LEARN JAVASCRIPT

BEGINNERS EDITION

A Complete Beginner's Guide to Learn JavaScript

Suman Kunwar

Table of Contents

Dedication

This book is dedicated, in respect and admiration, to the spirit of computers and programming languages in our world.

> “The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.”
>
> \- Edsger Dijkstra

Copyright

Preface

"Learn JavaScript: Beginner's Edition" is a book that offers a comprehensive exploration of JavaScript, positioning it as a vital language in the ever-changing digital landscape. With a focus on foundation and practicality, this resource caters to everyone who wishes to learn the JavaScript programming language.

The book begins by covering the fundamental aspects of JavaScript, gradually progressing towards more advanced techniques. It addresses key topics such as variables, data types, control structures, functions, object-oriented programming, closures, promises, and modern syntax. Each chapter builds upon the previous one, providing a solid foundation for learners and facilitating the comprehension of complex concepts.

A standout feature of "Learn JavaScript" is its practical approach. The book offers hands-on exercises, coding challenges, and real-world problems that allow readers to apply their knowledge and develop essential skills. By engaging with tangible examples, readers gain the confidence to tackle common web development problems and unlock JavaScript's potential for innovative solutions.

Complex ideas such as closures and asynchronous programming are demystified through intuitive explanations and practical examples. The emphasis on clarity and simplicity enables learners of all levels to grasp and retain the key concepts effectively. The book is structured into three parts, with the first 14 chapters delving into the core concepts. The subsequent four chapters elaborate on the utilization of JavaScript for web browser programming, while the final two chapters cover miscellaneous subjects and offer exercises. The Miscellaneous section explores significant themes and scenarios pertaining to JavaScript programming, followed by exercises for practice.

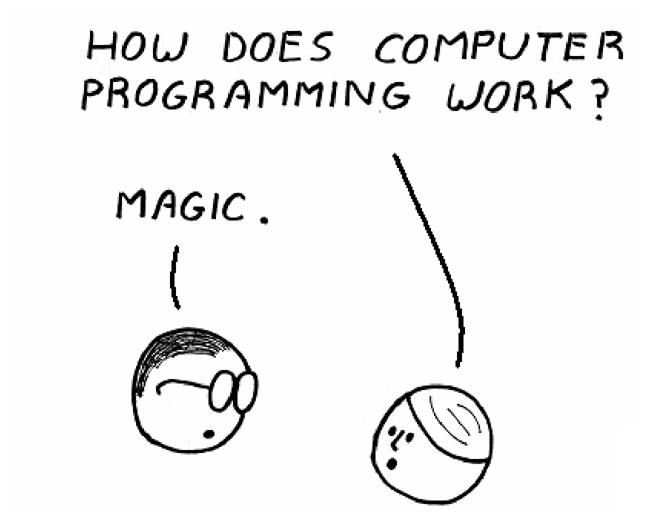
In conclusion, "Learn JavaScript: Beginner's Edition" is an essential companion for those seeking to master JavaScript and excel in web development. With its comprehensive coverage, practical approach, clear explanations, real-world application focus, and commitment to ongoing learning, this book serves as a valuable resource. By immersing themselves in its content, readers will gain the skills and knowledge necessary to build dynamic and interactive web applications, unlocking their full potential as JavaScript developers.

Chapter 1

Introduction

Computers are common in today's world, as they are able to perform a wide variety of tasks quickly and accurately. They are used in many different industries, such as business, healthcare, education, and entertainment, and have become an essential part of daily life for many people. Besides this, they are also used to perform complex scientific and mathematical calculations, to store and process large amounts of data, and to communicate with people around the world.

Programming involves creating a set of instructions, called a program, for a computer to follow. Writing a program can be tedious and frustrating at times because computers are very precise and need specific instructions in order to complete tasks.



Programming languages are artificial languages used to give instructions to computers. They are used in most programming tasks and are based on the way humans communicate with each other. Like human languages, programming languages allow words and phrases to be combined to express new concepts. It is interesting to note that the most effective way to communicate with computers involves using a language that is similar to human language.

In the past, the primary way to interact with computers was through language-based interfaces like BASIC and DOS prompts. These have been largely replaced by visual interfaces, which are easier to learn but offer less flexibility. However, computer languages like *JavaScript* are still in use and can be found in modern web browsers and on most devices.

JavaScript (*JS for short*) is the programming language that is used to create dynamic interaction while developing webpages, games, applications, and even servers. JavaScript started at Netscape, a web browser developed in the 1990s, and is today one of the most famous and used programming languages.

Initially, it was created for making webpages alive and was able to run only on a browser. Now, it runs on any device that supports the JavaScript engine. Standard objects such as Array, Date, and Math are available in JavaScript, as well as operators, control structures, and statements. *Client-side JavaScript* and *Server-side JavaScript* are the extended versions of core JavaScript.

- *Client-side JavaScript* enables the enhancement and manipulation of web pages, and client browsers. Responses to user events such as mouse clicks, form input, and page navigation are some of its examples.

- *Server-side JavaScript* enables access to servers, databases, and file systems.

JavaScript is an interpreted language. While running Javascript an interpreter interprets each line and runs it. The modern browser uses Just In Time (JIT) technology for compilation, which compiles JavaScript into executable bytecode.

"LiveScript" was the initial name given to JavaScript.

Code, and what to do with it and Typographic conventions

Code is the written instructions that make up a program. Here, many chapters contain a lot of code, and it is important to read and write code as part of learning how to program. You should not just quickly scan the examples - read them carefully and try to understand them. This may be difficult at first, but with practice, you will improve. The same goes for the exercises - make sure you actually try to write a solution before assuming you understand them. It is also helpful to try running your solutions to the exercises in a JavaScript interpreter, as this will allow you to see if your code is working correctly and may encourage you to experiment and go beyond the exercises.

Here, text is written in a monospaced font represents elements of a program. This can be a self-contained fragment or a reference to part of a nearby program. Programs, like the one shown below, are written in this way:

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

Sometimes, the expected output of a program is written after it, preceded by two slashes with a *Result*, like this:

```
console.log(txt);

// Result: txt = '45491625'
```

Chapter 2

Basics

In this first chapter, we'll learn the basics of programming and the Javascript language.

Programming means writing code. A book is made up of chapters, paragraphs, sentences, phrases, words, and finally punctuation and letters, likewise a program can be broken down into smaller and smaller components. For now, the most important is a statement. A statement is analogous to a sentence in a book. On its own, it has structure and purpose, but without the context of the other statements around it, it isn't that meaningful.

A statement is more casually (and commonly) known as a *line of code*. That's because statements tend to be written on individual lines. As such, programs are read from top to bottom, left to right. You might be wondering what code (also called source code) is. That happens to be a broad term which can refer to the whole of the program or the smallest part. Therefore, a line of code is simply a line of your program.

Here is a simple example:

```
let hello = "Hello";
let world = "World";

// Message equals "Hello World"
let message = hello + " " + world;
```

This code can be executed by another program called an *interpreter* that will read the code, and execute all the statements in the right order.

Comments

Comments are statements that will not be executed by the interpreter, comments are used to mark annotations for other programmers or small descriptions of what code does, thus making it easier for others to understand what your code does. They are also used to temporarily disable code without affecting the program control flow.

In JavaScript, comments can be written in 2 different ways:

- *Single-line comments*: It starts with two forward slashes (`//`) and continue until the end of the line. Anything following the slashes is ignored by the JavaScript interpreter. For example:

```
// This is a comment, it will be ignored by the interpreter
let a = "this is a variable defined in a statement";
```

- *Multi-line comments*: It starts with a forward slash and an asterisk (`/*`) and end with an asterisk and a forward slash (`*/`). Anything between the opening and closing markers is ignored by the JavaScript interpreter. For example:

```
/*
This is a multi-line comment,
it will be ignored by the interpreter
*/
let a = "this is a variable defined in a statement";
```

Including comments in code is essential for maintaining code quality, enabling collaboration, and simplifying the debugging process. By providing context and explanations for various parts of the program, comments make it easier to understand the code in the future. Therefore, it is considered a beneficial practice to include comments in code.

Variables

The first step towards really understanding programming is looking back at algebra. If you remember it from school, algebra starts with writing terms such as the following.

```
3 + 5 = 8
```

You start performing calculations when you introduce an unknown, for example, x below:

```
3 + x = 8
```

Shifting those around you can determine x:

```
x = 8 - 3
-> x = 5
```

When you introduce more than one you make your terms more flexible - you are using variables:

```
x + y = 8
```

You can change the values of x and y and the formula can still be true:

```
x = 4
y = 4
```

or

```
x = 3
y = 5
```

The same is true for programming languages. In programming, variables are containers for values that change. Variables can hold all kinds of values and also the results of computations. Variables have a `name` and a `value` separated by an equals sign (=). However, it is important to keep in mind that different programming languages have their own limitations and constraints on what can be used as variable names. This is because certain words may be reserved for specific functions or operations within the language.

Let's check out how it works in Javascript. The following code defines two variables, computes the result of adding the two, and defines this result as a value of a third variable.

```
let x = 5;
let y = 6;
let result = x + y;
```

There are certain guidelines that needs to be followed while naming variables. They are

- Variable names have to start with a letter, an underscore (_), or a dollar sign ($).
- After the first character, we can use letters, numbers, underscores, or dollar signs.
- JavaScript distinguishes between uppercase and lowercase letters (case-sensitive), so myVariable, MyVariable, and MYVARIABLE are all separate variables.

- To make your code easy to read and maintain, it's recommended to use descriptive variable names that accurately reflect their purpose.

Exercise

Define a variable `x` equal to 20.

```
let x =
```

ES6 Version

ECMAScript 2015 or ES2015 also known as E6 is a significant update to the JavaScript programming language since 2009. In ES6 we have three ways of declaring variables.

```
var x = 5;
const y = 'Test';
let z = true;
```

The types of declaration depend upon the scope. Unlike the `var` keyword, which defines a variable globally or locally to an entire function regardless of block scope, `let` allows you to declare variables that are limited in scope to the block, statement, or expression in which they are used. For example.

```
function varTest(){
    var x=1;
    if(true){
        var x=2; // same variable
        console.log(x); //2
    }
    console.log(x); //2
}

function letTest(){
    let x=1;
    if(true){
        let x=2;
        console.log(x); // 2
    }
    console.log(x); // 1
}
```

`const` variables are immutable meaning that they are not allowed to be re-assigned.

```
const x = "hi!";
x = "bye"; // this will occurs an error
```

Types

Computers are sophisticated and can make use of more complex variables than just numbers. This is where variable types come in. Variables come in several types and different languages support different types.

The most common types are:

- **Number**: Numbers can be integers (e.g., `1`, `-5`, `100`) or floating-point values (e.g., `3.14`, `-2.5`, `0.01`). JavaScript does not have a separate type for integers and floating-point values; it treats them both as numbers.
- **String**: Strings are sequences of characters, represented by either single quotes (e.g., `'hello'`) or double quotes (e.g., `"world"`).
- **Boolean**: Booleans represent a true or false value. They can be written as `true` or `false` (without quotes).
- **Null**: The null type represents a null value, which means "no value." It can be written as `null` (without quotes).
- **Undefined**: The undefined type represents a value that has not been set. If a variable has been declared, but has not been assigned a value, it is `undefined`.
- **Object**: An object is a collection of properties, each of which has a name and a value. You can create an object using curly braces (`{}`) and assigning properties to it using name-value pairs.
- **Array**: An array is a special type of object that can hold a collection of items. You can create an array using square brackets (`[]`) and assigning a list of values to it.
- **Function**: A function is a block of code that can be defined and then called by name. Functions can accept arguments (inputs) and return a value (output). You can create a function using the `function` keyword.

JavaScript is a "*loosely typed*" language, which means that you don't have to explicitly declare what type of data the variables are. You just need to use the `var` keyword to indicate that you are declaring a variable, and the interpreter will work out what data type you are using from the context, and use of quotes.

Exercise

Declare three variables and initialize them with the following values: `age` as a number, `name` as a string and `isMarried` as a boolean.

```
let age =
let name =
let isMarried =
```

The `typeof` operator is used to checking the datatypes of a variable.

```
typeof "John"                 // Returns "string"
typeof 3.14                   // Returns "number"
typeof NaN                    // Returns "number"
typeof false                  // Returns "boolean"
typeof [1,2,3,4]              // Returns "object"
typeof {name:'John', age:34}  // Returns "object"
typeof new Date()             // Returns "object"
typeof function () {}         // Returns "function"
typeof myCar                  // Returns "undefined" *
typeof null                   // Returns "object
```

Data types used in JavaScript can be differentiated into two categories based on containing values.

Data types that can contain values:

- `string`
- `number`
- `boolean`
- `object`
- `function`

> `Object` , `Date` , `Array` , `String` , `Number` , and `Boolean` are the types of objects available in JavaScript.

Data types that cannot contain values:

- `null`
- `undefined`

A primitive data value is a simple data value with no additional properties and methods, and is not an object. They are immutable, meaning that they can't be altered. There are 7 primitive data types:

- string
- number
- bigint
- boolean
- undefined
- symbol
- null

Exercise

Declare a variable called `person` and initialize it with an object that contains the following properties: `age` as a number, `name` as a string and `isMarried` as a boolean.

```
let person =
```

Equality

While writing a program we frequently need to determine the equality of variables in relation to other variables. This is done using an equality operator. The most basic equality operator is the `==` operator. This operator does everything it can to determine if two variables are equal, even if they are not of the same type.

For example, assume:

```
let foo = 42;
let bar = 42;
let baz = "42";
let qux = "life";
```

`foo == bar` will evaluate to `true` and `baz == qux` will evaluate to `false`, as one would expect. However, `foo == baz` will also evaluate to `true` despite `foo` and `baz` being different types. Behind the scenes the `==` equality operator attempts to force its operands to the same type before determining their equality. This is in contrast to the `===` equality operator.

The `===` equality operator determines that two variables are equal if they are of the same type *and* have the same value. With the same assumptions as before, this means that `foo === bar` will still evaluate to `true`, but `foo === baz` will now evaluate to `false`. `baz === qux` will still evaluate to `false`.

Exercise

Use the `==` and `===` operator to compare the values of `str1` and `str2`.

```
let str1 = "hello";
let str2 = "HELLO";
let bool1 = true;
let bool2 = 1;
// compare using ==
let stringResult1 =
let boolResult1 =
// compare using ===
let stringResult1 =
let boolResult2 =
```

Chapter 3

Numbers

JavaScript has **only one type of number** – 64 bit float point. It's the same as Java's `double`. Unlike most other programming languages, there is no separate integer type, so 1 and 1.0 are the same value. Creating a number is easy, it can be done just like for any other variable type using the `var` keyword.

Numbers can be created from a constant value:

```
// This is a float:
let a = 1.2;

// This is an integer:
let b = 10;
```

Or, from the value of another variable:

```
let a = 2;
let b = a;
```

The precision of integers is accurate up to 15 digits and the maximum number is 17.

```
let x = 999999999999999;   // x will be 999999999999999
let y = 9999999999999999;  // y will be 10000000000000000
```

The numeric constants are interpreted as hexadecimal if they are preceded by an `0x`.

```
let z = 0xFF; // 255
```

Math

The `Math` object allows performing mathematical operations in JavaScript. It is static and doesn't have a constructor. One can use method and properties of Math object without creating a Math object first. For accessing its property one can use *Math.property.* Some of the math properties are described below:

```
Math.E; // returns Euler's number
Math.PI; // returns PI
Math.SQRT2; // returns the square root of 2
Math.SQRT1_2; // returns the square root of 1/2
Math.LN2; // returns the natural logarithm of 2
Math.LN10; // returns the natural logarithm of 10
Math.LOG2E; // returns base 2 logarithm of E
Math.LOG10E; // returns base 10 logarithm of E
```

Examples of some of the math methods are:

```
Math.pow(8, 2); // 64
Math.round(4.6); // 5
Math.ceil(4.9); // 5
Math.floor(4.9); // 4
Math.trunc(4.9); // 4
Math.sign(-4); // -1
Math.sqrt(64); // 8
Math.abs(-4.7); // 4.7
Math.sin((90 * Math.PI) / 180); // 1 (the sine of 90 degrees)
Math.cos((0 * Math.PI) / 180); // 1 (the cos of 0 degrees)
Math.min(0, 150, 30, 20, -8, -200); // -200
Math.max(0, 150, 30, 20, -8, -200); // 150
Math.random(); // 0.44763808380924375
Math.log(2); // 0.6931471805599453
Math.log2(8); // 3
Math.log10(1000); // 3
```

To access maths method, one can call its methods directly with arguments wherever necessary.

Method	Description
`abs(x)`	Returns absolute value of `x`
`acos(x)`	Returns arccosine of `x` , in radians
`acosh(x)`	Returns hyperbolic arccosine of `x`
`asin(x)`	Returns arcsine of `x` , in radians
`asinh(x)`	Returns hyperbolic arcsine of `x`
`atan(x)`	Returns arctangent of `x` as a numeric value between `-PI/2` and `PI/2` radians
`atan2(y,x)`	Returns arctangent of the quotient of its arguments
`atanh(x)`	Returns hyperbolic arctangent of `x`
`crbt(x)`	Returns cubic root of `x`
`ceil(x)`	Returns rounded upwards to the nearest integer of `x`
`cos(x)`	Returns consine of `x` , in radians
`cosh(x)`	Returns hyperbolic cosine of `x`
`exp(x)`	Returns exponential value of `x`
`floor(x)`	Returns round downwards to the nearest integer of `x`
`log(x)`	Returns natural logarithmetic of `x`
`max(x,y,z,... n)`	Returns number with the highest value
`min(x,y,z,... n)`	Returns number with the lowest value
`pow(x,y)`	Returns value of `x` to the power of `y`
`random()`	Returns number between 0 and 1
`round(x)`	Rounds number to the nearest `x`
`sign(x)`	Returns if x is negative, `null` or positive (-1,0,1)
`sin(x)`	Returns sine of `x` , in radians
`sinh(x)`	Returns hyperbolic sine of `x`
`sqrt(x)`	Returns square root of `x`
`tan(x)`	Returns tangent of an angle
`tanh(x)`	Returns hyperbolic tangent of `x`
`trunc(x)`	Returns integer part of a number (`x`)

Basic Operators

In JavaScript, an operator is a symbol or keyword user to perform operations on operands (values and variables). For example,

```
2 + 3; //5
```

Here `+` is an operator that performs addition, and `2` and `3` are operands.

Types of Operators

There are various operators supported by JavaScript. They are as follows:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Ternary Operators
- Bitwise Operators
- `typeof`` Operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on values. They include

- Addition (`+`) operator
- Subtraction (`-`) operator
- Multiplication (`*`) operator
- Division (`/`) operator
- Remainder (`%`) operator

Addition Operator (`+`)

The addition operator adds two numbers together. For example:

```
console.log(1 + 2); // 3
console.log(1 + -2); // -1
```

Subtraction operator (`-`)

The subtraction operator subtracts one number from another. For example:

```
console.log(3 - 2); // 1
console.log(3 - -2); // 5
```

Multiplication operator (`*`)

The multiplication operator multiplies two numbers. For example:

```
console.log(2 * 3); // 6
console.log(2 * -3); // -6
```

Division operator (`/`)

The division operator divides one number by another. For example:

```
console.log(6 / 2); // 3
console.log(6 / -2); // -3
```

Remainder operator (`%`)

The remainder operator returns the remainder of a division operation. For example:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

The JavaScript interpreter works from left to right. One can use parentheses just like in math to separate and group expressions: `c = (a / b) + d`

> JavaScript uses the `+` operator for both addition and concatenation. Numbers are added whereas strings are concatenated.

The term `NaN` is a reserved word indicating that a number is not a legal number, this arises when we perform arithmetic with a non-numeric string will result in `NaN` (Not a Number).

```
let x = 100 / "10";
```

The `parseInt` method parses a value as a string and returns the first integer.

```
parseInt("10"); // 10
parseInt("10.00"); // 10
parseInt("10.33"); // 10
parseInt("34 45 66"); // 34
parseInt(" 60 "); // 60
parseInt("40 years"); //40
parseInt("He was 40"); //NaN
```

In JavaScript, if we calculate a number outside the largest possible number it returns `Infinity`.

```
let x = 2 / 0; // Infinity
let y = -2 / 0; // -Infinity
```

Exercise

Use the math operators +, -, *, /, and % to perform the following operations on `num1` and `num2`.

```
let num1 = 10;
let num2 = 5;

// Add num1 and num2.
let addResult =
// Subtract num2 from num1.
let subtractResult =
// Multiply num1 and num2.
let multiplyResult =
// Divide num1 by num2.
let divideResult =
// Find the remainder num1 is divided by num2.
let reminderResult =
```

Assignment Operators

Assignment operators are used to assign values to variables or evaluates the assigned value. *Chaining the assignmentoperator is possible in order to assign a single value to multiple values.* They includes the assignment(`=`) operator and compound assignment operators like `+=` , `-=` , `*=` and `/=` .

`=` (Assignment Operator)

This operator is used to assign the value on the right side to the variable on the left side. For examples:

```
let x = 10; //Assigns the value 10 to the variable x.
```

Compound Assignment Operators

These operator combine the arithmetic operation with the assignment operation. They are shortcuts for performing an operation and then assigning the result back to the variable. For example:

`+=` (Addition Assignment)

It adds the value on the right side to the variable and assigns the result back to the variable.

`-=` (Subtraction Assignment)

It subtracts the value on the right side from the variable and assigns the result back to the variable.

`*=-` (Multiplication Assignment)

It multiplies the variable by the value on the right side and assigns the result back to the variable.

`/=` (Division Assignment)

It divides the variable by the value on the right side and assigns the result back to the variable.

`%=` (Modules/Remainder Assignment)

It computes the remainder when the variable is divided by the value on the right side and assigns the result back to the variable.

```
let a = 10;
a += 5; // Equivalent to a = a + 5; (a becomes 15)
a -= 3; // Equivalent to a = a - 3; (a becomes 12)
a *= 2; // Equivalent to a = a * 2; (a becomes 24)
a /= 4; // Equivalent to a = a / 4; (a becomes 6)
a %= 5; // Equivalent to a = a % 5; (a becomes 1)
```

Comparison Operators

Comparison operators are used to compare two values or expressions and return a `boolean` result, which is either `true` or `false`. These operators are commonly used in conditional statements to make decision or evaluate conditions.

Equal to (`==`)

This operator checks if the values on the left and right sides are equal. If they are equal, it returns `true` otherwise, it returns false. It does not consider data types.

```
5 == 5; // true
"5" == 5; // true (implicit type conversion)
```

Not equal to (`!=`)

This operator checks if the values on the left and right sides are not equal. If they are not equal, it returns `true` otherwise, it returns `false`.

```
5 != 3; // true
"5" != 5; // false (implicit type conversion)
```

Strict Equal to (`===`)

This operator checks if the values on the left and right sides are equal and have the same data type. If both the value and data type match, it returns `true` otherwise, it returns `false`.

```
5 === 5; // true
"5" === 5; // false (different data types)
```

Strict Not equal to (`!==`)

This operator checks if the values on the left and right sides are not equal or have different data types. If they are not equal or have different data types, it returns `true` otherwise, it returns `false`.

```
5 !== "5"; // true (different data types)
5 !== 5; // false
```

Greater than (`>`)

This operator checks if the value on the left is greater than the value on the right. If the left value is greater, it returns `true` otherwise, it returns `false`.

```
8 > 5; // true
3 > 10; // false
```

Less than (`<`)

This operator checks if the value on the left is less than the value on the right. If the left value is less, it returns `true` otherwise, it returns `false` .

```
3 < 5; // true
8 < 2; // false
```

Greater than or equal to(`>=`)

This operator checks if the value on the left is greater than or equal to the value on the right. If the left value is greater or equal, it returns `true` otherwise, it returns `false` .

```
8 >= 5; // true
3 >= 8; // false
```

Less than or equal to (`>=`)

This operator checks if the value on the left is less than or equal to the value on the right. If the left value is less or equal, it returns `true` otherwise, it returns `false` .

```
3 <= 5; // true
8 <= 2; // false
```

Logical Operators

Logical operators are used to perform logical operations on Boolean values or expressions. These operators allow you to combine or manipulate Boolean values to make decisions or evaluate complex conditions.

Logical AND (`&&`)

The logical AND operator returns `true` if both operands are `true` . If at least one of the operands is `false` , it returns `false` .

```
true && true; // true
true && false; // false
false && true; // false
false && false; // false
```

Logical OR (`||`)

The logical OR operator returns `true` if at least one of the operands is `true` . It returns `false` only if both operands are `false` .

```
true || true; // true
true || false; // true
false || true; // true
false || false; // false
```

Logical NOT (`!`)

The logical NOT operator negates the value of an operand. It returns `true` if the operand is `false` , and it returns `false` if the operand is `true` .

```
!true; // false
!false; // true
```

Ternary Operators

Ternary operator has three operands. It is the simplified operator of if/else.

It is the short form of the `if-else` condition.

Syntax

```
Y =  ? A : B
If the condition is true then Y = A otherwise Y = B
```

```
let isEven = 8 % 2 === 0 ? "Even" : "Odd";
console.log(isEven); // "Even"
```

Bitwise Operators

Bitwise operators are used to manipulate individual bits of binary numbers. They perform operations at the bit level, which is especially useful in situations where you need to control or analyze low-level data.

Bitwise AND (`&`)

This operator compares each bit of two numbers and returns 1 for each bit that is 1 in both numbers. All other bits are set to 0.

```
1010 & 1100; // 1000
```

Bitwise OR (`|`)

This operator compares each bit of two numbers and returns 1 for each bit that is 1 in at least one of the numbers.

```
1010 | 1100; // 1110
```

Bitwise XOR (`^`)

This operator compares each bit of two numbers and returns 1 for each bit that is 1 in one number but not in both.

```
1010 ^ 1100; // 0110
```

Bitwise NOT (`~`)

This operator inverts (flips) all the bits of a number. It changes each 0 to 1 and each 1 to 0.

```
~1010; // 0101
```

Left Shift (`<<`)

This operator shifts the bits of a number to the left by a specified number of positions, filling the shifted-in positions with 0.

```
1010 << 2; // 101000 (shifted left by 2 positions)
```

Right Shift (`>>`)

This operator shifts the bits of a number to the right by a specified number of positions. The shifted-in positions are filled based on the leftmost bit (sign bit).

```
1010 >> 2; // 0010 (shifted right by 2 positions)
```

`typeof` Operators

It returns the operand type, The possible types that exist in javascript are undefined, Object, boolean, number, string, symbol, and function.

```
let value1 = 42;
let value2 = "Hello, World!";
let value3 = true;
let value4 = null;

console.log(typeof value1); // "number"
console.log(typeof value2); // "string"
console.log(typeof value3); // "boolean"
console.log(typeof value4); // "object" (Note: `typeof null` returns "object" due to historical reasons)
```

Advanced Operators

When operators are put together without parenthesis, the order in which they are applied is determined by the *precedence* of the operators. Multiplication `(*)` and division `(/)` has higher precedence than addition `(+)` and subtraction `(-)`.

```
// multiplication is done first, which is then followed by addition
let x = 100 + 50 * 3; // 250
// with parenthesis operations inside the parenthesis are computed first
let y = (100 + 50) * 3; // 450
// operations with the same precedences are computed from left to right
let z = 100 / 50 * 3;
```

Several advanced math operators can use be used while writing program. Here is a list of some of the main advanced math operators:

- **Modulo operator (`%`)**: The modulo operator returns the remainder of a division operation. For example:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

- **Exponentiation operator (* *)**: The exponentiation operator raises a number to the power of another number. It is a newer operator and is not supported in all browsers, so you may need to use the `Math.pow` function instead. For example:

```
console.log(2 ** 3); // 8
console.log(3 ** 2); // 9
console.log(4 ** 3); // 64
```

- **Increment operator (`++`)**: The increment operator increments a number by one. It can be used as a prefix (before the operand) or a postfix (after the operand). For example:

```
let x = 1;
x++; // x is now 2
++x; // x is now 3
```

- **Decrement operator (`--`)**: The decrement operator decrements a number by one. It can be used as a prefix (before the operand) or a postfix (after the operand). For example:

```
let y = 3;
y--; // y is now 2
--y; // y is now 1
```

- **Math object**: The `Math` object is a built-in object in JavaScript that provides mathematical functions and constants. You can use the methods of the `Math` object to perform advanced math operations, such as finding the square root of a number, calculating the sine of a number, or generating a random number. For example:

```
console.log(Math.sqrt(9)); // 3
console.log(Math.sin(0)); // 0
console.log(Math.random()); // a random number between 0 and 1
```

These are just a few examples of the advanced math operators and functions available in JavaScript. There are many more that you can use to perform advanced math operations while writing program.

Exercise

Use the following advanced operators to perform operations on `num1` and `num2`.

```
let num1 = 10;
let num2 = 5;

// ++ operator to increment the value of num1.
const result1 =
// -- operator to decrement the value of num2.
const result2 =
//  += operator to add num2 to num1.
const result3 =
// -= operator to subtract num2 from num1.
const result4 =
```

Nullish coalescing operator '??'

The `nullish` coalescing operator returns the first argument if it's not `null/undefined`, else the second one. It is written as two question marks `??`. The result of `x ?? y` is:

- if `x` is defined, then `x`,
- if `y` isn't defined, then `y`.

It's a recent addition to the language and might need polyfills to support old browsers

Chapter 4

Strings

JavaScript strings share many similarities with string implementations from other high-level languages. They represent text-based messages and data. In this course, we will cover the basics. How to create new strings and perform common operations on them.

Here is an example of a string:

```
"Hello World"
```

String indexes are zero-based, meaning that starting position of the first character at `0` followed by others in incremental order. Various methods are supported by string and return a new value. These methods are described below.

Name	Description
`charAt()`	Returns character at specified index
`charCodeAt()`	Returns Unicode character at specified index
`concat()`	Returns two or more combined strings
`constructor`	Returns string's constructor function
`endsWith()`	Checks if a string ends with a specified value
`fromCharCode()`	Returns Unicode values as characters
`includes()`	Checks if a string contains with a specified value
`indexOf()`	Returns the index of its first occurrence
`lastIndexOf()`	Returns the index of its last occurrence
`length`	Returns the length of the string
`localeCompare()`	Compares two strings with locale
`match()`	Matches a string against a value or regular expression
`prototype`	Used to add properties and method of an object
`repeat()`	Returns new string with number of copies specified
`replace()`	Returns a string with values replaced by a regular expression or a string with a value
`search()`	Returns an index based on a string's match against a value or regular expression
`slice()`	Returns a string containing part of a string
`split()`	Splits string into array of substrings
`startsWith()`	Checks strings beginning against specified character
`substr()`	Extracts part of string, from start index
`substring()`	Extracts part of string, between two indices
`toLocalLowerCase()`	Returns string with lowercase characters using host's locale
`toLocalUpperCase()`	Returns string with uppercase characters using host's locale
`toLowerCase()`	Returns string with lowercase characters
`toString()`	Returns string or string object as string
`toUpperCase()`	Returns string with uppercase characters
`trim()`	Returns string with removed whitespaces
`trimEnd()`	Returns string with removed whitespaces from end
`trimStart()`	Returns string with removed whitespaces from start
`valueOf()`	Returns primitive value of string or string object

Creation

Strings can be defined by enclosing the text in single quotes or double quotes:

```
// Single quotes can be used
let str = "Our lovely string";

// Double quotes as well
let otherStr = "Another nice string";
```

In Javascript, Strings can contain UTF-8 characters:

```
"中文 español English हिन्दी العربية português বাংলা русский 日本語 ਪੰਜਾਬੀ 한국어";
```

You can also use the `String` constructor to create a string object:

```
const stringObject = new String('This is a string');
```

However, it is generally not recommended to use the `String` constructor to create strings, as it can cause confusion between string primitives and string objects. It is usually better to use string literals to create strings.

You can also use template literals to create strings. Template literals are strings that are enclosed in backticks `(``)` and can contain placeholders for values. Placeholders are denoted with the `` `${}` `` syntax.

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

Template literals can also contain multiple lines and can include any expression inside the placeholders.

> Strings can not be subtracted, multiplied, or divided.

Exercise

Use a template literal to create a string that includes the values of `name` and `age`. The string should have the following format: "My name is John and I am 25 years old.".

```
let name = "John";
let age = 25;

// My name is John and I am 25 years old.
let result =
```

Replace

The `replace` method allows us to replace a character, word, or sentence with a string. For example.

```
let str = "Hello World!";
let new_str = str.replace("Hello", "Hi");

console.log(new_str);

// Result: Hi World!
```

> To replace a value on all instances of a regular expression with a `g` modifier is set.

It searches for a string for a value or a regular expression and returns a new string with the value(s) replaced. It doesn't change the original string. Let's see the global case-insensitive replacement example.

```
let text = "Mr Blue has a blue house and a blue car";
let result = text.replace(/blue/gi, "red");

console.log(result);
//Result: Mr red has a red house and a red car
```

Length

It's easy in Javascript to know how many characters are in a string using the property `.length`. The `length` property returns the number of characters in the string, including spaces and special characters.

```
let size = "Our lovely string".length;
console.log(size);
// size: 17

let emptyStringSize = "".length
console.log(emptyStringSize);
// emptyStringSize: 0
```

The length property of an empty string is `0`.

> The `length` property is a read-only property, so you cannot assign a new value to it.

Concatenation

Concatenation involves adding two or more strings together, creating a larger string containing the combined data of those original strings. The concatenation of a string appends one or more strings to another string. This is done in JavaScript using the following ways.

- using the `+` operator
- using the `concat()` method
- using the array `join()` method
- using the template literal (introduced in ES6)

The string `concat()` method accepts the list of strings as parameters and returns a new string after concatenation i.e. combination of all the strings. Whereas the array `join()` method is used to concatenate all the elements present in an array by converting them into a single string.

The template literal uses backtick ``(``)`` and provides an easy way to create multiline strings and perform string interpolation. An expression can be used inside the backtick using `$` sign and curly braces `${expression}`.

```
const icon = '👋';
// using template Strings
`hi ${icon}`;

// using join() Method
['hi', icon].join(' ');

// using concat() Method
''.concat('hi ', icon);

//  using + operator
'hi ' + icon;
// hi 👋
```

Split

The split() method divides a string into a list of substrings and returns them as an array.

- using the `split()` method
- using the template literal (introduced in ES6)

The split() method takes in:

separator (optional) - The pattern (string or regular expression) describing where each split should occur. limit (optional) - A non-negative integer limiting the number of pieces to split the given string into.

```
console.log("ABCDEF".split("")); // [ 'A', 'B', 'C', 'D', 'E', 'F' ]

const text = "Java is awesome. Java is fun.";

let pattern = ".";
let newText = text.split(pattern);
console.log(newText); // [ 'Java is awesome', ' Java is fun', '' ]

let pattern1 = ".";
// only split string to maximum to parts
let newText1 = text.split(pattern1, 2);
console.log(newText1); // [ 'Java is awesome', ' Java is fun' ]

const text2 = "JavaScript ;  Python ;C;C++";
let pattern2 = ";";
let newText2 = text2.split(pattern2);
console.log(newText2); // [ 'JavaScript ', '  Python ', 'C', 'C++' ]

// using RegEx
let pattern3 = /\s*(?:;|$)\s*/;
let newText3 = text2.split(pattern3);
console.log(newText3); // [ 'JavaScript', 'Python', 'C', 'C++' ]

//Output
[ 'A', 'B', 'C', 'D', 'E', 'F' ]
[ 'Java is awesome', ' Java is fun', '' ]
[ 'Java is awesome', ' Java is fun' ]
[ 'JavaScript ', '  Python ', 'C', 'C++' ]
[ 'JavaScript', 'Python', 'C', 'C++' ]
```

CharAt

The `str.charAt()` method returns the character at the given index of the string index holds the array element position.

- using the `charAt()` method
- using the template literal (introduced in ES6)

The charAt() method takes in:

Arguments: The only argument to this function is the index in the string from where the single character is to be extracted.

index: The range of this index is between 0 and length – 1. If no index is specified then the first character of the string is returned as 0 is the default index used for this function. Return Value: Returns a single character located at the index specified as the argument to the function. If the index is out of range, then this function returns an empty string.

```
//Example 1:
function func() {
    // Original string
    let str = 'JavaScript is object oriented language';

    // Finding the character at given index
    let value = str.charAt(0);
    let value1 = str.charAt(4);
    console.log(value);
    console.log(value1);
}
func();

//Output
j
s

//Example 2:
function func() {

    // Original string
    let str = 'JavaScript is object oriented language';

    // Finding the character at given index
    let value = str.charAt(9);
    console.log(value);
}
func();


//Output
t
```

Substring

The `string.substring()` is an inbuilt function in JavaScript that is used to return the part of the given string from the start index to the end index. Indexing start from zero (0).

Syntax:

string.substring(Startindex, Endindex)

Syntax:

*str.substr(start , length)

- using the `substr()` method
- using the template literal (introduced in ES6)

The substr() method takes in:

Parameters: Here the Startindex and Endindex describe the part of the string to be taken as a substring. Here the Endindex is optional. Return value: It returns a new string that is part of the given string. JavaScript code to show the working of string.substring() function:

```
//Example 1:
// JavaScript to illustrate substr() method

const str = "GeeksforGeeks";
const result = str.substring(8);
console.log(result);



//Output
for
```

```
//Example 2:
// Taking a string as letiable
let string = "geeksforgeeks";
a = string.substring(-1)
b = string.substring(2.5)
c = string.substring(2.9)

// Printing new string which are
// the part of the given string
console.log(a);
console.log(b);
console.log(c);


//Output
geeksforgeeks
eksforgeeks
eksforgeeks
```

Chapter 5

Conditional Logic

A condition is a test for something. Conditions are very important for programming, in several ways:

First of all, conditions can be used to ensure that your program works, regardless of what data you throw at it for processing. If you blindly trust data, you’ll get into trouble and your programs will fail. If you test that the thing you want to do is possible and has all the required information in the right format, that won’t happen, and your program will be a lot more stable. Taking such precautions is also known as programming defensively.

The other thing conditions can do for you is allow for branching. You might have encountered branching diagrams before, for example when filling out a form. Basically, this refers to executing different “branches” (parts) of code, depending on if the condition is met or not.

In this chapter, we'll learn the basis of conditional logic in JavaScript.

If

The easiest condition is an if statement and its syntax is `if(condition){ do this … }`. The condition has to be true for the code inside the curly braces to be executed. You can for example test a string and set the value of another string dependent on its value as described below.

```
let country = "France";
let weather;
let food;
let currency;

if (country === "England") {
  weather = "horrible";
  food = "filling";
  currency = "pound sterling";
}

if (country === "France") {
  weather = "nice";
  food = "stunning, but hardly ever vegetarian";
  currency = "funny, small and colourful";
}

if (country === "Germany") {
  weather = "average";
  food = "worst thing ever";
  currency = "funny, small and colourful";
}

let message =
  "this is " +
  country +
  ", the weather is " +
  weather +
  ", the food is " +
  food +
  " and the " +
  "currency is " +
  currency;

console.log(message);
// 'this is France, the weather is nice, the food is stunning, but hardly ever vegetarian and the currency is f
```

Nested If-Else

In JavaScript, you can use nested `if-else` statements to create more complex conditional logic.

Basic Syntax

```
if (condition1) {
  // Code to execute when condition1 is true
} else {
  if (condition2) {
    // Code to execute when condition1 is false and condition2 is true
  } else {
    // Code to execute when both condition1 and condition2 are false
  }
}
```

The following program determines a person's student status based on their age and prints a corresponding message.

```
let age = 20;
let isStudent = true;

if (age >= 18) {
  if (isStudent) {
    console.log("You are an adult student.");
  } else {
    console.log("You are an adult, but not a student.");
  }
} else {
  console.log("You are not an adult.");
}

// Output: You are an adult student.
```

This program checks for rain, temperature, and snow to provide weather advice.

```
let temperature = 25;
let isRaining = true;
let isSnowing = false;

if (isRaining) {
  console.log("It's raining. Don't forget your umbrella.");

  if (temperature < 10) {
    console.log("And it's cold. You might need a coat too.");
  }
} else if (isSnowing) {
  console.log("It's snowing. Be prepared for slippery roads.");
} else {
  console.log("No rain or snow. Enjoy the weather!");
}

// Output: It's raining. Don't forget your umbrella.
```

This program checks a person's age, prior driving experience, and written test status to determine eligibility for a driver's license.

```
let age = 19;
let hasPriorExperience = true;
let hasPassedWrittenTest = true;

if (age >= 18) {
  if (hasPriorExperience) {
    console.log("Congratulations! You are eligible for a driver's license.");
  } else {
    console.log("Sorry, you need prior driving experience to get a driver's license.");
  }
} else {
  console.log("Sorry, you must be 18 or older to apply for a driver's license.");

  if (hasPassedWrittenTest) {
    console.log("You've passed the written test, but you need to wait until you're 18 to apply.");
  } else {
    console.log("You need to pass the written test first and wait until you're 18 to apply.");
  }
}

// Output: Congratulations! You are eligible for a driver's license.
```

Else

There is also an `else` clause that will be applied when the first condition isn't true. This is very powerful if you want to react to any value, but single out one in particular for special treatment.

```
let umbrellaMandatory;

if (country === "England") {
  umbrellaMandatory = true;
} else {
  umbrellaMandatory = false;
}
```

The `else` clause can be joined with another `if`. Let's remake the example from the previous article.

```
if (country === "England") {
  ...
} else if (country === "France") {
  ...
} else if (country === "Germany") {
  ...
}
```

Exercise

From the following values write a conditional statement that checks if `num1` is greater than `num2`. If it is, assign "num1 is greater than num2" to the `result` variable. If it is not, assign "num1 is less than or equal to num2".

```
let num1 = 10;
let num2 = 5;
let result;

// check if num1 is greater than num2
if( condition ) {

}else {

}
```

Switch

A `switch` is a conditional statement that performs actions based on different conditions. It uses strict (`===`) comparison to match the conditions and executes the code blocks of matched condition. The syntax of the `switch` expression is shown below.

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

The expression is evaluated once and is compared with each case. If a match is found, then the associated code block is executed if not `default` code block is executed. The `break` keyword stops the execution and can be placed anywhere. In its absence, the next condition is evaluated even if the conditions are not matched.

An example of getting a weekday name based on the switch condition is shown below.

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

In multiple matching cases, the **first** matching value is selected, if not the default value is selected. In the absence of default and no matching case, the program continues to the next statement(s) after switch conditions.

Exercise

From the following values write a `switch` statement that checks the value of dayOfWeek. If dayOfWeek is "Monday", "Tuesday", "Wednesday", "Thursday", or "Friday", assign "It's a weekday" to the result variable. If `dayOfWeek` is "Saturday" or "Sunday", assign "It's the weekend" to the result.

```
let dayOfWeek = "Monday";
let result;
// check if it's a weekday or the weekend
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Comparators

Lets now focus on the conditional part:

```
if (country === "France") {
    ...
}
```

The conditional part is the variable `country` followed by the three equal signs (`===`). Three equal signs tests if the variable `country` has both the correct value (`France`) and also the correct type (`String`). You can test conditions with double equal signs, too, however a conditional such as `if (x == 5)` would then return true for both `var x = 5;` and `var x = "5";` . Depending on what your program is doing, this could make quite a difference. It is highly recommended as a best practice that you always compare equality with three equal signs (`===` and `!==`) instead of two (`==` and `!=`).

Other conditional tests:

- `x > a` : is x bigger than a?
- `x < a` : is x less than a?
- `x <= a` : is x less than or equal to a?
- `x >=a` : is x greater than or equal to a?
- `x != a` : is x not a?
- `x` : does x exist?

Logical Comparison

In order to avoid the if-else hassle, simple logical comparisons can be utilised.

```
let topper = marks > 85 ? "YES" : "NO";
```

In the above example, `?` is a logical operator. The code says that if the value of marks is greater than 85 i.e. `marks > 85` , then `topper = YES` ; otherwise `topper = NO` . Basically, if the comparison condition proves true, the first argument is accessed and if the comparison condition is false, the second argument is accessed.

Concatenate

Furthermore, you can concatenate different conditions with " `or` " or " `and` " statements, to test whether either statement is true, or both are true, respectively.

In JavaScript "or" is written as `||` and "and" is written as `&&`.

Say you want to test if the value of x is between 10 and 20. You could do that with a condition stating:

```
if (x > 10 && x < 20) {
    ...
}
```

If you want to make sure that country is either "England" or "Germany" you use:

```
if (country === "England" || country === "Germany") {
    ...
}
```

> **Note**: Just like operations on numbers, conditions can be grouped using parenthesis, ex: `if ( (name === "John" || name === "Jennifer") && country === "France")`.

Chapter 6

Arrays

Arrays are a fundamental part of programming. An array is a list of data. We can store a lot of data in one variable, which makes our code more readable and easier to understand. It also makes it much easier to perform functions on related data.

The data in arrays are called **elements**.

Here is a simple array:

```
// 1, 1, 2, 3, 5, and 8 are the elements in this array
let numbers = [1, 1, 2, 3, 5, 8];
```

Arrays can be created easily using array literals or with a `new` keyword.

```
const cars = ["Saab", "Volvo", "BMW"]; // using array literals
const cars = new Array("Saab", "Volvo", "BMW"); // using the new keyword
```

An index number is used to access the values of an array. The index of the first element in an array is always `0` as array indexes start with `0`. The index number can also be used to change the elements of an array.

```
const cars = ["Saab", "Volvo", "BMW"];
console.log(cars[0]);
// Result: Saab

cars[0] = "Opel"; // changing the first element of an array
console.log(cars);
// Result: ['Opel', 'Volvo', 'BMW']
```

> Arrays are a special type of object. One can have objects in an array.

The `length` property of an array returns the count of numbers elements. Methods supported by Arrays are shown below:

Name	Description
`concat()`	Returns two or more combined arrays
`join()`	Joins all elements in an array into a string
`push()`	Adds one or more elements at the end of the array and returns the length
`pop()`	Removes the last element of an array and returns that element
`shift()`	Removes the first element of an array and returns that element
`unshift()`	Adds one or more elements at the front of an array and returns the length
`slice()`	Extracts the section of an array and returns the new array
`at()`	Returns element at the specified index or `undefined`
`splice()`	Removes elements from an array and (optionally) replaces them, and returns the array
`reverse()`	Transposes the elements of an array and returns a reference to an array
`flat()`	Returns a new array with all sub-array elements concatenated into it recursively up to the specified depth
`sort()`	Sorts the elements of an array in place, and returns a reference to the array
`indexOf()`	Returns the index of the first match of the search element
`lastIndexOf()`	Returns the index of the last match of the search element
`forEach()`	Executes a callback in each element of an array and returns undefined
`map()`	Returns a new array with a return value from executing `callback` on every array item.
`flatMap()`	Runs `map()` followed by `flat()` of depth 1
`filter()`	Returns a new array containing the items for which `callback` returned `true`
`find()`	Returns the first item for which `callback` returned `true`
`findLast()`	Returns the last item for which `callback` returned `true`
`findIndex()`	Returns the index of the first item for which `callback` returned `true`
`findLastIndex()`	Returns the index of the last item for which `callback` returned `true`
`every()`	Returns `true` if `callback` returns `true` for every item in the array
`some()`	Returns `true` if `callback` returns `true` for at least one item in the array
`reduce()`	Uses `callback(accumulator, currentValue, currentIndex, array)` for reducing purpose and returns the final value returned by `callback` function
`reduceRight()`	Works similarly lie `reduce()` but starts with the last element

Unshift

The `unshift` method adds new elements sequentially to the start, or front of the array. It modifies the original array and returns the new length of the array. For example.

```
let array = [0, 5, 10];
array.unshift(-5);  // 4

// RESULT: array = [-5 , 0, 5, 10];
```

> The `unshift()` method overwrites the original array.

The `unshift` method takes one or more arguments, which represent the elements to be added to the beginning of the array. It adds the elements in the order they are provided, so the first element will be the first element of the array.

Here is an example of using `unshift` to add multiple elements to an array:

```
const numbers = [1, 2, 3];
const newLength = numbers.unshift(-1, 0);
console.log(numbers); // [-1, 0, 1, 2, 3]
console.log(newLength); // 5
```

Map

The `Array.prototype.map()` method iterates over an array and modifies its elements using a callback function. The callback function will then be applied to each element of the array.

Here's the syntax for using `map` .

```
let newArray = oldArray.map(function(element, index, array) {
  // element: current element being processed in the array
  // index: index of the current element being processed in the array
  // array: the array map was called upon
  // Return element to be added to newArray
});
```

For example, let's say you have an array of numbers and you want to create a new array that doubles the values of the numbers in the original array. You could do this using `map` like this.

```
const numbers = [2, 4, 6, 8];

const doubledNumbers = numbers.map(number => number * 2);

console.log(doubledNumbers);

// Result: [4, 8, 12, 16]
```

You can also use the arrow function syntax to define the function passed to `map` .

```
let doubledNumbers = numbers.map((number) => {
  return number * 2;
});
```

or

```
let doubledNumbers = numbers.map(number => number * 2);
```

The `map()` method doesn't execute function for empty elements and doesn't change the original array.

Spread

An array or object can be quickly copied into another array or object by using the Spread Operator `(...)`. It allows an iterable such as an array to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

Here are some examples of it:

```
let arr = [1, 2, 3, 4, 5];

console.log(...arr);
// Result: 1 2 3 4 5

let arr1;
arr1 = [...arr]; //copies the arr into arr1

console.log(arr1);    //Result: [1, 2, 3, 4, 5]

arr1 = [6,7];
arr = [...arr,...arr1];

console.log(arr);   //Result: [1, 2, 3, 4, 5, 6, 7]
```

The spread operator only works in modern browsers that support the feature. If you need to support older browsers, you will need to use a transpiler like Babel to convert the spread operator syntax to equivalent ES5 code.

Shift

The `shift` method deletes the first index of that array and moves all indexes to the left. It changes the original array. Here's the syntax for using `shift` :

```
array.shift();
```

For example:

```
let array = [1, 2, 3];
array.shift();

// Result: array = [2,3]
```

You can also use the `shift` method in conjunction with a loop to remove all elements from an array. Here's an example of how you might do this:

```
while (array.length > 0) {
  array.shift();
}

console.log(array); // Result: []
```

> The `shift` method only works on arrays, and not on other objects that are similar to arrays such as arguments objects or NodeList objects. If you need to shift elements from one of these types of objects, you will need to convert it to an array first using the `Array.prototype.slice()` method.

Pop

The `pop` method removes the last element from an array and returns that element. This method changes the length of the array.

Here's the syntax for using `pop`:

```
array.pop();
```

For example:

```
let arr = ["one", "two", "three", "four", "five"];
arr.pop();

console.log(arr);

// Result: ['one', 'two', 'three', 'four']
```

You can also use the `pop` method in conjunction with a loop to remove all elements from an array. Here's an example of how you might do this:

```
while (array.length > 0) {
  array.pop();
}

console.log(array); // Result: []
```

> The `pop` method only works on arrays, and not on other objects that are similar to arrays such as arguments objects or NodeList objects. If you need to pop elements from one of these types of objects, you will need to convert it to an array first using the `Array.prototype.slice()` method.

Join

The `join` method, makes an array turn into a string and joins it all together. It does not change the original array. Here's the syntax for using `join`:

```
array.join([separator]);
```

The `separator` argument is optional and specifies the character to be used to separate the elements in the resulting string. If omitted, the array elements are separated with a comma (`,`).

For example:

```
let array = ["one", "two", "three", "four"];

console.log(array.join(" "));

// Result: one two three four
```

> Any separator can be specified but the default one is a comma `(,)`.

In the above example, a space is used as a separator. You can also use `join` to convert an array-like object (such as an arguments object or a NodeList object) to a string by first converting it to an array using the `Array.prototype.slice()` method:

```
function printArguments() {
  console.log(Array.prototype.slice.call(arguments).join(', '));
}

printArguments('a', 'b', 'c'); // Result: "a, b, c"
```

Length

Arrays have a property called `length`, and it's pretty much exactly as it sounds, it's the length of the array.

```
let array = [1, 2, 3];

let l = array.length;

// Result: l = 3
```

The length property also sets the number of elements in an array. For example.

```
let fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length = 2;

console.log(fruits);
// Result: ['Banana', 'Orange']
```

You can also use the `length` property to get the last element of an array by using it as an index. For example:

```
console.log(fruits[fruits.length - 1]); // Result: Orange
```

You can also use the `length` property to add elements to the end of an array. For example:

```
fruits[fruits.length] = "Pineapple";
console.log(fruits); // Result: ['Banana', 'Orange', 'Pineapple']
```

The `length` property is automatically updated when elements are added or removed from the array.

It's also worth noting that the `length` property is not a method, so you don't need to use parentheses when accessing it. It's simply a property of the array object that you can access like any other object property.

Push

One can push certain items to an array making the last index the newly added item. It changes the length of an array and returns a new length.

Here's the syntax for using `push`:

```
array.push(element1[, ...[, elementN]]);
```

The `element1, ..., elementN` arguments represent the elements to be added to the end of the array.

For example:

```
let array = [1, 2, 3];
array.push(4);

console.log(array);

// Result: array = [1, 2, 3, 4]
```

You can also use `push` to add elements to the end of an array-like object (such as an arguments object or a NodeList object) by first converting it to an array using the `Array.prototype.slice()` method:

```
function printArguments() {
  let args = Array.prototype.slice.call(arguments);
  args.push('d', 'e', 'f');
  console.log(args);
}

printArguments('a', 'b', 'c'); // Result: ["a", "b", "c", "d", "e", "f"]
```

> **Note**: The `push` method modifies the original array. It does not create a new array.

For Each

The `forEach` method executes a provided function once for each array element. Here's the syntax for using `forEach`:

```
array.forEach(function(element, index, array) {
  // element: current element being processed in the array
  // index: index of the current element being processed in the array
  // array: the array forEach was called upon
});
```

For example, let's say you have an array of numbers and you want to print the double of each number to the console. You could do this using `forEach` like this:

```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(number) {
  console.log(number * 2);
});
```

You can also use the arrow function syntax to define the function passed to `forEach`:

```
numbers.forEach((number) => {
  console.log(number * 2);
});
```

or

```
numbers.forEach(number => console.log(number * 2));
```

The `forEach` method does not modify the original array. It simply iterates over the elements of the array and executes the provided function for each element.

> The `forEach()` method is not executed for the empty statement.

Sort

The `sort` method sorts the items of an array in a specific order (ascending or descending). By default, the `sort` method sorts the elements as strings and arranges them in ascending order based on their UTF-16 code unit values.

Here's the syntax for using `sort`:

```
array.sort([compareFunction]);
```

The `compareFunction` argument is optional and specifies a function that defines the sort order. If omitted, the elements are sorted in ascending order according to their string representation.

For example:

```
let city = ["California", "Barcelona", "Paris", "Kathmandu"];
let sortedCity = city.sort();

console.log(sortedCity);

// Result: ['Barcelona', 'California', 'Kathmandu', 'Paris']
```

Numbers can be sorted incorrectly when they are sorted. For example, "35" is bigger than "100", because "3" is bigger than "1".

To fix the sorting issue in numbers, compare functions are used. Compare functions defines sort orders and return a **negative**, **zero**, or **positive** value based on arguments, like this:

- A negative value if `a` should be sorted before `b`
- A positive value if `a` should be sorted after `b`
- `0` if `a` and `b` are equal and their order doesn't matter

```
const points = [40, 100, 1, 5, 25, 10];
points.sort((a, b) => {return a-b});

// Result: [1, 5, 10, 25, 40, 100]
```

The `sort()` method overrides the original array.

Indices

So you have your array of data elements, but what if you want to access a specific element? That is where indices come in. An **index** refers to a spot in the array. Indices logically progress one by one, but it should be noted that the first index in an array is 0, as it is in most languages. Brackets `[]` are used to signify you are referring to an index of an array.

```
// This is an array of strings
let fruits = ["apple", "banana", "pineapple", "strawberry"];

// We set the variable banana to the value of the second element of
// the fruits array. Remember that indices start at 0, so 1 is the
// second element. Result: banana = "banana"
let banana = fruits[1];
```

You can also use an array index to set the value of an element in an array:

```
let array = ['a', 'b', 'c', 'd', 'e'];
//  indices:  0    1    2    3    4
array[4] = 'f';
console.log(array); // Result: ['a', 'b', 'c', 'd', 'f']
```

Note that if you try to access or set an element using an index that is outside the bounds of the array (i.e., an index that is less than 0 or greater than or equal to the length of the array), you will get an `undefined` value.

```
console.log(array[5]); // Output: undefined
array[5] = 'g';
console.log(array); // Result: ['a', 'b', 'c', 'd', 'f', undefined, 'g']
```

Reverse

The `reverse` method can be used on any type of array, including arrays of strings, arrays of numbers, and arrays of objects. For example.

```
let users = [{
  name: "John Smith",
  age: 30
}, {
  name: "Jane Doe",
  age: 25
}];

users.reverse();

console.log(users);

// RESULT:
[{
  name: "Jane Doe",
  age: 25
}, {
  name: "John Smith",
  age: 30
}];
```

The `reverse` method transposes the elements of the calling array object in place, mutating the array, and returning a reference to the array. Here is an example of using `reverse` of an array:

```
const numbers = [1, 2, 3];
const newLength = numbers.reverse();
console.log(numbers); // [3, 2, 1]
```

Slice

The `slice` method accepts two parameters begin and end.

- begin: This parameter defines the starting index from where the portion is to be extracted. If this argument is missing then the method takes begin as 0 as it is the default start value.
- end: This parameter is the index up to which the portion is to be extracted (excluding the end index). If this argument is not defined then the array till the end is extracted as it is the default end value If the end value is greater than the length of the array, then the end value changes to the length of the array.

```
function func() {
    //Original Array
    let arr = [23, 56, 87, 32, 75, 13];
    //Extracted array
    let new_arr = arr.slice();
    console.log(arr);
    console.log(new_arr);
}
func();


// RESULT:
[23,56,87,32,75,13]
[23,56,87,32,75,13]
```

The `slice()` method copies object references to the new array. (For example, a nested array) So if the referenced object is modified, the changes are visible in the returned new array.

```
let human = {
  name: "David",
  age: 23,
};

let arr = [human, "Nepal", "Manager"];
let new_arr = arr.slice();

// original object
console.log(arr[0]); // { name: 'David', age: 23 }

// making changes to the object in new array
new_arr[0].name = "Levy";

// changes are reflected
console.log(arr[0]); // { name: 'Levy', age: 23 }
```

Chapter 7

Loops

Loops are repetitive instructions where one variable in the loop changes. Loops are handy, if you want to run the same code over and over again, each time with a different value.

Instead of writing:

```
doThing(cars[0]);
doThing(cars[1]);
doThing(cars[2]);
doThing(cars[3]);
doThing(cars[4]);
```

You can write:

```
for (var i = 0; i < cars.length; i++) {
  doThing(cars[i]);
}
```

For

The easiest form of a loop is the for statement. This one has a syntax that is similar to an if statement, but with more options:

Syntax

The syntax of `for` loop in javascript is given below

```
for (initialization; end condition; change) {
    // do it, do it now
}
```

Explanation:

- In the `initialization` part, executed before the first iteration, initialize your loop variable
- In the `end codition` part, put a condition that may be checked before each iteration. The moment the condition becomes `false`, the loop ends.
- In the `change` part, tell the program how to update the loop variable. Let's see how to execute the same code ten-times using a `for` loop:

```
for (let i = 0; i < 10; i = i + 1) {
  // do this code ten-times
}
```

> ***Note***: `i = i + 1` can be written `i++`.

`for...in` loop

To loop through the enumerable properties of an object `for in` loop can be used. For each distinct property, JavaScript executes the specified statements.

Syntax

```
for (variable in object) {
  // iterate each property in the object
}
```

Example

Let use suppose we have following object:

```
const person = {
  fname: "John",
  lname: "Doe",
  age: 25,
};
```

Then, with the help of `for in` loop we can iterate over the `person` object to access it property like `fname`, `lname` and `age` as shown below.

```
let info = "";
for (let x in person) {
  console.log(person[x]);
}
```

The output of above code snippet will be:

```
John
Doe
25
```

> **Note: The iterable objects such as `Arrays` , `Strings` , `Maps` , `NodeLists` can be looped using `for in` statement.**

```
// Example with Arrays
const myArray = [1, 2, 3, 4, 5];
for (const item of myArray) {
  console.log(item);
}

// Example with Strings
const myString = "Hello, World!";
for (const char of myString) {
  console.log(char);
}

// Example with Maps
const myMap = new Map();
myMap.set("name", "John");
myMap.set("age", 30);

for (const [key, value] of myMap) {
  console.log(key, value);
}

// Example with NodeLists (HTML elements)
const paragraphs = document.querySelectorAll("p");
for (const paragraph of paragraphs) {
  console.log(paragraph.textContent);
}
```

`for...of` loop

The `for...of` loop was introduced in the later versions of **JavaScript ES6. The `for...of` statement executes a loop that operates on a sequence of values sourced from an iterable objects such as Array, String, TypedArray, Map, Set, NodeList(and other DOM collections).

Syntax

The syntax of the `for...of` loop is:

```
for (element of iterable) {
  //body of for...of
}
```

Here,

- **iterable** - an iterable object

- **element** - items in the iterable

In plain English, you can read the above code as: *for every element in the iterable, run the body of the loop.

Examples

Let use suppose we have following object:

```
const person = ["John Doe", "Albert", "Neo"];
```

Then, with the help of `for of` loop we can iterate over the `person` object to access it individual element as shown below.

```
let info = "";
for (let x of person) {
  console.log(x);
}
```

The output of above code snippet will be:

```
John Doe
Albert
Neo
```

The use of `for...of` loop with string, maps and nodelist are given below:

```
// Example with Strings
const text = "Hello, World!";
for (const char of text) {
  console.log(char);
}

// Example with Maps
const person = new Map();
person.set("name", "John");
person.set("age", 30);
for (const [key, value] of person) {
  console.log(key, value);
}

// Example with NodeLists (HTML elements)
const paragraphs = document.querySelectorAll("p");
for (const paragraph of paragraphs) {
  console.log(paragraph.textContent);
}
```

While

While loops repetitively execute a block of code as long as a specified condition is true. It provides a way to automate repetitive tasks and perform iterations based on the condition's evaluation.

```
while (condition) {
  // do it as long as condition is true
}
```

For example, the loop in this example will repetitively execute its block of code as long as the variable i is less than 5:

```
var i = 0,
x = "";
while (i < 5) {
  x = x + "The number is " + i;
  i++;
}
```

> Be careful to avoid infinite looping if the condition is always true!

Do...While

The do...while statement creates a loop that executes specified bloc statement until the test condition evaluates to be false. The condition is evaluated after executing the block. The syntax for do... while is

```
do {
  // statement
} while (expression);
```

Lets for example see how to print numbers less than 10 using `do...while` loop:

```
var i = 0;
do {
  document.write(i + " ");
  i++; // incrementing i by 1
} while (i < 10);
```

> ***Note***: `i = i + 1` can be written `i++`.

Chapter 8

Functions

Functions are one of the most powerful and essential notions in programming. Functions like mathematical functions perform transformations, they take input values called **arguments** and **return** an output value.

Functions can be created in two ways: using `function declaration` or `function expression` . The *function name* can be omitted in `function expression` making it an `anonymous function` . Functions, like variables, must be declared. Let's declare a function `double` that accepts an *argument* called `x` and **returns** the double of x :

```
// an example of a function declaration
function double(x) {
  return 2 * x;
}
```

> *Note:* the function above **may** be referenced before it has been defined.

Functions are also values in JavaScript; they can be stored in variables (just like numbers, strings, etc ...) and given to other functions as arguments :

```
// an example of a function expression
let double = function (x) {
  return 2 * x;
};
```

> *Note:* the function above **may not** be referenced before it is defined, just like any other variable.

A callback is a function passed as an argument to another function.

An arrow function is a compact alternative to traditional functions which has some semantic differences with some limitations. These function doesn't have their own bindings to `this` , `arguments` and `super` , and cannot be used as constructors. An example of an arrow function.

```
const double =  (x) =>  2 * x;
```

The `this` keyword in the arrow function represents the object that defined the arrow function.

Higher order

Higher order functions are functions that manipulate other functions. For example, a function can take other functions as arguments and/or produce a function as its return value. Such *fancy* functional techniques are powerful constructs available to you in JavaScript and other high-level languages like python, lisp, etc.

We will now create two simple functions, `add_2` and `double` , and a higher order function called `map` . `map` will accept two arguments, `func` and `list` (its declaration will therefore begin `map(func,list)`), and return an array. `func` (the first argument) will be a function that will be applied to each of the elements in the array `list` (the second argument).

```
// Define two simple functions
let add_2 = function (x) {
  return x + 2;
};
let double = function (x) {
  return 2 * x;
};

// map is cool function that accepts 2 arguments:
//  func    the function to call
//  list    a array of values to call func on
let map = function (func, list) {
  let output = []; // output list
  for (idx in list) {
    output.push(func(list[idx]));
  }
  return output;
};

// We use map to apply a function to an entire list
// of inputs to "map" them to a list of corresponding outputs
map(add_2, [5, 6, 7]); // => [7, 8, 9]
map(double, [5, 6, 7]); // => [10, 12, 14]
```

The functions in the above example are simple. However, when passed as arguments to other functions, they can be composed in unforeseen ways to build more complex functions.

For example, if we notice that we use the invocations `map(add_2, ...)` and `map(double, ...)` very often in our code, we could decide we want to create two special-purpose list processing functions that have the desired operation baked into them. Using function composition, we could do this as follows:

```
process_add_2 = function (list) {
  return map(add_2, list);
};
process_double = function (list) {
  return map(double, list);
};
process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Now let's create a function called `buildProcessor` that takes a function `func` as input and returns a `func` -processor, that is, a function that applies `func` to each input in list.

```
// a function that generates a list processor that performs
let buildProcessor = function (func) {
  let process_func = function (list) {
    return map(func, list);
  };
  return process_func;
};
// calling buildProcessor returns a function which is called with a list input

// using buildProcessor we could generate the add_2 and double list processors as follows:
process_add_2 = buildProcessor(add_2);
process_double = buildProcessor(double);

process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Let's look at another example. We'll create a function called `buildMultiplier` that takes a number `x` as input and returns a function that multiplies its argument by `x` :

```
let buildMultiplier = function (x) {
  return function (y) {
    return x * y;
  };
};

let double = buildMultiplier(2);
let triple = buildMultiplier(3);

double(3); // => 6
triple(3); // => 9
```

Chapter 9

Objects

In javascript the objects are **mutable** because we change the values pointed by the reference object, instead, when we change a primitive value we are changing its reference which now is pointing to the new value and so primitive are **immutable**. The primitive types of JavaScript are `true` , `false` , `numbers` , `strings` , `null` and `undefined` . Every other value is an `object` . Objects contain `propertyName` : `propertyValue` pairs. There are three ways to create an `object` in JavaScript:

1. literal

   ```
   let object = {};
   // Yes, simply a pair of curly braces!
   ```

   > ***Note:*** this is the **recommended** way.

2. object-oriented

   ```
   let object = new Object();
   ```

   > ***Note:*** it's almost like Java.

3. and using `object.create`

   ```
   let object = Object.create(proto[, propertiesObject]);
   ```

   > ***Note:*** it creates a new object with the specified prototype object and properties.

Properties

Object's property is a `propertyName` : `propertyValue` pair, where **property name can be only a string**. If it's not a string, it gets casted into a string. You can specify properties **when creating** an object **or later**. There may be zero or more properties separated by commas.

```
let language = {
  name: "JavaScript",
  isSupportedByBrowsers: true,
  createdIn: 1995,
  author: {
    firstName: "Brendan",
    lastName: "Eich",
  },
  // Yes, objects can be nested!
  getAuthorFullName: function () {
    return this.author.firstName + " " + this.author.lastName;
  },
  // Yes, functions can be values too!
};
```

The following code demonstrates how to **get** a property's value.

```
let variable = language.name;
// variable now contains "JavaScript" string.
variable = language["name"];
// The lines above do the same thing. The difference is that the second one lets you use litteraly any string a
variable = language.newProperty;
// variable is now undefined, because we have not assigned this property yet.
```

The following example shows how to **add** a new property **or change** an existing one.

```
language.newProperty = "new value";
// Now the object has a new property. If the property already exists, its value will be replaced.
language["newProperty"] = "changed value";
// Once again, you can access properties both ways. The first one (dot notation) is recommended.
```

Mutable

The difference between objects and primitive values is that we can **change objects**, whereas primitive values are **immutable**.

For example:

```
let myPrimitive = "first value";
myPrimitive = "another value";
// myPrimitive now points to another string.
let myObject = { key: "first value" };
myObject.key = "another value";
// myObject points to the same object.
```

You can add, modify, or delete properties of an object using the dot notation or the square bracket notation.

```
let object = {};
object.foo = 'bar'; // Add property 'foo'
object['baz'] = 'qux'; // Add property 'baz'
object.foo = 'quux'; // Modify property 'foo'
delete object.baz; // Delete property 'baz'
```

> Primitive values (such as numbers and strings) are immutable, while objects (such as arrays and objects) are mutable.

Reference

Objects are **never copied**. They are passed around by reference. An object's reference is a value that refers to an object. When you create an object using the `new` operator or object literal syntax, JavaScript creates an object and assigns a reference to it.

Here's an example of creating an object using the object literal syntax:

```
var object = {
  foo: 'bar'
};
```

Here's an example of creating an object using the `new` operator:

```
var object = new Object();
object.foo = 'bar';
```

When you assign an object reference to a variable, the variable simply holds a reference to the object, not the object itself. This means that if you assign the object reference to another variable, both variables will point to the same object.

For example:

```
var object1 = {
  foo: 'bar'
};

var object2 = object1;

console.log(object1 === object2); // Output: true
```

In the example above, both `object1` and `object2` are variables that hold references to the same object. The `===` operator is used to compare the references, not the objects themselves, and it returns `true` because both variables hold references to the same object.

You can use the `Object.assign()` method to create a new object that is a copy of an existing object.

Following is an example of creating an object by reference.

```
// Imagine I had a pizza
let myPizza = { slices: 5 };
// And I shared it with You
let yourPizza = myPizza;
// I eat another slice
myPizza.slices = myPizza.slices - 1;
let numberOfSlicesLeft = yourPizza.slices;
// Now We have 4 slices because myPizza and yourPizza
// reference to the same pizza object.
let a = {},
  b = {},
  c = {};
// a, b, and c each refer to a
// different empty object
a = b = c = {};
// a, b, and c all refer to
// the same empty object
```

Prototype

Every object is linked to a prototype object from which it inherits properties. The objects created from object literals (`{}`) are automatically linked to `Object.prototype` , which is an object that comes standard with JavaScript.

When a JavaScript interpreter (a module in your browser) tries to find a property, that you want to retrieve, like in the following code:

```
let adult = { age: 26 },
  retrievedProperty = adult.age;
// The line above
```

First, the interpreter looks through every property the object itself has. For example, `adult` has only one own property — `age` . But besides that one, it actually has a few more properties, which were inherited from `Object.prototype.`

```
let stringRepresentation = adult.toString();
// the variable has value of '[object Object]'
```

The `toString` is an Object.prototype's property, which was inherited. It has a value of a function, which returns a string representation of the object. If you want it to return a more meaningful representation, then you can override it. Simply add a new property to the adult object.

```
adult.toString = function () {
  return "I'm " + this.age;
};
```

If you call the `toString` function now, the interpreter will find the new property in the object itself and stop.

Thus the interpreter retrieves the first property it will find on the way from the object itself and further through its prototype.

To set your own object as a prototype instead of the default Object.prototype, you can invoke `Object.create` as follows:

```
let child = Object.create(adult);
/* This way of creating objects lets us easily replace the default Object.prototype with the one we want. In th
child.age = 8;
/* Previously, child didn't have its own age property, and the interpreter had to look further to the child's p
 Now, when we set the child's own age, the interpreter will not go further.
 Note: adult's age is still 26. */
let stringRepresentation = child.toString();
// The value is "I'm 8".
/* Note: we have not overridden the child's toString property, thus the adult's method will be invoked. If adul
```

The `child` 's prototype is `adult` , whose prototype is `Object.prototype` . This sequence of prototypes is called a **prototype chain**.

Delete Operator

The `delete` operator can be used to **remove a property** from an object. When a property is deleted, it is removed from the object and cannot be accessed or enumerated (i.e., it does not show up in a for-in loop).

Here's the syntax for using `delete` :

```
delete object.property;
```

For example:

```
let adult = { age: 26 },
  child = Object.create(adult);

child.age = 8;

delete child.age;

/* Remove age property from child, revealing the age of the prototype, because then it is not overridden. */

let prototypeAge = child.age;
// 26, because child does not have its own age property.
```

> The `delete` operator only works on own properties of an object, and not on inherited properties. It also does not work on properties that have the `configurable` attribute set to `false` .

The `delete` operator does not modify the object's prototype chain. It simply removes the specified property from the object and also it does not actually destroy the object or its properties. It simply makes the properties inaccessible. If you need to destroy an object and release its memory, you can set the object to `null` or use a garbage collector to reclaim the memory.

Enumeration

Enumeration refers to the process of iterating over the properties of an object and performing a certain action for each property. There are several ways to enumerate the properties of an object in JavaScript.

One way to enumerate the properties of an object is to use the `for-in` loop. The `for-in` loop iterates over the enumerable properties of an object in an arbitrary order, and for each property it executes a given block of code.

The `for in` statement can loop over all of the property names in an object. The enumeration will include functions and prototype properties.

```
let fruit = {
    apple: 2,
    orange: 5,
    pear: 1,
  },
  sentence = "I have ",
  quantity;
for (kind in fruit) {
  quantity = fruit[kind];
  sentence += quantity + " " + kind + (quantity === 1 ? "" : "s") + ", ";
}
// The following line removes the trailing comma.
sentence = sentence.substr(0, sentence.length - 2) + ".";
// I have 2 apples, 5 oranges, 1 pear.
```

Another way to enumerate the properties of an object is to use the `Object.keys()` method, which returns an array of the object's own enumerable property names.

For example:

```
let object = {
  foo: 'bar',
  baz: 'qux'
};

let properties = Object.keys(object);
properties.forEach(function(property) {
  console.log(property + ': ' + object[property]);
});

// foo: bar
// baz: qux
```

Chapter 10

Date and Time

The `date` object stores date and time and provides methods for managing it. Date objects are static and use a browser's default timezone to display the date as a full-text string.

To create `date` we use a `new Date()` constructor and can be created in the following ways.

```
new Date()
new Date(date string)
new Date(year,month)
new Date(year,month,day)
new Date(year,month,day,hours)
new Date(year,month,day,hours,minutes)
new Date(year,month,day,hours,minutes,seconds)
new Date(year,month,day,hours,minutes,seconds,ms)
new Date(milliseconds)
```

> Months can be specified from `0` to `11`, more than that will result in an overflow to the next year.

Methods and properties supported by date are described below:

Name	Description
`constructor`	Returns function that created the Date object's prototype
`getDate()`	Returns the day (1-31) of a month
`getDay()`	Returns the day (0-6) of a week
`getFullYear()`	Returns the year (4 digits)
`getHours()`	Returns the hour (0-23)
`getMilliseconds()`	Returns the milliseconds (0-999)
`getMinutes()`	Returns the minutes (0-59)
`getMonth()`	Returns the month (0-11)
`getSeconds()`	Returns the seconds (0-59)
`getTime()`	Returns the numeric value of a specified date in milliseconds since midnight Jan 1 1970
`getTimezoneOffset()`	Returns timezone offset in minutes
`getUTCDate()`	Returns the day (1-31) of a month according to universal time
`getUTCDay()`	Returns the day (0-6) according to universal time
`getUTCFullYear()`	Returns the year (4-digits) according to universal time
`getUTCHours()`	Returns the hours (0-23) according to universal time
`getUTCMilliseconds()`	Returns the milliseconds (0-999) according to universal time
`getUTCMinutes()`	Returns the minutes (0-59) according to universal time
`getUTCMonth()`	Returns the month (0-11) according to universal time
`getUTCSeconds()`	Returns the seconds (0-59) according to universal time
`now()`	Returns the numeric value in milliseconds since midnight Jan 1, 1970
`parse()`	Parses the date string and returns the numeric value in milliseconds since midnight Jan 1, 1970
`prototype`	Allows to add properties
`setDate()`	Sets the day of a month
`setFullYear()`	Sets the year
`setHours()`	Sets the hour
`setMilliseconds()`	Sets the milliseconds
`setMinutes()`	Sets the minutes
`setMonth()`	Sets the month
`setSeconds()`	Sets the second
`setTime()`	Sets the time
`setUTCDate()`	Sets the day of the month according to universal time
`setUTCFullYear()`	Sets the year according to the universal time
`setUTCHours()`	Sets the hour according to the universal time

Name	Description
`setUTCMilliseconds()`	Sets the milliseconds according to the universal time
`setUTCMinutes()`	Sets the minutes according to the universal time
`setUTCMonth()`	Sets the month according to the universal time
`setUTCSeconds()`	Sets the second according to the universal time
`toDateString()`	Returns the date in human readable format
`toISOString()`	Returns the date according to the ISO format
`toJSON()`	Returns the date in a string, formatted as a JSON date
`toLocaleDateString()`	Returns the date in a string using locale conventions
`toLocaleTimeString()`	Returns the time in a string using locale conventions
`toLocaleString()`	Returns date using locale conventions
`toString()`	Returns string representation of the specified date
`toTimeString()`	Returns the *time* portion into a human-readable format
`toUTCString()`	Converts date into a string according to the universal format
`toUTC()`	Returns the milliseconds since midnight Jan 1 1970 in UTC format
`valueOf()`	Returns the primitive value of `Date`

Chapter 11

JSON

Java**S**cript **O**bject **N**otation (JSON) is a text-based format for storing and transporting data. The Javascript Objects can be easily converted into JSON and vice versa. For example.

```
//  a JavaScript object
let myObj = { name:"Ryan", age:30, city:"Austin" };

// converted into JSON:
let myJSON = JSON.stringify(myObj);
console.log(myJSON);
// Result: '{"name":"Ryan","age":30,"city":"Austin"}'

//converted back to JavaScript object
let originalJSON = JSON.parse(myJSON);
console.log(originalJSON);

// Result: {name: 'Ryan', age: 30, city: 'Austin'}
```

`stringify` and `parse` are the two methods supported by JSON.

Method	Description
`parse()`	Returns JavaScript object from the parsed JSON string
`stringify()`	Returns JSON string from JavaScript Object

The following data types are supported by JSON.

- string
- number
- array
- boolean
- object with valid JSON values
- `null`

It can not be `function` , `date` or `undefined` .

Chapter 12

Error Handling

Errors are an inevitable part of software development. Handling them effectively is crucial for writing robust and reliable JavaScript code. This guide will walk you through the fundamentals of error handling in JavaScript, including why it's important, types of errors, and how to use the `try...catch` statement.

Why Error Handling Matters

Error handling is essential for several reasons:

- **Graceful Recovery**: It allows your code to recover gracefully from unexpected issues and continue executing.
- **User Experience**: Effective error handling enhances the user experience by providing meaningful error messages.
- **Debugging**: Properly handled errors make debugging easier as you can pinpoint issues quickly.
- **Code Reliability**: Error handling ensures that your code is reliable and robust, reducing the risk of crashes.

Types of Errors

JavaScript errors can be categorized into several types, including:

- **Syntax Errors**: Errors that occur due to incorrect syntax.
- **Runtime Errors**: Errors that happen during code execution.
- **Logic Errors**: Errors resulting from flawed logic in the code.

Common Use Cases

Handling network requests that might fail. Parsing and validating user input. Managing third-party library errors.

Advantages of Error Handling

Effective error handling offers several advantages:

-Prevents script termination. -Allows for controlled handling of errors. -Provides detailed error information for debugging. -Enhances code reliability and user experience.

Best Practices

To make the most of error handling, consider these best practices:

-Use specific error types whenever possible. -Log errors for debugging purposes. -Provide clear and user-friendly error messages. -Handle errors as close to their source as possible.

try...catch Error Handling:

One of the common error handling techniques is the try...catch block, which is described in the following sections.

- try...catch
- try...catch...finally

Conclusion

Error handling is a critical aspect of JavaScript development. By understanding the types of errors, and following best practices, you can write more reliable and user-friendly applications.

try... catch

Instead of halting the code execution, we can use the `try...catch` construct that allows catching errors without dying the script. The `try...catch` construct has two main blocks; `try` and then `catch`.

```
try {
  // code...
} catch (err) {
  // error handling
}
```

At first, the code in the `try` block is executed. If no errors are encountered then it skips the `catch` block. If an error occurs then the `try` execution is stopped, moving the control sequence to the `catch` block. The cause of the error is captured in `err` variable.

```
try {
  // code...
  alert('Welcome to Learn JavaScript');
  asdk; // error asdk variable is not defined
} catch (err) {
  console.log("Error has occurred");
}
```

`try...catch` works for runtime errors meaning that the code must be runnable and synchronous.

To throw a custom error, a `throw` statement can be used. The error object, that gets generated by errors has two main properties.

- **name**: error name
- **message**: details about the error

If we don't need an `error` message catch can be omitted.

try...catch...finally

We can add one more construct to `try...catch` called `finally`, this code executes in all cases. i.e. after `try` when there is no error and after a `catch` in case of error. The syntax for `try ...catch...finally` is shown below.

```
try {
   // try to execute the code
} catch (err) {
    // handle errors
} finally {
   // execute always
}
```

Running real-world example code.

```
try {
  alert( 'try' );
} catch (err) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

In the above example, the `try` block is executed first which is then followed by `finally` as there are no errors.

Exercise

Write a function `divideNumbers()` that takes two arguments numerator and denominator and returns the result of dividing numerator by denominator using following settings.

```
function divideNumbers(numerator, denominator) {
    try {
      // try statement to divide numerator by denominator.
    } catch (error) {
      // print error message
    } finally {
      // print execution has finished
    }
   // return result
  }
  let answer = divideNumbers(10, 2);
```

Chapter 13

Modules

In the real world, a program grows organically to cope with the needs of new functionality. With growing codebase structuring and maintaining the code requires additional work. Though it will pay off in the future, it's tempting to neglect it and allow programs to be deeply tangled. In reality, it increases the complexity of the application, as one is forced to build a holistic understanding of the system and has difficulty to look any piece in isolation. Secondly, one has to invest more time in untangling to use its functionality.

Modules come to avoid these problems. A `module` specifies which pieces of code it depends on, along with what functionality it provides for other modules to use. Modules that are dependent on another module are called *dependencies*. Various module libraries are there to organize code into modules and load it on demand.

- AMD - one of the oldest module systems, initially used by require.js.
- CommonJS - module system created for Node.js server.
- UMD - module system that is compatible with AMD and CommonJS.

Modules can load each other, and use special directives `import` and `export` to interchange functionality, and call functions of each other.

- `export` - labels functions and variables that should be accessible from outside the current module
- `import` - imports functionality from outside module

Let's see the `import` , and `export` mechanism in modules. We have `sayHi` function exported from `sayHi.js` file.

```
// 📁 sayHi.js
export const sayHi = (user) => {
  alert(`Hello, ${user}!`);
}
```

The `sayHi` function is consumed in the `main.js` file with the help of the `import` directive.

```
// 📁 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('Kelvin'); // Hello, Kelvin!
```

Here, the import directive loads the module by importing the relative path and assigns the `sayHi` variable.

Modules can be exported in two ways: **Named** and **Default**. Furthermore, the Named exports can be assigned inline or individually.

```
// 📁 person.js

// inlined named exports
export const name = "Kelvin";
export const age = 30;

// at once
const name = "Kelvin";
const age = 30;
export {name, age};
```

> One can only have one default `export` in a file.

```
// 📁 message.js
const message = (name, age) => {
    return `${name} is ${age} years old.`;
};
export default message;
```

Based on the type of export, we can import it in two ways. The named export are constructed using curly braces whereas, default exports are not.

```
import { name, age } from "./person.js"; // named export import
import message from "./message.js"; // default export import
```

While assigning modules, we should avoid *circular dependency*. Circular dependency is a situation where module A depends on B, and B also depends on A directly or indirectly.

Chapter 14

Regular Expression

A regular expression is an object that can either be constructed with the `RegEx` constructor or written as a literal value by enclosing a pattern in a forward slash `(/)` characters. The syntaxes for creating a regular expression are shown below.

```
// using regular expression constructor
new RegExp(pattern[, flags]);

// using literals
/pattern/modifiers
```

The flags are optional while creating a regular expression using literals. Example of creating identical regular using above mentioned method is as follows.

```
let re1 = new RegExp("xyz");
let re2 = /xyz/;
```

Both ways will create a regex object and have the same methods and properties. There are cases where we might need dynamic values to create a regular expression, in that case, literals won't work and have to go with the constructor.

In cases where we want to have a forward slash to be a part of a regular expression, we have to escape the forward slash `(/)` with backslash `(\)`.

The different modifiers that are used to perform case-insensitive searches are:

- `g` - global search (finds all matches instead of stopping after the first match)

Example :

```
const str = "Hello world, hello again!";
const regex = /hello/g;
const matches = str.match(regex);
// If you are thinking about .match() Read This 👇
// It is a built-in method in JavaScript that is used to search a string for a match against an expression.
// If the match is found, It returns an array of all the matches that were found. and if not, the .match() meth

console.log(matches); // ["Hello", "hello"]
```

- `i` - case insensitive search

Example :

```
const str = "HeLlO WoRlD";
const regex = /hello/i;
const match = regex.test(str);
// The .test() method returns a boolean value:
// true if the match is found, and false if the match is not found.

console.log(match); // true
```

- `m` - multiline matching

Example :

```
const str = "This is a\nmultiline string.";
const regex = /./mg;
const matches = str.match(regex);
// the m flag is used to match newline characters (\n).
// This means that the regex will match all 26 characters in the string,
//  including the newline character.

console.log(matches.length); // 26
```

Brackets are used in a regular expression to find a range of characters. Some of them are mentioned below.

- `[abc]` - find any character between the brackets

Example :

```
const str = "The cat and the dog are both animals.";
const regex = /[abc]/g;
const matches = str.match(regex);

console.log(matches); // Array of all occurrences of a, b, and c

[
  'c', 'a', 'a',
  'a', 'b', 'a',
  'a'
]
```

- `[^abc]` - find any character, not between the brackets

Example :

```
const str = "The cat and dog.";
const regex = /[^abc]/g; // Matches any character that is not 'a', 'b', or 'c'
const matches = str.match(regex);

console.log(matches); // Array of all occurrences of characters not 'a', 'b', or 'c'

[
  'T', 'h', 'e', ' ',
  't', ' ', 'n', 'd',
  ' ', 'd', 'o', 'g',
  '.'
]
```

- `[0-9]` - find any digit between the bracket

Example :

```
const str = "The price of the item is $25, but it may change to $30.";
const regex = /[0-9]/g; // Matches any digit from 0 to 9
const matches = str.match(regex);

console.log(matches); // Array of all occurrences of digits

[
  '2', '5', '3', '0'
]
```

- `[^0-9]` - find any character, not between the brackets (non-digit)

Example :

```
const str = "The price is $25.";
const regex = /[^0-9]/g; // Matches any character that is not a digit
const matches = str.match(regex);

console.log(matches); // Array of all occurrences of non-digits

[
  'T', 'h', 'e', ' ',
  'p', 'r', 'i', 'c',
  'e', ' ', 'i', 's',
  ' ', '$', '.'
]
```

- `(x|y)` - find any of the alternatives separated by |

Example :

```
const str = "The words 'xylophone' and 'yellow' contain the letters 'x' and 'y'.";
const regex = /(x|y)/g; // Matches either 'x' or 'y'
const matches = str.match(regex);

console.log(matches); // Array of all occurrences of 'x' or 'y'

[
  'x', 'y', 'y', 'x', 'x', 'y'
]
```

Metacharacters are special character that has special meaning in the regular expression. These characters are further described below:

Metacharacter	Description
`.`	Match a single character excpet newline or a terminator
`\w`	Match a word character (alphanumeric character `[a-zA-Z0-9_]`)
`\W`	Match a non word character (same as `[^a-zA-Z0-9_]`)
`\d`	Match any digit character(same as `[0-9]`)
`\D`	Match any non digiti character
`\s`	Match a whitespace character (spaces, tabs etc)
`\S`	Match a non whitespace character
`\b`	Match at the beginning / end of a word
`\B`	Match but not at the begining / end of a word
`\0`	Match a `NULL` character
`\n`	Match a new line character
`\f`	Match a form feed character
`\r`	Match a carriage return character
`\t`	Match a tab character
`\v`	Match a tab vertical character
`\xxx`	Match a character specified by an octal number `xxx`
`\xdd`	Match a character specified by a hexadecimal number `dd`
`\udddd`	Match Unicode character specified by a hexadecimal number `dddd`

Properties and methods supported by RegEx are listed below.

Name	Description
`constructor`	Returns function that created RegExp object's protype
`global`	Checks if the `g` modifier is set
`ignoreCase`	Checks if the `i` modifier is set
`lastIndex`	Specifies the index at which to start the next match
`multiline`	Checks if the m modifier is set
`source`	Returns the text of the string
`exec()`	Test for the match and returns the first match, if no match then it returns `null`
`test()`	Test for the match and returns the `true` or `false`
`toString()`	Returns the string value of the regular exression

A `complie()` method complies the regular expression and is deprecated.

A common example of regular expression

```
let text = "The best things in life are free";
let result = /e/.exec(text); // looks for a match  of e in a string
// result: e


let helloWorldText = "Hello world!";
// Look for "Hello"
let pattern1 = /Hello/g;
let result1 = pattern1.test(helloWorldText);
// result1: true

let pattern1String = pattern1.toString();
// pattern1String : '/Hello/g'
```

A Real Life Example of Regex in Pincode Checking

```
const handleSubmit = (e) => {
  // Prevent the default form submission behavior
  e.preventDefault();

  // Define a list of valid pincodes
  const validPincodes = [
    110001, 110002, 110003, 110004, 110005, 110006, 110007, 110008, 110009,
    110010, 110011, 110012, 110013, 110014, 110015, 110016, 110017, 110018,
    110019, 110020, 110021, 110022, 110023, 110050, 110051, 110056, 110048,
    110057, 110058, 110059, 110060, 110061, 110062, 110063, 110064
  ];

  // Convert the valid pincodes to strings
  const validPincodeStrings = validPincodes.map((pincode) => String(pincode));

  // Create a regular expression pattern to match valid pincodes
  const regexPattern = new RegExp(`^(${validPincodeStrings.join("|")})$`);

  // Get the submitted pincode from the input field
  const submittedPincode = pincode; // Make sure 'pincode' is defined elsewhere

  // Check if the submitted pincode matches the valid pincode pattern
  if (regexPattern.test(submittedPincode)) {
    // Display a success message
    // ...
  } else if (submittedPincode === "") {
    // Display an error message for empty input
    // ...
  } else if (submittedPincode.length < 6) {
    // Display an error message for invalid pincode length
    // ...
  }
}
```

Chapter 15

Classes

Classes are templates for creating an object. It encapsulates data with code to work on with data. The keyword `class` is used to create a class. And a specific method called `constructor` is used for creating and initializing an object created with a `class` . An example of car class is shown below.

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age() {
    let date = new Date();
    return date.getFullYear() - this.year;
  }
}

let myCar = new Car("Toyota", 2021);
console.log(myCar.age()) // 1
```

Class must be defined before its usage.

In the class body, methods or constructors are defined and executed in `strict mode` . Syntax not adhering to the strict mode results in error.

Static

The `static` keyword defines the static methods or properties for a class. These methods and properties are called in the class itself.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
let myCar = new Car("Toyota");

console.log(myCar.hello()); // This will throw an error
console.log(Car.hello(myCar));
// Result: Hello Toyota
```

One can access the static method or property of another static method of the same class using `this` keyword.

Inheritance

The inheritance is useful for code reusability purposes as it extends existing properties and methods of a class. The `extends` keyword is used to create a class inheritance.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

let myCar = new Model("Toyota", "Camry");
console.log(myCar.show()); // I have a Camry, it is a Toyota.
```

The prototype of the parent class must be an `Object` or `null`.

The `super` method is used inside a constructor and refers to the parent class. With this, one can access the parent class properties and methods.

Access Modifiers

`public` , `private` , and `protected` are the three access modifiers used in class to control its access from the outside. By default, all members (properties, fields, methods, or functions) are publicly accessible from outside the class.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
let myCar = new Car("Toyota");
console.log(Car.hello(myCar)); // Hello Toyota
```

`private` members can access only internally within the class and cannot be accessible from outside. Private should start with `#` .

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
  #present(carname) {
    return 'I have a ' + this.carname;
  }
}
let myCar = new Car("Toyota");
console.log(myCar.#present("Camry")); // Error
console.log(Car.hello(myCar)); // Hello Toyota
```

`protected` fields are accessible only from inside the class and those extending it. These are useful for the internal interface as the inheriting class also gains access to the parent class. Protected fields with `_` .

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  _present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this._present() + ', it is a ' + this.model;
  }
}
let myCar = new Model("Toyota", "Camry");
console.log(myCar.show()) // I have a Toyota, it is a Camry
```

Chapter 16

Browser Object Model (BOM)

The browser object model lets us interact with the browser window. The `window` object represents the browser's window and is supported by all browsers.

Object `window` is the default object for a browser, so we can specify `window` or call directly all the functions.

```
window.alert("Welcome to Learn JavaScript");

alert("Welcome to Learn JavaScript")
```

In a similar fashion, we can call other properties underneath the window object such as history, screen, navigator, location, and so on.

Window

The `window` object represents the browser window and is supported by the browsers. Global variables, objects, and functions are also part of the window object.

Global **variables** are **properties** and **functions** are **methods** of the window object.

Let's take an example of the screen properties. It is used to determine the size of the browser window and is measured in pixels.

- `window.innerHeight` - the inner height of the browser window
- `window.innerWidth` - the inner width of the browser window

> ***Note***: `window.document` is same as `document.location` as the document object model(DOM) is part of window object.

Few examples of the window methods

- `window.open()` - open a new window
- `window.close()` - close the current window
- `window.moveTo()` - move the current window
- `window.resizeTo()` - resize the current window

Popup

Popups are an additional way to show information, take user confirmation, or take user input from additional documents. A popup can navigate to a new URL and send information to the opener window. **Alert box**, **Confirmation box**, and **Prompt box** are the global functions where we can show the popup information.

1. **alert()**: It displays information to the user and has an "**OK**" button to proceed.

   ```
   alert("Alert message example");
   ```

2. **confirm()**: Use as a dialog box to confirm or accept something. It has "**Ok**" and "**Cancel**" to proceed. If the user clicks "**Ok**" then it returns `true`, if click "**Cancel**" it returns `false`.

   ```
   let txt;
   if (confirm("Press a button!")) {
     txt = "You pressed OK!";
   } else {
     txt = "You pressed Cancel!";
   }
   ```

3. **prompt()**: Takes user input value with "**Ok"** and "**Cancel"** buttons. It returns `null` if the user does not provide any input value.

   ```
   //syntax
   //window.prompt("sometext","defaultText");

   let person = prompt("Please enter your name", "Harry Potter");

   if (person == null || person == "") {
     txt = "User cancelled the prompt.";
   } else {
     txt = "Hello " + person + "! How are you today?";
   }
   ```

Screen

The `screen` object contains the information about the screen on which the current window is being rendered. To access `screen` object we can use the `screen` property of `window` object.

```
window.screen
//or
screen
```

The `window.screen` object has different properties, some of them are listed here:

Property	Description
`height`	Represents the pixel height of the screen.
`left`	Represents the pixel distance of the current screen's left side.
`pixelDepth`	A read-only property that returns the bit depth of the screen.
`top`	Represents the pixel distance of the current screen's top.
`width`	Represents the pixel width of the screen.
`orientation`	Returns the screen orientation as specified in the Screen Orientation API
`availTop`	A read-only property that returns the first pixel from the top that is not taken up by system elements.
`availWidth`	A read-only property that returns the pixel width of the screen excluding system elements.
`colorDepth`	A read-only property that returns the number of bits used to represent colors.

Navigator

The `window.navigator` or `navigator` is a **read-only** property and contains different methods and functions related to the browser.

Let's look at a few examples of navigation.

1. **navigator.appName**: It gives the name of the browser application

   ```
   navigator.appName;
   // "Netscape"
   ```

   > ***Note:*** "Netscape" is the application name for IE11, Chrome, Firefox, and Safari.

2. **navigator.cookieEnabled**: Returns a boolean value based on the cookie value in the browser.

   ```
   navigator.cookieEnabled;
   //true
   ```

3. **navigator.platform**: Provides information about the browser operating system.

   ```
   navigator.platform;
   "MacIntel"
   ```

Cookies 🍪

Cookies are pieces of information that are store on a computer and can be accessed by the browser.

Communication between a web browser and the server is stateless meaning that it treats each request independently. There are cases where we need to store user information and make that information available to the browser. With cookies, information can be fetched from the computer whenever it is required.

Cookies are saved in name-value pair

```
book = Learn JavaScript
```

The `document.cookie` property is used to create, read and delete cookies. Creating cookie is pretty easy you need to provide the name and value

```javascript
document.cookie = "book=Learn JavaScript";
```

By default, a cookie gets deleted when the browser is closed. To make it persistent, we need to specify the expiry date (in UTC time).

```javascript
document.cookie = "book=Learn JavaScript; expires=Fri, 08 Jan 2022 12:00:00 UTC";
```

We can add a parameter to tell which path the cookie belongs to. By default, the cookie belongs to the current page.

```javascript
document.cookie = "book=Learn JavaScript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
```

Here is a simple example of a cookie.

```javascript
let cookies = document.cookie;
// a simple way to retrieve all cookie.

document.cookie = "book=Learn JavaScript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
// setting up a cookie
```

History

When we open a web browser and surf a web page it creates a new entry in the history stack. As we keep navigating to different pages new entries get pushed into the stack.

To access the history object we can use

```
window.history
// or
history
```

To navigate between the different history stack we can use `go()` , `forward()` and `back()` methods of **history** object.

1. **go()**: It is used to navigate the specific URL of the history stack.

   ```
   history.go(-1); // moves page backward
   history.go(0);  // refreshes the current page
   history.go(); // refreshes the current page
   history.go(1) // moves page forward
   ```

   > ***Note:*** the current page position in history stack is **0**.

2. **back()** : To navigate page backward we use `back()` method.

   ```
   history.back();
   ```

3. **forward()**: It loads the next list in the browser history. It is similar to clicking the forward button in the browser.

   ```
   history.forward();
   ```

Location

The `location` object is used to retrieve the current location (URL) of the document and provides different methods to manipulate document location. One can access the current location by

```
window.location
//or
document.location
//or
location
```

> ***Note***: `window.location` and `document.location` references the same location object.

Let's take an example of the following URL and explore the different properties of `location`

`http://localhost:3000/js/index.html?type=listing&page=2#title`

```
location.href //prints current document URL
location.protocol //prints protocol like http: or https:
location.host //prints hostname with port like localhost or localhost:3000
location.hostname //prints hostname like localhost or www.example.com
location.port //prints port number like 3000
location.pathname //prints pathname like /js/index.html
location.search //prints query string like ?type=listing&page=2
location.hash //prints fragment identifier like #title
```

Chapter 17

Events

In programming, *events* are actions or occurrences in a system that the system informs you about so you can respond to them. For example, when you click the reset button it clears the input.

Interactions from the keyboard such as keypresses need to be constantly read to catch the key's state before it's released again. Performing other time-intensive computations might cause you to miss a key press. This used to be the input handling mechanism of some primitive machines. A further step up is to use a queue, i.e. a program that periodically checks the queue for new events and reacts to it. This approach is called *polling*.

The main drawback of this approach is that it has to look at the queue every now and then, causing disruption when an event is triggered. The better mechanism for this is to notify the code when an event occurs. This is what modern browsers do by allowing us to register functions as *handlers* for specific events.

```
<p>Click me to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("clicked");
  });
</script>
```

Here, the `addEventListener` is called on the `window` object (built-in object provided by the browser) to register a handler for the whole `window`. Calling its `addEventListener` method registers the second argument to be called whenever the event described by its first argument occurs.

Event listeners are called only when the event happens in the context of the object they are registered on.

Some of the common HTML events are mentioned here.

Event	Description
`onchange`	When the user changes or modifies the value of form input
`onclick`	When the user clicks on the element
`onmouseover`	When cursor of the mouse comes over the element
`onmouseout`	When cursor of the mouse comes leaves the element
`onkeydown`	When the user press and then releases the key
`onload`	When the browser has finished the loading

It is common for handlers registered on nodes with children to also receive events from the children. For example, if a button inside a paragraph is clicked, handlers registered on the paragraph will also receive the click event. In case of the presence of handlers in both, the one at the bottom gets to go first. The event is said to *propagate* outward, from the initiating node to its parent node and on the root of the document.

The event handler can call the `stopPropagation` method on the event object to prevent handlers further up from receiving the event. This is useful in cases like, you have a button inside a clickable element and you don't want to trigger the outer element's clickable behavior from a button click.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p"),
      button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Paragraph handler.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Button handler.");
    event.stopPropagation();
  });
</script>
```

Here, the *`mousedown`* handlers are registered by both paragraph and button. Upon clicking the button, the handler for the button calls `stopPropagation` , which will prevent the handler on the paragraph from running.

Events can have a default behavior. For example, links navigate to the link's target upon click, you get navigated to the bottom of a page upon clicking the down arrow, and so on. These default behaviors can be prevented by calling a `preventDefault` method on the event object.

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

Here, the default behavior of the link upon click is prevented, i.e. navigating towards the link's target.

Chapter 18

Promise, async/await

Promises are now used by most modern APIs. It is therefore important to understand how they work and to know how to use them to optimize the code. In this chapter, we'll define in detail what promises are and how to use them in asynchronous operations.

Promise, async/await

Imagine you are a popular book writer, and you are planning to release a new book on a certain day. Readers who have an interest in this book are adding this book to their wishlist and are notified when published or even if the release day got postponed too. On the release day, everyone gets notified and can buy the book making all parties happy. This is a real-life analogy that happens in programming.

1. A "*producing code*" is something that takes time and accomplishes something. Here it's a book writer.
2. A "*consuming code*" is someone who consumes the "producing code" once it's ready. In this case, it's a "reader".
3. The linkage between the "*producing code*" and the "*consuming code*" can be called a *promise* as it assures getting the results from the "*producing code*" to the "*consuming code*".

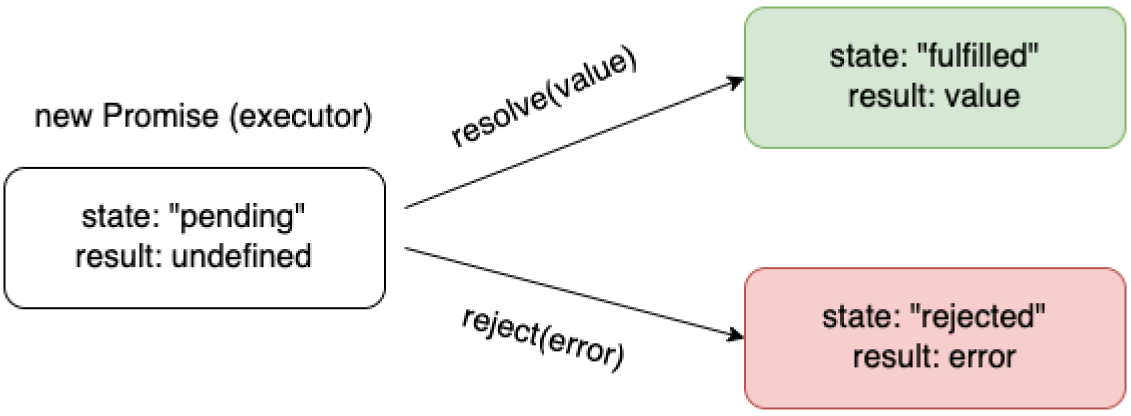
Promise

The analogy that we made is also true for the JavaScript `promise` object. The constructor syntax for the `promise` object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "writer")
});
```

Here, a function is passed to `new Promise` also known as the *executor*, and runs automatically upon creation. It contains the producing code that gives the result. `resolve` and `reject` are the arguments provided by the JavaScript itself and are called one of these upon results.

- `resolve(value):` a callback function that returns `value` upon result
- `reject(error)` : a callback function that returns `error` upon error, it returns an error object



The internal properties of `promise` object returned by the `new Promise` constructor are as follows:

- `state` - initially `pending,` then changes to either `fulfill` upon `resolve` or `rejected` when `reject` is called
- `result` - initially `undefined` , then changes to `value` upon `resolve` or `error` when `reject` is called

> One cannot access promise properties: `state` and `result` . Promise methods are needed to handle promises.

Example of a promise.

```
let promiseOne = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1-second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
})

let promiseTwo = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1-second signal that the job is done with the result "error"
  setTimeout(() => reject(new Error("Whoops!")), 1000);
})
```

Here, the `promiseOne` is an example of a "*fulfilled promise*" as it successfully resolves the values, whereas the `promiseTwo` is a "*rejected promise*" as it gets rejected. A promise that is neither rejected or resolved is called a *settled* promise, as opposed to an initially *pending* promise. Consuming function from the promise can be registered using the `.then` and `.catch` methods. We can also add `.finally` method for performing cleanup or finalizing after previous methods have been completed.

```
let promiseOne = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promiseOne.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);

let promiseTwo = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promiseTwo.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);

let promiseThree = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promiseThree.catch(alert); // shows "Error: Whoops!" after 1 second
```

In the `Promise.then()` method, both callback arguments are optional.

Async/Await

With promises, one can use a `async` keyword to declare an asynchronous function that returns a promise whereas the `await` syntax makes JavaScript wait until that promise settles and returns its value. These keywords make promises easier to write. An example of async is shown below.

```javascript
//async function f
async function f() {
  return 1;
}
// promise being resolved
f().then(alert); // 1
```

The above example can be written as follows:

```javascript
function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

`async` ensures that the function returns a promise, and wraps non-promises in it. With `await` , we can make JavaScript wait until the promise is settled with its value returned.

```javascript
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Welcome to Learn JavaScript!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)
  alert(result); // "Welcome to Learn JavaScript!"
}

f();
```

The `await` keyword can only be used inside an `async` function.

Chapter 19

Miscellaneous

In this chapter we will be discussing various topics that will come across through while writing code. The topics are listed below:

- Template Literals
- Hoisting
- Currying
- Polyfills and Transpilers
- Linked List
- Global Footprint
- Debugging
- Callback
- Web API and AJAX
- Single Thread Nature
- ECMA Script
- Building and Deploying JS application
- Testing

Template Literals

Template literals are literals delaminated with backtick `` (``) `` and are used in variable and expression interpolation into strings.

```
let text = `Hello World!`;
// template literals with both single and double code inside a single string
let text = `He's often called "Johnny"`;
// template literals with multiline strings
let text =
`The quick
brown fox
jumps over
the lazy dog`;

// template literals with variable interpolation
const firstName = "John";
const lastName = "Doe";

const welcomeText = `Welcome ${firstName}, ${lastName}!`;

// template literals with expression interpolation
const price = 10;
const VAT = 0.25;

const total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
```

Hoisting

Hosting is a default behavior in JavaScript of moving declarations at the top. While executing a code, it creates a global execution context: creation and execution. In the creation phase, JavaScript moves the variable and function declaration to the top of the page, which is known as hoisting.

```
// variable hoisting
console.log(counter);
let counter = 1; // throws ReferenceError: Cannot access 'counter' before initialization
```

Although the `counter` is present in the heap memory but hasn't been initialized so, it throws an error. This happens because of hoisting, the `counter` variable is hoisted here.

```
// function hoisting
const x = 20,
    y = 10;

let result = add(x,y); // ❌ Uncaught ReferenceError: add is not defined
console.log(result);

let add = (x, y) => x + y;
```

Here, the `add` function is hoisted and initialized with `undefined` in heap memory in the creation phase of the global execution context. Thus, throwing an error.

Currying

`Currying` is an advanced technique in functional programming that consists of transforming a function with multiple arguments into a sequence of nesting functions. It transforms a function from callable `f(a,b,c)` into callable as `f(a)(b)(c)`. It doesn't call a function instead it transforms it.

To get a better understanding of currying let's create a simple `add` function add that takes three arguments and returns the sum of them. Then, we create a `addCurry` function that takes a single input and returns a series of functions with its sum.

```
// Noncurried version
const add = (a, b, c)=>{
    return a+ b + c
}
console.log(add(2, 3, 5)) // 10

// Curried version
const addCurry = (a) => {
    return (b)=>{
        return (c)=>{
            return a+b+c
        }
    }
}
console.log(addCurry(2)(3)(5)) // 10
```

Here, we can see that both the curried and noncurried versions returned the same result. Currying can be beneficial for many reasons, some of which are mentioned here.

- It helps to avoid passing the same variable again and again.
- It divides the function into smaller chunks with a single responsibility, making the function less error-prone.
- It is used in functional programming to create a high-order function.

Polyfills and Transpilers

JavaScript evolves every now and then. Regularly, new language proposals are submitted, analyzed, and added to https://tc39.github.io/ecma262/ and then incorporated into the specification. There may be differences in how it is implemented in JavaScript engines depending on the browser. Some may implement the draft proposals, while others wait until the whole specification is released. Backward compatibility issues arise as new things are introduced.

To support the modern code in old browsers we use two tools: `transpilers` and `polyfills` .

Transpilers

It is a program that translates modern code and rewrites it using older syntax constructs so, that the older engine can understand it. For example, " `nullish` coalescing operator" `??` was introduced in 2020, and outdated browsers can't understand it.

Now, it's the transpiler's job to make the `nullish` coalescing operator" `??` understandable to the old browsers.

```
// before running the transpiler
height = height ?? 200;

// after running the transpiler
height = (height !== undefined && height !== null) ? height: 200;
```

Babel is one of the most prominent transpilers. In the development process, we can use build tools like webpack or parcel to transpile code.

Polyfills

There are times when new functionality isn't available in outdated browser engines. In this case, the code that uses the new functionality won't work. To fill the gaps, we add the missing functionality which is called a `polyfill` . For example, the `filter()` method was introduced in ES5 and is not supported in some of the old browsers. This method accepts a function and returns an array containing only the values of the original array for which the function returns `true`

```
const arr = [1, 2, 3, 4, 5, 6];
const filtered = arr.filter((e) => e % 2 === 0); // filter outs the even number
console.log(filtered);

// [2, 4, 6]
```

The polyfill for the filter is.

```javascript
Array.prototype.filter = function (callback) {
  // Store the new array
  const result = [];
  for (let i = 0; i < this.length; i++) {
    // call the callback with the current element, index, and context.
    //if it passes the test then add the element in the new array.
    if (callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }
  //return the array
  return result
}
```

caniuse shows the updated functionality and syntax supported by different browser engines.

Linked List

It is a common data structure found in all programming languages. A Linked List is very similar to a normal array in Javascript, it just acts a little bit differently.

Here each element in the list is a separate object containing a link or a pointer to the next. There is no built-in method or function for Linked Lists in Javascript so one has to implement it. An example of a linked list is shown below.

```
["one", "two", "three", "four"]
```

Types of Linked Lists

There are three different types of linked lists:

1. **Singly Linked Lists:** Each node contains only one pointer to the next node.
2. **Doubly Linked Lists:** There are two pointers at each node, one to the next node and one to the previous node.
3. **Circular Linked Lists:** A circular linked list forms a loop by having the last node pointing to the first node or any other node before it.

Add

The `add` method is created here to add value to a linked list.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    append = (value) => {
        const newNode = new Node(value)
        let current = this.head
        if (!this.head) {
            this.head = newNode
            return
        }
        while (current.next) {
            current = current.next
        }
        current.next = newNode
    }
}
```

Pop

Here, a `pop` method is created to remove a value from the linked list.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    pop = () => {
        let current = this.head
        while (current.next.next) {
            current = current.next
        }
        current.next = current.next.next
    }
}
```

Prepend

Here, a `prepend` method is created to add a value before the first child of the linked list.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    prepend = (value) => {
        const newNode = new Node(value)
        if (!this.head) {
            this.head = newNode
        }
        else {
            newNode.next = this.head
            this.head = newNode
        }
    }
}
```

Shift

Here, the `shift` method is created to remove the first element of the Linked List.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    shift = () => {
        this.head = this.head.next
    }
}
```

Global footprint

If you are developing a module, which might be running on a web page, which also runs other modules, then you must beware of the variable name overlapping.

Suppose we are developing a counter module:

```
let myCounter = {
  number: 0,
  plusPlus: function () {
    this.number = this.number + 1;
  },
  isGreaterThanTen: function () {
    return this.number > 10;
  },
};
```

> ***Note:*** This technique is often used with closures, to make the internal state immutable from the outside.

The module now takes only one variable name — `myCounter`. If any other module on the page makes use of such names like `number` or `isGreaterThanTen` then it's perfectly safe because we will not override each other's values.

Debugging

In programming, errors can occur while writing code. It could be due to syntactical or logical errors. The process of finding errors can be time-consuming and tricky and is called code debugging.

Fortunately, most modern browsers come with built-in debuggers. These debuggers can be switched on and off, forcing errors to be reported. It is also possible to set up breakpoints during the execution of code to stop execution and examine variables. For this one has to open a debugging window and place the `debugger` keyword in the JavaScript code. The code execution is stopped in each breakpoint, allowing developers to examine the JavaScript values and, resume the execution of code.

One can also use the `console.log()` method to print the JavaScript values in the debugger window.

```
const a = 5, b = 6;
const c = a + b;
console.log(c);
// Result : c = 11;
```

Browser Developer Tools

Modern browsers come equipped with powerful developer tools that aid in debugging JavaScript, inspecting HTML and CSS, and monitoring network requests. Here's a brief overview of some essential tools:

Chrome DevTools: Google Chrome's developer tools offer a wide range of features for debugging web applications.

Firefox DevTools: Mozilla Firefox's developer tools are another excellent option, offering similar capabilities.

Microsoft Edge DevTools: For Microsoft Edge users, the built-in developer tools provide essential debugging features.

Safari Web Inspector: Safari's Web Inspector is a robust toolset for debugging and profiling web applications.

Using Breakpoints

Modern browsers offer developer tools with debugging capabilities. Set breakpoints to pause code execution and inspect variables and call stacks. Step through code to understand its flow. Browser Developer Tools

Browsers provide a set of developer tools that allow you to inspect HTML, CSS, and JavaScript. You can access them by right-clicking on a web page and selecting "Inspect" or by pressing `F12` or `Ctrl+Shift+I`. Key features include:

Console: View and interact with console output.

Elements: Inspect and modify the DOM.

Sources: Debug JavaScript with breakpoints and watch expressions.

Network: Monitor network requests and responses. Using debugger Statement

Insert the debugger statement in your code to create breakpoints programmatically. When the code encounters debugger, it will pause execution and open the browser's developer tools.

Building and Deploying JavaScript Applications

Developing and deploying a JavaScript application involves a series of steps that range from setting up the development environment to deploying the application on a web server or hosting platform. The following is a detailed guide to assist individuals through this process:

Setting up the Development Environment

Before commencing the development process, it is essential for the developer to ensure that Node.js and npm (Node Package Manager) are installed on their system. These vital tools can be acquired from the official website of Node.js Node.js. Additionally, the developer should select an appropriate code editor or Integrated Development Environment (IDE) for JavaScript development. Some of the popular choices include Visual Studio Code, Sublime Text, and WebStorm.

The installation of Node.js and npm provides access to the essential tools and libraries required for JavaScript development. The careful selection of the appropriate code editor or IDE can substantially enhance productivity and code quality.

Choosing a JavaScript Framework or Library

The choice of a JavaScript framework or library is contingent upon the specific requirements of the project at hand. Developers can opt to work with well-established frameworks such as React, Angular, Vue.js, or adhere to the use of vanilla JavaScript, depending on the complexity and demands of the project. The selection is fundamentally guided by the need for structure and pre-built components that can expedite the development process and bolster maintainability.

Creating the Project

Project initiation is facilitated by the utilization of a package manager such as npm or yarn to establish a new project. For instance, the execution of the command `npm init` can be employed to set up a new Node.js project. The adoption of a package manager during project initiation ensures the establishment of a standardized project structure and streamlines the management of dependencies. This approach significantly aids in maintaining project organization and manageability.

Application Development

Throughout the process of coding the JavaScript application, the developer is advised to diligently organize modules and components. This practice is crucial for enabling ease of maintenance in the future. The development of organized and modular code is pivotal for ensuring the application remains easily maintainable and straightforward to debug. Additionally, this approach fosters code reusability and encourages collaboration among developers working on the project.

Application Testing

The developer is encouraged to create unit tests and integration tests employing testing frameworks such as Jest, Mocha, or Jasmine. This practice serves the purpose of verifying that the application functions in accordance with its intended objectives. The creation of tests serves as a proactive measure to identify and preemptively address any potential bugs, thereby instilling confidence in the reliability of the application.

Building the Application

To optimize the JavaScript code, CSS, and assets for production, it is recommended to employ a suitable build tool such as Webpack, Parcel, or Rollup. These tools bundle and optimize code and assets, leading to reduced loading times and improved performance. Furthermore, they contribute to the organization of code and facilitate the segregation of concerns within the application.

Deployment Configuration

The developer should make a well-informed decision regarding the deployment location. Deployment options encompass traditional web hosting, cloud services such as AWS or Google Cloud, or platforms like Netlify, Vercel, or GitHub Pages. The choice of deployment platform should be in alignment with the project's specific requirements and budget constraints. Different platforms offer varying levels of scalability, security, and ease of use.

Creating a Production Build

Generating a production-ready version of the application entails executing the build process. This typically involves code minification and optimization, resulting in reduced bandwidth usage and an enhanced user experience. Furthermore, a production build ensures that the application performs optimally in production environments.

Deploying the Application

The deployment process necessitates strict adherence to the instructions provided by the hosting platform. This may involve utilizing FTP, SSH, or platform-specific deployment tools. Adhering to best practices during deployment is crucial for ensuring seamless user access to the application. Deployment can be accomplished through various means, including manual uploads or automated deployment pipelines.

Domain and DNS Setup (if applicable)

For those utilizing custom domains, configuring DNS settings to direct traffic to the hosting provider or server is a requisite step. This configuration enables users to access the application through a user-friendly domain name, thereby enhancing branding and accessibility.

Continuous Integration and Deployment (CI/CD)

The developer may opt to establish a Continuous Integration and Continuous Deployment (CI/CD) pipeline. This can be achieved through the utilization of CI/CD tools such as Jenkins, Travis CI, CircleCI, or GitHub Actions. The automation of testing and deployment processes in response to code changes minimizes the potential for human error and ensures that code alterations undergo rigorous testing before reaching the production environment. This approach significantly elevates code quality and reliability.

Monitoring and Maintenance

Post-deployment, vigilance is required to monitor the application for errors, performance issues, and security vulnerabilities. Regularly updating dependencies is instrumental in enhancing security and leveraging new features. This proactive approach guarantees that the application retains its reliability, security, and performance over time.

Scaling (if necessary)

In scenarios where the application experiences growth and an increase in traffic and workload, scaling the infrastructure may become imperative. Cloud service providers offer solutions designed to accommodate such scaling requirements. These solutions enable the application to seamlessly manage heightened loads while preserving performance and availability.

Backup and Disaster Recovery (if necessary)

The implementation of backup and disaster recovery strategies is indispensable to safeguard the application's data in the event of unforeseen disruptions. These strategies are instrumental in ensuring business continuity and mitigating the risk of data loss during unexpected events.

Callback Functions in JavaScript

Callback functions are a fundamental concept in JavaScript, enabling asynchronous and event-driven programming. This Markdown document provides an in-depth explanation of callback functions, their purpose, and how to use them effectively.

What is a Callback Function?

- A **callback function** is a JavaScript function that is passed as an argument to another function.
- It is typically invoked or executed at a later time, often after some asynchronous operation or event.
- Callbacks are essential for handling tasks like data retrieval, event handling, and dealing with asynchronous behavior.

Why Use Callback Functions?

- **Asynchronous Operations**: Callbacks are crucial for managing asynchronous operations like file reading, API requests, and timers.
- **Event Handling**: They are used to respond to events like button clicks, user input, or network responses.
- **Modular Code**: Callbacks help write modular and reusable code by separating concerns and promoting the single-responsibility principle.

Anatomy of a Callback Function

A typical callback function has the following structure:

```
function callbackFunction(arg1, arg2, ..., callback) {
    // Perform some operations
    // ...

    // Call the callback function when done
    callback(result);
}
```

- **callbackFunction** is the function that takes a callback as an argument.It may perform some operations asynchronously.
- It eventually calls the **callback** function, passing it a result or an error.

Handling Errors in Callbacks

In JavaScript, callback functions can handle errors by convention. It's common to pass an error object as the first argument or use the second argument to represent errors. Developers should check for errors and handle them appropriately within the callback function.

Alternative Approaches to Callbacks

1. **Promises**: Promises offer a structured way to handle asynchronous code and errors. They have three states: pending, fulfilled, and rejected. Promises use the `.then()` and `.catch()` methods to handle success and error scenarios.
2. **Async/Await**: Async/await is a more recent addition to JavaScript. It simplifies asynchronous code by allowing developers to write it in a more synchronous style. It's built on top of Promises and is especially useful for handling asynchronous operations with a more linear code flow.
3. **Event Emitters**: In Node.js, the `EventEmitter` class allows you to create custom event-driven architectures for handling asynchronous tasks.

Callback Hell (Callback Pyramid) and Example

Callback hell, also known as the "pyramid of doom," is a common issue in JavaScript when working with deeply nested callback functions. This phenomenon occurs when multiple asynchronous operations are chained one after the other, making the code difficult to read and maintain. This Markdown document explains callback hell and provides a simple example.

What is Callback Hell?

- **Callback hell** occurs when asynchronous functions are nested within each other, leading to deeply indented code structures.
- It makes code harder to understand, debug, and maintain due to excessive indentation levels.
- Callback hell often results from handling multiple asynchronous operations sequentially, such as making API requests or reading/writing files.

Example of Callback Hell

```
asyncOperation1(function (result1) {
    // Callback 1
    asyncOperation2(result1, function (result2) {
        // Callback 2
        asyncOperation3(result2, function (result3) {
            // Callback 3
            asyncOperation4(result3, function (result4) {
                // Callback 4
                asyncOperation5(result4, function (result5) {
                    // Callback 5
                    // ... and so on
                });
            });
        });
    });
});
```

Problems with Callback Hell

- **Readability**: Callback hell leads to deeply indented code, making it challenging to read and understand. This can hinder code reviews and collaboration.
- **Maintainability**: As more asynchronous operations are added, callback hell makes the codebase difficult to maintain. Modifying existing functionality or adding new features becomes error-prone.

- **Error Handling**: Managing errors becomes complex in nested callbacks. Handling exceptions and propagating errors to higher levels can be challenging.

Mitigating Callback Hell

1. Named Functions

- Break down callback functions into separate, named functions. This improves code readability by giving meaningful names to individual functions.

2. Promises

- Promises provide a more structured way to handle asynchronous code. They allow you to chain asynchronous operations, making the code more linear and easier to read.

3. Async/Await

- Async/await is a more recent addition to JavaScript. It simplifies asynchronous code by allowing you to write it in a more synchronous style. It is built on top of Promises and is especially useful for handling asynchronous operations with a more linear code flow.

4. Modularization

- Organize code into smaller, reusable modules. This reduces the complexity of individual functions and makes it easier to manage asynchronous operations.

Conclusion

Effective error handling is crucial in asynchronous programming. Callbacks can handle errors by convention, but alternative approaches like Promises, async/await, and event emitters provide more structured and readable ways to manage asynchronous code. The choice of which approach to use depends on the specific requirements and coding style preferences. Callback hell is a common issue in JavaScript when working with deeply nested callback functions for handling asynchronous operations. It can lead to code that is challenging to read, maintain, and debug. Mitigation strategies, such as using named functions, Promises, async/await, or modularization, can significantly improve code structure and readability when dealing with asynchronous tasks, making your code more maintainable and error-resistant.

Web API and AJAX

In this section, we will discuss API and AJAX, the two important technologies that enable applications to interact with servers and send or retrieve data without the need for a full page reload.

API (Application Programming Interface)

An **API** (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other. It defines the methods and data formats that applications can use to request and exchange information.

Key Points

- APIs enable developers to access the functionality of other software components, services, or platforms without needing to understand their internal workings.
- APIs provide a way for applications to send requests and receive responses, allowing them to interact and share data seamlessly.
- Common types of APIs include **web APIs** that allow web applications to communicate over the internet, **library APIs** that provide reusable code functions, and **operating system APIs** that enable interaction with the underlying OS.
- APIs are essential for creating integrations, building software on top of existing platforms, and enabling interoperability between different systems.
- API documentation, often provided by developers or service providers, explains how to use an API, including available endpoints, request methods, and response formats.
- Examples of popular APIs include social media APIs (e.g., Facebook Graph API), payment gateway APIs (e.g., PayPal API), and cloud service APIs (e.g., AWS API).

Benefits of APIs

- **Interoperability:** APIs allow different software systems to work together, promoting compatibility and data exchange.
- **Efficiency:** Developers can leverage existing APIs to save time and effort, focusing on building specific functionality.
- **Scalability:** APIs enable the expansion of services and features by integrating with third-party tools and services.
- **Innovation:** APIs foster innovation by opening up opportunities for developers to create new applications and services.
- **Security:** APIs often include authentication and authorization mechanisms to ensure secure access to data and services.

AJAX (Asynchronous JavaScript and XML)

AJAX, short for **Asynchronous JavaScript and XML**, is a foundational technology in web development. It enables web applications to make asynchronous requests to a server, retrieve data, and update parts of a web page without requiring a full page reload. While the name suggests XML, AJAX can work with various data formats, with JSON being the most common.

What is AJAX?

At its core, AJAX is a technique that allows web pages to communicate with a server in the background, without disrupting the user's interaction with the page. This asynchronous behavior is achieved using JavaScript and enables the development of more interactive and responsive web applications.

How does AJAX work?

1. **JavaScript**: AJAX heavily relies on JavaScript to initiate requests and handle responses asynchronously.
2. **XMLHttpRequest (XHR) Object**: While historically the `XMLHttpRequest` object was used, modern web development often employs the `fetch` API for AJAX requests, which provides a more intuitive and flexible approach.
3. **Server Communication**: When a user triggers an event, such as clicking a button, JavaScript sends an HTTP request to the server. These requests can be GET (for fetching data) or POST (for sending data to the server).
4. **Asynchronous Processing**: AJAX requests are asynchronous, meaning that the browser can continue executing other code while waiting for the response. This prevents the user interface from freezing.
5. **Response Handling**: Once the server processes the request, it sends a response back to the client. JavaScript then handles this response, typically by updating the Document Object Model (DOM) with the new data.
6. **Rendering**: The updated content is rendered on the web page without requiring a full page reload, resulting in a smoother user experience.

Benefits of AJAX

- **Improved User Experience**: AJAX allows web applications to load and display data without the need for full page refreshes, making the user experience more seamless and interactive.
- **Efficiency**: AJAX requests are lightweight and only transfer the necessary data, reducing bandwidth usage and enhancing application performance.
- **Real-time Updates**: AJAX is crucial for building real-time features like chat applications, live notifications, and dynamic content updates.
- **Dynamic Loading**: Content can be loaded on-demand, leading to faster initial page load times and more responsive applications.

Common Use Cases

Some common scenarios where AJAX is used include:

- **Form Submissions**: Submitting forms without reloading the entire page for validation and data submission.

- **Infinite Scrolling**: Loading additional content as the user scrolls down a page, providing a continuous browsing experience.
- **Auto-suggestions**: Providing real-time search suggestions as users type in search queries.
- **Updating Content**: Dynamically updating content like news feeds, weather information, or sports scores without requiring manual page refreshes.

Fetching Weather Data with OpenWeatherMap API using AJAX

In this example, we'll demonstrate how to use AJAX (Asynchronous JavaScript and XML) to fetch weather information from the OpenWeatherMap API and display it on a web page.

Introduction

The OpenWeatherMap API is a powerful tool for retrieving weather information for locations around the world. This example demonstrates how to use the API to fetch current weather data for a specific city and display it in your application or documentation.

API Key

Before getting started, you need to sign up for an API key from OpenWeatherMap to access their weather data API. Replace `'YOUR_API_KEY'` in the code below with your actual API key.

```
const apiKey = 'YOUR_API_KEY';
```

Simple Weather App HTML

In this example, we'll provide the HTML structure for a simple weather app that fetches and displays weather data from the OpenWeatherMap API.

HTML Structure

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Weather App</title>
</head>
<body>
  <h1>Weather Report</h1>
  <button id="fetchButton">Fetch Data</button>
  <div id="weather-info">
    <!-- Data will be displayed here -->
  </div>
  <script src="script.js"></script>
</body>
</html>
```

JavaScript (script.js)

Create a JavaScript file named `script.js` to handle the AJAX request and update the weather data on the page:

```
// API Endpoint and Location
const apiUrl = 'https://api.openweathermap.org/data/2.5/weather';
const location = 'New York'; // Replace with your desired location

// Fetch Weather Data
const xhr = new XMLHttpRequest();
xhr.open('GET', `${apiUrl}?q=${location}&appid=${apiKey}`, true);

xhr.onload = function () {
  if (xhr.status === 200) {
    const weatherInfo = document.getElementById('weather-info');
    const data = JSON.parse(xhr.responseText);
    const temperature = (data.main.temp - 273.15).toFixed(2); // Convert from Kelvin to Celsius

    const html = `
      <p><strong>Location:</strong> ${data.name}, ${data.sys.country}</p>
      <p><strong>Temperature:</strong> ${temperature} °C</p>
      <p><strong>Weather:</strong> ${data.weather[0].description}</p>
    `;

    weatherInfo.innerHTML = html;
  } else {
    const weatherInfo = document.getElementById('weather-info');
    weatherInfo.innerHTML = '<p>Failed to fetch weather data.</p>';
  }
};

xhr.send();
```

Result

When you open the HTML file in a web browser, it will display the weather information for the specified location (New York in this case). Make sure to replace `'YOUR_API_KEY'` with your actual OpenWeatherMap API key.

This example demonstrates how to fetch weather data from the OpenWeatherMap API using AJAX and display it on a simple web page.

Remember to host your HTML and JavaScript files on a web server if you plan to access the API from a live website.

That's it! You've successfully fetched and displayed weather data using the OpenWeatherMap API and AJAX.

Conclusion

APIs play a crucial role in modern software development by enabling applications to collaborate and share data effectively. Understanding how to use APIs and integrate them into your projects is fundamental for building feature-rich and interconnected software.

AJAX is a foundational technology in modern web development that empowers developers to create dynamic and responsive web applications. While the name suggests XML, AJAX is compatible with various data formats, making it a versatile tool for enhancing the user experience and creating interactive web applications.

Single-Threaded Nature of JavaScript

JavaScript is a single-threaded programming language, executing code sequentially in one main thread. It relies on non-blocking asynchronous patterns to handle tasks efficiently without blocking the main thread, ensuring responsiveness in web applications. While simplifying concurrency, it requires effective use of callbacks and event-driven programming.

Understanding Single-Threaded JavaScript

Here are some key points to understand about JavaScript's single-threaded execution:

1. **One Thread, One Task:** JavaScript operates within a single execution thread, which means it can perform only one task at a time. This thread is often referred to as the "main thread" or the "event loop."
2. **Blocking vs. Non-Blocking:** JavaScript code is inherently non-blocking. This means that when a time-consuming operation (such as a network request or file read) is encountered, JavaScript does not wait for it to complete. Instead, it delegates the task to another part of the environment (e.g., the browser or Node.js runtime) and continues executing other code.
3. **Asynchronous Programming:** To handle potentially time-consuming operations without blocking the main thread, JavaScript relies heavily on asynchronous programming patterns. Functions like callbacks, Promises, and async/await allow developers to work with asynchronous operations effectively.
4. **Event-Driven:** JavaScript is often described as "event-driven." This means that it listens for and responds to events, such as user interactions (clicks, keystrokes), timers, or network responses. When an event occurs, a corresponding callback function is executed.
5. **Concurrency Model:** While JavaScript runs in a single thread, the concurrency model enables concurrent execution of code. This is achieved through mechanisms like the event loop, which manages the execution of asynchronous tasks in a way that ensures responsiveness and non-blocking behavior.
6. **Browser and Environment Interaction:** In web development, JavaScript interacts with the browser's Document Object Model (DOM) and other browser APIs. To maintain a responsive user interface, JavaScript code must execute quickly and efficiently and delegate time-consuming operations to separate threads when necessary.

Single-Threaded Asynchronous Example

```
// Simulating an asynchronous operation with a callback
function simulateAsyncOperation(callback) {
  setTimeout(function () {
    console.log("Async operation completed.");
    callback();
  }, 2000); // Simulating a 2-second delay
}

console.log("Start of the program");

// Initiating an asynchronous operation
simulateAsyncOperation(function () {
  console.log("Callback executed: Handling the result.");
});

console.log("End of the program");
```

In this example, we demonstrate the single-threaded nature of JavaScript and how it handles asynchronous operations using callbacks.

Code Explanation:

- We define a function `simulateAsyncOperation` that simulates an asynchronous operation using `setTimeout`. This function takes a callback as an argument, which will be executed when the asynchronous operation is completed.
- We start the program by logging "Start of the program."
- We initiate the asynchronous operation using `simulateAsyncOperation`, passing in a callback function. This function will be executed after the 2-second delay.
- Immediately after initiating the asynchronous operation, we log "End of the program."

Execution Flow:

- When you run this code, you'll notice that even though the asynchronous operation takes 2 seconds to complete, the program does not block. The "End of the program" message is logged immediately after initiating the asynchronous operation, demonstrating JavaScript's non-blocking behavior.
- After the 2-second delay, the "Async operation completed." message is logged, followed by "Callback executed: Handling the result," indicating that the callback function was executed when the asynchronous operation finished.

Key Takeaways:

- JavaScript operates in a single thread, and asynchronous operations are handled through callbacks.
- The single-threaded nature allows JavaScript to remain responsive even during time-consuming tasks.
- Callbacks are a fundamental mechanism for working with asynchronous code in JavaScript.

Benefits and Challenges

Benefits:

- Simplicity: Single-threaded execution simplifies the programming model and reduces the risk of complex concurrency-related bugs.

- Predictability: The single-threaded nature makes it easier to reason about the order of execution and the state of your program.

Challenges:

- Blocking Operations: Long-running operations can potentially block the main thread, leading to a poor user experience, especially in web applications.
- Callback Hell: Excessive use of callbacks (often referred to as "callback hell") can make the code harder to read and maintain.
- Concurrency Bottleneck: CPU-bound tasks cannot fully utilize multi-core processors because JavaScript runs in a single thread.

In summary, JavaScript's single-threaded nature is a defining feature of the language. While it simplifies certain aspects of programming, it also presents challenges in terms of handling asynchronous tasks and ensuring responsive applications. Effective use of asynchronous patterns and understanding the event-driven model are essential for JavaScript developers.

ECMAScript (ES)

ECMAScript, commonly abbreviated as ES, is a standardized scripting language specification. It serves as the foundation for several programming languages, with JavaScript being the most well-known and widely used implementation. This Markdown document provides an overview of ECMAScript, its history, features, and its role in web development.

What is ECMAScript?

- **Standardized Language**: ECMAScript is a standardized scripting language specification maintained by ECMA International. It defines the syntax and semantics of the language to ensure consistency and interoperability.
- **JavaScript Implementation**: JavaScript is the most prominent implementation of ECMAScript, but other languages like ActionScript also use this specification as a foundation.

History of ECMAScript

- **ES1 (ECMAScript 1)**: Released in 1997, ES1 laid the foundation for JavaScript as we know it today.
- **ES3 (ECMAScript 3)**: Released in 1999, ES3 introduced significant improvements and is considered the version that brought JavaScript into mainstream web development.
- **ES5 (ECMAScript 5)**: Released in 2009, ES5 added new features and improved existing ones, making JavaScript more robust.
- **ES6 (ECMAScript 2015)**: Released in 2015, ES6 was a major milestone, introducing significant changes such as arrow functions, classes, modules, and more.
- **ESNext**: Refers to the ongoing development of ECMAScript, where new features and improvements are continually proposed and added.

Why ECMAScript (ES) is Standardized for JavaScript

This section of the document elaborates on why ECMAScript is crucial for JavaScript, its role in standardization, and its benefits for the language.

The Need for Standardization

- **Language Consistency**: JavaScript, as a widely used programming language for web development, needed a standardized specification to ensure consistency across various implementations and environments.
- **Interoperability**: Different web browsers and engines may have their own interpretations of JavaScript. A standard helps ensure that JavaScript code behaves consistently across all platforms.

The Role of ECMAScript

- **Defining the Language**: ECMAScript defines the core features of JavaScript, including its syntax, data types, functions, and fundamental objects.

- **Standardization Body**: ECMAScript is maintained and developed by ECMA International (European Computer Manufacturers Association), a standards organization. This organization ensures that JavaScript remains a well-defined and stable language.
- **Version Evolution**: ECMAScript introduces new language features and improvements in each new version, keeping JavaScript up-to-date with the needs of modern web development.

Benefits of ECMAScript Standardization

- **Consistency**: Standardization ensures that JavaScript behaves consistently across different platforms and browsers, reducing compatibility issues.
- **Interoperability**: Developers can write JavaScript code with confidence, knowing that it will work as intended across various environments.
- **Innovation**: ECMAScript's ongoing development allows for the introduction of new language features and improvements, enabling JavaScript to evolve with the ever-changing web landscape.
- **Cross-Platform Development**: Standardization makes it easier for developers to write code that works on both client and server-side environments.

JavaScript Implementations

- **Major Browsers**: Popular web browsers like Chrome, Firefox, Safari, and Edge all implement ECMAScript to execute JavaScript code.
- **Node.js**: Node.js, a server-side JavaScript runtime, also adheres to ECMAScript standards, enabling JavaScript to be used for server-side programming.

Key Features of ECMAScript

- **Arrow Functions**: Provide concise syntax for defining functions and lexical binding of `this` .
- **Classes**: Introduced class syntax for object-oriented programming.
- **Modules**: Added native support for module imports and exports.
- **Promises**: Introduced Promises for improved handling of asynchronous operations.
- **Async/Await**: Simplified asynchronous code with the introduction of async functions.
- **let and const**: Block-scoped variables with `let` and constants with `const` .
- **Destructuring**: Allows for easy extraction of values from arrays and objects.
- **Template Literals**: Introduced template literals for more flexible string interpolation.

The Role of ECMAScript in Web Development

- **Client-Side Scripting**: ECMAScript is the foundation for client-side scripting in web development. It powers interactive web applications.
- **Compatibility**: While modern browsers support the latest ECMAScript features, developers need to consider backward compatibility for older browsers.

- **Transpilers**: Tools like Babel can transpile newer ECMAScript code into older versions for wider browser support.
- **TypeScript**: TypeScript, a superset of ECMAScript, adds static typing for enhanced tooling and code safety.

Conclusion

ECMAScript is a fundamental part of web development, shaping how we create dynamic and interactive web applications. Staying informed about the latest ECMAScript features is essential for modern JavaScript development. ECMAScript plays a crucial role in providing a standardized foundation for JavaScript, ensuring consistency, interoperability, and continuous improvement of the language. This standardization allows developers to write JavaScript code with confidence, knowing that it will work reliably across different platforms and environments.

Unit Testing

Unit testing is a fundamental practice in web development. It involves testing individual components or functions to ensure they work as expected. This practice can catch bugs early, improve code quality, and make refactoring safer Unit testing is essential for the following reasons:

- It verifies that individual parts of your codebase are working correctly.
- It provides a safety net when refactoring or making changes.
- It helps document the expected behavior of functions and components.

Testing Frameworks

Testing frameworks streamline the process of writing and running tests. Two popular frameworks are Jest and Mocha.

Jest

Jest is a zero-config, all-in-one popular testing framework. It's suitable for both unit and integration testing. Let's see how to get started with Jest.

Install Jest using npm or yarn:

```
npm install --save-dev jest
```

Create a test file (e.g., `myFunction.test.js`) for the function you want to test.

Write a test case using Jest's test function:

```
const myFunction = require('./myFunction');

test('should return the sum of two numbers', () => {
  expect(myFunction(2, 3)).toBe(5);
});
```

Run tests using the jest command:

```
npx jest
```

Mocha

Mocha is a flexible testing framework. It provides the structure for running tests but requires additional libraries for assertions and mocking.

Getting Started with Mocha

Install Mocha and an assertion library like Chai:

```
npm install --save-dev mocha chai
```

Create a `test` directory and add your test files.

Write your test cases using Mocha's describe and it functions and Chai's assertion functions.

```
const chai = require('chai');
const expect = chai.expect;
const myFunction = require('./myFunction');

describe('myFunction', () => {
  it('should return the sum of two numbers', () => {
    expect(myFunction(2, 3)).to.equal(5);
  });
});
```

Conclusion

In this chapter, we've explored the fundamentals of testing in web development and discussed the significance of Unit testing and other testing frameworks and tools that are vital for any web developer. With consistent practice and access to the right set of tools, one can write dependable code and ensure that applications perform optimally.

Chapter 20

Server Side Code

Server-side code refers to the code that runs on a *web server* rather than in a user's web browser. It is responsible for processing requests from clients (typically web browsers) and generating dynamic web pages or providing data to the client.

Client-side code refers to the code that runs in a user's *web browser* rather than on a web server. It is responsible for generating the user interface and handling user interactions. Client-side code is typically written in JavaScript and is executed by the browser.

Why do we need server-side code?

Server-side code is essential in web development for several reasons:

- **Security**: Server-side code is not visible to the user, so it is more secure than client-side code.
- **Performance**: Server-side code can be used to perform computationally expensive tasks, such as data processing, without affecting the user's experience.
- **Data Storage**: Server-side code can be used to store data in a database, which can then be accessed by the client-side code.
- **User Authentication**: Server-side code can be used to authenticate users and restrict access to certain parts of the website.
- **Dynamic Content**: Server-side code can be used to generate dynamic web pages, which can be customized for each user.

Server-side vs. Client-side

The differences are summarized in the table below:

Server-side Code	Client-side Code
Runs on a web server	Runs in a web browser
Has access to server resources (file system, databases, etc.).	Has access to client resources (cookies, local storage, etc.).
Can be written in a variety of languages (PHP, Python, Ruby, Java, C#, etc.).	Can only be written in JavaScript.
May use server-side rendering (SSR) to generate HTML on the server.	Uses client-side rendering (CSR) to generate HTML in the browser.
Better for SEO as content is readily available for search engines.	Worse for SEO as content is not readily available for search engines.
Can leverage caching and Content Delivery Networks (CDNs) for performance.	Limited control over caching, relies on browser cache.

Why use JavaScript for server-side code?

Unlike client-side code, which can only be written in JavaScript, server-side code can be written in a variety of languages, including PHP, Python, Ruby, Java, C#, and many more. So why use JavaScript for server-side code? There are several reasons:

- **Unified Language**: Developers can use the same language and programming paradigms throughout the entire application stack, which can lead to code reusability and easier collaboration among front-end and back-end developers.
- **Large Ecosystem**: JavaScript has a vast ecosystem of libraries and packages available through npm (Node Package Manager). This rich ecosystem simplifies the development process by providing pre-built modules for various functionalities, from server routing to database connectivity.
- **JSON**: JavaScript Object Notation (JSON) is a popular data format that is used to transmit data between a server and a web application. JSON is based on JavaScript, so it is easy to work with JSON data in JavaScript.

Up next, we will learn how to use JavaScript for server-side code with Node.js and how to use Server Side Rendering (SSR) to generate HTML on the server.

Node.js

Node.js is a JavaScript runtime environment that allows developers to run JavaScript code outside of a web browser. It is built on top of the V8 JavaScript engine, which is the same engine used by Google Chrome. Node.js is open-source and cross-platform, which means it can run on Windows, macOS, and Linux.

Node.js is not a programming language

Many people mistakenly believe that Node.js is a programming language. This is not true. Node.js is a JavaScript runtime environment, which means it provides an environment for JavaScript code to run. It is not a programming language itself.

Node.js is **not** the only JavaScript runtime environment. There are many others, including Deno, Nashorn, and most recently, Bun. But Node.js is by far the most popular and widely used JavaScript runtime environment.

Getting started with Node.js

To get started with Node.js, you will need to install it on your computer. You can download the latest version of Node.js from the official website at nodejs.org. Once you have downloaded and installed Node.js, you can verify the installation by running the following command in your terminal:

```
node --version
```

This should print the version number of Node.js like this:

```
v20.7.0
```

Writing your first Node.js program

Now that you have installed Node.js, let's write our first Node.js program. Create a new file called `hello.js` and add the following code:

```
console.log('Hello World!');
```

To run this program, open your terminal and navigate to the directory where you saved the `hello.js` file. Then run the following command:

```
node hello.js
```

This should print the following output:

```
Hello World!
```

Writing a simple web server using Express and Node.js

Express is a popular web framework for Node.js. It provides a simple and elegant API for building web applications. Let's use Express to create a simple web server that will respond to HTTP requests with a "Hello World!" message.

First, we need to install Express. To do this, run the following command in your terminal:

```
npm install express
```

This will install Express and all its dependencies. Once the installation is complete, create a new file called `server.js` and add the following code:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () => {
  console.log('Server is listening on port 3000');
});
```

This code creates a new Express application and defines a route for the root path (`/`). When a request is made to this route, the server will respond with a "Hello World!" message.

To run this program, open your terminal and navigate to the directory where you saved the `server.js` file. Then run the following command:

```
node server.js
```

This should print the following output:

```
Server is listening on port 3000
```

Now open your web browser and go to http://localhost:3000. You should see a "Hello World!" message.

Server Side Rendering (SSR)

Normally, when a user visits a website, the browser sends a request to the server, which responds with HTML, CSS, and JavaScript. But with libraries like *React* and *Vue*, the server only sends a blank HTML page along with a JavaScript file. The JavaScript file then renders the page in the browser. This is called **Client Side Rendering (CSR)**.

Server Side Rendering (SSR) is a technique where the server processes the request and generates the HTML on the server from the React or Vue components. The server then sends the generated HTML to the browser, which can then render the page without having to wait for the JavaScript to load.

Why use SSR?

There are several advantages to using SSR over CSR:

- **Better for SEO**: Search engines can crawl and index the content of your website more easily if it is rendered on the server. This can lead to better search engine rankings and more traffic from search engines.
- **Faster initial page load**: Since the HTML is generated on the server, the browser does not have to wait for JavaScript to load before rendering the page. This can lead to a faster initial page load time.
- **Better performance on low-end devices**: Since the HTML is generated on the server, the browser does not have to do as much work to render the page. This can lead to better performance on low-end devices, such as mobile phones and tablets.

Disadvantages of SSR

There are also some disadvantages to using SSR:

- **More complex development process**: SSR requires more work on the server side, which can make the development process more complex.
- **More server resources**: SSR requires more server resources, which can lead to higher hosting costs.
- **Limited client-side functionality**: SSR does not allow you to use client-side libraries, such as jQuery or Bootstrap, since they are not available on the server.

How to implement SSR?

Each library has its own way of implementing SSR. For example, for React, you can use Next.js or Gatsby. For Vue, you can use Nuxt.js. For Svelte you can use SvelteKit.

Conclusion

In this chapter, we learned about Server Side Rendering (SSR) and how it can improve the performance of your website. We also learned about the advantages of using SSR over CSR and how to implement SSR with React, Vue, and Svelte.

Chapter 21

Exercises

In this chapter we will be performing exercises to test our knowledge in JavaScript. The exercises that we will be performing are listed below:

- Console
- Multiplication
- User Input Variables
- Constants
- Concatenation
- Functions
- Conditional Statements
- Objects
- FizzBuzz Problem
- Get the Titles!

Console

In JavaScript, we use `console.log()` to write a message (the content of a variable, a given string, etc.) in `console`. It is mainly used for debugging purposes, ideally to leave a trace of the content of variables during the execution of a program.

Example:

```
console.log("Welcome to Learn JavaScript Beginners Edition");
let age = 30;
console.log(age);
```

📝 Tasks:

- [ ] Write a program to print `Hello World` on the console. Feel free to try other things as well!
- [ ] Write a program to print variables to the `console`.
  1. Declare a variable `animal` and assign the dragon value to it.
  2. Print the `animal` variable to the `console`.

💡 Hints:

- Visit the variable chapter to understand more about variables.

Multiplication

In JavaScript, we can perform the multiplication of two numbers by using the asterisk `(*)` arithmetic operators.

Example:

```
let resultingValue = 3 * 2;
```

Here, we stored the product of `3 * 2` into a `resultingValue` variable.

📝 Tasks:

- [ ] Write a program to store the product of `23` times `41` and print its value.
- [ ] Write a program that generates a multiplication table for a specific number. The program should take a number as input and then display the multiplication table for that number, from 1 to 10.

💡 Hints:

- Visit the Basic Operators chapter to understand the mathematical operations.

User Input Variables

In JavaScript, we can take input from users and use it as a variable. One doesn't need to know their value to work with them.

📝 Tasks:

- [ ] Write a program to take input from a user and add `10` to it, and print its result.

💡 Hints:

- The content of a variable is determined by the user's inputs. The `prompt()` method saves the input value as a string.
- You will need to make sure that the string value is converted into an integer for calculations.
- Visit the Basic Operators chapter for the type conversion of `string` to `int` .

Constants

Constants were introduced in ES6(2015), and use a `const` keyword. Variables that are declared with `const` cannot be reassigned or redeclared.

Example:

```
const VERSION = "1.2";
```

📝 Task:

- [ ] Run the program mentioned below and fix the error that you see in the console. Make sure that the code result is `0.9` when it is fixed in the console.

  ```
  const VERSION = "0.7";
  VERSION = "0.9";
  console.log(VERSION);
  ```

- [ ] Write a program that prompts user to enter a temperature in *degrees Celsius*, then use constant `CONVERSION_FACTOR` which equals to `9/5` to convert to *degrees Fahrenheit*.

  ```
  const CONVERSION_FACTOR = 9 / 5;

  /* Start your code from here*/
  ```

💡 Hints:

- Visit the Variables chapter for more info about const and also look for "*TypeError assignment to constant variable*" in search engines to learn a fix.

Concatenation

In any programming language, string concatenation simply means appending one or more strings to another string. For example, when strings "*World*" and "*Good Afternoon*" are concatenated with string "*Hello*", they form the string "*Hello World, Good Afternoon*". We can concatenate a string in several ways in JavaScript.

Example:

```
const icon = "👋";

// using template Strings
`hi ${icon}`;

// using join() Method
["hi", icon].join(" ");

// using concat() Method
"".concat("hi ", icon);

//  using + operator
"hi " + icon;

// RESULT
// hi 👋
```

📝 Task:

- [ ] Write a program to set the values for `str1` and `str2` so the code prints '*Hello World*' to the console.
- [ ] Write a program that prompt user to enter their first name(`first_name`) and last name(`last_name`). Then, use string concatenation to create and display their full name(`full_name`).
- [ ] Write a program that prompt user to enter their name. Then, use string concatenation to create a greeting message that include their name. For examples: `Good morning, Aman` .

💡 Hints:

- Visit the concatenation chapter of strings for more info about string concatenation.

Functions

A function is a block of code designed to perform a specific task and executed when "something" invokes it. More info about functions can be found in the functions chapter.

📝 Task:

- [ ] Write a program to create a function named `isOdd` that passes a number `45345` as an argument and determines whether the number is odd or not.
- [ ] Call this function to get the result. The result should be in a boolean format and should return `true` in `console`.
- [ ] Write a program to create a function named `calculateRectangleArea` that takes two parameters `width` and `height` of the rectange and should return `area` of the rectangle.

💡 Hints:

- Visit the functions chapter to understand functions and how to create them.

Conditional Statements

Conditional logic is vital in programming as it makes sure that the program works regardless of what data you throw in and also allows branching. This exercise is about the implementation of various conditional logic in real-life problems.

📝 Task:

- [ ] Write a program to create a prompt "*How many km is left to go?*" and based on the user and the following conditions, print the results in the `console`.
    - If there are more than 100 km left to go, print: `"You still have a bit of driving left to go"`.
    - If there are more than 50 km, but less or equal to 100 km, print: `"I'll be there in 5 minutes"`.
    - If there are less than or equal to 50 km, print: `"I'm parking. I'll see you right now"`.
- [ ] Write a program that checks whether a person is eligible to vote or not based on their age.
    - If the age of user is 18 or older then print `You are eligible to vote`
    - If the age of user is less than 18 then print `You aren't eligible to vote`.

    Note: `age` ***can be between*** `1` ***to*** `100` ***.***

💡 Hints:

- Visit the conditional logic chapter to understand how to use conditional logic and conditional statements.

Objects

Objects are the collection of `key`, `value` pairs and each pair of key-value are known as a property. Here, the property of the `key` can be a `string` whereas its `value` can be of any value.

📝 Tasks:

Given a Doe family that includes two members, where each member's information is provided in form of an object.

```
let person = {
    name: "John",                    //String
    lastName: "Doe",
    age: 35,                         //Number
    gender: "male",
    luckyNumbers: [ 7, 11, 13, 17], //Array
    significantOther: person2       //Object,
};

let person2 = {
    name: "Jane",
    lastName: "Doe",
    age: 38,
    gender: "female",
    luckyNumbers: [ 2, 4, 6, 8],
    significantOther: person
};

let family = {
    lastName: "Doe",
    members: [person, person2]       //Array of objects
};
```

- [ ] Find a way to print the name of the first member of the Doe family in a `console`.
- [ ] Change the fourth `luckyNumbers` of the second member of the Doe family to `33`.
- [ ] Add a new member to the family by creating a new person (`Jimmy` `Doe`, `13`, `male`, `[1, 2, 3, 4]`, `null`) and update the member list.
- [ ] Print the `SUM` of the lucky numbers of Doe family in the `console`.

💡 Hints:

- Visit the objects chapter to understand how the object work.
- You can get `luckyNumbers` from each person object inside the family object.
- Once you get each array just loop over it adding every element and then add each sum of the 3 family members.

FizzBuzz Problem

The *FizzBuzz* problem is one of the commonly asked questions, here one has to print *Fizz* and *Buzz* upon some conditions.

📝 Tasks:

- [ ] Write a program to print all the numbers between 1 to 100 in such a way that the following conditions are met.
    - For multiples of 3, instead of the number, print `Fizz`.
    - For multiples of 5, print `Buzz`.
    - For numbers that are multiples of both 3 and 5, print `FizzBuzz`.

    ```
    /
    1
    2
    Fizz
    4
    Buzz
    Fizz
    7
    8
    Fizz
    Buzz
    11
    Fizz
    13
    14
    FizzBuzz
    16
    ....
    ....
    98
    Fizz
    Buzz
    /
    ```

💡 Hints:

- Visit the loops chapter to understand how the loop works.

Get the Titles!

The *Get the Titles!* problem is an interesting problem where we have to get the title from a list of books. This is a good exercise for the implementation of arrays and objects.

📝 Tasks:

Given an array of objects that represent books with an author.

```
const books = [
  {
    title: "Eloquent JavaScript, Third Edition",
    author: "Marijn Haverbeke"
  },
  {
    title: "Practical Modern JavaScript",
    author: "Nicolás Bevacqua"
  }
]
```

- [ ] Write a program to create a function `getTheTitles` that takes the array and returns the array of title and print its value in the `console` .

💡 Hints:

- Visit the arrays and objects chapter to understand how the array and object work.

Chapter 22

Interview questions

This chapter discusses various questions to better prepare candidate on their understanding about JavaScript. It is divided into three parts: basic, intermediate and advance level.

- Basic Level
- Intermediate Level
- Advance Level

Basic JavaScript Interview Questions

1. History and Defining Variables.

1.1. What is JavaScript?

Answer: JavaScript is a high-level, interpreted programming language commonly used for web development to add interactivity and dynamic behavior to websites.

1.2. Who created/Developed JavaScript?

Answer: JavaScript was created by *Brendan Eich* while he was working at **Netscape Communications Corporation**. He developed the language in just ten days in May 1995. JavaScript was originally called "*Mocha*" but was later renamed "*LiveScript*" and eventually "*JavaScript*" as part of a marketing collaboration with **Sun Microsystems** (now **Oracle Corporation**), which had a programming language called **Java** that was gaining popularity at the time. Despite the name similarity, *JavaScript* and *Java* are entirely different programming languages with distinct purposes and characteristics.

1.3. How do you declare a variable in JavaScript?

Answer: You can declare a variable using `var` , `let` , or `const` :

- `var` (function-scoped)
- `let` (block-scoped)
- `const` (block-scoped, for constants)

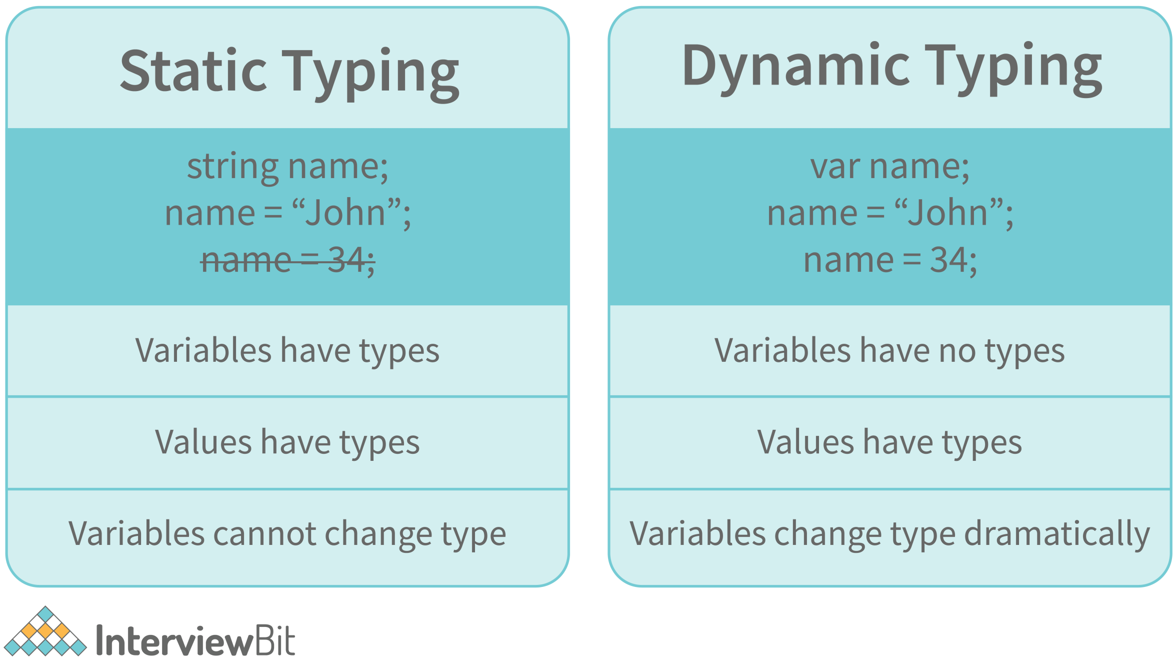
1.4. What is the difference between `let` , `var` , and `const` ?

Answer:

- `var` is function-scoped, while `let` and `const` are block-scoped.
- `let` allows variable reassignment, while `const` is used for constants.
- Variables declared with `var` are hoisted, whereas `let` and `const` are not hoisted.

1.5. Is javascript a statically typed or a dynamically typed language?

Answer: JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during run-time in contrast to a statically typed language, where the type of a variable is checked during compile-time.



Static Typing	Dynamic Typing

string name;
name = “John”;
~~name = 34;~~
	var name;
	name = “John”;
	name = 34;

Variables have types	Variables have no types
Values have types	Values have types
Variables cannot change type	Variables change type dramatically

InterviewBit

Since javascript is a *loosely(dynamically)* typed language, variables in JS are not associated with any type. A variable can hold the value of any data type.

For example, a variable that is assigned a number type can be converted to a string type:

```
var a = 23;
var a = "Hello World!";
```

1.6. What are the types of errors in javascript?

Answer: There are seven types of errors in javascript.

1. **Syntax error** - The error occurs when you use a predefined syntax incorrectly.

```
const func = () =>
console.log(hello)
}
```

2. **Reference Error** - In a case where a variable reference can't be found or hasn't been declared, then a Reference error occurs.

```
console.log(x);
```

3. **Type Error** - An error occurs when a value is used outside the scope of its data type.

```
let num = 15;
console.log(num.split(""));
```

4. **Evaluation Error** - Current JavaScript engines and EcmaScript specifications do not throw this error. However, it is still available for backward compatibility. The error is called when the eval() backward function is used, as shown in the following code block

```
try{
  throw new EvalError("'Throws an error'")
}catch(error){
  console.log(error.name, error.message)
}
```

1. **RangeError** - There is an error when a range of expected values is required.

```
const checkRange = (num)=>{
  if (num < 30) throw new RangeError("Wrong number");
  return true
}

checkRange(20);
```

1. **URI Error** - When the wrong character(s) are used in a URI function, the error is called uri error

   ```
   console.log(decodeURI("https://www.educative.io/shoteditor"))
   console.log(decodeURI("%sdfk"));
   ```

2. **Internal Error** - In the JS engine, this error occurs most often when there is too much data and the stack exceeds its critical size. When there are too many recursion patterns, switch cases, etc., the JS engine gets overwhelmed.

   ```
   switch(condition) {
   case 1:
   ...
   break
   case 2:
   ...
   break
   case 3:
   ...
   break
   case 4:
   ...
   break
   case 5:
   ...
   break
   case 6:
   ...
   break
   case 7:
   ...
   break
   ... up to 500 cases
   }
   ```

1.7. Mention some advantages of javascript.

Answer: There are many advantages of javascript. Some of them are

- Javascript is executed on the client-side as well as server-side also. There are a variety of Frontend Frameworks that you may study and utilize. However, if you want to use JavaScript on the backend, you'll need to learn NodeJS. It is currently the only JavaScript framework that may be used on the backend.
- Javascript is a simple language to learn.
- Web pages now have more functionality because of Javascript.
- To the end-user, Javascript is quite quick.

1.8. What is the ‘this’ keyword in JavaScript?

Answer:

The Keyword ‘this’ in JavaScript is used to call the current object as a constructor to assign values to object properties.

2. Functions

2.1. How do you create a function in JavaScript?

Answer:

You can create a function using the `function` keyword or arrow functions (`=>`): **Example**:

```
function myFunction() {
  // Function body
}

const myArrowFunction = () => {
  // Function body
};
```

2.2. What are Callbacks?

Answer: A callback is a function that will be executed after another function gets executed. In javascript, functions are treated as first-class citizens, they can be used as an argument of another function, can be returned by another function, and can be used as a property of an object.

Functions that are used as an argument to another function are called callback functions. **Example**:

```
function divideByHalf(sum) {
  console.log(Math.floor(sum / 2));
}

function multiplyBy2(sum) {
  console.log(sum * 2);
}

function operationOnSum(num1, num2, operation) {
  var sum = num1 + num2;
  operation(sum);
}

operationOnSum(3, 3, divideByHalf); // Outputs 3

operationOnSum(5, 5, multiplyBy2); // Outputs 20
```

- In the code above, we are performing mathematical operations on the sum of two numbers. The `operationOnSum` function takes 3 arguments, the first number, the second number, and the operation that is to be performed on their sum (callback).
- Both `divideByHalf` and `multiplyBy2` functions are used as callback functions in the code above.
- These callback functions will be executed only after the function `operationOnSum` is executed.
- Therefore, a callback is a function that will be executed after another function gets executed.

2.3. Explain Scope and Scope Chain in javascript.

Answer: Scope in JS determines the accessibility of variables and functions at various parts of one's code.

In general terms, the scope will let us know at a given part of code, what are variables and functions we can or cannot access.

There are three types of scopes in JS:

- Global Scope
- Local or Function Scope
- Block Scope

Global Scope: Variables or functions declared in the global namespace have global scope, which means all the variables and functions having global scope can be accessed from anywhere inside the code.

```
var globalVariable = "Hello world";

function sendMessage() {
  return globalVariable; // can access globalVariable since it's written in global space
}
function sendMessage2() {
  return sendMessage(); // Can access sendMessage function since it's written in global space
}
sendMessage2(); // Returns “Hello world”
```

Function Scope: Any variables or functions declared inside a function have `local/function scope`, which means that all the variables and functions declared inside a function, can be accessed from within the function and not outside of it.

```
function awesomeFunction() {
  var a = 2;

  var multiplyBy2 = function () {

    console.log(a * 2); // Can access variable "a" since a and multiplyBy2 both are written inside the same fun
  };
}
console.log(a); // Throws reference error since a is written in local scope and cannot be accessed outside

multiplyBy2(); // Throws reference error since multiplyBy2 is written in local scope
```

Block Scope: `Block scope` is related to the variables declared using let and const. Variables declared with var do not have block scope. Block scope tells us that any variable declared inside a block `{ }`, can be accessed only inside that block and cannot be accessed outside of it.

```
{
  let x = 45;
}

console.log(x); // Gives reference error since x cannot be accessed outside of the block

for (let i = 0; i < 2; i++) {
  // do something
}

console.log(i); // Gives reference error since i cannot be accessed outside of the for loop block
```

Scope Chain: JavaScript engine also uses Scope to find variables. Let's understand that using an example:

```
var y = 24;

function favFunction() {
  var x = 667;
  var anotherFavFunction = function () {
    console.log(x); // Does not find x inside anotherFavFunction, so looks for variable inside favFunction, out
  };

  var yetAnotherFavFunction = function () {
    console.log(y); // Does not find y inside yetAnotherFavFunction, so looks for variable inside favFunction a
  };

  anotherFavFunction();
  yetAnotherFavFunction();
}
favFunction();
```

As you can see in the code above, if the javascript engine does not find the variable in local scope, it tries to check for the variable in the outer scope. If the variable does not exist in the outer scope, it tries to find the variable in the global scope.

If the variable is not found in the global space as well, a reference error is thrown.

2.4. Explain Higher Order Functions in javascript.

Answer: Functions that operate on other functions, either by taking them as arguments or by returning them, are called *higher-order functions*.

Higher-order functions are a result of functions being **first-class citizens** in javascript.

Examples of higher-order functions:

```
function higherOrder(fn) {
  fn();
}

higherOrder(function () {
  console.log("Hello world");
});
function higherOrder2() {
  return function () {
    return "Do something";
  };
}
var x = higherOrder2();
x(); // Returns "Do something"
```

2.5. What do you mean by Self Invoking Functions in javascript?

Answer: Without being requested, a self-invoking expression is automatically invoked (initiated). If a function expression is followed by (), it will execute automatically. A function declaration cannot be invoked by itself.

Normally, we declare a function and call it, however, anonymous functions may be used to run a function automatically when it is described and will not be called again. And there is no name for these kinds of functions.

2.6. What is the difference between exec () and test () methods in javascript?

Answer:

- test () and exec () are RegExp expression methods used in javascript.
- We'll use exec () to search a string for a specific pattern, and if it finds it, it'll return the pattern directly; else, it'll return an 'empty' result.
- We will use a test () to find a string for a specific pattern. It will return the Boolean value 'true' on finding the given text otherwise, it will return 'false'

2.7. What is the difference between Function declaration and Function expression?

Answer:

Function declaration:

A. Declared as a separate statement within the main JavaScript code.
B. Can be called before the function is defined.
C. Offers better code readability and better code organization.

Example:

```
function abc() {
    return 5;
}
```

Function expression:

A. Created inside an expression or some other construct.
B. Created when the execution point reaches it; can be used only after that.
C. Used when there is a need for a conditional declaration of a function.

Example:

```
var a = function abc() {
    return 5;
}
```

2.8. What are the arrow functions in JavaScript?

Answer: Arrow functions are a short and concise way of writing functions in JavaScript. The general syntax of an arrow function is as below: const helloWorld = () => { console.log("hello world!"); };

2.9. Passed by value and passed by reference :

Answer:

- Passed By Values Are Primitive Data Types.
  Consider the following example:

Here, the a=432 is a primitive data type i.e. a number type that has an assigned value by the operator. When the var b=a code gets executed, the value of ‘var a’ returns a new address for ‘var b’ by allocating a new space in the memory, so that ‘var b’ will be operated at a new location.

Example:

```
var a = 432;
var b = a;
```

Passed_by_values_new

- Passed by References Are Non-primitive Data Types.

Consider the following example:

The reference of the 1st variable object i.e. ‘var obj’ is passed through the location of another variable i.e. ‘var obj2’ with the help of an assigned operator.

Example:

```
var obj = { name: "Raj", surname: "Sharma" };
var obj2 = obj;
```

3. Data Types and Operator

3.1. What are the different data types present in javascript?

Answer:

1. **Primitive types**
    - `String` - It represents a series of characters and is written with quotes. A string can be represented using a single or a double quote.

      Example :

      ```
      var str = "Vivek Singh Bisht"; //using double quotes
      var str2 = "John Doe"; //using single quotes
      ```

    - `Number` - It represents a number and can be written with or without decimals.

      Example :

      ```
      var x = 3; //without decimal
      var y = 3.6; //with decimal
      ```

    - `BigInt` - This data type is used to store numbers which are above the limitation of the Number data type. It can store large integers and is represented by adding “n” to an integer literal.

      Example :

      ```
      var bigInteger = 234567890123456789012345678901234567890;
      ```

    - `Boolean` - It represents a logical entity and can have only two values : true or false. Booleans are generally used for conditional testing.

      Example :

      ```
      var a = 2;
      var b = 3;
      var c = 2;
      (a == b)(
        // returns false
        a == c
      ); //returns true
      ```

- `Undefined` - When a variable is declared but not assigned, it has the value of undefined and it’s type is also undefined.

  Example :

  ```
  var x; // value of x is undefined
  var y = undefined; // we can also set the value of a variable as undefined
  ```

- `Null` - It represents a non-existent or a invalid value.

  Example :

  ```
  var z = null;
  ```

- `Symbol` - It is a new data type introduced in the ES6 version of javascript. It is used to store an anonymous and unique value.

  Example:

  ```
  var symbol1 = Symbol('symbol');
  typeof of primitive types :
  typeof "John Doe" // Returns "string"
  typeof 3.14 // Returns "number"
  typeof true // Returns "boolean"
  typeof 234567890123456789012345678901234567890n // Returns bigint
  typeof undefined // Returns "undefined"
  typeof null // Returns "object" (kind of a bug in JavaScript)
  typeof Symbol('symbol') // Returns Symbol 2. Non-primitive types
  ```

Primitive data types can store only a single value. To store multiple and complex values, non-primitive data types are used.

1. **Non-Primitive types**

- `Object` - Used to store collection of data.

  Example:

  ```
  // Collection of data in key-value pairs
  var obj1 = {
    x: 43,
    y: "Hello world!",
    z: function () {
      return this.x;
    },
  };
  ```

- `Array`

  Example:

  ```
  // Collection of data as an ordered list

  var array1 = [5, "Hello", true, 4.1];
  ```

> **Note- It is important to remember that any data type that is not a primitive data type, is of `Object` type in javascript.**

3.2 Difference between `==` and `===` operators.

Answer: Both are comparison operators. The difference between both the operators is that `==` is used to compare values whereas, `===` is used to compare both values and types.

Example:

```
var x = 2;
var y = "2";
(x == y)(
  // Returns true since the value of both x and y is the same
  x === y
); // Returns false since the typeof x is "number" and typeof y is "string"
```

3.3. What is NaN property in JavaScript?

Answer: `NaN` property represents the “**Not-a-Number**” value. It indicates a value that is not a legal number.

`typeof` of `NaN` will return a Number.

To check if a value is `NaN`, we use the `isNaN()` function,

> Note- `isNaN()` function converts the given value to a Number type, and then equates to NaN.

Example:

```
isNaN("Hello"); // Returns true
isNaN(345); // Returns false
isNaN("1"); // Returns false, since '1' is converted to Number type which results in 0 ( a number)
isNaN(true); // Returns false, since true converted to Number type results in 1 ( a number)
isNaN(false); // Returns false
isNaN(undefined); // Returns true
```

3.4. Which method is used to retrieve a character from a certain index?

Answer: The `charAt()` function of the JavaScript string finds a char element at the supplied index. The index number begins at `0` and continues up to `n-1`, Here `n` is the string length. The index value must be positive, higher than, or the same as the string length.

4. Some important concepts

4.1. What is Hoisting in JavaScript?

Answer: Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. Inevitably, this means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

Example 1: Hoisting of variable

```
hoistedVariable = 3;
console.log(hoistedVariable); // outputs 3 even when the variable is declared after it is initialized
var hoistedVariable;
```

Example 2: Hoisting of function

```
hoistedFunction();  // Outputs " Hello world! " even when the function is declared after calling

function hoistedFunction(){
  console.log(" Hello world! ");
}
```

Example 3: Hoisting of function expression

```
// Hoisting takes place in the local scope as well
function doSomething(){
  x = 33;
  console.log(x);
  var x;
}
doSomething(); // Outputs 33 since the local variable “x” is hoisted inside the local scope

/* Note - Variable initializations are not hoisted, only variable declarations are hoisted: */
var x;
console.log(x); // Outputs "undefined" since the initialization of "x" is not hoisted
x = 23;

/* Note - To avoid hoisting, you can run javascript in strict mode by using “use strict” on top of the code: */
"use strict";
x = 23; // Gives an error since 'x' is not declared
var x;
```

4.2. Why do we use the word “debugger” in javascript?

Answer:

The `debugger` keyword is used to create breakpoints in the code. When the browser finds the debugger keyword in the code, it stops executing the code and opens the debugging tool of the browser.

4.3. What is currying in JavaScript?

Answer:

Currying is an advanced technique to transform a function of arguments n, to n functions of one or fewer arguments.

Example of a curried function:

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

add(3)(4)
```

For Example, if we have a function `f(a,b)`, then the function after currying, will be transformed to `f(a)(b)`.

By using the currying technique, we do not change the functionality of a function, we just change the way it is invoked.

Let’s see currying in action:

```
function multiply(a,b){
  return a*b;
}

function currying(fn){
  return function(a){
    return function(b){
      return fn(a,b);
    }
  }
}

var curriedMultiply = currying(multiply);

multiply(4, 3); // Returns 12

curriedMultiply(4)(3); // Also returns 12
```

As one can see in the code above, we have transformed the function multiply(a,b) to a function curriedMultiply , which takes in one parameter at a time.

4.4. What are some advantages of using External JavaScript?

Answer:

External JavaScript is the JavaScript Code (script) written in a separate file with the extension.js, and then we link that file inside the or element of the HTML file where the code is to be placed.

Some advantages of external javascript are

- It allows web designers and developers to collaborate on HTML and javascript files.
- We can reuse the code.
- Code readability is simple in external javascript.

4.5. What is a closure in JavaScript?

Answer:

A closure is a function that has access to its outer function scope even after the outer function has returned. This means a closure can remember and access variables and arguments of its outer function even after the function has finished. In Short- A closure is a function that has access to variables from its outer (enclosing) function scope, even after the outer function has finished executing.

4.6. What is the DOM in JavaScript?

Answer:

The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page.

4.7. What is event delegation?

Answer:

Event delegation is a technique for listening to events where you delegate a parent element as the listener for all of the events that happen inside it. The events are handled by the callback function of the parent element.

4.8. How can you make an AJAX request in JavaScript?

Answer:

AJAX stands for Asynchronous JavaScript and XML. It is a set of web development techniques using many web technologies on the client-side to create asynchronous web applications. With Ajax, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behavior of the existing page.

You can make AJAX requests using the XMLHttpRequest object or by using the fetch API. Here's an example using fetch:

```
fetch('https://example.com/api/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

4.9. What is a promise in JavaScript?

Answer:

A promise is an object that may produce a single value sometime in the future: either a resolved value or a reason that it’s not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

4.10. Why do you need a promise in JavaScript ?

Answer:

Promises are used to handle asynchronous operations. They provide an alternative approach for callbacks by reducing the callback hell and writing the cleaner code.

4.11. Explain equality in JavaScript ?

Answer:

JavaScript provides two types of equality operators: strict equality (===) and loose equality (==)

- Strict Equality (===): This operator compares two values without performing any type conversion. If the values have different types, they are considered unequal. If the values have the same type, are not numbers, and have the same value, they are considered equal. For numbers, they are considered equal if they are both not NaN and have the same value, or if one is +0 and the other is -0
- Loose Equality (==): This operator performs type conversion when comparing the operands. If the operands have the same type, they are compared in the same way as the strict equality operator. If the operands have different types, JavaScript attempts to convert them to a common type and then compare them. The rules for type conversion can sometimes lead to unexpected results, so it's generally recommended to use the strict equality operator to avoid potential issues

5. Object

5.1. What are the possible ways to create objects in JavaScript?

Answer:

There are many ways to create objects in javascript as below

Object constructor:

i. The simplest way to create an empty object is using the Object constructor. Currently this approach is not recommended.

```
var object = new Object();
```

The Object() is a built-in constructor function so "new" keyword is not required. the above can be written as:

```
var object = Object();
```

ii. Object's create method:

The create method of Object creates a new object by passing the prototype object as a parameter

```
var object = Object.create(null);
```

iii. Object literal syntax:

The object literal syntax (or object initializer), is a comma-separated set of name-value pairs wrapped in curly braces.

```
var object = {
     name: "Sudheer",
     age: 34
};

Object literal property values can be of any data type, including array, function, and nested object.
```

Note: This is an easiest way to create an object

iv. Function constructor:

Create any function and apply the new operator to create object instances,

```
function Person(name) {
  this.name = name;
  this.age = 21;
}
var object = new Person("Sudheer");
```

v. Function constructor with prototype:

This is similar to function constructor but it uses prototype for their properties and methods,

```
function Person() {}
Person.prototype.name = "Sudheer";
var object = new Person();
```

This is equivalent to an instance created with an object create method with a function prototype and then call that function with an instance and parameters as arguments.

```
function func() {}

new func(x, y, z);
```

(OR)

```
// Create a new instance using function prototype.
var newInstance = Object.create(func.prototype)

// Call the function
var result = func.call(newInstance, x, y, z),

// If the result is a non-null object then use it otherwise just use the new instance.
console.log(result && typeof result === 'object' ? result : newInstance);
```

vi. ES6 Class syntax:

ES6 introduces class feature to create the objects

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

var object = new Person("Sudheer");
```

6.Miscellaneous

6.1. What is a strict mode in JavaScript ?

Answer:

Strict Mode is a new feature in ECMAScript 5 that allows you to place a program, or a function, in a "strict" operating context. This way it prevents certain actions from being taken and throws more exceptions. The literal expression "use strict"; instructs the browser to use the javascript code in the Strict mode.

6.2. What is null value in JavaScript ?

Answer:

The value null represents the intentional absence of any object value. It is one of JavaScript's primitive values. The type of null value is object. You can empty the variable by setting the value to null.

```
var user = null;
console.log(typeof user); //object
```

6.3. What is eval in JavaScript ?

Answer:

The eval() function evaluates JavaScript code represented as a string. The string can be a JavaScript expression, variable, statement, or sequence of statements.

```
console.log(eval("1 + 2")); //  3
```

6.4. Is JavaScript a compiled or interpreted language ?

Answer:

JavaScript is an interpreted language, not a compiled language. An interpreter in the browser reads over the JavaScript code, interprets each line, and runs it. Nowadays modern browsers use a technology known as Just-In-Time (JIT) compilation, which compiles JavaScript to executable bytecode just as it is about to run.

6.5. Is JavaScript a case-sensitive language ?

Answer:

Yes, JavaScript is a case sensitive language. The language keywords, variables, function & object names, and any other identifiers must always be typed with a consistent capitalization of letters.

7.JSON

7.1. What is JSON ?

Answer:

JSON (JavaScript Object Notation) is a lightweight format that is used for data interchanging. It is based on a subset of JavaScript language in the way objects are built in JavaScript.

7.2. What are the syntax rules of JSON ?

Answer:

Below are the list of syntax rules of JSON

- The data is in name/value pairs
- The data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

7.3. What is the purpose JSON stringify ?

Answer:

When sending data to a web server, the data has to be in a string format. You can achieve this by converting JSON object into a string using stringify() method.

```
var userJSON = { name: "John", age: 31 };
var userString = JSON.stringify(userJSON);
console.log(userString); //"{"name":"John","age":31}"
```

7.4. How do you parse JSON string ?

Answer:

When receiving the data from a web server, the data is always in a string format. But you can convert this string value to a javascript object using parse() method.

```
var userString = '{"name":"John","age":31}';
var userJSON = JSON.parse(userString);
console.log(userJSON); // {name: "John", age: 31}
```

7.5. Why do you need JSON ?

Answer:

When exchanging data between a browser and a server, the data can only be text. Since JSON is text only, it can easily be sent to and from a server, and used as a data format by any programming language.

7.6. How do you define JSON arrays ?

Answer:

JSON arrays are written inside square brackets and arrays contain javascript objects. For example, the JSON array of users would be as below,

```
"users":[
  {"firstName":"John", "lastName":"Abrahm"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Shane", "lastName":"Warn"}
]
```

7.6. In JSON, what is the purpose of square brackets, and how are they used?

Answer:

In JSON, square brackets `[ ]` are used to encapsulate and define arrays within JSON data structures. JSON arrays can contain a collection of values, which can be of various data types, including objects, strings, numbers, and other JSON arrays.

Intermediate Level JavaScript Interview Questions

1. Loops

1.1. What is the definition of an iteration in a JavaScript loop?

Answer:

An iteration in a JavaScript loop refers to each individual execution of the loop's body, typically corresponding to one cycle of the loop.

1.2. What are all the looping structures in JavaScript?

Answer:

While loop: A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

For loop: A for loop provides a concise way of writing the loop structure. Unlike a while loop, for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

Do while: A do-while loop is similar to while loop with the only difference that it checks the condition after executing the statements, and therefore is an example of Exit Control Loop.

1.3. How does the break statement work in a loop?

Answer:

The break statement terminates the current loop or switch statement and transfers program control to the statement following the terminated statement. It can also be used to jump past a labeled statement when used within that labeled statement.

1.4. How does the continue statement work in a loop?

Answer:

The continue directive is a "lighter version" of the break statement. It does not stop the whole loop; instead, it stops the current iteration and forces the loop to start a new one (if the condition allows).

2. Switch statement

2.1. What is a switch statement in JavaScript?

Answer:

A switch statement in JavaScript is a control flow statement that evaluates an expression and executes a specific block of code based on the matched case.

2.2. What are the advantages of employing a Switch statement?

Answer:

A switch statement can replace multiple checks, and it is more descriptive and easier to read. Switch statements improve code readability, provide better performance, simplify complex conditionals, enhance maintainability, and support cleaner syntax.

2.3. Is the order of case statements important in a switch statement?

Answer:

The order of case statements is important in a switch statement, especially when employing fall-through behavior. Cases are evaluated sequentially, so a matching case found earlier will prevent subsequent cases from being tested, affecting execution and performance.

3. JavaScript cookies

3.1. What are JavaScript Cookies ?

Answer:

Cookies are small files that are stored on a user's computer. They are used to hold a modest amount of data specific to a particular client and website and can be accessed either by the web server or by the client's computer. When cookies were invented, they were basically little documents containing information about you and your preferences. For instance, when you select the language in which you want to view your website, the website would save the information in a document called a cookie on your computer, and the next time when you visit the website, it would be able to read a cookie saved earlier.

3.2. How to create a cookie using JavaScript?

Answer:

To create a cookie by using JavaScript you just need to assign a string value to the document.cookie object.

```
document.cookie = "key1 = value1; key2 = value2; expires = date";
```

3.3. How to read a cookie using JavaScript?

Answer:

The value of the document.cookie is used to create a cookie. Whenever you want to access the cookie you can use the string. The document.cookie string keep a list of name = value pairs separated by semicolons, where name is the name of a cookie and the value is its string value.

3.4. How to delete a cookie using JavaScript?

Answer:

Deleting a cookie is much easier than creating or reading a cookie, you just need to set the expires = “past time” and make sure one thing defines the right cookie path unless few will not allow you to delete the cookie.

4. Pop-up boxes in JavaScript

4.1. What are the types of Pop up boxes available in JavaScript?

Answer:

There are three types of pop boxes available in JavaScript: Alert, Confirm, Prompt.

4.2. What is the difference between an alert box and a confirmation box?

Answer:

An alert box will display only one button which is the OK button. It is used to inform the user about the agreement has to agree. But a Confirmation box displays two buttons OK and cancel, where the user can decide to agree or not.

5. Arrow Functions

5.1. What is the definition of an arrow function?

Answer:

An arrow function is a concise syntax for defining anonymous functions in JavaScript, using the arrow notation. It offers shorter syntax, lexical scoping of "this", and can't be used as a constructor.

5.2. How do arrow functions differ from function expressions?

Answer:

Arrow functions provide a shorter syntax, don't have their own this, arguments, super, or new.target, and can't be used as constructors, unlike function expressions.

5.3. What does lexical 'this' binding mean in arrow functions?

Answer:

"Lexical this" binding in arrow functions means they don't create their own 'this' context; instead, they inherit 'this' from their surrounding, enclosing scope, reducing common 'this'-related issues.

5.4. What are the advantages of using arrow functions?

Answer:

The advantages of using arrow functions in JavaScript include shorter syntax, implicit return, and lexical ‘this’ binding.

5.5. What are the common use cases for arrow functions?

Answer:

Arrow functions are commonly used for object methods, event listeners, callbacks, and other functions that require shorter, more concise syntax.

Temporal Dead Zone

What is the Temporal Dead Zone in ES6?

Answer:

In ES6 let and const are hoisted (like var , class and function), but there is a period between entering scope and being declared where they cannot be accessed. This period is the temporal dead zone (TDZ)

```
//console.log(aLet)  // would throw ReferenceError
let aLet;
console.log(aLet); // undefined
aLet = 10;
console.log(aLet); // 10
```

Truthy and Falsy

What are Truthy and Falsy Values in JavaScript?

Answer:

In JavaScript, "truthy" and "falsy" are terms used to describe how values are evaluated in Boolean contexts, such as conditions in if statements and loops. Understanding truthy and falsy values is crucial when working with conditional logic.

Falsy Values:

- `false` : The Boolean value false.
- `0` : The numeric value zero.
- `""` : An empty string.
- `null` : A special value indicating the absence of an object.
- `undefined` : A variable that has not been assigned a value.
- `NaN` : Stands for "Not-a-Number" and represents an invalid number.
- When any of these values are used in a Boolean context, they are treated as "falsy," meaning they are considered equivalent to false.

Example:

```
if (false) {
    // This code block won't execute because false is falsy.
}

if (0) {
    // This code block won't execute because 0 is falsy.
}

if ("" === false) {
    // This comparison is true because an empty string is falsy.
}

if (null) {
    // This code block won't execute because null is falsy.
}

if (undefined) {
    // This code block won't execute because undefined is falsy.
}

if (NaN) {
    // This code block won't execute because NaN is falsy.
}
```

Truthy Values:

Any value that is not explicitly "falsy" is considered "truthy" in JavaScript. These values are treated as equivalent to true in Boolean contexts.

Example:

```
if (true) {
    // This code block will execute because true is truthy.
}

if (42) {
    // This code block will execute because 42 is truthy.
}

if ("Hello") {
    // This code block will execute because a non-empty string is truthy.
}

if ({} === true) {
    // This comparison is false because an empty object is truthy but not equal to true.
}

if ([] === true) {
    // This comparison is false because an empty array is truthy but not equal to true.
}
```

Understanding truthy and falsy values allows us to write more concise and expressive code, especially when dealing with conditional logic. We can use this behavior to write shorter and more readable code when evaluating conditions and choosing between two values or actions.

6. Regular Expression

6.1. What is a Regular Expression ?

Answer:

A regular expression is a sequence of characters that forms a search pattern. You can use this search pattern for searching data in a text. These can be used to perform all types of text search and textreplace operations. Let's see the syntax format now,

```
 /pattern/modifiers;
```

For example, the regular expression or search pattern with case-insensitive username would be,

```
/John/i;
```

6.2. What are the string methods available in Regular expression ?

Answer:

Regular Expressions has two string methods: search() and replace(). The search() method uses an expression to search for a match, and returns the position of the match.

```
var msg = "Hello John";
var n = msg.search(/John/i); // 6
```

The replace() method is used to return a modified string where the pattern is replaced.

```
var msg = "Hello John";
var n = msg.replace(/John/i, "Buttler"); // Hello Buttler
```

6.3. What are modifiers in regular expression ?

Answer:

Modifiers can be used to perform case-insensitive and global searches. Let's list down some of the modifiers,

| Modifier | Description | | i | Perform case-insensitive matching | | g | Perform a global match rather than stops at first match | | m | Perform multiline matching |

Let's take an example of global modifier,

```
var text = "Learn JS one by one";
var pattern = /one/g;
var result = text.match(pattern); // one,one
```

6.4. What are regular expression patterns ?

Answer:

Regular Expressions provide a group of patterns in order to match characters. Basically they are categorized into 3 types,

- i.Brackets: These are used to find a range of characters. For example, below are some use cases,
  - a. [abc]: Used to find any of the characters between the brackets(a,b,c).
  - b. [0-9]: Used to find any of the digits between the brackets.
  - c. (a|b): Used to find any of the alternatives separated with |
- ii.Metacharacters: These are characters with a special meaning For example, below are some use cases
  - a. \d: Used to find a digit
  - b. \s: Used to find a whitespace character
  - c. \b: Used to find a match at the beginning or ending of a word
- iii.Quantifiers: These are useful to define quantities For example, below are some use cases
  - a. n+: Used to find matches for any string that contains at least one n
  - b. n*: Used to find matches for any string that contains zero or more occurrences of n
  - c. n?: Used to find matches for any string that contains zero or one occurrences of n

6.5. What is a RegExp object ?

Answer:

RegExp object is a regular expression object with predefined properties and methods. Let's see the simple usage of RegExp object,

```
var regexp = new RegExp("\\w+");
console.log(regexp);
// expected output: /\w+/
```

6.6. How do you search a string for a pattern ?

Answer:

You can use the test() method of regular expression in order to search a string for a pattern, and return true or false depending on the result.

```
var pattern = /you/;
console.log(pattern.test("How are you?")); //true
```

6.7. What is Currying in Javascript?

Answer:

Currying in JavaScript transforms a function with multiple arguments into a nested series of functions, each taking a single argument. Currying helps you avoid passing the same variable multiple times, and it helps you create a higher order function. That is, when we turn a function call sum(1,2,3) into sum(1)(2)(3).

The number of arguments a function takes is also called arity.

```
function sum(a, b) {
    // do something
}
function _sum(a, b, c) {
    // do something
}
```

The function sum takes two arguments (two-arity function) and _sum takes three arguments (three-arity function).

Curried functions are constructed by chaining closures and by defining and immediately returning their inner functions simultaneously.

Advanced JavaScript Interview Questions

1. Closures and Scoping

1.1. What is a closure in JavaScript? Provide an example where using closures can be beneficial.

Answer:

A closure in JavaScript is a function that has access to its enclosing scope's variables, even after the outer function has finished executing. This mechanism allows functions to maintain state between executions.

Example: One common use of closures is to create factory functions or private variables. For instance, if we wanted to generate unique ID values for elements:

1.2. How do closures relate to variables' scope and lifetime?

Answer:

Closures allow a function to access all the variables, as well as functions, that are in its lexical scope, even after the outer function has completed. This results in the variables being preserved in memory, effectively allowing for variables to have a prolonged lifetime compared to standard local variables which would typically be garbage collected after their parent function has executed.

1.3. Give some examples of uses of closures in javascript?

Answer:

Here are some example of closures.

- Module Design Pattern.
- Currying.
- Memoize

2. Prototypal Inheritance

2.1. Explain the difference between classical inheritance and prototypal inheritance.

Answer:

Classical inheritance is a concept most often found in traditional Object-Oriented Programming languages like Java or C++, where a class can inherit properties and methods from a parent class. Prototypal inheritance, on the other hand, is unique to JavaScript. In JavaScript, each object can have another object as its prototype, and it can inherit properties from its prototype.

The primary difference is that classical inheritance is class-based, whereas prototypal inheritance is object-based. Although ES6 introduced the `class` keyword to JavaScript, it's syntactical sugar over the existing prototypal inheritance.

2.2. How can you extend built-in JavaScript objects?

Answer:

To extend built-in JavaScript objects, we can add methods or properties to their prototype. However, it's generally discouraged to modify native prototypes because it can lead to compatibility issues and unexpected behavior, especially if there are future changes to the JavaScript specification.

3. Asynchronous JavaScript

3.1. Explain the event loop in JavaScript. How does it relate to the call stack?

Answer:

The event loop is a fundamental concept in JavaScript and is responsible for handling the execution of multiple chunks of program over time, each run to completion. It works as a continuous loop that checks if there are tasks waiting in the message queue. If there are tasks and the main thread (call stack) is empty, it dequeues the task and executes it.

The call stack, on the other hand, is a data structure that tracks the execution of functions in a program. When a function is called, it is added to the call stack, and when it finishes executing, it is removed from the stack.

In the context of JavaScript, the event loop continuously checks the call stack to determine if it is empty. If it is empty and there are callback functions waiting in the message queue, those callbacks are executed.

3.2. What are promises, and how do they differ from callbacks in managing asynchronous operations?

Answer:

Promises are objects representing the eventual completion (or failure) of an asynchronous operation and its resulting value. A `Promise` is in one of these states:

- `pending` : initial state, neither fulfilled nor rejected.
- `fulfilled` : meaning the promised operation has completed and the promise has a resulting value.
- `rejected` : meaning the operation failed, and the promise will never be fulfilled.

Callbacks are functions that are passed into another function as arguments and are executed after the outer function has completed. While both promises and callbacks can handle asynchronous operations, promises provide a more robust way of handling them.

The key differences include:

- Promises allow for better chaining of asynchronous operations.
- Callbacks can lead to callback hell or pyramid of doom, where the code becomes hard to read and manage due to nested callbacks.
- Promises have a standardized error handling mechanism using `.then` and `.catch` .

3.3. Describe async/await. How does it simplify working with asynchronous code?

Answer:

`async/await` is a syntactic feature introduced in ES8 (or ES2017) to work with asynchronous code in a more synchronous-like fashion. It allows for writing asynchronous operations in a linear manner without callbacks, leading to cleaner, more readable code.

The `async` keyword is used to declare an asynchronous function, which ensures that the function returns a promise. The `await` keyword is used inside an `async` function to pause the execution until the promise is resolved or rejected.

Using `async/await` simplifies error handling, as we can use traditional try/catch blocks instead of `.catch` with promises.

4. Advanced Array Methods

4.1. Describe the functions of `map`, `reduce`, and `filter`. Provide an example of a practical use case for each.

Answer:

- `map`: It transforms each element of an array based on a function, returning a new array of the same length. **Example:** Doubling each number in an array.

  ```
  const numbers = [1, 2, 3, 4];
  const doubled = numbers.map((num) => num * 2); // [2, 4, 6, 8] ```
  ```

4.2. What are some limitations or pitfalls when using arrow functions?

Answer: Arrow functions introduce a concise way to write functions in JavaScript, but they come with certain limitations:

1. **No `this` Binding**: Arrow functions do not bind their own `this`. They inherit the `this` binding of the surrounding scope. This can be problematic, especially when using them as methods in objects or as event handlers.
2. **No `arguments` Object**: Arrow functions do not have the `arguments` object of their own. If we need to access the arguments object, we'd have to use traditional function expressions.
3. **Cannot be Used as Constructors**: Arrow functions cannot be used as constructors with the `new` keyword because they don't have the `[[Construct]]` internal method.
4. **No `prototype` Property**: Unlike regular functions, arrow functions do not have a `prototype` property.
5. **Less Readable for Complex Logic**: For simple operations, the concise syntax is beneficial. However, for functions containing complex logic, the concise syntax might make the code less readable.

5. "this" Keyword

5.1. Explain the behavior of the `this` keyword in different contexts, such as in a method, a standalone function, an arrow function, and an event handler.

Answer: The `this` word can vary on depending upon the context it's used. Some of them are explored below:

- In Method: It refers to the object that the method is called on.

```
const person = {
  name: "Alice",
  sayHello: function () {
    console.log(`Hello, my name is ${this.name}`);
  },
};

person.sayHello(); // Output: Hello, my name is Alice
```

- In Standalone Function: Here, the `this` behavior depends on how function is called. If the function is called in the global scope, `this` refer to the global object.

```
function greet() {
  console.log(`Hello, my name is ${this.name}`);
}

const name = "Alice";
greet(); // Output: Hello, my name is Alice
```

- In Arrow Function: Arrow functions capture the this value from their surrounding lexical scope, unlike regular functions. This means they lack their own this context.

```
const person = {
  name: "Bob",
  sayHello: () => {
    console.log(`Hello, my name is ${this.name}`);
  },
};

person.sayHello(); // Output: Hello, my name is undefined
```

5.2. How can you ensure a function uses a specific object as its `this` value?

Answer: We can ensure a function uses a specific object as its this value in JavaScript using methods like bind, arrow functions, call, apply, or by defining methods within ES6 classes. These techniques allow us to control the context in which the function operates and ensure it accesses the desired object's properties and methods.

6. Memory Management

6.1. What are memory leaks in JavaScript? Discuss potential causes and how to prevent them.

Answer: Memory leaks in JavaScript occur when the program unintentionally retains references to objects that are no longer needed, leading to increased memory usage and potential application issues. Common causes include unused variables, closures, event listeners, and circular references. To prevent memory leaks, developers should explicitly remove references, manage event listeners, avoid circular dependencies, use weak references, employ memory profiling tools, conduct testing and code reviews, and utilize linters and static analysis tools to detect potential issues early in the development process.

6.2. Describe the difference between shallow copy and deep copy. How can you achieve each in JavaScript?

Answer: Shallow copy and deep copy are methods for duplicating objects or arrays in JavaScript.

Shallow copy duplicates the top-level structure and values of an object or array but retains references to nested objects or arrays. Changes to nested structures affect both the original and the copy. Deep copy creates a new object or array and recursively duplicates all levels of nested structures, ensuring changes in the copy do not affect the original. To achieve a shallow copy, we can use methods like the spread operator or slice(). For a deep copy, custom logic or libraries like lodash's cloneDeep are necessary due to the lack of built-in deep copy methods in JavaScript.

7. ES6 and Beyond

7.1. Explain the purpose and usage of JavaScript's destructuring assignment.

Answer: JavaScript's destructuring assignment is a feature that simplifies the extraction of values from objects and arrays, making code more concise and readable. It allows us to assign values to variables based on property names (object destructuring) or position (array destructuring). Destructuring can also be used in function parameters and supports the rest syntax to capture remaining elements. It's a powerful tool for working with complex data structures.

7.2. Describe the significance of JavaScript modules and the ES6 `import/export` syntax.

Answer: JavaScript modules, along with the ES6 import/export syntax, are crucial for modern JavaScript development. They enable developers to organize, reuse, and maintain code effectively. Modules encapsulate related code, promote code reusability, manage dependencies, and improve code scalability. The ES6 import/export syntax provides a standardized way to declare and use modules, making it easier to structure and share code in a clean and maintainable manner. These features have become essential for building modular and maintainable JavaScript applications, both on the client and server sides.

7.3. How do template literals enhance string manipulation in ES6? Provide examples.

Answer: Template literals in ES6 enhance string manipulation by allowing developers to create strings with embedded expressions and multiline content in a more readable and flexible way. They support variable interpolation, multiline strings, expression evaluation, function calls, and even advanced use cases like tagged templates. This feature improves code readability and maintainability when working with complex strings that involve dynamic content or expressions.

8. Functional Programming

8.1. How does functional programming differ from imperative programming in JavaScript?

Answer:

Functional programming and imperative programming are two predominant programming paradigms.

- **Imperative Programming**: This paradigm is about telling the computer "how" to do something and relies on statements that change a programs state. In essence, it focuses on describing the steps to achieve a particular task. This often involves loops, conditionals, and statements that modify variables.

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

Functional Programming (FP)

FP is more about instructing the computer "what" to achieve, rather than detailing "how" to achieve it. It treats computational tasks as evaluations of mathematical functions and steers clear of mutable data and state alterations. In the context of JavaScript and most FP languages:

- **Pure Functions**: These are functions where the output value is determined solely by its input values, without observable side effects. This means, for the same input, the function will always produce the same output.
- **Immutable Data**: Once data is created, it can never change. Instead of altering existing data, functional programming practices involve creating new data structures.
- **First-Class and Higher-Order Functions**: In FP, functions are first-class citizens. This means they can be assigned to variables, passed into other functions as parameters, and returned as values. A higher-order function is a function that receives another function as an argument, returns a function, or both.

8.2. Explain first-class functions and their importance in functional programming.

Answer:

In JavaScript and many other programming languages, functions are considered as "first-class citizens." This means that functions can be:

- Assigned to variables.
- Passed as arguments to other functions.
- Returned from other functions as values.
- Stored in data structures like arrays and objects.

Here's a simple example demonstrating these properties:

```
// Assigning a function to a variable
const greet = function(name) {
  return "Hello, " + name;
}
// Passing a function as an argument to another function
function runFunction(fn, value) {
  return fn(value);
}
runFunction(greet, 'John'); // Returns: "Hello, John"
// Returning a function from another function
function multiplier(factor) {
  return function(number) {
    return number \* factor;
  }
}
const double = multiplier(2);
double(5); // Returns: 10
// Storing function in an array
const functions = [greet, double];
```

8.3. What is Execution Context and Lexical Environment?

Answer:

Generally, a function has its imaginary container or we can say some sort of context API. It provides the function with 3 things:

- Variables declared in the function
- The functions defined in the function
- Lexical environment This is known as Execution Context of a function.

AND

The lexical environment is a type of information source which provides the parent function with the scope of variables it can use. For ex:

```
// Assigning a function to a variable
function parent() {
  var a;
  var b;

  function child() {
    var x;
    var y;
    {rest code}
  }
}
```

Here, the lexical environment will have the information that parent function can use the variable a and b but not x and y (provides scope to the parent).

---

9. Storing data in browser

A. Local storage and Session storage

9.1 what are the key differences between Local Storage and Session Storage?

Answer:

Web Storage is a web API that provides two mechanisms for storing data in a web browser: Local Storage and Session Storage. The key differences are:

- Lifetime: Local Storage data persists even after the browser is closed, while Session Storage data is only available for the duration of the page session.
- Scope: Local Storage data is accessible across multiple windows and tabs from the same origin, whereas Session Storage data is limited to the current page or tab.
- Storage Limit: Local Storage typically has a larger storage limit (around 5-10 MB) compared to Session Storage (about 5-10 MB as well).

9.2 How do you store data in Local Storage and Session Storage using JavaScript?

Answer:

You can use the localStorage and sessionStorage objects to store data. Here's an example of storing data in Local Storage:

```
localStorage.setItem('username', 'JohnDoe');
```

To store data in Session Storage, replace localStorage with `sessionStorage.`

9.3 How can you clear or remove data from Local Storage and Session Storage?

Answer:

You can remove an item from storage using the removeItem method. To clear all items, you can use the clear method. For example:

Remove an item : `localStorage.removeItem('username');`

Clear all items : `localStorage.clear();`

9.4 Explain the security concerns associated with Web Storage.

Answer:

Web Storage is domain-specific, meaning that data is accessible only from the same domain that stored it. However, there are security concerns related to storing sensitive information in Web Storage. Data is not encrypted, and it's vulnerable to cross-site scripting (XSS) attacks, where malicious scripts can access and modify the stored data.

B. IndexDB

IndexedDB can be thought of as a “localStorage on steroids”. It’s a simple key-value database, powerful enough for offline apps, yet simple to use.

9.5 What is IndexDB, and how does it differ from Web Storage (Local Storage and Session Storage)?

Answer:

IndexDB is a low-level JavaScript-based database for storing large amounts of structured data. It differs from Web Storage in several ways:

- Data Structure: IndexedDB stores structured data, while Web Storage stores key-value pairs.
- Storage Limit: IndexedDB typically offers a larger storage limit (often in megabytes) compared to the limited storage of Web Storage.
- API Complexity: IndexedDB has a more complex API, requiring developers to define a database schema and work with transactions.

9.6 How do you open a database and create an object store in IndexedDB using JavaScript?

Answer:

You can open a database and create an object store like this:

Open a database (or create if it doesn't exist) :

```
const request = indexedDB.open('myDatabase', 1);
```

Create an object store :

```
request.onupgradeneeded = (event) => {
  const db = event.target.result;
  db.createObjectStore('myStore', { keyPath: 'id' });
};
```

10. Code Optimization

10.1. What is tree shaking ?

Answer:

Tree shaking is a form of dead code elimination. It means that unused modules will not be included in the bundle during the build process and for that it relies on the static structure of ES2015 module syntax,(i.e. import and export). Initially this has been popularized by the ES2015 module bundler rollup.

10.2. What is the need of tree shaking ?

Answer:

Tree Shaking can significantly reduce the code size in any application. i.e, The less code we send over the wire the more performant the application will be. For example, if we just want to create a “Hello World” Application using SPA frameworks then it will take around a few MBs, but by tree shaking it can bring down the size to just a few hundred KBs. Tree shaking is implemented in Rollup and Webpack bundlers.

10.3. Explain the role of the static structure of ES2015 module syntax in tree shaking. How does tree shaking leverage this structure to eliminate dead code?

Answer:

Tree shaking relies on the static structure of ES2015 module syntax, which means that the import and export statements have a clear and static structure at compile time. During the build process, the bundler (e.g., Rollup or Webpack) analyzes the import statements to determine which modules are being used and which are not. It then eliminates the unused modules from the final bundle, resulting in smaller and more efficient code.

10.4. What steps can you take to optimize tree shaking in a complex JavaScript project with multiple dependencies and deep module hierarchies?

Answer:

To optimize tree shaking in a complex project:

- Ensure all dependencies use ES2015 module syntax.
- Configure your bundler to perform tree shaking.
- Use the "sideEffects" property in your package.json to mark files or directories as side-effect free.
- Minimize the use of dynamic imports.
- Regularly audit and update your code to remove unused exports and functions.

Chapter 23

Design Patterns

Design patterns are object oriented solutions that you can implement to solve common programming problems that may occur during the development process. Design patterns are not the same as algorithms, they are a coding concept that you can use to resolve specific kinds of issues.

There are a total of 23 design patterns, these are not unique to any one language. These are fundamental concepts that can be applied across a wide variety of programming languages. Today we will learn about each design pattern and how to implement them using javascript. Design patterns can be classified under one of three categories

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Creational Design Patterns

Creational design patterns focus on object creation mechanisms

1. Factory Method

A factory function is just a function that creates an object and returns it. It is a creational design pattern that allows you to create objects without specifying the exact class or constructor to be used. It centralizes object creation logic, allowing for flexibility in creating different types of objects. Lets say you have a website and you want to create a method that will allow you to easily create html objects and add it to the DOM. A factory is the perfect solution for this and here is how we can implement it

1.1. Components of the Factory Method

Creator

This is the method implemented in the Factory that creates new products.

Abstract Product

An interface for the product being created.

Concrete Product

This is the actual object being created.

1.2. Benefits of the Factory Method

Abstraction of Object Creation

It removes the complexity of creating an object, allowing the client code to just focus on the created objects.

Flexibility and Customization

Factories enable customization of the object creation process, allowing for variations in the created objects

Encapsulation of Creation Logic

The creation logic is encapsulated within the factory, making it easier to modify or extend the creation process without affecting client code

Complex Object Creation

Factories are useful when the creation of objects is complex, involves multiple steps, or requires certain conditions to be met.

1.3. Example

```
    function elementFactory(type, text, color){
        const newElement = document.createElement(type)
        newElement.innerText = text
        newElement.style.color = color
        document.body.append(newElement)


    function setText(newText) {
         newElement.innerText = newText;
    }

    function setColor(newColor) {
        newElement.innerText = newColor;
    }

    return {
        newElement,
        setText,
        setColor,
        }

    }

const h1Tag = elementFactory('h1','Initial Text','Blue');

h1Tag.setText('Hello world');

h1Tag.setColor('Red');
```

2. Abstract Factory Method

Abstract factories are another creational design pattern. Its main goal is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern ensures that the created objects are compatible and work together.

2.1. The 4 Components of an Abstract Factory

Abstract Factories

This defines the interface for creating the abstract products, which are related families of objects (e.g. UI components).The abstract factory declares creation methods for each type of product in the family.

Concreate Factories

These are classes that implement the abstract factory interface, providing specific implementations for creating the concrete products. Each concrete factory creates a family of related products (e.g. UI factory might create a button or checkbox).

Abstract Products

These are the interfaces or base classes for the products that the abstract factory creates. Each product type in the family has its own abstract product definition (e.g., Button, Checkbox).

Concrete Products

These are the actual implementations of the abstract products. Each concrete factory creates its own set of concrete products.Concrete products implement the abstract product interfaces defined for their family (e.g., HTMLButton, WindowsButton).

2.2. Benefits of Abstract Factories

Consistency

It ensures that created objects are compatible and follow a consistent theme or style.

Isolation of Responsibilities

It isolates the creation of objects from the client code, promoting a clean separation of concerns.

Flexibility and Scalability

It allows for easy addition of new product families without modifying existing client code.

2.3. Example

```
 // Abstract Factory: UIFactory
class UIFactory {
    createButton() {
        throw new Error('createButton method must be overridden');
    }

    createCheckbox() {
        throw new Error('createCheckbox method must be overridden');
    }
}

// Concrete Factory: WindowsUIFactory
class WindowsUIFactory extends UIFactory {
    createButton() {
        return new WindowsButton();
    }

    createCheckbox() {
        return new WindowsCheckbox();
    }
}

// Concrete Factory: MacUIFactory
class MacUIFactory extends UIFactory {
    createButton() {
        return new MacButton();
    }

    createCheckbox() {
        return new MacCheckbox();
    }
}

// Abstract Product: Button
class Button {
    render() {
        throw new Error('render method must be overridden');
    }
}

// Concrete Product: WindowsButton
class WindowsButton extends Button {
    render() {
        console.log('Rendering a Windows button');
    }
}

// Concrete Product: MacButton
class MacButton extends Button {
    render() {
        console.log('Rendering a Mac button');
    }
}

// Abstract Product: Checkbox
class Checkbox {
    render() {
        throw new Error('render method must be overridden');
    }
}

// Concrete Product: WindowsCheckbox
class WindowsCheckbox extends Checkbox {
    render() {
        console.log('Rendering a Windows checkbox');
    }
}
```

```
// Concrete Product: MacCheckbox
class MacCheckbox extends Checkbox {
    render() {
        console.log('Rendering a Mac checkbox');
    }
}

// Usage
const windowsFactory = new WindowsUIFactory();
const macFactory = new MacUIFactory();

const windowsButton = windowsFactory.createButton();
windowsButton.render();  // Output: Rendering a Windows button

const macCheckbox = macFactory.createCheckbox();
macCheckbox.render();  // Output: Rendering a Mac checkbox
```

3. Builder

The goal of a builder is to separate the construction of an object from its representation. What the builder pattern does is basically allow the client to construct a complex object by just passing in the type and content of the object only. The client does not have to worry about the construction details.

3.1. The 4 Components of a Builder

Builder

The builder usually contains a series of methods to build various parts of the object.

Concrete Builder

Implements methods from the builder interface to construct parts of the object

Director (Optional)

This is not always necessary but can help with constructing the final object using a specific construction process

Object

Representation of the final product. Contains parts that were constructed by the builder

3.2. Benefits of the Builder Pattern

Separation of Concerns

The Builder pattern separates the construction of a complex object from its representation, allowing different implementations of builders to vary the internal representation.

Flexible Object Creation

It allows for the creation of different configurations of a complex object by using a common construction process. Builders can be tailored to create specific variations of the object

Improved Readability

Using a builder can improve code readability by clearly outlining the steps needed to construct an object. It's easy to understand what each step contributes to the final object.

Parameterized Construction

Builders allow you to construct an object by passing parameters to the construction steps, enabling fine-grained control over the object's creation and configuration.

Reusability

Builders can be reused to create multiple instances of the complex object with different configurations, promoting code reuse and minimizing duplication of construction logic.

3.3. Example

```
//Builder
class ComputerBuilder {
    buildCPU() {}
    buildRAM() {}
    buildStorage() {}
    getResult() {}
}

//Concrete Builders
class GamingComputerBuilder extends ComputerBuilder {
    // Implement specific steps to build a gaming computer
}

class OfficeComputerBuilder extends ComputerBuilder {
    // Implement specific steps to build an office computer
}

//Object class
class Computer {
    constructor() {
        this.parts = [];
    }

    addPart(part) {
        this.parts.push(part);
    }
}

// Director (Optional)
class ComputerAssembler {
    constructor(builder) {
        this.builder = builder;
    }

    assembleComputer() {
        this.builder.buildCPU();
        this.builder.buildRAM();
        this.builder.buildStorage();
        return this.builder.getResult();
    }
}
```

4. Singleton

A singleton is an object that can only be instantiated once. Singletons are useful when system wide actions need to be coordinated from a single central place. Singletons reduce the need for global variables which is particularly important in javascript because it limits namespace pollution.

4.1. Components of a Singleton

Anonymous Function

A singleton is implemented using a anonymous function

getInstance Function

This is a function which returns the unique instantiated object

Constructor (Optional)

In javascript, a constructor is not necessary for implementing the singleton pattern but having a constructor is common because it allows you to configure the singleton and add initialization logic.

4.2. Benefits of a Singleton

Reduce Global Variables

Singletons can help reduce the number of global variables required in your program, promoting better code organization and maintainability.

Memory Efficient

Because a Singleton ensures there is only ever one instance that exists at a time, memory is saved because you avoid having multiple instances of the same class.

Global Point of Access

Singletons provide a global point of access to the instance. This allows other parts of the program to access and use the same instance without needing to pass it around.

Resource Sharing

Singletons are especially useful when it comes to tasks like managing shared resources. Singletons can be used to manage database connections, file handlers, and even thread pools, ensuring that these resources are shared efficiently across the application.

4.3. Example

```
class Singleton {
  constructor() {
    const privateVariable = 'This is a private variable';

    function privateMethod() {
      console.log('This is a private method.');
    }

    return {
      publicMethod: function() {
        console.log('This is a public method.');
      },
      publicVariable: 'I am public'
    };
  }

  static getInstance() {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
}

const singletonInstance1 = Singleton.getInstance();
const singletonInstance2 = Singleton.getInstance();

console.log(singletonInstance1 === singletonInstance2); // Outputs: true
```

5. Prototype

The prototype pattern is an alternative way to implement inheritance but the main difference is instead of inheriting properties from a class, objects inherit properties from a prototype object. The prototype pattern is also referred to as the properties pattern and Javascript has native support for prototypes. In Javascript, each object has a prototype (reference to another object). When you attempt to access a property that does not exist in the object itself Javascript will look for it in the object's prototype and continue up the prototype chain until it finds it or reaches the end of the chain.

5.1. Components of the Prototype Pattern

Prototype Object

Contains the properties and methods that all the new instances will inherit

Client

The client is responsible for creating new objects based on the prototype. The client can create new instances based on the prototype and modify their properties accordingly.

Clone/Creation Mechanism

The mechanism used to create a new object based on the prototype. In Javascript this can be achieved using the `Object.create()` function.

5.2. Benefits of the Prototype Pattern

Efficient Instance Creation

Creating new instances of the Prototyope is more efficient that using traditional classes and constructors. Objects are created by cloning the prototype, which reduces the need for setting up classes and initialization logic.

Code Reusability

The Prototype pattern allows you to define a set of default properties and methods in a prototype object. This allows multiple instances to share the same behaviour and structure without duplicating the code. This also reduces memory usage since each instance does not need to store duplicates of the prototypes properties.

Flexible Object Creation

Objects created using the Prototype pattern can be easily customized by modifying their properties or adding new properties specific to the instance.

Dynamic Runtime Changes

Changes made to the prototype object at runtime are reflected in all instances based on the prototype. This behavior allows for updates and modifications to the prototype, impacting all instances sharing the same prototype.

5.3. Example

```
const cameraPrototype = {
    model = 'default',
    make = 'default',
    shutter: function () {
        console.log(`The ${this.make} ${this.model} has taken a photo`);
    }
};

const camera1 = Object.create(cameraPrototype);
camera1.model = 'X-Pro 3';
camera1.make = 'Fujifilm';

const camera1 = Object.create(cameraPrototype);
camera1.model = 'R5';
camera1.make = 'Canon';
```

Structural Design Patterns

Focus on how classes and objects are composed to form larger structures

1. Adapter

The Adapter is a structural design pattern that enables you to make make different interfaces with different methods work together without changing their code. The purpose of an Adapter is to make two incompatible interfaces work together seamlessly.

1.1 Components of the Adapter

Target Interface/Class

This is the interface or class that the client will work with. It contains all the methods and properties that the client code will use.

Adaptee

The Adaptee is the old interface/class that contains properties and methods that are incompatible with the new interface/class.

Adapter

The Adapter is what bridges the gap between the Adaptee and the Target interface/class

1.2. Benefits of Adapters

Easy Integration

Adapters create an easy way for new code or systems to interact with existing ones. By using Adapters, integrating new code becomes smoother and less error-prone.

Compatibility and Reusability

Adapters promote code reuse and extends the usability of existing code by making older code compatible with newer code.

Gradual System Integration

In situations where a new system needs to be implemented gradually, Adapters can serve as intermediaries, allowing new features to come in slowly while maintaining compatibility with the existing system.

Improved Testability

Adapters facilitate easier testing by allowing mocking or stubbing of the adaptee during testing of the client code. This improves the testability of the client code and helps in wrtiting more comprehensible unit tests.

1.3. Example

```
// Adaptee: EU charging brick
class EUChargingBrick {
  chargeWithEUPlug() {
    console.log('Charging with EU plug');
  }
}

// Adaptee: US charging brick
class USChargingBrick {
  chargeWithUSPlug() {
    console.log('Charging with US plug');
  }
}

// Target: Common charging interface expected by the client
class ChargingInterface {
  charge() {
    console.log('Charging...');
  }
}

// Adapter for EU charging brick
class EUChargingAdapter extends ChargingInterface {
  constructor(euChargingBrick) {
    super();
    this.euChargingBrick = euChargingBrick;
  }

  charge() {
    this.euChargingBrick.chargeWithEUPlug();
  }
}

// Adapter for US charging brick
class USChargingAdapter extends ChargingInterface {
  constructor(usChargingBrick) {
    super();
    this.usChargingBrick = usChargingBrick;
  }

  charge() {
    this.usChargingBrick.chargeWithUSPlug();
  }
}

// Client
function chargeDevice(chargingInterface) {
  chargingInterface.charge();
}

// Usage
const euChargingBrick = new EUChargingBrick();
const euAdapter = new EUChargingAdapter(euChargingBrick);

const usChargingBrick = new USChargingBrick();
const usAdapter = new USChargingAdapter(usChargingBrick);

console.log('Charging with EU charging brick:');
chargeDevice(euAdapter);

console.log('Charging with US charging brick:');
chargeDevice(usAdapter);
```

2. Bridge

The Bridge is a structural design pattern that is designed to split a very large class into two separate hierarchies which can be developed independendently. The two hierarchies are referred to as the Abstraction level and the Implementation level. Basically if you have a class that has multiple variants of some functionality, you can use a Bridge pattern to divide and organize the class into two easier to understand hierarchies.

2.1. Components of the Bridge

Abstraction

This is the high-level interface or abstraction. It defines the abstract functionality that the clients will use.

Refined Abstraction

These are subclasses or extensions of the abstraction layer. These provide additional features or refinements. It extends the functionality defined by the abstraction.

Implementor

This is the interface that defines the implementation methods, It usually doesn't mirror the abstraction interface, but its a lower-level interface that the abstraction relies upon.

Concrete Implementor

Concrete classes that implement the implementor interface. Theses classes provide specific implementations of the methods defined by the implementor interface.

2.2. Benefits of the Bridge Pattern

Decouples Abstraction from Implementation

The primary benefit of the Bridge pattern is it splits the abstraction layer from the implementation layer. This allows both sections to evolve independently, making the code base easier to modify.

Improves Maintainability

Since the code base is split into two sections, making changes to one part of the system is most likely not going to impact the other part. Which makes maintaining the code base easier and more efficient

Improves Testing

Testing is a lot easier when you have a bridge pattern in your code base because you can focus on testing the abstraction layer separately from testing the implementation layer. This allows for easier and more targeted testing.

Improves Readability

The Bridge pattern creates a clear hierarchy in the code base. Organzing the code base in this way helps in understanding the relationships between different parts of the system.

2.3. Example

```
// Abstraction
class Shape {
  constructor(color) {
    this.color = color;
  }

  draw() {
    console.log(`Drawing a shape with color ${this.color}`);
  }
}

// Implementations
class RedColor {
  applyColor() {
    console.log('Applying red color');
  }
}

class BlueColor {
  applyColor() {
    console.log('Applying blue color');
  }
}

// Bridge
class ShapeWithColor extends Shape {
  constructor(color, colorImplementation) {
    super(color);
    this.colorImplementation = colorImplementation;
  }

  draw() {
    super.draw();
    this.colorImplementation.applyColor();
  }
}

// Usage
const redShape = new ShapeWithColor('red', new RedColor());
const blueShape = new ShapeWithColor('blue', new BlueColor());

redShape.draw();  // Output: Drawing a shape with color red, Applying red color
blueShape.draw(); // Output: Drawing a shape with color blue, Applying blue color
```

3. Composite

The composite design pattern allows for the creation of objects with properties that are primitive items or a collection of objects. Imagine a tree like structure, where you have single objects (leaf nodes) or groups of objects (branches). The composite design pattern allows you to create this type of structure and be able to perform operations on each level in a consistent manner.

3.1 Components of the Composite

Component

This is the interface/class that represents both leaf nodes (individual elements) and composite nodes (collection of elements). The component defines operations that can be applied to both types of nodes.

Leaf

This represents individual objects in the tree that do not have any children. They implement the operations that are defined in the component interface.

Composite

This represents the composites or containers that can hold a collection of leaf nodes or other composite nodes.

3.2. Benefits of Composites

Uniformly and Consistency

The Composite design pattern provides a uniform way to treat both individual objects and compositions of objects. Clients have one common interface to use to operate on these objects which simplifes the code base and object interactions.

Flexibility

This design pattern allows for flexibility in adding new types of components or modifying existing ones without affecting the client code. You can introduce new types of leaf or composite objects easily.

Simplified Client Code

The client code doesn't need to distinguish between individual and composite components, making it simpler and more intuitive to work with the structure.

3.3. Example

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

  display() {
    console.log('Block:', this.name);
  }
}

class BlockCollection {
  constructor(name) {
    this.name = name;
    this.blocks = [];
  }

  add(block) {
    this.blocks.push(block);
  }

  remove(block) {
    this.blocks = this.blocks.filter(b => b !== block);
  }

  display() {
    console.log('Block Collection:', this.name);

    for (const block of this.blocks) {
      block.display();
    }
  }
}

// Usage
const block1 = new SingleBlock('Block 1');
const block2 = new SingleBlock('Block 2');
const block3 = new SingleBlock('Block 3');

const blockGroup1 = new BlockCollection('Block Group 1');
blockGroup1.add(block1);
blockGroup1.add(block2);

const blockGroup2 = new BlockCollection('Block Group 2');
blockGroup2.add(block3);

const megaStructure = new BlockCollection('Mega Structure');
megaStructure.add(blockGroup1);
megaStructure.add(blockGroup2);

megaStructure.display();
```

4. Decorator

The Decorator design pattern can be used to modify an objects behavior either statically or dynamically without affecting the behavior of other objects from the same class. Decorators are particularly useful when you want to add features to an object in a flexible and reusable way.

4.1. Components of a Decorator

Component Interface

This defines the logic for the objects that can have resposibilities added to them dynamically.

Concrete Components

This is the initial object to which additional functionalities can be added.

Decorator

This is an interface that extends the functionality of the concrete components. It has a reference to a component instance and implements the component interface.

Concrete Decorators

These are the concrete implementations of the decorator class, they add specific behavior to the desired component by extending the decorator class.

4.2. Benefits of Decorators

Extensibility and Flexibility

Decorators allow you to add new functionalities or behaviors to objects dynamically at runtime. This promotes extensibility without modifying the existing codebase and provides flexibility in how you can compose and use these additional functionalities.

Modularity

Decorators enable a more modular approach to code by breaking down functionality into smaller, more manageable units. These units can be combined and reused in various ways.

Runtime Configuration

Decorators allow you to dynamically configure an object at runtime. This allows you to add or remove functionalities without impacting the core components or needing to recompile the code.

Reduce Subclassing

Without Decorators, extending functionalities often involves creating numerous subclasses for each combination of behaviors. Decorators eliminate the need for subclasses which results in an cleaner and easier to understand code base.

4.3. Example

```
class Coffee {
  cost() {
    return 5;
  }
}

class MilkDecorator {
  constructor(coffee) {
    this.coffee = coffee;
  }

  cost() {
    return this.coffee.cost() + 2;
  }
}

class SugarDecorator {
  constructor(coffee) {
    this.coffee = coffee;
  }

  cost() {
    return this.coffee.cost() + 1;
  }
}

// Usage
let coffee = new Coffee();
console.log('Cost of plain coffee:', coffee.cost());

let milkCoffee = new MilkDecorator(coffee);
console.log('Cost of milk coffee:', milkCoffee.cost());

let sugarMilkCoffee = new SugarDecorator(milkCoffee);
console.log('Cost of sugar milk coffee:', sugarMilkCoffee.cost());
```

5. Facade

The Facade design pattern is basically a simplified interface that the client can interact with to use other low level operations hidden elsewhere in the code base. This design pattern is often used in systems that are built around a multi-layer architecture. Facades allow the client to perform certain tasks without needing to understand the complexity of the underlying system.

5.1. Components of the Facade

Facade

The facade is the interface that the client will interact with. It provides a simple and unified interface that delegates the clients requests to the appropriate objects within the subsystem

Subsystem

The subsystem consists of all the various components and functionalities that the Facade wraps around. The subsystem and the Facade interact with eachother but operate independently.

5.2. Benefits of the Facade

Simplified Interface

The Facade provides a simple and easy to understand interface

Code Organization

The Facade helps organize the code by encapsulating the subsystem's functionality and providing a clear separation of concerns

Easier Maintenance

Changes to the subsystem can be isolated within the facade, reducing the impact on the client code.

5.3. Example

```
// Plumbing subsystem
class PlumbingSubsystem {
  constructor() {}

  turnOnWater() {
    console.log('Plumbing: Water turned on');
  }

  turnOffWater() {
    console.log('Plumbing: Water turned off');
  }
}

// Electrical subsystem
class ElectricalSubsystem {
  constructor() {}

  turnOnElectricity() {
    console.log('Electrical: Electricity turned on');
  }

  turnOffElectricity() {
    console.log('Electrical: Electricity turned off');
  }
}

// House facade
class HouseFacade {
  constructor() {
    this.plumbingSubsystem = new PlumbingSubsystem();
    this.electricalSubsystem = new ElectricalSubsystem();
  }

  enterHouse() {
    this.plumbingSubsystem.turnOnWater();
    this.electricalSubsystem.turnOnElectricity();
    console.log('You have entered the house.');
  }

  leaveHouse() {
    this.plumbingSubsystem.turnOffWater();
    this.electricalSubsystem.turnOffElectricity();
    console.log('You have left the house.');
  }
}

// Client
const client = () => {
  const house = new HouseFacade();

  // Enter the house
  house.enterHouse();

  // Leave the house
  house.leaveHouse();
};

// Run the client
client();
```

6. Flyweight

The Flyweight design pattern aims to minimize memory usage or computaional expenses by storing intrinsic values (similar properties) of similar object in an application, reducing the amount of duplicate code. This is particularly useful when dealing with a large number of similar objects in an application.

6.1. Components of a Flyweight

FlyweightFactory

The flyweight factory creates the flyweight objects. It contains a function that will create a flyweight if one does not already exist and it stores newly created flyweights for future request.

Flyweight

The flyweight contains the intrinsic data that will be shared across the application

6.2. Benefits of Flyweights

Memory Efficiency

By sharing intrinsic data among multiple objects, the Flyweight pattern significantly reduces memory usage especially when dealing with a large number of instances.

Performance Improvements

Due to reduced memory usage, the application's overall performance improves. Lower memory usage typically leads to faster execution times and smoother application performance.

Simplifies State Management

By separating intrinsic data (shared values) and extrinisc data (unique values), Flyweights simplify the management of these states. It allows for a cleaner separation of concerns and more organized approach to state handling.

6.3. Example

```
// Flyweight object for Camera
function Camera(make, model, resolution) {
    this.make = make;
    this.model = model;
    this.resolution = resolution;
}

// Flyweight factory for Camera
var FlyWeightCameraFactory = (function () {
    var flyweights = {};

    return {
        get: function (make, model, resolution) {
            if (!flyweights[make + model]) {
                flyweights[make + model] = new Camera(make, model, resolution);
            }
            return flyweights[make + model];
        },

        getCount: function () {
            var count = 0;
            for (var f in flyweights) count++;
            return count;
        }
    };
})();

// Camera collection
function CameraCollection() {
    var cameras = {};
    var count = 0;

    return {
        add: function (make, model, resolution, serial) {
            cameras[serial] = {
                flyweight: FlyWeightCameraFactory.get(make, model, resolution),
                serial: serial
            };
            count++;
        },

        get: function (serial) {
            return cameras[serial];
        },

        getCount: function () {
            return count;
        }
    };
}

// Run the example
function run() {
    var cameras = new CameraCollection();

    cameras.add("Canon", "EOS R5", "45MP", "A1234");
    cameras.add("Nikon", "D850", "45.7MP", "B5678");
    cameras.add("Sony", "A7R III", "42.4MP", "C9101");
    cameras.add("Canon", "EOS R5", "45MP", "D1212"); // Reusing existing flyweight

    console.log("Cameras: " + cameras.getCount());
    console.log("Flyweights: " + FlyWeightCameraFactory.getCount());
}

// Run the example
run();
```

7. Proxy

The Proxy design pattern is a structural design pattern that allows you to provide a substitute or placeholder for another object. This proxy object can control access to the original object. In Javascript, the `proxy` object is built into the language and facilitates the implementation of the Proxy design pattern.

7.1. Components of the Proxy

Proxy

The Proxy contains an interface that is similar to the real object, it maintains a reference that lets the proxy access the real object and it handles requests and forwards them to the real object.

RealSubject

This is the actual object that the Proxy is substituting for.

7.2. Benefits of Proxies

Controlled Access

Proxies allow you to control access to the original object, enabling you to implement access control logic such as permissions, restrictions, or validations before allowing access to the underlying object.

Behavior Augmentation

Proxies can add additional behavior or functionality before or after the invocation of methods or access to properties of the original object. This is useful for implementing cross-cutting concerns like logging, caching, or error handling.

Caching

Proxies can implement caching mechanism to store results of expensive operations, improving performance and efficiency

Lazy Initialization

Proxies enable lazy initialization, where you can delay the creation of the actual object until its needed. This can improve performance by reducing upfront resource usage.

7.3. Example

```javascript
// Original object representing a bank account
const bankAccount = {
  balance: 1000,

  deposit(amount) {
    this.balance += amount;
    console.log(`Deposited ${amount}. New balance: ${this.balance}`);
  },

  withdraw(amount) {
    if (amount <= this.balance) {
      this.balance -= amount;
      console.log(`Withdrew ${amount}. New balance: ${this.balance}`);
    } else {
      console.log('Insufficient funds.');
    }
  }
};

// Create a proxy for the bank account
const bankAccountProxy = new Proxy(bankAccount, {
  // Intercept property access
  get(target, property) {
    if (property === 'balance') {
      // Add some custom behavior before accessing 'balance'
      console.log('Balance accessed.');
    }
    return target[property];
  },

  // Intercept method invocation
  apply(target, thisArg, args) {
    // Add some custom behavior before invoking a method
    console.log(`Method "${args[0]}" called.`);
    return target.apply(thisArg, args);
  }
});

// Accessing the proxy
console.log(bankAccountProxy.balance); // This will trigger the custom behavior

bankAccountProxy.deposit(500); // This will trigger the custom behavior for method invocation
```

Behavioral Design Patterns

Focus on how objects communicate with each other and assigning responsibilities to them.

1. Chain of Responsibility

The Chain of Responsibility is a behavioural design pattern and its main purpose is to take a request and pass it along a chain of handlers. When the request goes through the chain each handler will decide either to process the request or pass it to the next handler in the chain. This pattern allows multiple handlers to handle the request without the sender needing to know which one will process it.

1.1. Components of the Chain of Responsibility

Request

The request is just the object the client sends that needs to be processed. The request goes through the chain of handlers until it is processed or reaches the end of the chain.

Abstract Handler Interface/Class

This is the base interface/class that defines the methods that will be used for handling the request. The handler interface contains the logic for the order of the chain and how the request gets passed through.

Concrete Handlers

These are the methods/classes that implement the abstract handler. Each concrete handler contains logic for handling a particular type of request. If the concrete handler can process the request it will, if it can't then it will pass the request to the next handler.

1.2. Benefits of the Chain of Responsibility

Ease of Use

The sender does not need to know the specific method to process the request, the sender just needs to pass it to the chain. This makes it easier for the sender to process requests.

Flexibility and Extensibility

New Handlers can be added to the chain or removed from the chain without modifying the sender's code. This allows for dynamic modification of the processing order.

Promotes Responsible Segregation

Handlers are responsible for processing specific types of requests based on their defined logic. This promotes a clear separation of responsibilities, making it easier to manage and maintain each handler independently.

Sequential Request Processing

The pattern ensures that each request is processed sequentially through the chain of handlers. Each handler can choose to process the request or pass it to the next handler. This can be particularly useful in scenarios where requests need to be processed in a specific order.

1.3. Example

```
// Handler interface
class CoffeeHandler {
  constructor() {
    this.nextHandler = null;
  }

  setNext(handler) {
    this.nextHandler = handler;
  }

  processRequest(request) {
    throw new Error('processRequest method must be implemented by subclasses');
  }
}

// Concrete handler for ordering coffee
class OrderCoffeeHandler extends CoffeeHandler {
  processRequest(request) {
    if (request === 'Coffee') {
      return 'Order placed for a cup of coffee.';
    } else if (this.nextHandler) {
      return this.nextHandler.processRequest(request);
    } else {
      return 'Sorry, we do not serve ' + request + '.';
    }
  }
}

// Concrete handler for preparing coffee
class PrepareCoffeeHandler extends CoffeeHandler {
  processRequest(request) {
    if (request === 'PrepareCoffee') {
      return 'Coffee is being prepared.';
    } else if (this.nextHandler) {
      return this.nextHandler.processRequest(request);
    } else {
      return 'Cannot prepare ' + request + '.';
    }
  }
}

// Client code
const orderHandler = new OrderCoffeeHandler();
const prepareHandler = new PrepareCoffeeHandler();

// Set up the chain
orderHandler.setNext(prepareHandler);

// Order coffee
console.log(orderHandler.processRequest('Coffee'));  // Output: Order placed for a cup of coffee.

// Prepare coffee
console.log(orderHandler.processRequest('PrepareCoffee'));  // Output: Coffee is being prepared.

// Try ordering something else
console.log(orderHandler.processRequest('Tea'));  // Output: Sorry, we do not serve Tea.
```

2. Command

The command design pattern is a behavioral design pattern that allows you to encapsulate a request as an object, that object will contain all the necessary information for the request's execution. This pattern allows for the parameterization and queuing of requests and provides the ability to undo operations.

2.1. Components of the Command

Invoker

This is the object that requests the execution of a command. It has a reference to a command and can execute the command by calling its `execute` method. The Invoker does not need to know the specifics of how the command is executed. It just triggers the command.

Command

This is the interface or abstract class that declares the `execute` method. It defines the common method that concrete command classes should implement.

Receiver

This is an object that performs the actual work when the `execute` method of a command is called. The receiver knows how to carry out the action associate with a specific command.

2.2. Benefits of the Command

Flexibility and Extensibility

This pattern allows for easy addition of new commands without needing to modify the invoker or receiver.

Undo and Redo Operations

The command pattern facilitates the implementation of undo and redo operations.Each command object can keep track of the previous state, enabling the reversal of the executed action.

Parameterization and Queuing

Commands can be parameterized with arguments, allowing for the customization of actions at runtime. Additionally, the pattern enables the queuing and scheduling of requests, providing control over the order of execution.

History and Logging

It's possible to maintain a history of executed commands, which can be useful for auditing, logging, or tracking user actions.

2.3. Example

```
class Command {
  constructor(receiver) {
    this.receiver = receiver;
  }

  execute() {
    throw new Error('execute() method must be implemented');
  }
}

class ConcreteCommand extends Command {
  constructor(receiver, parameter) {
    super(receiver);
    this.parameter = parameter;
  }

  execute() {
    this.receiver.action(this.parameter);
  }
}

class Receiver {
  action(parameter) {
    console.log(`Receiver is performing action with parameter: ${parameter}`);
  }
}

class Invoker {
  constructor() {
    this.commands = [];
  }

  addCommand(command) {
    this.commands.push(command);
  }

  executeCommands() {
    this.commands.forEach(command => command.execute());
    this.commands = []; // Clear the executed commands
  }
}

// Usage
const receiver = new Receiver();
const command1 = new ConcreteCommand(receiver, 'Command 1 parameter');
const command2 = new ConcreteCommand(receiver, 'Command 2 parameter');

const invoker = new Invoker();
invoker.addCommand(command1);
invoker.addCommand(command2);

invoker.executeCommands();
```

3. Interperator

The interperator design pattern is used to define a grammar for a language and provide an interpreter to interpret sentences in that language. It's typically use for creating language interpreters or parsers but you could also use them inside your application. Imagine you have a complex desktop application, you could design a basic scripting language which allows the end-user to manipulate your application through simple instructions.

3.1. Components of the Interperator

Context

A global state or context that the Interpreter uses to interpret expressions. It often contains information that is relevant during the interpretation of expressions.

Abstract Expressions

Defines an interface for all types of expressions in the language's grammar. These expressions are typically represented as an abstract class or interface.

Terminal Expressions

Represents the terminal symbols in the grammar. These are the leaves of the expression tree. Terminal expression implement the interface defined by the abstract expression.

Non-terminal Expression

Represents the non-terminal symbols in the grammar. Non-terminal expressions use terminal and/or other non-terminal expressions to define complex expressions by combining or composing them.

Expression Tree

Represents the hierarchical structure of the language's expressions. The expression tree is built by combining terminal and non-terminal expressions based on the rules of the language's grammar.

Interpreter

Defines an interface or class that interprets the abstract syntax tree created by the expression tree. It typically includes an `interpret` method that takes a context and interprets the expression based on the given context.

client

Builds the abstract syntax tree using terminal and non-terminal expressions based on the language's grammar. The client then uses the interpreter to interpret the expression.

3.2. Benefits of Interpreters

Ease of Grammar Interpretation

The pattern simplifies the interpretation of complex grammatical rules by breaking them down into smaller, more manageable expressions. Each expression type handles its own interpretation, making the overall interpretation process easier to manage.

Better Error Handling

Since each expression type handles its own interpretation, error handling can be tailored to specific expressions. This allows for more precise and meaningful error messages when parsing or interpreting the input.

Flexibility and Extensibility

The interpreter pattern provides a flexible way to define and extend grammatical rules and language expressions without modifying the core interpreter logic. You can easily add new expressions by creating new terminal and non-terminal expression classes.

Integration with other Design Patterns

The Interpreter pattern can be combined with other design patterns, such as Composite, to build complex hierarchies of expressions. This allows for the creation of powerful and feature-rich interpreters.

3.3. Example

```
// Abstract Expression
class Expression {
  interpret(context) {
    // To be overridden by subclasses
  }
}

// Terminal Expression: NumberExpression
class NumberExpression extends Expression {
  constructor(number) {
    super();
    this.number = number;
  }

  interpret(context) {
    return this.number;
  }
}

// Terminal Expression: VariableExpression
class VariableExpression extends Expression {
  constructor(variable) {
    super();
    this.variable = variable;
  }

  interpret(context) {
    return context[this.variable] || 0;
  }
}

// Non-terminal Expression: AddExpression
class AddExpression extends Expression {
  constructor(expression1, expression2) {
    super();
    this.expression1 = expression1;
    this.expression2 = expression2;
  }

  interpret(context) {
    return this.expression1.interpret(context) + this.expression2.interpret(context);
  }
}

// Non-terminal Expression: SubtractExpression
class SubtractExpression extends Expression {
  constructor(expression1, expression2) {
    super();
    this.expression1 = expression1;
    this.expression2 = expression2;
  }

  interpret(context) {
    return this.expression1.interpret(context) - this.expression2.interpret(context);
  }
}

// Client code
const context = { a: 10, b: 5, c: 2 };

const expression = new SubtractExpression(
  new AddExpression(
    new VariableExpression('a'),
    new VariableExpression('b')
  ),
  new VariableExpression('c')
);
```

```
const result = expression.interpret(context);
console.log('Result:', result); // Output: Result: 13
```

4. Iterator

The Iterator pattern allows clients to effectively loop over a collection of objects

4.1. Components of the Iterator

Iterator

Implements Iterator interface or class with methods like `first()` , `next()` . The Iterator keeps track of the current position when traversing collections.

Items

These are the individual objects of the collection that the Iterator will traverse

4.2. Benefits of iterators

Compatibility with Different Data Structures

The Iterator pattern allows the same iteration logic to be applied to different data structures.

Support for concurrent Iteration

Iterators can support concurrent iteration over the same collection without interfering with each other, this enables the client to iterate over the collection in different ways simultaneously.

Lazy Loading

Iterators can be designed to load elements on demand, which is beneficial for large collections where loading all elements at once might be impractical or resource-intensive. Elements can be fetched as needed, optimizing memory usage.

Simplified Interface

The Iterator pattern provides a clean and consistent interface for accessing elements in a collection without exposing the internal structure of the collection. This simplifies the usage and understanding of the collection.

4.3. Example

```
// Car class representing a car
class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }

  getInfo() {
    return `${this.make} ${this.model}`;
  }
}

// Iterator interface
class Iterator {
  constructor(collection) {
    this.collection = collection;
    this.index = 0;
  }

  next() {
    return this.collection[this.index++];
  }

  hasNext() {
    return this.index < this.collection.length;
  }
}

// Collection of cars
class CarCollection {
  constructor() {
    this.cars = [];
  }

  addCar(car) {
    this.cars.push(car);
  }

  createIterator() {
    return new Iterator(this.cars);
  }
}

// Usage
const carCollection = new CarCollection();
carCollection.addCar(new Car('Toyota', 'Corolla'));
carCollection.addCar(new Car('Honda', 'Civic'));
carCollection.addCar(new Car('Ford', 'Mustang'));

const iterator = carCollection.createIterator();

while (iterator.hasNext()) {
  const car = iterator.next();
  console.log(car.getInfo());
}
```

5. Mediator

The Mediator pattern acts as a middle man between a group of objects by encapsulating how these objects interact. The mediator facilitates communication between different components of a system without them needing to directly reference each other.

5.1. Components of the Mediator

Mediator

The mediator manages central control over operations. It contains an interface for communicating with the Colleague objects and has a reference to each Colleague object.

Colleague

The colleagues are the objects that are being mediated, each colleague has a reference to the mediator.

5.2. Benefits of the Mediator

Centralized Control

Centralizing communication within the Mediator allows for better control and coordination of interactions between components. The ,Mediator can manage message distribution, prioritize messages, and apply specific logic based on the system's requirements.

Simplified Communication

Mediators simplify communication logic, as components send messages to the Mediator instead of dealing with the complexity of direct communication. This simplifies the interaction between components and allows for easier maintenance and updates.

Reusability of Mediator

The Mediator can be reused across various components and scenarios, allowing for a single point of control for different parts of the application. This reusability promotes consistency and helps in managing the communication flow efficiently.

Promotes Maintainability

The Mediator pattern promotes maintainability by reducing the complexity of direct inter-component communication. As the system grows, changes and updates can be made within the Mediator without affecting the individual components, making maintenance easier and less error-prone.

```
var Participant = function (name) {
    this.name = name;
    this.chatroom = null;
};

Participant.prototype = {
    send: function (message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function (message, from) {
        console.log(from.name + " to " + this.name + ": " + message);
    }
};

var Chatroom = function () {
    var participants = {};

    return {

        register: function (participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },

        send: function (message, from, to) {
            if (to) {                      // single message
                to.receive(message, from);
            } else {                       // broadcast message
                for (key in participants) {
                    if (participants[key] !== from) {
                        participants[key].receive(message, from);
                    }
                }
            }
        }
    };
};

function run() {

    var yoko = new Participant("Yoko");
    var john = new Participant("John");
    var paul = new Participant("Paul");
    var ringo = new Participant("Ringo");

    var chatroom = new Chatroom();
    chatroom.register(yoko);
    chatroom.register(john);
    chatroom.register(paul);
    chatroom.register(ringo);

    yoko.send("All you need is love.");
    yoko.send("I love you John.");
    john.send("Hey, no need to broadcast", yoko);
    paul.send("Ha, I heard that!");
    ringo.send("Paul, what do you think?", paul);
}
```

6. Memento

The Memento design pattern allows an objects state to be captured and restored at a later time without exposing its internal structure. The Memento is a separate object that stores the state of the original object.

6.1. Components of the Memento

Originator

This is the object that gets saved. It creates a Memento object to store its internal state.

Memento

The Memento is responsible for storing the state of the originator. It has methods to retrieve and set the state of the originator but does not expose the originator's internal structure.

Caretaker

The caretaker is responsible for keeping track and managing the Mementos. It does not modify or inspect the contents of the Mementos

6.2. Benefits of the Memento

State Preservation and Restoration

Mementos allow an objects internal state to be captured and restored at a later time.

Undo/Redo Operations

Memento facilitates implementing undo and redo functionality easily. Since Mementos store the objects state at various points in time, you can support undoing changes made to the object or redoing changes made to the object

Improved Performance

Storing the object's state in a Memento allows for more efficient storage and retrieval operations compared to other approaches.

Flexible Design

It provides a flexible way to manage the history of an object's state. The caretaker can decide which states to keep and when to restore them, allowing for a customizable approach based on the application's requirements.

6.3. Example

```javascript
// Computer class (originator)
class Computer {
  constructor() {
    this.os = '';
    this.version = '';
  }

  setOS(os, version) {
    this.os = os;
    this.version = version;
  }

  getState() {
    return {
      os: this.os,
      version: this.version
    };
  }

  restoreState(state) {
    this.os = state.os;
    this.version = state.version;
  }
}

// Caretaker
class Caretaker {
  constructor() {
    this.mementos = {};
    this.nextKey = 1;
  }

  add(memento) {
    const key = this.nextKey++;
    this.mementos[key] = memento;
    return key;
  }

  get(key) {
    return this.mementos[key];
  }
}

function run() {
  const computer = new Computer();
  const caretaker = new Caretaker();

  // Save state
  const originalState = computer.getState();
  const key = caretaker.add(originalState);

  // Mess up the state
  computer.setOS('Windows', '11');

  // Restore original state
  const restoredState = caretaker.get(key);
  computer.restoreState(restoredState);

  console.log(computer.getState()); // Output: { os: '', version: '' }
}

run();
```

7. Observer

The observer pattern allows many objects to subscribe to events that are broadcast by another object.

7.1. Components of the Observer

Subject

The subject is an object that implements an interface that lets observer objects subscribe to it and send notifications to observers when its state changes.

Observers

The Observer subscribes to the subject and and typically has a function that get invoked when notified by the Subject

7.2. Benefits of Observers

Simplified Event Handling

The Observer pattern can simplify event handling mechanisms, as events can be treated as notifications to observers about a state change.

Reduces Duplicate code

Instead of duplicating the same code to respond to state changes in multiple places, the Observer pattern allows for a centralized place to manage these responses, promoting cleaner and more maintainable code.

Support for Broadcast Communication

The Observer pattern facilitates a "one-to-many" communication model, where a single event triggers actions in multiple observers. This is useful in scenarios where multiple components need to react to a change in state.

Modularity and Resusability

Observers can be reused across different subjects, promoting modularity and code reusability. This allows for more flexible and maintainable code.

7.3. Example

```
function Click() {
    this.handlers = [];  // observers
}

Click.prototype = {

    subscribe: function (fn) {
        this.handlers.push(fn);
    },

    unsubscribe: function (fn) {
        this.handlers = this.handlers.filter(
            function (item) {
                if (item !== fn) {
                    return item;
                }
            }
        );
    },

    fire: function (o, thisObj) {
        var scope = thisObj || window;
        this.handlers.forEach(function (item) {
            item.call(scope, o);
        });
    }
}

function run() {

    var clickHandler = function (item) {
        console.log("fired: " + item);
    };

    var click = new Click();

    click.subscribe(clickHandler);
    click.fire('event #1');
    click.unsubscribe(clickHandler);
    click.fire('event #2');
    click.subscribe(clickHandler);
    click.fire('event #3');
}
```

8. State

The State pattern is a behavioral design pattern that allows you to have a base object and provide it with additional functionality based on its state. This pattern is particularly useful when an object needs to change its behavior based on its internal state.

8.1. Components of the State

State

This is the object that encapsulates the State values and the associated behavior of the State.

Context

This is the object that maintains a reference to a State object that defines the current State. It also includes an interface that allows other State objects to change its current State to a different State.

8.2. Benefits of the State

Modular and Organized Code

Each State is encapuslated within its own class, making the code modular and easy to manage.

No Need for Switch Statements

Switch statements can also be used to change the behavior of an object but the problem with this method is that switch statements can become very lengthy as your project scales. The State pattern fixes this issue.

Promotes Reusability

States can be reused across different contexts, this reduces code duplication.

Simplifies Testing

Testing individual state classes in isolation is easier and more effective than testing a monolithic object with complex conditional logic.

8.3. Example

```
class Car {
  constructor() {
    this.state = new ParkState();
  }

  setState(state) {
    this.state = state;
    console.log(`Changed state to: ${state.constructor.name}`);
  }

  park() {
    this.state.park(this);
  }

  drive() {
    this.state.drive(this);
  }

  reverse() {
    this.state.reverse(this);
  }
}

class ParkState {
  park(car) {
    console.log("Car is already in Park");
  }

  drive(car) {
    console.log("Shifting to Drive");
    car.setState(new DriveState());
  }

  reverse(car) {
    console.log("Shifting to Reverse");
    car.setState(new ReverseState());
  }
}

class DriveState {
  park(car) {
    console.log("Shifting to Park");
    car.setState(new ParkState());
  }

  drive(car) {
    console.log("Car is already in Drive");
  }

  reverse(car) {
    console.log("Shifting to Reverse");
    car.setState(new ReverseState());
  }
}

class ReverseState {
  park(car) {
    console.log("Shifting to Park");
    car.setState(new ParkState());
  }

  drive(car) {
    console.log("Shifting to Drive");
    car.setState(new DriveState());
  }

  reverse(car) {
    console.log("Car is already in Reverse");
  }
}

// Example usage
const car = new Car();

car.drive();
car.reverse();
car.drive();
car.park();
car.drive();  // Trying to drive while parked
```

9. Strategy

The Strategy pattern is essentially a design pattern that allows you to have a group of algorithms (strategies) that are interchangeable.

9.1. Components of Strategy

Strategy

This is an algorithm that implements the Strategy interface.

Context

This is the object that maintains a reference to the current strategy. It defines an interface that allows the client to change the current Strategy to a different Strategy or retrieve calculations from the current Strategy referenced.

9.2. Benefits of Strategy

Dynamically Swappable Algorithms

Strategies can be swapped at runtime, allowing for dynamic selection of algorithms based on different conditions or requirements. This is particularly useful when the appropriate algorithm may vary based on user input, configuration settings, or other factors.

Flexibility and Maintainability

Strategies can be changed or extended without modifying the context using them. This makes the system more flexible and easier to maintain since changes in one strategy do not affect others.

Simplifies Testing

Testing strategies in isolation is easier since each strategy is a separate class. This allows for targeted testing and ensures that changes to one strategy don't inadvertently affect others.

Reusability

Strategies can be reused in multiple contexts or applications, promoting code reuse and minimizing redundancy.

9.3. Example

```
class RegularCustomerStrategy {
  calculatePrice(bookPrice) {
    // Regular customers get a fixed discount of 10%
    return bookPrice * 0.9;
  }
}

class VIPCustomerStrategy {
  calculatePrice(bookPrice) {
    // VIP customers get a fixed discount of 20%
    return bookPrice * 0.8;
  }
}

class BookStore {
  constructor(pricingStrategy) {
    this.pricingStrategy = pricingStrategy;
  }

  setPricingStrategy(pricingStrategy) {
    this.pricingStrategy = pricingStrategy;
  }

  calculatePrice(bookPrice) {
    return this.pricingStrategy.calculatePrice(bookPrice);
  }
}

// Usage
const regularCustomerStrategy = new RegularCustomerStrategy();
const vipCustomerStrategy = new VIPCustomerStrategy();

const bookstore = new BookStore(regularCustomerStrategy);

console.log('Regular customer price:', bookstore.calculatePrice(50)); // Outputs: 45 (10% discount)
bookstore.setPricingStrategy(vipCustomerStrategy);
console.log('VIP customer price:', bookstore.calculatePrice(50)); // Outputs: 40 (20% discount)
```

10. Template Method

The Template Method is a behavioral design pattern that defines the program skeleton of an algorithm in a method but lets subclasses override specific steps of the algorithm without changing its structure.

10.1. Components of the Template Method

Abstract Class

The abstract class is the template for the algorithm. It defines an interface for the client to invoke its method. It also contains all the functions that can be overridden by subclasses.

Concrete Class

Implements the steps as defined in the Abstract Class and can make changes to the steps.

10.2. Benefits of the Template Method

Code Reusability

The pattern promotes code reusability by defining the algorithm's skeleton in a base class. Subclasses can reuse this structure and only need to provide implementations for specific steps.

Easy Maintenance

Making changes to the algorithm is simplified because modifications only need to be made in one place—the template method in the abstract class—rather than in multiple subclasses. This reduces the chances of errors and makes maintenance more straightforward.

Extension and Variation

The pattern allows for easy extension and variation of the algorithm. Subclasses can override certain steps to provide custom implementations, effectively extending or modifying the behavior of the algorithm without altering its core structure.

Control Flow

The template method defines the control flow of the algorithm, making it easier to manage and understand the sequence of operations in the algorithm.

10.3. Example

```javascript
class Camera {
  // Template method defining the common steps for capturing a photo
  capturePhoto() {
    this.turnOn();
    this.initialize();
    this.setExposure();
    this.capture();
    this.turnOff();
  }

  // Common steps for turning on the camera
  turnOn() {
    console.log('Turning on the camera');
  }

  // Abstract method for initializing the camera (to be overridden by subclasses)
  initialize() {
    throw new Error('Abstract method: initialize() must be implemented by subclasses');
  }

  // Abstract method for setting exposure (to be overridden by subclasses)
  setExposure() {
    throw new Error('Abstract method: setExposure() must be implemented by subclasses');
  }

  // Common steps for capturing a photo
  capture() {
    console.log('Capturing a photo');
  }

  // Common steps for turning off the camera
  turnOff() {
    console.log('Turning off the camera');
  }
}

class DSLRCamera extends Camera {
  initialize() {
    console.log('Initializing DSLR camera');
  }

  setExposure() {
    console.log('Setting exposure for DSLR camera');
  }
}

class MirrorlessCamera extends Camera {
  initialize() {
    console.log('Initializing mirrorless camera');
  }

  setExposure() {
    console.log('Setting exposure for mirrorless camera');
  }
}

// Usage
const dslrCamera = new DSLRCamera();
console.log('Capturing photo with DSLR camera:');
dslrCamera.capturePhoto();
console.log('');

const mirrorlessCamera = new MirrorlessCamera();
console.log('Capturing photo with mirrorless camera:');
mirrorlessCamera.capturePhoto();
```

11. Visitor

The visitor design pattern is a behavioral design pattern that allows you to separate algorithms or operations from the object on which they operate.

11.1 Components of the Visitor

ObjectStructure

Maintains a collection of Elements which can be iterated over.

Elements

The element contains an accept method that accepts the visitor objects.

Visitor

Implements a visit method where the argument of the method is the element being visited. This is how changes to the element get made.

11.2. Benefits of the Visitor

Open/Closed Principle

The pattern aligns with the Open/Closed Principle, which states that software entities (classes, modules, functions) should be open for extension but closed for modification. You can introduce new operations (new visitors) without modifying the existing object structure or elements.

Extensibility

You can introduce new behaviors or operations by adding new visitor implementations without modifying the existing elements or the object structure. This makes the system more extensible, allowing for new features or behaviors to be added easily.

Centralized Behavior

The Visitor pattern centralizes behavior-related code within the visitor classes. Each visitor encapsulates a specific behavior, which can be reused across different elements, promoting code reuse and modularity.

Consistency in Operations

With the Visitor pattern, you can ensure that a specific operation (visitor method) is applied consistently across various elements, as each element's accept method calls the appropriate visitor method for that element type.

11.3 Example

```
class GymMember {
    constructor(name, subscriptionType, fitnessScore) {
        this.name = name;
        this.subscriptionType = subscriptionType;
        this.fitnessScore = fitnessScore;
    }

    accept(visitor) {
        visitor.visit(this);
    }

    getName() {
        return this.name;
    }

    getSubscriptionType() {
        return this.subscriptionType;
    }

    getFitnessScore() {
        return this.fitnessScore;
    }

    setFitnessScore(score) {
        this.fitnessScore = score;
    }
}

class FitnessEvaluation {
    visit(member) {
        member.setFitnessScore(member.getFitnessScore() + 10);
    }
}

class MembershipDiscount {
    visit(member) {
        if (member.getSubscriptionType() === 'Premium') {
            console.log(`${member.getName()}: Fitness score - ${member.getFitnessScore()}, Membership type - ${
        } else {
            console.log(`${member.getName()}: Fitness score - ${member.getFitnessScore()}, Membership type - ${
        }
    }
}

function run() {
    const gymMembers = [
        new GymMember("Alice", "Basic", 80),
        new GymMember("Bob", "Premium", 90),
        new GymMember("Eve", "Basic", 85)
    ];

    const fitnessEvaluation = new FitnessEvaluation();
    const membershipDiscount = new MembershipDiscount();

    for (let i = 0; i < gymMembers.length; i++) {
        const member = gymMembers[i];

        member.accept(fitnessEvaluation);
        member.accept(membershipDiscount);
    }
}

run();
```

Chapter 24

File system

Filesystem operations in JavaScript are used to interact with the file system of the host environment, which can be a web browser (client-side JavaScript) or a server (Node.js). JavaScript provides various APIs and methods for reading from and writing to the file system. These operations are essential for tasks like saving data, reading configuration files, and processing user-uploaded files. Below is an overview of filesystem operations in JavaScript:

Asynchronous and Non-Blocking I/O:

In Node.js, I/O operations can be performed asynchronously, meaning that they don't block the execution of the program. Instead, they are placed in a queue, and the program continues executing other tasks. When the I/O operation is completed, a callback function is called to handle the result. This is particularly useful for I/O operations that may take a significant amount of time.

Read:

In this example, you are using the fs.readFile function to read data from the 'test.txt' file asynchronously. It takes a callback function that gets executed when the read operation is finished. The console.log("This gets printed at First") line is executed immediately after initiating the read operation, and it doesn't wait for the reading to complete.

```
const fs= require('fs');
//async
//non  blokcing i/0 thats why runs at last as ti takes moer time
fs.readFile('test.txt','utf8',(err,data) => {
    console.log(err,data)
})
console.log("This gets printed at First")
```

Write:

Here, you use fs.writeFile to write data to the 'test.txt' file asynchronously. The callback function is executed when the write operation is finished. It prints "This runs after writing in a file: written to file" after the write operation is complete.

```
fs.writeFile("test.txt","mahima is good girl", () => {
    console.log("This runs after writting in a file: written to file")
})
```

Synchronous I/O:

Synchronous I/O operations block the execution of the program until the operation is finished. In Node.js, synchronous operations should be used sparingly, especially for file I/O, as they can lead to performance issues and block the event loop.

Read:

The fs.readFileSync function is used for synchronous file reading. It blocks the execution until the entire file is read, and then it continues with the rest of the code. This is generally not recommended because it can cause the program to become unresponsive during the file read.

```
const a=fs.readFileSync("test.txt") //nodejs intetntionally blocks
console.log(a.toString())
console.log("At last")
```

Write:

fs.writeFileSync is used for synchronous file writing. It blocks the program's execution until the write operation is complete. Again, this should be used cautiously, as it can block the program for an extended period during the write operation.

```
fs.writeFileSync("test.txt","mahima is good girl",() => {
    console.log("This is sync: intentionally process is blocked ")
})
```

In summary, Node.js provides both synchronous and asynchronous file I/O options. Asynchronous I/O is typically preferred for better performance and responsiveness, while synchronous I/O is used only when necessary and with caution, as it can block the program's execution.

Chapter 25

File system

Overview of ES6 (ECMAScript 2015)

ES6, also known as ECMAScript 2015, is a significant upgrade to the JavaScript language. It introduces a wealth of new features, syntax improvements, and enhancements, making it a foundational milestone in the evolution of JavaScript. ES6 brought about a more mature and powerful language, addressing long-standing challenges and empowering developers to write cleaner, more efficient, and maintainable code.

One of the core goals of ES6 was to provide developers with tools to simplify common programming tasks, reduce code complexity, and enhance overall code quality. It achieved this by introducing numerous features and concepts that have become standard in modern JavaScript development.

ES6 includes notable features such as arrow functions, the `let` and `const` keywords for variable declaration, template literals, and the powerful `class` syntax for object-oriented programming. These enhancements significantly improve developer productivity and the readability of code.

Additionally, ES6 introduces concepts like Promises and async/await, which streamline asynchronous programming, making it more elegant and less error-prone. This simplification of asynchronous code has had a profound impact on web development and has led to a more intuitive way of handling asynchronous operations, such as HTTP requests.

Arrow functions in ES6 offer a concise syntax for writing functions, and they automatically bind the `this` context to the surrounding code, which alleviates issues with `this` binding in JavaScript. The introduction of block-scoped variable declarations with `let` and `const` improved variable management and scoping, reducing common sources of bugs.

Template literals provide a convenient way to create dynamic strings with embedded expressions, while destructuring simplifies the extraction of values from arrays and objects, leading to cleaner, more readable code.

Overall, ES6 has become a cornerstone for modern JavaScript development. It is well-supported across modern web browsers and has reshaped the way developers write JavaScript code. Understanding ES6 is crucial for any JavaScript developer looking to stay up-to-date and write more efficient, maintainable, and readable code in today's web development landscape.

In this chapter, we are going to talk about following ES6 syntax.

- let const
- map
- arrow functions
- destructuring
- template literals

let and const in ES6

1. `let` **Declaration:**
   - In ES6, the `let` declaration is introduced to create block-scoped variables. This means that a variable declared with `let` is only accessible within the block (e.g., inside a function or a pair of curly braces) where it's defined.
   - `let` variables are not hoisted to the top of their containing function or block. This prevents issues where variables are accessed before they're declared.
   - `let` allows you to reassign a value to the variable after its initial declaration, making it a mutable variable.
   - Example:

     ```
     function exampleFunction() {
       if (true) {
         let x = 10;
         console.log(x); // 10
       }
       console.log(x); // Error: x is not defined
     }
     ```

2. `const` **Declaration:**
   - The `const` declaration also introduces block-scoped variables, but it comes with an additional constraint: variables declared with `const` cannot be reassigned after their initial assignment. They are constant.
   - `const` is often used for values that should not change, such as constants or references to immutable objects.
   - Example:

     ```
     const pi = 3.14159;
     pi = 3.14; // Error: Assignment to constant variable.
     ```

Comparison with ES5:

In ES5, JavaScript primarily used the `var` keyword for variable declaration. Here are the key differences when comparing `let` and `const` in ES6 with `var` in ES5:

1. **Block Scope vs. Function Scope:**
   - ES6 (`let` and `const`): Variables declared with `let` and `const` are block-scoped, meaning they are limited to the block where they are defined, be it a block within a function or a standalone block.
   - ES5 (`var`): Variables declared with `var` are function-scoped, meaning they are accessible throughout the entire function containing the `var` declaration.
2. **Hoisting:**
   - ES6 (`let` and `const`): Variables declared with `let` and `const` are not hoisted to the top of their containing block, which prevents the use of variables before they are declared.
   - ES5 (`var`): Variables declared with `var` are hoisted to the top of their containing function, which can lead to unexpected behavior if not managed carefully.
3. **Reassignment:**

   - ES6 (`let` and `const`): Variables declared with `let` can be reassigned after their initial declaration. Variables declared with `const` cannot be reassigned, making them constants.
   - ES5 (`var`): Variables declared with `var` can be reassigned at any point in the function where they are declared.
4. **Constants:**
   - ES6 (`const`): ES6 introduces the `const` keyword, allowing the creation of constants, which cannot be changed after their initial assignment.

In summary, `let` and `const` in ES6 provide more predictable and controlled variable scoping compared to `var` in ES5. They help avoid common issues associated with variable hoisting and provide the flexibility to choose between mutable and immutable variables based on your needs.

Map

Map is a collection of keyed data items, just like an `Object`. But the main difference is that `Map` allows keys of any type.

Methods and properties are:

`new Map()` – creates the map.
`map.set(key, value)` – stores the value by the key.
`map.get(key)` – returns the value by the key, undefined if key doesn't exist in map. `map.has(key)` – returns true if the key exists, false otherwise.
`map.delete(key)` – removes the element (the key/value pair) by the key.
`map.clear()` – removes everything from the map.
`map.size` – returns the current element count.

For example

```
let map = new Map();

map.set('1', 'str1');   // a string key
map.set(1, 'num1');     // a numeric key
map.set(true, 'bool1'); // a boolean key

// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3
```

The differences from a regular `Object`:

- Any keys, objects can be keys.
- Additional convenient methods, the size property.

Conclusion

Maps are a versatile and powerful data structure that provides key-value pairs for efficient data management. Maps are often a preferred choice over plain objects for tasks involving key-value associations, as they provide better control and performance.

Arrow Functions in ES6

Arrow functions are a concise way to write anonymous functions in JavaScript, introduced in ES6 (ECMAScript 2015). They provide a more compact and readable syntax for defining functions, especially when you have simple, single-expression functions. Arrow functions are a fundamental feature of modern JavaScript, and they offer several advantages over traditional function expressions.

Syntax:

The syntax for arrow functions is straightforward:

```
const functionName = (parameters) => expression;
```

- `const functionName` : This is where you assign the function to a variable. You can omit the function name for anonymous functions.
- `(parameters)` : These are the input parameters (arguments) the function accepts. If there's only one parameter, you can omit the parentheses.
- `=>` : The fat arrow `=>` denotes that you are defining an arrow function.
- `expression` : This is the value that the function returns. If the function consists of a single statement, you can omit the curly braces and the `return` keyword.

Examples:

Here are some examples to illustrate the syntax of arrow functions:

1. A simple arrow function without parameters:

```
const sayHello = () => "Hello, World!";
```

1. An arrow function with one parameter:

```
const double = (x) => x * 2;
```

1. An arrow function with multiple parameters:

```
const add = (a, b) => a + b;
```

Use Cases:

Arrow functions are commonly used in the following scenarios:

1. **Short, Anonymous Functions:** Arrow functions are perfect for short, one-line functions. They reduce the need for writing a full function expression.
2. **Iterating Arrays:** Arrow functions work well with array methods like `map` , `filter` , and `reduce` to simplify iteration over arrays.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((x) => x * 2);
```

1. **Callback Functions:** They are often used as callback functions for asynchronous operations like `setTimeout` and `fetch`.

```
setTimeout(() => {
  console.log("Timer finished");
}, 1000);
```

1. **Binding `this` Context:** Arrow functions inherit the `this` context from their containing function, making them useful for defining methods in objects without worrying about changing `this`.

```
const person = {
  name: "John",
  greet: function() {
    setTimeout(() => {
      console.log(`Hello, my name is ${this.name}`);
    }, 1000);
  },
};

person.greet();
```

It's important to note that arrow functions are not suitable for every situation. They lack their own `this` context, cannot be used as constructors, and may not be appropriate for functions with a more complex, multi-line structure. For such cases, traditional function expressions are still the preferred choice. Arrow functions are most effective when used for simple, concise, and one-line functions.

Destructuring in ES6: Unpacking Arrays and Objects

Destructuring is a powerful feature introduced in ES6 (ECMAScript 2015) that simplifies the process of extracting values from arrays and properties from objects. It allows you to "unpack" values into variables with concise and readable syntax.

Destructuring Arrays:

Syntax:

```
const [variable1, variable2, ...rest] = array;
```

- `variable1` , `variable2` : These are variables where elements from the array are assigned.
- `...rest` (rest operator): This gathers the remaining elements into a new array variable.

Example:

```
const colors = ["red", "green", "blue"];
const [firstColor, secondColor] = colors;
console.log(firstColor); // Output: "red"
console.log(secondColor); // Output: "green"
```

Destructuring Objects:

Syntax:

```
const { property1, property2, ...rest } = object;
```

- `property1` , `property2` : These are variables where object properties are assigned.
- `...rest` (rest operator): This gathers the remaining properties into a new object.

Example:

```
const person = { name: "Alice", age: 30, city: "New York" };
const { name, age } = person;
console.log(name); // Output: "Alice"
console.log(age); // Output: 30
```

Use Cases:

Destructuring is commonly used for various tasks, including:

1. **Simplifying Assignment:** Quickly assign array elements or object properties to variables.
2. **Swapping Variables:** Easily swap the values of two variables without a temporary variable.
3. **Function Parameters:** Extract specific properties from an object passed as a function parameter.
4. **Rest Parameters:** Gather remaining elements or properties into an array or object.

By employing destructuring, you can make your code cleaner, more expressive, and less prone to errors when working with arrays and objects in JavaScript.

Template Literals in ES6: Creating Dynamic Strings

Template literals, introduced in ES6 (ECMAScript 2015), provide a powerful way to create dynamic strings in JavaScript. These literals are enclosed in backticks (` `) instead of single or double quotes and allow for seamless embedding of expressions directly within the string.

Syntax:

```
const dynamicString = `This is a dynamic string with ${expression}`;
```

- `dynamicString` : This is where you store the dynamic string.
- `${expression}` : This is where you embed JavaScript expressions, variables, or functions, which are evaluated and included within the string.

Example:

Here's a simple example of using template literals to create dynamic strings:

```
const name = "John";
const greeting = `Hello, ${name}!`;
console.log(greeting); // Output: Hello, John!
```

Use Cases:

Template literals are commonly used for various purposes, including:

1. **String Interpolation:** Inserting variables or expressions within strings.
2. **Multi-line Strings:** Creating multi-line strings without needing line breaks and concatenation.
3. **Dynamic HTML:** Generating HTML content dynamically for web applications.
4. **Tagged Templates:** Allowing for custom processing of template literals through template tag functions.

By using template literals, you can simplify string concatenation, enhance the readability of your code, and make dynamic string creation a breeze in JavaScript.

References

Ballard, P. (2018). JavaScript in 24 Hours, Sams Teach Yourself. Sams Publishing.

Crockford, D. (2008). JavaScript: The Good Parts. O'Reilly Media.

Duckett, J. (2011). HTML & CSS: Design and Build Websites. Wiley.

Duckett, J. (2014). JavaScript and JQuery: Interactive Front-End Web Development. Wiley.

Flanagan, D. (2011). JavaScript: The Definitive Guide. O'Reilly Media.

Freeman, E., & Robson, E. (2014). Head First JavaScript Programming: A Brain-Friendly Guide. O'Reilly Media.

Frisbie, M. (2019). Professional JavaScript for Web Developers. Wrox.

Haverbeke, M. (2019). Eloquent JavaScript: A Modern Introduction to Programming. No Starch Press.

Herman, D. (2012). Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript. Addison-Wesley Professional.

McPeak, J., & Wilton, P. (2015). Beginning JavaScript. Wiley.

Morgan, N. (2014). JavaScript for Kids: A Playful Introduction to Programming. No Starch Press.

Murphy C, Clark R, Studholme O, et al. (2014). Beginning HTML5 and CSS3: The Web Evolved. Apress.

Nixon, R. (2021). Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5. O'Reilly Media.

Powell, T., & Schneider, F. (2012). JavaScript: The Complete Reference. McGraw-Hill Education.

Resig, J. (2007). Pro JavaScript Techniques. Apress.

Resig, J., & Bibeault, B. (2016). Secrets of the JavaScript Ninja. Manning Publications.

Robbins, J. N. (2014). Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics. O'Reilly Media.

Learning Resources

Articles for learning JavaScript

1. Introduction to JavaScript by Hostinger Link
2. JavaScript tutorial by Geeks for Geeks Link
3. JavaScript basics by Mozilla Developer Link
4. Introduction to JavaScript by Microverse Link
5. JavaScript Tutorial by Stackify Link

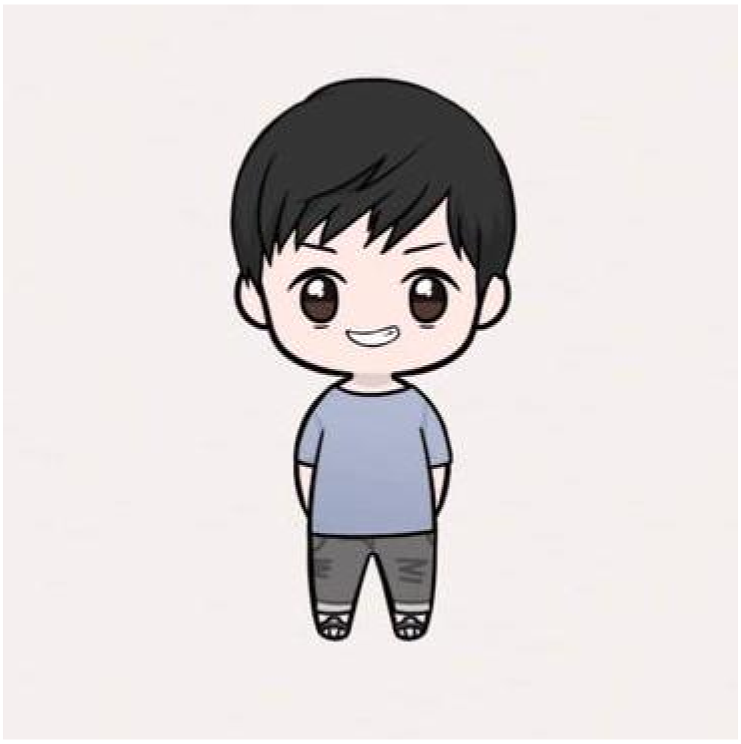
Books for learning JavaScript

1. You Don't Know JS – Kyle Simpson Link
2. A Smarter Way to Learn JavaScript – Mark Myers Link
3. Eloquent JavaScript – Marijn Haverbeke Link
4. JavaScript: The Good Parts – Douglas Crockford Link
5. Professional JavaScript for Web Developers Link
6. Head First JavaScript Link
7. Secrets of the JavaScript Ninja Link
8. The Principles of Object-Oriented JavaScript Link

YouTube Resources to follow for learning JavaScript

1. Traversy Media - Link
2. The Net Ninja - Link
3. freeCodeCamp - Link
4. CodeWithHarry - Link
5. Academind - Link
6. JavaScript Mastery - Link
7. SuperSimpleDev - Link
8. Dave Gray - Link
9. Clever Programmer - Link
10. Akshay Saini - Link
11. Hitesh Choudhary - Link
12. thenewboston - Link

Maintainer



Suman Kunwar

Contributors