200 Preguntas sobre JavaScript y React - Guía Completa de Entrevista

JAVASCRIPT - 100 PREGUNTAS

Conceptos Fundamentales (20 preguntas)

1. ¿Cuáles son los tipos de datos primitivos en JavaScript? Los tipos primitivos son: `number`, `string`, `boolean`, `undefined`, `null`, `symbol` (ES6) y `bigint` (ES2020). Son inmutables y se almacenan por valor.

2. ¿Cuál es la diferencia entre `null` y `undefined`?

- `undefined`: Variable declarada pero no inicializada, o propiedad no existente
- `null`: Valor asignado intencionalmente que representa "sin valor" o "vacío"

3. ¿Qué es el hoisting en JavaScript? Es el comportamiento donde las declaraciones de variables (`var`) y funciones se "elevan" al top de su scope durante la fase de compilación. Solo se elevan las declaraciones, no las inicializaciones.

```javascript
console.log(x); // undefined (no error)
var x = 5;

// Se interpreta como:
var x;
console.log(x); // undefined
x = 5;
```

4. ¿Cuál es la diferencia entre `var`, `let` y `const`?

- `var`: function-scoped, hoisting, puede redeclararse
- `let`: block-scoped, temporal dead zone, puede reasignarse
- `const`: block-scoped, temporal dead zone, no puede reasignarse

5. ¿Qué es el Temporal Dead Zone? Es el período entre la entrada al scope y la declaración donde la variable existe pero no puede ser accedida. Aplica a `let` y `const`.

```javascript
console.log(x); // ReferenceError
let x = 5;
```

6. ¿Cuál es la diferencia entre `==` y `===`?

- `==`: Comparación con coerción de tipos
- `===`: Comparación estricta sin coerción de tipos

```javascript
'5' == 5; // true (coerción)
'5' === 5; // false (tipos diferentes)
```

7. ¿Qué es la coerción de tipos en JavaScript? Es la conversión automática o explícita de valores de un tipo a otro. JavaScript realiza coerción implícita en operaciones y comparaciones.

8. ¿Cómo funciona el operador `typeof`? Devuelve una string indicando el tipo de la variable. Casos especiales: `typeof null` devuelve `"object"` (bug histórico).

9. ¿Qué es NaN y cómo lo detectas? `NaN` (Not a Number) representa un valor numérico inválido. Se detecta con `Number.isNaN()` o `isNaN()`, siendo `Number.isNaN()` más preciso.

```javascript
Number.isNaN(NaN);    // true
Number.isNaN('hello'); // false
isNaN('hello');       // true (coerción)
```

10. ¿Qué es el scope en JavaScript? El scope determina la accesibilidad de variables en diferentes partes del código. Tipos: global, function, block (ES6).

11. ¿Qué es el scope chain? Es la cadena de scopes que JavaScript recorre para resolver referencias de variables, desde el scope actual hasta el global.

12. ¿Qué son las expresiones y statements?

- **Expresión**: Produce un valor (`2 + 2`, `true`, `function() {}`)
- **Statement**: Ejecuta una acción (`if`, `for`, `var x = 5`)

13. ¿Qué es el strict mode? `"use strict"` habilita un modo más estricto que previene errores comunes: no permite variables no declaradas, elimina `this` global, etc.

14. ¿Cómo se maneja la herencia en JavaScript? A través del prototype chain. Cada objeto tiene una referencia a su prototipo, formando una cadena hasta `Object.prototype`.

15. ¿Qué es el operador de coalescencia nula (`??`)? Devuelve el operando derecho si el izquierdo es `null` o `undefined`. Diferente de `||` que considera todos los valores falsy.

```javascript
null ?? 'default';    // 'default'
undefined ?? 'default'; // 'default'
0 ?? 'default';       // 0
'' ?? 'default';      // ''
```

16. ¿Qué es el optional chaining (`?.`)? Permite acceder a propiedades anidadas sin errores si algún eslabón es `null` o `undefined`.

```javascript
user?.address?.street; // undefined si user o address es null/undefined
```

17. ¿Qué son los template literals? Strings delimitados por backticks que permiten interpolación de expresiones y strings multilínea.

```javascript
const name = 'John';
const greeting = `Hello, ${name}!`;
```

18. ¿Qué es destructuring? Sintaxis que permite extraer valores de arrays o propiedades de objetos en variables distintas.

```javascript
const [a, b] = [1, 2];
const {name, age} = {name: 'John', age: 30};
```

19. ¿Qué son los spread y rest operators?

- **Spread (`...`)**: Expande elementos de un iterable
- **Rest (`...`)**: Agrupa elementos restantes en un array

```javascript
```

```javascript
const arr = [1, 2, 3];
const newArr = [...arr, 4]; // spread
const [first, ...rest] = arr; // rest
```

20. ¿Qué es el operador ternario? Operador condicional que evalúa una expresión y devuelve uno de dos valores según el resultado.

```javascript
const result = condition ? valueIfTrue : valueIfFalse;
```

Funciones (20 preguntas)

21. ¿Cuáles son las formas de declarar funciones en JavaScript?

- Function declaration: `function name() {}`
- Function expression: `const name = function() {}`
- Arrow function: `const name = () => {}`
- Constructor: `const name = new Function()`

22. ¿Cuál es la diferencia entre function declaration y function expression?

- **Declaration**: Hoisting completo, disponible antes de su declaración
- **Expression**: Solo se eleva la variable, no la función

23. ¿Qué son las arrow functions y cómo difieren de las funciones regulares? Funciones con sintaxis más concisa. Diferencias: no tienen `this` propio, no tienen `arguments`, no pueden ser constructores.

```javascript
// Regular function
function regular() {
  console.log(this); // contexto dinámico
}

// Arrow function
const arrow = () => {
  console.log(this); // contexto léxico
};
```

24. ¿Qué es el `this` keyword? Referencia al contexto de ejecución actual. Su valor depende de cómo se invoca la función: método de objeto, function call, arrow function, etc.

25. ¿Cómo funciona el binding de `this`? Se puede controlar con:

- `call()`: invoca con `this` específico y argumentos individuales
- `apply()`: igual que call pero argumentos en array
- `bind()`: retorna nueva función con `this` fijo

```javascript
const obj = {name: 'John'};
function greet() { console.log(this.name); }

greet.call(obj);   // 'John'
greet.apply(obj);  // 'John'
const bound = greet.bind(obj); // función con this fijo
```

26. ¿Qué son las closures? Una closure es la combinación de una función y el entorno léxico en el que fue declarada, permitiendo acceso a variables del scope externo.

```javascript
```

```
function outer(x) {
  return function inner(y) {
    return x + y; // accede a 'x' del scope externo
  };
}
const add5 = outer(5);
add5(3); // 8
```

27. ¿Para qué sirven las closures?

- Encapsulación y privacidad
- Factory functions
- Module pattern
- Callbacks que mantienen estado
- Event handlers con contexto

28. ¿Qué es el currying? Técnica que transforma una función con múltiples parámetros en una secuencia de funciones que toman un parámetro cada una.

```javascript
// Normal function
function add(a, b, c) {
  return a + b + c;
}

// Curried function
const curriedAdd = (a) => (b) => (c) => a + b + c;
curriedAdd(1)(2)(3); // 6
```

29. ¿Qué son las Higher-Order Functions? Funciones que reciben otras funciones como argumentos o retornan funciones. Ejemplos: `map`, `filter`, `reduce`.

30. ¿Qué es el object `arguments`? Objeto array-like disponible en funciones regulares que contiene todos los argumentos pasados. No disponible en arrow functions.

```javascript
function example() {
  console.log(arguments.length);
  console.log(Array.from(arguments));
}
```

31. ¿Qué son los parámetros rest? Sintaxis que permite representar un número indefinido de argumentos como un array.

```javascript
function sum(...numbers) {
  return numbers.reduce((a, b) => a + b, 0);
}
```

32. ¿Qué son los parámetros por defecto? Valores asignados a parámetros cuando no se proporciona argumento o es `undefined`.

```javascript
function greet(name = 'World') {
  return `Hello, ${name}!`;
}
```

33. ¿Qué es la recursión? Técnica donde una función se llama a sí misma. Útil para estructuras de datos anidadas, algoritmos divide y vencerás.

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

34. ¿Qué es memoization? Técnica de optimización que almacena resultados de funciones costosas para evitar recálculos.

```javascript
function memoize(fn) {
  const cache = {};
  return function(...args) {
    const key = JSON.stringify(args);
    if (key in cache) return cache[key];
    return cache[key] = fn.apply(this, args);
  };
}
```

35. ¿Qué son las funciones puras? Funciones que siempre devuelven el mismo resultado para los mismos argumentos y no tienen efectos secundarios.

36. ¿Qué es el concepto de "first-class functions"? En JavaScript, las funciones son ciudadanos de primera clase: pueden ser asignadas a variables, pasadas como argumentos, retornadas de otras funciones.

37. ¿Qué es una IIFE? Immediately Invoked Function Expression. Función que se ejecuta inmediatamente tras su definición.

```javascript
(function() {
  console.log('IIFE executed');
})();

// Arrow function IIFE
(() => {
  console.log('Arrow IIFE');
})();
```

38. ¿Cuándo usarías una IIFE?

- Crear scope privado
- Evitar contaminar el namespace global
- Módulos (antes de ES6 modules)
- Inicialización única

39. ¿Qué es function hoisting? Las function declarations se elevan completamente (declaración e implementación), mientras que las function expressions solo elevan la variable.

40. ¿Cómo implementarías un debounce function?

```javascript
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => func.apply(this, args), delay);
  };
}
```

Objetos y Prototipos (15 preguntas)

41. ¿Cómo se crean objetos en JavaScript?

- Object literal: `{}`
- Constructor function: `new Object()`
- `Object.create()`
- Factory functions
- ES6 classes

42. ¿Qué es el prototype en JavaScript? Mecanismo por el cual los objetos heredan propiedades y métodos de otros objetos. Cada función tiene una propiedad `prototype`.

43. ¿Cuál es la diferencia entre `__proto__` y `prototype`?

- `__proto__`: Propiedad de instancias que apunta al prototype del constructor
- `prototype`: Propiedad de funciones constructoras que define el prototipo para instancias

44. ¿Cómo funciona la herencia prototípica? Los objetos heredan directamente de otros objetos a través del prototype chain. Si una propiedad no existe en el objeto, se busca en su prototipo.

```javascript
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  console.log(`${this.name} makes a sound`);
};

function Dog(name) {
  Animal.call(this, name);
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
```

45. ¿Qué hace `Object.create()`? Crea un nuevo objeto usando un objeto existente como prototype del nuevo objeto.

```javascript
const animal = {
  speak() {
    console.log(`${this.name} speaks`);
  }
};

const dog = Object.create(animal);
dog.name = 'Buddy';
dog.speak(); // 'Buddy speaks'
```

46. ¿Cuáles son los métodos útiles de Object?

- `Object.keys()`: array de propiedades enumerables
- `Object.values()`: array de valores
- `Object.entries()`: array de pares [key, value]
- `Object.assign()`: copia propiedades
- `Object.freeze()`: hace inmutable
- `Object.seal()`: previene agregar/eliminar propiedades

47. ¿Cuál es la diferencia entre `Object.freeze()` y `Object.seal()`?

- `freeze()`: Inmutable completo (no se puede modificar, agregar o eliminar)
- `seal()`: Solo se pueden modificar propiedades existentes

48. ¿Qué es property descriptor? Objeto que describe las características de una propiedad: `value`, `writable`, `enumerable`, `configurable`.

```javascript
Object.defineProperty(obj, 'prop', {
  value: 42,
  writable: false,
  enumerable: true,
  configurable: false
});
```

49. ¿Cómo verificas si una propiedad existe en un objeto?

- `in` operator: incluye prototype chain
- `hasOwnProperty()`: solo propiedades propias
- `Object.hasOwn()`: versión moderna de hasOwnProperty

50. ¿Qué son los getters y setters? Métodos que permiten definir el comportamiento de acceso y asignación a propiedades.

```javascript
const obj = {
  _value: 0,
  get value() {
    return this._value;
  },
  set value(val) {
    this._value = val;
  }
};
```

51. ¿Cómo iteras sobre las propiedades de un objeto?

- `for...in`: propiedades enumerables (incluye prototype)
- `Object.keys()`: solo propiedades propias
- `Object.getOwnPropertyNames()`: incluye no enumerables

52. ¿Qué es el `constructor` property? Propiedad que referencia la función que creó la instancia del objeto.

53. ¿Cómo clonas objetos en JavaScript?

- Shallow copy: `Object.assign()`, spread operator
- Deep copy: `JSON.parse(JSON.stringify())` (limitado), libraries como Lodash

54. ¿Qué son las computed property names? Sintaxis ES6 que permite usar expresiones como nombres de propiedades en object literals.

```javascript
const prop = 'dynamicKey';
const obj = {
  [prop]: 'value',
  [`${prop}2`]: 'value2'
};
```

55. ¿Cómo funciona el operador `new`?

1. Crea un objeto vacío
2. Asigna el prototype del constructor al objeto
3. Ejecuta el constructor con `this` apuntando al objeto
4. Retorna el objeto (o el valor retornado por el constructor si es objeto)

Arrays (15 preguntas)

56. ¿Cuáles son los métodos principales de Array?

- **Mutating**: `push()`, `pop()`, `shift()`, `unshift()`, `splice()`, `sort()`, `reverse()`
- **Non-mutating**: `slice()`, `concat()`, `map()`, `filter()`, `reduce()`

57. ¿Cuál es la diferencia entre `slice()` y `splice()`?

- `slice()`: No muta, extrae porción del array
- `splice()`: Muta, remueve/agrega elementos

58. ¿Cómo funciona `map()`? Crea nuevo array con los resultados de aplicar una función a cada elemento.

```javascript
const numbers = [1, 2, 3];
const doubled = numbers.map(x => x * 2); // [2, 4, 6]
```

59. ¿Cómo funciona `filter()`? Crea nuevo array con elementos que pasan la prueba de la función callback.

```javascript
const numbers = [1, 2, 3, 4];
const evens = numbers.filter(x => x % 2 === 0); // [2, 4]
```

60. ¿Cómo funciona `reduce()`? Aplica función reductora a cada elemento del array, resultando en un valor único.

```javascript
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) => acc + curr, 0); // 10
```

61. ¿Cuál es la diferencia entre `forEach()` y `map()`?

- `forEach()`: Ejecuta función para cada elemento, no retorna nada
- `map()`: Transforma elementos y retorna nuevo array

62. ¿Cómo encuentras elementos en un array?

- `find()`: primer elemento que cumple condición
- `findIndex()`: índice del primer elemento que cumple condición
- `includes()`: si contiene el valor
- `indexOf()`: índice de la primera ocurrencia

63. ¿Cómo verificas si algo es un array?

- `Array.isArray()`: método recomendado
- `instanceof Array`: puede fallar con frames
- `Object.prototype.toString.call()`: más robusto

64. ¿Qué son los array-like objects? Objetos que tienen índices numéricos y propiedad `length` pero no son arrays. Ejemplo: `arguments`, NodeLists.

```javascript
```

```javascript
// Convertir a array
Array.from(arrayLike);
Array.prototype.slice.call(arrayLike);
[...arrayLike];
```

65. ¿Cómo ordenas arrays? `sort()` convierte elementos a strings por defecto. Para orden numérico:

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

66. ¿Cómo eliminas duplicados de un array?

```javascript
// Con Set
const unique = [...new Set(array)];

// Con filter
const unique = array.filter((item, index) => array.indexOf(item) === index);

// Con reduce
const unique = array.reduce((acc, curr) =>
  acc.includes(curr) ? acc : [...acc, curr], []);
```

67. ¿Cómo aplanas arrays anidados?

```javascript
// Un nivel
[1, [2, 3]].flat(); // [1, 2, 3]

// Múltiples niveles
[1, [2, [3, 4]]].flat(2); // [1, 2, 3, 4]

// Todos los niveles
array.flat(Infinity);
```

68. ¿Qué es `flatMap()`? Combina `map()` y `flat()` en una operación.

```javascript
const arr = [1, 2, 3];
arr.flatMap(x => [x, x * 2]); // [1, 2, 2, 4, 3, 6]
```

69. ¿Cómo verificas si todos/algunos elementos cumplen una condición?

- `every()`: todos los elementos deben cumplir
- `some()`: al menos uno debe cumplir

70. ¿Cómo creas arrays con valores específicos?

```javascript
// Array vacío con longitud
new Array(5); // [empty × 5]

// Array con valores
Array(5).fill(0); // [0, 0, 0, 0, 0]

// Secuencia de números
Array.from({length: 5}, (_, i) => i); // [0, 1, 2, 3, 4]
```

Asincronía (15 preguntas)

71. ¿Qué es el Event Loop? Mecanismo que permite a JavaScript ejecutar código asíncrono en un entorno single-threaded, manejando el call stack, callback queue y microtask queue.

72. ¿Cuál es la diferencia entre Call Stack, Callback Queue y Microtask Queue?

- **Call Stack**: Pila de ejecución síncrona
- **Callback Queue**: Cola de callbacks (setTimeout, eventos)
- **Microtask Queue**: Cola de alta prioridad (Promises, queueMicrotask)

73. ¿Qué son las Promises? Objetos que representan la eventual finalización o falla de una operación asíncrona.

```javascript
const promise = new Promise((resolve, reject) => {
  // operación asíncrona
  if (success) resolve(value);
  else reject(error);
});

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

74. ¿Cuáles son los estados de una Promise?

- **Pending**: Estado inicial, ni cumplida ni rechazada
- **Fulfilled**: Operación completada exitosamente
- **Rejected**: Operación falló

75. ¿Qué es Promise chaining? Técnica para ejecutar múltiples operaciones asíncronas en secuencia usando `.then()`.

```javascript
fetch('/api/user')
  .then(response => response.json())
  .then(user => fetch(`/api/posts/${user.id}`))
  .then(response => response.json())
  .then(posts => console.log(posts));
```

76. ¿Qué son async/await? Sintaxis que hace que el código asíncrono se vea y comporte más como código síncrono.

```javascript
async function fetchUser() {
  try {
    const response = await fetch('/api/user');
    const user = await response.json();
    return user;
  } catch (error) {
    console.error('Error:', error);
  }
}
```

77. ¿Cuál es la diferencia entre Promise y async/await?

- **Promise**: Basado en callbacks (.then/.catch)
- **async/await**: Sintaxis más limpia, permite try/catch, mejor debugging

78. ¿Cómo manejas múltiples Promises?

- `Promise.all()`: Espera que todas se resuelvan (falla si una falla)

- `Promise.allSettled()`: Espera que todas se completen (exitosa o con error)
- `Promise.race()`: Resuelve/rechaza con la primera que se complete
- `Promise.any()`: Resuelve con la primera exitosa

79. ¿Qué es callback hell? Patrón donde callbacks anidados crean código difícil de leer y mantener. Se soluciona con Promises o async/await.

```javascript
// Callback hell
getData(function(a) {
  getMoreData(a, function(b) {
    getEvenMoreData(b, function(c) {
      // ...
    });
  });
});

// Con Promises
getData()
  .then(a => getMoreData(a))
  .then(b => getEvenMoreData(b))
  .then(c => /* ... */);
```

80. ¿Qué es setTimeout y cómo funciona? Función que ejecuta código después de un delay mínimo. No garantiza ejecución exacta debido al Event Loop.

```javascript
setTimeout(() => {
  console.log('Executed after 1000ms');
}, 1000);
```

81. ¿Cuál es la diferencia entre setTimeout y setInterval?

- `setTimeout`: Ejecuta una vez después del delay
- `setInterval`: Ejecuta repetidamente cada intervalo

82. ¿Qué son los microtasks vs macrotasks?

- **Microtasks**: Promises, queueMicrotask (alta prioridad)
- **Macrotasks**: setTimeout, setInterval, eventos (baja prioridad)

83. ¿Cómo cancelarías una Promise? Las Promises no son cancelables nativamente. Opciones:

- AbortController con fetch
- Wrapper con flag de cancelación
- Libraries como p-cancelable

84. ¿Qué es el patrón de error-first callback? Convención donde el primer parámetro del callback es el error.

```javascript
fs.readFile('file.txt', (error, data) => {
  if (error) {
    console.error(error);
    return;
  }
  console.log(data);
});
```

85. ¿Cómo convertirías callback-based functions a Promises?

```javascript
```

```javascript
// Manual
function promisify(fn) {
  return function(...args) {
    return new Promise((resolve, reject) => {
      fn(...args, (error, result) => {
        if (error) reject(error);
        else resolve(result);
      });
    });
  };
}

// Node.js built-in
const { promisify } = require('util');
const readFile = promisify(fs.readFile);
```

ES6+ Features (15 preguntas)

86. ¿Qué son los módulos ES6? Sistema nativo de módulos que permite importar y exportar código entre archivos.

```javascript
// export
export const name = 'John';
export default function greet() {}

// import
import greet, { name } from './module.js';
import * as utils from './utils.js';
```

87. ¿Cuál es la diferencia entre named exports y default export?

- **Named**: Múltiples exports por módulo, deben importarse con el mismo nombre
- **Default**: Un export por módulo, puede importarse con cualquier nombre

88. ¿Qué son los Symbols? Tipo primitivo único e inmutable, útil para propiedades privadas y evitar colisiones de nombres.

```javascript
const sym = Symbol('description');
const obj = {
  [sym]: 'value'
};

// Well-known symbols
Symbol.iterator, Symbol.hasInstance, Symbol.toPrimitive
```

89. ¿Qué son los iteradores e iterables?

- **Iterable**: Objeto que implementa Symbol.iterator
- **Iterator**: Objeto con método `next()` que retorna `{value, done}`

```javascript
```

```javascript
const iterable = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
  }
};

for (const value of iterable) {
  console.log(value); // 1, 2, 3
}
```

90. ¿Qué son los generators? Funciones que pueden pausar y reanudar su ejecución, retornando múltiples valores.

```javascript
function* fibonacci() {
  let a = 0, b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

const fib = fibonacci();
console.log(fib.next().value); // 0
console.log(fib.next().value); // 1
```

91. ¿Qué son los Proxy objects? Permiten interceptar y personalizar operaciones en objetos (get, set, has, etc.).

```javascript
const target = {};
const proxy = new Proxy(target, {
  get(obj, prop) {
    console.log(`Getting ${prop}`);
    return obj[prop];
  },
  set(obj, prop, value) {
    console.log(`Setting ${prop} to ${value}`);
    obj[prop] = value;
    return true;
  }
});
```

92. ¿Qué son los WeakMap y WeakSet? Colecciones donde las referencias a objetos son "débiles", permitiendo garbage collection.

```javascript
const wm = new WeakMap();
const obj = {};
wm.set(obj, 'value');
// obj puede ser garbage collected si no hay otras referencias

const ws = new WeakSet();
ws.add(obj);
```

93. ¿Qué son Map y Set?

- **Map**: Colección de pares key-value donde keys pueden ser cualquier tipo
- **Set**: Colección de valores únicos

```javascript
const map = new Map();
map.set('key', 'value');
map.set(1, 'number key');

const set = new Set([1, 2, 3, 2]); // {1, 2, 3}
```

94. ¿Qué son las tagged template literals? Función que procesa template literals, recibiendo strings y valores por separado.

```javascript
function tag(strings, ...values) {
  console.log(strings); // ['Hello ', ' world']
  console.log(values);  // ['beautiful']
  return strings[0] + values[0] + strings[1];
}

const name = 'beautiful';
tag`Hello ${name} world`; // 'Hello beautiful world'
```

95. ¿Qué es la for...of loop? Itera sobre objetos iterables (arrays, strings, Maps, Sets).

```javascript
for (const value of [1, 2, 3]) {
  console.log(value); // 1, 2, 3
}

for (const char of 'hello') {
  console.log(char); // h, e, l, l, o
}
```

96. ¿Cuál es la diferencia entre for...in y for...of?

- `for...in`: Itera sobre propiedades enumerables (keys)
- `for...of`: Itera sobre valores de objetos iterables

97. ¿Qué son las clases ES6? Syntactic sugar sobre funciones constructoras y prototipos, proporcionando una sintaxis más limpia para OOP.

```javascript
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound`);
  }

  static species() {
    return 'Unknown';
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks`);
  }
}
```

98. ¿Qué son los métodos estáticos en clases? Métodos que pertenecen a la clase en sí, no a las instancias. Se llaman en la clase directamente.

99. ¿Qué son los private fields en clases? Campos que solo son accesibles dentro de la clase (sintaxis con `#`).

```javascript
class MyClass {
  #privateField = 'private';

  #privateMethod() {
    return this.#privateField;
  }

  publicMethod() {
    return this.#privateMethod();
  }
}
```

100. ¿Qué son los BigInt? Tipo de datos para representar enteros más grandes que `Number.MAX_SAFE_INTEGER`.

```javascript
const big = BigInt(9007199254740991);
const big2 = 9007199254740991n; // literal syntax
console.log(big + 1n); // 9007199254740992n
```

REACT - 100 PREGUNTAS

Conceptos Fundamentales (20 preguntas)

1. ¿Qué es React? React es una biblioteca de JavaScript para construir interfaces de usuario, especialmente aplicaciones de una sola página (SPAs). Se basa en componentes reutilizables y el concepto de Virtual DOM.

2. ¿Qué es JSX? JSX (JavaScript XML) es una extensión de sintaxis que permite escribir HTML dentro de JavaScript. Se transpila a llamadas de `React.createElement()`.

```javascript
const element = <h1>Hello, World!</h1>;
// Se convierte en:
const element = React.createElement('h1', null, 'Hello, World!');
```

3. ¿Qué es el Virtual DOM? Representación en memoria del DOM real. React usa el Virtual DOM para:

- Comparar el estado anterior con el nuevo (diffing)
- Actualizar solo las partes que cambiaron (reconciliation)
- Mejorar el rendimiento

4. ¿Cuál es la diferencia entre elementos y componentes?

- **Elemento**: Objeto plano que describe lo que quieres ver en pantalla
- **Componente**: Función o clase que acepta props y retorna elementos

5. ¿Qué son los componentes funcionales vs componentes de clase?

- **Funcionales**: Funciones que reciben props y retornan JSX (preferidos con hooks)
- **Clase**: Clases ES6 que extienden React.Component (legacy approach)

```javascript
```