Using Objects as Keys:

```
let objKeyMap = new Map();

let keyObject = { id: 1 };

objKeyMap.set(keyObject, "Value for the object key");
```

Maps provide a versatile way to store and manage key-value pairs, offering more flexibility than objects for certain use cases. They are particularly useful when you need to associate unique keys with specific values and require efficient methods for manipulation and retrieval.

typeof

The typeof operator is used to determine the data type of a variable or an expression. It returns a string representing the type of the operand. Here are some examples of how the typeof operator works:

typeof for Primitive Data Types:

```
let numberVar = 42;
console.log(typeof numberVar); // Output: "number"

let stringVar = "Hello, World!";
console.log(typeof stringVar); // Output: "string"

let booleanVar = true;
console.log(typeof booleanVar); // Output: "boolean"
```

typeof for Undefined and Null:

```
let undefinedVar;
console.log(typeof undefinedVar); // Output: "undefined"

let nullVar = null;
console.log(typeof nullVar); // Output: "object"
```

Note: The typeof null returning "object" is considered a historical mistake in JavaScript.

typeof for Objects:

```
let objectVar = { key: "value" };
console.log(typeof objectVar); // Output: "object"

let arrayVar = [1, 2, 3];
console.log(typeof arrayVar); // Output: "object"
```

```
let functionVar = function() {};
console.log(typeof functionVar); // Output: "function"
```

typeof for Symbols:

```
let symbolVar = Symbol("uniqueSymbol");
console.log(typeof symbolVar); // Output: "symbol"
```

typeof for Functions:

```
function exampleFunction() {}
console.log(typeof exampleFunction); // Output: "function"
```

typeof for BigInt:

```
let bigIntVar = BigInt(10);
console.log(typeof bigIntVar); // Output: "bigint"
```

typeof for NaN:

```
let nanVar = NaN;
console.log(typeof nanVar); // Output: "number"
```

Note: NaN is considered a numeric value, and typeof NaN returns "number."

typeof for Strings created with String Constructor:

```
let stringObjectVar = new String("Hello");
console.log(typeof stringObjectVar); // Output: "object"
```

Note: Strings created with the String constructor are considered objects, not primitives.

The typeof operator is a handy tool for checking the type of a variable dynamically, which can be useful for conditional logic or debugging purposes. It's important to note that typeof has some quirks, especially when it comes to null and objects, so it's essential to be aware of its behavior in different scenarios.

Type Conversion

JavaScript supports automatic and explicit type conversion. Type conversion is the process of converting a value from one data type to another.

Here are some examples of type conversion in JavaScript:

Implicit Type Conversion (Coercion): JavaScript automatically converts values from one type to another in certain situations.

```
let numberVar = 42;
let stringVar = "The answer is " + numberVar; // Number to
String
console.log(stringVar); // Output: "The answer is 42"
```

Explicit Type Conversion: You can explicitly convert values from one type to another using conversion functions or operators.

String to Number:

```
let numericString = "123";
let numberFromStr = Number(numericString);
console.log(numberFromStr); // Output: 123
```

Number to String:

```
let numberVar = 42;
let stringFromNumber = String(numberVar);
console.log(stringFromNumber); // Output: "42"
```

Boolean to Number:

```
let boolVar = true;
let numberFromBool = Number(boolVar);
console.log(numberFromBool); // Output: 1
```

```
Number to Boolean:

let zero = 0;
let boolFromNumber = Boolean(zero);
console.log(boolFromNumber); // Output: false

Implicit Type Conversion in Operations: JavaScript performs
implicit type conversion in operations involving different
data types.

let result = "5" * 2; // String to Number (multiplication)
console.log(result); // Output: 10
```

NaN (Not a Number): Some operations result in NaN, which stands for "Not a Number."

```
let notANumber = "Hello" / 2; // Division of a non-numeric
string
console.log(notANumber); // Output: NaN
```

parseInt and parseFloat Functions: These functions convert strings to integers and floating-point numbers, respectively.

```
let integerFromString = parseInt("42");
let floatFromString = parseFloat("3.14");
```

Concatenating Strings and Numbers: When you use the + operator with a string and a number, JavaScript converts the number to a string and concatenates them.

```
let concatString = "The answer is " + 42;
console.log(concatString); // Output: "The answer is 42"
```

Array to String: Using the join method to convert an array to a string.

```
let array = [1, 2, 3];
let arrayAsString = array.join(", ");
console.log(arrayAsString); // Output: "1, 2, 3"
```

String to Date: Using the Date constructor to convert a string to a date object.

```
let dateString = "2023-01-01";
let dateObject = new Date(dateString);
```

Type conversion is a crucial aspect of JavaScript, and understanding how it occurs allows you to write more robust and predictable code. It's important to be aware of potential pitfalls, especially when working with mixed data types or relying on implicit conversions.

Bitwise Operations

JavaScript supports bitwise operations, which operate on the individual bits of integers. Bitwise operations can be useful in scenarios where you need to manipulate or extract specific bits within integer values. Here are some examples of bitwise operations in JavaScript:

Bitwise AND (&): Performs a bitwise AND operation on each pair of corresponding bits.

```
let result = 5 & 3;
console.log(result); // Output: 1 (binary: 101 & 011 = 001)
```

Bitwise OR (|): Performs a bitwise OR operation on each pair of corresponding bits.

```
let result = 5 | 3;
console.log(result); // Output: 7 (binary: 101 | 011 = 111)
```

Bitwise XOR (^): Performs a bitwise XOR (exclusive OR) operation on each pair of corresponding bits.

```
let result = 5 ^ 3;
console.log(result); // Output: 6 (binary: 101 ^ 011 = 110)
```

Bitwise NOT (~): Performs a bitwise NOT operation, which inverts the bits of its operand.

```
let result = ~5;
console.log(result); // Output: -6 (binary: ~0101 = 1010,
then interpreted as a signed two's complement)
```

Left Shift (<<): Shifts the bits of the left operand to the left by a specified number of positions.

```
let result = 5 << 2;
```

```
console.log(result); // Output: 20 (binary: 00000101 << 2 =
00010100)
```

Right Shift (>>): Shifts the bits of the left operand to the right by a specified number of positions.

```
let result = 16 >> 2;
console.log(result); // Output: 4 (binary: 00010000 >> 2 =
00000004)
```

Zero-fill Right Shift (>>>): Similar to the right shift (>>), but fills the vacant bits with zeros.

```
let result = -16 >>> 2;
console.log(result);
// Output: 1073741820 (binary:
11111111111111111111111111110000 >>> 2)
```

Bitwise operations are often used in low-level programming and can be helpful in scenarios where you need to perform operations at the bit level, such as optimizing algorithms or working with binary data. However, they may not be as commonly used in higher-level JavaScript programming compared to other languages.

Regular Expressions

Regular Expressions (RegEx) in JavaScript provide a powerful and flexible way to search, match, and manipulate text. They are used for pattern matching within strings.

Here are some key aspects and examples of using regular expressions in JavaScript:

Creating a Regular Expression: You can create a regular expression using the RegExp constructor or a literal notation.

```
// Using RegExp constructor
let regex1 = new RegExp("pattern");

// Using literal notation
let regex2 = /pattern/;
```

Basic Matching:

```
let text = "The cat and the hat";
let pattern = /at/g;

// Using test() method to check if the pattern exists
console.log(pattern.test(text)); // Output: true

// Using match() method to get an array of matches
console.log(text.match(pattern)); // Output: [ 'at', 'at' ]
```

Anchors and Boundaries:

```
let text = "This is a cat";
let pattern = /^cat/;

// ^ asserts the start of the string
```

```
console.log(pattern.test(text)); // Output: false

let pattern2 = /\bis\b/;

// \b asserts a word boundary
console.log(pattern2.test(text)); // Output: true
```

Character Classes:

```
let text = "The quick brown fox jumps over the lazy dog";
let pattern = /[aeiou]/g;

// Matches any vowel
console.log(text.match(pattern)); // Output: [ 'e', 'u',
'i', 'o', 'o', 'o', 'u', 'e', 'o', 'e', 'e', 'a', 'o' ]
```

Quantifiers:

```
let text = "aaaaabbbbcccc";
let pattern = /a{3,5}/g;

// Matches 'aaa' or 'aaaaa'
console.log(text.match(pattern)); // Output: [ 'aaaaa' ]
```

Wildcard (Dot):

```
let text = "cat, hat, mat, rat";
let pattern = /.at/g;

// Matches 'cat', 'hat', 'mat', 'rat'
console.log(text.match(pattern));
// Output: [ 'cat', 'hat', 'mat', 'rat' ]
```

Character Sets and Ranges:

```
let text = "123-456-7890";
let pattern = /\d{3}-\d{3}-\d{4}/;

// Matches a phone number pattern
console.log(pattern.test(text)); // Output: true
```

Modifiers:

```
let text = "Case-Insensitive";
let pattern = /insensitive/i;

// i modifier makes the pattern case-insensitive
console.log(pattern.test(text)); // Output: true
```

Capturing Groups:

```
let text = "John Doe, Jane Smith";
let pattern = /(\w+) (\w+)/g;

// Matches full names and captures first and last names
let matches = text.match(pattern);
console.log(matches);
// Output: [ 'John Doe', 'Jane Smith' ]
console.log(matches[1]); // Output: 'John'
console.log(matches[2]); // Output: 'Doe'
```

Replacing Text:

```
let text = "Replace blue with red.";
let pattern = /blue/;
let replacement = "red";

// Replaces 'blue' with 'red'
let newText = text.replace(pattern, replacement);
console.log(newText); // Output: 'Replace red with red.'
```

Regular expressions offer a wide range of features and options. Understanding and mastering regular expressions can significantly enhance your ability to work with text data in a flexible and efficient manner.

Operador precedence

Operator precedence in JavaScript determines the order in which operators are evaluated within an expression. Operators with higher precedence are evaluated first. If operators have the same precedence, the order of evaluation depends on their associativity (left-to-right or right-to-left).

Highest Precedence:
(): Parentheses (grouping)

Unary Operators:
++ --: Postfix increment/decrement
+ - ! ~ typeof void delete: Unary plus, unary minus, logical NOT, bitwise NOT, typeof, void, delete
++ -- + -: Prefix increment/decrement, unary plus, unary minus
new: Creating new objects
. []: Member access, array element access

Multiplicative Operators:
*** / %:** Multiplication, division, modulo

Additive Operators:
+ -: Addition, subtraction

Bitwise Shift Operators:
<< >> >>>: Left shift, right shift, zero-fill right shift

Relational Operators:
< > <= >= instanceof in: Less than, greater than, less than or equal, greater than or equal, instanceof, in

Equality Operators:
== != === !==: Equality, inequality, strict equality, strict inequality

Bitwise AND Operator:

&: Bitwise AND

Bitwise XOR Operator:
^: Bitwise XOR

Bitwise OR Operator:
|: Bitwise OR

Logical AND Operator:
&&: Logical AND

Logical OR Operator:
||: Logical OR

Conditional (Ternary) Operator:
? :: Conditional (ternary) operator

Assignment Operators:
= += -= *= /= %= <<= >>= >>>= &= ^= |=: Assignment, addition assignment, subtraction assignment, multiplication assignment, division assignment, modulo assignment, left shift assignment, right shift assignment, zero-fill right shift assignment, bitwise AND assignment, bitwise XOR assignment, bitwise OR assignment

Lowest Precedence:
,: Comma (separates expressions)

Understanding operator precedence is crucial for writing code that behaves as expected. When in doubt, you can use parentheses to explicitly control the order of evaluation in complex expressions.

Errors

Errors can occur during the execution of a program. When an error occurs, it disrupts the normal flow of the program and may prevent it from running correctly. JavaScript has several types of errors, each represented by a specific object. Here are some common JavaScript error types:

SyntaxError: This error occurs when the JavaScript engine encounters a syntax mistake in the code.

```
let x = 10
console.log(x;
// SyntaxError: missing ) after argument list
```

ReferenceError: Reference errors occur when trying to reference a variable or function that has not been declared.

```
console.log(y);
// ReferenceError: y is not defined

function myFunction() {
  console.log(z);
}
myFunction();
// ReferenceError: z is not defined
```

TypeError: Type errors occur when an operation is performed on an inappropriate type.

```
let x = "Hello";
x.toUpperCase();
// TypeError: x.toUpperCase is not a function
```

RangeError: Range errors occur when trying to manipulate an object with a length outside the valid range.

```
let arr = new Array(-1);
// RangeError: Invalid array length
```

EvalError: Eval errors are deprecated and no longer used in modern JavaScript.

URIError: URI errors occur when using methods like decodeURIComponent or encodeURIComponent with invalid input.

```
decodeURIComponent('%');
// URIError: URI malformed
```

Custom Errors: Developers can create custom error objects by extending the built-in Error object.

```
class MyCustomError extends Error {
    constructor(message) {
      super(message);
      this.name = 'MyCustomError';
    }
  }

  throw new MyCustomError('This is a custom error.');
```

Handling Errors: To handle errors in JavaScript, you can use try...catch blocks:

```
try {
    // Code that might throw an error
  } catch (error) {
    // Code to handle the error
    console.error(error.message);
  } finally {
    // Code that runs regardless of whether an error
occurred
  }
```

Handling errors is essential for creating robust and resilient applications. It allows you to gracefully handle unexpected situations and prevent them from crashing the entire application.

Scope

JavaScript scope refers to the context in which variables and functions are declared and can be accessed. The scope determines the visibility and lifetime of variables and functions. There are two main types of scope in JavaScript: global scope and local scope.

Global Scope: Variables and functions declared outside of any function or block have global scope. They can be accessed from any part of the code, including inside functions.

```
let globalVar = "I am global"; // Global variable

function myFunction() {
  console.log(globalVar); // Accessing global variable
inside a function
}

myFunction(); // Output: "I am global"
```

Local Scope: Variables and functions declared inside a function or block have local scope. They are only accessible within that specific function or block.

Function Scope: Variables declared using var inside a function have function scope.

```
function myFunction() {
    var localVar = "I am local"; // Local variable
    console.log(localVar);
  }

  myFunction(); // Output: "I am local"
  console.log(localVar); // ReferenceError: localVar is not
defined (outside the function)
```

Block Scope (ES6 and later): Variables declared using let and const inside a block (e.g., if statement or loop) have block scope.

```
if (true) {
    let blockVar = "I am inside a block"; // Block-scoped
variable
    console.log(blockVar);
  }

  console.log(blockVar); // ReferenceError: blockVar is not
defined (outside the block)
```

Lexical Scope: JavaScript uses lexical scoping, which means that the scope of a variable is determined by its position in the code. Inner functions have access to variables declared in their containing (outer) functions.

```
function outerFunction() {
    let outerVar = "I am outer";

    function innerFunction() {
      console.log(outerVar); // Accessing outer variable
from inner function
    }

    innerFunction();
  }

  outerFunction(); // Output: "I am outer"
```

Scope Chain: When a variable is referenced, JavaScript looks for the variable in the current scope. If the variable is not found, it looks in the outer scope, continuing up the scope chain until the variable is found or the global scope is reached.

```
let globalVar = "I am global";
```

```
function outerFunction() {
  let outerVar = "I am outer";

  function innerFunction() {
    console.log(outerVar); // Accessing outerVar from inner
function
    console.log(globalVar); // Accessing globalVar from
inner function
  }

  innerFunction();
}

outerFunction();
```

Understanding JavaScript scope is crucial for writing maintainable and bug-free code. It helps prevent naming conflicts, ensures proper variable access, and contributes to code organization and readability.