



Gobierno del
CHACO

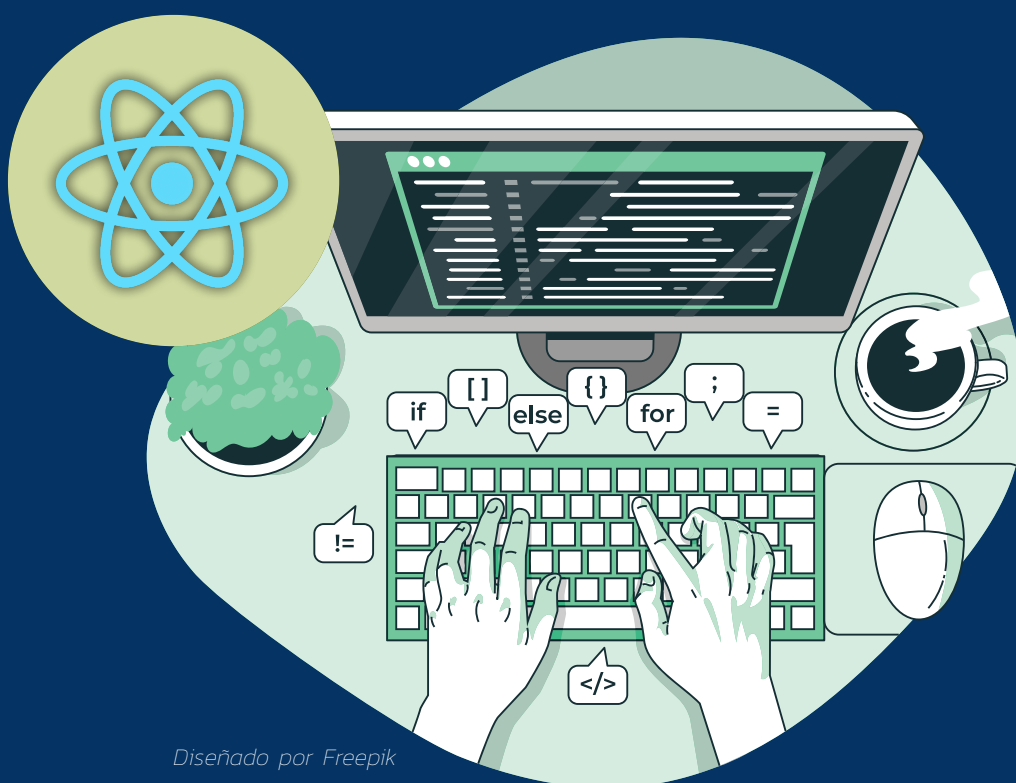
Ministerio
de la Producción y el Desarrollo
Económico Sostenible



INFORMATARIO

ETAPA 3: ESPECIALIZACIONES

REACT



Diseñado por Freepik

Apunte N° 1

JAVASCRIPT PARA REACT

1



Sentencia vs expresiones



Objetivo: Diferenciar entre lo que produce un valor (expresión) y lo que ejecuta una acción (sentencia).

Definiciones

Expresión:

Fragmento de código que devuelve un valor.

// Ejemplos de expresiones

`5 * 10`

// Devuelve 50

`"Hola"`

// Devuelve la cadena "Hola"

`[1, 2, 3].pop()`

// Devuelve 3

`8 > 50`

// Devuelve false

Sentencia:

Instrucción que realiza una acción (declara variables, controla flujo, lanza errores).

// Ejemplos:

`if`

`for`

`throw`

`const x = 10;`

Ejemplo

```
console.log(5 * 10);
```

`5 * 10` es una expresión dentro de la sentencia `console.log`

- **Expresión:** `5 * 10` (evalúa a 50).
- **Sentencia:** `console.log(...)` (ejecuta la acción de imprimir).

Características

Expresiones:

- Pueden combinarse: `(4 + 4) * 2`.
- Son valores en asignaciones: `const result = 1 + 3;`
- Funcionan como condiciones: `if (result > 10) { ... }`.

Sentencias:

- Usan palabras clave: `if`, `for`, `throw`.
- Contienen expresiones: `for (let i; i > 0; i++) { ... }`.
- No pueden anidarse donde se espera un valor. Ej: `console.log(if (...) {})`.



¿Cómo recordarlo?

Metáfora:

- **Expresión:** Como los ingredientes de una receta (valores concretos).
- **Sentencia:** Como los pasos de la receta (acciones a seguir).

Regla práctica:

- Si puedes ponerlo dentro de `console.log()`, es una expresión.
- Si controla cuándo o cómo se ejecuta el código, es una sentencia.

Casos Comunes

Expresión de función:

`const suma = function() { ... }`
(la función es un valor asignado).

Sentencia de función:

`function suma() { ... }`
(declara una acción reutilizable).

Interpolación de strings



Objetivo: Diferenciar entre lo que produce un valor (expresión) y lo que ejecuta una acción (sentencia).

Dos Formas de Strings Dinámicos

Concatenación tradicional (+):

Une strings y variables con el operador +.
Ejemplo:

`'Hola ' + firstName + '!'`

(difícil de leer en casos complejos).

Template literals (`` y \${}):

Permiten incrustar expresiones directamente dentro del string.
Ejemplo:

`Hola ${firstName}!`

(más limpio y flexible).

```
const firstName = 'Ivan';
const edad = 25;

// Usando concatenación tradicional:
const saludoClasico = 'Hola ' + firstName + '!';

// Usando template literals (recomendado):
const saludoModerno = `Hola ${firstName}!`;

// Interpolación con operaciones:
console.log(`El año que viene: ${edad + 1} años`);
```



Características de Template Literals

Sintaxis:

- Strings entre backticks (``).
- Expresiones dentro de `${}`.

Ventajas:

- **Legibilidad:** Mejor manejo de variables y texto.
- **Expresiones:** Cualquier operación válida en JS funciona dentro de `${}`.

```
${edad * 2}
${nombre.toUpperCase()}
```

- **Multi-línea:** Permiten saltos de línea sin `\n`.

Casos de Uso

- **Variables simples:**

```
`Total: ${precio}` // "Total: $100"
```

- **Operaciones matemáticas:**

```
`Promedio: ${ (a + b) / 2 }`
```

- **Llamadas a funciones:**

```
`Fecha: ${new Date().toLocaleDateString()}`
```

Funciones



Objetivo: Entender diferentes formas de declarar y usar funciones.

Tipos de Funciones

Declaración:

- Se define con `function`.
- **Hoisting:** Puede usarse antes de su declaración.

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

Expresión:

- Función asignada a una variable.
- **Anónima:**

```
const multiply = function(x, y) { ... };
```



(Continuación de página anterior: tipo de función de expresión)

- **Nombrada:**

```
const multiply = function funcName(x, y) { ... };
```

(Útil para depuración/recursión).

Arrow Function:

- Sintaxis concisa: **(x, y) => x * y.**
- Sin contexto propio (**this** heredado).

Método de Objeto:

- Función dentro de un objeto:

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

Constructor:

- Crea instancias con **new**:

```
function Car(make, model) {
  this.make = make;
}
const car = new Car('honda', 'civic');
```

Conceptos Clave

Parámetros vs Argumentos:

MLos parámetros son variables en la definición (x, y), los argumentos son valores concretos al llamar la función.

Valor de retorno:

Si no hay **return**, la función devuelve **undefined**.

Diferencias entre Arrow y tradicional:

- **Arrow:** Sin **arguments**, no puede ser constructor.
- **Tradicional:** Tiene su propio **this**.

Object destructuring



Objetivo: Extraer valores de objetos de forma concisa y mejorar la legibilidad del código.



1. Desestructuración Básica

```
const user = {
  name: 'Ivan Sevilla',
  country: 'Argentina'
};

// Destructuración tradicional:
const name = user.name;      // Forma verbosa
const country = user.country;

// Con destructuración:
const { name, country } = user;
console.log(name); // 'Ivan Sevilla'
```

Beneficio: Extrae múltiples propiedades en una línea.

Propiedades faltantes: Si la propiedad no existe, la variable será **undefined**:

```
const { pepeveraz } = user; // pepeveraz = undefined
```

2. Desestructuración en Parámetros de Funciones

Evolución del código:

a). Sin desestructuración:

```
function validateUser(user) {
  if (typeof user.name !== 'string') return false;
  if (user.password.length < 12) return false;
  return true;
}
```

b). Desestructuración interna:

```
function validateUser(user) {
  const { name, password } = user; // Extrae dentro de la función
  // ... misma lógica
}
```

c). Desestructuración directa en parámetros:

```
function validateUser({ name, password }) { // ¡Aquí la magia!
  if (typeof name !== 'string') return false;
  if (password.length < 12) return false;
  return true;
}
```

Ventajas:

- Claridad inmediata sobre las propiedades necesarias.
- Reduce repetición de **user.propiedad**.

3. Casos Útiles

Renombrar variables:

```
const { name: nombreCompleto } = user;
console.log(nombreCompleto); // 'Ivan Sevilla'
```

Valores por defecto:

```
const { role = 'usuario' } = user; // Si user.role no existe, usa 'usuario'
```

Objetos anidados:

```
const usuario = {
  datos: { email: 'ivan@ejemplo.com' }
};
const { datos: { email } } = usuario;
console.log(email); // 'ivan@ejemplo.com'
```

Property Value Shorthand



Objetivo: Simplificar la creación de objetos cuando los nombres de propiedades coinciden con variables existentes.

1. Concepto Clave

Cuando el nombre de una propiedad de un objeto es igual al nombre de la variable que contiene su valor, puedes omitir la repetición:

Resultado en ambos casos (*imagen 10*):

```
{ firstName: 'Ivan', age: 25 }.
```

2. Beneficios

- **Menos código:** Elimina repetición de nombres.
- **Legibilidad:** Clarifica que el nombre de la propiedad y la variable son el mismo.
- **Mantenimiento:** Si renombra la variable, la propiedad se actualiza automáticamente (dependiendo del IDE).

```
// Sin shorthand (redundante):
const user = {
  firstName: firstName,
  age: age
};

// Con shorthand (simplificado):
const userWithPropertyShorthand = { firstName, age };
```

imagen 10

3. Casos de Uso Comunes

Inicialización rápida de objetos:

```
const id = 100;
const rol = 'admin';
const usuario = { id, rol }; // { id: 100, rol: 'admin' }
```

Retorno de objetos en funciones:

```
const crearPerfil = (nombre, email) => ({ nombre, email });
crearPerfil('Ana', 'ana@ejemplo.com'); // { nombre: 'Ana', email: 'ana@ejempl
o.com' }
```

Métodos concisos en objetos:

```
const accion = 'guardar';
const estado = 'éxito';
const evento = { accion, estado, timestamp: Date.now() }; // Mezcla con propi
edades normales
```




Array destructuring



Objetivo: Extraer elementos de arrays de forma concisa usando patrones de asignación.

1. Concepto Básico

- **Sintaxis clave:** `[variable1, variable2, ...]`.
- **Espacios vacíos (, ,):** Ignoran elementos intermedios.

```
const fruits = ['apple', 'banana', 'cantaloupe', 'kiwi'];

// Sin destructuración:
const firstFruit = fruits[0]; // 'apple'
const thirdFruit = fruits[2]; // 'cantaloupe'

// Con destructuración:
const [first, , third] = fruits; // first = 'apple', third = 'cantaloupe'
```

3. Casos de Uso

Valores por defecto:

```
const [a = 0, b = 10] = [5];
// a = 5, b = 10
```

Intercambio de variables:

```
let x = 1, y = 2;
[x, y] = [y, x]; // x = 2, y = 1
```

Combinar con object destructuring:

```
const [ , { nombre }] = ['dato', { nombre: 'Ivan' }]; // nombre = 'Ivan'
```

Operador rest (...):

```
const [primero, ...resto] = [1, 2, 3]; // resto = [2, 3]
```



Valores Truthy y Falsy



Objetivo: Entender qué valores se evalúan como **true** o **false** en contextos booleanos (como condicionales).

1. Valores Falsy

Son los únicos 6 valores que se convierten a false:

```
false
null
undefined
'' // (string vacío)
0 // (incluye 0.0, -0, 0n)
NaN
```

2. Valores Truthy

Cualquier valor que no sea falsy. Ejemplos comunes:

```
'hola' // (strings no vacíos)
42, 10, 3.14 // (números ≠ 0)
[], {} // (arrays/objetos, incluso vacíos)
true, function(){}, new Date()
```

3. Conversión a Booleano

```
// Usando Boolean():
Boolean(4); // true (truthy)
Boolean(''); // false (falsy)

// Usando doble NOT (!!):
!!'Hola'; // true
!!0; // false

// Con operador NOT simple (!):
!4; // false (porque 4 es truthy)
!NaN; // true (porque NaN es falsy)
```



Operadores lógicos



Objetivo: Controlar flujos lógicos y acceder a valores de forma segura y eficiente.

1. Operador Ternario (? :)

- Equivalente a un `if/else` en una línea.
- Regla: `condición ? valor si true : valor si false`.

```
const message = bottle.fullOfSoda
  ? 'Tiene soda'
  : 'No tiene soda';
```

2. AND Lógico (&&)

Retorna:

- El primer valor falsy encontrado, o
- El último valor truthy si todos son verdaderos

```
if (isLoggedIn && userRole === 'admin') { ... }
```

Ejemplo útil:

```
const result = myAge < 50 && myAge; // Retorna 35 si myAge = 35
```

3. OR Lógico (||)

Retorna:

- El primer valor truthy encontrado, o
- El último valor si todos son falsy.

Cuidado: Considera `0`, `' '`, o `false` como falsy (no siempre deseado).

```
const src = userImageSrc || teamImageSrc || defaultImageSrc;
```

4. Operador Nullish (??)



Retorna: el lado derecho solo si el izquierdo es **null** o **undefined** :

```
a = a ?? 0; // Asigna 0 solo si `a` es null/undefined
```

Diferencia con **||** :

```
0 ?? 24 // → 0 (válido)
0 || 24 // → 24 (pérdida del 0)
```

5. Optional Chaining (?.)

Retorna **undefined** si alguna propiedad en la cadena no existe.
(Evita errores cuando una propiedad no existe).

Funciona para:

- Propiedades: **obj.prop?.nested**
- Funciones: **obj.method?.()**

```
const dogName = adventurer.dog?.name; // undefined (sin error)
```

Asignar vs mutar



Objetivo: Entender por qué **const** permite modificar objetos/arrays pero no reasignar variables.

1. Asignación:

- **Definición:** Cambiar la referencia de una variable (apuntarla a un nuevo valor).
- **Regla con **const**:** Las variables declaradas con **const** no pueden ser reasignadas.

```
const nombre = 'Juan';
nombre = 'Ivan'; // ❌ Error: "Assignment to constant variable"
```



2. Mutación:

- **Definición:** Modificar el contenido interno de un objeto/array (sin cambiar su referencia en memoria).
- **Regla con `const`:** Los objetos/arrays declarados con `const` pueden mutarse (sus propiedades/elementos sí se modifican).

```
const persona = { nombre: 'Juan' };
persona.nombre = 'Ivan'; // ✓ Válido (mutación)
console.log(persona); // { nombre: 'Ivan' }

persona = { edad: 30 }; // ✗ Error (reasignación)
```

3. ¿Por qué funciona así?:

* `const` protege la dirección de memoria (no el contenido).

Ejemplo visual:

a). Inicio

Imagina un post-it con una dirección de memoria

```
Post-it: "Ve al estante 0x123"
Estante 0x123: { nombre: "Ana" }
```

b). Intento de reasignación (Prohibido)

Quiero cambiar el post-it a otro estante (0x456)

```
Post-it: "Ve al estante 0x456" ✗ ERROR
```

* No se puede cambiar la dirección del post-it (No se puede reasignar persona).

c). Mutación (Permitido)

Quiero modificar el contenido del estante (0x123)

```
Post-it: "Ve al estante 0x123"
Estante 0x123: { nombre: "Carlos" } ✓ PERMITIDO
```

* Puedo cambiar los valores dentro del estante, pero no mover el post-it.

Conclusión:

- `const` fija la referencia (el post-it no cambia de estante).
- Los valores internos sí pueden cambiar (podemos modificar el contenido del estante).



Código equivalente en JavaScript:

```
const persona = { nombre: "Ana" };
```

```
// ❌ ERROR: No se puede reasignar  
persona = { nombre: "Carlos" };
```

```
// ✅ Permitido: Mutación  
persona.nombre = "Carlos";
```

3. Casos Comunes

Arrays:

```
const numeros = [1, 2];  
numeros.push(3); // ✅ Válido (mutación)  
numeros = [4, 5]; // ❌ Error (reasignación)
```

Congelar objetos:

- Usa `Object.freeze()` para evitar mutaciones:

```
const persona = Object.freeze({ nombre: 'Juan' });  
persona.nombre = 'Ivan'; // ❌ No surte efecto (modo estricto: error)
```

Rest vs spread



Objetivo: Manipular arrays y objetos de forma flexible, evitando mutaciones innecesarias.

1. Operador Rest (...)

Función:

Agrupar elementos en un array (útil en parámetros de funciones). (ver ejemplo debajo)

```
function sum(...valores) { // Agrupa todos los argumentos en un array  
  return valores.reduce((total, num) => total + num, 0);  
}  
console.log(sum(1, 2, 3)); // 6
```



2. Operador Spread (...)

Función:

Expandir un iterable (array/objeto) en elementos individuales.

Casos de uso:

Llamar funciones:

```
const numeros = [1, 2, 3];
console.log(sum(...numeros)); // 6 (equivale a sum(1, 2, 3))
```

Copiar arrays/objetos:

```
const copiaArray = [...numeros]; // Crea un nuevo array
const copiaObjeto = { ...objeto }; // Crea un nuevo objeto
```

Combinar estructuras:

```
const combinado = [...array1, ...array2]; // Unión de arrays
const nuevoObjeto = { ...obj1, ...obj2 }; // Fusión de propiedades
```

3. Reglas Clave

- **Rest:** Siempre en parámetros de funciones o al desestructurar arrays.
- **Spread:** En argumentos de funciones, literales de arrays/objetos, o para sobrescribir propiedades.
- **Objetos:** Spread solo copia propiedades enumerables (no métodos privados o prototipos).

Métodos de arrays



Objetivo: Manipular y consultar arrays de objetos de forma eficiente.



1. Métodos más utilizados

```
const dogs = [
  {
    id: 'dog-1',
    name: 'Poodle',
    temperament: [
      'Intelligent',
      'Active',
      'Alert',
      'Faithful',
      'Trainable',
      'Instinctual',
    ],
  },
  {
    id: 'dog-2',
    name: 'Bernese Mountain Dog',
    temperament: ['Affectionate', 'Intelligent', 'Loyal', 'Faithful'],
  },
  {
    id: 'dog-3',
    name: 'Labrador Retriever',
    temperament: [
      'Intelligent',
      'Even Tempered',
      'Kind',
      'Agile',
      'Outgoing',
      'Trusting',
      'Gentle',
    ],
  },
];
```

```
dogs.forEach((dog) => console.log(dog));

dogs.find((dog) => dog.name === 'Bernese Mountain Dog');
// {id: 'dog-2', name: 'Bernese Mountain Dog', ...etc}

dogs.some((dog) => dog.temperament.includes('Aggressive'));
// false
```




```
dogs.some((dog) => dog.temperament.includes('Trusting'));
// true

dogs.every((dog) => dog.temperament.includes('Trusting'));
// false

dogs.every((dog) => dog.temperament.includes('Intelligent'));
// true

dogs.map((dog) => dog.name);
// ['Poodle', 'Bernese Mountain Dog', 'Labrador Retriever']

dogs.filter((dog) => dog.temperament.includes('Faithful'));
// [{id: 'dog-1', ..etc}, {id: 'dog-2', ...etc}]
```

2. Características

Método	Retorna	¿Modifica el array original?
<code>forEach</code>	<code>void</code>	No
<code>find</code>	Primer elemento encontrado	No
<code>some</code>	<code>true/false</code>	No
<code>every</code>	<code>true/false</code>	No
<code>map</code>	Nuevo array (transformado)	No
<code>filter</code>	Nuevo array (elementos filtrados)	No

3. Reglas prácticas

- **find vs filter:** Usa `find` para un solo elemento, `filter` para múltiples.
- **some vs every:**
 - ¿Basta con uno? → `some`.
 - ¿Necesitas todos? → `every`.
- **Inmutabilidad:** `map` y `filter` devuelven nuevos arrays (ideal para React).



ES Modules



Objetivo: Exportar e importar valores entre archivos usando la sintaxis modular estándar.

1. Named Exports (Exportaciones Nombradas)

- **Uso:** Para múltiples valores que necesitan nombres específicos.
- **Importar:** Requiere usar llaves `{}` y el nombre exacto (o alias).

```
// Exportar valores individualmente (variables/funciones)
export const NUMERO_MAGICO = 5;

// Exportar una función
export function doubleNum(num) {
  return num * 2;
}

// Exportar después de declarar
const mensaje = 'Hola módulos!';
export { mensaje };
```

2. Default Export (Exportación por Defecto)

- **Uso:** Para exportar un valor principal (ej: una clase, función o objeto).
- **Importar:** Sin llaves, y puede renombrarse libremente.

```
// Solo un valor por archivo puede ser default
const valorDefault = 99;
export default valorDefault;
```


3. Importar Valores

- **Alias:** Usa `as` para renombrar (útil para evitar conflictos).
- **Rutas:** Siempre incluye la extensión (`.js`) en navegadores.

```
// Importar default + named exports
import miDefault, {
  NUMERO_MAGICO,
  doubleNum,
  mensaje as alias
} from './modulo.js';

console.log(miDefault); // 99
console.log(doubleNum(NUMERO_MAGICO)); // 10
```






Asincronismo

 **Objetivo:** Manejar operaciones no bloqueantes (como llamadas a APIs) usando Promesas y **async/await**.

1. Promesas (.then())

Flujo:

- fetch** inicia una petición HTTP (devuelve una Promesa).
- .then()** procesa la respuesta cuando está lista.
- Encadenamiento:** Cada **.then()** retorna una nueva Promesa.






-  Inicio
-  Enviando petición a API... (fetch)
-  Esperando respuesta...
-  Respuesta recibida → Ejecutando .then()
-  Datos listos para usar

```
fetch('https://rickandmortyapi.com/api/character')
  .then(res => res.json())
  .then(data => console.log('Nombre:', data.results[0].name));
```

2. Async/Await (Sintaxis Moderna)

Reglas:




- await** pausa la ejecución hasta que la Promesa se resuelve.
- Solo funciona dentro de funciones marcadas con **async**.
- Ventaja:** Código más lineal, secuencial y legible vs **.then()**.

-  Inicio
-  Enviando petición a API...
-  Esperando respuesta...
-  Respuesta recibida → Continuar ejecución
-  Nombre: Rick Sanchez

```
async function pedirDatos() {
  const res = await fetch('https://rickandmortyapi.com/api/character');
  const data = await res.json();
  console.log('Nombre:', data.results[1].name);
}
pedirDatos();
```

3. Orden de Ejecución (Event Loop)

Ejemplo de salida esperada:

 Inicio
 Fin (antes de obtener datos)
 Nombre: Rick Sanchez (después de la respuesta)

Explicación:

- Las operaciones asíncronas no bloquean el hilo principal.
- “Fin” aparece antes del resultado de la API, porque el fetch es asíncrono.

```
console.log('Inicio');
pedirDatos(); // Llamada asíncrona
console.log('Fin');
```

4. Manejo de Errores

Con Promesas: Usa `.catch()`:

```
fetch(url)
  .then(res => res.json())
  .catch(error => console.error('Falló:', error));
```

Con Async/Await: Usa `try/catch`:

```
async function pedirDatos() {
  try {
    const res = await fetch(url);
    // ...
  } catch (error) {
    console.error('Falló:', error);
  }
}
```



5. Conclusión:

- **Promesas** (`.then()`) → Permiten manejar asincronismo, pero pueden volverse anidadas.
- **async/await** → Código más claro y fácil de leer.
- **Event Loop** → Las tareas asíncronas no bloquean la ejecución principal.
- **Errores** → Usa `.catch()` en promesas y `try/catch` con `async/await`.