



# Curso de Back-End con Node.js (Inicial)

## Clase 01



# *¡Muchas gracias por habernos elegido!*

Esperamos que puedan **disfrutar** y **aprovechar** este curso al máximo, y ojalá logremos transmitirles la **pasión** que nosotros sentimos por estos temas.

Si en algún momento del curso ven algo que le gustaría **mejorar**, por favor háganoslo saber lo antes posible. Además, al finalizar tendrán una **encuesta** de satisfacción anónima para hacernos llegar sus comentarios.



# Presentarnos

(30 segundos cada uno)

*Nombre | Qué hago | Qué espero del curso*



# Pedidos

# Pedidos (1/2)






- Respetar los **horarios** ⌚ de clase.
- Cuidar la **limpieza** 🧹 de salones, baños, cocina y demás lugares de la academia.
- No tirar vasos de café/té ☕ con líquido en las papeleras.
- Tener cuidado con los líquidos y las notebooks 💧💻.  
No nos responsabilizamos por accidentes de este tipo.
- Si vienen en bicicleta 🚲, la misma debe quedar en el biciclero del patio del frente y con candado. No nos responsabilizamos por daño o hurto.

Esto aplica sólo para  
las clases presenciales.

# Pedidos (2/2)



¡Participen mucho!


- Queremos oír la **voz** de todos. 
- Queremos que compartan su **pantalla** . Queremos ver lo que hicieron, y si está mal, mejor. De los **errores** se aprende.
- La clase es mucho más **divertida** si todos participan. 
- En la **interacción** con ustedes es donde podemos aportar más **valor**. Para tener una clase “unilateral” (dónde sólo habla el docente), probablemente les rinda más un curso de Udemy.



# Tips generales



# Tips generales

- ¡Hagan **muchas preguntas!** (no sientan vergüenza).
- **Googleen** mucho. También consulten [Stack Overflow](#).
- Intenten **ayudarse** entre ustedes. Enseñar es una gran forma de aprender.
- No copien/peguen código salvo que entiendan o sepan lo que están copiando. Copiar código no está mal, pero sean criteriosos.
-  Cuidado con las actualizaciones automáticas del sistema operativo. Sobre todo en Windows. Eviten actualizar su equipo en medio de la clase.





# Uso de Microsoft Teams



# Uso de Microsoft Teams (1/4)

Desde la fundación de Hack Academy en el año 2016, hemos usado [Slack](#) como la principal herramienta de comunicación con los alumnos.

A partir del año 2020, hemos empezado a incorporar, de forma paulatina, el uso de [Microsoft Teams](#), ya que permite manejar en una misma herramienta todo lo referente a **mensajería**, intercambio de **materiales** y **videollamadas**.

Independientemente de la herramienta utilizada, es importante respetar algunas buenas prácticas para hacer la comunicación más eficiente, las cuales se listan en la siguiente diapositiva.





# Uso de Microsoft Teams (2/4)

Buenas prácticas:

- Usar su **nombre completo** + **foto** de perfil.
- Escribir mensajes en los **canales** correspondientes.
- Compartir **código** con el formato adecuado (no como “texto plano”).
- Escribir las **consultas** en un **único mensaje**. Evitar escribir varios mensajes para una misma consulta. Ej: No escribir “*Hola*”, “*¿Cómo están?*”, “*Tengo una consulta*”, “*No logré hacer funcionar...*”, en 4 mensajes diferentes. Con esto se evitan las “cataratas” de notificaciones.



# Uso de Microsoft Teams (3/4)

Buenas prácticas (continuación):

- Respetar los **hilos** (*threads*) de comunicación al responder un mensaje.
- **Evitar** hacer **consultas** a los **docentes vía mensajes privados**. 🙏 🙏 🙏  
Por consultas administrativas, escribir a [hola@ha.dev](mailto:hola@ha.dev).
- No sientan vergüenza de usar el **canal de dudas**. Esto permite que cualquier persona pueda responder, ¡incluso otro alumno!
- Si bien Slack y Teams se pueden usar vía la web, para lograr una mejor experiencia, recomendamos descargar las **apps** para **desktop** y **mobile**.



# Uso de Microsoft Teams (4/4)

Buenas prácticas (continuación):

- Las **notificaciones** de **Teams** las recibirán en su **casilla de correo** **@student.ha.edu.uy** (que es una casilla de [Microsoft Outlook](#)).

👉 Recomendamos que configuren dicha casilla para que los mensajes entrantes se **redirijan** (*forwardeen*) automáticamente a su casilla personal (ej: a su casilla de Gmail o Hotmail). De esta forma no se perderán las notificaciones que hagamos por Teams.






# Clases por videollamada

Recomendaciones

# Clases por videollamada (1/2)

Recomendaciones para mejorar la experiencia de las clases por videollamada:

-  Usar un **Monitor Externo** o TV, de tal forma que en una pantalla puedan seguir la clase y en la otra pantalla hacer sus ejercicios.  
 En realidad, esto es sumamente útil más allá de las videollamadas. Casi todos los programadores trabajan con dos pantallas (¡o más!). Es de las mejores inversiones que pueden hacer, incluso antes de actualizar su PC. No deberían invertir demasiado, se pueden conseguir monitores nuevos y usados a muy buen precio.
-  Asegúrense de que su **webcam** y **micrófono** funcionen correctamente. No precisan nada sofisticado; en general, los que vienen integrados en una notebook son suficientes. A veces también es cómodo contar con unos auriculares o *headset*.



# Clases por videollamada (2/2)

## Recomendaciones (continuación):

- Cerrar programas de **Torrents** o aplicaciones que demanden mucho ancho de banda como Netflix o YouTube. Para medir la velocidad de su conexión a Internet sugerimos entrar a: <https://www.speedtest.net>.
- Desactivar las instalaciones automáticas del sistema operativo, como puede ser **Windows Update**.
- Reiniciar su **router** y/o **modem** un rato antes de cada clase. La forma más sencilla de hacer esto es “desenchufar y enchufar” el aparato. Esto puede ser útil para prevenir que la conexión a internet se tranque o se corte en medio de la clase.





# Uso del teclado



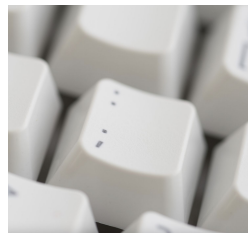
# Uso del teclado (1/4)

A la hora de programar, es necesario escribir caracteres especiales como:

< > \ / ( ) [ ] | { } # \$ & \* : ; + = " ' ` - \_

Por lo tanto, es fundamental identificar estos caracteres con rapidez.

La mejor recomendación que les podemos hacer es que configuren su teclado de tal manera que **cuando presionen un carácter en el teclado físico, dicho carácter aparezca en la pantalla.** 🙌 Por favor no intenten recordar nada de memoria.



Al presionar esta tecla, debería aparecer un “*punto y coma*” en su pantalla.

# Uso del teclado (2/4)

Si su teclado físico tiene las teclas en inglés 🇺🇸...



Generalmente  
porque no tiene eñe.

...recomendamos configurarlo como **U.S. International** 🙌

Ver instrucciones en la siguiente diapositiva.



# Uso del teclado (3/4)

Para los que tengan **teclado físico en inglés** 🇺🇸 recomendamos:

- Configurado en su sistema operativo como **U.S. International** o Estados Unidos Internacional. 🖱 [Ver video de configuración](#) en Windows 10.
- Borrar otros teclados de inglés/español que hayan configurado anteriormente, ya que la configuración **U.S. International** sirve tanto para escribir en inglés como en español. 🇪🇸 🇬🇧 🇪🇸 🇬🇧 🇪🇸

Gracias a esta configuración **no tendrán que estar cambiando el idioma** ni acordarse de memoria donde está cada carácter en el teclado.

# Uso del teclado (4/4)

- **Escribir rápido** con el teclado les ayudará mucho.  
Pruébense aquí: <http://www.typingtest.com>.
- Acostúmbrense a los **atajos** (shortcuts) del teclado:
  - CTRL + C = Copiar.
  - CTRL + X = Cortar.
  - CTRL + V = Pegar.
  - CTRL + S = Guardar.
  - CTRL + Z = Deshacer.
  - CTRL + T = Abrir nueva pestaña (tab).
  - CTRL + W = Cierra la pestaña actual.
  - F5 = Recargar Página.

En Mac: sustituir CTRL por CMD. Para recargar la página usar: CMD+R.



Vamos a empezar a usar el mouse  
cada vez menos.



# Requisitos previos



# Requisitos previos

- HTML y CSS.
- JavaScript:
  - Variables y tipos de datos.
  - Operadores.
  - If/Else.
  - For / ForEach.
- JavaScript (cont):
  - Funciones.  
Parámetros por referencia y por valor.
  - Callbacks.
  - Eventos.
  - Arrays (Arreglos).
  - Objetos.

Saber inglés  es una gran ventaja.



# Contenido del curso





# Contenido del curso

## Herramientas / Tecnologías:

- JavaScript (ES6+).
- Node.js.
- npm.
- Express.
- SQL y MySQL.
- Deployment.
- Consola / Shell.
- Git y GitHub.

## Conceptos:

- Rutas (Routing).
- Requests/Responses.
- Controladores.
- Modelos.
- Vistas.
- Middleware.
- Autenticación.
- Bases de datos.



# ECMAScript



# ECMAScript (1/2)

ECMAScript (ES) es una **especificación** (un standard) de lenguaje de programación creada por la organización [ECMA International](#), pero no es un lenguaje de programación propiamente dicho.

La especificación tiene varias **implementaciones** como JScript (Microsoft) y ActionScript (Adobe/Macromedia), pero la más famosa es **JavaScript**. Por lo tanto, actualmente, los términos “ECMAScript” y “JavaScript” se pueden usar (casi) indistintamente.

La especificación cuenta con varias versiones, siendo la primera de 1997 (ES1). En 2009 se publicó la versión 5 (**ES5**) que se mantuvo incambiada hasta el año 2015 y es compatible con la gran mayoría de los navegadores web desde Internet Explorer 9.




# ECMAScript (2/2)

A partir del año 2015, todos los meses de junio se publica una nueva versión de ECMAScript, la cual incorpora mejoras o nuevas funcionalidades al lenguaje.

- ES6 = ES2015.
- ES7 = ES2016.
- ...
- ES10 = ES2019.
- ES11 = ES2020.

A todas estas nuevas versiones se les llama comúnmente **ES6+** para hacer referencia a “JavaScript moderno”.

 **Importante:** tener presente que no todos los navegadores (en todas sus versiones) soportan las últimas funcionalidades del lenguaje. Afortunadamente, para solucionar este problema se puede usar una herramienta como [Babel](#).



# ES6+

(Primera Parte)

# ES6+



Algunas características de ES6+ que veremos son:

- Declaración de variables usando `var`, `let` y `const`.
- Arrow functions.
- Default values.
- Template Strings.
- Object y Array Destructuring.
- For Of Loop.
- Mejoras con Object Literals.
- Spread operator " . . . ".
- Imports / Exports.
- Promesas.
- Clases.

*\* Estas no son las únicas características nuevas en ES6.*



var, let *y* const



# Definición de variables: `var`, `let` y `const` (1/7)

En ES5 siempre se utiliza `var` para declarar variables. En el siguiente ejemplo, se podría suponer que en consola se imprimirá “Domingo”, dado que la asignación de “Lunes” ocurre dentro de un bloque. Sin embargo, la asignación persiste luego del bloque.

```
var mensaje = "Domingo";

if (true) {
    var mensaje = "Lunes";
}

console.log(mensaje); // Imprime Lunes.
```

Un **bloque** es cualquier pedazo de código delimitado por llaves `{ }`.

En ES5 existe el llamado *function scoping*, donde las variables existen únicamente dentro de las funciones donde fueron definidas.

Para los demás bloques (`for`, `if`, etc) no hay *scope* y el comportamiento es igual al lado izquierdo de la diapositiva .

```
var mensaje = "Domingo";

function diaDeLaSemana() {
    var mensaje = "Lunes";
}

console.log(mensaje); // Imprime Domingo.
```





## Definición de variables: `var`, `let` y `const` (2/7)

Dado que a `var` sólo le puede definir un alcance dentro de una función, el siguiente código funciona sin problema:

```
if (true) {  
    var mensaje = "Lunes";  
}
```

```
console.log(mensaje); // La variable existe. Se imprime Lunes.
```



# Definición de variables: var, let y const (3/7)

En ES6 se introduce la palabra clave `let` que sí tiene scope dentro de los bloques que no sean funciones (como un `if` o un `for`).

```
let mensaje = "Domingo";

{
  let mensaje = "Lunes"; // Sólo tiene efecto dentro del bloque.
}

console.log(mensaje); // Se imprime Domingo.
```

Ver [documentación](#) en MDN.



## Definición de variables: `var`, `let` y `const` (4/7)

Suele ser útil usar `let` en lugar de `var`, para evitar que la variable de iteración persista fuera del `for`. Ejemplo:

```
for (var i = 0; i <= 10; i++) {  
    console.log(i);  
}
```

```
console.log(i);  
// Funciona. Se imprime 11.
```

```
for (let i = 0; i <= 10; i++) {  
    console.log(i);  
}
```

```
console.log(i);  
// Error, porque i no está definida.
```



# Definición de variables: `var`, `let` y `const` (5/7)

Hasta ES5 cuando se desea declarar algo como **constante**, como buena práctica se declara en mayúsculas, pero eso no impide que se vuelva a modificar su valor en el código.

```
var PI = 3.1416;  
PI = 10;  
console.log(PI); // Se imprime 10.
```

En ES6 se introduce la palabra clave **const** para declarar variables que no se quiere que se les pueda modificar el valor. Son *Read-Only*.

```
const PI = 3.1416;  
PI = 10; // -> Esto tira un error de Read Only en consola.  
console.log(PI);
```

Ver [documentación](#) en MDN.



# Definición de variables: var, let y const (6/7)

Es importante notar que la variable no es constante, sino que es una **referencia constante**. Si se declara un **objeto con const**, igual se le puede asignar propiedades sin que tire error de Read-Only, pero no se puede re-asignar el objeto a un string por ejemplo.

```
const persona = {};  
persona.edad = 26; // Funciona.  
persona.nombre = "María"; // Funciona.  
persona = "Luis"; // NO funciona --> Tira un error de Read Only en consola.
```

Al igual que let, const sólo existe dentro del bloque donde es declarado.

```
if (true) { const precio = 120; }  
console.log(precio); // --> Tira un error.
```



# Definición de variables: `var`, `let` y `const` (7/7)

Entonces, ¿qué usar? No hay consenso al respecto, por un [criterio](#) bastante común es el siguiente:

- Usar `const` por defecto.
- Sólo usar `let` si es necesario hacer una reasignación (*rebinding*).
- Nunca usar `var` en ES6+.

👉 Usar `const` con un objeto asegura que la variable siempre referencie al mismo objeto. Pero los atributos del objeto pueden cambiar. En caso de necesitar que todo el objeto sea inmutable, se puede utilizar el método `Object.freeze()`.



# Arrow functions



# Arrow functions (1/2)

Es una nueva forma de **declarar las funciones anónimas**, usando el operador `=>` en lugar de la palabra `function`.

// ES5:

```
var crearMensaje = function(nombre, mensaje) {  
    var resultado = mensaje + " " + nombre;  
    return resultado;  
}
```

// ES6:

```
const crearMensaje = (nombre, mensaje) => {  
    const resultado = mensaje + " " + nombre;  
    return resultado;  
}
```

Ver [documentación](#) en MDN.





## Arrow functions (2/2)

Cuando se tiene sólo **una línea de código** dentro de la función, se puede escribir todo en una línea sin necesidad de escribir `return` ni de encapsular el cuerpo de la función entre llaves `{ ... }`

```
const crearMensaje = (nombre, mensaje) => mensaje + " " + nombre;
```

Y si la función sólo tiene **un parámetro**, no son necesarios los paréntesis:

```
const crearMensaje = nombre => "Bienvenido " + nombre;
```

```
const elevarAlCuadrado = x => x * x;
```

👉 Si la función no recibe parámetro, simplemente se colocan dos paréntesis `()`.



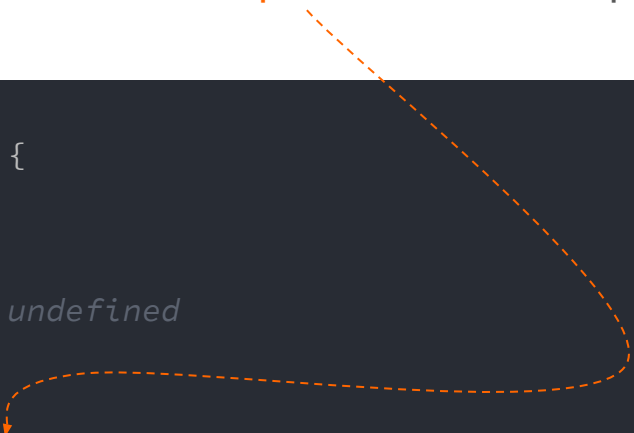
# Default values



# Default values

En ES6 se introduce la posibilidad de declarar **valores por defecto** en los parámetros de la declaración de una función.

```
function mostrarMensaje(mensaje, nombre) {  
  console.log(mensaje + ", " + nombre);  
}  
mostrarMensaje(); // Imprime: undefined, undefined  
  
// En ES6:  
function mostrarMensaje(mensaje, nombre = "María") {  
  console.log(mensaje + ", " + nombre);  
}  
mostrarMensaje(); // Imprime: undefined, María  
mostrarMensaje("Hola"); // Imprime: Hola, María
```





# Templates Strings

(también llamado Template Literals)



# Templates Strings (1/2)

Hasta ES5 es muy común el concatenar variables con strings de la siguiente forma.

```
let nombre = "María";  
let mensaje = "Hola " + nombre + ", ¡bienvenida!";  
console.log(mensaje); // Hola María, ¡bienvenida!.
```

En ES6 se puede utilizar el acento grave (backtick) ``` (Alt + 96) para envolver un *template string*, dentro del cual se puede acceder a evaluar JavaScript dentro de `${...}`

```
let nombre = "María";  
let mensaje = `Hola ${nombre}, ¡bienvenida!`;   
console.log(mensaje); // Hola María, ¡bienvenida!.
```

Ver [documentación](#) en MDN.



# Templates Strings (2/2)

También se pueden utilizar expresiones dentro de las llaves. Otro detalle es que respeta las líneas en blanco y los tabs que se agreguen al texto.

```
let x = 5;  
const ecuacion = `${x} * ${x} = ${x*x}`;  
console.log(ecuacion); // Imprime: 5 * 5 = 25
```

```
let nombre = "María";
```

```
let saludo = `
```

```
Hola
```

```
  ${nombre},
```

```
    bienvenida!`;
```

```
console.log(saludo);
```

Hola

María,

¡bienvenida!



# Object Destructuring



# Object Destructuring (1/5)

En ES5, para acceder a una propiedad dentro de un objeto, se utiliza el punto ".".

A partir de ES6 también se puede hacer a través de *destructuring*.

```
const persona = {  
  nombre: "María"  
}  
  
console.log(persona.nombre); // Imprime "María".  
  
// En ES6+, usando Destructuring:  
const { nombre } = persona;  
console.log(nombre); // Imprime "María".
```

Ver [documentación](#) en MDN.





# Object Destructuring (2/5)

Puede ser útil cuando se tiene un objeto con varias propiedades y sólo interesan algunas.

```
const persona = {  
  nombre: "María",  
  apellido: "Rodríguez",  
  ciudad: "Montevideo",  
  edad: 30  
};  
  
const { nombre, ciudad } = persona;  
console.log(nombre); // Imprime "María".  
console.log(ciudad); // Imprime "Montevideo".
```



# Object Destructuring (3/5)

Un típico caso es cuando se tiene **una función que retorna un objeto**, pero solo se requieren algunas de sus propiedades.

```
function getUsuario() {  
  return {  
    nombre: "María",  
    apellido: "Rodríguez",  
    ciudad: "Montevideo",  
    edad: 30  
  };  
}  
  
const { nombre, ciudad } = getUsuario();  
console.log(nombre); // Imprime "María".  
console.log(ciudad); // Imprime "Montevideo".
```



# Object Destructuring (4/5)

También es posible **cambiarle el nombre a las variables** que se crean con respecto al nombre que tienen las propiedades del objeto.

```
function getUsuario() {  
  return {  
    nombre: "María",  
    apellido: "Rodríguez",  
    ciudad: "Montevideo",  
    edad: 30  
  };  
}  
  
const { nombre:firstname, apellido:lastname } = getUsuario();  
console.log(firstname); // Imprime "María".  
console.log(lastname); // Imprime "Rodríguez".
```



# Object Destructuring (5/5)

Otro uso común, es usar Object Destructuring como parámetro de una función, cuando dicha función recibe como parámetro un objeto.

```
const persona = {  
  nombre: "María",  
  apellido: "López",  
  edad: 35,  
};  
  
function mostrarNombreCompleto({ nombre, apellido }) {  
  return nombre + " " + apellido;  
}  
  
mostrarNombreCompleto(persona);
```



# Array Destructuring



# Array Destructuring (1/2)

También es posible hacer *destructuring* de un Array (de hecho, es un objeto):

```
const jugadores =  
  ["María", "Luis", "José", "Ana", "Victoria"];
```

```
const [ jugador1, jugador2 ] = jugadores;  
const [ „jugador3, jugador4 ] = jugadores;
```

```
console.log(jugador1); // Imprime: "María".
```

```
console.log(jugador2); // Imprime: "Luis".
```



## Array Destructuring (2/2)

Algo útil de Array Destructuring, es la posibilidad de hacer *swap* de variables.

```
let jugadorA = "María";
```

```
let jugadorB = "Ana";
```

```
[jugadorA, jugadorB] = [jugadorB, jugadorA];
```

```
console.log(jugadorA); // Imprime "Ana".
```

```
console.log(jugadorB); // Imprime "María".
```



# For...Of Loop





# For...Of Loop

En ES6 se introdujo el For...Of Loop, que permite recorrer un array y otros elementos iterables (como un [String](#) y un [Map](#)), pero no para objetos.

```
const jugadores = ["María", "Luis", "José", "Ana", "Victoria"];

for (let jugador of jugadores) {
  console.log(jugador);
}
```

👉 Nota: El `forEach` sólo funciona con Arrays.



# Mejoras con Object Literals



# Mejoras con Object Literals (1/2)

Con ES6 es más fácil crear un objeto a partir de variables pre-existentes, cuando se quiere que las propiedades tengan el mismo nombre que dichas variables.

```
let nombre = "María";  
let apellido = "Rodríguez";  
  
// En ES5:  
var persona = { nombre: nombre, apellido: apellido };  
  
// En ES6:  
const persona = { nombre, apellido };
```

```
// Nota: Las variables podrían ser otros objetos.  
const equipo = "Barcelona";  
const fan = { persona, equipo };
```



# Mejoras con Object Literals (2/2)

En ES6 se pueden declarar métodos de un objeto, de una forma más corta.

```
// ES5:
var auto = {
  vender: function (precio) {
  }
}
```

```
// ES6:
const auto = {
  vender (precio) {
  }
}
```



# Spread Operator ...



# Spread Operator ... (1/4)

Este nuevo operador permite tomar un array y separarlo en cada uno de sus ítems. Ejemplo:

```
console.log([1,2,3]); // Imprime [1, 2, 3]
```

```
console.log(...[4,5,6]); // Imprime 4 5 6
```



# Spread Operator ... (2/4)

Problema: ¿cómo se podrían unir dos arrays?

Por las dudas, aclaramos que no funciona hacerlo de esta manera:

```
let primero = [1,2,3];  
let segundo = [4,5,6];  
primero.push(segundo);  
console.log(primero); // Imprime [1, 2, 3, [4, 5, 6]].
```

Pero usando el Spread Operator, ¡sí funciona!

```
let primero = [1,2,3];  
let segundo = [4,5,6];  
primero.push(...segundo);  
console.log(primero); // Imprime [1, 2, 3, 4, 5, 6].
```



# Spread Operator ... (3/4)

El operador funciona muy bien para **clonar** un array:

```
const marcas = ["Kia", "Volvo", "Peugeot", "Chevrolet", "Fiat"];

const nuevasMarcas = [...marcas];
```





# Spread Operator ... (4/4)

También se puede utilizar para separar un array en los parámetros que recibe una función.

```
let primero = [1,2,3];  
let segundo = [4,5,6];  
  
function sumarTresNumeros(a, b, c) {  
  console.log( a + b + c );  
}  
  
sumarTresNumeros(...primero); // Imprime 6.  
sumarTresNumeros(...segundo); // Imprime 15.
```



Imports – Exports



# Imports – Exports (1/3)

En ES6 se pueden exportar e importar valores y funciones, sin necesidad de acudir a *global namespaces*. Para ello se utilizan las palabras clave `export` e `import`.

En archivo `utils/math.js`:

```
function cuadrado(a) { return a*a; }  
  
export { cuadrado };
```

En archivo `utils/math.js`:

```
// También se puede exportar así:  
  
export function cuadrado(a) { return a*a; }
```

En archivo `index.js`:

```
import { cuadrado } from "utils/math.js";  
  
// Finalmente, se utiliza la función.  
console.log(cuadrado(5)); // Imprime: 25.
```

Ver [documentación](#) en MDN.



# Imports – Exports (2/3)

Otros ejemplos:

En archivo `utils/math.js`:

```
export function cuadrado(a) { return a*a; }  
export function multiplicar(a,b) { return a*b; }
```

En archivo `index.js`:

```
import { cuadrado as miCuadrado, multiplicar } from "utils/math.js";  
  
console.log(miCuadrado(5)); // Imprime 25.  
console.log(multiplicar(5, 3)); // Imprime 15.
```



# Imports – Exports (3/3)

También es posible importar **todas** las funciones de un archivo (externo) utilizando un **asterisco** (\*), lo cual es útil si se tienen muchas funciones y no se quiere importar una por una.

Siguiendo con el ejemplo anterior:

En archivo `index.js`:

```
import * as math from "utils/math.js";  
  
console.log(math.cuadrado(5)); // Imprime 25.  
console.log(math.multiplicar(5, 3)); // Imprime 15.
```



# Ejercicios

(para practicar JavaScript 💪)



# Ejercicio 1

Crear una función llamada `removeVocales` que reciba como parámetro un string y retorne un nuevo string que sea igual que el recibido pero sin las vocales.

Ejemplos:

Input	Output
<code>removeVocales("Hola")</code>	<code>"Hl"</code>
<code>removeVocales("Hola Mundo")</code>	<code>"Hl Mnd"</code>
<code>removeVocales("Hola URUGUAY")</code>	<code>"Hl RGY"</code>



## Ejercicio 2

Crear una función llamada `encontrarImpar` que reciba como parámetro un array de números enteros (positivos y/o negativos) y retorna el número del array que aparezca un número impar de veces.

Siempre se recibirá un array con un sólo número con esas características.

Ejemplos:

Input	Output
<code>encontrarImpar([8])</code>	8
<code>encontrarImpar([2,2,2,2,8,2,2])</code>	8
<code>encontrarImpar([20,1,-1,2,-2,3,3,5,5,1,2,4,20,4,-1,-2,5])</code>	5





## Ejercicio 3

Crear una función llamada `sumarMultiplos` que reciba como parámetro un número natural (entero positivo) y retorne la suma de todos los números múltiplos de 3 o 5, que sean menores al número recibido por parámetro.

Si un número fuese múltiplo de 3 y 5, se considerará una sola vez.

Ejemplos:

Input	Output
<code>sumarMultiplos(1)</code>	0
<code>sumarMultiplos(2)</code>	0
<code>sumarMultiplos(3)</code>	0
<code>sumarMultiplos(6)</code>	8
<code>sumarMultiplos(10)</code>	23