a permitir definir una plantilla y, junto con un modelo de datos, obtener un nuevo documento, sobre todo en entornos web.

Para saber más sobre Thymelaf, recomendamos meterse en su documentación de Thymeleaf:

- <https://www.thymeleaf.org/>

¿QUE ES EXACTAMENTE UN MOTOR DE PLANTILLAS?

El motor de plantillas (utilizado específicamente aquí para el desarrollo web) se genera para separar la interfaz de usuario (Vistas), de los datos comerciales (Modelos), puede generar documentos en un formato específico y el motor de plantillas para el sitio web generará un estándar Documento HTML.

Las plantillas, o más exactamente los motores de plantillas (templates engines) leen un fichero de texto, que contiene la presentación ya preparada en HTML, e inserta en él la información dinámica que le ordena el Controlador, la parte que une la vista con la información.

Veamos un ejemplo para ver las posibilidades de las plantillas, que no acaban, ni mucho menos, en la web. La sintaxis a utilizar depende del motor de plantillas utilizado, pero todos son muy similares. Los motores de plantillas suelen tener un pequeño lenguaje de script que permite generar código dinámico, como listas o cierto comportamiento condicional, pero esto también depende del lenguaje.

Este lenguaje de script es absolutamente mínimo, lo justo para posibilitar ese comportamiento dinámico:

```
<html>
<body>
Hola ${nombre}
</body>
</html>
```

Está claro que este ejemplo, que es una pequeña variación del famoso "Hola Mundo", es bastante simple. Lo que está sucediendo, al procesar este fichero, el motor de plantillas lo recorrerá, analizará y sustituirá esa "*etiqueta clave*" **`\${nombre}`** por el texto que le hallamos indicado, el nombre del visitante, por ejemplo, de forma que tengamos una presentación personalizada.

Básicamente, el motor de plantillas se encarga de recibir, una variable de tipo String llamada nombre, que se la va a enviar el controlador a la vista y la hará parte del HTML, haciéndolo dinámico. Por lo que los diferentes usuarios verán diferentes resultados.

VENTAJAS THYMELEAF

Permite realizar tareas que se conocen como **natural templating**. Es decir, como está basada en añadir atributos y etiquetas, sobre todo HTML, va a permitir que nuestras plantillas se puedan renderizar en local, y esa misma plantilla después utilizarla también para que sea procesada dentro del motor de plantillas. Por lo cual **las tareas de diseño y programación se pueden llevar conjuntamente**.

TIPOS DE EXPRESIONES

Permite trabajar con varios tipos de expresiones:

Expresiones variables: Son quizás las más utilizadas, como por ejemplo **`\${...}`**

Expresiones de selección: Son expresiones que nos permiten reducir la longitud de la expresión si prefijamos un objeto mediante una expresión variable, como por ejemplo ***{...}**

Expresiones de enlace: Nos permiten crear URL que pueden tener parámetros o variables, como por ejemplo **@{...}**

EXPRESIONES VARIABLES

Algunos ejemplos de expresiones variables son:

Podemos usar la notación de puntos para acceder a las propiedades de un objeto.

`\${sesión.usuario.nombre}`

Uno de los atributos que podemos usar es **th:text** con diferentes etiquetas HTML, para poder mostrar, por ejemplo, el nombre del autor de un libro. También podemos navegar entre objetos.

```
<span th:text="${libro.autor.nombre}">
```

También podemos llamar a métodos definidos en nuestros propios objetos, lo vamos a poder hacer desde las plantillas.

```
<td th:text="${myObject.myMethod()}">
```

ATRIBUTOS BÁSICOS

Los atributos básicos más conocidos con los que nos podemos encontrar son:

TH:TEXT

th:text: Permite reemplazar el texto de la etiqueta por el valor de la expresión que le demos.

```
<p th:text="${saludo}">saludo</p>
```

TH:EACH

th:each: Nos va a permitir repetir tantas veces como se indique o iterar sobre los elementos de una colección.

```
<li th:each="libro : ${libros}"  
th:text="${libro.titulo}">El Quijote</li>
```

La plantilla recibe la colección libros, y crea una variable llamada libro que va a ser en algún momento todos los elementos de nuestra colección, al igual que el for each de Java. Después, usamos la variable libro para acceder solo al título y al th:text para mostrar en el HTML el título en cuestión.

TH:VALUE

En la guía de HTML cuando estudiamos los inputs, aprendimos que a los inputs en algunos casos puede resultarnos interesante asignar un valor definido al campo en cuestión.

Este valor inicial del campo podía ser expresado mediante el atributo **value**. Thymeleaf nos ofrece hacer esto, pero de manera dinámica, con el atributo **th:value**, para darle a los inputs valores iniciales distintos, dependiendo de que envié el controlador.

```
<input type="text" name="instituto" th:value="${nombreInstituto}">
```

Esto nos haría pensar que es el mismo atributo que th:text, pero no, ya que th:text nos permite darle un valor a cualquier etiqueta de texto, mientras que th:value solo sirve para las etiquetas input.

TH:IF

A veces, vamos a necesitar que un **fragmento de nuestra plantilla** solo aparezca cuando se cumple una **determinada condición**.

Los atributos **th:if** y **th:unless**, nos permiten mostrar un elemento de HTML dependiendo del resultado de una determinada condición.

```
<td>  
  <span th:if="${persona.sexo == 'F'}">Femenino</span>  
  <span th:unless="${persona.sexo == 'M'}">Masculino</span>  
</td>
```

Si el valor de persona.sexo es igual a F, entonces el elemento span va a mostrar la palabra **Femenino**.

En cambio, si el valor es M, entonces el elemento muestra la palabra **Masculino**.

TH:HREF

Sirve para construir URLs que podemos utilizar en cualquier tipo de contexto.

Podríamos utilizarlas para hacer enlaces para URLs que sean absolutas o relativas al propio contexto de la aplicación, al servidor, al documento, etc.

Estos son unos ejemplos:

```
<a th:href="@{order/details}">...</a>
<a th:href="@{/documents/report}">...</a>
<a th:href="@{http://www.micom.es/index}">...</a>
```

MODELMAP

Ya vimos cómo, gracias a Thymeleaf podemos recibir del controlador una variable y mostrarla en nuestro HTML, pero, ¿cómo hacemos para enviar esa variable desde nuestro controlador a nuestro HTML?

Para resolver este problema vamos a utilizar el objeto **ModelMap**, este objeto es parte del paquete **org.springframework.ui.ModelMap**.

El objeto ModelMap tiene el método **addAttribute** que nos permite enviar variables nuestro HTML, la ventaja del objeto ModelMap, es que también nos permite enviar Colecciones a nuestro HTML.

El método addAttribute(String variable, Objeto nombreObjeto), recibe dos argumentos, una es variable de tipo String, que va ser el identificador que le vamos a poner al objeto o colección, que es el identificador con el que va a viajar al HTML y que va a tener que coincidir con la variable de Thymeleaf, y la otra es el objeto de Java que queremos mandar al HTML .

Pongamos un ejemplo, supongamos que tenemos la siguiente etiqueta en HTML con Thymeleaf, en el th:text, decimos que va a recibir una variable llamada nombre:

```
<p>Hola<span th:text="${nombre}"></p>
```

En el controlador tendremos el siguiente método:

```
@Controller
public class ControladorHola {
    @GetMapping("/")
    String home(ModelMap model) {
        String nombre = "Fernando"
        model.addAttribute("nombre", nombre)
        return "paginaHTML";
    }
}
```

Como podemos ver en el controlador, recibimos como argumento un ModelMap, esto es para que podemos recibir cualquier modelo que venga de la petición y para que podamos enviar los modelos que queramos como respuesta de x peticiones.

Usando model.addAttribute(), pasamos dos cosas, uno el identificador con el que el objeto va a viajar a la vista, que es “**nombre**”, recordemos que tiene que coincidir con la variable de Thymeleaf, y dos pasamos el objeto en sí.

Entonces cuando se llame a este controlador en localhost:8080/, se enviará el modelo “nombre”, lo recibirá la vista, gracias a Thymeleaf y mostrará el nombre Fernando.

MODEL Y MODELANDVIEW

Además del ModelMap, también podemos encontrarnos con Model o con Model And View. Si bien en el curso recomendamos utilizar ModelMap, a continuación, dejamos una breve descripción de estas herramientas.

MODEL

Es una Interfaz. Define un contenedor para los atributos del modelo y está diseñado principalmente para agregar atributos al modelo.

Ejemplo:

```
@GetMapping("/")
public String imprimirHola(Model model) {
    model.addAttribute("mensaje", "¡¡Hola mundo!!!");
    return "hola";
}
```

MODELANDVIEW

ModelAndView es un objeto que contiene tanto el modelo como la vista. El controlador devuelve el objeto ModelAndView y DispatcherServlet resuelve la vista utilizando View Resolvers y View.

La vista es un objeto que contiene el nombre de la vista en forma de cadena y el modelo es un mapa para agregar varios objetos.

Ejemplo:

```
@GetMapping("/")
public ModelAndView helloWorld() {
    String message = "Hello World!";
    return new ModelAndView("welcome", "message", message);
}
```

EJERCICIOS DE APRENDIZAJE

Para la realización de este trabajo práctico **se recomienda ver todos los videos de Spring**, de esta manera sabemos todos lo que tenemos que hacer, antes de empezar a hacerlo.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

SITIO DE NOTICIAS EGG NEWS

El objetivo de este ejercicio consiste en el desarrollo de un sistema web en Java utilizando una base de datos MySQL, JPA Repository para persistir objetos y Spring Boot como framework de desarrollo web.

Creación de la Base de Datos MySQL

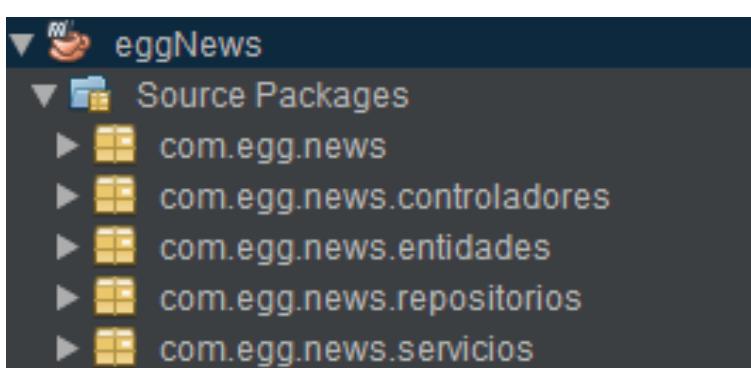
Crear el esquema sobre el cual operará el sistema de noticias. Para esto, en el IDE de base de datos que esté utilizando (por ejemplo, Workbench) ejecute la sentencia:

CREATE DATABASE noticia;

Paquetes del Proyecto

Los paquetes que se deben utilizar para el proyecto se deben estructurar de la siguiente manera:

- **vistas:** en este paquete se almacenarán aquellas clases que se utilizarán como vistas con el usuario.
- **controladores:** en este paquete se almacenarán aquellas clases que se utilizarán para mediar entre la vista con el usuario y las capas inferiores.
- **servicios:** en este paquete se almacenarán aquellas clases que llevarán adelante lógica del negocio.
- **repositorios:** en este paquete se crearán los repositorios que servirán como interfaces entre el modelo de objetos y la base de datos relacional.
- **entidades:** en este paquete se almacenarán aquellas clases que es necesario persistir en la base de datos.



Nota: Abajo explicamos en detalle cada capa.

Capa entidades

Spring utiliza una anotación para identificar aquellas clases que serán entidades y repositorios. Todas las entidades deben estar marcadas con la anotación @Entity y los repositorios con la anotación @Repository, los repositorios heredarán la interfaz JPRepository, que nos dará todos los métodos para persistir, editar, eliminar, etc.

Entidad Noticia

La entidad Noticia modela las noticias que se publicarán en la web. En esta entidad, el atributo “título” contiene el nombre con el cuál vamos a encontrar la noticia, mientras que el atributo “cuerpo” contiene toda la información que queremos que el usuario pueda leer.

Por otro lado, el atributo “foto” permite cargar una imagen identificatoria para cada noticia.

El repositorio que persiste a esta entidad (NoticiaRepository) debe contener los métodos necesarios para guardar/actualizar noticias en la base de datos, realizar consultas o dar de baja según corresponda.

Capa de Servicios

Spring utiliza una anotación para identificar aquellas clases que serán servicios. Todos los servicios deben estar marcados con la anotación @Service.

NoticiaServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar noticias (consulta, creación, modificación y dar de baja).

Capa de Comunicación

Spring utiliza una anotación para identificar aquellas clases que serán controladores.

Todos los controladores deben estar marcados con la anotación @Controller.

NoticiaControlador

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de noticias (guardar/modificar noticia, listar noticias, dar de baja, etc).

Capa de Vistas

Esta capa tiene la responsabilidad de llevar adelante las funcionalidades necesarias para interactuar con el usuario. Las vistas para este proyecto tienen que estar desarrolladas en HTML5 y se debe utilizar la biblioteca Thymeleaf y CSS para implementar las plantillas. Además, se debe utilizar el framework de Bootstrap para los componentes.

Se deben diseñar y crear todas las vistas web necesarias para llevar a cabo las siguientes funcionalidades:

- **Vista inicio:** en esta vista deben estar las tarjetas(bootstrap) con el título y la foto de cada noticia, ordenadas de más reciente a más antigua.
- **Vista noticia:** en esta vista tendremos el acceso a la noticia completa (cuerpo). Es la vista que se abre cuando hacemos click en alguna tarjeta de la vista inicio.
- **Vista panelAdmin:** en esta vista es donde gestionaremos las noticias. Aquí encontraremos los formularios necesarios para crear, modificar o eliminar una Noticia.

BIBLIOGRAFÍA

- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/protocolo-http/>
- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/http-request/>
- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/una-mirada-a-los-codigos-de-estado-http-mas-comunes/>
- <https://edytapukocz.com/url-partes-ejemplos-facil/>
- <https://howtodoinjava.com/maven/maven-dependency-management/>
- http://chuwiki.chuidiang.org/index.php?title=Dependencias_con_maven
- <https://www.arquitecturajava.com/que-es-spring-framework/>
- <https://programandointentandolo.com/2013/05/inyeccion-de-dependencias-en-spring.html>
- <https://proitcsolution.com.ve/inyeccion-de-dependencias-spring/>
- <https://www.java67.com/2019/04/top-10-spring-mvc-and-rest-annotations-examples-java.html>
- <https://www.danvega.dev/blog/2017/03/27/spring-stereotype-annotations/>
- <https://www.arquitecturajava.com/spring-stereotypes/>
- <https://www.arquitecturajava.com/que-es-spring-boot/>
- <https://openwebinars.net/blog/que-es-thymeleaf/>

CURSO DE PROGRAMACIÓN FULL STACK

SPRING FRAMEWORK GUIA 2

ROLES Y SEGURIDAD



 Thymeleaf



EGG

INTRODUCCIÓN

En la guía anterior aprendimos a trabajar con una arquitectura Modelo Vista Controlador (MVC) que separa los datos de una aplicación, la interfaz de usuario, ¡y la lógica de control en tres componentes distintos!

Ahora ya estamos de condiciones de darle más forma y aprender nuevas maneras de desarrollar una aplicación completa tanto a nivel lógico como funcional.

¡Es hora de interactuar con el Usuario! En esta instancia abordaremos las posibilidades que nos brindan las herramientas de seguridad web a la hora de construir una aplicación.

El registro de una cuenta de usuario, el inicio de sesión o las opciones 'administrador' son algunas de las funcionalidades que vamos a desarrollar en esta guía.

¡También aprenderemos a enviar correos electrónicos y a utilizar Lombok! una librería para Java que a través de anotaciones nos reduce el código que codificamos, es decir, nos ahorra tiempo y mejora la legibilidad del mismo.

SPRING SECURITY

Spring Security es el framework (marco de trabajo) encargado de gestionar todo lo relativo a la seguridad dentro de Spring/Spring Boot.

Algunas de las funciones/características principales las que se encarga son Spring Security son:

- Protocolos de seguridad.
- Roles para los accesos a los recursos o no.
- Autenticación y autorización con Spring Security.

Mediante a estos mecanismos vamos a intentar proteger la aplicación de una manera sencilla y sin afectar a la lógica de negocio.

AUTORIZACIÓN Y AUTENTICACIÓN BÁSICA

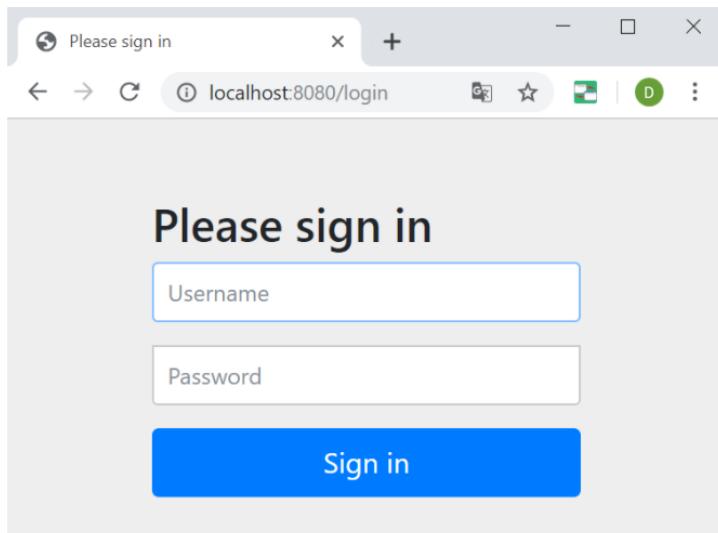
Autenticación: verificamos la identidad del usuario.

Autorización: tipo de permisos que tiene ese usuario.

Lo primero que haremos es incluir la dependencia de Spring Security en el archivo pom.xml del proyecto.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Si creamos un endpoint cualquiera e intentamos consumirlo, se mostrará un formulario de inicio de sesión proporcionado por Spring Security.



Si compruebas la salida por consola, se podrá ver una *contraseña generada automáticamente*. El nombre de usuario por defecto es «user».

Using generated security password: 30f20aec-fa73-4dee-a972-22f8ff77a1a5

Para crear una clase de seguridad personalizada, necesitamos usar `@EnableWebSecurity` y extender la clase con `@WebSecurityConfigurerAdapter` para que podamos redefinir algunos de los métodos proporcionados. Spring Security te fuerza a hashear las contraseñas para que no se guarden en texto plano. Para los siguientes ejemplos, vamos a usar `BCryptPasswordEncoder`.

Crearemos una clase "WebSecurity" donde ubicaremos la anotación `@Configuration`.

```
@Configuration  
@EnableWebSecurity  
@EnableGlobalMethodSecurity(prePostEnabled=true)  
public class WebSecurity extends WebSecurityConfigurerAdapter{  
  
    @Autowired  
    public UsuarioService usuarioService;
```

AUTENTICACIÓN

A continuación del código anterior, vamos a utilizar el método `configureGlobal` que recibe como parámetro `AuthenticationManagerBuilder auth` y tiene el método `userDetailsService`, además del encriptador de contraseñas.

```
@Autowired  
Public void configureGlobal(AuthenticationManagerBuilder auth) throws  
exception{  
    auth.userDetailsService(usuarioService).  
    passwordEncoder(new BCryptPasswordEncoder());  
}
```

ROLES Y AUTORIZACIÓN

Para comenzar, vamos a definir que roles vamos a manejar dentro de nuestra aplicación. Hay diversas maneras de crear los roles, pero en este caso como son roles muy puntuales los manejaremos desde la aplicación en un Enum.

```
public enum Rol {  
    ADMIN,  
    USER;  
}
```

Vamos a crear la clase Usuario con los atributos username, password y rol.

```
@Entity  
public class Usuario {  
  
    @Id  
    @GeneratedValue(generator = "uuid")  
    private String id;  
  
    private String username;  
    private String password;  
    private String email;  
  
    @Enumerated(EnumType.STRING)  
    private Rol rol;
```

Para obtener los usuarios, utilizaremos un Service que se conectará a un repositorio que implementa la interfaz JPA para acceder a nuestra base de datos.

Para que el Service funcione, tiene que estar anotado como @Service y debe implementar la interfaz *UserDetailsService*.

La interfaz *UserDetailsService* se utiliza para recuperar datos relacionados con el usuario. Tiene un método llamado *loadUserByUsername()* que se puede sobrescribir para personalizar el proceso de búsqueda del usuario.

En este caso, vamos a realizar acciones como buscar un Usuario por mail en la base de datos. Al implementar la interfaz *UserDetailsService*, el IDE nos pedirá que implementemos todos los métodos abstractos. De esta manera, se implementará el @Override del método *loadUserByUsername*.

```

@Service
public class UsuarioServicio implements UserDetailsService {

    @Autowired
    private UsuarioRepositorio usuarioRepositorio;

```

El repositorio debe ser una interfaz y debe implementar la interfaz JpaRepository.

```

@Repository
public interface UsuarioRepositorio extends JpaRepository<Usuario, String> {

    public Usuario buscarPorEmail(String email);

```

Veamos el metodo **loadUserByUsername** completo, prestando atención a las referencias del número de línea en el que se encuentra cada parte del código.

```

42    @Override
43    public UserDetails loadUserByUsername(String email)
44        throws UsernameNotFoundException {
45
46        Usuario user = usuarioRepositorio.buscarPorEmail(email);
47
48        if (user != null) {
49
50            List<GrantedAuthority> permissions = new ArrayList<>();
51
52            GrantedAuthority p = new SimpleGrantedAuthority("ROLE_" + user.getRol().toString());
53
54            permissions.add(p);
55
56            ServletRequestAttributes attr = (ServletRequestAttributes) RequestContextHolder.currentRequestAttributes();
57
58            HttpSession session = attr.getRequest().getSession(true);
59
60            session.setAttribute("usuario", user);
61
62            return new User(user.getEmail(), user.getPassword(), permissions);
63        }
64        return null;
65    }
66}

```

46- Instanciamos un objeto del tipo Usuario, haciendo uso del método buscarPorEmail(email) de la clase usuarioRepositorio.

48- Verificamos que el objeto Usuario no esté nulo.

50- Otorgaremos PERMISOS según el ROL que tenga cada usuario. Si el usuario existe, es decir, si la búsqueda por mail del repositorio nos retorna un Usuario, vamos a crear una lista de permisos llamada *permissions*

52- Creamos un objeto GrantedAuthority 'p' y concatenamos la palabra ROLE_ + el rol del usuario.

54- Agregamos el objeto 'p' a la lista de permisos.

56/58- Utilizamos los atributos que nos otorga el pedido al servlet, para poder guardar la información de nuestra HttpSession.

62- por último, retornamos un User con su email, contraseña y permisos!

RESTRINGIR ACCESOS

Por último, la idea sería que apliques tu propia lógica a la hora de restringir el acceso por roles a los usuarios. Para esto, en la clase **WebSecurity**, extendemos de *WebSecurityConfigurerAdapter*, y se sobreescribe el método *configure* para configurar la entidad http y con ello las url(endpoints) de la página web. En el siguiente ejemplo, sólo los usuarios **con rol ADMIN** podrán ingresar a la ruta ".../admin"

Del mismo modo, podemos configurar que a la ruta .../index puede ingresar cualquier rol existente.

```
public class WebSecurity extends WebSecurityConfigurerAdapter {  
  
    @Autowired  
  
    public UsuarioServicio usuarioServicio;  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
  
        http  
            .authorizeRequests()  
            .antMatchers("/admin").hasRole("ROLE_ADMIN")  
            .antMatchers("/index").hasAnyRole()  
            .and()  
            .formLogin();  
  
    }  
}
```

LOMBOK

A pesar de que NetBeans hace el trabajo por ti cuando creas tus clases entidad, da pereza generar los getters, setters, equals, hashCode, toString... sobre todo cuando tu modelo de datos es grande.

Todo eso nos lo podemos evitar, ya que, **si utilizamos Project Lombok, lo va a hacer por nosotros.**

Lombok es un procesador de anotaciones Java que nos permite agilizar el proceso de desarrollo de una aplicación de software, se ejecuta en tiempo de compilación, durante este proceso crea e inyecta automáticamente el código determinado por la correspondiente anotación Java.

INSTALACION LOMBOK

Para instalar Project Lombok, lo primero que tenemos que hacer es descargar el fichero lombok.jar desde la web de Project Lombok: <https://projectlombok.org/>

Una vez descargado, ejecutamos el mismo según el entorno en el que trabajemos, y seguimos el asistente que se nos propone.

Lo primero que hará el instalador será escanear los posibles IDEs que tengamos instalados en el sistema. Si no encontrara ninguno, lo que tenemos que hacer es buscarlo de forma manual, seleccionando **Specify location...** y pulsar en **Install/Update** para instalar Lombok.

AGREGANDO LA DEPENDENCIA

Project Lombok es una dependencia que tenemos que incluir en nuestro proyecto para que podamos añadir una serie de anotaciones que personalizaran nuestro IDE.

```
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.10</version>
    </dependency>
</dependencies>
```

En el siguiente ejemplo, a la izquierda vemos la clase Person con sus correspondientes campos privados y sus métodos get y set, en el lado derecho vemos la misma clase utilizando lombok, solamente agregando las anotaciones `@Getter` y `@Setter` le indicamos que deseamos generar los métodos get y set para todos los campos de la clase Person.

```
public class Person {
    private String nombre;
    private String apellido;
    private Integer edad;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getApellido() {
        return apellido;
    }

    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

    public Integer getEdad() {
        return edad;
    }

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

lombok

```
@Getter @Setter
public class Person {

    private String nombre;
    private String apellido;
    private Integer edad;
}
```

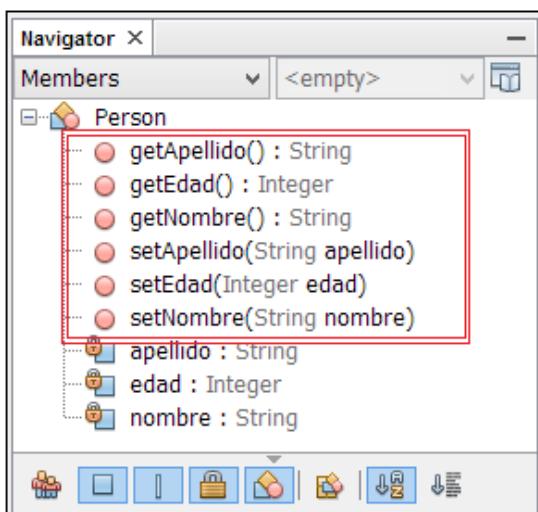
ANOTACIONES LOMBOK

@Getter y @Setter

Estas anotaciones son usadas para generar los métodos getter y setter de manera automática.

```
import lombok.Getter;  
import lombok.Setter;  
@Getter @Setter  
public class Person {  
    private String nombre;  
    private String apellido;  
    private Integer edad;  
}
```

Para asegurarnos de que funciona correctamente nos vamos al menú principal de NetBeans y pulsamos la opción: Window | Navigator, esto nos permite mostrar una ventana donde se observa la clase que se está editando con todos los elementos de la misma.



@ToString

Esta anotación la utilizaremos para generar el método `toString()`, la aplicamos a nivel de clase y cuenta con las siguientes configuraciones:

Por defecto la cadena generada corresponde al nombre de la clase seguido de una lista de los campos y su valor actual separados por coma.

```
@Getter @Setter
```

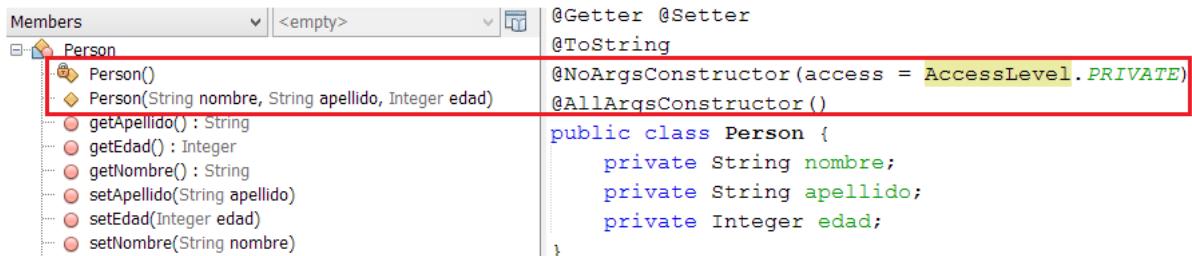
@ToString

```
public class Person {  
    private String nombre;  
    private String apellido;  
    private Integer edad;  
}
```

```
PERSON: Person(nombre=Leo, apellido=Amadeus, edad=null)
-----
BUILD SUCCESS
```

@NoArgsConstructorConstructor y @AllArgsConstructorConstructor

Usaremos estas anotaciones para generar el constructor de la clase, la primera crea un constructor sin argumentos y el segundo un constructor que permite inicializar cada uno de los campos de la clase.



Controlamos el nivel de acceso con access = AccessLevel.*., en nuestro ejemplo el constructor sin argumentos es privado, si no indicamos el AccessLevel será público como el segundo constructor generado con @AllArgsConstructorConstructor.

@Data

Esta anotación es **una combinación de todas las anotaciones que vimos anteriormente** en este tutorial con excepción de @AllArgsConstructorConstructor, es muy útil cuando trabajamos con herramientas como el ORM Hibernate.

@Data

```
public class Person {  
    private String nombre;  
    private String apellido;  
    private Integer edad;  
}
```

Existen muchas otras anotaciones. En la web <https://projectlombok.org/> verás varios ejemplos de cómo utilizarlas.

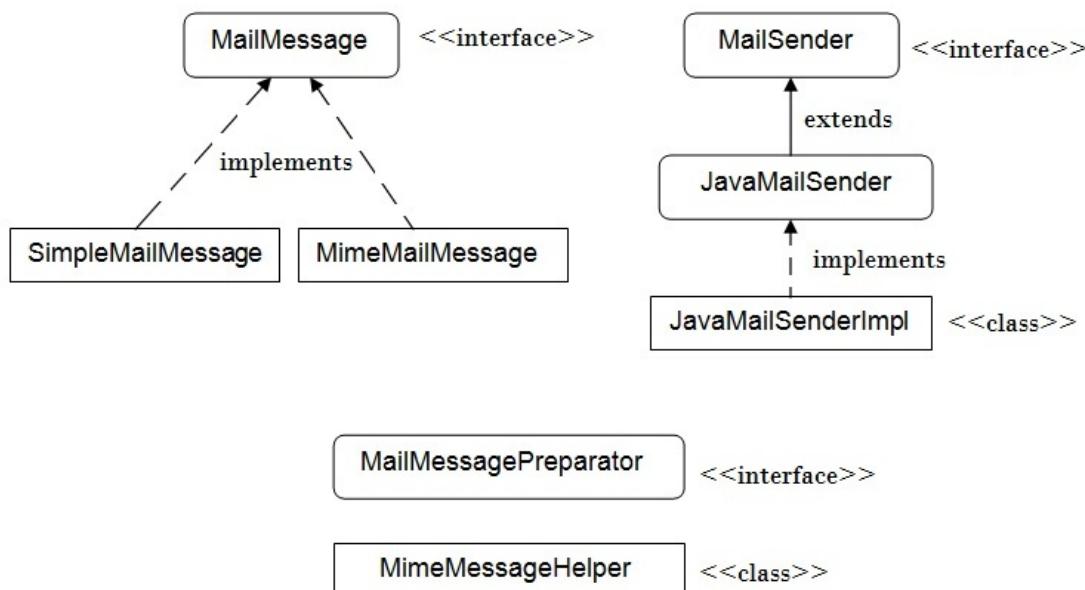
JAVA MAIL SENDER

Spring framework proporciona muchas interfaces y clases útiles para enviar y recibir correos.

El paquete **org.springframework.mail** es el paquete raíz que proporciona soporte de correo electrónico.

API DE CORREO SPRING JAVA

Las interfaces y clases para el soporte de correo de Java en Spring Framework son las siguientes:



- **Interfaz MailSender:** Es la interfaz raíz. Proporciona la funcionalidad básica para enviar correos electrónicos simples.
- **Interfaz JavaMailSender:** Es la subinterfaz de MailSender. Soporta mensajes MIME. Se usa principalmente con la clase **MimeMessageHelper** para la creación de JavaMail **MimeMessage**, que nos permite enviar correos con archivos adjuntos, etc. Spring Framework recomienda el mecanismo **MimeMessagePreparator** para usar esta interfaz.
- **Clase JavaMailSenderImpl:** proporciona la implementación de la interfaz JavaMailSender. Es compatible con JavaMail MimeMessages y Spring SimpleMailMessages.
- **Clase SimpleMailMessage:** se utiliza para crear un mensaje de correo simple que incluye: de, para, cc, asunto y texto.
- **Interfaz MimeMessagePreparator:** es la interfaz de devolución de llamada para la preparación de mensajes JavaMail MIME.
- **Clase MimeMessageHelper:** es la clase auxiliar para crear un mensaje MIME. Ofrece soporte para elementos en línea como imágenes, archivos adjuntos de correo típicos y contenido de texto HTML.

1. DECLARAR DEPENDENCIA PARA SPRING BOOT MAIL

Lo primero que vamos a hacer es seleccionar la dependencia Spring Boot Starter Mail a la hora de crear nuestro proyecto; o agregarla al archivo pom.xml si queremos incluirla en un proyecto existente:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

2. CONFIGURAR PROPIEDADES DE CORREO

Para enviar correos electrónicos desde la aplicación, debemos configurar los ajustes del servidor SMTP en el archivo de configuración de la aplicación Spring Boot (**application.properties**) de la siguiente manera:

```
spring.mail.host=smtp.gmail.com //WEVO VERDE → ESTA CONFIG ES PARA GMAIL//
spring.mail.port=8080
spring.mail.username=direccionDeCorreo
spring.mail.password=contraseñaDelCorreo
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

3. CONFIGURAR UN JAVAMAILSENDER

Spring Mail proporciona JavaMailSender, que es la interfaz clave que define métodos comunes para enviar correos electrónicos. Después de configurar las propiedades del correo en el archivo application.properties, vamos a indicarle a Spring Framework que inyecte la implementación predeterminada de JavaMailSender en una clase Controlador:

```
@Controller
public class AppController {
    @Autowired
    private JavaMailSender mailSender;
    public void sendEmail() {
        // use mailSender here...
    }
}
```

También en la clase de servicio:

```
@Service  
Public class BusinessService{  
  
    @Autowired  
    private JavaMailSender mailSender;  
  
    public void sendEmail(){  
        //usar mailSender acá...  
    }  
}
```

4. EJEMPLO DE CÓDIGO PARA ENVIAR UN CORREO ELECTRÓNICO SIMPLE (TEXTO SIN FORMATO)

Necesitamos los siguiente datos:

```
String from = "sender@gmail.com";//dirección de correo que hace el envío.  
String to = "recipient@gmail.com";//dirección de correo que recibe el mail.
```

El método sendEmail() se vería así:

```
public void sendEmail(String from, String to) {  
  
    SimpleMailMessage message = new SimpleMailMessage();  
    message.setFrom(from);  
    message.setTo(to);  
    message.setSubject("Asunto del correo");  
    message.setText("Este es un correo automático!");  
    mailSender.send(message); //método Send(envio), propio de Java Mail Sender.  
  
}
```

EJERCICIOS DE APRENDIZAJE

Para la realización de este trabajo práctico **se recomienda ver todos los videos de Spring**, de esta manera sabemos todos lo que tenemos que hacer, antes de empezar a hacerlo.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

SITIO DE NOTICIAS EGG NEWS PARTE 2

El cliente nos envió nuevos requerimientos para la aplicación web que desarrollamos en la guía anterior. Es por esto que vamos a continuar trabajando sobre el mismo proyecto y usando la misma base de datos para completar nuestro EggNews.

Además de las entidades que ya teníamos, crearemos tres nuevas entidades: Usuario, Periodista y Administrador.

Entidad Usuario

La entidad Usuario modela los usuarios que se registran y loguean en la web. Esta entidad debe poseer los siguientes atributos:

- String nombreUsuario
- String password
- Date fecha de alta.
- Rol rol
- Boolean activo

Un Usuario debe loguearse para ver las noticias, y perderá la posibilidad de ingresar a la vista **panelAdmin**.

Las entidades Periodista y Administrador **deben extender** de Usuario.

Entidad Periodista

La entidad Periodista tendrá como atributos extras:

- ArrayList<Noticia> misNoticias
- Integer sueldoMensual.

Un Periodista debe loguearse en el sistema para poder acceder a **crear y modificar** Noticias. Esta acción deberá realizarse desde la vista **panelAdmin**.

Entidad Administrador

La entidad Administrador podrá:

- Podrá crear, modificar y **eliminar** noticias
- Dar de alta o baja a Periodistas (modificar el atributo activo).
- Indicar cuál va a ser el sueldoMensual de cada Periodista.

Un Administrador debe loguearse en el sistema para poder acceder a sus funcionalidades.

Entidad Noticia

Agregaremos un nuevo atributo Periodista a la entidad Noticia que se va a relacionar con la entidad periodista:

- Periodista **creador;** (relación).

EJERCICIOS EXTRAS

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por último, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

SISTEMA DE ESTANCIAS EN EL EXTRANJERO WEB

El objetivo de este ejercicio consiste en el desarrollo de un sistema web para una pequeña empresa que se dedica a organizar estancias en el extranjero dentro de una familia. El sistema debe registrar la reserva de casas por parte de los clientes que desean realizar alguna estancia. El objetivo es el desarrollo de un sistema web de reservas de casas para realizar estancias en el exterior, utilizando el lenguaje JAVA, una base de datos MySQL, el framework de persistencia JPA y Spring Boot como framework de desarrollo web.

Creación de la Base de Datos MySQL

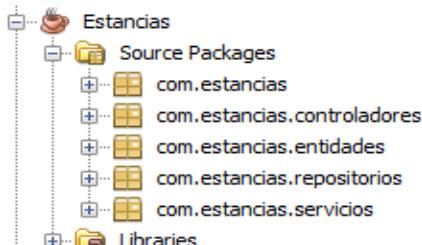
Crear el esquema sobre el cual operará el sistema de reservas de casas. Para esto, en el IDE de base de datos que esté utilizando (por ejemplo, Workbench) se debe ejecutar la sentencia:

```
CREATE DATABASE estancias;
```

De esta manera se creará una base de datos vacía llamada estancias.

Paquetes del Proyecto

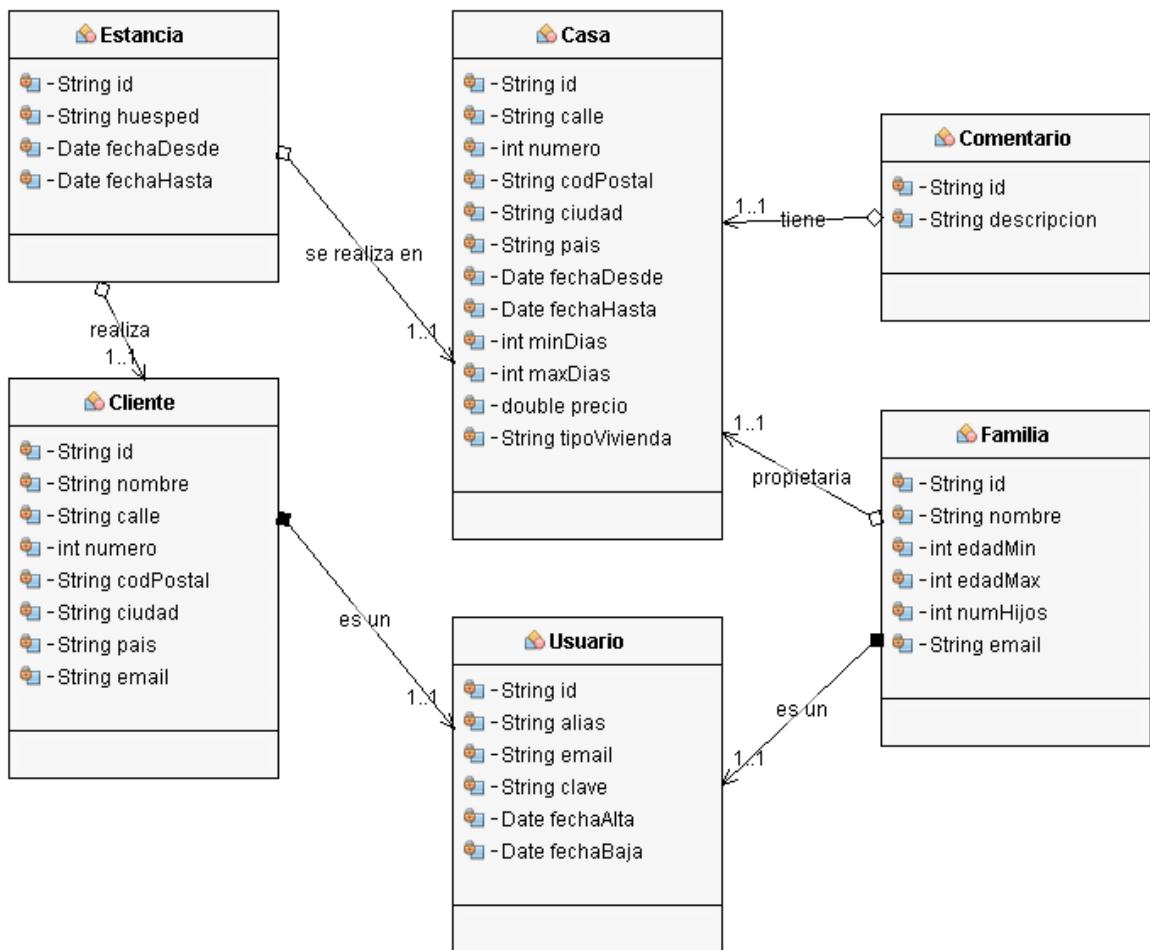
En este proyecto se debe utilizar la misma estructura de capas que en el ejercicio anterior.



Capa de Datos

Entidades y Repositorios

Crear el siguiente modelo de entidades y agregar los repositorios correspondientes. Todas las entidades deben estar marcadas con la anotación `@Entity` y los repositorios con la anotación `@Repository`.



Entidad Usuario

La entidad usuario modela los datos de un usuario que accede al sistema para registrarse como familia y ofrecer una habitación de su casa para estancias, o bien, como un cliente que necesita realizar una reserva. De cada usuario se debe registrar el nombre de usuario (alias), el correo electrónico, el password y la fecha de alta. El repositorio que persiste a esta entidad (`UsuarioRepository`) debe contener los métodos necesarios para registrar el usuario en la base de datos, realizar consultas y eliminar.

Entidad Familia

La entidad familia modela las familias que habitan en diferentes países y que ofrecen alguna de las habitaciones de su hogar para acoger a algún chico (por un módico precio). De cada una de estas familias se conoce el nombre, la edad mínima y máxima de sus hijos, número de hijos y correo electrónico. El repositorio que persiste a esta entidad (`FamiliaRepository`) debe contener los métodos necesarios para guardar/actualizar los datos de las familias en la base de datos, realizar consultas y eliminar o dar de baja según corresponda.

Entidad Casa

La entidad casa modela los datos de las casas donde las familias ofrecen alguna habitación. De cada una de las casas se almacena la dirección (calle, numero, código postal, ciudad y país), el periodo de disponibilidad de la casa (fecha_desde, fecha_hasta), la cantidad de días mínimo de estancia y la cantidad máxima de días, el precio de la habitación por día y el tipo de vivienda. El repositorio que persiste a esta entidad (`CasaRepository`) debe contener los métodos necesarios para guardar/actualizar los datos de una vivienda, realizar consultas y eliminar.

Entidad Cliente

La entidad cliente modela información de los clientes que desean mandar a sus hijos a alguna de las casas de las familias. Esta entidad es modelada por el nombre del cliente, dirección (calle, numero, código postal, ciudad y país) y su correo electrónico. El repositorio que persiste a esta entidad (ClienteRepositorio) debe contener los métodos necesarios para guardar/actualizar los datos de un cliente, realizar consultas y eliminar.

Entidad Reserva

La entidad reserva modela los datos de las reservas y estancias realizadas por alguno de los clientes. Cada estancia o reserva la realiza un cliente, y además, el cliente puede reservar varias habitaciones al mismo tiempo (por ejemplo para varios de sus hijos), para un periodo determinado (fecha_llegada, fecha_salida). El repositorio que persiste a esta entidad (ReservaRepositorio) debe contener los métodos necesarios para realizar una reserva, actualizar los datos (por ejemplo, fecha de la reserva), realizar consultas de las reservas realizadas para una determinada vivienda y eliminar reserva.

Entidad Comentario

La entidad comentario permite almacenar información brindada por los clientes sobre las casas en las que ya han estado. El repositorio que persiste a esta entidad (ComentarioRepositorio) debe contener los métodos necesarios para guardar los comentarios que realizan los clientes sobre una determinada una vivienda.

Capa de Servicios

Utiliza la anotación @Service para identificar aquellas clases que serán servicios. Todos los servicios deben estar marcados con esta anotación.

UsuarioServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar usuarios (alta de usuario, consultas, y baja o eliminación).

FamiliaServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar familias (creación, consulta, modificación y eliminación).

CasaServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar las casas (creación, consulta, modificación y eliminación).

ClienteServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar clientes (creación, consulta, modificación y eliminación).

ReservaServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para realizar las reservas de viviendas (reservar, consultar reservas realizadas, modificación y eliminación).

Capa de Comunicación

Spring utiliza una anotación para identificar aquellas clases que serán controladores. Todos los controladores deben estar marcadas con la anotación @Controller. Algunos de los controladores a desarrollar son los siguientes.

UsuarioController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de usuarios (dar de alta un usuario, cambiar clave, listar usuarios registrados, dar de baja un usuario).

FamiliaController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de familias (guardar/modificar datos de la familia, listar familias, eliminar).

CasaController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de viviendas (guardar/modificar datos de la casa, listar viviendas, eliminar).

ClienteController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista del usuario diseñada para la gestión de clientes (guardar/modificar, listar clientes, eliminar).

ReservaController

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para operar con la vista o portal para gestionar reservas de estancias (guardar/modificar, listar estancias reservadas/realizadas, eliminación).

Capa de Vista

Esta capa tiene la responsabilidad de llevar adelante las funcionalidades necesarias para interactuar con el usuario. Las vistas para este proyecto tienen que estar desarrolladas en HTML5 y se debe utilizar la biblioteca Thymeleaf y CSS para implementar las plantillas.

Se deben diseñar y crear todas las vistas web necesarias para llevar a cabo las siguientes funcionalidades:

- **Administrar usuarios:** registrar nuevos usuarios en el sistema, cambiar clave, listar usuarios, dar de baja o eliminar.
- **Administrar familias:** cargar datos de una familia, modificar datos, consultar familias, eliminar familias que no ofrecen más sus viviendas para estancias. Las familias deben darse de alta una vez que hayan sido registradas como usuario.
- **Administrar casas:** cargar los datos de una vivienda y asociar a la familia correspondiente, modificar los datos (por ejemplo, fechas en las cuales se encuentra disponible), listar casas (país, el periodo de disponibilidad, cantidad de días mínima y máxima de estancia, el precio de la habitación por día, el tipo de vivienda y el nombre del propietario). Se debe eliminar los datos de una casa cada vez que se da de baja la familia propietaria.

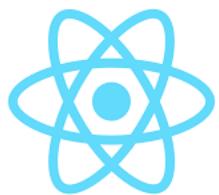
- **Administrar clientes:** cargar datos de los clientes que desean reservar una habitación para realizar una estancia, modificar datos de los clientes, realizar consultas y eliminar clientes. Al igual que las familias, un cliente puede darse de alta una vez que se haya creado el usuario correspondiente.
- **Realizar Reservas:** El portal principal debería permitir que una persona que ingresa a la web pueda consultar las viviendas que se encuentran disponibles para realizar estancias (validando fechas disponibles). Una vez que el usuario encuentra una vivienda que se adapta a sus preferencias y quiere realizar una reserva, recién en ese momento se solicita que se registre como usuario y luego se procede a realizar la reserva.
 - Cuando el usuario se registra se le debe dar la opción de elegir si se quiere registrar como familia que ofrece una vivienda o como cliente. Dependiendo de la opción elegida se pide que complete los datos correspondientes (familia o cliente) para continuar con su registro.
 - Si el usuario ya se encuentra registrado entonces debe realizar el login para poder continuar.
 - Se debe tener en cuenta que para realizar una reserva la vivienda no debe estar ya reservada por otro cliente.
 - Se debe permitir que un cliente modifique su reserva, por ejemplo: cambiar las fechas, o la elimine en caso de no poder realizarla.
 - Se deben poder listar las reservas realizadas por parte de los clientes.

BIBLIOGRAFÍA

- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/protocolo-http/>
- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/http-request/>
- <https://www.ionos.es/digitalguide/hosting/cuestiones-tecnicas/una-mirada-a-los-codigos-de-estado-http-mas-comunes/>
- <https://edytapukocz.com/url-partes-ejemplos-facil/>
- <https://howtodoinjava.com/maven/maven-dependency-management/>
- http://chuwiki.chuidiang.org/index.php?title=Dependencias_con_maven
- <https://www.arquitecturajava.com/que-es-spring-framework/>
- <https://programandointentandolo.com/2013/05/inyeccion-de-dependencias-en-spring.html>
- <https://proitcsolution.com.ve/inyeccion-de-dependencias-spring/>
- <https://www.java67.com/2019/04/top-10-spring-mvc-and-rest-annotations-examples-java.html>
- <https://www.danvega.dev/blog/2017/03/27/spring-stereotype-annotations/>
- <https://www.arquitecturajava.com/spring-stereotypes/>
- <https://www.arquitecturajava.com/que-es-spring-boot/>
- <https://openwebinars.net/blog/que-es-thymeleaf/>

CURSO DE PROGRAMACIÓN FULL STACK

INTRODUCCIÓN A REACT



EGG

REACT

¿QUÉ ES NODE?

Node.js es un entorno de tiempo de ejecución de JavaScript (de ahí su terminación en .js haciendo alusión al lenguaje JavaScript). Este **entorno de tiempo** de ejecución en tiempo real incluye todo lo que se necesita para ejecutar un programa escrito en JavaScript.

Node.js fue creado por los **desarrolladores originales de JavaScript**. Lo transformaron de algo que solo podía ejecutarse en el navegador en algo que se podría ejecutar en los ordenadores como si de aplicaciones independientes se tratara. Gracias a Node.js se puede ir un paso más allá en la programación con JavaScript no solo creando sitios web interactivos, sino teniendo la capacidad de hacer cosas que otros lenguajes de secuencia de comandos como Python pueden crear.

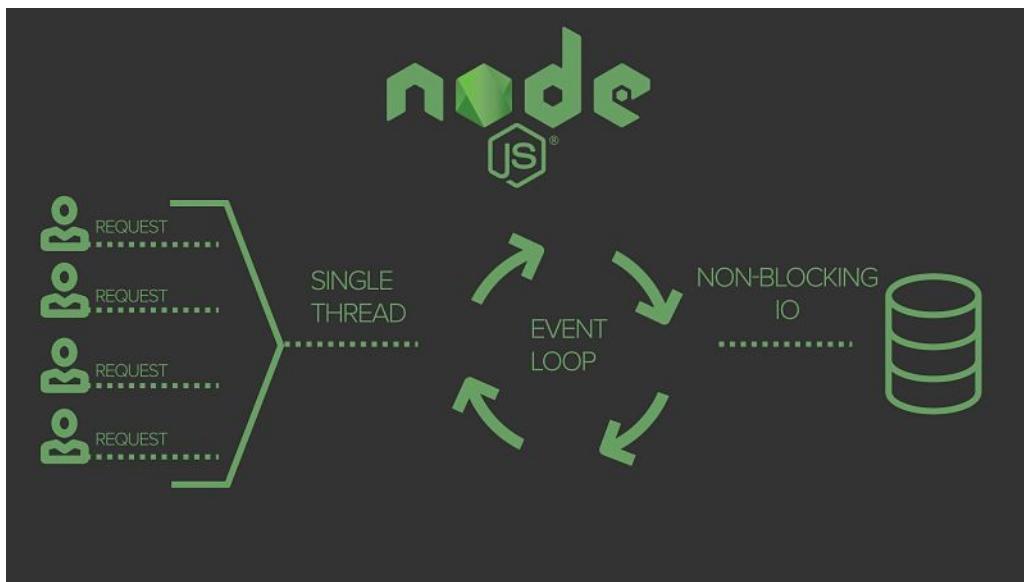
¿CÓMO FUNCIONA NODE.JS?

Para entender como funciona NodeJS, haremos uso de una metáfora:

Imagina que vas a un restaurante. El mesero te atiende, toma tu orden y la lleva a la cocina, luego va y atiende a otra mesa, mientras el cocinero prepara tu orden. Entonces, una misma persona puede atender múltiples mesas, no tiene que esperar que el cocinero termine de preparar una orden antes de pasar a atender a otra mesa. A esta naturaleza o arquitectura la llamamos **asíncrona o no bloqueante**. Así es como funcionan las aplicaciones de Node.js. El camarero viene a ser un **hilo o thread** asignado para manejar solicitudes. De esta forma un sólo hilo puede manejar múltiples solicitudes.

Imaginemos el caso de una aplicación como Twitter donde se tienen muchísimas conexiones simultáneas, para cada una se debe generar un hilo, y naturalmente, en algún punto, no vamos a tener mas hilos o memoria disponibles para atender mas peticiones. Entonces las nuevas peticiones deben esperar hasta que se libere algún hilo, se libere memoria, o debemos agregar mas hardware.

En Node.js no tenemos ese inconveniente, ya que **un solo hilo se encarga de procesar todas las peticiones**. Cuando llega una petición, el hilo de Node.js se encarga de procesarla. Si necesitamos hacer una petición a una base de datos, nuestro hilo no tiene que esperar hasta que la BD devuelva una respuesta. Mientras la petición a la base de datos se está procesando, nuestro hilo puede atender otras peticiones. Cuando la respuesta de la base de datos está lista, se coloca en algo llamado **Event Queue o Cola de Eventos**. Node.js está constantemente monitoreando esta cola de eventos a mediante el **Event Loop**, de tal manera que cuando la respuesta a alguna petición pendiente está lista, Node.js la toma, la procesa y la devuelve.



- **Request:** petición
- **Single Thread:** único hilo
- **Event Loop:** cola de eventos
- **Non-Blocking I/O(Input/Output):** IO no bloqueante

Este tipo de arquitectura hace a Node.js **ideal para construir aplicaciones** que involucran una gran cantidad de accesos a disco, peticiones de red (consultar bases de datos, consultar servicios web), etc. Podemos atender una gran cantidad de clientes sin la urgencia de disponer de más hardware. Y es por ésto que las aplicaciones en Node.js son altamente escalables.

NODE Y REACT

Nodejs es la plataforma más conveniente para alojar y ejecutar un servidor web para una aplicación React. Es por dos razones principales:

- Utilizando un NPM (Node Package Manager), Node trabaja junto con el registro de NPM para instalar fácilmente cualquier paquete a través de la CLI de NPM.
- Node agrupa una aplicación React en un solo archivo para una fácil compilación usando webpack y varios otros módulos de Node.

¿QUÉ ES NPM?

npm es el Node Package Manager que viene incluido y ayuda a cada desarrollo asociado a Node.

Esta herramienta funciona de dos formas:

- Como un repositorio ampliamente utilizado para la publicación de proyectos Node.js de código abierto. Lo que significa que es una plataforma en línea donde cualquiera puede publicar y compartir herramientas escritas en JavaScript.

- Como una herramienta de línea de comandos que ayuda a interactuar con plataformas en línea, como navegadores y servidores. Esto ayuda a instalar y desinstalar paquetes, gestión de versiones y gestión de dependencias necesarias para ejecutar un proyecto.

Para usarlo, debes instalar **node.js**, ya que están desarrollados de forma agrupada.

La utilidad de línea de comando del administrador de paquetes Node permite que **node.js** funcione correctamente.

Para usar paquetes, tu proyecto debe contener un archivo llamado **package.json**. Dentro de ese archivo, encontrarás metadatos específicos para los proyectos.

Los metadatos muestran algunos aspectos del proyecto en el siguiente orden:

- El nombre del proyecto
- La versión inicial
- Descripción
- El punto de entrada
- Comandos de prueba
- El repositorio git
- Palabras clave
- Licencia
- Dependencias
- Dependencias de desarrollo

Los metadatos ayudan a identificar el proyecto y actúan como una línea de base para que los usuarios obtengan información al respecto.

NPX

Desde la versión 5.2.0 de npm, npx está preinstalado con npm. Así que es más o menos estándar hoy en día.

npx es también una herramienta CLI cuyo propósito es facilitar la instalación y la gestión de las dependencias alojadas en el registro npm.

Ahora es muy fácil ejecutar cualquier tipo de ejecutable basado en Node.js que normalmente se instalaría a través de npm.

NPX VS NPM

npx nos ayuda a evitar el versionado, los problemas de dependencia y la instalación de paquetes innecesarios que sólo queremos probar.

También proporciona una forma clara y fácil de ejecutar paquetes, comandos, módulos e incluso listas y repositorio de GitHub.

¿QUÉ ES REACT?

React es una librería Javascript focalizada en el desarrollo de interfaces de usuario. Así se define la propia librería y evidentemente, esa es su principal área de trabajo. Sin embargo, lo cierto es que en React encontramos un excelente aliado para hacer todo tipo de aplicaciones web, **SPA** (Single Page Application) o incluso aplicaciones para móviles. Para ello, alrededor de React existe un completo ecosistema de módulos, herramientas y componentes capaces de ayudar al desarrollador a cubrir objetivos avanzados con relativamente poco esfuerzo.

Por tanto, React representa una base sólida sobre la cual se puede construir casi cualquier cosa con Javascript. Además facilita mucho el desarrollo, ya que nos ofrece muchas cosas ya listas, en las que no necesitamos invertir tiempo de trabajo.

El nombre de React proviene de su capacidad de crear **interfaces de usuario reactivas**, la cual es la capacidad de una aplicación para actualizar toda la interfaz gráfica en cadena, como si se tratara de una formula en Excel, donde al cambiar el valor de una celda automáticamente actualiza todas las celdas que depende del valor actualizado y esto se repite con las celdas que a la vez dependía de estas últimas. Esto se lo conoce como programación reactiva y react lo hará mediante componentes, que los veremos más adelante.

CARACTERÍSTICAS DE REACT

Al igual que otros marcos web JavaScript, React.SJ tiene muchas características principales que podrían ser su consideración. Son:

- Composición de componentes.
- Desarrollo Declarativo Vs Imperativo.
- Performance gracias al DOM Virtual.
- Isomorfismo.
- Elementos y JSX.
- Componentes con y sin estado.
- Ciclo de vida de los componentes.
- Ideal para aplicaciones de alta demanda
- Permite el desarrollo de aplicaciones móviles

En comparación a otros marcos web de JavaScript, que lo hacen de los mejores:

- Fácil de aprender, tiene la curva de aprendizaje más rápida en comparación a otros marcos web.
- Programación reactiva de manera nativa

Estos son los que consideramos son las principales ventajas de React por sobre otros marcos web.

CREATE REACT APP (PRIMEROS PASOS EN REACT)

A no ser que seas un experto desarrollador frontend, la mejor alternativa para dar los primeros pasos con React es usar el paquete `create-react-app`. Te permitirá empezar muy rápido y ahorrarte muchos pasos de configuración inicial de un proyecto.

Generalmente cuando se construye un sitio o app web se tiene que lidiar con una serie de herramientas que forman parte del tooling de un frontend developer, como gestores de paquetes, de tareas, transpiladores, linters, builders, live reload, etc. Toda esta serie de herramientas pueden tener su complejidad si se quieren aprender con el suficiente detalle como para comenzar a usarlas en un proyecto, pero son esenciales para un buen workflow.

Por tanto, si queremos ser detallistas y proveernos de las herramientas necesarias para ser productivos y eficientes, se puede hacer difícil la puesta en marcha en un proyecto Frontend en general. Ahí es donde entra **Create React App**, ofreciéndonos todo lo necesario para comenzar una app con React, pero sin tener que perder tiempo configurando herramientas. Comenzaremos una app con un par de comandos sencillos, obteniendo muchos beneficios de desarrolladores avanzados.

Dentro del CMD en Windows y el Terminal en Mac. Instalamos Create React App con el siguiente comando de npm:

```
npm install -g create-react-app
```

Una vez lo hemos instalado de manera global, nos paramos con el CMD o Terminal en la carpeta creada para nuestro proyecto React y lanzamos el comando para comenzar una nueva aplicación:

```
npx create-react-app mi-app
```

"mi-app" será el nombre de tu aplicación React. Obviamente, podrás cambiar ese nombre por uno de tu preferencia.

Mediante el anterior comando se cargarán los archivos de un proyecto vacío y todas las dependencias de npm para poder contar con el tooling que hemos mencionado. Una vez terminado el proceso, que puede tardar un poco, podemos entrar dentro de la carpeta de nuestra nueva app.

```
cd mi-app/
```

Y una vez dentro hacer que comience la magia con el comando:

```
npm start
```

Observarás que, una vez lanzas el comando para iniciar la app, se abre una página en tu navegador con un mensaje de bienvenida. Ese es el proyecto que acabamos de crear. No obstante, en el propio terminal nos indicarán la URL del servidor web donde está funcionando nuestra app, para cualquier referencia posterior. De manera predeterminada será el `http://localhost:3000/`, aunque el puerto podría cambiar si el 3000 está ocupado.

CARPETAS DE NUESTRA APP REACT

El listado de nuestra app recién creada es bastante sencillo. Observarás que tenemos varias carpetas:

- **node_modules**: con las dependencias npm del proyecto
- **public**: esta es la raíz de nuestro servidor donde se podrá encontrar el index.html, el archivo principal y el favicon.ico que sería el icono de la aplicación.
- **src**: aquí es donde vamos a **trabajar principalmente para nuestro proyecto**, donde vamos a colocar los archivos de nuestros componentes React.

Además encontrarás archivos sueltos como:

- **README.md** que es el readme de Create React App, con cantidad de información sobre el proyecto y las apps que se crean a partir de él.
- **package.json**, que contiene información del proyecto, así como enumera las dependencias de npm, tanto para desarrollo como para producción. Si conoces npm no necesitarás más explicaciones.
- **.gitignore** que es el típico archivo para decirle a git que ignore ciertas cosas del proyecto a la hora de controlar el versionado del código.

COMPONENTE RAÍZ DEL PROYECTO CON CREATE REACT APP

Para nuestros primeros pasos con React no necesitamos más que entrar en la carpeta src y empezar a editar su código. De entre todos los archivos que encuentras en la carpeta src por ahora nos vamos a quedar con uno en concreto, src/App.js, en el que se crea el componente principal de nuestra app.

De momento en nuestra aplicación sólo tenemos un componente, el mencionado componente raíz localizado en src/App.js. Sin embargo a medida que se vaya desarrollando la aplicación se irán creando nuevos componentes para realizar tareas más específicas, e instalando componentes de terceros, que nos ayuden a realizar algunas tareas sin necesidad de invertir tiempo en programarlas de nuevo. Así es como llegamos a la arquitectura frontend actual, basada en componentes.

HOLA MUNDO EN REACT

En el mencionado archivo src/App.js nos encontraremos poco código, pero para no despistarnos en nuestro primer "Hola Mundo", podemos borrar gran parte y quedarnos solamente con esto:

```
import React from 'react';
function App() {
  return (
    <div>
      <h1>Hola Mundo!</h1>
      <p>Bienvenidos a los primeros pasos con React</p>
    </div>
  );
}
export default App;
```

En la primera línea "import React..." se está importando la librería React.

Las funciones que utilizamos para implementar a los componentes React, son funciones de JavaScript y retornan elementos de React que describen lo que debe aparecer en la pantalla.

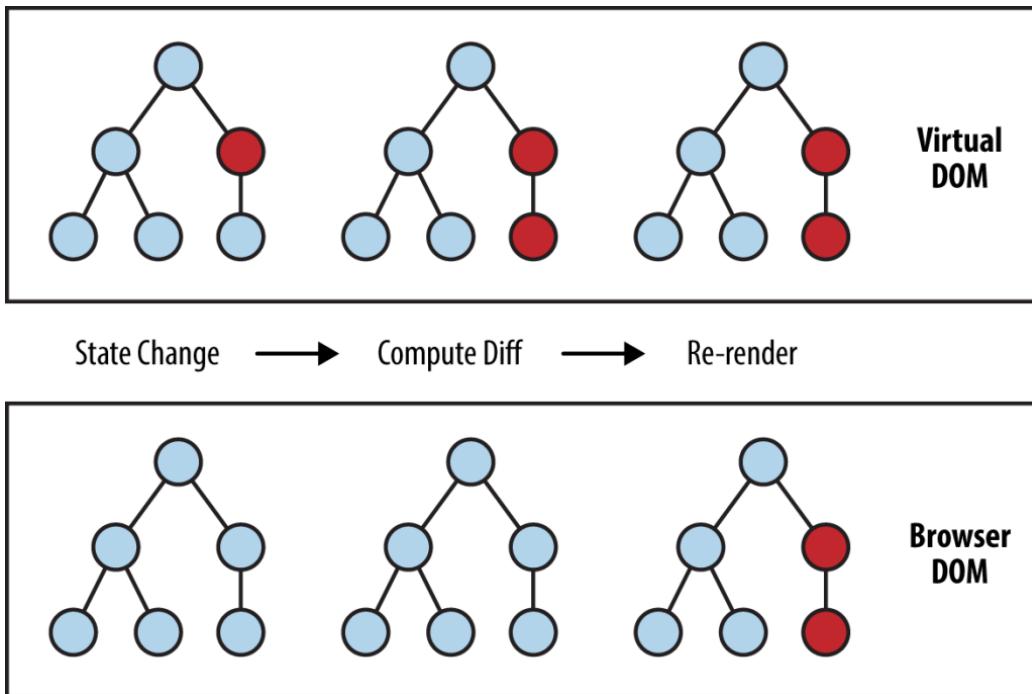
Guardando el archivo podrás observar como el navegador refresca la página automáticamente, gracias a las herramientas incluidas de serie dentro de Create React App.

DOM VIRTUAL

El *Virtual DOM* es una representación en memoria del *DOM real* (que es el DOM que conocemos de JS) que actúa de intermediario entre el estado de la aplicación y el DOM de la interfaz gráfica que está viendo el usuario.

Puesto que cada elemento es un nodo en el árbol del DOM, cada vez que se produce un cambio en cualquiera de estos elementos (o un nuevo elemento es añadido) se genera un nuevo Virtual DOM con el árbol resultante. Dado que este DOM es virtual, la interfaz gráfica aún no es actualizada, sino que se compara el DOM real con este DOM virtual con el objetivo de calcular la forma más óptima de realizar los cambios (es decir, de renderear los menos cambios posibles). De este modo se consigue reducir el coste en términos de rendimiento de actualizar el DOM real.

El siguiente gráfico ejemplifica el proceso:



1. **State Change:** En este primer paso, se produce un cambio en el estado del nodo de color de rojo, lo cual provoca que se genere en memoria un **Virtual DOM** con el árbol resultante tras ese cambio.
2. **Compute diff:** A continuación se realiza la comparación entre el árbol del Virtual DOM y el del navegador (DOM real) con el fin de detectar los cambios producidos. Como veis, el cambio afecta a toda la rama descendiente del nodo cuyo estado cambió.
3. **Re-render:** Finalmente, se consolida el cambio en el DOM real y la interfaz gráfica es actualizada de golpe.

¿CÓMO USA REACT EL VIRTUAL DOM?

En React, cada pieza de la UI es un componente y cada componente posee un estado interno. Este estado es *observado* por la librería con el fin de detectar cambios en él de modo que, cuando se produce un cambio, React actualiza el árbol de su Virtual DOM y sigue el mismo proceso para trasladar los cambios resultantes a la interfaz presentada en el navegador. Esto le permite tener un rendimiento mejor que las librerías que manipulan el DOM directamente, pues React **sólo** actualiza aquellos objetos en los que ha detectado cambios durante el proceso de *diffing*.

Además, React traslada los cambios al DOM de la interfaz gráfica de forma masiva lo cual también incrementa el rendimiento. Esto se debe a que en vez de enviar múltiples cambios, React los junta todos en uno para reducir el número de veces que la interfaz gráfica debe pintarse.

Finalmente, otra de las ventajas que nos proporciona React es la abstracción de todo el proceso de actualización del DOM que nos proporciona. Es decir, nos permite olvidarnos de actualizar los atributos o el valor de los nodos que componen nuestra interfaz gráfica ya que todo esto sucede a nivel interno.

JSX

JSX es una extensión de la sintaxis de JavaScript para su uso en React. JSX, produce **elementos** de React. Un elemento es un bloque más de las aplicaciones de React y describe lo que quieras ver en pantalla.

Se puede utilizar JSX dentro de declaraciones IF y bucles FOR, asignarlo a variables, aceptarlo como argumento y retornarlo desde dentro de funciones.

ATRIBUTOS CON JSX

Puedes utilizar comillas para especificar Strings como atributos:

```
const elemento = <div tabIndex="0"></div>
```

También usar llaves para insertar una expresión JS:

```
const elemento = <img src={user.avatarURL}></img>
```

JSX REPRESENTA OBJETOS

Estos objetos son llamados "Elementos de React". React lee estos objetos y los usa para construir el DOM.

Supongamos que tenemos el siguiente elemento:

```
const elemento = (<h1 className = 'saludo' > Hola Mundo </h1>);
```

Este elemento va a ser idéntico al siguiente elemento

```
const elemento = React.createElement('h1',  
{className = 'saludo'},  
Hola Mundo );
```

RENDERIZAR UN ELEMENTO

El renderizado se va a ejecutar dentro del return de cada componente, es decir, todo lo que se va a "mostrar" (renderizar) va a ser lo que contenga ese return. El document.getElementById("root") y el ReactDOM.render() se ejecutan una vez y es el primer paso para que después se renderice el componente App.js (componente principal por defecto) y cada componente que este mismo llame.

Supongamos que tenemos las siguientes variables:

```
const name = 'Lionel Messi',  
const elemento = <h1> Hola, {name} </h1>
```

Para renderizarlo, imagina que tenemos el siguiente div en alguna parte del HTML:

```
<div id="root"></div>
```

Para renderizar un elemento de React en un nodo raíz del DOM, pasa ambos a ReactDOM.render()

```
ReactDOM.render(
```

```
element,  
document.getElementById("root")  
);
```

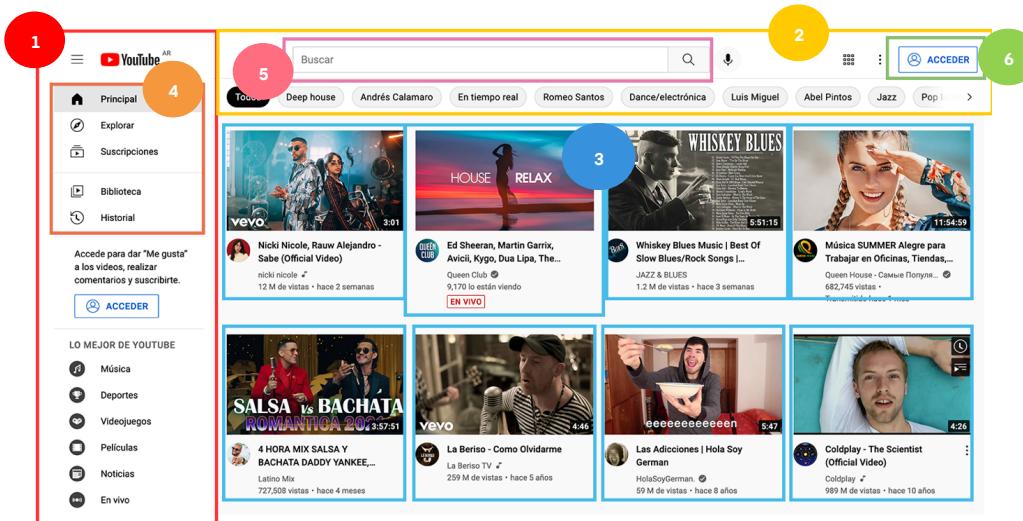
Muestra: Hola Lionel Messi

ACTUALIZANDO EL ELEMENTO

React solo actualiza lo que sea necesario. ReactDOM compara el elemento y sus hijos con el elemento anterior, y solo aplica las actualizaciones del DOM que son necesarias para que el DOM esté en el estado deseado.

COMPONENTES

React, como dijimos, es una librería JavaScript para crear interfaces de usuario dependiendo de su estado. Las aplicaciones React se construyen mediante componentes, los cuales son elementos independientes y pueden ser reutilizados, además, describen cómo tienen que visualizarse y cómo tienen que comportarse. Los componentes permiten separar la interfaz de usuario en piezas independientes.



Dentro de una página como YouTube, podemos marcar varios componentes:

1) Un Sidebar

2) Un Header

3) Cada carta es su propio componente

También dentro de cada componente existen otros componentes:

4) Los botones Principal, Explorar, Suscripciones, Biblioteca e Historial. Cada uno de esos botones es un componente, que vive dentro del componente Sidebar.

5) El buscador es un componente dentro del Header

6) El botón de login es otro componente dentro del Header

Y en esta pagina podríamos nombrar muchos componentes más.

Cada aplicación React comienza con un componente de raíz y se compone de muchos **componentes en una formación de árbol**. Los componentes en React son “funciones” que dejan la UI basada en la data (apoyo y estado) que recibe.

COMPONENTES EN REACT

Conceptualmente, los componentes son como las funciones de JavaScript. Aceptan entradas arbitrarias (llamadas “props”) y devuelven a React **elementos** que describen lo que debe aparecer o se debe ver en la pantalla. Los props **son las propiedades o información que viaja de un componente padre a un hijo**.

Además existen dos tipos de componentes en React:

- **Componentes funcionales:** Solo tienen propiedades.
- **Componentes de clase:** Tienen propiedades, ciclos de vida y estado.

La forma más sencilla de definir un componente es escribir una función de JavaScript:

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

Esta función es un componente de React válido porque acepta un solo **argumento de objeto “props”** (que proviene de propiedades) con datos y devuelve un elemento de React. Llamamos a dichos componentes “funcionales” porque literalmente son funciones JavaScript.

También puedes utilizar una clase de ES6 para definir un componente:

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

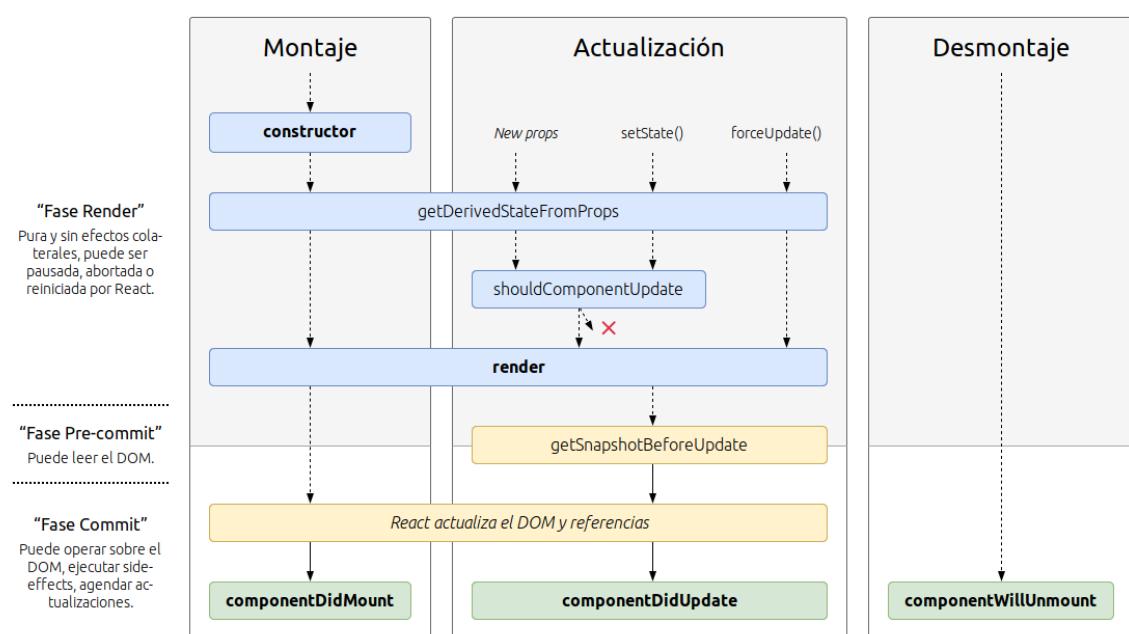
CICLO DE VIDA DE UN COMPONENTE

Vimos que existen dos formas de crear componentes (*funcionales* y de *clases*).

Cuando creamos un componente de clase, tenemos la ventaja, que al extenderlo de *React.Component* nos va a proporcionar más *métodos* y *propiedades*, que van a mejorar de manera exponencial el componente.

```
class Title extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}!!!</h1>;  
  }  
  
}  
  
ReactDOM.render(<Title name="Mauricio" />,  
document.getElementById("root"));
```

Para entender mejor el ejemplo, es necesario saber qué *React.Component*, tiene tres *ciclos de vida*:



a. Montaje

Cuando se crea una instancia de un componente y se inserta en el DOM.

b. Actualización

Cuando sufre algún cambio las propiedades (props) o el estado (state) del componente.

c. Desmontaje

Cuando el componente se elimina del DOM.

FUNCTIONAL COMPONENT (REACT HOOKS)

Habíamos hablado de los componentes “funcionales” que son funciones JavaScript, y habíamos dicho que la forma más sencilla de definir un componente es escribir una función de JavaScript. Es aquí, donde entran los **React Hooks**.

Los **Hooks** son una nueva característica en React 16.8. Estos te permiten usar el estado y otras características de React sin escribir una clase. Los vamos a utilizar junto con los componentes funcionales y de esa manera, evitar usar clases y solo trabajar con un solo tipo de componente.

```
import React, { useState } from 'react';

function Example() {

  // Declara una nueva variable de estado, la cual llamaremos "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Esta nueva función **useState** es el primer “Hook” que vamos a aprender, pero este ejemplo es solo una introducción. ¡No te preocupes si aún no tiene sentido!

Antes de continuar, debes notar que los Hooks son:

- **Completamente opcionales.** Puedes probar Hooks en unos pocos componentes sin reescribir ningún código existente. Pero no tienes que aprender o usar Hooks ahora mismo si no quieres.
- **100% compatibles con versiones anteriores.** Los Hooks no tienen cambios con rupturas con respecto a versiones existentes.
- **Disponibles de inmediato.** Los Hooks ya están disponibles con el lanzamiento de la versión v16.8.0.

¿PERO QUÉ ES UN HOOK?

Los Hooks son funciones que te permiten “enganchar” el estado de React y el ciclo de vida desde componentes de función. Los hooks no funcionan dentro de las clases — te permiten usar React sin clases. (No recomendamos reescribir tus componentes existentes de la noche a la mañana, pero puedes comenzar a usar Hooks en los nuevos si quieras).

React proporciona algunos Hooks incorporados como useState. También puedes crear tus propios Hooks para reutilizar el comportamiento con estado entre diferentes componentes. Primero veremos los Hooks incorporados.

HOOK DE ESTADO

Este ejemplo renderiza un contador. Cuando haces click en el botón, incrementa el valor:

```
import React, { useState } from 'react';

function Example() {

  // Declara una nueva variable de estado, que llamaremos "count".  const
  [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Aquí, useState es un Hook (hablaremos de lo que esto significa en un momento). Lo llamamos dentro de un componente de función para agregarle un estado local. React mantendrá este estado entre re-renderizados. useState devuelve un par: el valor de estado actual y una función que le permite actualizarlo. Puedes llamar a esta función desde un controlador de eventos o desde otro lugar. Es similar a this.setState en una clase, excepto que no combina el estado antiguo y el nuevo.

El único argumento para useState es el estado inicial. En el ejemplo anterior, es 0 porque nuestro contador comienza desde cero. Ten en cuenta que a diferencia de this.state, el estado aquí no tiene que ser un objeto — aunque puede serlo si quisieras. El argumento de estado inicial solo se usa durante el primer renderizado.

HOOK DE EFECTO

Es probable que hayas realizado recuperación de datos, suscripciones o modificación manual del DOM desde los componentes de React. Llamamos a estas operaciones “efectos secundarios” (o “efectos” para abreviar) porque pueden afectar a otros componentes y no se pueden hacer durante el renderizado.

El Hook de efecto, `useEffect`, agrega la capacidad de realizar efectos secundarios desde un componente de función. Tiene el mismo propósito que `componentDidMount`, `componentDidUpdate` y `componentWillUnmount` en las clases React, pero unificadas en una sola API.

Por ejemplo, este componente establece el título del documento después de que React actualiza el DOM:

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);
  // Similar a componentDidMount y componentDidUpdate:
  useEffect(() => {
    // Actualiza el título del documento usando la Browser API
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Cuando llamas a `useEffect`, le estás diciendo a React que **ejecute tu función de “efecto”** después de vaciar los cambios en el DOM. Los **efectos** se declaran dentro del componente para que tengan **acceso a sus props y estado**. De forma **predeterminada**, React **ejecuta los efectos después de cada renderizado** — incluyendo el primer renderizado.

En el ejemplo que tenemos del `useEffect`, va a llamar las funciones sobre los componentes de manera infinita, esto a veces nos puede ser útil, pero también nos puede generar un bucle infinito, entonces supongamos que queremos cargar unas cartas, estas se van a cargar de manera indefinida y va a romper nuestra pagina.

1. Para ejecutar en cada reenvío del componente:

```
useEffect(() => {  
})
```

Este es el que nos puede generar un bucle infinito.

2. Para ejecutar algo **solo una vez** después del montaje del componente (se procesará una vez), debe usar:

```
useEffect(() => {  
  // hará algo solo una vez si pasas un arreglo vacío []  
  // tener en cuenta que ese componente se renderizará una vez (con los  
  // valores predeterminados) antes de que lleguemos aquí  
}, [])
```

3. Para ejecutar cualquier cosa **una vez** en el montaje de componentes y en el cambio de datos / datos2 :

```
const [data, setData] = useState(false)  
  
const [data2, setData2] = useState('default value for first render')  
  
useEffect(() => {  
  
  // si pasas alguna variable, el componente se volverá a procesar después  
  // del montaje del componente una vez y una segunda vez si esto (en mi  
  // caso, data o data2) se cambia  
  
}, [data, data2])
```

Según que tenemos que hacer elegiríamos una opción o la otra.

¿QUÉ ES REACT ROUTER?

React Router es una colección de componentes de navegación la cual podemos usar como ya lo mencione tanto en web o en móvil con React Native. Con esta librería vamos a obtener un enrutamiento dinámico gracias a los componentes, en otras palabras tenemos unas rutas que renderizan un componente.

BENEFICIOS DE REACT ROUTER

- Establecer rutas en nuestra aplicación ej: Home, About, User.
- Realizar redirecciones
- Acceso al historial del navegador
- Manejo de rutas con parámetros
- Páginas para el manejo de errores como 404

COMPONENTES PILARES DE REACT ROUTER

BrowserRouter

Este componente es el encargado de envolver nuestra aplicación dándonos acceso al API historial de HTML5 (pushState, replaceState y el evento popstate) para mantener su UI sincronizada con la URL.

Routes

Genera un árbol de rutas y a partir de este nos permite reemplazar la vista con el componente que coincide con la URL de nuestra barra de navegación y nos va a renderizar solamente dicho componente.

Route

Con Route podemos definir las rutas de nuestra aplicación, quizás sea el componente más importante de React Router para llegar a comprender todo el manejo de esta librería. Cuando definimos una ruta con Route le indicamos que componente debe renderizar.

Este componente cuanta con algunas propiedades.

Path: la ruta donde debemos renderizar nuestro componente podemos pasar un string o un array de string.

Exact: Solo vamos a mostrar nuestro componente cuando la ruta sea exacta. Ej: /home === /home.

Strict: Solo vamos a mostrar nuestro componente si al final de la ruta tiene un slash. Ej: /home/ === /home/

Sensitive: Si le pasamos true vamos a tener en cuenta las mayúsculas y las minúsculas de nuestras rutas. Ej: /Home === /Home

Element: Le pasamos un componente para renderizar solo cuando la ubicación coincide. En este caso el componente se monta y se desmonta no se actualiza.

Render: Le pasamos una función para montar el componente en línea.

¿QUE ES UNA URL?

Una cosa importante para trabajar con React Router es entender que es un URL y las partes que componen a una URL. Ya que vamos a trabajar con dichas partes de la URL con React Router.

Un URL (Uniform Resource Locator) hace referencia más comúnmente a páginas web (HTTP), pero también puede aplicarse a la transferencia de archivos (FTP), correo electrónico (mailto), acceso a bases de datos (ODBC) y muchas otras aplicaciones. Una URL HTTP también puede describir la ubicación de recursos externos como una imagen, hoja de estilo, script e incluso una sección específica dentro de un documento HTML. Las URL completas o al menos parciales aparecen en la barra de direcciones de la mayoría de los navegadores web.

PARTES DE UNA URL

1

2

3

4

5

6

7

8

<https://www.example.com:3000/path/resource?id=123#section-id>

- 1) **Protocolo o esquema (Scheme):** define como el recurso se va a obtener.
- 2) **Subdominio (SubDomain):** www es el más común pero no es necesario.
- 3) **Dominio (Domain):** valor único dentro del dominio de nivel superior.
- 4) **Dominio de nivel superior (Top-level Domain):** existen cientos de opciones.
- 5) **Puerto (Port):** si es omitido, HTTP se conectará al puerto 80, HTTPS al 443.
- 6) **Ruta (Path):** especifica y capaz encuentra el recurso requerido por el usuario.
- 7) **Parámetro (Query String):** datos pasados al servidor, si está presente
- 8) **Etiqueta (Fragment Identifier):** especifica un lugar concreto de una pagina HTML

1. Protocolo

El esquema suele ser el nombre de un protocolo, que define cómo se obtendrá el recurso. Sin embargo, esquemas como "archivo" y "mailto" no especifican un protocolo. Los clientes, como los navegadores web, se conectan a sitios web a través del **Protocolo de transferencia de Hipertexto**, también conocido como HTTP.

2. Subdominio

Aunque www es el más común, los subdominios se pueden configurar con cualquier valor que consista en caracteres ASCII alfanuméricos que no distinguen entre mayúsculas y minúsculas. Se permiten guiones si están rodeados por otros caracteres ASCII u otros guiones. Los subdominios también se pueden eliminar, acortando y simplificando toda la URL.

3-4. Dominio + nivel superior

En octubre de 1984, el conjunto original de dominios de nivel superior se definió como:

- .com
- .edu
- .gov
- .mil
- .org
- .neto

Desde 1984 se crearon un puñado de nuevos dominios de nivel superior, pero el cambio más significativo se produjo en 2012 cuando ICANN (Corporación de Internet para la Asignación de Nombres y Números) autorizó la creación de casi dos mil nuevos dominios de nivel superior.

5. Puerto

El propósito de un puerto es identificar de forma única diferentes procesos o aplicaciones que se ejecutan en un solo servidor. Los números de puerto permiten que cada proceso o aplicación comparta una conexión de red. Las URL públicas a menudo no incluyen un número de puerto. En esos casos, se utiliza el puerto predeterminado del esquema. Los puertos predeterminados para HTTP y HTTPS son 80 y 443, respectivamente.

6. Ruta

Una ruta describe una ubicación específica dentro de un sistema de archivos. En el contexto de una URL HTTP, una ruta describe un recurso específico, como un documento HTML.

7. Parámetro

Los parámetros contienen datos que se enviarán al servidor para procesamiento adicional. Puede contener pares de nombre / valor separados por un ampersand (&), así:

?first_name=Alfred&last_name=Pennyworth

En el ejemplo anterior, los datos serían:

"Nombre: Alfred"

"Apellido: Pennyworth"

Luego, estos datos se pasan a los scripts del lado del servidor para su procesamiento. Si esta es una consulta válida, el servidor devolverá información pertinente a Alfred Pennyworth.

8. Etiqueta

Las etiquetas en una URL aparecen después del hashtag #.

Su función, entre otras cosas, consiste en permitir hacer scroll hasta un elemento en concreto. Por ejemplo, si mandamos a alguien una URL que contenga una etiqueta, ésta le dirigirá a la parte exacta de la página en cuestión.

INSTALACIÓN DE REACT ROUTER

Para instalar la librería solo tenemos que ir a la terminal estar ubicados en la raíz de nuestro proyecto y ejecutar el siguiente comando.

```
npm install react-router-dom
```

TRABAJANDO CON REACT ROUTER

Teniendo todo listo ahora si vamos a nuestro editor de código y abrimos el archivo **App.js** que está ubicado en `src/App.js` acá vamos a limpiar muchas cosas hasta que al final tengamos algo como el siguiente código.

```
import React from 'react'
import { BrowserRouter, Route, Routes } from 'react-router-dom';
import About from './pages/About';
import Home from './pages/Home';
import PageNotFound from './pages/PageNotFound';
import "./App.css"

function App() {
  return (
    <BrowserRouter>
      <div className='App'>
        <Routes>
          <Route exact path='/' element={<Home />} />
          <Route exact path='/about' element={<About />} />
          <Route path='*' element={<PageNotFound />} />
        </Routes>
      </div>
    </BrowserRouter>
  );
}

export default App;
```

Importamos nuestro componente `BrowserRouter`, también importamos `Route` de `react-router-dom`. Envolvemos nuestra aplicación con `Router` y definimos nuestra primera ruta en este caso nuestro `home` le indicamos que debe ser exacta la ruta y que haga render de nuestro componente `Home` pero donde esta nuestro componente `Home`. Vamos a crearlo!

```
// Home.js
import React from 'react'
export default function Home() {
  return (
    <div>
      <h3>Home</h3>
    </div>
  )
}
```

Este es nuestro componente `Home` que está dentro de una carpeta llamada `pages/Home` y solo tenemos un `h3` con un texto. Si vamos al navegador a `http://localhost:3000/` vamos a ver solo el texto. Así como este vamos a crear los componentes `About` y `PageNotFound`.

LINK

Con Link vamos a poder navegar por nuestra aplicación, este componente recibe las siguientes propiedades.

To: le podemos pasar un string, object o una function para indicarle la ruta a la cual queremos navegar.

Replace: cuando es verdadero, y hacemos clic en el enlace reemplazará la entrada actual en la pila del historial en lugar de agregar una nueva.

```
<nav>
  <ul>
    <li>
      <Link to="/">Home</Link>
    </li>
    <li>
      <Link to="/asd">PageNotFound</Link>
    </li>
    <li>
      <Link to="/users">Users</Link>
    </li>
  </ul>
```

Agregamos el siguiente código en nuestro componente *App.js* dentro del **div** con la clase **App**, ahora podemos ver un menú de navegación en nuestra aplicación. Me faltó mencionar debemos hacer el **import** de **Link**.

```
// App.js
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Link
} from "react-router-dom";
```

[Home](#) [About](#) [PageNotFound](#)

Home

NAVIGATE

Con este componente podemos causar un redireccionamiento a una ruta diferente a la ruta actual reemplazando el location actual y el historial de navegación.

Tiene la siguiente propiedad:

To: es un propiedad de **navigate** donde le enviamos un string u objeto para decir hacia dónde vamos a realizar el redireccionamiento.

```
<Route exact path="/redirect" element={<Navigate to={"/"} />} />
```

Agregamos ese componente dentro de **Routes**, ahora si vamos al navegador y en la barra de dirección escribimos lo siguiente `http://localhost:3000/redirect` vamos a ver que nos hace un redirect a la ruta /. No olvides hacer el **import** de **Navigate**.

HOOK USEPARAMS

Ahora vamos a ver unos Hooks que nos van a ayudar a realizar operaciones con los componentes del navegador.

Con este **Hook** podemos acceder a los **params** de las rutas veamos un ejemplo para eso debemos crear un nuevo componente el cual se va encargar de usar el hook y debemos agregar un nuevo link al menú de navegación al igual que un **Route**

```
// App.js
import Users from './pages/Users';
```

```
// App.js
<li>
  <Link to="/users/1">User id 1</Link>
</li>
```

```
<Route exact path='/users/:id' element={<Users />} />
```

```
// pages/Users.js
import React from 'react'
import { useParams } from 'react-router-dom'

export default function Users() {
  const { id } = useParams()
  return (
    <div>
      <h3>Mostrando usuario con id: {id}</h3>
    </div>
  )
}
```

Lo primero es importar el hook y obtener el params que definimos en el Route, con este hook es muy fácil acceder al params que indicamos en el Route si vamos al navegador y damos click en el nuevo link que tiene como nombre **User id 1** vamos a ver lo siguiente.

[Home](#) [About](#) [User id 1](#) [PageNotFound](#)

Mostrando usuario con id: 1

EJERCICIOS DE APRENDIZAJE

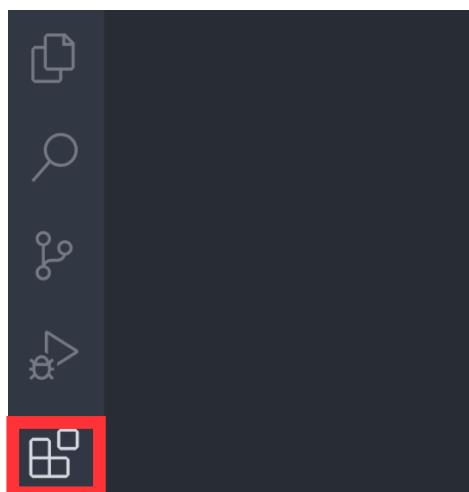
¡¡Vamos a ponernos manos a obra!! Ahora vamos a practicar lo visto en la guía con unos ejercicios de React. **Nota:** recomendamos que para cualquier vista que vayan a realizar usen BootStrap, de esta manera podemos tener vistas de manera “rápida” y podemos trabajar con React.



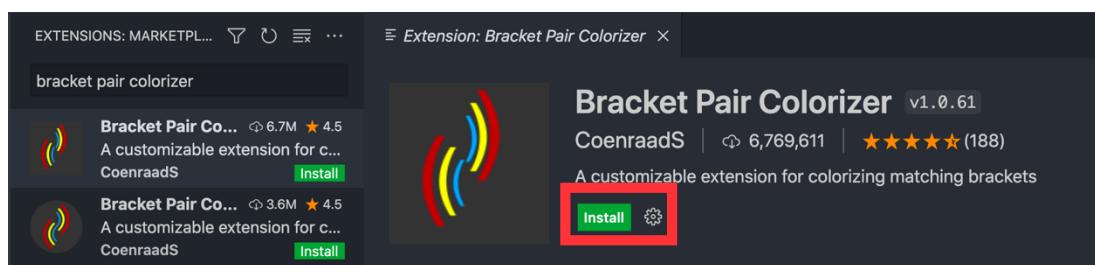
VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

Importante: Para la realización de los ejercicios deberemos instalar las siguientes extensiones en Visual Studio Code: Bracket Pair Colorizer – Bootstrap 4, Font awesome 4, Font Awesome 5 Free & Pro snippets – Auto Close Tag – ES7 React/Redux/GraphQL/React-Native snippets – Error Lens – Auto Import – Auto Import ES6, TS, JSX, TSX.

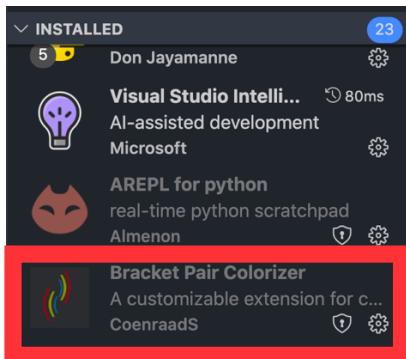
Para instalar una extensión en Visual Studio Code, haremos click en el siguiente botón:



Esto nos abrirá una pestaña que nos muestra las extensiones instaladas y nos da un buscador para instalar más extensiones. En ese buscador, buscaremos alguna de las extensiones mencionadas, cuando la encontramos, le daremos a **Install**.



Una vez que la hayamos instalado nos va a aparecer en nuestras extensiones instaladas



También deberemos instalar las siguientes extensiones en Google Chrome o el navegador que usen: Allow Cors – Json Formatter – React Developer Tools – Redux DevTools. Hecho esto ya podemos empezar con los ejercicios.

1. Crear un proyecto compuesto de un **solo Functional Component**. En dicho componente mostrar al menos dos datos, como por ejemplo titulo y subtítulo.

El componente debe ser llamado desde App, a continuación, se propondrá la jerarquía del árbol de componentes y de como es el llamado desde index.js

- Index.js
 - App
 - Ejemplo

```
export const Ejemplo = () => {  
  
    const titulo = "Hello Dog";  
    const subTitulo = "Sub titulo";  
  
    return (  
        <div>  
            <h1>{titulo}</h1>  
            <h2>{subTitulo}</h2>  
        </div>  
    );  
};
```

2. Crear un proyecto compuesto por tres componentes bajo la misma jerarquía. Crear un Navbar, Main y Footer.

- Index.js

- App

- Navbar
- Main
- Footer

Necesitamos hacer que Footer, Main y Navbar muestren al menos un dato, de la misma manera que el ejercicio anterior.

3. Crear un Componente Main el cual llame dos veces a un mismo componente, es decir, que Main anide a Hijo e Hijo.

- Index.js

- App

- Main
 - Hijo
 - Hijo

Al primer Componente anidado pasarle como props el nombre Chiquito y al segundo el nombre Filomena. Desde los componentes Hijos atrapar los valores mediante las props.

```
//Envio dato desde donde lo llamo
<Hijo nombre="Chiquito" />;
<Hijo nombre="Filomena" />

//Recibo dato
export function Hijo(props) {
  return <h1>Hola, {props.nombre}</h1>;
}
```

4. Crear un proyecto compuesto por 4 componentes bajo la siguiente jerarquía.

- Index.js

- App

- Navbar
- Main1 o Main2
- Footer

Al hacer click sobre las dos posibles opciones en el NavBar, se deberá cambiar entre Main1 y Main2 dependiendo de la navegación. Para lograrlo se deberá instalar y usar React Router Dom.

A continuación, se propone como será la jerarquía de los componentes

```
<div className='App'>
  <NavBar />
  <Routes>
    <Route exact path='main1' element={<Main1 />} />
    <Route exact path='main2' element={<Main2 />} />
  </Routes>
  <Footer />
</div>
```

En la siguiente imagen se vera como establecer la navegación. Se sugiere ver los videos explicativos que encontrarán en el canal de Youtube de Egg.

```
<Link to={"/main1"}>
  Main1
</Link>
```

5. Crear un proyecto compuesto de un solo componente y hacer uso de **useState** y mostrar el **state** del componente.

Se podrá crear un **contador de clicks** o crear un input que mediante **onChange** cambie el valor del **state**. Cualquiera de las dos opciones, son validas para este ejercicio.

6. Crear un proyecto compuesto de un **solo componente y un servicio**, quien deberá ser capaz de llamar desde el servicio, mediante la **funcionalidad Fetch**, a la API de Rick and Morty (<https://rickandmortyapi.com/api/character>) .

Una vez llamado los datos desde el servicio, hacer uso de **useEffect** en el componente creado, deberá mostrar una lista compuesta de los nombres de los 20 primeros personajes.