Desarrollo en el cliente

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez
ruben.rodriguez@urjc.es
URJC

Natalia Madrueño
natalia.madrueno@urjc.es
URJC

2025-09-09

Universidad Rey Juan Carlos

DSLAB
DATA SCIENCE LABORATORY

Tabla de contenidos

Introducción a JavaScript

Introducción

JavaScript es el lenguaje fundamental para desarrollo web interactivo y videojuegos web.

Características principales:

- Scripting (no necesita compilador)
- Tipado dinámico
- Funcional
- Orientado a objetos (prototipos)

Aplicaciones:

- Interactividad en páginas web
- Modificación del DOM
- Peticiones AJAX
- Videojuegos web

Versiones de ECMAScript

- Primera versión en 10 días (1995).
- **ES5 (2011)**: Base sólida que estableció JavaScript moderno en todos los navegadores
- **ES6/ES2015**: Revolución del lenguaje - introdujo sintaxis moderna que cambió cómo programamos
- **Actualizaciones anuales**: Cada año se añaden mejoras sin romper código existente (compatibilidad hacia atrás)
- **ES2015 marcó un antes y después**: La mayoría del código moderno usa características de ES6+

Versión	Año	Características
ES5	2011	Popularizó JavaScript
ES2015 (ES6)	2015	Clases, módulos, arrow functions, promises
ES2016-2024	2016-2024	Actualizaciones anuales con compatibilidad

JavaScript vs Java

> ⚠ **¡Importante!**
>
> Aunque la sintaxis recuerda a Java, son lenguajes **completamente diferentes**.

- **El nombre “JavaScript” fue puro marketing**: Se eligió para aprovechar la popularidad de Java
- **Inicialmente se llamó LiveScript**: Cambió de nombre antes de su lanzamiento oficial
- **Java estaba en auge** cuando se publicó JavaScript (1995)
- **Diferentes propósitos**: Java para aplicaciones robustas, JavaScript para web interactivo

DOM y BOM

DOM (Document Object Model)

- **Representa HTML como árbol de objetos**: Cada etiqueta HTML es un objeto manipulable
- **Modificación dinámica**: Cambia contenido sin recargar la página completa
- **Gestión de eventos**: Captura clicks, teclas, movimientos del ratón, etc.
- **Ejemplo**: `document.getElementById('boton')` accede a un elemento

BOM (Browser Object Model)

- **Controla el navegador entero**: No solo el documento, sino ventanas, historial, URL
- **window.location**: Navega a otras URLs o recarga la página
- **window.history**: Retrocede/avanza en el historial del navegador
- **window.localStorage**: Guarda datos en el navegador
- **El BOM contiene al DOM**: `window.document` es el DOM

Librerías JavaScript

Para videojuegos:

- Phaser: Framework completo para juegos 2D - maneja física, colisiones, animaciones

Herramientas de desarrollo:

- Webpack: Empaquetador de módulos - combina y optimiza tu código JavaScript y assets
- Babel: Transpilador - convierte código JavaScript moderno a versiones compatibles con navegadores antiguos

Backend y comunicación:

- Express.js: Framework minimalista para crear servidores web y APIs en Node.js
- WS (ws.js): Biblioteca para WebSockets - permite comunicación en tiempo real bidireccional

Configuración del Entorno

Node.js

- **Ejecuta JavaScript fuera del navegador**: En tu ordenador, servidores, etc.
- **npm (Node Package Manager)**: Gestor de paquetes para instalar librerías
- **Entorno de desarrollo moderno**: Necesario para usar herramientas como Webpack

Node.js permite ejecutar JavaScript fuera del navegador.

Instalación:

1. Descargar desde nodejs.org
2. Instalar versión LTS (Long Term Support - más estable)

Verificar instalación:

```
# Muestra la versión de Node.js instalada
node --version

# Muestra la versión de npm (viene incluido con Node)
npm --version
```

Inicializar un Proyecto

- **npm init -y**: Crea `package.json` con configuración por defecto (sin preguntas)
- **package.json**: Archivo que describe tu proyecto y sus dependencias
- **src/**: Carpeta con tu código fuente JavaScript
- **dist/**: Carpeta con archivos finales optimizados listos para producción
- **public/**: Recursos estáticos como HTML, imágenes, CSS que no se procesan

```
1 # Inicializar proyecto npm (crea package.json) dentro de una carpeta
2 npm init -y
```

Estructura recomendada:

```
mi-juego-web/
├── package.json       # Configuración del proyecto
├── webpack.config.js  # Configuración de Webpack
├── src/               # Tu código fuente
├── dist/              # Archivos compilados (generado)
└── public/            # HTML y recursos estáticos
    └── index.html
```

Instalación de Dependencias

- **Dependencias de producción**: Librerías que necesita tu juego para funcionar
- **Dependencias de desarrollo**: Herramientas solo para programar (no van al juego final)
- **npm install**: Descarga e instala paquetes desde el registro de npm
- **–save-dev**: Marca como dependencia de desarrollo

Dependencias de producción:

```
# Phaser: motor de juegos 2D
# Lodash: utilidades para datos
# Axios: peticiones HTTP
npm install phaser lodash axios
```

Dependencias de desarrollo:

```
# Webpack: empaqueta y optimiza código
npm install --save-dev webpack webpack-cli webpack-dev-server

# Plugins de Webpack para HTML y CSS
npm install --save-dev html-webpack-plugin css-loader style-loader

# Herramientas de calidad de código
npm install --save-dev eslint prettier
```

Configuración de Webpack

- **Webpack**: Empaquetador que une todos tus archivos JS en uno solo optimizado
- **entry**: Punto de entrada - primer archivo que se ejecuta
- **output**: Dónde guardar el resultado final
- **plugins**: Extensiones que añaden funcionalidades (ej: generar HTML)
- **devServer**: Servidor de desarrollo con recarga automática

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  // **entry**: Punto de entrada de la aplicación
  entry: './src/index.js',

  // **output**: Dónde se guarda el bundle generado
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
    clean: true // Limpia la carpeta 'dist' antes de cada build
  },

  // **plugins**: Añade funcionalidades extra (como generar el HTML automáticamente)
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html', // Usa esta plantilla HTML
```

Scripts de Desarrollo

- **Scripts npm**: Comandos personalizados para automatizar tareas comunes
- **npm run dev**: Inicia servidor de desarrollo con recarga automática
- **npm run build**: Crea versión optimizada para producción
- **npm run lint**: Revisa errores de código con ESLint
- **npm run format**: Formatea código con Prettier

package.json:

```
{
  "scripts": {
    // Construye el paquete para producción usando Webpack
    "build": "webpack --mode production",

    // Inicia el servidor de desarrollo con Webpack Dev Server y habilita hot-reload
    "dev": "webpack serve --mode development",

    // Inicia el servidor en modo producción (requiere que server.js esté correctamente configurado)
    "start": "node server.js",

    // Observa los cambios en los archivos y recompila automáticamente (sin servidor)
    "watch": "webpack --watch",
```

El Lenguaje JavaScript

Características del Lenguaje

- **Imperativo y Estructurado**: Escribe instrucciones paso a paso, como Java o C
- **Lenguaje de Script**: El navegador ejecuta el código directamente, sin compilar primero
- **Tipado Dinámico**: Las variables pueden cambiar de tipo (`let x = 5; x = "texto";`)
- **Orientado a Objetos**: Usa prototipos en lugar de clases tradicionales (hasta ES6)
- **Funcional**: Las funciones son valores - pueden pasarse como argumentos

Modo Estricto

- **Detecta errores que normalmente JavaScript ignora**: Convierte errores silenciosos en excepciones
- **Prohíbe sintaxis peligrosa**: Variables sin declarar, duplicar parámetros, etc.
- **Mejor rendimiento**: Permite optimizaciones del motor JavaScript
- **En módulos ES2015+ ya está activo**: No necesitas añadirlo manualmente
- También se puede utiliziar `node --use_strict` en node.

```
1 // Activar modo estricto (poner al inicio del archivo o función)
2 "use strict";
3
4 // Ahora esto genera error (sin strict mode, crea variable global)
5 playerName = "Juan"; // Error: playerName is not defined
```

Integración con HTML

- **Scripts en** `<head>` : Se ejecutan antes de cargar el contenido (puede bloquear renderizado)
- **Scripts al final de** `<body>` : Mejor práctica - el HTML ya está cargado
- **async**: Descarga el script en paralelo, ejecuta en cuanto esté listo (orden no garantizado)
- **defer**: Descarga en paralelo, pero ejecuta en orden después del HTML

```html
<html>
<head>
    <!-- Script en head - se ejecuta inmediatamente -->
    <script src="js/config.js"></script>
</head>
<body>
    <!-- Contenido HTML -->
    <!-- Scripts al final - mejor rendimiento (HTML ya cargado) -->
    <script src="js/game.js"></script>

    <!-- async: descarga paralela, ejecuta inmediatamente -->
    <script src="js/game.js" async></script>

    <!-- defer: descarga paralela, ejecuta después del HTML -->
    <script src="js/game.js" defer></script>
</body>
</html>
```

Mostrar Información

- **document.write()**: Escribe directamente en el HTML (evitar - obsoleto)
- **console.log()**: Muestra información en consola del navegador (F12)
- **console.error()**: Muestra errores en rojo - útil para debugging
- **console.warn()**: Muestra advertencias en amarillo

```
1  // Escribir en el documento HTML (no recomendado)
2  document.write('Texto');
3
4  // Consola del navegador (debugging) - Pulsa F12 para verla
5  console.log('Información de debug');
6  console.error('Error crítico');
7  console.warn('Advertencia');
8
9  // También puedes mostrar objetos
10 console.log('Jugador:', { nombre: 'Juan', vida: 100 });
```

Comentarios

- Igual que en Java.

```
// Comentario de una línea – se ignora al ejecutar

/*
 * Comentario
 * multilínea
 * útil para documentar funciones
 */

// Los comentarios explican el "por qué", no el "qué"
// Malo: let x = 5; // Asigna 5 a x
// Bueno: let maxRetries = 5; // Límite de reintentos antes de fallar
```

Variables

- **const**: Usa por defecto - evita modificaciones accidentales de valores que no deben cambiar
- **let**: Solo cuando necesites reasignar el valor (contadores, acumuladores)
- **var**: Obsoleto - tiene problemas de scope que causan bugs difíciles de encontrar
- **Block scope**: Variables solo existen dentro del bloque `{}` donde se declaran

```
// const - valor inmutable (no se puede reasignar), block scope
const MAX_LIVES = 3;
const GAME_CONFIG = {
    width: 800,
    height: 600
};

// let - valor mutable (se puede reasignar), block scope
let currentLives = MAX_LIVES;
let playerPosition = { x: 0, y: 0 };

// var - evitar en código nuevo (function scope - problemas)
var oldStyle = "no recomendado";

// Ejemplos de uso
currentLives = 2; // OK - let permite reasignar
MAX_LIVES = 5; // ERROR - const no permite reasignar
```

Ámbito de Variables

- **Block scope** (`let` , `const`): Variable solo existe dentro de las llaves `{}` donde se declaró
- **Function scope** (`var`): Variable existe en toda la función, ignorando bloques `{}`
- **Variables sin declarar**: Se vuelven globales (accesibles desde cualquier lugar) - PELIGROSO
- **Modo estricto**: Convierte variables sin declarar en error

```
function ejemploScope() {
    if (true) {
        let bloqueVariable = "solo aquí";
        var funcionVariable = "toda la función";
    }

    // console.log(bloqueVariable); // ERROR - no existe fuera del if
    console.log(funcionVariable);   // OK - var tiene function scope
}

// Sin modo estricto
function peligro() {
    sinDeclarar = "ups"; // Crea variable global - BAD!
}
```

Tipos de Datos Primitivos

- **Number**: Un solo tipo para enteros y decimales (64 bits de precisión)
- **String**: Cadenas de texto - pueden usar `""` , `''` o `` `` `` (template literals)
- **Boolean**: Solo dos valores: `true` o `false`
- **null**: Ausencia intencional de valor - “esto está vacío a propósito”
- **undefined**: Variable declarada pero sin valor asignado - “aún no tiene valor”

```
// Number – enteros y decimales (punto flotante de 64 bits)
let score = 1000;
let health = 75.5;
let infinity = Infinity; // Valor especial

// String – cadenas de caracteres
let playerName = "Jugador1";

// Boolean – verdadero o falso
let isGameRunning = true;
let isPaused = false;

// Tipos especiales
let powerUp = null;         // Ausencia intencional – "no hay powerup"
let specialAbility;         // undefined – no inicializada aún
```

Template Literals (ES2015+)

- **Backticks (`)**: En lugar de comillas normales
- **Interpolación** `${}` : Inserta variables o expresiones dentro del string
- **Multilínea**: Puedes escribir strings en varias líneas sin concatenar
- **Expresiones**: Dentro de `${}` puedes poner cualquier código JavaScript

```
let level = 5;
let experience = 1250;

// Interpolación de strings - inserta valores de variables
let status = `Nivel ${level} - EXP: ${experience}`;

// También puedes usar expresiones dentro de ${}
let progress = `Progreso: ${(experience / 2000) * 100}%`;

// Strings multilínea - se respetan los saltos de línea
let gameInfo = `
Jugador: ${playerName}
Nivel: ${level}
Puntuación: ${score}
Estado: ${score > 1000 ? 'Pro' : 'Novato'}
`;
```

Operadores Aritméticos

- **Operadores matemáticos básicos**: `+` , `-` , `*` , `/` , `%` (módulo/resto)
- **Comparación**: `>` , `<` , `>=` , `<=` comparan valores
- **Lógicos**: `&&` (y), `||` (o), `!` (no) para combinar condiciones
- **Precedencia**: Multiplicación y división antes que suma y resta

```
1  // Operadores aritméticos - similares a Java
2  let damage = baseDamage + bonus;
3  let remaining = total - used;
4  let area = width * height;
5  let average = sum / count;
6  let remainder = value % modulo; // Resto de división
7
8  // Lógicos - combinan condiciones -> Devuelven true or false
9  let canAct = isAlive && !isStunned; // Y lógico, NO lógico
10 let shouldRespawn = isDead || health <= 0; // O lógico
```

Operadores de Comparación

- **=== (estricto)**: Compara valor Y tipo - **siempre recomendado**
- **== (débil)**: Convierte tipos antes de comparar - **evitar** (causa bugs)
- **!== (desigualdad estricta)**: Diferente valor O tipo
- **Ejemplo**: `5 == "5"` es `true`, pero `5 === "5"` es `false`

```
// Igualdad estricta (recomendado) - compara valor Y tipo
if (playerID === targetID) {
    // Solo true si ambos tienen el mismo valor Y tipo
}

// Desigualdad estricta - diferente valor O tipo
if (level !== previousLevel) {
    // true si son diferentes
}

// Igualdad débil (EVITAR) - hace conversión de tipos
if (score == "100") {
    // true - convierte "100" a número 100. Puede causar bugs difíciles de encontrar
}

// Comparación de tipos diferentes
console.log(5 === "5");  // false - número vs string
console.log(5 == "5");   // true  - convierte a mismo tipo
```

Operadores Modernos (ES2020+)

- **Nullish coalescing (`??`)**: Valor por defecto solo si `null` o `undefined`
- **Optional chaining (`?.`)**: Accede a propiedades sin error si no existen
- **Logical assignment**: Asigna solo si cumple condición
- **Diferencia con `||`** : `||` también considera `0` , `""` , `false` como “falsy”

```
// Nullish coalescing - valor por defecto solo para null/undefined
let playerName = savedName ?? "Jugador Anónimo";
// Si savedName es 0 o "" NO usa el default (solo null/undefined)

// Optional chaining - evita errores si propiedades no existen
let weapon = player.inventory?.equipment?.weapon;
// Si inventory o equipment es null/undefined, retorna undefined sin error

// Logical assignment
playerName ||= "Jugador por defecto";  // Asigna solo si falsy
playerName ??= "Valor por defecto";   // Asigna solo si null/undefined

// Diferencia entre || y ??
let count = 0;
let result1 = count || 10;  // 10 (0 es falsy)
let result2 = count ?? 10;  // 0  (0 no es null/undefined)
```

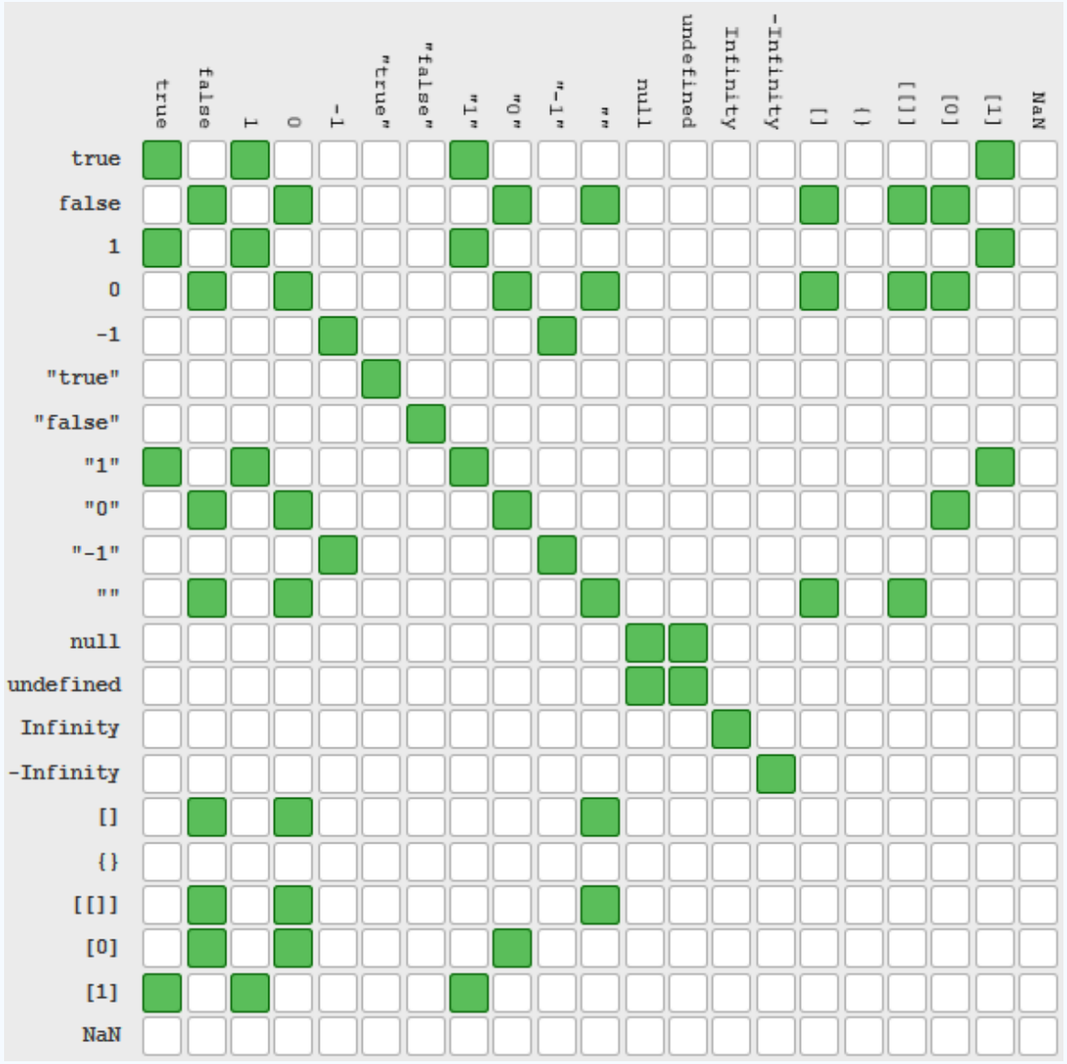
Valores Falsy

- **JavaScript evalúa algunos valores como `false`** en condiciones
- **6 valores falsy**: `false`, `0`, `""`, `null`, `undefined`, `NaN`
- **Todos los demás son truthy**: Incluso `"0"`, `[]`, `{}`
- **Útil para**: Validaciones y valores por defecto

JavaScript considera **falso** (falsy):

- `false` - booleano falso
- `null` - sin valor
- `undefined` - no definido (verificación de existencia)
- `""` - string vacío
- `0` - cero numérico
- `NaN` - “Not a Number”

Utiliza siempre ===



Arrays

Creación de Arrays

- **Arrays dinámicos**: Pueden crecer o reducirse automáticamente
- **Tipos mezclados**: Puedes guardar diferentes tipos en el mismo array
- **Índice desde 0**: El primer elemento está en posición 0
- **length**: Propiedad que indica cuántos elementos tiene

```
1  // Creación de arrays - literal (forma más común)
2  let empty = [];
3  let numbers = [1, 2, 3, 4, 5];
4  let mixed = ["texto", 42, true, null]; // Tipos mezclados OK
5
6  // Constructor Array (menos común)
7  let inventory = new Array(10); // Crea array con 10 espacios vacíos
8
9  // Acceso y modificación por índice
10 console.log(numbers[0]);        // 1 - primer elemento
11 numbers[2] = 999;              // Modifica tercer elemento
12
13 // Arrays dinámicos - se expanden automáticamente
14 numbers[10] = 100;             // Salta del índice 4 al 10
15 console.log(numbers.length);    // 11 - ahora tiene 11 elementos
16 // Los índices 5-9 quedan vacíos (undefined)
```

Métodos de Array Modernos

- **filter()**: Crea nuevo array con elementos que cumplen condición
- **map()**: Crea nuevo array transformando cada elemento
- **No modifican el original**: Retornan nuevo array (inmutabilidad)
- **Callback**: Función que se ejecuta por cada elemento

```
// Array de ejemplo
let enemies = [
    { id: 1, health: 100, type: "orc" },
    { id: 2, health: 50, type: "goblin" },
    { id: 3, health: 0, type: "orc" }
];

// filter() - selecciona elementos que cumplen condición
// Retorna nuevo array con enemigos vivos
let aliveEnemies = enemies.filter(enemy => enemy.health > 0);
// [{ id: 1, ... }, { id: 2, ... }]

// Solo los orcos
let orcs = enemies.filter(enemy => enemy.type === "orc");

// map() - transforma cada elemento
// Extrae solo la salud de cada enemigo
let healthValues = enemies.map(enemy => enemy.health);
// [100, 50, 0]
```

Más Métodos de Arrays

- **find()**: Retorna el **primer** elemento que cumple condición
- **findIndex()**: Retorna el **índice** del primer elemento que cumple condición
- **every()**: Retorna `true` si **todos** los elementos cumplen condición
- **some()**: Retorna `true` si **algún** elemento cumple condición
- **reduce()**: Reduce array a un único valor (suma, acumulación, etc.)

```
// find() - retorna PRIMER elemento que cumple condición
let firstOrc = enemies.find(enemy => enemy.type === "orc"); // { id: 1, health: 100, type: "orc" }

// findIndex() - retorna índice del primer match
let orcIndex = enemies.findIndex(enemy => enemy.type === "orc"); // 0

// every() - ¿TODOS cumplen la condición?
let allDead = enemies.every(enemy => enemy.health === 0); // false -> no todos muerto

// some() - ¿ALGUNO cumple la condición?
let someDead = enemies.some(enemy => enemy.health === 0); // true -> al menos un muerto

// reduce() - reduce array a un solo valor
// Suma total de la salud de todos los enemigos
let totalHealth = enemies.reduce((sum, enemy) =>
    sum + enemy.health, 0); // 0 es valor inicial
// 150 (100 + 50 + 0)
```

Modificar Arrays

- **push()**: Añade elemento al final - modifica el array original
- **pop()**: Elimina y retorna último elemento
- **shift()**: Elimina y retorna primer elemento
- **unshift()**: Añade elemento al principio
- **splice()**: Elimina/inserta elementos en cualquier posición

```
1  // push() - añadir al final (modifica original)
2  enemies.push({ id: 4, health: 75, type: "troll" });
3
4  // shift() - quitar del principio
5  let firstEnemy = enemies.shift();
6
7  // pop() - quitar del final
8  let lastEnemy = enemies.pop();
9
10 // splice(inicio, cantidad) - eliminar elementos
11 enemies.splice(1, 2); // Eliminar 2 elementos desde índice 1
12
13 // splice(inicio, 0, elemento) - insertar sin eliminar
14 enemies.splice(1, 0, newEnemy); // Insertar en índice 1
15
16 // unshift() - añadir al principio
17 enemies.unshift({ id: 0, health: 50, type: "scout" });
```

Destructuring de Arrays

- **Destructuring**: Extraer valores de arrays a variables individuales
- **Rest operator (...)**: Captura “el resto” de elementos
- **Sintaxis limpia**: Evita acceder por índice repetidamente

```
// Extraer valores de array a variables
let coordinates = [100, 200];
let [x, y] = coordinates;  // x = 100, y = 200

// Ignorar elementos
let [first, , third] = [1, 2, 3]; // first = 1, third = 3

// Rest operator - captura el resto de elementos
let [first, second, ...rest] = inventory;
// first = primer elemento
// second = segundo elemento
// rest = array con todos los demás

// En parámetros de función
function showCoords([x, y]) {
    console.log(`X: ${x}, Y: ${y}`);
}
```

Control de Flujo

Sentencias Básicas

- **if-else**: Ejecuta código según condición verdadera o falsa
- **else if**: Permite múltiples condiciones en cadena
- **Condiciones**: Expresiones que evalúan a `true` o `false`
- **Bloques** `{}` : Agrupan varias instrucciones

```
// if - else - estructura condicional básica
if (health > 50) { // Siempre recomendado con llaves
    // Se ejecuta si la condición es verdadera
    statusColor = "green";
} else if (health > 20) {
    // Se ejecuta si la primera es falsa y esta es verdadera
    statusColor = "yellow";
} else {
    // Se ejecuta si todas las anteriores son falsas
    statusColor = "red";
}

// Sin llaves para una sola instrucción (no recomendado)
if (isDead) gameOver();
```

Switch

- **switch**: Compara una expresión contra múltiples valores
- **case**: Cada posible valor a comparar
- **break**: Sale del switch (sin él, continúa al siguiente case)
- **default**: Se ejecuta si ningún case coincide

```
// switch – útil cuando comparas misma variable con muchos valores
switch (gameState) {
    case "menu":
        // Se ejecuta si gameState === "menu"
        showMenu();
        break; // Sale del switch.

    case "rollback": // Como no hay break se ejecuta el siguiente
        rollback();

    case "playing":
        updateGame();
        break;

    default:
        // Se ejecuta si ningún case coincide
        handleUnknownState();
}
```

Loops

- **for tradicional**: Cuando sabes cuántas iteraciones necesitas
- **for…of**: Itera sobre valores de un array (ES2015+) - más limpio
- **for…in**: Itera sobre claves/propiedades de un objeto
- **break**: Sale del loop inmediatamente
- **continue**: Salta a la siguiente iteración

```
// for tradicional - control total sobre índice
for (let i = 0; i < enemies.length; i++) {
    updateEnemy(enemies[i]);
    // i = 0, 1, 2, ... hasta length-1
}

// for...of - itera valores (ES2015+) - MÁS LIMPIO
for (let enemy of enemies) {
    updateEnemy(enemy); // enemy es el valor directamente
}

// for...in - itera propiedades/claves (para objetos)
for (let key in gameConfig) {
    console.log(key, gameConfig[key]);
    // key = "width", "height", etc.
}
```

While

- **while**: Repite mientras la condición sea verdadera
- **Cuidado con loops infinitos**: Asegúrate que la condición eventualmente sea falsa
- **do…while**: Ejecuta al menos una vez, luego verifica condición

```
// while - repite mientras la condición sea true
while (isGameRunning && playerLives > 0) {
    processGameFrame();
    // IMPORTANTE: algo dentro debe cambiar la condición
    // o será un loop infinito
}

// do...while - ejecuta AL MENOS una vez
let input;
do {
    input = prompt("Ingresa comando:");
} while (input !== "quit");
```

Funciones

Declaración de Funciones

- **function**: Palabra clave para declarar funciones
- **Parámetros**: Valores que recibe la función (entre paréntesis)
- **return**: Devuelve un valor y termina la función
- **Expresión de función**: Asignar función a una variable
- **Hoisting**: Las declaraciones se mueven al inicio (se pueden llamar antes de declarar)

```
// Declaración tradicional - se puede llamar antes de declarar
function calculateDamage(baseDamage, criticalHit) {
    // Verifica si hay golpe crítico
    if (criticalHit) {
        return baseDamage * 2; // Retorna el doble
    }
    return baseDamage; // Retorna daño normal
}

// Uso
let damage = calculateDamage(50, true); // 100
```

```
// Expresión de función - asignada a variable
let heal = function(amount) {
    player.health += amount; // Suma salud
    // Limita la salud al máximo
    if (player.health > player.maxHealth) {
        player.health = player.maxHealth;
    }
};

// Uso
heal(20);
```

Arrow Functions (ES2015+)

- **Sintaxis más concisa** que funciones tradicionales
- **No tienen su propio `this`** : Heredan `this` del contexto (útil en callbacks)
- **Retorno implícito**: Si es una sola expresión, retorna automáticamente (sin `return`)
- **Ideal para**: Callbacks, funciones cortas, map/filter/reduce

```
1  // Arrow function completa con llaves
2  let movePlayer = (deltaX, deltaY) => {
3      player.x += deltaX; // Mueve en X
4      player.y += deltaY; // Mueve en Y
5  };
6
7  // Arrow function con una expresión - retorno implícito
8  let isAlive = (entity) => entity.health > 0;
9  // Equivalente a: function(entity) { return entity.health > 0; }
10
11 // Sin parámetros - paréntesis vacíos
12 let generateRandomID = () => Math.random().toString(36);
13
14 // Un parámetro - paréntesis opcionales
15 let double = x => x * 2;
```

Parámetros de Función

- **Parámetros por defecto**: Valor asignado si no se pasa argumento
- **Rest parameters (...)**: Captura argumentos restantes en un array
- **Destructuring**: Extrae propiedades de objetos directamente en parámetros
- **Orden**: Parámetros normales, luego con default, luego rest

```
// Parámetros por defecto (ES2015+)
function createEnemy(health = 100, damage = 10) {
    return { health, damage };
}
// Si no pasas argumentos, usa los valores por defecto
createEnemy(); // { health: 100, damage: 10 }
createEnemy(50); // { health: 50, damage: 10 }

// Rest parameters - captura argumentos restantes en array
function logMessage(level, ...messages) {
    console.log(`[${level}]`, ...messages);
}
logMessage("ERROR", "Falló", "al cargar", "recurso");
// messages = ["Falló", "al cargar", "recurso"]
```

Closures

- **Closure**: Función que “recuerda” variables de su contexto exterior
- **Encapsulación**: Variables privadas que solo la función puede modificar
- **Estado persistente**: Mantiene estado entre llamadas sin variables globales
- **Útil para**: Contadores, factory functions, datos privados

```
// Closure – función que retorna objeto con métodos
function createCounter(initialValue = 0) {
    // Variable privada – solo accesible dentro
    let count = initialValue;

    // Retorna objeto con métodos que acceden
    // a 'count'
    return {
        // Incrementa y retorna nuevo valor
        increment: () => ++count,

        // Decrementa y retorna nuevo valor
        decrement: () => --count,

        // Retorna valor actual
        getValue: () => count
    };
}
```

```
// Cada contador tiene su propio estado
let scoreCounter = createCounter(0);
scoreCounter.increment(); // 1
scoreCounter.increment(); // 2
console.log(scoreCounter.getValue()); // 2

// 'count' NO es accesible desde fuera
// console.log(count);
// ERROR – no existe aquí
```

Manejo de Excepciones

Try-Catch-Finally

- **try**: Bloque donde puede ocurrir un error
- **catch**: Captura el error si ocurre y maneja la situación
- **finally**: Se ejecuta SIEMPRE (haya error o no) - útil para limpieza
- **error.message**: Descripción del error

```
// Manejo de errores al cargar partida guardada
try {
    // Intenta parsear JSON - puede fallar si está corrupto
    let gameData = JSON.parse(savedGameString);
    loadGame(gameData); // Carga el juego

} catch (error) {
    // Se ejecuta SOLO si hay error en try
    console.error('Error loading game:', error.message);
    showErrorDialog('No se pudo cargar la partida');

} finally { // Opcional
    // Se ejecuta SIEMPRE (con o sin error)
    hideLoadingSpinner();
}
```

Lanzar Excepciones

- **throw**: Lanza un error manualmente
- **new Error()**: Crea objeto de error con mensaje
- **Validaciones**: Úsalo para validar inputs antes de procesar
- **El error sube**: Si no hay catch, el error se propaga hacia arriba

```
// Función que valida input y lanza errores si es inválido
function validatePlayerInput(input) {
    // Verifica que no esté vacío
    if (!input || input.trim() === '') {
        throw new Error('El nombre no puede estar vacío');
    }

    // Verifica longitud máxima
    if (input.length > 20) {
        throw new Error('El nombre es demasiado largo');
    }

    // Si pasa las validaciones, retorna input limpio
    return input.trim();
}
```

Almacenamiento de Datos

Local Storage

- **localStorage**: Almacena datos en el navegador de forma permanente
- **Capacidad**: ~5-10MB (depende del navegador)
- **Solo strings**: Debes convertir objetos a JSON con `JSON.stringify()`
- **Persiste**: Datos permanecen incluso cerrando navegador/reiniciando PC
- **Por dominio**: Cada sitio web tiene su propio localStorage

```
// Guardar datos simples (solo strings)
localStorage.setItem('playerName', 'Jugador1');
localStorage.setItem('highScore', '15000');

// Guardar objetos - primero convertir a JSON string
const gameConfig = { volume: 0.8, difficulty: 'normal' };
localStorage.setItem('gameConfig', JSON.stringify(gameConfig));

// Leer datos simples
const playerName = localStorage.getItem('playerName');
// "Jugador1"

// Eliminar un dato específico
localStorage.removeItem('playerName');

// Eliminar TODOS los datos (usar con cuidado)
localStorage.clear();
```

Session Storage

- **sessionStorage**: API idéntica a localStorage
- **Duración**: Solo durante la sesión actual (cerrar pestaña = se borra)
- **Por pestaña**: Cada pestaña tiene su propio sessionStorage
- **Uso típico**: Datos temporales como estado actual del juego
- **No se comparte**: Entre pestañas ni ventanas

```
1 // API idéntica a localStorage
2 sessionStorage.setItem('tempData', 'valor temporal');
3 const tempData = sessionStorage.getItem('tempData');
4 sessionStorage.removeItem('tempData');
5 sessionStorage.clear();
6
7 // Ideal para estado temporal del juego actual
8 sessionStorage.setItem('currentGameState',
9     JSON.stringify(gameState));
```

Cookies

- **Cookies**: Método antiguo de almacenamiento
- **Capacidad limitada**: Solo 4KB
- **Se envían al servidor**: En cada petición HTTP (aumenta tráfico)
- **API manual**: Más complicado que localStorage
- **Expiración configurable**: Puedes definir cuándo expiran

```
// Escribir Cookie con expiración (30 días desde ahora)
// Sin fecha es la sesión
const fecha = new Date();
fecha.setTime(fecha.getTime() + (30 * 24 * 60 * 60 * 1000));
document.cookie = `highScore=15000; expires=${fecha.toUTCString()}; path=/`;

// Leer cookie (complicado - no hay API directa)
function getCookie(nombre) {
    const value = `; ${document.cookie}`;
    const parts = value.split(`; ${nombre}=`);
    if (parts.length === 2) return parts.pop().split(';').shift();
    return null; // No encontrada
}

// Uso
let highScore = getCookie('highScore'); // "15000"

```

Comparación de Métodos

- **localStorage**: Mejor opción para datos persistentes (configuración, progreso)
- **sessionStorage**: Perfecto para datos temporales de la sesión actual
- **Cookies**: Solo si necesitas comunicar con el servidor

Característica	localStorage	sessionStorage	Cookies
Capacidad	~5-10MB	~5-10MB	4KB
Persistencia	Hasta eliminar manualmente	Solo sesión actual	Configurable (expires)
Envío al servidor	No	No	Sí (automático)
API	Síncrona y simple	Síncrona y simple	Manual y compleja
Compartido	Entre pestañas	NO (por pestaña)	Entre pestañas