JavaScript - Complete Guide

Summary of Key Concepts

JavaScript Fundamentals

- **Variable Declarations**: Three types - `var`, `let`, and `const`, each with different scoping and hoisting behavior
- **Type Conversion**: JavaScript automatically converts types when using operators; different rules for `+` vs other operators
- **Equality Operators**: `==` performs type conversion, `===` checks both value and type
- **Global Objects**: In browsers, `window` is the global object that contains global variables and functions

Advanced JavaScript Concepts

- **Closures**: Allow functions to access parent scope variables even after parent function execution
- **Arrow Functions vs. Regular Functions**: Differ in `this` binding, with arrow functions inheriting from surrounding scope
- **Working with Objects**: Various methods to create, access, and iterate through object properties
- **The `this` Keyword**: Behavior depends on function type and call context
- **Asynchronous JavaScript**: Use of `setTimeout` and `setInterval` for delayed or repeated execution

1. JavaScript Fundamentals

Variable Declarations and Type Conversion

javascript

Copy

```javascript
// Three types of variable declarations:
var a = 1;    // Function-scoped, can be redeclared
let b = 2;    // Block-scoped, can be reassigned but not redeclared
const c = 3;  // Block-scoped, cannot be reassigned

// Type conversion with + operator
"10" + 5;     // "105" (converts number to string)

// Other operators convert to numbers
"10" * 5;     // 50 (converts string to number)
"10" - 5;     // 5
"10" / 5;     // 2
"10" % 5;     // 0

// Boolean to number conversion
true * 10;    // 10 (true converts to 1)
false * 10;   // 0 (false converts to 0)
```

Scoping and Hoisting

javascript

Copy

```javascript
// var - function scoped
var a = 1;
{
  var a = 2;    // Modifies the outer 'a'
}
console.log(a); // 2

// let - block scoped
let x = 1;
{
  let x = 2;    // Different variable, only exists in this block
}
console.log(x); // 1

// Hoisting differences
console.log(varVariable);    // undefined (hoisted with undefined value)
var varVariable = 10;

// console.log(letVariable); // ReferenceError (in temporal dead zone)
let letVariable = 10;

// Function declarations are completely hoisted
sayHello();                  // Works!
function sayHello() {
  console.log("Hello");
}
```

Global Objects and Window

javascript

Copy

```javascript
// var in global scope creates window property
var a = 10;
console.log(window.a);  // 10

// let and const don't create window properties
let b = 20;
console.log(window.b);  // undefined
```

Equality Operators

javascript Copy

```javascript
// == (loose equality) - converts types
console.log(2 == "2");   // true
console.log(true == 1);  // true

// === (strict equality) - checks value and type
console.log(2 === "2");  // false
console.log(true === 1); // false
```

2. Closures

Closures allow functions to access variables from their parent scope, even after that parent function has completed execution. This enables data encapsulation and the creation of private variables.

Basic Closure Example

javascript Copy

```javascript
function pizza() {
  let slices = 6; // This variable is private to the pizza function

  function eat() {
    slices--; // Accessing the parent variable
    console.log(`Ate a slice! Remaining: ${slices}`);
  }

  return eat; // Return the function that maintains access to slices
}

const pineapplePizza = pizza(); // This returns the eat function
pineapplePizza(); // "Ate a slice! Remaining: 5"
pineapplePizza(); // "Ate a slice! Remaining: 4"

// The slices variable is not directly accessible from outside
// console.log(slices); // ReferenceError: slices is not defined
```

Practical Closure Example: Counter

javascript

Copy

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

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

const counter = createCounter();
counter.increment();
counter.increment();
console.log(counter.getCount()); // 2
counter.decrement();
console.log(counter.getCount()); // 1

// count variable is private and cannot be accessed directly
// console.log(count); // ReferenceError
```

Key points about closures:

- They allow functions to maintain access to their parent scope's variables
- They enable private variables in JavaScript
- Each function call creates its own closure (environment)
- The enclosed variables are preserved between function calls
- Used extensively in module patterns, event handlers, and callback functions

3. Arrow Functions vs. Regular Functions

Arrow functions differ from regular functions in several important ways:

The `this` Keyword Behavior

javascript

Copy

```javascript
// Regular function - 'this' is determined by how the function is called
const obj = {
  name: "Rohit",
  regularFunction: function() {
    console.log(this.name); // 'this' refers to obj
  },

  // Arrow function - 'this' is inherited from the surrounding scope
  arrowFunction: () => {
    console.log(this.name); // 'this' refers to the outer scope (often window)
  }
};

obj.regularFunction(); // Outputs: "Rohit"
obj.arrowFunction(); // Outputs: undefined (if there's no global 'name' variable)
```

Key differences:

1. **Regular functions** create their own `this` context based on how they're called
2. **Arrow functions** don't create their own `this` - they inherit it from the surrounding scope at creation time

When to Use Each Type:

- Use **regular functions** when you need dynamic `this` binding, especially for object methods
- Use **arrow functions** when you want to preserve the `this` value from the surrounding scope

javascript

Copy

```javascript
// Common example where arrow functions are useful
const user = {
  name: "John",
  friends: ["Alex", "Mary"],

  printFriends() {
    // Arrow function preserves 'this' from the outer method
    this.friends.forEach(friend => {
      console.log(`${this.name} is friends with ${friend}`);
    });
  }
};

user.printFriends();
// "John is friends with Alex"
// "John is friends with Mary"
```

Other Differences

Arrow functions also:

- Don't have an `arguments` object
- Can't be used with the `new` operator
- Don't have a `prototype` property
- Can't use `yield` (can't be generator functions)

4. Working with Objects

Basic Object Operations

javascript

Copy

```javascript
// Creating an object
const user = {
  name: "John",
  age: 30,
  "likes birds": true, // Multi-word property name requires quotes
  greeting: function() {
    console.log(`Hello, ${this.name}!`);
  },
  // Shorthand method syntax
  sayHi() {
    console.log("Hi!");
  }
};

// Accessing properties
console.log(user.name); // John
console.log(user["likes birds"]); // true (square bracket notation for multi-word properties)

// Adding or modifying properties
user.isAdmin = true;
user.age = 31;

// Removing properties
delete user.age;
```

Object Methods and Iteration

javascript

Copy

```javascript
// Getting keys, values, and entries
const user = {
  name: "John",
  age: 30,
  isAdmin: true
};

// Get all keys
console.log(Object.keys(user)); // ["name", "age", "isAdmin"]

// Get all values
console.log(Object.values(user)); // ["John", 30, true]

// Get all key-value pairs as arrays
console.log(Object.entries(user)); // [["name", "John"], ["age", 30], ["isAdmin", true]]

// Looping through properties with for...in
for (let key in user) {
  console.log(`${key}: ${user[key]}`);
}

// Looping through entries with destructuring
for (let [key, value] of Object.entries(user)) {
  console.log(`${key}: ${value}`);
}

// Getting the number of properties in an object
console.log(Object.keys(user).length); // 3
```

5. The `this` Keyword In Different Contexts

The value of `this` depends on how a function is called:

javascript

Copy

```javascript
// 1. In global context
console.log(this); // window object (in browser)

// 2. In regular functions
function showThis() {
  console.log(this);
}
showThis(); // window object (in browser)

// 3. In methods (functions in objects)
const user = {
  name: "John",
  sayHi() {
    console.log(this.name);
  }
};
user.sayHi(); // "John" (this = user)

// 4. In arrow functions
const arrowFunc = () => {
  console.log(this);
};
arrowFunc(); // window object (inherits from surrounding scope)

// 5. With call, apply, bind
function greet() {
  console.log(`Hello, ${this.name}!`);
}

const person = { name: "Alice" };
greet.call(person); // "Hello, Alice!" (sets this to person)
greet.apply(person); // "Hello, Alice!" (same but different argument syntax)
const boundGreet = greet.bind(person); // Creates a new function with this bound to person
boundGreet(); // "Hello, Alice!"
```

Changing Context with call, apply, and bind

javascript

Copy

```javascript
function multiply(x, y) {
  return x * y * this.factor;
}

const calc1 = { factor: 2 };
const calc2 = { factor: 3 };

// call - immediately invokes the function with a specific context
console.log(multiply.call(calc1, 3, 4)); // 3 * 4 * 2 = 24

// apply - like call but takes arguments as array
console.log(multiply.apply(calc2, [2, 5])); // 2 * 5 * 3 = 30

// bind - returns a new function with context permanently bound
const doubleMultiply = multiply.bind(calc1);
console.log(doubleMultiply(2, 6)); // 2 * 6 * 2 = 24

// bind with preset parameters
const tripleMultiplyByFive = multiply.bind(calc2, 5);
console.log(tripleMultiplyByFive(4)); // 5 * 4 * 3 = 60
```

Common `this` Pitfalls

javascript

Copy

```javascript
const user = {
  name: "John",
  sayHi() {
    console.log(`Hi, ${this.name}!`);
  },
  sayHiLater() {
    // Problem: 'this' is lost in the callback
    setTimeout(function() {
      console.log(`Later: Hi, ${this.name}!`); // this = window
    }, 1000);
  },
  sayHiLaterArrow() {
    // Solution 1: Arrow function preserves 'this'
    setTimeout(() => {
      console.log(`Later with arrow: Hi, ${this.name}!`); // this = user
    }, 1000);
  },
  sayHiLaterBind() {
    // Solution 2: Explicitly bind 'this'
    setTimeout(function() {
      console.log(`Later with bind: Hi, ${this.name}!`);
    }.bind(this), 1000);
  }
};

user.sayHi(); // "Hi, John!"
user.sayHiLater(); // "Later: Hi, undefined!"
user.sayHiLaterArrow(); // "Later with arrow: Hi, John!"
user.sayHiLaterBind(); // "Later with bind: Hi, John!"
```

6. Variable Declarations and Global Properties

javascript

Copy

```javascript
// 'var' creates a property on the window object in global scope
var x = 10;
console.log(window.x); // 10

// 'let' and 'const' don't create properties on the window object
let y = 20;
console.log(window.y); // undefined

const z = 30;
console.log(window.z); // undefined

// The difference in scope
// 'var' is function-scoped
function varTest() {
  var a = 1;
  if (true) {
    var a = 2; // Same variable!
    console.log(a); // 2
  }
  console.log(a); // 2
}

// 'let' and 'const' are block-scoped
function letTest() {
  let b = 1;
  if (true) {
    let b = 2; // Different variable!
    console.log(b); // 2
  }
  console.log(b); // 1
}

// Reassignment
var c = 1;
c = 2; // OK

let d = 1;
d = 2; // OK

const e = 1;
// e = 2; // Error: Assignment to constant variable

// But you can modify object properties even with const
const obj = { prop: 1 };
```

```
obj.prop = 2; // OK
// obj = {}; // Error: Assignment to constant variable
```

7. Asynchronous JavaScript Basics

setTimeout and setInterval

javascript

Copy

```javascript
// setTimeout - executes code once after a delay
setTimeout(() => {
  console.log("This runs after 2 seconds");
}, 2000);

// setInterval - executes code repeatedly at intervals
const intervalId = setInterval(() => {
  console.log("This runs every 1 second");
}, 1000);

// Stop the interval after 5 seconds
setTimeout(() => {
  clearInterval(intervalId);
  console.log("Interval stopped");
}, 5000);

// Using this in setTimeout with a regular function
const user = {
  name: "John",
  sayHi() {
    console.log(`Hi, ${this.name}!`);
  },
  sayHiLater() {
    // 'this' will be lost in the callback - will be window
    setTimeout(function() {
      console.log(`Hi, ${this.name}!`);
    }, 1000);
  }
};

// Using arrow functions to preserve 'this'
const user2 = {
  name: "Jane",
  sayHiLater() {
    // Arrow function preserves 'this' value
    setTimeout(() => {
      console.log(`Hi, ${this.name}!`);
    }, 1000);
  }
};

user.sayHiLater(); // "Hi, undefined!" (this = window)
user2.sayHiLater(); // "Hi, Jane!" (this = user2)
```

Best Practices Summary

1. **Closures**:
   - Use closures for data privacy and encapsulation
   - Be mindful of memory usage as closures keep references to their parent scope
2. **Arrow Functions vs. Regular Functions**:
   - Use arrow functions when you want to preserve the lexical `this`
   - Use regular functions for methods that need their own `this` value
   - Don't use arrow functions for methods in objects if they need to access the object with `this`
3. **Objects**:
   - Use dot notation for simple property names
   - Use bracket notation for dynamic property names or names with special characters
   - Use Object.keys/values/entries for iteration over object properties
4. `this` **Keyword**:
   - Be aware of how different function types handle `this`
   - Use arrow functions in callbacks to preserve the outer `this` value
   - Use bind/call/apply to explicitly control `this` when needed
5. **Variable Declarations**:
   - Prefer `const` by default for variables that won't be reassigned
   - Use `let` for variables that need reassignment
   - Avoid `var` in modern code due to its function scope and global object pollution
6. **Asynchronous JavaScript**:
   - Use arrow functions in callbacks to preserve `this`
   - Be aware of the event loop and how async code is executed
   - Use clearTimeout/clearInterval to cancel scheduled executions