Clase 3

JavaScript parte 1

IIC2513 - Tecnologías y Aplicaciones Web

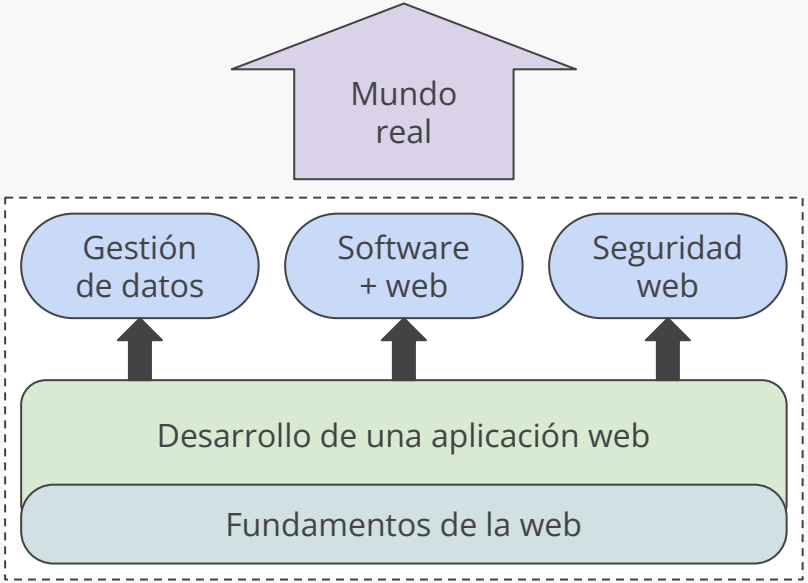
Antonio Ossa Guerra
aaossa@ing.puc.cl

1. Respondan la primera ECA (hoy)
2. Esta semana sí habrá ayudantía

Anuncios

Contenidos del curso

(1) Fundamentos de la web
(2) Desarrollo de una app web
(3) Gestión de datos
(4) Software + web
(5) Seguridad web
(6) El mundo real

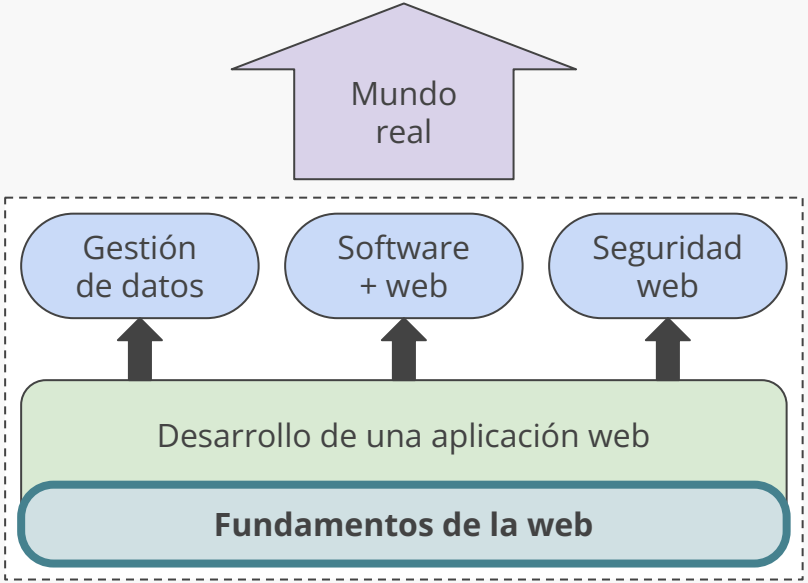


Contenidos del curso

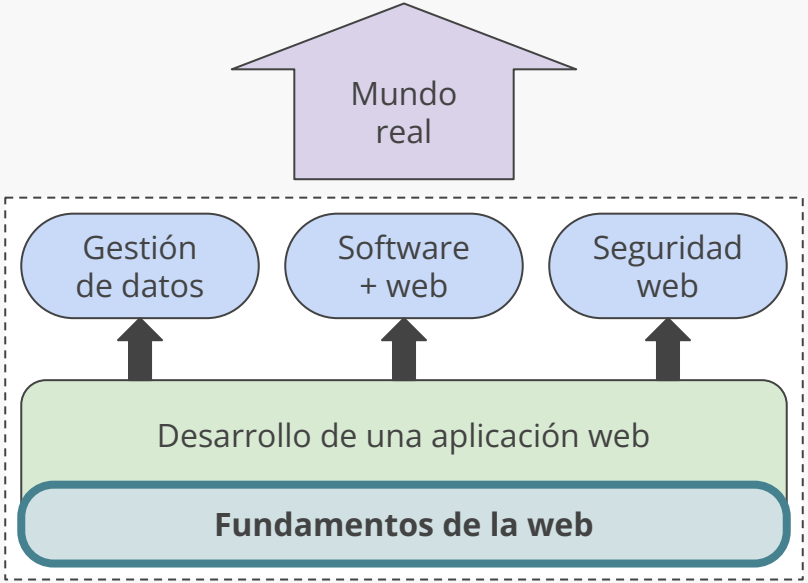
Fundamentos de la web

Veremos definiciones relacionadas con internet, protocolos y qué son HTML, CSS y **JavaScript**...

... para comprender la web como una **plataforma de desarrollo**, su funcionamiento básico, y protocolos y **tecnologías** detrás de esta



Contenidos del curso



Contenidos del curso



Contenidos del curso

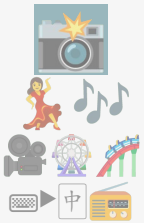


Contenidos del curso

Pregunta#¿Qué es JavaScript y cómo se relaciona con el mundo del desarrollo web?

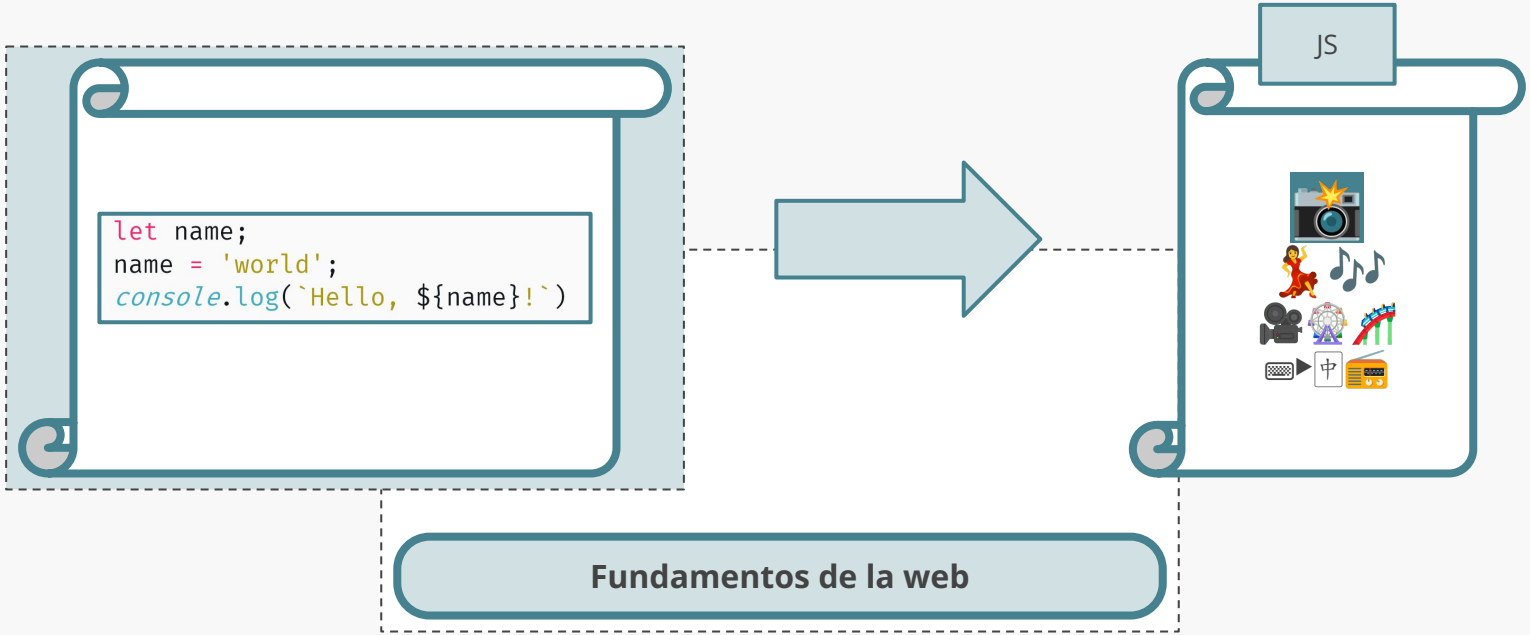
Título
Sección
Contenido
y más

Título
Sección
Contenido
y más



Fundamentos de la web

Contenidos del curso



Agenda

✨ JavaScript (JS)

Importante:

Esta clase es una **introducción a JavaScript**. Hay muchos elementos y *features* del lenguaje que no están incluidos pero que iremos viendo en clases y cápsulas durante el semestre...

Este es el **punto de partida** 📍, por lo que la invitación es a que practiquen por su cuenta y se familiaricen con JavaScript para usarlo **en el curso y más allá** 🏁

Agenda

✨ **JavaScript (JS)**

- Variables
- Tipos de datos
- Control de flujo
- Funciones
- Elementos modernos

JavaScript

JavaScript (JS) es un lenguaje de programación interpretado. Fue inventado en 1995 por Brendan Eich, desarrollado para el navegador Netscape 2, y se convirtió en un estándar ECMA en 1997

***Importante**:*

JavaScript y Java no se relacionan en nada...
"se parecen tanto como casilla a silla..."




```
// Este es un pequeño programa...

// ... con un mensaje...
let mensaje = 'Hola mundo!';

// ... que vamos a imprimir
console.log(mensaje);
```

Historia temprana de JS

- Primera versión por Brendan Eich para Netscape con el nombre **Mocha** en 1995
- Se renombró a **LiveScript** y se incluyó en versiones beta de Netscape 2 (Sept. 1995)
- Se renombró a **JavaScript** cuando Netscape agregó compatibilidad con Java en su navegador, al mismo tiempo que Netscape se unió a Sun (hoy Oracle, dueño de la marca Java), con la idea de que JavaScript fuera un complemento que acompañe a Java
- Como **JavaScript** es una marca registrada de Oracle, al definir el estándar en ECMA, se utilizó el nombre **ECMAScript**

JavaScript desde entonces

- ECMAScript (ECMA-262, 1997)
- ECMAScript, 2nd Edition (1998)
- ECMAScript, 3rd Edition (1999)
- ECMAScript, 4th Edition (iniciada en 2003) **abandonada en 2008**
  - El comité que acordaba ECMAScript 4 se dividió en dos bandos:
    - ECMAScript 3.1: Microsoft y Yahoo querían un cambio incremental
    - ECMAScript 4: Adobe, Mozilla, Opera y Google querían un gran *upgrade*
  - La solución fue renombrar ECMAScript 4 a ES5, que ES5 fuera un cambio incremental sobre ES3, que las *features* de ECMAScript 4 fueran recogidas en futuras versiones, y que el comité desarrolle una nueva versión más ambiciosa que ES5 (ES6 o “Harmony”)
- ECMAScript, 5th Edition (2009)
- ECMAScript, 6th Edition... luego nombrada ECMAScript 2015 (2015):
  - Declaración de clases, módulos, iteradores, generadores, f. flecha

Fuente: *ECMAScript version history | Wikipedia*

JavaScript desde entonces

- ECMAScript 2016 (2016)
  - await y async para programación asíncrona (para ES2017)
- ECMAScript 2017 (...)
  - Uso de generadores y promesas para programación asíncrona
- ECMAScript 2018
- ECMAScript 2019
- ECMAScript 2020
  - Introducción del tipo BigInt y operador “nullish coalescing” (??)
- ECMAScript 2021
- ECMAScript 2022
- ECMAScript 2023
- ECMAScript 2024
- **ES.Next** siempre se refiere a la siguiente versión (lanzada en junio)[1]

Fuente: *ECMAScript version history | Wikipedia*

Extra

¿Cómo se introducen cambios en EcmaScript?

TC39 es un grupo de desarrolladores, implementadores, académicos y otros especialistas que colaboran con la comunidad para mantener y evolucionar la definición de JavaScript

Las propuestas se manejan por etapas (*stages*), siguiendo The TC39 Process. Todas las propuestas, al igual que el estándar, son abiertos:

- tc39/proposals: Tracking ECMAScript Proposals
- Ejemplo de propuesta en *stage 3*:
  - GitHub - tc39/proposal-temporal: Provides standard objects and functions for working with dates and times
  - Notas durante la última presentación ante el TC39
  - Presentación utilizadas en dicha reunión
  - Artículo en mdn: JavaScript Temporal is coming | MDN Blog



Fuente: *TC39*

Lecturas recomendadas

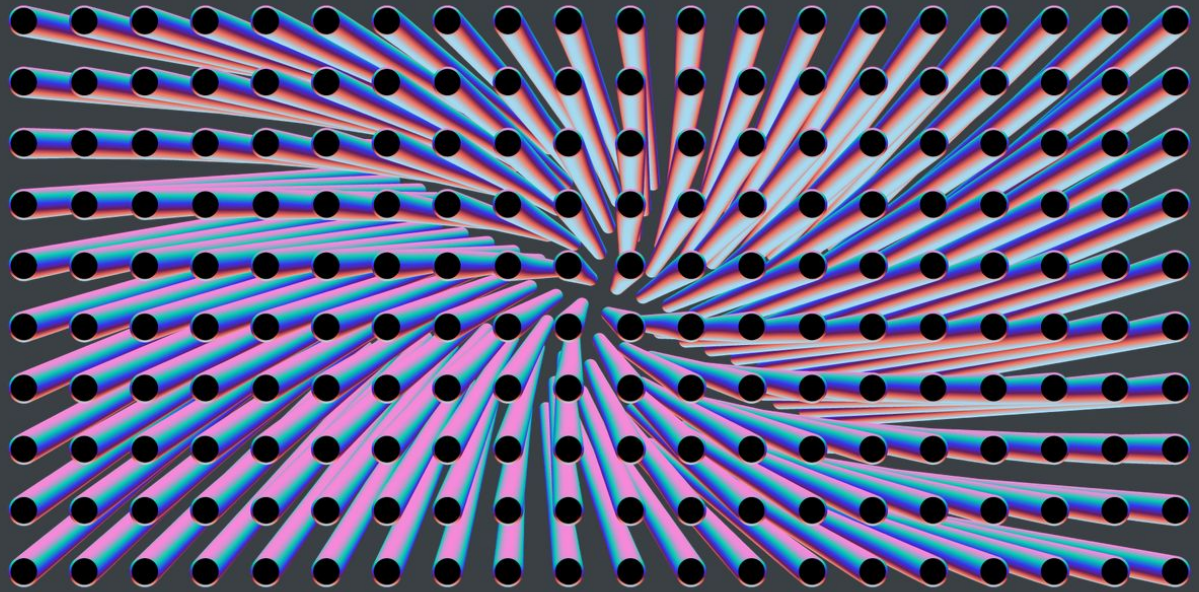
Deno v. Oracle (arco en desarrollo sobre liberar la marca JavaScript):

- javascript.tm: “Oracle, it’s time to free JavaScript”. Carta abierta a Oracle.
- Deno v. Oracle: Canceling the JavaScript Trademark: Petición formal para cancelar la marca ante la USPTO
- Oracle justified its JavaScript trademark with Node.js—now it wants that ignored: Oracle pide a la USPTO descartar la petición

2024 Stack Overflow Developer Survey: Resultados de la encuesta anual de StackOverflow

2024 Developer Survey

Products



In May 2024, over 65,000 developers responded to our annual survey about coding, the technologies and tools they use and want to learn, AI, and developer experience at work. Check out the results and see what's new for Stack Overflow users.

Developer Profile | Technology | AI | Work | Community | Professional Developers | Methodology

Methodology

65,437

Developer profile → Key territories

Top 10 countries

Community

Advertising & Talent

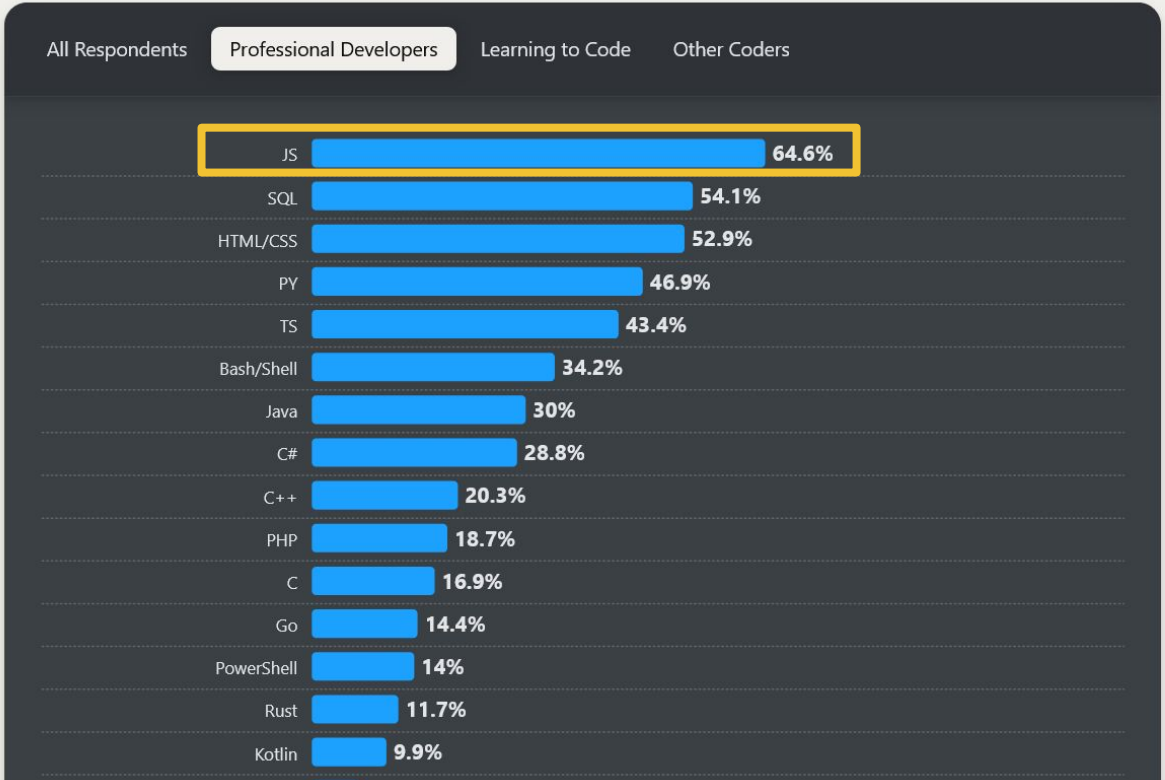
[2024 Stack Overflow Developer Survey](): Resultados de la encuesta anual de StackOverflow

Most popular technologies 2.1

Programming, scripting, and markup languages

JavaScript has been a mainstay in the developer survey and on Stack Overflow since our first survey. The most popular programming language has been JavaScript every year we have done the survey except for 2013 and 2014, when SQL was the most popular language.

Which **programming, scripting, and markup languages** have you done extensive development work in over the past year, and which do you want to work in over the next year? (If you both worked with the language and want to continue to do so, please check both boxes in that row.)



Octoverse 2024: The state of open source: Reporte anual de actividad en GitHub



Octoverse 2024: The state of open source: Reporte anual de actividad en GitHub

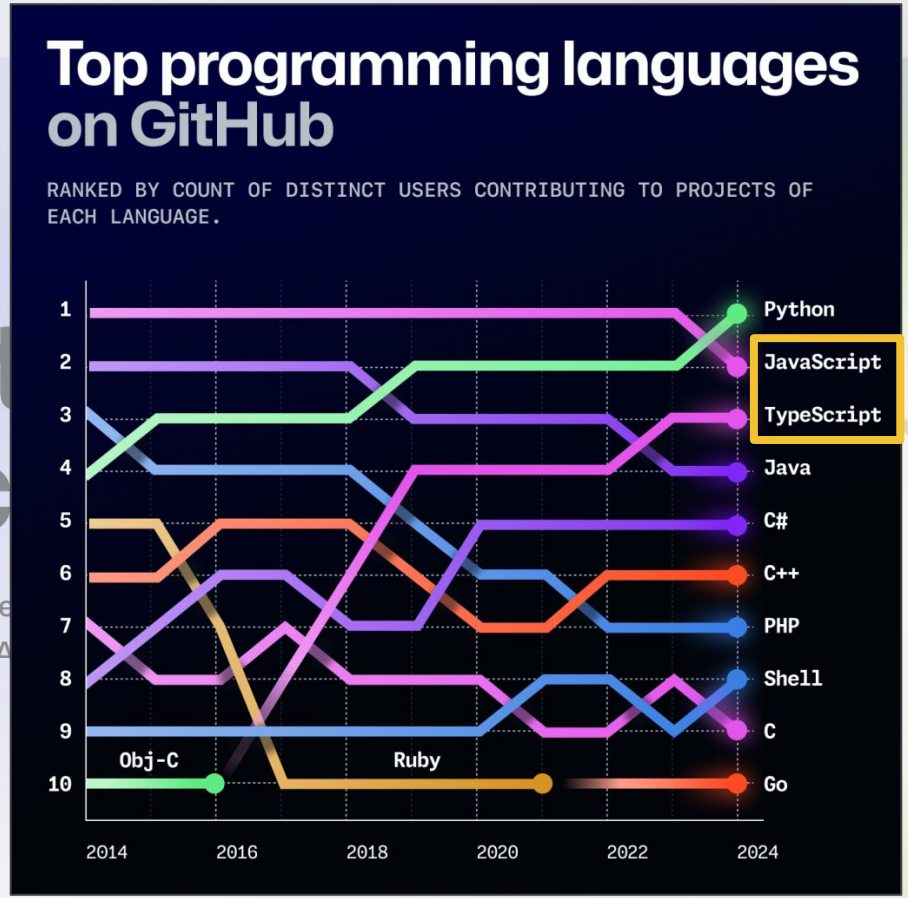
Start a free trial

OCTOVERSE 2024

The st
sourc

In this year's Octoverse
on GitHub shows how A
surges in size.

Read now →



State of JS: Encuesta anual a desarrolladores sobre el ecosistema JavaScript

Other Surveys: State of AI State of CSS State of Devs State of GraphQL State of HTML State of React

State of JavaScript

The annual developer survey of the JavaScript ecosystem

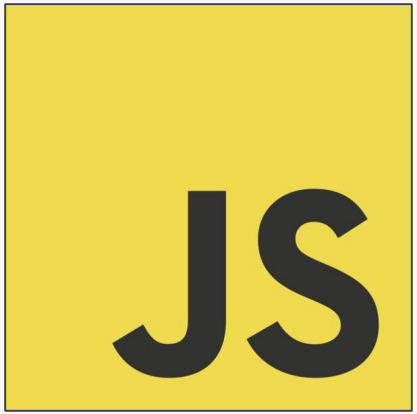
MOST RECENT 2024



View Questions View Results

¿Cómo es JavaScript como lenguaje?

- Es un lenguaje de *scripting*: **interpretado** como Python, es decir, “llegar y usar”
- Como todo lenguaje, JavaScript tiene partes buenas y no tan buenas (**¿mañas?**)
- Está en **constante extensión** (versión cada año)
- En sintaxis, es más **similar a C** que a otra cosa
- Es de **tipado dinámico**: variables tienen el tipo del último valor dado



¿Cómo es JavaScript como lenguaje?

Notar que JavaScript, para evitar errores producidos por conversión de tipos, intenta realizar conversiones apropiadas para llegar a un resultado

Aunque JS no muestre errores... hay que cuidarse de mezclar tipos de datos

```
1 "a" + 10;          // "a10"
2 "1" + 10;          // "110"
3 [] + 10;           // "10"
4 [2] + 10;          // "210"
5 [2, 2, 7] + 10; // "2,2,710"
6 +"2" + 10;         // 12
7 -"1" + 10;         // 9
8 -"a" + 10;         // NaN
```

TypeScript

TypeScript (TS) es superset de JavaScript, desarrollado y mantenido por Microsoft, que añade tipos estáticos y objetos basados en clases

Implementa una formalización de los tipos de JavaScript, permitiendo definir variables y funciones tipadas




```
const user = {
  firstName: 'Antonio',
  lastName: 'Ossa',
  role: 'Teacher',
};

console.log(user.name);
```

Property 'name' does not exist on type '{ firstName: string; lastName: string; role: string; }'

TypeScript

TypeScript (TS) es superset de JavaScript

Clase 16#*Tecnologías adicionales*

Implementa una formalización de los tipos de JavaScript, permitiendo definir variables y funciones tipadas




```
  lastName: 'Ossa',
  role: 'Teacher',
};

console.log(user.name);
Property 'name' does not exist on type '{ firstName: string;
lastName: string; role: string; }'
```

Agenda

JavaScript (JS)

- ✨ **Variables**
- Tipos de datos
- Control de flujo
- Funciones
- Elementos modernos

(Declaración vs. inicialización)

En JavaScript, hacemos la distinción entre **declaración** e **inicialización**

La **declaración** permite que la variable sea accesible sin lanzar un `ReferenceError`, mientras que la **inicialización** le asigna un valor

Si una variable se declara pero no se inicializa, su valor será `undefined`

```
// Declaración
let variable;
// Inicialización
variable = 17;

// Declaración e inicialización juntas
let variable = 17;

// Ambos códigos son equivalentes
```

Cómo declarar variables

`let`: Declara una variable local **en el contexto del bloque en que se define**, y opcionalmente la inicia a un valor

`const`: Declara e inicializa un nombre de constante de **solo lectura** y **dentro del contexto del bloque en que se define**

`var`: Declara una variable, opcionalmente la inicia a un valor (💀)

```
// Variable (let)
let valorQueCambia = 1;
valorQueCambia = 4;

// Constante (const)
const valorFijo = 1;
valorFijo = 4; // ERROR

// Variable (var) [no recomendado]
var valorConVar = 1;
valorConVar = 4;
```

Fuente: *Gramática y Tipos - JavaScript | MDN*

Cómo declarar variables

`let`: Declara una variable local **en el contexto del bloque en que se define**, y opcionalmente la inicia a un valor

`const`: Declara e inicializa un nombre de constante de **solo lectura** y **dentro del contexto del bloque en que se define**

`var`: Declara una variable, opcionalmente la inicia a un valor (💀)

Al usar `var` o `let`, podemos separar declaración de inicialización, pero en el caso de `const` estamos obligados a inicializar

```
// Variable (let)
let valorQueCambia = 1;
valorQueCambia = 4;

// Constante (const)
const valorFijo = 1;
valorFijo = 4; // ERROR

// Variable (var) [no recomendado]
var valorConVar = 1;
valorConVar = 4;
```

Fuente: *Gramática y Tipos - JavaScript | MDN*

Cómo declarar variables

`let`: Declara una variable local **en el contexto del bloque en que se define**, y opcionalmente la inicia a un valor

`const`: Declara e inicializa un nombre de constante de **solo lectura** y **dentro del contexto del bloque en que se define**

`var`: Declara una variable, opcionalmente la inicia a un valor (💀)

No utilicen `var`: `const` y `let` son menos flexibles y por lo mismo tienen un comportamiento menos inesperado

```
// Variable (let)
let valorQueCambia = 1;
valorQueCambia = 4;

// Constante (const)
const valorFijo = 1;
valorFijo = 4; // ERROR

// Variable (var) [no recomendado]
var valorConVar = 1;
valorConVar = 4;
```

Fuente: *Gramática y Tipos - JavaScript | MDN*

Entendiendo el *scope* de una variable

Una variable declarada fuera de cualquier función se conoce como una variable global, pues está disponible para todo el resto del código en el documento actual

Una variable declarada dentro de una función es una variable local, pues está disponible sólo dentro del contexto de dicha función

```
// Variable global
let globalVariable = 'Soy una variable global';

function showVariables() {
  // Variable local
  let localVariable = 'Soy una variable local';

  console.log('Dentro de la función:');
  console.log('Variable global:', globalVariable);
  console.log('Variable local:', localVariable);
}

showVariables();
console.log('Fuera de la función:');
console.log('Variable global:', globalVariable);
console.log('Variable local:', localVariable);
ReferenceError: localVariable is not defined
```

Entendiendo el *scope* de una variable

De manera similar, en JavaScript entendemos el alcance (*scope*) de una variable en base a bloques de código

Un bloque de código se crea entre un par de corchetes. Funciones, condicionales (`if`, `else` ...) y *loops* (`while`, `for`, `do` ...) crean bloques

Las variables declaradas con `let` y `const` pueden pertenecer a un scope de bloque, mientras que que las variables creadas con `var` no están limitadas por un bloque (por eso es "mala práctica" usar var)

```
// Declaración no sale del scope
if (true) {
  let unValor = 42;
  const otroValor = 17;
}

console.log(unValor);
ReferenceError: unValor is not defined
console.log(otroValor);
ReferenceError: unValor is not defined
```

Fuente: *Scope en JavaScript - Variables locales y globales - Blog de Informática ✏ Luisi*

Entendiendo el *scope* de una variable

De manera similar, en JavaScript entendemos el alcance (*scope*) de una variable en base a bloques de código

Un bloque de código se crea entre un par de corchetes. Funciones, condicionales (`if`, `else` ...) y *loops* (`while`, `for`, `do` ...) crean bloques

Las variables declaradas con `let` y `const` pueden pertenecer a un scope de bloque, mientras que que las variables creadas con `var` no están limitadas por un bloque (por eso es "mala práctica" usar var)

```
// Inicialización sí pasa del scope
let variable = 1;
console.log(variable);
1
if (true) {
  variable = 2;
  console.log(variable);
  2
}
console.log(variable);
2
```

Fuente: *Scope en JavaScript - Variables locales y globales - Blog de Informática ✏ Luisi*

Cómo declarar variables

Recuerden:

✅ **`const`** es para valores que queremos que no puedan ser re-asignados

✅ **`let`** es para variables que queremos estén disponibles en el scope/bloque actual

❌ **`var`** es para variables que queremos estén disponibles en adelante (a la Python)

```
// var (al igual que let y const) declara una variable...
var variable;
// ... opcionalmente, también la inicializa
var otra_variable = 6;

// let declara una variable local
// (Solo existe en el scope actual)
if (true) {
  let variableLocal = 7;
  if (!false) {
    console.log("Aquí sí..." + variableLocal);
  }
}
console.log(variableLocal);
ReferenceError: variableLocal is not defined

// const declara una constante
// (Solo lectura y en scope actual)
const CONSTANTE = 8;
CONSTANTE = 9;
Cannot assign to "CONSTANTE" because it is a constant
```

Fuente: *Gramática y Tipos - JavaScript | MDN*

¿Por qué el concepto de *scope* es un tema?

- ES6 fue la versión que introdujo ***scopes* explícitos** con `let` y `const`
- Variables globales pueden causar problemas en el **navegador** (módulos pueden tener conflictos al declarar sus variables globales)
- Algunas guías de estilo sugieren declarar variables y funciones **antes de usarlas**
- ***Hoisting**** puede causar confusión en *scopes* locales

**hoisting: proceso por el cual el intérprete "mueve" la declaración de variables y funciones al comienzo del código.(lo que realmente hace es asignar una dirección de memoria a estos durante compilación)*

Hoisting

En la práctica, en el proceso de *hoisting*, las declaraciones de variables y funciones son asignadas en memoria durante la fase de compilación, lo que permite utilizarlos antes de su declaración en el código

Si bien los valores están disponibles, lo que se mueve es la declaración, no la inicialización

```
// Hoisting de variables
function demo() {
  console.log(value); // undefined (solo para var)
  var value = 5; // ¿let y const? ReferenceError: ...
}
demo();

// Hoisting de funciones (Solo declaradas)
console.log(f());

function f() {
  return 'Hola IIC2513';
}
```

Fuente: *Hoisting - Glosario de MDN Web Docs: Definiciones de términos relacionados con la Web | MDN*

Hoisting

En la práctica, en el proceso de *hoisting*, las declaraciones de variables y funciones son asignadas en memoria durante la fase de compilación, lo que permite utilizarlos antes de su declaración en el código

Si bien los valores están disponibles, lo que se mueve es la declaración, no la inicialización

```
const x = 1;
{
  console.log(x);   // ReferenceError
  const x = 2;
}
```

Fuente: *Hoisting - Glosario de MDN Web Docs: Definiciones de términos relacionados con la Web | MDN*

Agenda

JavaScript (JS)

- Variables
- ✨ **Tipos de datos**
- Control de flujo
- Funciones
- Elementos modernos

Tipos de datos primitivos

1. **Boolean**: `true` y `false`
2. **null**: declaración de un valor nulo (`null`, `NULL`, `Null` son distintos)
3. **Undefined**: valor no definido aún
4. **Number**: sea entero (`42`) o *float* (`3.14159`)
5. **BigInt**: entero con precisión arbitraria (como `9007199254740992n`)
6. **String**: secuencia de caracteres que representa un texto (como `"Hola"`)
7. **Symbol**: tipo especial que permite crear elementos únicos, que difieren de todos los demás símbolos (`Symbol('dog')`)

```
// Tipos primitivos
let bool = true;
let n = null;
let undf; // undefined
let num = 123;
let bignum = 1209385108239n;
let str = 'Hola';
let sym = Symbol('Symbol');
```

Fuente: *Tipos de datos y estructuras en JavaScript - JavaScript | MDN*

... más sobre números

JavaScript tiene soporte para números expresados en binario (`0b1101`), octal (`0o237`), decimal (`17`), hexa (`0xff3e`) o también como `float` (solo decimal)

Además de todo eso, existe `BigInt` para representar valores numéricos que son muy grandes para ser representados por el primitivo `number`

```
const PI = 3.14;
const octal = 0x237;
const hex = 0x6f;

console.log(octal, hex);
567 111

console.log(hex.toString(16));
'6f'

console.log(1 / 0 == Infinity);
true

console.log(Math.sqrt(-1));
NaN
console.log(Math.sqrt(-1) == NaN);
false
console.log(isNaN(Math.sqrt(-1)));
true

console.log(typeof NaN);
'number'

console.log(0.1 + 0.2);
0.30000000000000004
```

Extra

... más sobre *strings*

Los *strings* se representan como secuencias de caracteres delimitadas por comillas simples (`'`), dobles (`"`) o *backticks* (`` ` ``):

- `"hola"`
- `'hola'`
- `` `hola` ``

También soporta interpolación, pero solo con *backticks*:

`` `hola ${nombre}` ``

```
let mensaje = "Este mensaje es para";
console.log(mensaje.length);
20

mensaje = mensaje + " IIC2513";
console.log(mensaje);
'Este mensaje es para IIC2513'

let otroMensaje = 'con comillas simples';
console.log(otroMensaje);
'Este mensaje es para IIC2513'

let nuevoMensaje = `Con interpolación: ${mensaje}`;
console.log(nuevoMensaje);
'Con interpolación: Este mensaje es para IIC2513'
```

… más sobre valores especiales

Como es de esperar, los valores posibles para *boolean* en JavaScript solo son `true` y `false`. Sin embargo, además existen las siguientes nociones:

- ***"falsy"***: valores que se consideran `false` como `false`, `0`, `NaN`, `" "`, `undefined` y `null`
- ***"truthy"***: valores que se consideran verdaderos, es decir, cualquier valor no *"falsy"*

Otra pregunta común: ¿qué diferencia a `undefined` de `null`?

- `undefined`: no tiene valor definido (asignado)
- `null`: ningún valor
- Ambos son "falsy": `null == undefined` pero `null !== undefined` (valores equivalentes pero distinto tipos de dato)

Otros tipos de datos

1. **Object**: tipo estructural especial que no es de datos (no es “data type”) pero sí es para cualquier instancia de objeto que se utilice como estructura de datos:
   a. **`Object`**: colección de datos definidos y entidades más complejas
   b. **`Array`**: colección tipo lista
   c. **`Map`** (y **`WeakMap`**): colección tipo key-value
   d. **`Set`** (y **`WeakSet`**): colección de valores únicos
   e. **`Date`**: representa un momento en el tiempo
   f. y casi todo lo que se crea con new
2. **Function**: estructura que no contiene datos (aunque sí responde a `typeof`). Forma abreviada para funciones

```
// Object
let obj = { name: 'Antonio', age: 30 };

function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
const nissan = new Car('Nissan', '300ZX', 1992);
const mazda = new Car('Mazda', 'Miata', 1990);

// Dot notation
console.log(nissan.make);
'Nissan'

// Bracket notation
console.log(mazda['model']);
'Miata'
```

Fuente: *Tipos de datos y estructuras en JavaScript - JavaScript | MDN*

... más sobre objetos

```
// Un objeto es una colección ordenada de pares
// nombre-valor llamados propiedades
let unObjetoVacio = {};
let persona = { name: 'Alice', age: 23, country: 'Chile' };

// Se puede acceder a las propiedades tanto como
// si fueran atributos como si fuera un hash/diccionario
console.log(persona.name);
'Alice'
console.log(persona['country']);
'Chile'
// ¿Y si no existe un nombre de propiedad?
console.log(persona.estoNoExiste);
undefined

// Puedo agregar propiedades
persona.course = 'IIC2513';
// Puedo quitar propiedades
delete persona.country;

// Puedo listar los nombres disponibles de un objeto
console.log(Object.keys(persona));
[ 'name', 'age', 'course' ]
```

… más sobre objetos

En JavaScript, tanto variables como propiedades de objetos pueden almacenar cosas que no sean objetos (es decir, cualquier tipo de dato)

Lo importante es recordar que almacenan **referencias a objetos**, no los objetos en sí

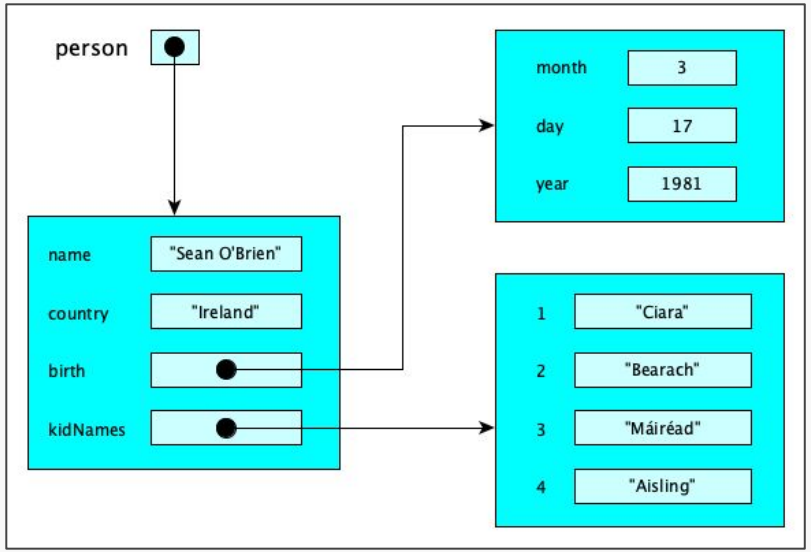
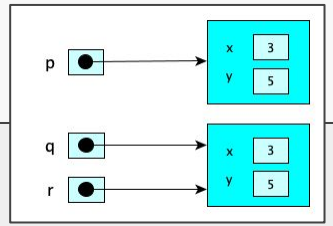


Figura: *JavaScript | Ray Toal*

… más sobre objetos

Para JavaScript, “igualdad e identidad son lo mismo”

¿Qué significa esto? Al comparar dos objetos, no se mirarán sus propiedades y valores, sino si estos refieren a la misma ubicación en memoria




```
const p = { x: 3, y: 5 };
const q = { x: 3, y: 5 };
const r = q; // Copiamos la REFERENCIA

console.log(p !== q); // ¿Son distintos objetos?
true

console.log(q === r); // ¿Son el mismo objeto?
true
```

Otros tipos de datos

1. **Object**: tipo estructural especial que no es de datos (no es “data type”) pero sí es para cualquier instancia de objeto que se utilice como estructura de datos:
   a. **`Object`**: colección de datos definidos y entidades más complejas
   b. **`Array`**: colección tipo lista
   c. **`Map`** (y **`WeakMap`**): colección tipo key-value
   d. **`Set`** (y **`WeakSet`**): colección de valores únicos
   e. **`Date`**: representa un momento en el tiempo
   f. y casi todo lo que se crea con new
2. **Function**: estructura que no contiene datos (aunque sí responde a `typeof`). Forma abreviada para funciones

```
// Array
// Para JavaScript son objetos
let lenguajesFav = ['JS', 'Python'];

// Puedo agregar elementos
lenguajesFav.push('C');

// Puedo extraer elementos
const elMasFav = lenguajesFav[0];
console.log(elMasFav);
'JS'

// Consultar su largo
const largo = lenguajesFav.length;
console.log(largo);
3

// Pueden contener distintos tipos y elementos vacíos
lenguajesFav[10] = 'PHP';
console.log(lenguajesFav);
[ 'JS', 'Python', 'C', <7 empty items>, 'PHP' ]
```

Fuente: *Tipos de datos y estructuras en JavaScript - JavaScript | MDN*

Otros tipos de datos

1. **Object**: tipo estructural especial que no es de datos (no es “data type”) pero sí es para cualquier instancia de objeto que se utilice como estructura de datos:
   a. **`Object`**: colección de datos definidos y entidades más complejas
   b. **`Array`**: colección tipo lista
   c. **`Map`** (y **`WeakMap`**): colección tipo key-value
   d. **`Set`** (y **`WeakSet`**): colección de valores únicos
   e. **`Date`**: representa un momento en el tiempo
   f. y casi todo lo que se crea con new
2. **Function**: estructura que no contiene datos (aunque sí responde a `typeof`). Forma abreviada para funciones

```
// Map
// 1. Create a Map
let map = new Map();

// 2. Add elements to the Map
map.set('name', 'SomeName');
map.set('age', 30);
map.set('city', 'SomeCity');

// 3. Access elements from the Map
console.log(map.get('name')); // Outputs: SomeName
console.log(map.get('age')); // Outputs: 30

// 4. Update elements in the Map
map.set('age', 31); // Updates 'age' to 31

// 5. Remove elements from the Map
map.delete('city'); // Removes the entry with key 'city'

// 6. Check if a key exists in the Map
console.log(map.has('name')); // Outputs: true
console.log(map.has('city')); // Outputs: false
```

Fuente: *Tipos de datos y estructuras en JavaScript - JavaScript | MDN*

Otros tipos de datos

1. **Object**: tipo estructural especial que no es de datos (no es “data type”) pero sí es para cualquier instancia de objeto que se utilice como estructura de datos:
   a. **`Object`**: colección de datos definidos y entidades más complejas
   b. **`Array`**: colección tipo lista
   c. **`Map`** (y **`WeakMap`**): colección tipo key-value
   d. **`Set`** (y **`WeakSet`**): colección de valores únicos
   e. **`Date`**: representa un momento en el tiempo
   f. y casi todo lo que se crea con new
2. **Function**: estructura que no contiene datos (aunque sí responde a `typeof`). Forma abreviada para funciones

```
// Set
// 1. Create a Set
let set = new Set();

// 2. Add elementos to the set
set.add('Apple');
set.add('Banana');
set.add('Cherry');

// 3. Access elements in the set
console.log(set.has('Banana'));
true
console.log(set.has('Grapes'));
false

// 4. Update elements in the set
set.add('Banana'); // Adding again does not change set

// 5. Remove elements from the set
set.delete('Cherry'); // Removes 'Cherry

// 6. Check the size of the set
console.log(set.size);
2

// 7. Clear all elements from the set
set.clear();
console.log(set.size);
0
```

Fuente: *Tipos de datos y estructuras en JavaScript - JavaScript | MDN*

Otros tipos de datos

1. **Object**: tipo estructural especial que no es de datos (no es “data type”) pero sí es para cualquier instancia de objeto que se utilice como estructura de datos:
   a. **`Object`**: colección de datos definidos y entidades más complejas
   b. **`Array`**: colección tipo lista
   c. **`Map`** (y **`WeakMap`**): colección tipo key-value
   d. **`Set`** (y **`WeakSet`**): colección de valores únicos
   e. **`Date`**: representa un momento en el tiempo
   f. y casi todo lo que se crea con new
2. **Function**: estructura que no contiene datos (aunque sí responde a `typeof`). Forma abreviada para funciones

```
// Current date and time
const now = new Date();
console.log('Current Date and Time:', now);

// Extract components
console.log('Year:', now.getFullYear());
console.log('Month (0-11):', now.getMonth());
console.log('Day of Month: ', now.getDate());
console.log('Day of Week: ', now.getDay());
console.log('Hours: ', now.getHours());
console.log('Minutes: ', now.getMinutes());
console.log('Seconds: ', now.getSeconds());

// Set specific date and time
const specificDate = new Date();
specificDate.setFullYear(2025);
specificDate.setMonth(11); // December
specificDate.setDate(25);
specificDate.setHours(10);
specificDate.setMinutes(30);
specificDate.setSeconds(0);
console.log('Specific Date and Time:', specificDate);

// Additional methods
console.log('Timestamp (ms since 1970):', now.getTime());
console.log('ISO String: ', now.toISOString());
console.log('Locale String: ', now.toLocaleString());
console.log('Date String: ', now.toDateString());
console.log('Time String: ', now.toTimeString());
console.log('UTC String:', now.toUTCString());
```

Fuente: *Tipos de datos y estructuras en JavaScript - JavaScript | MDN*

Otros tipos de datos

1. **Object**: tipo estructural especial que no es de datos (no es “data type”) pero sí es para cualquier instancia de objeto que se utilice como estructura de datos:
   a. **`Object`**: colección de datos definidos y entidades más complejas
   b. **`Array`**: colección tipo lista
   c. **`Map`** (y **`WeakMap`**): colección tipo key-value
   d. **`Set`** (y **`WeakSet`**): colección de valores únicos
   e. **`Date`**: representa un momento en el tiempo
   f. y casi todo lo que se crea con new
2. **Function**: estructura que no contiene datos (aunque sí responde a `typeof`). Forma abreviada para funciones

```
// Function
// 1. Define a function
function greet(name) {
  return `Hello, ${name}!`;
}

// 2. Function expression
const add = function (a, b) {
  return a + b;
};

// 3. Arrow function (ES6)
const multiply = (x, y) => x * y;
```

Fuente: *Tipos de datos y estructuras en JavaScript - JavaScript | MDN*

Tipado dinámico

Que JavaScript sea un lenguaje de tipado dinámico significa que el tipo de una variable se determina en tiempo de ejecución (*runtime*) y puede cambiar a medida que la variable es reasignada

JavaScript no requiere declarar explícitamente el tipo de una variable, sino que se infiere del valor asignado a la variable

```
// Ejemplo de tipado dinámico en JavaScript

let variable = 10;
console.log(typeof variable);
'number'

variable = 'Hello';
console.log(typeof variable);
'string'

variable = [1, 2, 3];
console.log(typeof variable);
'object'

variable = true;
console.log(typeof variable);
'boolean'
```

TypeScript

TypeScript (TS) es superset de JavaScript

Clase 16#*Tecnologías adicionales*

Implementa una formalización de los tipos de JavaScript, permitiendo definir variables y funciones tipadas




```
  lastName: 'Ossa',
  role: 'Teacher',
};

console.log(user.name);
Property 'name' does not exist on type '{ firstName: string;
lastName: string; role: string; }'
```

Sobre operadores

Además de los operadores que podrían conocer de otros lenguajes, JavaScript incluye algunos que podrían ser nuevos:

- Operador condicional (ternario)
  usado como if/else *one-liner*
- Operador lógico OR
  usado de *fallback* para “falsey”
- *Nullish coalescing*
  usado de *fallback* para nulos
- Operadores unarios + y –
  usado para conversión a números

```
// Operador ternario
let resultado = condicion ? siTriuthy : siFalsey;

// Fallback (respaldo) para "falsey"
let value = podriaSerFalsey || fallback;

// Fallback para "nullish"
let another = podriaSerNullOUndefined ??
fallback;

// Operador unario
const podriaSerNumero = '42';
console.log(typeof podriaSerNumero);
'string'
console.log(typeof +podriaSerNumero);
'number'
```

Resumen de operaciones básicas

Para determinar el tipo de una expresión: `typeof exp`

Para números:

```
+x    -x    x + y    x - y    x * y    x / y    x % y    x ** y
~x      x & y    x | y    x ^ y
x > y    x < y    x >= y    x <= y
x.toString    Math.floor(x)    Math.ceil(x)    Math.round(x)
Math.pow(x, y)    Math.min(...)    Math.max(...)
```

Para *strings*:

```
s[i]    s.slice(i, j)    s.length    s.trim()    s + t
s.concat(s1, s2, …)    s.repeat(count)    s.split(strOrRegex)
s.startsWith(t)    s.indexOf(t)    s.toUpperCase()
```

Agenda

JavaScript (JS)

- Variables
- Tipos de datos
- ✨ **Control de flujo**
- Funciones
- Elementos modernos

Primero: sobre bloques de código

En cuanto a uso de llaves, realmente no importa la indentación

Aplica para cualquier bloque (`if`, `while`, `for`), pero consideren la legibilidad del código

```
// Si es solo una línea, podemos omitir las llaves
if (condicion) console.log('Esto se cumple');

// Si hay más de una línea, debemos usar llaves
// No importa la indentación mientras use llaves
if (variable > 5) {
  variable = variable / 2;
}

for (let i = 0; i < 10; i++)
{variable += i;}

function miFuncion(arg1, arg2) {
return arg1 + 2 * arg2;
}
```

Comparación de valores

Para comparar valores:

`x === y`	`true` si `x` e `y` son IGUALES (mismo tipo y valor, `1 !== true`)
`x !== y`	negación de lo anterior
`x == y`	`true` si `x` e `y` son SIMILARES (valor equivalente, `1 == true`)
`x != y`	negación de lo anterior
`x && y`	si `x` es “falso” entonces `x`, sino `y`
`x || y`	si `x` es “verdadero” entonces `x`, sino `y`
`!x`	si `x` es “verdadero” entonces `false`, sino `true`
`x ?? y`	si `x` es “nulo” entonces `y`, sino `x`

Ejemplo de control de flujo

```
if (variable > 5 && constante == 1) {
    // Ambas condiciones se cumplen (AND)
    variable = variable / 2;
} else if (variable < 5 || constante == 1) {
    // Alguna de las dos cumple (OR)
    variable -= 2;
} else {
    variable = 0;
}

if !(variable > 5) {
    // Si NO se cumple la condición (NOT)
    variable = -1;
}
```

Loops `for`, `while` y… `do while`?

```
for (let i = 5; i < 10; i++) {
  console.log(i);
}

let j = 5;
while (j < 10) {
  console.log(j);
  j++;
}

let k;
do {
  k = prompt('hola');
} while (k !== 'hola');
console.log('Al fin dijiste hola');
```

`for (begin; condition; step)`
Notar que usar `let` en *begin* hace que la variable solo exista en el bloque del loop

`while (condition)`
Revisa la condición dada al iniciar cada ciclo

`do while (condition)`
Revisa la condición dada al terminar cada ciclo

Las palabras `break` y `continue` terminan el bloque o vuelven al inicio, respectivamente

Ejemplos de uso de `for`

```
// Iteramos sobre un rango (contando)
let suma = 0;
for (let contador = 0; contador < 10; contador += 1) {
  suma += 2;
}

// Iteramos sobre un arreglo
const arreglo = [1, 2, 3];
for (const element of arreglo) {
  suma += element;
}

// Iteramos sobre un objeto
const objeto = { a: 1, b: 2, c: 3 };
for (const propiedad in objeto) {
  suma += objeto[propiedad];
}
```

Switch

Al igual que algunos lenguajes, JavaScript también tiene una declaración *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.

```
let grade = 'A';

switch (grade) {
  case 'A':
  case 'B+':
    console.log('Muy bien!');
    break;
  case 'B':
  case 'C':
    console.log('Bien');
  case 'D':
  case 'E':
    console.log('Hay que estudiar más');
    break;
  default:
    console.log('¿Me repites la nota?');
}
```

Fuente: *The "switch" statement | JavaScript.info*

Agenda

JavaScript (JS)

- Variables
- Tipos de datos
- Control de flujo
- ✨ **Funciones**
- Elementos modernos

Creando funciones

JavaScript cuenta con 4 tipos de funciones (regulares, generadoras, asíncronas y generadoras asíncronas)

Por ahora, nos centraremos en funciones regulares: pueden retornar cualquier cosa y corren completamente una vez invocadas

Existen múltiples mecanismos y sintaxis para definir funciones según su tipo, pero nos centraremos en 3 formas:

- Declaración de función
- Expresión de función
- Función flecha (*arrow function*)

```
// Declaration
function multiply(x, y) {
  return x * y;
}

// Expression (función anónima pero en variable)
const multiply = function (x, y) {
  return x * y;
};
// Expression (no anónima)
const multiply = function someName(x, y) {
  return x * y;
};

// Arrow function
const multiply = (x, y) => x * y;

// Method
const obj = {
  multiply(x, y) {
    return x * y;
  },
};

// Constructor [Usa eval]
const multiply = new Function('x', 'y', 'return x * y');
```

Fuente: *Funciones - JavaScript | MDN*

Declaración de función

El comportamiento de una función declarada es similar al de una función regular en otros lenguajes de programación: tiene variables locales (visibles dentro de la función), puede acceder y modificar variables externas (globales), podemos definir valores por defecto y puede devolver un valor

```
function showMessage(from, text = '<vacio>') {
  console.log(`${from}: ${text}`);
}

showMessage('Antonio', '¡Hola!');
Antonio: ¡Hola!
showMessage('Antonio');
Antonio: <vacio>
```

Una función declarada puede ser llamada antes de ser definida (*hoisting*), por lo que en general se usan para definir funciones en el *scope* global

Fuente: *Funciones | JavaScript.info*

… un poco más de funciones regulares

```
// Hoisting también aplica a funciones
console.log(factorial(10)); // Funciona perfectamente

// Para JavaScript, las funciones también son objetos
function factorial(n) {
  // console.log(arguments);  // Argumentos recibidos
  if (n <= 1) {
    return 1;
  }
  return n * factorial(n - 1);
}

console.log(typeof factorial); // function
```

Expresión de función

Tal como en otros lenguajes, las funciones en JavaScript son valores. Esto nos permite almacenarlas en variables y tratarlas como objetos

```
let factorial = function (n) {
  if (n <= 1) {
    return 1;
  }
  return n * factorial(n - 1);
};
```

¿Diferencias con una función declarada? Una función como expresión se crea cuando la ejecución alcanza dicha expresión (no hay *hoisting*)

En general, se usan funciones por expresión cuando buscamos limitar dónde está disponible una función, y mantener el scope global y la sintaxis limpios

Fuente: *Expresiones de función | JavaScript.info*

Función flecha (*arrow function*)

Una tercera manera de definir funciones es utilizando *arrow functions* o funciones flecha. Estas funciones, además de ser "anónimas" (no requieren un nombre), no tienen *this*, por lo que puede ser muy útil al anidar funciones o definir *callbacks*

```
const materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium'];


console.log(materials.map(material => material.length));
Array[(8, 6, 7, 9)]
```

En general, utilizamos funciones flecha cuando queremos usar funciones anónimas (sin nombre) o necesitamos callbacks en contextos dinámicos

Fuente: *Arrow functions, the basics | JavaScript.info*

Agenda

- JavaScript (JS)
  - Variables
  - Tipos de datos
  - Control de flujo
  - Funciones
  - ✨ **Elementos modernos**

Mejoras en funciones (*default values*)

En JavaScript, siempre se ha podido pasar un número variable de parámetros en la declaración de una función, por medio de `arguments.length`:

```
function foo() {
  for (var i = 0; i < arguments.length; i++) {
    console.log(arguments[i])
  }
}
```

En ECMAScript 6 los podemos inicializar, evitando tener que hacer chequeos adicionales:

```
function foo(x, y) {
  x = x || 0;
  y = y || 0;
}

function foo(x=0, y=0) {
}
```

```
function selectEnt(options) {
  options = options || {};
  var start = options.start || 0;
  var end = options.end || -1;
  var step = options.step || 1;
}

function selectEnt({ start=0, end=-1, step=1 } = {}) {
}
```

Fuente: *Funciones | JavaScript.info*

Clases

Muchos desarrolladores creen que no son necesarias pero muchas librerías las implementan

ES6 incorpora clases aunque son algo distintas a lo usual

```
const rectangle = new Rectangle(10, 10);
console.log(Rectangle.describe());
```

```
const square = new Square(15);
console.log(Square.describe());
```

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  area() {
    return this.width * this.height;
  }
  static describe() {
    return "A Rectangle";
  }
}

class Square extends Rectangle {
  constructor(size) {
    super(size, size);
  }
  static describe() {
    return "A Square";
  }
}
```

Arrays

`Array.from` permite convertir listas (por ejemplo resultado de operaciones del DOM a arrays)

`Array.of` permite generar un *array* a partir de una lista de parámetros

```
console.log(Array.from('foo'));
// Array ["f", "o", "o"]

console.log(Array.from([1, 2, 3], x => x + x));
// Array [2, 4, 6]



console.log(Array.of('foo', 2, 'bar', true));
// Array ["foo", 2, "bar", true]

console.log(Array.of());
// Array []
```

Fuente: *Array.from() - JavaScript | MDN* & Array.of() - JavaScript | MDN

Maps y Sets

Se introdujeron nuevos tipos para almacenar colecciones de datos, tales como `Map` (almacenamiento tipo key-value) y `Set` (tipo conjunto de datos, sin duplicados)

```
let map = new Map();
const KEY = {};

map.set(KEY, 123);
map.get(KEY);  // 123
map.has(KEY);
map.delete(KEY);
```

```
let s = new Set();

s.add("hello");
s.add("goodbye").add("hello");
console.log(s.size === 2);
console.log(s.has("hello"));
```

Modularización

Se permite importar y exportar funciones, objetos o datos primitivos de un módulo a otros:

```
// lib.js
export const sqrt = Math.sqrt;

export function square(x) {
  return x * x;
}

export function diag(x, y) {
  return sqrt(square(x) + square(y));
}
```

```
// main1.js
import { square, diag } from 'lib';

console.log(square(11));  // 121
console.log(diag(3, 4));  // 5
```

```
// main2.js
import * as lib from 'lib';

console.log(lib.square(11));  // 121
console.log(lib.diag(3, 4));  // 5
```

Fuente: *Array.from() - JavaScript | MDN* & Array.of() - JavaScript | MDN

Bibliografía y recursos recomendados

Para leer y aprender sobre JavaScript:

- Guía de JavaScript | MDN: Cómo usar JavaScript y una perspectiva general del lenguaje
- JavaScript | MDN: Aprende a programar en JavaScript con guías y tutoriales
- JavaScript | Ray Toal: Vistazo general a JavaScript (en inglés)

Para practicar programando en JavaScript:

- JavaScript Tutorial | W3Schools: Tutorial de JavaScript (en inglés)
- Learn JavaScript: Tutorial interactivo de JavaScript moderno (en inglés)

Editores *online*:

- JSFiddle
- CodePen

Más:

- airbnb/javascript @ GitHub: Guía de estilo de JS de Airbnb

Lecturas recomendadas

Sobre versiones recientes de JavaScript:

- ECMAScript Versions in JavaScript: ECMAScript 1 a ECMAScript 2023 (ES14)
- The Complete ECMAScript 2015-2019 Guide: ECMAScript 2015-2019 Guide

Deno v. Oracle (arco en desarrollo sobre liberar la marca JavaScript):

- javascript.tm: “Oracle, it’s time to free JavaScript”. Carta abierta a Oracle.
- Deno v. Oracle: Canceling the JavaScript Trademark: Petición formal para cancelar la marca ante la USPTO
- Oracle justified its JavaScript trademark with Node.js—now it wants that ignored: Oracle pide a la USPTO descartar la petición

Agenda

- JavaScript (JS)
  - Variables
  - Tipos de datos
  - Control de flujo
  - Funciones
  - Elementos modernos
- ✨ **¿Qué se viene pronto?**

Próxima clase:

Seguiremos con JavaScript:

- Manejo del DOM
- Eventos del navegador
- *Callbacks*
- Programación asíncrona (*async*)
- Promesas en JavaScript

Próximas fechas importantes: clases y ayudantías

	Clases y ayudantías						
	Lunes	**Martes**	**Miércoles**	**Jueves**	**Viernes**	**Sábado**	**Domingo**
Semana 2 (10/3 - 16/3)		Desde el navegador a mi sitio web		HTML y CSS			
Semana 3 (17/3 - 23/3)		HOY		JavaScript pt. 2 (DOM y async)	Ayudantía: JavaScript		
Semana 4 (24/3 - 30/3)		React		Exploración y consumo de APIs	Ayudantía: React		
Semana 5 (31/3 - 06/4)		Node.js		Desarrollando servidores	Ayudantía: Consumo de APIs		
Semana 6 (07/4 - 13/4)		Construcción de APIs		QA, CI & CD	Ayudantía: Construcción de APIs		

Próximas fechas importantes: evaluaciones

	Evaluaciones						
	Lunes	**Martes**	**Miércoles**	**Jueves**	**Viernes**	**Sábado**	**Domingo**
Semana 2 (10/3 - 16/3)							
Semana 3 (17/3 - 23/3)		HOY					
Semana 4 (24/3 - 30/3)	Control 1				Inicio T1 React		
Semana 5 (31/3 - 06/4)					Fin T1 (React)		
Semana 6 (07/4 - 13/4)	Control 2				Inicio T2 API		

Clase 3

JavaScript parte 1

IIC2513 - Tecnologías y Aplicaciones Web

Antonio Ossa Guerra
aaossa@ing.puc.cl