



Curso de React & React Native

Material Previo



Objetivos del curso



Objetivos del curso

- Profundizar conocimientos de Front-End utilizando **React**, que es una librería open-source creada por Facebook para desarrollar interfaces de usuario muy potentes.
- Aprender a separar la UI en **componentes reutilizables**.
- Desarrollar **aplicaciones móviles** nativas usando JavaScript.



Software para la primera clase



Software para la primera clase

- Instalar **Google Chrome** (<https://www.google.com/chrome>).
- Instalar Microsoft **Visual Studio Code** (<https://code.visualstudio.com>).
- Instalar **Node.js** (<https://nodejs.org/en>) – Versión 16 LTS.
- Crearse una cuenta de **GitHub** (<https://github.com>).



Cuidado con las actualizaciones automáticas del sistema operativo. Sobre todo en Windows. Eviten actualizar su notebook 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.
- 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.dev** (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.



Requisitos del curso



Requisitos del curso (1/2)


La siguiente es una lista de temas de **JavaScript** que el alumno debería dominar antes de anotarse al curso de React:

- Tipos de datos.
- Operadores.
- If/Else.
- Funciones.
Parámetros por referencia y por valor.
- Scope global y local.
- *Callbacks*.
- Eventos.
- *Arrays* (Arreglos).
- Objetos.
- Novedades de ES6+. Ejemplos:
 - *Arrow Functions*.
 - `const` y `let`.
 - *Spread Operator*.
 - Promesas. `Async/Await`.
- Métodos de *arrays* como `map`, `filter` y `reduce`.
También `for` y `for..of`.



Requisitos del curso (2/2)

Además:

- Dominio de HTML y CSS.
- Nociones básicas de Node.js.
- Nociones básicas de uso de la Consola / Shell.
- Git y GitHub.
- JSON.
- Saber inglés  es una gran ventaja.



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.
- ...
- ES11 = ES2020.
- ES12 = ES2021.

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](#).



Repaso de ES6+



Repaso de ES6+

Algunas características de ES6+ que se repasan en este material 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 `()`.



Arrow functions y `this`



Arrow functions y `this` (1/4)

Las *arrow functions* no “tienen” su propio `this`, como sí lo tienen las funciones tradicionales. Esto evita problema comunes en JavaScript como el siguiente:

```
function Persona() { // Función "constructora" ≈ "Clase".
    this.edad = 0; // Atributo de la "Clase", inicializado en cero.
    setInterval(function crecer() {
        this.edad++; // Este `this`, no hace referencia al `this` de Persona. 🤖
    }, 1000);
}

var p1 = new Persona();
p1.edad; // Vale cero, siempre. 🤖
```

⚠ El concepto de `this` en JavaScript es bastante particular. No siempre queda claro a qué hace referencia el `this`. Recomendamos leer [esto](#), [esto](#) y [esto](#).

👉 Nota, el nombre `crecer` no era necesario. Se podría haber omitido y dejar la función anónima. Igual esto no cambia el problema.



Arrow functions y `this` (2/4)

Si la función que se le pasa a `setInterval` se define con una *arrow function* no se tiene ese problema, ya que el valor de `this` dentro del *arrow function* se mantiene igual al del contexto actual.

```
function Persona() {  
    this.edad = 0;  
    setInterval(() => {  
        this.edad++; // Ahora, `this` apunta al objeto Persona.  
    }, 1000);  
}  
  
var p1 = new Persona();  
p1.edad;
```




Arrow functions y `this` (3/4)

👉 Nota: Antes de existir las *arrow functions*, el problema anterior se resolvía de la siguiente forma:

```
function Persona() {  
    var self = this; // También es común usar `that` en lugar de `self`.  
    self.edad = 0;  
    setInterval(function crecer() {  
        self.edad++;  
    }, 1000);  
}
```



Arrow functions y `this` (4/4)

👉 Nota: No siempre es aconsejable usar *arrow functions*. De hecho, hay casos en los que ni siquiera otorgan el comportamiento deseado. Por ejemplo: a la hora de crear **métodos**. En estos casos, lo mejor es usar una función común y corriente.

```
const persona = {  
  nombre : "María",  
  apellido: "López",  
  nombreCompleto: () => {  
    return this.nombre + " " + this.apellido;  
  }  
}  
  
persona.nombreCompleto(); // Retorna `undefined`. 🤖
```

En este caso, `this` apunta al objeto `Window`.



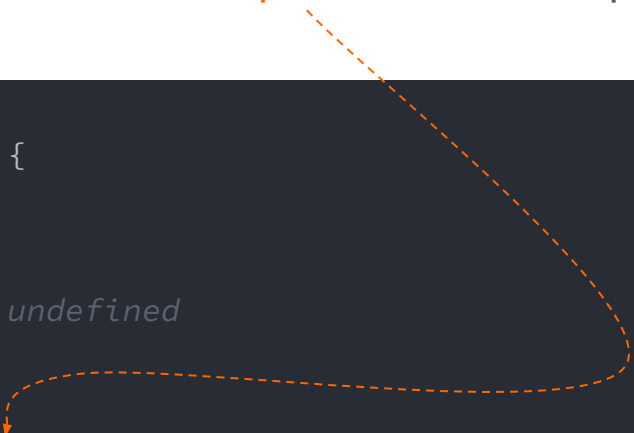
Default values



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 *destructing* de un *array* (de hecho, es un objeto):

```
const jugadores =  
    ["María", "Luis", "José", "Ana", "Victoria"];  
  
const [ jugador1, jugador2 ] = 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 (const 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.
```



Promesas



Promesas

Las promesas (en inglés: *promises*) fueron creadas para **dar consistencia** y **estandarizar** el manejo de operaciones **asíncronas**.

Hasta ahora, la forma más usual de tener comportamiento **asíncrono** en JavaScript era a través de los famosos ***callbacks***, los cuales funcionan bien pero la forma de usarlos depende mucho de cada implementación. Por ejemplo, cada librería puede implementar *callbacks* de forma diferente y eso implica tener que apoyarse mucho en cada documentación, lo cual no es ideal. Además, era fácil caer en un ***Callback Hell***.

Documentación:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.



Ejemplos de *callbacks* en jQuery

Si bien en este curso no se usará la librería jQuery, se la usará como ejemplo en esta diapositiva.

Para los que no saben, jQuery es una librería JavaScript creada en el año 2006 y que fue, y en cierta manera sigue siendo, [sumamente popular](#).

jQuery incluye una función asíncrona llamada `ajax` que permite hacer llamados HTTP. Los creadores de la librería decidieron implementar dicha función de tal forma que reciba como parámetro un objeto con un método llamado `success`, que es una función *callback* que se ejecutará cuando la llamada finalice exitosamente.

Si bien esta sintaxis propuesta por los creadores de jQuery funciona, la misma no sigue ningún estándar. Por ejemplo, en lugar de llamarle `success` al método, le podrían haber puesto `successful` u `ok`. Otras librerías podrían haber elegido otras nomenclaturas y/o sintaxis. Justamente **este tipo de situaciones son las que se quieren evitar con las *promesas***.

```
$.ajax({  
  url: "https://swapi.dev/api/people/1/?format=json",  
  success: function(datosObtenidos) {  
    console.log("Nombre del personaje: " + datosObtenidos.name);  
  }  
});
```

Promesas (cont)

Una promesa es algo que sucederá en algún **momento futuro**.

Es un **objeto** que representa la eventual completitud (o falla) de una operación asíncrona, y el resultado de dicha operación.

El uso de promesas se resume a dos acciones:

1. **Crear** una promesa para exponer una funcionalidad asíncrona. Esto generalmente lo hace el creador de la librería o paquete que quiere exponer la funcionalidad. No suele ser algo que tengamos que hacer nosotros.
2. **Consumir** una promesa ya creada. 🙌 Lo que comúnmente hacemos.



Promesas (Promise): Creación (1/2)

Sintaxis:

```
const promesa = new Promise((resolve, reject) => {  
    // Hacer algo, probablemente asíncrono...  
  
    if (/* Si todo salió bien */) {  
        resolve("¡Funcionó! El resultado es...");  
    } else {  
        reject(Error("Hubo un error"));  
    }  
});
```

⚠ En general, esto no es algo que debemos hacer nosotros. Esto lo suele hacer el creador de la librería o paquete que desea exponer una funcionalidad asíncrona.



Promesas (Promise): Creación (2/2)

Ejemplo:

```
const usuarioPromesa = new Promise((resolve, reject) => {  
  const usuario = obtenerUsuarioPorId(2); // Llamada a BD.  
  if (usuario) {  
    resolve(usuario);  
  } else {  
    reject(Error("Hubo un error. No se pudieron obtener los datos"));  
  }  
});
```

⚠ En general, esto no es algo que debamos hacer nosotros. Esto lo suele hacer el creador de la librería o paquete que desea exponer una funcionalidad asíncrona.



Promesas (Promise): Consumo (1/2)

Ejemplo:

```
usuarioPromesa
  .then(usuario => console.log(usuario))
  .catch(error => console.log(error));
```

A cualquier operación que retorne una promesa al ejecutarse, se le puede concatenar un `.then()` y un `catch()`. Todas las promesas se comportan igual, por lo que la sintaxis no depende del creador de la librería o paquete.

Esto otorga **consistencia** para consumir resultados de operaciones asíncronas.



Promesas (Promise): Consumo (2/2)

Otro ejemplo:

```
fetch("https://swapi.dev/api/people/1/?format=json")  
  .then(respuesta => respuesta.json())  
  .then(personaje => console.log(personaje))  
  .catch(error => console.log(error));
```

👉 Nota: El método `fetch` es algo nuevo que reemplaza a `XMLHttpRequest` (que es la forma en la que tradicionalmente se realizaban las llamadas AJAX). Pero no es un concepto de ES6.

Este método, recibe como argumento una ruta (*path*) y retorna una `Promise`. A su vez, el método `respuesta.json()` retorna otra `Promise`, por lo que fue necesario encadenar otro `then`. Se pueden encadenar tantos `then` como sea necesario, con la gran ventaja de que quedan ordenados uno abajo del otro, en lugar de producir un *Callback Hell* como sucedía antes. El método `catch` se ejecuta en caso de que haya ocurrido algún error en el camino.



Async/Await



Async/Await (1/2)

A la hora de trabajar con **promesas**, en lugar de usar la sintaxis **then/catch** (ES6) se puede usar la sintaxis **async/await** (ES7). Son equivalentes.

Ahora, el ejemplo de la diapositiva anterior (`fetch`), que se escribió con `then/catch`, se puede re-escribir de la siguiente manera con `async/await`.

```
async function obtenerPersonajeDeStarWars() {  
  const respuesta = await fetch("https://swapi.dev/api/people/1/?format=json");  
  const personaje = await respuesta.json();  
  console.log(personaje);  
}  
  
obtenerPersonajeDeStarWars();
```

Tener en cuenta que si se desea "atrapar" ("catchear") un posible error que surja en durante la operación, en el caso de usar `async/await`, habría que agregar un `try/catch`. Ver la siguiente diapositiva.



Async/Await (2/2)

Ejemplo completo incluyendo `try/catch`:

```
async function obtenerPersonajeDeStarWars() {  
  try {  
    const respuesta = await fetch("https://swapi.dev/api/people/1/?format=json");  
    const personaje = await respuesta.json();  
    console.log("Personaje", personaje);  
  } catch (error) {  
    console.log("Ocurrió un error en el fetch", error);  
  }  
}
```

Como con todo parámetro, el nombre de parámetro `error` es arbitrario. Se le podría haber puesto `e` o `err`.



Classes

Clases

Las clases fueron introducidas a JavaScript con ES6 con el fin de hacer más amigable la Programación Orientada a Objetos para programadores acostumbrados a otros lenguajes como Java o C#. Pero en el fondo, se siguen usando los Prototypes.

```
class Alumno {  
    nombreCompleto;  
}  
  
const a1 = new Alumno;  
a1.nombreCompleto = "María";
```

Clases – Constructor

Ejemplo de un constructor.

```
class Alumno {  
  nombreCompleto;  
  constructor (nombre, apellido) {  
    this.nombreCompleto = nombre + " " + apellido;  
  }  
}  
  
const a2 = new Alumno("María", "Rodríguez");
```

👉 Nota: Las propiedades de clase, como `nombreCompleto`, no son parte de ES6 sino de ES7.



Clases – Herencia

Ejemplo de una herencia entre Persona y Alumno.

```
class Persona {  
    cedulaDeIdentidad;  
    nombreCompleto;  
    constructor (nombre, apellido) {  
        this.nombreCompleto = nombre + " " + apellido;  
    }  
}
```

```
class Alumno extends Persona {  
    cursos = [];  
    constructor(nombre, apellido, cursos) {  
        super(nombre, apellido);  
        this.cursos = cursos;  
    }  
}  
  
let a1 = new Alumno("María", "Rodríguez", []);
```



npm

npm (1/5)

Es equivalente a **Composer** en PHP o **NuGet** en .NET.



- npm es el **gestor (o manejador) de paquetes** que viene con Node.
- Tiene un **registro público** donde desarrolladores pueden publicar paquetes, para que el resto de la comunidad pueda usarlos.
- Además, incluye una **cli** (*command-line interface*) que permite usar npm desde la consola para instalar y publicar dichos paquetes.
- npm Inc. es una empresa privada.
- La palabra npm de por sí, no es una sigla y no obedece a nada. Pero le suele decir *node package manager*.
- Link: <https://www.npmjs.com>.





npm (2/5)

- npm también sirve para **dar comienzo a un proyecto** de Node.
- Esto se hace mediante el comando: `npm init`.
- Al hacerlo, el cli nos hará algunas preguntas para facilitar la creación del proyecto, aquellas que no queremos responder o que no sabemos cómo, simplemente las ignoramos.
- Al terminar, tendremos un nuevo archivo, llamado `package.json`.




npm (3/5) – package.json

El archivo `package.json` contendrá toda la **información necesaria sobre nuestro proyecto** como ser nombre, versión, descripción, autores, contribuidores, licencia, *punto de entrada*, y principalmente: **dependencias** y **scripts**.

Aquí un ejemplo de un archivo `package.json` creado con el comando `npm init`:

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

npm (4/5)

- Se utilizará npm para instalar **dependencias**.
- Una dependencia es un **módulo externo** a nuestra aplicación la cual sólo funcionará en tanto dicho módulo esté instalado. En nuestro caso, éstos módulos externos provienen del registro público de npm.
- Las dependencias en los proyectos Node se instalan en un directorio, ubicado en la raíz del proyecto, llamado **node_modules**.
-  El directorio **node_modules** no debe ser editado jamás por nosotros. La estructura de carpetas y los archivos que hay allí dentro sólo deben ser manejados por npm.



npm (5/5)

Para **instalar dependencias**, tenemos que estar ubicados en un proyecto de Node ya iniciado (es decir, que fue creado previamente usando el comando `npm init`) y luego ejecutar el comando:

```
npm install <nombre_del_modulo_externo>
```

Este comando consiste de:

- Invocar `npm`.
- Comando `install` (o simplemente `i`).
- Nombre del módulo externo (dependencia) que se desea instalar, tal como aparece en npmjs.com.

👉 Este comando instalará la dependencia en el directorio `node_modules` y agregará una entrada en el archivo `package.json`.



Buenos hábitos

a la hora de escribir código



Buenos hábitos a la hora de escribir código

- Cuidar la **indentación** de los archivos.
- Usar **espacios** adecuadamente.
- No olvidar los punto & coma (por más de que no sean obligatorios).
- Usar **nombres descriptivos** para variables, funciones y argumentos (parámetros). Evitar nombres como `arg1`, `arg2` que no dicen nada.
- Ser **consistentes** con cierto estilo de código. Elegir un estilo y respetarlo.
- Evitar el uso de variables globales.

Se sugiere usar la extensión de Visual Studio Code llamada [Prettier](#).

👉 Pueden leer más sobre **buenas prácticas** de programación JavaScript [aquí](#).