# JavaScript

# General

## Introduction

JavaScript is a programming language that powers the dynamic behavior on most websites. Alongside HTML and CSS, it is a core technology that makes the web run.

## Comments

- Single-line comments are created with two consecutive forward slashes `//`.
- Multi-line comments are created by surrounding the lines with `/*` at the beginning and `*/` at the end.
- Comments are good ways for a variety of reasons like explaining a code block or indicating some hints, etc.

```JavaScript
// This line will denote a comment

/*
Hi!
I'm Jacky :)!
The following is my code.
*/
```

## `console.log()`

`console.log()` is used to log (print) messages to the console. It can also be used to print objects and other info.

```JavaScript
console.log('Hi there!');
// Prints: Hi there!
```

# Variables

## Introduction

A variable is a container for data that is stored in computer memory. It is referenced by a descriptive name that a programmer can call to assign a specific value and retrieve it.

```JavaScript
// Examples of variables
let myName = "Tammy";
const found = false;
var age = 3;
console.log(myName, found, age);
// Prints: Tammy false 3
```

## Declaring Variables

To declare (make) a variable in JavaScript, any of these three keywords can be used along with a variable name:

- `var` is used in pre-ES6 versions of JavaScript (not recommended).
- `let` is the preferred way to declare a variable when it can be reassigned.
- `const` is the preferred way to declare a variable with a constant value.

```JavaScript
// different types of variable declarations
var age;
let weight;
const numberOfFingers = 20;

// let example
let count;
console.log(count); // Prints: undefined
count = 10;
console.log(count); // Prints: 10

// const example
const numberOfColumns = 4;
numberOfColumns = 8; // TypeError: Assignment to constant variable.
```

# Data Types

## Introduction

Data types are the categories used in programming to define the type, or kind, of information a variable can hold. Each type specifies the kind of operations that can be performed on the data.

For example, numbers are used for math operations, strings represent text, and booleans represent true/false values. Understanding data types helps programmers organize and manipulate information correctly within a program.

## Strings

Strings (text) are a type of data. They are any grouping of characters (letters, spaces, numbers, or symbols) surrounded by single quotes ' or double quotes ".

```JavaScript
let singleQuotes = 'Wheres my bandit hat?';
let doubleQuotes = "Wheres my bandit hat?";
```

## Numbers

Numbers are a type of data. They include the set of all integers (whole) and floating point (decimal) numbers.

```JavaScript
let amount = 6;
let price = 4.99;
```

## Booleans

Booleans are a type of data. They can be either true or false.

```javascript
let lateToWork = true;
```

# Conditionals

## Introduction

The “control flow” is the order in which statements are executed in a program. The default control flow is for statements to be read and executed in order from left-to-right, top-to-bottom in a program file.

Control structures such as conditionals (`if` statements and the like) alter control flow by only executing blocks of code if certain conditions are met. These structures essentially allow a program to make decisions about which code is executed as the program runs.

## Comparison Operators

Comparison operators are used to compare two values and return true or false depending on the validity of the comparison. In JavaScript, `==` (loose equality) compares two values for equality after converting them to a common type, while `===` (strict equality) checks for equality without type conversion, meaning both the value and type must match. Here are some examples of comparison operators:

- `==` loose equal
- `!=` loose not equal
- `==` strict equal
- `!=` strict not equal
- `>` greater than
- `>=` greater than or equal
- `<` less than
- `<=` less than or equal

JavaScript

```
1 > 3       // false
3 > 1       // true
250 >= 250  // true

1 === 1     // true
1 === 2     // false
1 === '1'   // false
1 == '1'    // true
1 == '2'    // false

1 != 2      // true
1 != 1      // false
1 !== '1'   // true
1 != '2'    // true
```

## `if` Statement

An `if` statement accepts an expression with a set of parentheses:

- If the expression evaluates to a truthy value, then the code within its code body executes.
- If the expression evaluates to a falsy value, its code body will not execute.

JavaScript

```
const isMailSent = true;
const homeworkDone = false;

if (isMailSent) {
  console.log('Mail sent to recipient');
}
// Prints: Mail sent to recipient

if (homeworkDone) {
  console.log('Time to sleep');
}
// Prints:
// Nothing is printed because "homeworkDone" is not equal to a truthy value
```

## `else` Statement

An `else` block can be added to an `if` block. The `else` block will be executed only if the `if` condition fails.

```JavaScript
const isTaskCompleted = false;

if (isTaskCompleted) {
  console.log('Task completed');
} else {
  console.log('Task incomplete');
}
// Prints: Task incomplete
```

### `else if` Clause

After an initial `if` block, `else if` blocks can each check an additional condition. An optional `else` block can be added after the `else if` block(s) to run by default if none of the conditionals evaluated to truthy.

```JavaScript
const size = 10;

if (size > 100) {
  console.log('Big');
} else if (size > 20) {
  console.log('Medium');
} else if (size > 4) {
  console.log('Small');
} else {
  console.log('Tiny');
}
// Print: Small
```

# Logical Operators

### Introduction

Logical operators are symbols used in programming to combine or modify boolean values, which are either true or false. These operators allow for more complex decision-making in code, helping to control the flow of programs based on multiple conditions.

## Logical Operator `!`

The logical NOT operator `!` can be used to do one of the following:

- Invert a Boolean value.
- Invert the truthiness of non-Boolean values.

```JavaScript
let lateToWork = true;
let oppositeValue = !lateToWork;

console.log(lateToWork);
// Prints: true
console.log(oppositeValue);
// Prints: false
```

## Logical Operator `||`

The logical OR operator `||` checks two values and returns a boolean. If one or both values are truthy, it returns `true`. If both values are falsy, it returns `false`.

| A | B | A \|\| B |
|---|---|---|
| `false` | `false` | `false` |
| `false` | `true` | `true` |
| `true` | `false` | `true` |
| `true` | `true` | `true` |

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

```
10 > 5 || 10 > 20;    // true
false || false;       // false
10 > 100 || 10 > 20;  // false
```

### Logical Operator `&&`

The logical AND operator `&&` checks two values and returns a boolean. If *both* values are truthy, then it returns `true`. If one, or both, of the values is falsy, then it returns `false`.

| A | B | A && B |
|---|---|---|
| `false` | `false` | `false` |
| `false` | `true` | `false` |
| `true` | `false` | `false` |
| `true` | `true` | `true` |

JavaScript
```
true && true;      // true
1 > 2 && 2 > 1;    // false
true && false;     // false
4 === 4 && 3 > 1;  // true
```

# Functions

### Introduction

Functions are one of the fundamental building blocks in JavaScript. A *function* is a reusable set of statements to perform a task or calculate a value. Functions can be passed one or more values and can return a value at the end of their execution. In order to use a function, you must call it. **We will primarily only work with CALLING functions in Sprig, not defining functions.**

## Function Declaration

Function *declarations* are used to create named functions. These functions can be called using their declared name. Function declarations are built from:

- The `function` keyword.
- The function name.
- An optional list of parameters separated by commas enclosed by a set of parentheses `()`.
- A function body enclosed in a set of curly braces `{}`.

```JavaScript
function sum(num1, num2) {
  return num1 + num2;
}
```

## Calling Functions

Functions can be *called*, or executed, elsewhere in code using parentheses following the function name. When a function is called, the code inside its function body runs. *Arguments* are values passed into a function when it is called.

```JavaScript
// Defining the function
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function
sum(3, 5); // 8

let age = sum(2, 4);
console.log(age);
// Prints: 6
```

## Function Parameters (OPTIONAL, ADVANCED)

Inputs to functions are known as *parameters* when a function is declared or defined. Parameters are used as variables inside the function body. When the function is called, these parameters will have the value of whatever is *passed* in as arguments. It is possible to define a function without parameters.

```javascript
// The parameter is name
function sayHello(name) {
  return 'Hello, ' + name + '!';
}

let myName = "bob";
console.log(sayHello(myName));
// Prints: Hello, bob!

console.log(sayHello("Joe"));
// Prints: Hello, Joe!
```

## `return` Keyword (OPTIONAL, ADVANCED)

Functions return (pass back) values using the `return` keyword. `return` ends function execution and returns the specified value to the location where it was called. A common mistake is to forget the `return` keyword, in which case the function will return `undefined` by default.

```javascript
// With return
function sum(num1, num2) {
  return num1 + num2;
}

// Without return, so the function doesn't output the sum
function sum(num1, num2) {
  num1 + num2;
}
```

## Arrow Functions (OPTIONAL, ADVANCED)

Arrow function expressions were introduced in ES6. These expressions are clean and concise. The syntax for an arrow function expression does not require the `function` keyword and uses a fat arrow `=>` to separate the parameter(s) from the body.

There are several variations of arrow functions:

- Arrow functions with a single parameter do not require `()` around the parameter list.
- Arrow functions with a single expression can use the concise function body which returns the result of the expression without the `return` keyword.

JavaScript

```
// Arrow function with two parameters
const sum = (firstParam, secondParam) => {
  return firstParam + secondParam;
};
console.log(sum(2,5)); // Prints: 7

// Arrow function with no parameters
const printHello = () => {
  console.log('hello');
};
printHello(); // Prints: hello

// Arrow functions with a single parameter
const checkWeight = weight => {
  console.log(`Baggage weight : ${weight} kilograms.`);
};
checkWeight(25); // Prints: Baggage weight : 25 kilograms.

// Concise arrow functions
const multiply = (a, b) => a * b;
console.log(multiply(2, 30)); // Prints: 60
```

# Arrays

## Introduction

Arrays are lists of ordered, stored data. They can hold items that are of any data type. Arrays are created by using square brackets, with individual elements separated by commas.

```javascript
// An array containing numbers
const numberArray = [0, 1, 2, 3];

// An array containing different data types
const mixedArray = [1, 'chicken', false];
```

## Index

Array elements are arranged by *index* values, starting at `0` as the first element index. Elements can be accessed by their index using the array name, and the index surrounded by square brackets.

```javascript
// Accessing an array element
const myArray = [100, 200, 300];

console.log(myArray[0]); // Prints: 100
console.log(myArray[1]); // Prints: 200
console.log(myArray[2]); // Prints: 300
```

## `.length`

The `.length` property of a JavaScript array indicates the number of elements the array contains.

```javascript
const numbers = [1, 2, 3, 4];

numbers.length // 4
```

## Mutable

JavaScript arrays are *mutable*, meaning that the values they contain can be changed.

Even if they are declared using `const`, the contents can be manipulated by reassigning internal values or using methods like `.push()` and `.pop()`.

```JavaScript
const names = ['Alice', 'Bob'];

names.push('Carl');
// ['Alice', 'Bob', 'Carl']

names[2] = 'Jacky';
// ['Alice', 'Bob', 'Jacky']
```

## `.push()`

The `.push()` method of JavaScript arrays can be used to add one or more elements to the end of an array. `.push()` mutates the original array and returns the new length of the array.

```JavaScript
// Adding a single element:
const cart = ['apple', 'orange'];
cart.push('pear'); // ['apple', 'orange', 'pear']

// Adding multiple elements:
const numbers = [1, 2];
numbers.push(3, 4, 5); // [1, 2, 3, 4, 5]
```

## `.pop()`

The `.pop()` method removes the last element from an array and returns that element.

JavaScript

```
const ingredients = ['eggs', 'flour', 'chocolate'];

const poppedIngredient = ingredients.pop(); // 'chocolate'
console.log(ingredients); // Prints: ['eggs', 'flour']
console.log(poppedIngredient); // Prints: 'chocolate'
```

# Loops

## Introduction

A *loop* is a programming tool that is used to repeat a set of instructions. *Iterate* is a generic term that means “to repeat” in the context of *loops*. A *loop* will continue to *iterate* until a specified condition, commonly known as a *stopping* condition, is met.

## While Loop

The `while` loop creates a loop that is executed as long as a specified condition evaluates to `true`. The loop will continue to run until the condition evaluates to `false`. The condition is specified before the loop, and usually, some variable is incremented or altered in the `while` loop body to determine when the loop should stop.

JavaScript

```
let i = 0;

while (i < 5) {
  console.log(i);
  i += 1;
}

// Output: 0, 1, 2, 3, 4
```

## For Loop

A `for` loop declares looping instructions, with three important pieces of information separated by semicolons `;`:

- The *initialization* defines where to begin the loop by declaring (or referencing) the iterator variable
- The *stopping condition* determines when to stop looping (when the expression evaluates to `false`)
- The *iteration statement* updates the iterator each time the loop is completed

```
JavaScript
for (let i = 0; i < 4; i += 1) {
  console.log(i);
};

// Output: 0, 1, 2, 3
```

## Break Keyword

Within a loop, the `break` keyword may be used to exit the loop immediately, continuing execution after the loop body. Here, the `break` keyword is used to exit the loop when `i` is greater than 5.

```
JavaScript
for (let i = 0; i < 99; i += 1) {
  if (i > 5) {
     break;
  }
  console.log(i)
}

// Output: 0 1 2 3 4 5
```