JavaScript: Fundamentos del Lenguaje

1. Variables y Tipos de Datos

JavaScript es un lenguaje de tipado dinámico que soporta varios tipos de datos primitivos y objetos.

Los tipos primitivos incluyen: string (cadenas de texto), number (números enteros y decimales), boolean (true/false), null, undefined, symbol y bigint.

Para declarar variables usamos tres palabras clave:
- var: tiene scope de función y permite redeclaración (uso legacy, evitar)
- let: tiene scope de bloque, permite reasignación pero no redeclaración
- const: tiene scope de bloque, no permite reasignación ni redeclaración

Ejemplo:
```
const nombre = "María";
let edad = 25;
let activo = true;
```

Las constantes deben inicializarse al declararse y su valor no puede cambiar. Sin embargo, si la constante es un objeto o array, sus propiedades internas sí pueden modificarse.

2. Funciones

Las funciones en JavaScript pueden declararse de varias formas:

Declaración de función (function declaration):
```
function sumar(a, b) {
  return a + b;
}
```

Expresión de función (function expression):
```
const restar = function(a, b) {
  return a - b;
};
```

Funciones flecha (arrow functions):
```
const multiplicar = (a, b) => a * b;
const dividir = (a, b) => {
  if (b === 0) throw new Error("División por cero");
  return a / b;
};
```

Las funciones flecha no tienen su propio 'this', lo heredan del contexto léxico donde fueron creadas. Esto las hace ideales para callbacks y métodos de arrays.

Los parámetros pueden tener valores por defecto:
```
function saludar(nombre = "Invitado") {
```

```
  return "Hola, " + nombre;
}
```

También existe el operador rest (...) para recibir múltiples argumentos:

```
function sumarTodos(...numeros) {
  return numeros.reduce((acc, n) => acc + n, 0);
}
```

3. Arrays y Métodos de Array

Los arrays en JavaScript son colecciones ordenadas que pueden contener cualquier tipo de dato.

Creación:

```
const frutas = ["manzana", "banana", "naranja"];
const numeros = new Array(1, 2, 3, 4, 5);
```

Métodos importantes para transformación (no mutan el original):

- map(): transforma cada elemento
  ```
  const dobles = numeros.map(n => n * 2); // [2, 4, 6, 8, 10]
  ```

- filter(): filtra elementos según condición
  ```
  const pares = numeros.filter(n => n % 2 === 0); // [2, 4]
  ```

- reduce(): reduce a un único valor
  ```
  const suma = numeros.reduce((acc, n) => acc + n, 0); // 15
  ```

- find(): encuentra el primer elemento que cumple condición
  ```
  const mayor3 = numeros.find(n => n > 3); // 4
  ```

- some(): verifica si algún elemento cumple condición
  ```
  const hayPares = numeros.some(n => n % 2 === 0); // true
  ```

- every(): verifica si todos cumplen condición
  ```
  const todosPares = numeros.every(n => n % 2 === 0); // false
  ```

Métodos que mutan el array original:

- push(): añade al final
- pop(): elimina del final
- shift(): elimina del inicio
- unshift(): añade al inicio
- splice(): añade/elimina en posición específica
- sort(): ordena (¡cuidado con números!)

Para ordenar números correctamente:

```
numeros.sort((a, b) => a - b); // ascendente
numeros.sort((a, b) => b - a); // descendente
```

4. Objetos y Desestructuración

Los objetos son colecciones de pares clave-valor que representan entidades.

Creación de objetos:

```
const usuario = {
  nombre: "Ana",
  edad: 28,
  email: "ana@ejemplo.com",
  direccion: {
    ciudad: "Madrid",
    pais: "España"
  }
};

Acceso a propiedades:
console.log(usuario.nombre); // "Ana"
console.log(usuario["edad"]); // 28

Desestructuración de objetos:
const { nombre, edad } = usuario;
const { direccion: { ciudad } } = usuario; // desestructuración anidada

Desestructuración con alias:
const { nombre: nombreUsuario } = usuario;

Desestructuración con valores por defecto:
const { telefono = "No disponible" } = usuario;

Spread operator para copiar/combinar objetos:
const usuarioActualizado = { ...usuario, edad: 29 };
const completo = { ...usuario, ...datosAdicionales };

Object methods útiles:
- Object.keys(obj): array de claves
- Object.values(obj): array de valores
- Object.entries(obj): array de pares [clave, valor]
- Object.assign(target, source): copia propiedades
```

5. Programación Asíncrona

JavaScript maneja operaciones asíncronas mediante callbacks, Promises y async/await.

```
Callbacks (patrón antiguo):
function obtenerDatos(callback) {
  setTimeout(() => {
    callback(null, { data: "resultado" });
  }, 1000);
}

Promises (ES6):
const promesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve({ data: "resultado" });
  }, 1000);
});

promesa
```

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

Async/Await (ES2017):
async function obtenerUsuario(id) {
  try {
    const respuesta = await fetch("/api/usuarios/" + id);
    const usuario = await respuesta.json();
    return usuario;
  } catch (error) {
    console.error("Error:", error);
    throw error;
  }
}

Ejecución paralela con Promise.all():
async function obtenerTodo() {
  const [usuarios, productos] = await Promise.all([
    fetch("/api/usuarios").then(r => r.json()),
    fetch("/api/productos").then(r => r.json())
  ]);
  return { usuarios, productos };
}

Promise.allSettled() espera a que todas terminen sin importar si fallan:
const resultados = await Promise.allSettled([promesa1, promesa2, promesa3]);
```

6. Manejo de Errores

El manejo correcto de errores es fundamental para aplicaciones robustas.

```
Try-catch básico:
try {
  const resultado = operacionRiesgosa();
  console.log(resultado);
} catch (error) {
  console.error("Ocurrió un error:", error.message);
} finally {
  // Se ejecuta siempre, haya error o no
  limpiarRecursos();
}

Crear errores personalizados:
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

function validarEdad(edad) {
  if (edad < 0 || edad > 150) {
    throw new ValidationError("Edad inválida: " + edad);
```

```
  }
  return true;
}
```

Manejo de errores en async/await:

```
async function procesarDatos() {
  try {
    const datos = await obtenerDatos();
    return transformar(datos);
  } catch (error) {
    if (error instanceof NetworkError) {
      // Reintentar
      return await procesarDatos();
    }
    throw error; // Re-lanzar otros errores
  }
}
```

Es importante siempre manejar los errores en Promises:

```
promesa
  .then(resultado => procesar(resultado))
  .catch(error => manejarError(error)); // ¡No olvidar!
```