INTRODUCTION TO JAVASCRIPT SYNTAX

Let's start exploring the basic syntax of JavaScript.

Expressions and Statements

Expressions

In JavaScript, an **expression** is any valid unit of code that **resolves to a value**. It can be a literal, a variable name, an operator, a function call, or any combination of these elements that evaluates to a single value. Here are a few examples of expressions:

```
5;              // a numeric literal
'hello';        // a string literal
x;              // a variable
sum(x, y);      // a function call
5 + 6;          // an arithmetic operation
x < y;          // a comparison operation
```

Each of these examples is an expression that evaluates to a single value: `5`, `'hello'`, the value of `x`, the return value of `sum(x, y)`, `11`, and a boolean value (`true` or `false`), respectively.

Statements

A statement, on the other hand, is a piece of code that **performs an action**. Statements in JavaScript can include declarations, assignments, function calls, control flow commands (like loops and conditional statements), and more. While statements might contain expressions, not all expressions qualify as full statements on their own. Here are some examples of statements:

```
let x = 5;                          // variable declaration
and assignment
y = 6;                              // variable assignment
const z = sum(5, 6);                // variable declaration
and assignment using a function call
if (x < y) { ... }                  // if statement
for (let i = 0; i < 10; i++) {      // for loop statement
  ...
}
console.log('Hello');               // function call
statement
```

In these examples, **actions are being performed**, such as declaring and assigning variables, evaluating conditions, or executing loops. Note that function calls like `sum(5, 6)` can act as part of expressions when their returned values are used or assigned. However, a standalone function call like `myFunction();` can be considered a statement if it's used for its side effects and not for the return value. Thus, the context determines whether a function call is treated primarily as an expression or a statement.

Let and Const

JavaScript has two main ways to declare variables: `let` and `const` (we are not going to lean about `var` intensionally).

Variable declaration statement with `let` :

The `let` keyword allows you to declare a variable that can be reassigned. Here's an example:

```
let name = 'Alice';
```

In this statement, we're declaring a variable called `name` and assigning it the value `"Alice"` . The `let` keyword allows us to change the assignment of the `name` variable at a later point in the program. For example (in the second assignment we don't use the `let` keyword):

```
name = 'Suzan';
```

This reassigns the value of `name` from `"Alice"` to `"Suzan"`.

Variable declaration statement with `const`:

On the other hand, the `const` keyword allows you to declare a variable that is read-only after the initial assignment. That means once you set a value to a variable declared with `const`, you cannot reassign a new value to it. This is helpful for values that you want to remain the same throughout your code.

Here's an example:

```
const pi = 3.14159;
```

Here, we're declaring a constant variable `pi` and assigning it the value `3.14159`. The `const` keyword signifies that the variable cannot be reassigned.

The `const` keyword is typically used more often than `let` because in most cases, we do not want to change the value assigned to a variable. The `let` keyword is mainly used when we know we will need to change the variable's value later in the program.

It's important to note that when an object or an array is assigned to a `const` variable, the content of the object or array can be altered, but the variable cannot be reassigned entirely. Here's an example:

```
const numbers = [1, 2, 3];
numbers.push(4); // This is okay
console.log(numbers); // Outputs: [1, 2, 3, 4]

numbers = [5, 6, 7]; // This will throw an error because
we tried to assign a new array to the variable
```

In this example, although `numbers` is a `const` variable, we can still modify its content by adding elements to the array. However, we can't reassign `numbers` to a completely new array. That would throw an error.

```
const student = {
    name: 'John',
    age: 20,
};
```

In this statement, we're declaring an object `student` and assigning it properties `name` and `age`.

Even though `student` is declared with `const`, we can still change its properties:

```
student.name = 'Bob';
student.age = 21;
console.log(student); // Outputs: { name: "Bob", age: 21
}
```

But if we try to reassign `student` to a completely new object, it will throw an error:

```
student = {
    name: 'Charlie',
    age: 22,
}; // This will throw an error
```

So, similar to arrays, when an object is declared with `const`, its properties or values can be changed, but the variable itself cannot be reassigned to a new primitive, object or array.

If statement:

```
if (name === 'Alice') {
    console.log('Hello, Alice!');
}
```

This is an if statement. If the condition inside the parentheses is `true` (in this case, if `name` is exactly equal to `"Alice"`), then the code inside the braces `{}` will execute.

```
let age = 18;

if (age > 21) {
    console.log('You may enter the club.');
} else {
    console.log(
        'Sorry, you must be at least 21 to enter.',
    );
}
```

In this example, the if statement first checks if the `age` is greater than `21`. If this condition is `true`, the console logs "You may enter the club.". If the condition is not met (i.e., `age` is not greater than `21`), then the code in the `else` block is executed, logging "Sorry, you must be at least 21 to enter.". The `else` statement provides an alternative action for the program to perform when the `if` condition is not met.

Function declaration statement:

```
let age = 20;

function celebrateBirthday() {
    age = age + 1;
}
```

In this example, we declare a `let` variable `age` and assign it the value `20`. Then, we declare a function `celebrateBirthday` that, when called, will increase the value of `age` by 1.

Please note that this function doesn't return anything and directly modifies the `let` variable `age`, thus creating a side effect. It is called an "impure functions". Generally, this is not a good practice to have impure functions with side effects, as it can lead to unexpected results in larger, more complex programs.

Return statement:

```
function celebrateBirthday(age) {
    return age + 1;
}
```

In this example, the `return` statement is used within the `celebrateBirthday` function, giving back the result of adding 1 to the `age` input. So, if we call `celebrateBirthday(20)`, the function will return `21`.

This function is a "pure function", meaning it doesn't alter any external state (like global variables or console output), and **given the same input, it will always produce the same output**. That means, that the function will always return `21` if we will pass to it `20`.

Typically, when a function returns a value, it's good practice to assign that returned value to a variable for later use. It's best to use a `const` declaration for this if we don't plan on reassigning the variable. For example:

```
const myAge = 20;
const myNewAge = celebrateBirthday(myAge);
```

Now, `myNewAge` will hold the value returned from `celebrateBirthday(myAge)`, which is `21`. Typically, we will not be interested in changing this assignment.

Using pure functions is a best practice in functional programming as it minimizes side effects. Side effects refer to changes that a function makes that affect states outside its scope. This could include altering a variable declared outside the function, writing to the console, etc.

By minimizing side effects, our code becomes more predictable, easier to test, understand, and debug. In other words, pure functions enhance code reliability and readability by providing predictable outputs and not altering any external state.

Data Types

JavaScript provides us with several built-in data types:

- **String:** A sequence of characters used for text. It's written inside quotes, like `"Hello"` or `'Hello'`. Strings can hold any character like `''שלום` or `'مرحبا'` or `'Aa1234!@'`.
- **Number:** Represents numeric values and can hold only numbers (without quotes): `534`.
- **Boolean:** Represents either of two values: `true` or `false` (without quotes).
- **Object:** A collection of related data and/or functionality (which consists of several variables and functions — which are called properties and methods when they are inside objects). For example (note that the properties are written without quotes):

```
const car = {
    model: 'Tesla',
    isElectric: true,
    year: 2023,
    getDetails: function () {
        if (this.isElectric) {
            return `This car is an electric car. The
model is ${this.model} and it was manufactured in
${this.year}.`;
        } else {
            return `This car is not an electric car. The
model is ${this.model} and it was manufactured in
${this.year}.`;
        }
    },
};
```

Null: Null in JavaScript is a falsy value, represents the intentional absence of any value. It is often used when a variable's value is still unknown, not yet determined, or intentionally left empty. It signals that the variable should not hold any value or reference to an array or an object at this point in time. It is useful when we request data from an external API, and the data isn't available yet. So we initialize the variable with `null`, and once we get the data from the API, we assign it to the variable.

Null is also a falsy value, and any other value is true, so we can use this fact in a conditional expression, to check whether we have a value or not.

Undefined: In JavaScript, a variable that has been declared but not yet assigned a value will automatically be given the falsy value, `undefined`. This is different from `null` as it's not a value that developers should assign. Instead, it's the default state for variables that have been declared without a value. This helps to indicate that the variable's value is not yet known or not yet relevant in the current context of the program or that an error occurred. never assign an `undefined` value to a variable. Instead, and if needed, initialize the variable with `null`.

```
const text = 'Hello, World!'; // String
const num = 10; // Number
const isTrue = true; // Boolean
const person = {  // Object
    firstName: 'Alice',
    lastName: 'Smith',
};
let absent = null; // Null
let notAssigned; // Undefined, as we've declared it but
not assigned any value yet
```

Operators

We use operators to perform operations on variables and values. JavaScript has arithmetic, assignment, comparison and logical operators.

JavaScript uses arithmetic operators (+ - * /) to compute values

```
(5 + 6) * 10;
let a = 5; // Assignment operator
let b = 10; // Assignment operator
let sum = a + b; // Assignment and an arithmetic
operation
a += b; // Equivalent to a = a + b, it's an example of an
assignment operator
```

JavaScript Identifiers / Names

In JavaScript, identifiers are used for naming variables. An identifier can contain letters, digits, underscores (_), and dollar signs ($). However, the first character cannot be a digit.

A JavaScript name must begin with:

- A letter (A-Z or a-z)
- A dollar sign ($)
- Or an underscore (_)

Examples of valid JavaScript identifiers:

```
let name;
let NAME;
let _name;
let $name;
let name123;
let _123;
```

JavaScript is Case Sensitive

JavaScript is case-sensitive. This means `age` , `Age` , and `AGE` are three different variables. Always remember this when naming your variables:

```
let age = 25;
let Age = 30;
let AGE = 35;

console.log(age); // Output: 25
console.log(Age); // Output: 30
console.log(AGE); // Output: 35
```

JavaScript and Camel Case

In JavaScript, camel case often begins with a lowercase letter and the first letter of each subsequent concatenated word is capitalized. This convention is commonly used in JavaScript for variable and function names.

Here's an example:

```
let myNameIsAlice;
let numberOfApples;
```

In the examples above, `myNameIsAlice` and `numberOfApples` are examples of camel case notation. This style is easy to read and is widely used in JavaScript.

Functions

Functions are block of code designed to perform a particular task. A JavaScript function is defined with the `function` keyword, followed by a name, followed by parentheses `()`. The code to be executed by the function is placed inside curly brackets `{}`.

```
function greet(name) {
    console.log(`Hello ${name}!`);
}

greet('Ori'); // Calling the function will output: Hello Ori!
```

Function Declarations

A function declaration (also known as a function statement) defines a function with the specified parameters.

```
function add(x, y) {
    return x + y;
}

console.log(add(5, 3)); // Output: 8
```

In this example, `add` is a function that takes two parameters `x` and `y`, and returns their sum.

Function Expressions

A function expression is a way to define functions inside an expression. Here, the function can be anonymous; it does not need a name because it is stored in a variable.

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

console.log(multiply(5, 3)); // Output: 15
```

In this example, `multiply` is a variable that holds a function. The function takes two parameters `x` and `y`, and returns their product.

In JavaScript, functions are first-class objects. That means that functions can be treated like any other value. You can assign them to variables, put them in arrays, pass them as arguments to other functions, or even return them as values from other functions.

A **"function expression"** is a way to define a function and at the same time assign it to a variable. It's just another way to create a function, but it shows how flexible JavaScript is with functions.