JavaScript Core Concepts - Complete Guide

Type Conversion

JavaScript automatically converts types when using operators. The plus operator (+) with strings performs concatenation, converting other types to strings. Other operators like multiplication, subtraction, and division attempt to convert operands to numbers. Boolean values convert to 1 (true) or 0 (false) in numeric operations.

javascript

Copy

```javascript
// String concatenation 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)

// If conversion to number isn't possible
"a" * 5;        // NaN
```

Variable Declarations

JavaScript has three variable declaration keywords:

- `var`: function-scoped, can be redeclared, hoisted with an initial value of undefined
- `let`: block-scoped, can be reassigned but not redeclared within the same scope
- `const`: block-scoped, cannot be reassigned after initialization

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

// const - block scoped, cannot be reassigned
const y = 1;
// y = 2;       // Error: Assignment to constant variable
```

Hoisting

JavaScript "hoists" declarations to the top of their scope. Function declarations are completely hoisted and can be used before their definition. Variables declared with `var` are hoisted with an initial value of undefined. Variables declared with `let` and `const` are hoisted but placed in a "temporal dead zone" where they can't be accessed before declaration.

javascript

Copy

```javascript
// var is hoisted with undefined value
console.log(a);    // undefined (not error)
var a = 10;

// let and const are hoisted but in "temporal dead zone"
// console.log(b); // ReferenceError: Cannot access 'b' before initialization
let b = 10;

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

// Function expressions are not hoisted
// sayHi();       // Error
var sayHi = function() {
  console.log("Hi");
};
```

Global Objects and Window

In browsers, the `window` object is the global object that contains all global variables and functions. Variables declared with `var` in the global scope become properties of the window object, while `let` and `const` declarations do not.

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 has two types of equality operators:

- `==` (loose equality): converts operands to the same type before comparison
- `===` (strict equality): checks both value and type without conversion

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
```

The `this` Keyword

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

- In methods (functions within objects), `this` refers to the calling object
- In regular function calls, `this` refers to the global object (window in browsers)
- In arrow functions, `this` is lexically scoped (inherits from the surrounding scope)

javascript

Copy

```javascript
// In regular functions, 'this' refers to the object that called the function
const obj = {
  name: "Cat",
  say: function() {
    console.log(this); // 'this' refers to obj
  }
};
obj.say(); // {name: "Cat", say: [Function]}

// When called directly, 'this' refers to the global object (window)
function normalFunction() {
  console.log(this);
}
normalFunction(); // window

// Methods vs Functions
const obj1 = { name: "Object 1" };
const obj2 = { name: "Object 2" };

function abc() {
  console.log(this);
}

obj1.say = abc;
obj2.talk = abc;

abc();        // window (no calling object)
obj1.say();   // obj1 (calling object is obj1)
obj2.talk();  // obj2 (calling object is obj2)
```

Object Destructuring

Destructuring allows extracting properties from objects into individual variables. You can assign different variable names and provide default values for properties that don't exist in the original object.

javascript

Copy

```javascript
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30
};

// Normal property access
console.log(person.firstName); // "John"

// Destructuring
const { firstName, lastName } = person;
console.log(firstName); // "John"
console.log(lastName);  // "Doe"

// Destructuring with different variable names
const { firstName: fName, lastName: lName } = person;
console.log(fName);  // "John"
console.log(lName);  // "Doe"

// Default values
const { city = "Unknown" } = person;
console.log(city);  // "Unknown" (not present in original object)
```

Spread Syntax

The spread syntax (`...`) expands arrays or objects. It's useful for creating new arrays or objects with additional elements or properties.

javascript

Copy

```javascript
// Arrays
const a = [2, 3, 4];
const b = [1, ...a, 5];  // Opens up array a
console.log(b);  // [1, 2, 3, 4, 5]

// Objects
const obj1 = { x: 1, y: 2 };
const obj2 = { ...obj1, z: 3 };
console.log(obj2);  // {x: 1, y: 2, z: 3}
```

Array Methods

JavaScript provides powerful array methods like `find()`, which returns the first element that satisfies a condition, and returns undefined if no element passes the test.

javascript

Copy

```javascript
// find() returns the first element that passes the test
const numbers = [2, 3, 4, 5];
const odd = numbers.find(x => x % 2 !== 0);
console.log(odd); // 3 (first odd number)

// Returns undefined if no element passes the test
const bigNumber = numbers.find(x => x > 10);
console.log(bigNumber); // undefined
```

Bind, Call, and Apply

These methods allow controlling what `this` refers to in a function:

- `bind()` returns a new function with a bound `this` value and optionally pre-set arguments
- `call()` immediately calls a function with a specified `this` value and comma-separated arguments
- `apply()` immediately calls a function with a specified `this` value and an array of arguments

javascript

Copy

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

const obj = { a: 10 };

// bind - returns a new function with bound context and optionally parameters
const boundFunc = multiply.bind(obj, 5);
console.log(boundFunc(2)); // 5 * 2 * 10 = 100

// call - immediately calls with given context and comma-separated arguments
console.log(multiply.call(obj, 3, 4)); // 3 * 4 * 10 = 120

// apply - immediately calls with given context and array of arguments
console.log(multiply.apply(obj, [2, 3])); // 2 * 3 * 10 = 60
```

Getters and Setters

Getters and setters allow accessing and modifying properties through functions while using property syntax. They're defined using the `get` and `set` keywords in object literals.

javascript

Copy

```javascript
const person = {
  firstName: "Maria",
  lastName: "Smith",

  // Getter - access like a property but executes a function
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },

  // Setter - assign like a property but executes a function
  set fullName(value) {
    [this.firstName, this.lastName] = value.split(" ");
  }
};

console.log(person.fullName); // "Maria Smith"
person.fullName = "John Doe";
console.log(person.firstName); // "John"
console.log(person.lastName);  // "Doe"
```

Timing Functions

- `setTimeout()` executes code once after a specified delay
- `setInterval()` executes code repeatedly at specified intervals
- Both return IDs that can be used with `clearTimeout()` or `clearInterval()` to cancel execution

javascript

Copy

```javascript
// setTimeout - runs once after specified delay (in milliseconds)
setTimeout(() => {
  console.log("Executed after 1 second");
}, 1000);

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

// Clear the interval to stop execution
// clearInterval(intervalId);
```

Arrow Functions

Arrow functions provide a concise syntax for writing functions. They don't have their own `this` binding, instead inheriting it from the surrounding scope. They can be written with or without curly braces depending on complexity.

javascript

Copy

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

// Arrow function with body and explicit return
const add1 = (a, b) => {
  return a + b;
};

// Arrow function with implicit return (single expression)
const add2 = (a, b) => a + b;

// Arrow function with single parameter (parentheses optional)
const square = x => x * x;

// Arrow function with no parameters (parentheses required)
const getRandomNumber = () => Math.random();
```

These concepts form the foundation of JavaScript programming and are essential for building effective web applications. Understanding these core principles will help you write more efficient and maintainable JavaScript code.