Complete JavaScript Foundation for Frameworks

1. Variables and Scoping

Variable Declarations

javascript

```javascript
var oldStyle = "function-scoped, can be redeclared";
let modernBlock = "block-scoped, can be reassigned";
const constant = "block-scoped, cannot be reassigned";
```

Scoping Rules

- **Global Scope**: Variables declared outside functions
- **Function Scope**: Variables declared inside functions (var)
- **Block Scope**: Variables declared inside blocks {} (let, const)
- **Temporal Dead Zone**: let/const cannot be used before declaration

javascript

```javascript
function scopeExample() {
    if (true) {
        var functionScoped = "I'm available throughout the function";
        let blockScoped = "I'm only available in this block";
        const alsoBlockScoped = "Me too";
    }
    console.log(functionScoped); // Works
    console.log(blockScoped); // ReferenceError
}
```

2. Data Types and Type Coercion

Primitive Types

javascript

```javascript
let string = "text";
let number = 42;
let boolean = true;
let undefined = undefined;
let nullValue = null;
let symbol = Symbol("unique");
let bigint = 123n;
```

Reference Types

```javascript
let object = { name: "John", age: 30 };
let array = [1, 2, 3, 4];
let function = function() { return "I'm a function"; };
```

Type Coercion

```javascript
// Implicit coercion
"5" + 3; // "53" (string concatenation)
"5" - 3; // 2 (numeric subtraction)
true + 1; // 2
false == 0; // true

// Explicit conversion
Number("123"); // 123
String(123); // "123"
Boolean(0); // false
```

3. Functions

Function Declarations

```javascript
function regularFunction(param1, param2) {
    return param1 + param2;
}
```

Function Expressions

```javascript
const functionExpression = function(param) {
    return param * 2;
};
```

Arrow Functions

javascript

```javascript
const arrowFunction = (param) => param * 2;
const multiLine = (a, b) => {
    const result = a + b;
    return result;
};
```

Key Differences

- Arrow functions don't have their own `this`
- Arrow functions can't be used as constructors
- Arrow functions don't have `arguments` object

4. Control Flow

Conditional Statements

javascript

```javascript
if (condition) {
    // code
} else if (anotherCondition) {
    // code
} else {
    // code
}

// Ternary operator
const result = condition ? "true value" : "false value";

// Switch statement
switch (value) {
    case 'option1':
        // code
        break;
    case 'option2':
        // code
        break;
    default:
        // code
}
```

Loops

```javascript
// For loop
for (let i = 0; i < array.length; i++) {
    console.log(array[i]);
}

// For...of (values)
for (const item of array) {
    console.log(item);
}

// For...in (keys/indices)
for (const key in object) {
    console.log(key, object[key]);
}

// While loop
while (condition) {
    // code
}
```

5. Error Handling

Try-Catch-Finally

```javascript
try {
    // Code that might throw an error
    const result = riskyOperation();
} catch (error) {
    // Handle the error
    console.error("An error occurred:", error.message);
} finally {
    // Always runs
    cleanup();
}
```

Throwing Custom Errors

javascript

```javascript
function validateAge(age) {
    if (age < 0) {
        throw new Error("Age cannot be negative");
    }
    if (typeof age !== 'number') {
        throw new TypeError("Age must be a number");
    }
}
```

6. ES6+ Features

Destructuring

javascript

```javascript
// Array destructuring
const [first, second, ...rest] = [1, 2, 3, 4, 5];

// Object destructuring
const { name, age, city = "Unknown" } = person;

// Function parameter destructuring
function greet({ name, age }) {
    return `Hello ${name}, you are ${age} years old`;
}
```

Template Literals

javascript

```javascript
const name = "John";
const age = 30;
const message = `Hello ${name}, you are ${age} years old.
This is a multi-line string.`;
```

Spread and Rest Operators

```javascript
// Spread operator
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5]; // [1, 2, 3, 4, 5]

const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // { a: 1, b: 2, c: 3 }

// Rest operator
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}
```

7. Promises and Async/Await

Promises

```javascript
const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        if (Math.random() > 0.5) {
            resolve("Success!");
        } else {
            reject(new Error("Failed!"));
        }
    }, 1000);
});

promise
    .then(result => console.log(result))
    .catch(error => console.error(error))
    .finally(() => console.log("Done"));
```

Async/Await

javascript

```javascript
async function fetchData() {
    try {
        const response = await fetch('/api/data');
        const data = await response.json();
        return data;
    } catch (error) {
        console.error("Error fetching data:", error);
        throw error;
    }
}

// Using the async function
fetchData()
    .then(data => console.log(data))
    .catch(error => console.error(error));
```

8. Array Methods

Essential Array Methods

javascript

```javascript
const numbers = [1, 2, 3, 4, 5];

// Map - transform each element
const doubled = numbers.map(num => num * 2); // [2, 4, 6, 8, 10]

// Filter - select elements that meet criteria
const evens = numbers.filter(num => num % 2 === 0); // [2, 4]

// Reduce - combine elements into single value
const sum = numbers.reduce((total, num) => total + num, 0); // 15

// Find - get first element that matches
const found = numbers.find(num => num > 3); // 4

// Some/Every - test conditions
const hasEven = numbers.some(num => num % 2 === 0); // true
const allPositive = numbers.every(num => num > 0); // true

// ForEach - execute function for each element
numbers.forEach(num => console.log(num));
```

Chaining Methods

javascript

```javascript
const result = numbers
    .filter(num => num > 2)
    .map(num => num * 2)
    .reduce((sum, num) => sum + num, 0);
```

9. Object Methods and Manipulation

Object Creation and Manipulation

javascript

```javascript
const person = {
    name: "John",
    age: 30,
    greet() {
        return `Hello, I'm ${this.name}`;
    }
};

// Property access
person.name; // "John"
person["age"]; // 30

// Dynamic property names
const propName = "email";
person[propName] = "john@example.com";

// Object methods
Object.keys(person); // ["name", "age", "greet", "email"]
Object.values(person); // ["John", 30, function, "john@example.com"]
Object.entries(person); // [["name", "John"], ["age", 30], ...]
```

Object Destructuring in Functions

javascript

```javascript
function updatePerson({ name, age, ...otherProps }) {
    return {
        name: name.toUpperCase(),
        age: age + 1,
        ...otherProps
    };
}
```

10. DOM Manipulation

Selecting Elements

```javascript
const element = document.getElementById('myId');
const elements = document.querySelectorAll('.myClass');
const firstMatch = document.querySelector('div.container');
```

Modifying Elements

```javascript
element.textContent = "New text";
element.innerHTML = "<strong>Bold text</strong>";
element.setAttribute('class', 'new-class');
element.style.color = 'red';
```

Creating and Removing Elements

```javascript
const newDiv = document.createElement('div');
newDiv.textContent = "Hello World";
document.body.appendChild(newDiv);

element.remove(); // Remove element
parent.removeChild(child); // Alternative removal
```

11. Event Handling

Adding Event Listeners

```javascript
button.addEventListener('click', function(event) {
    event.preventDefault(); // Prevent default behavior
    console.log('Button clicked!');
});

// Arrow function version
button.addEventListener('click', (event) => {
    console.log('Clicked at:', event.clientX, event.clientY);
});
```

Event Propagation

javascript

```javascript
// Event bubbling (default)
element.addEventListener('click', handler, false);

// Event capturing
element.addEventListener('click', handler, true);

// Stop propagation
function handler(event) {
    event.stopPropagation(); // Stop bubbling/capturing
}
```

Event Delegation

javascript

```javascript
document.addEventListener('click', function(event) {
    if (event.target.matches('.button-class')) {
        // Handle click on any element with 'button-class'
        console.log('Dynamic button clicked');
    }
});
```

12. Higher-Order Functions and Callbacks

Callback Functions

javascript

```javascript
function processData(data, callback) {
    const result = data.map(item => item * 2);
    callback(result);
}

processData([1, 2, 3], function(result) {
    console.log(result); // [2, 4, 6]
});
```

Higher-Order Functions

javascript

```javascript
function createMultiplier(factor) {
    return function(number) {
        return number * factor;
    };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

13. Closures and Lexical Scoping

Understanding Closures

javascript

```javascript
function outerFunction(x) {
    // This is the outer scope

    function innerFunction(y) {
        // This inner function has access to outer scope
        return x + y;
    }

    return innerFunction;
}

const addFive = outerFunction(5);
console.log(addFive(3)); // 8
```

Practical Closure Example

javascript

```javascript
function createCounter() {
    let count = 0;

    return {
        increment() { return ++count; },
        decrement() { return --count; },
        getCount() { return count; }
    };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.getCount()); // 1
```

14. The `this` Keyword

Different Contexts of `this`

javascript

```javascript
const obj = {
    name: "Object",
    regularMethod: function() {
        console.log(this.name); // "Object"
    },
    arrowMethod: () => {
        console.log(this.name); // undefined (or global context)
    }
};

// Call, Apply, Bind
function greet() {
    console.log(`Hello, ${this.name}`);
}

const person = { name: "John" };

greet.call(person); // "Hello, John"
greet.apply(person); // "Hello, John"
const boundGreet = greet.bind(person);
boundGreet(); // "Hello, John"
```

Event Handler Context

javascript

```javascript
button.addEventListener('click', function() {
    console.log(this); // The button element
});

button.addEventListener('click', () => {
    console.log(this); // Global context (not the button)
});
```

15. Prototypes and Object-Oriented Concepts

Prototype Chain

javascript

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

Person.prototype.greet = function() {
    return `Hello, I'm ${this.name}`;
};

const john = new Person("John");
console.log(john.greet()); // "Hello, I'm John"
```

Classes (ES6+)

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

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

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

const dog = new Dog("Rex");
console.log(dog.speak()); // "Rex barks"
```

16. Module System

ES6 Modules

```javascript
// math.js
export const PI = 3.14159;
export function add(a, b) {
    return a + b;
}
export default function multiply(a, b) {
    return a * b;
}

// main.js
import multiply, { PI, add } from './math.js';
import * as MathUtils from './math.js';

console.log(add(2, 3)); // 5
console.log(multiply(2, 3)); // 6
console.log(MathUtils.PI); // 3.14159
```

CommonJS (Node.js)

javascript

```javascript
// math.js
const PI = 3.14159;
function add(a, b) {
    return a + b;
}

module.exports = { PI, add };

// main.js
const { PI, add } = require('./math');
```

17. Debugging Techniques

Console Methods

javascript

```javascript
console.log("Basic logging");
console.error("Error message");
console.warn("Warning message");
console.table([{name: "John", age: 30}, {name: "Jane", age: 25}]);
console.group("Grouped logs");
console.log("Inside group");
console.groupEnd();
```

Debugging with Breakpoints

javascript

```javascript
function debugExample(data) {
    debugger; // Execution will pause here in dev tools

    const processed = data.map(item => {
        console.log("Processing:", item); // Log each iteration
        return item * 2;
    });

    return processed;
}
```

18. Package Managers (npm/yarn)

Basic npm Commands

bash

```bash
# Initialize a project
npm init

# Install dependencies
npm install package-name
npm install --save-dev package-name  # Development dependency

# Install from package.json
npm install

# Run scripts
npm run script-name
npm start
npm test
```

Package.json Understanding

json

```json
{
  "name": "my-project",
  "version": "1.0.0",
  "scripts": {
    "start": "node index.js",
    "test": "jest",
    "build": "webpack"
  },
  "dependencies": {
    "express": "^4.18.0"
  },
  "devDependencies": {
    "jest": "^28.0.0"
  }
}
```

Framework-Specific Preparation Tips

For React

- Focus heavily on array methods (map, filter, reduce)
- Understand functional programming concepts
- Practice with arrow functions and destructuring
- Learn about immutability principles

For Vue

- Understand object reactivity
- Practice with template syntax concepts
- Learn about event handling patterns
- Understand component communication

For Angular

- Learn TypeScript basics
- Understand dependency injection concepts
- Practice with classes and decorators
- Learn about observables (RxJS basics)

Practice Exercises

1. **Build a TODO app** using vanilla JavaScript with DOM manipulation
2. **Create utility functions** that use array methods and higher-order functions
3. **Practice async operations** with fetch API and promises
4. **Build simple modules** and practice import/export
5. **Experiment with closures** by creating different factory functions
6. **Practice event delegation** with dynamic content
7. **Create object-oriented examples** using both prototypes and classes

Next Steps

Once comfortable with these concepts:

1. Choose your framework and dive into its documentation
2. Build small projects combining vanilla JavaScript with your chosen framework
3. Learn the framework's specific patterns and best practices
4. Understand the framework's state management solutions
5. Practice building increasingly complex applications

Remember: You don't need to master everything before starting with frameworks, but having a solid foundation in these areas will make learning frameworks much smoother and help you debug issues more effectively.