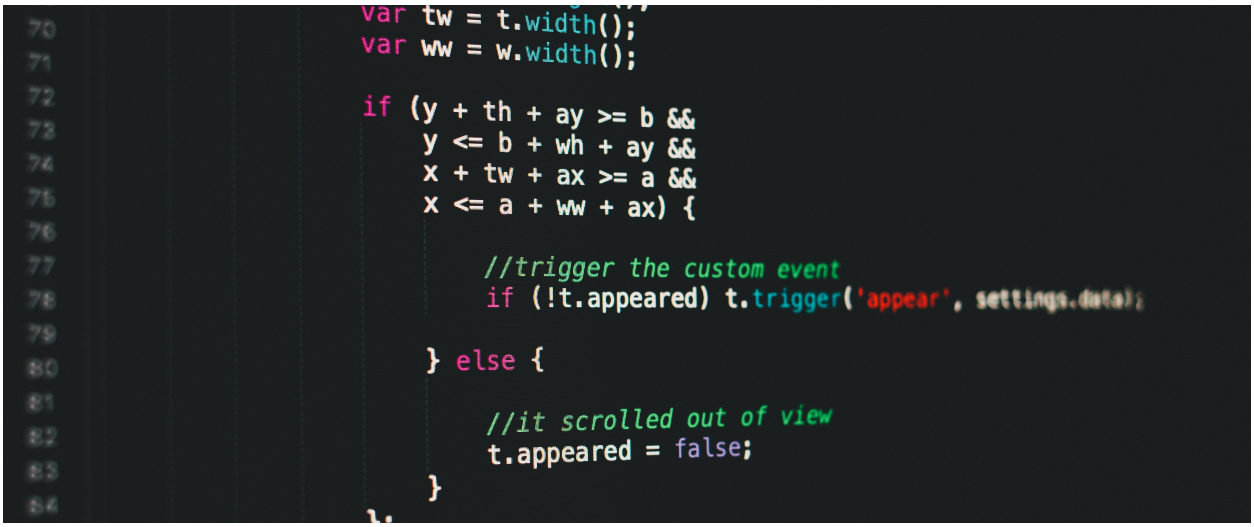

JavaScript

Variable naming conventions

- **Camel Case**: In camel case, each word in the variable name, except the first one, starts with a capital letter. For example: `myVariableName`, `firstName`, `numberOfStudents`.
- **Snake Case**: In snake case, words in the variable name are separated by underscores. For example: `my_variable_name`, `first_name`, `number_of_students`.

Data types

Primitive Data Types (Scalar Types):

1. **Integer (int)**: Represents whole numbers without a fractional component. Examples include `1`, `42`, and `10`. In some programming languages, integers may have different sizes (e.g., `int`, `long`, `short`) depending on the platform.
2. **Floating-Point (float or double)**: Represents numbers with a fractional part. Examples include `3.14` and `0.001`. `float` and `double` are typically used to represent real numbers with different levels of precision.
3. **Character (char)**: Represents a single character, such as `'A'`, `'5'`, or `'%'`. In some languages, characters are enclosed in single quotes.

4. **Boolean (bool)**: Represents a binary value that is either `true` or `false`. Booleans are used for logical operations and conditional statements.
5. **Void**: This data type is used to indicate that a function does not return any value. It's typically used in functions that perform actions without producing a result.

Non-Primitive Data Types (Composite Types or Reference Types):

1. **Array**: A collection of elements, often of the same data type, arranged in a sequential order. Arrays are mutable, and their size may be fixed or dynamic, depending on the programming language.
2. **String**: A sequence of characters that represents text. Strings are typically implemented as arrays of characters and are often treated as non-primitive due to their complexity and the availability of string manipulation methods.
3. **Object**: A complex data type that can hold key-value pairs, methods, and properties. Objects are used for modeling real-world entities and are a fundamental building block in many programming languages, especially in object-oriented programming.
4. **Function**: Functions are first-class citizens in many programming languages, meaning they can be assigned to variables, passed as arguments, and returned from other functions. Functions are used for encapsulating reusable blocks of code.
5. **Class**: In object-oriented programming languages, classes define blueprints for creating objects. A class encapsulates both data (attributes) and methods (functions) related to a particular type of object.
6. **Interface/Protocol (depending on the language)**: Defines a set of methods or properties that a class must implement. Interfaces help enforce a contract between different parts of a program.
7. **Enumeration (enum)**: A data type that consists of a set of named values. Enums are often used to define a finite set of constant values, such as days of the week or card suits.
8. **List, Set, Map (variations exist)**: Collections or data structures that group multiple values. Lists are ordered, sets contain unique elements, and maps associate keys with values.

9. **Pointer or Reference**: In low-level languages like C and C++, pointers and references allow indirect access to memory locations. They are used for advanced memory manipulation but come with potential pitfalls like null pointer errors.

JavaScript Strings

1. **Regular Strings**: These are the most common type of strings in JavaScript, created by enclosing text in either single quotes (`'`) or double quotes (`"`). For example:

   ```
   let singleQuotedString = 'Hello, World!';
   let doubleQuotedString = "JavaScript is versatile.";
   ```

   Both single-quoted and double-quoted strings behave the same way in JavaScript.

2. **Template Literals (Template Strings)**: Introduced in ECMAScript 6 (ES6), template literals allow you to create strings with embedded expressions using backticks (\`). These strings can span multiple lines and include placeholders indicated by ${}`. For example:

   ```
   let name = 'Alice';
   let greeting = `Hello, ${name}!`;
   ```

   Template literals offer a convenient way to create dynamic strings.

3. **String Concatenation**: You can concatenate strings using the `+` operator. This is particularly useful for combining multiple strings into one:

   ```
   let firstName = 'John';
   let lastName = 'Doe';
   let fullName = firstName + ' ' + lastName;
   ```

   You can also use the `+=` operator for in-place concatenation.

4. **String Methods**: JavaScript provides numerous built-in string methods to manipulate and work with strings. Some common ones include `charAt()`, `concat()`, `slice()`, `substring()`, `split()`, `indexOf()`, `replace()`, and `toUpperCase()`, among others.

5. **Escape Sequences**: JavaScript supports escape sequences in strings to represent special characters. For example, `\\n` represents a newline, `\\t` represents a tab, and `\\\\` represents a literal backslash within a string.
6. **Raw Strings (ES6)**: Raw strings are created using the `String.raw()` method in ES6. They are useful when you want to create strings with escape sequences preserved exactly as they are. For example:

   ```
   let rawString = String.raw`Hello\\nWorld`;
   ```

   In this case, the `\\n` is treated as a literal backslash followed by 'n' and not as a newline character.
7. **Unicode Characters**: JavaScript strings support Unicode characters, allowing you to work with characters from various languages and symbol sets.
8. **Empty Strings**: An empty string is a string with no characters in it. You can create an empty string using `''` or `""`.
9. **Immutable**: JavaScript strings are immutable, meaning once created, they cannot be modified. Any operation that appears to modify a string actually creates a new string.

JavaScript Operators

Arithmetic Operators:

1. **Addition (+)**: Adds two operands. For example, `x + y` adds the values of `x` and `y`.
2. **Subtraction (-)**: Subtracts the right operand from the left operand. For example, `x - y` subtracts `y` from `x`.
3. **Multiplication (*)**: Multiplies two operands. For example, `x * y` multiplies the values of `x` and `y`.
4. **Division (/)**: Divides the left operand by the right operand. For example, `x / y` divides `x` by `y`.
5. **Modulus (%)**: Returns the remainder of the division of the left operand by the right operand. For example, `x % y` gives the remainder when `x` is divided by `y`.

6. **Increment (++)**: Increases the value of a variable by 1. For example, `x++` or `++x` increases the value of `x` by 1.
7. **Decrement (--)**: Decreases the value of a variable by 1. For example, `x--` or `-x` decreases the value of `x` by 1.

Comparison Operators:

1. **Equal (==)**: Compares two operands for equality, allowing for type coercion. For example, `x == y` checks if `x` is equal to `y`.
2. **Not Equal (!=)**: Compares two operands for inequality, allowing for type coercion. For example, `x != y` checks if `x` is not equal to `y`.
3. **Strict Equal (===)**: Compares two operands for strict equality, without type coercion. For example, `x === y` checks if `x` is equal to `y` and has the same data type.
4. **Strict Not Equal (!==)**: Compares two operands for strict inequality, without type coercion. For example, `x !== y` checks if `x` is not equal to `y` or has a different data type.
5. **Greater Than (>)**: Checks if the left operand is greater than the right operand. For example, `x > y` checks if `x` is greater than `y`.
6. **Less Than (<)**: Checks if the left operand is less than the right operand. For example, `x < y` checks if `x` is less than `y`.
7. **Greater Than or Equal To (>=)**: Checks if the left operand is greater than or equal to the right operand. For example, `x >= y` checks if `x` is greater than or equal to `y`.
8. **Less Than or Equal To (<=)**: Checks if the left operand is less than or equal to the right operand. For example, `x <= y` checks if `x` is less than or equal to `y`.

Logical Operators:

1. **Logical AND (&&)**: Returns true if both operands are true. For example, `x && y` is true if both `x` and `y` are true.
2. **Logical OR (||)**: Returns true if at least one operand is true. For example, `x || y` is true if either `x` or `y` or both are true.
3. **Logical NOT (!)**: Returns the opposite Boolean value of the operand. For example, `!x` is true if `x` is false, and vice versa.

Assignment Operators:

1. **Assignment (=)**: Assigns a value to a variable. For example, `x = 10` assigns the value 10 to the variable `x`.
2. **Addition Assignment (+=)**: Adds the right operand to the left operand and assigns the result to the left operand. For example, `x += 5` is equivalent to `x = x + 5`.
3. **Subtraction Assignment (-=)**: Subtracts the right operand from the left operand and assigns the result to the left operand. For example, `x -= 3` is equivalent to `x = x - 3`.
4. **Multiplication Assignment (*=)**: Multiplies the left operand by the right operand and assigns the result to the left operand. For example, `x *= 2` is equivalent to `x = x * 2`.
5. *Division Assignment (/=

)**: Divides the left operand by the right operand and assigns the result to the left operand. For example, `x /= 4` is equivalent to `x = x / 4`.

1. **Modulus Assignment (%=)**: Computes the modulus of the left operand by the right operand and assigns the result to the left operand. For example, `x %= 3` is equivalent to `x = x % 3`.

Other Operators:

1. **Ternary Conditional (Conditional Operator) (?:)**: Allows you to assign a value to a variable conditionally based on a Boolean expression. For example, `x = (condition) ? valueIfTrue : valueIfFalse`.
2. **Typeof Operator**: Returns a string representing the data type of an operand. For example, `typeof x` returns a string indicating the data type of `x`.
3. **Instanceof Operator**: Checks if an object is an instance of a particular class or constructor function. For example, `obj instanceof MyClass` checks if `obj` is an instance of `MyClass`.

JavaScript Conditional Statements

1. **if Statement:**

The `if` statement is used to execute a block of code if a specified condition is `true`. It has the following syntax:

```
if (condition) {
    // Code to execute if the condition is true
}
```

Example:

```
let age = 20;
if (age >= 18) {
    console.log("You are an adult.");
}
```

2. **if...else Statement:**

   The `if...else` statement allows you to execute one block of code if a condition is `true` and another block of code if the condition is `false`. It has the following syntax:

   ```
   if (condition) {
       // Code to execute if the condition is true
   } else {
       // Code to execute if the condition is false
   }
   ```

   Example:

   ```
   let hour = 12;
   if (hour < 12) {
       console.log("Good morning!");
   } else {
       console.log("Good afternoon!");
   }
   ```

3. **if...else if...else Statement:**

The `if...else if...else` statement allows you to test multiple conditions and execute different code blocks based on which condition is true first. It has the following syntax:

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition2 is true
} else {
    // Code to execute if none of the conditions are true
}
```

Example:

```
let grade = 75;
if (grade >= 90) {
    console.log("A");
} else if (grade >= 80) {
    console.log("B");
} else if (grade >= 70) {
    console.log("C");
} else {
    console.log("D");
}
```

4. **Switch Statement:**

   The `switch` statement is used to select one of many code blocks to be executed. It's often used when you have multiple possible values to compare against a single variable. It has the following syntax:

```
switch (expression) {
    case value1:
        // Code to execute if expression matches value1
        break;
    case value2:
```

```
        // Code to execute if expression matches value2
        break;
    // ...
    default:
        // Code to execute if expression doesn't match any
case
}
```

Example:

```
let day = "Monday";
switch (day) {
    case "Monday":
        console.log("It's the start of the week.");
        break;
    case "Friday":
        console.log("It's almost the weekend.");
        break;
    default:
        console.log("It's a regular day.");
}
```

JavaScript Arrays

Creating Arrays:

1. **Array Literal:** You can create an array using square brackets `[]` and populate it with values.

   ```
   let fruits = ["apple", "banana", "cherry"];
   ```

2. **Array Constructor:** You can use the `Array` constructor to create an array.

```
let cars = new Array("Volvo", "BMW", "Ford");
```

Accessing Array Elements:

Array elements are accessed using their index, starting from 0 for the first element.

```
console.log(fruits[0]); // Output: "apple"
```

Modifying Array Elements:

You can change the value of an array element by assigning a new value to it.

```
fruits[1] = "orange"; // Changes the second element to "orang
e"
```

Array Length:

You can find the length of an array using the `length` property.

```
console.log(fruits.length); // Output: 3
```

Adding and Removing Elements:

1. **Adding Elements:**
   - `push()`: Adds one or more elements to the end of an array.
   - `unshift()`: Adds one or more elements to the beginning of an array.
2. **Removing Elements:**
   - `pop()`: Removes the last element from an array.
   - `shift()`: Removes the first element from an array.

Iterating Through Arrays:

You can loop through the elements of an array using various methods, such as `for` loops, `forEach()`, `for...of`, or `map()`.

```
fruits.forEach(function(fruit) {
   console.log(fruit);
```

```
});
```

Array Methods:

JavaScript provides a variety of built-in array methods for performing common operations on arrays. Some common methods include `push()`, `pop()`, `shift()`, `unshift()`, `slice()`, `splice()`, `concat()`, `join()`, `indexOf()`, `includes()`, `filter()`, `map()`, and `reduce()`.

Multidimensional Arrays:

Arrays can hold other arrays, creating multidimensional arrays. For example, a 2D array can be used to represent a grid or matrix.

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

Array Destructuring:

You can extract values from arrays using array destructuring.

```
let [first, second] = fruits; // Destructuring the first and
second elements
console.log(first); // Output: "apple"
console.log(second); // Output: "orange"
```

JavaScript For Loop

```
let arr1 = [23, 4, 2, 4, 328, 324, 3, 5];
let highestNumber = arr1[0];

for (let x = 1; x < arr1.length; x++) {
    highestNumber = (highestNumber < arr1[x]) ? arr1[x] : highes
```

```
}

console.log("The highest number is:", highestNumber);
```

```
 *  *  *  *  *
 *  *  *  *
 *  *  *
 *  *
 *
```

```
let value= ''
for (let index = 0; index < 5; index++) {
    value = ''
    for (let index1 = 5 - index; index1 > 0; index1--) {
    value = value + ' * '

}
    console.log(value)
}
```

JavaScript Functions

Defining a Function:

You can define a JavaScript function using the `function` keyword followed by a function name and a pair of parentheses that can hold parameters. The function body is enclosed in curly braces `{}`. Here's the basic syntax:

```
function functionName(parameters) {
    // Function body: code to be executed
}
```

For example:

```
function greet(name) {
    console.log("Hello, " + name + "!");
}
```

Calling a Function:

To execute a function, you simply call it by its name and pass any required arguments inside the parentheses. For example:

```
greet("Alice"); // Calls the greet function with "Alice" as t
he argument
```

Function Parameters:

Functions can accept parameters (also called arguments) that act as placeholders for values that are passed to the function when it's called. These parameters are used within the function's body. For example:

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

let result = add(5, 3); // Calls the add function with 5 and
3 as arguments
console.log(result); // Output: 8
```

Return Statement:

Functions can return values using the `return` statement. This allows functions to produce results that can be used in other parts of your code. For example:

```
function multiply(x, y) {
    return x * y;
}

let product = multiply(4, 2); // Calls the multiply function
```

```
and assigns the result to the 'product' variable
console.log(product); // Output: 8
```

Anonymous Functions (Function Expressions):

You can also define functions without a name, known as anonymous functions or function expressions. These functions can be assigned to variables and passed as arguments to other functions. For example:

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

console.log(square(5)); // Output: 25
```

Arrow Functions (ES6):

Arrow functions provide a concise syntax for defining functions, especially when they have a single statement. They are introduced in ES6 (ECMAScript 2015). For example:

```
let cube = (x) => x * x * x;

console.log(cube(3)); // Output: 27
```

Function Scopes:

JavaScript functions have their own scope. Variables declared inside a function are local to that function and cannot be accessed from outside. Variables declared outside of any function are considered global and can be accessed from anywhere in the code.

Function Hoisting:

JavaScript functions are hoisted, which means they can be called before they are defined in the code. However, it's a good practice to define functions before using them for better code readability.

Function Parameters vs. Arguments:

Function parameters are the placeholders listed in the function definition, while arguments are the actual values passed to the function when it's called.

Function Overloading:

JavaScript does not support function overloading based on the number or types of parameters, as some other languages do. However, you can achieve similar behavior by using conditional logic inside the function.

```
let firstName = 'Hanna'

function greet(name){
return 'hello' + name
}
console.info(greet(' hanna'))
```

Anonymous and Arrow Functions

Anonymous Functions:

An anonymous function, as the name suggests, is a function without a name. Instead of defining a named function, you declare it inline where you need it. Anonymous functions are also known as function expressions.

Here's an example of an anonymous function:

```
let greet = function(name) {
    console.log("Hello, " + name + "!");
};

greet("Alice"); // Output: Hello, Alice!
```

In this example, we've defined an anonymous function and assigned it to the variable `greet`. We can then call this function using the variable `greet`.

Anonymous functions are often used for short, one-off functions or as arguments to other functions, such as callback functions in asynchronous operations.

Arrow Functions (ES6):

Arrow functions are a more concise way to define functions, introduced in ECMAScript 2015 (ES6). They have a more streamlined syntax and, in some cases, a different behavior compared to traditional anonymous functions.

Here's the equivalent of the previous example using an arrow function:

```
let greet = (name) => {
    console.log("Hello, " + name + "!");
};

greet("Bob"); // Output: Hello, Bob!
```

Arrow functions have the following characteristics:

- They use the `=>` syntax.
- When there's only one parameter, you can omit the parentheses around the parameter. For example: `(name) => {...}` can be written as `name => {...}`.
- If the function body consists of a single expression, you can omit the braces `{}` and the `return` keyword. The expression's value is automatically returned.

Here's an example of an arrow function with a single expression:

```
let square = x => x * x;

console.log(square(5)); // Output: 25
```

Arrow functions are often used for short, concise functions and can be especially helpful when writing functional programming-style code or working with array methods like `map`, `filter`, and `reduce`.

It's worth noting that arrow functions have lexical scoping for the `this` keyword, which means they capture the value of `this` from their surrounding context. Traditional anonymous functions have their own `this` binding, which can lead to different behavior in certain cases. This is an important distinction when working with object methods and event handlers.