

# JS functions interview topics

I want to learn functions in JS. Can you provide some points or topics that I should learn from an Interview perspective

Absolutely! Here's a focused list of **JavaScript functions topics** that are important for interviews:

---

## 1. Function Basics

- Defining functions (function declaration vs function expression)
- Calling functions
- Parameters and arguments
- Return statement

## 2. Arrow Functions

- Syntax differences from regular functions
- `this` binding in arrow functions vs normal functions

## 3. Function Scope and Closures

- Function scope vs block scope (var vs let/const)
- What are closures and how do they work?
- Practical uses of closures (e.g., data privacy, function factories)

## 4. Higher-Order Functions

- Functions that take other functions as arguments or return functions
- Examples: `map()`, `filter()`, `reduce()`
- Callback functions and asynchronous callbacks

## 5. Immediately Invoked Function Expressions (IIFE)

- Syntax and purpose
- Why use IIFE in JavaScript?

## 6. Function Hoisting

- How function declarations and expressions are hoisted differently

- Temporal Dead Zone (TDZ) with function expressions (especially with `let` and `const` )

## 7. Function Arguments Object and Rest Parameters

- Using the `arguments` object
- Using rest parameters ( `...args` )
- Differences and use cases

## 8. Default Parameters

- Setting default values for function parameters

## 9. Function Methods: `call()`, `apply()`, and `bind()`

- How these methods work
- Differences between them
- Common use cases (changing `this` context)

## 10. Constructor Functions and the `new` Keyword

- How functions can be used as constructors
- What happens when a function is called with `new`

## 11. Recursion

- Understanding recursive functions
- Common examples (e.g., factorial, Fibonacci)

## 12. Async Functions and Promises (bonus, often combined with functions)

- `async` / `await`
- Returning promises from functions

---

If you want, I can also provide code examples or help with interview-style questions on any of these topics. Just say which one!

### 1. Function Basics

Defining functions (function declaration vs function expression)

Calling functions

Parameters and arguments

Return statement

## 2. Arrow Functions

Syntax differences from regular functions

this binding in arrow functions vs normal functions

## 3. Function Scope and Closures

Function scope vs block scope (var vs let/const)

What are closures and how do they work?

Practical uses of closures (e.g., data privacy, function factories)

## 4. Higher-Order Functions

Functions that take other functions as arguments or return functions

Examples: map(), filter(), reduce()

Callback functions and asynchronous callbacks

## 5. Immediately Invoked Function Expressions (IIFE)

Syntax and purpose

Why use IIFE in JavaScript?

Great! Here's a concise explanation and key points for each of these topics with simple examples to help you understand and prepare for interviews:

---

## 1. Function Basics

## Defining Functions

- **Function Declaration:**

```
js

function greet(name) {
  return `Hello, ${name}!`;
}
```

- **Function Expression:**

```
js

const greet = function(name) {
  return `Hello, ${name}!`;
};
```

## Calling Functions

- Simply use the function name with parentheses:

```
js

greet('Alice'); // "Hello, Alice!"
```

## Parameters and Arguments

- **Parameters** are the variables listed in the function definition.
- **Arguments** are the actual values passed to the function.

```
js

function add(a, b) {
  return a + b;
}

add(2, 3); // 5 (2 and 3 are arguments)
```

## Return Statement

- Functions return `undefined` by default if no return statement.
- Return a value to get the result:

```
js

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

```
}  
square(4); // 16
```

## 2. Arrow Functions

### Syntax Differences

```
js  
  
// Regular function  
const sum = function(a, b) {  
  return a + b;  
};  
  
// Arrow function  
const sum = (a, b) => a + b;
```

### this Binding

- Arrow functions **do not have their own** `this`; they inherit `this` from the surrounding context.
- Regular functions have their own `this` depending on how they're called.

Example:

```
js  
  
const obj = {  
  value: 10,  
  regularFunc: function() { return this.value; },  
  arrowFunc: () => this.value  
};  
  
obj.regularFunc(); // 10  
obj.arrowFunc(); // undefined (because `this` refers to the outer scope)
```

## 3. Function Scope and Closures

### Function Scope vs Block Scope

- `var` is function-scoped (ignores blocks like `if / for` )

- `let` and `const` are block-scoped

```
js

if (true) {
  var x = 1;
  let y = 2;
}

console.log(x); // 1
console.log(y); // ReferenceError
```

## Closures

- A function **remembers** the environment where it was created.

```
js

function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  };
}

const counter = outer();
console.log(counter()); // 1
console.log(counter()); // 2
```

## Practical Uses of Closures

- **Data privacy:** Variables in the outer function are private.
- **Function factories:** Return functions configured with specific values.

---

## 4. Higher-Order Functions

### Definition

- Functions that **accept functions as arguments** or **return functions**.

**Examples:** `map()`, `filter()`, `reduce()`

```
js
```

```
const nums = [1, 2, 3];
const doubled = nums.map(n => n * 2); // [2, 4, 6]
const evens = nums.filter(n => n % 2 === 0); // [2]
const sum = nums.reduce((acc, n) => acc + n, 0); // 6
```

## Callback Functions & Asynchronous Callbacks

- Callbacks run after some operation completes (sync or async).

js

```
setTimeout(() => {
  console.log("Executed after 1 second");
}, 1000);
```

---

## 5. Immediately Invoked Function Expressions (IIFE)

### Syntax and Purpose

- A function that runs as soon as it's defined:

js

```
(function() {
  console.log('IIFE runs immediately!');
})();
```

### Why Use IIFE?

- Create a private scope to avoid polluting global scope.
- Useful in module patterns and to isolate variables.

---

Want me to help you with code challenges or quizzes on any of these? Or maybe dive deeper into a specific topic?

## 6. Function Hoisting

How function declarations and expressions are hoisted differently

Temporal Dead Zone (TDZ) with function expressions (especially with `let` and `const`)

## 7. Function Arguments Object and Rest Parameters

Using the arguments object

Using rest parameters (...args)

Differences and use cases

## 8. Default Parameters

Setting default values for function parameters

Nice! Here's a clear breakdown of these topics with explanations and examples:

---

## 6. Function Hoisting

### What is Hoisting?

- JavaScript **hoists** declarations to the top of their scope before execution.
  - **Function declarations** are hoisted **with their definitions**.
  - **Function expressions** (especially those assigned to `var`, `let`, or `const`) behave differently.
- 

### Function Declarations Hoisted

```
js

console.log(foo()); // Works fine, prints "Hello!"

function foo() {
  return "Hello!";
}
```

- Function declarations are hoisted fully — you can call them before they appear in code.
-



## Function Expressions Not Hoisted (or TDZ with let/const)

```
js

console.log(bar()); // Error: bar is not a function

var bar = function() {
  return "Hi!";
};
```

- Only the variable `bar` is hoisted (initialized as `undefined`), not the function value.
- Calling before assignment throws `TypeError`.

With `let` or `const`:

```
js

console.log(baz()); // ReferenceError: Cannot access 'baz' before initialization

const baz = () => "Hey!";
```

- `let` and `const` variables are hoisted but in the **Temporal Dead Zone (TDZ)** until initialization.
- You cannot access them before the line they're declared.

---

## 7. Function Arguments Object and Rest Parameters

---

### `arguments` Object

- Available in **non-arrow functions**.
- Array-like object containing all arguments passed.
- Does **not** work in arrow functions.

Example:

```
js

function sum() {
  let total = 0;
  for (let i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }
}
```

```
}  
return total;  
}  
  
sum(1, 2, 3); // 6
```

---

## Rest Parameters ( `...args` )

- Collects all remaining arguments into a real **array**.
- Works in arrow functions and regular functions.

Example:

```
js  
  
function sum(...args) {  
  return args.reduce((acc, val) => acc + val, 0);  
}  
  
sum(1, 2, 3, 4); // 10
```

---

## Differences and Use Cases

| Feature                   | <code>arguments</code>        | Rest Parameters        |
|---------------------------|-------------------------------|------------------------|
| Array or Array-like       | Array-like (no array methods) | Real array             |
| Works in arrow functions? | No                            | Yes                    |
| Explicit naming           | No (it's a built-in object)   | Yes (named parameters) |

---

## 8. Default Parameters

- Allows setting default values for function parameters if no argument or `undefined` is passed.

Example:

js

```
function greet(name = "Guest") {  
  return `Hello, ${name}!`;  
}  
  
greet();      // "Hello, Guest!"  
greet("Alice"); // "Hello, Alice!"
```

- You can use expressions or other parameters as default values:

js

```
function multiply(a, b = a) {  
  return a * b;  
}  
  
multiply(5);    // 25 (b defaults to a)  
multiply(5, 2); // 10
```

---

Would you like me to generate practice problems or explain any of these concepts with more detailed examples?

generate practice problems

Absolutely! Here are practice problems for each of the topics you asked about — great for interview prep.

---

## 6. Function Hoisting

### Problem 1:

What will the following code output? Explain why.

js

```
console.log(foo());  
console.log(bar());
```

```
function foo() {  
  return "Function Declaration";  
}  
  
var bar = function() {  
  return "Function Expression";  
};
```

---

#### Problem 2:

What happens if you replace `var bar` with `let bar` in the above code? Why?

---

## 7. Function Arguments Object and Rest Parameters

#### Problem 3:

Write a function `multiplyAll` that takes any number of numeric arguments and returns their product. Use the `arguments` object.

---

#### Problem 4:

Rewrite the above `multiplyAll` function using rest parameters instead of `arguments`.

---

#### Problem 5:

Explain the difference in behavior when using arrow functions and `arguments`.

---

## 8. Default Parameters

#### Problem 6:

Write a function `createUser` that takes two parameters: `name` (default `"Anonymous"`) and `age` (default `18`). It should return an object with these properties.

---

### Problem 7:

What will this code output? Explain why.

```
js

function countdown(start = 5) {
  console.log(start);
  if (start > 0) countdown(start - 1);
}

countdown();
countdown(3);
```

If you want, I can provide solutions or hints. Just ask! Would you like to try these first?

#### 9. Function Methods: call(), apply(), and bind()

How these methods work

Differences between them

Common use cases (changing this context)

#### 10. Constructor Functions and the new Keyword

How functions can be used as constructors

What happens when a function is called with new

#### 11. Recursion

Understanding recursive functions

Common examples (e.g., factorial, Fibonacci)

#### 12. Async Functions and Promises (bonus, often combined with functions)

async / await

Got it! Here's a solid breakdown with explanations and examples for these topics — interview style:

---

### 9. Function Methods: `call()`, `apply()`, and `bind()`

#### How They Work

- All three methods let you **explicitly set the** `this` **context** for a function.
- 

#### `call()`

- Calls the function immediately.
- Takes arguments **one by one**.

js

```
function greet(greeting, punctuation) {  
  console.log(greeting + ', ' + this.name + punctuation);  
}  
  
const person = { name: 'Alice' };  
greet.call(person, 'Hello', '!'); // Hello, Alice!
```

#### `apply()`

- Calls the function immediately.
- Takes arguments **as an array**.

js

```
greet.apply(person, ['Hi', '...']); // Hi, Alice...
```

#### `bind()`

- Returns a **new function** with `this` bound to the provided object.
- Does **not call** immediately.

js

```
const greetAlice = greet.bind(person);  
greetAlice('Hey', '?'); // Hey, Alice?
```

---

## Common Use Cases

- Borrowing methods from other objects.
- Setting context in event handlers.
- Partial function application with `bind`.

---

## 10. Constructor Functions and the `new` Keyword

### Using Functions as Constructors

- When called with `new`, a function:
  - Creates a new empty object.
  - Sets `this` to that object.
  - Returns `this` implicitly (unless the function returns an object explicitly).

Example:

js

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
}  
  
const alice = new Person('Alice', 25);  
console.log(alice.name); // Alice
```

---

### What Happens Under the Hood

- `new Person('Alice', 25)` is like:
  1. `const obj = {};`
  2. `obj.__proto__ = Person.prototype;`
  3. `Person.call(obj, 'Alice', 25);`
  4. Return `obj`

---

## 11. Recursion

### Understanding Recursive Functions

- A function that calls itself until it reaches a **base case**.

---

#### Example: Factorial

```
js

function factorial(n) {
  if (n <= 1) return 1; // base case
  return n * factorial(n - 1);
}

factorial(5); // 120
```

---

#### Example: Fibonacci

```
js

function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

fibonacci(6); // 8
```

---

## 12. Async Functions and Promises

async / await

- `async` functions always return a **promise**.
- `await` pauses the execution until the promise resolves.

Example:

```
js
```



```
async function fetchData() {  
  const response = await fetch('https://api.example.com/data');  
  const data = await response.json();  
  return data;  
}  
  
fetchData().then(data => console.log(data));
```

---

## Returning Promises from Functions

```
js  
  
function delay(ms) {  
  return new Promise(resolve => setTimeout(resolve, ms));  
}  
  
delay(1000).then(() => console.log('1 second passed'));
```

---

Would you like me to create **practice problems** or **code challenges** for these too? Or maybe provide detailed explanations on any part?

yes, Would you like me to create practice problems or code challenges for these too?

Awesome! Here are some practice problems and challenges for these topics:

---

## 9. Function Methods: `call()`, `apply()`, and `bind()`

### Problem 1:

Given this code:

```
js  
  
const person = { name: 'John' };
```

```
function introduce(city, country) {  
  console.log(`${this.name} lives in ${city}, ${country}.`);  
}
```

- Use `call()` to print: `"John lives in New York, USA."`
  - Use `apply()` to do the same.
  - Use `bind()` to create a new function bound to `person` and call it with `"London", "UK"`.
- 

## 10. Constructor Functions and the `new` Keyword

### Problem 2:

Write a constructor function `car` that takes `make` and `model` and stores them on the object. Add a method `getDetails()` on the prototype that returns a string with make and model.

Create a new instance with `new` and call `getDetails()`.

---

## 11. Recursion

### Problem 3:

Write a recursive function `sumTo(n)` that returns the sum of numbers from 1 to `n`.

---

### Problem 4:

Write a recursive function to compute the `nth` Fibonacci number.

---

## 12. Async Functions and Promises

### Problem 5:

Write a function `wait(ms)` that returns a promise resolving after `ms` milliseconds. Use `async/await` to call it and print `"Done waiting"` after 2 seconds.

---

### Problem 6:

Convert this promise-based function to use `async/await`:

---

js

```
function fetchData() {  
  return fetch('https://jsonplaceholder.typicode.com/todos/1')  
    .then(response => response.json())  
    .then(data => console.log(data));  
}
```

---

Want me to provide solutions or hints? Or do you want to try them out first?