# Agenda

- Basics
- Datatypes
- Functions
- Higher Order Functions

# Types of javascript

1. Internal
    - It is inserted into the documents by using the script tag
    - spript tag provides a block to write the java script programs

```
<script>
    JS code goes here
</script>
```

2. External
    - To use the predefined programs of any javascript library.

```
<script src = "myscript.js"></script>
```

# Variables

- It is a container to store the data
- To declare a variable data type MUST NOT be used in its declaration
- To create variables in JS we can use

1. let
    - It is a normal variable whose value can be changed multiple times
2. const
    - It is a variable whose value cannot be changed once initialized.

- Syntax: let name = initial value; const name = initial value;
- E.g.

```
let num = 100; // number
const salary = 4.5; // number
let = “test”; // string
const firstName = ‘steve’; // string
let canVote = true; // Boolean
```

# Data Types

- In JS, all Data Types are inferred (automatically decided by JS .....)
- typeof operator can be used to get the type of the variable.
- Types

1. number:
    - It supports both whole and decimal numbers
    - E.g.
        - num = 100;
        - salary = 4.5;
2. string: collection of characters
    - E.g.
        - firstName = “steve”;
        - lastName = ‘Jobs’;
3. boolean:
    - may have only true or false value
    - E.g.
        - canVote = true;
        - canVote = false;
4. undefined:
5. object:
    - Represents JS object

## Built-in Values

1. NaN
    - Not a Number
    - Is of type number
    - E.g. console.log(parseInt(“test”));
2. Infinity:
    - When a number is divided by 0
    - E.g. answer = 10 / 0; // Infinity
3. undefined:
    - No value/type present
4. null:
    - No object.
    - is of type "object"



## Template Strings

- strings can be enclosed in '..', "..", or `` .
- templated strings to print var values: var=${varname}

```
console.log(`addition: ${n1} + ${n2} = ${res}`);
```

## Function

- function is a reusable block of statement
- defined once & can be called multiple times.
- Function must be declared along with its definition
- To declare a function, function keyword is used
- Syntax: function ( ) { // function body }

# Function Argument types

1. function argument types are inferred. fn can be called multiple times with different types of args.

```
// function arguments are not type-safe
function add(x, y) {
    const z = x + y;
    console.log(`${x} + ${y} = ${z} of type ${typeof z}`);
}

// function can be called with different arg types
add(22, 7);
add("Hello", "World");
add(10, true);
add("Bond", 7);
add("Flag", false);

// passing more arguments will ignore remaining args.
add(10, 20, 30);

// passing less arguments will consider remaining args undefined
add(10);
```

2. if multiple fns defined with same name, last fn definition override/replace the earlier ones.

```
function f1() {
console.log("f1() called.");
}
function f1(x, y) {
console.log("f1(x, y) called: " + x + ", " + y);
}
function f1(x) {
console.log("f1(x) called: " + x);
}
f1(10, 20); // calls last f1(x).
```

3. By default, if any arg is not passed, it is undefined. Instead default value can be provided a.k.a. "default parameters".

```
// Arguments are assigned to params from left to right (irrespective of default values given to params).

// function definition
function add(x = 0, y, z = 0) {
let r = x + y + z;
console.log(`add(): ${x} + ${y} + ${z} = ${r}`);
}
// function calls
add();
add(10);
add(22, 7);
add(11, 22, 33);
```

## Built-In Functions

- JS have built-in functions as well.
- e.g. parseInt(), parseFloat(), eval()

```
// string to number (int)
let n1 = parseInt("123");
console.log("n1 = " + n1); // 123

let n2 = parseInt("123.45");
console.log("n2 = " + n2); // 123

let n3 = parseInt("Bond007");
console.log("n3 = " + n3); // NaN

let n4 = parseInt("007Bond");
console.log("n4 = " + n4); // 7

 // string to number (float)
let v1 = parseFloat("3.14");
console.log("v1 = " + v1); // 3.14

let v2 = parseFloat("123");
console.log("v2 = " + v2); // 123

// eval() to evaluate string expr
let v3 = eval("2 + 3 * 4");
console.log("v3 = " + v3); // 14
```



## Function Alias

- In JS, function can be treated as object.
- It can be assigned to another reference a.k.a. function alias.
- Using alias the actual function can be called.

- Function alias concept can be used to pass fn as argument to fn, Return fn from another fn, Create anonymous fns and Lambda expressions.
- E.g.

```
// function definition
function add(x, y) {
let z = x + y;
console.log(`add(): ${x} + ${y} = ${z}`);
}
console.log("add() -- type = " + typeof add + ", name = " + add.name);
add(22, 7);

// function alias
const calc = add;
console.log("calc -- type = " + typeof calc + ", name = " + calc.name);
calc(23, 8);
```

- Anonymous functions are created using function keyword, but not given any name.
- Anonymous fns assigned to alias and called via aliases only.
- Function without a name is called as anonymous function
- E.g.

```
// anonymous function
const subtract = function (x, y) {
let z = x - y;

console.log(`subtract(): ${x} - ${y} = ${z}`);
};
subtract(22, 7);
```

## Arrow Functions

- Arrow functions are a modern, concise way to write function expressions in JavaScript, introduced in ECMAScript 2015 (ES6).
- They provide a shorter syntax than traditional function expressions, making the code more readable, especially for simple, one-line functions.
- An arrow function has three main parts:
    1. Parameters: Enclosed in parentheses ().
    2. The "fat arrow": The => syntax.
    3. Function body: The code to be executed.
- They can be one-liner or multiline. However, one-liners are preferred.

```
const add = (a, b) => {
    const res = a + b
    console.log(res)
}
```

```
// Omitting parentheses for a single parameter:
const square = n => {
    const res = n * n
    console.log(res)
}

//Omitting curly braces and return for a single expression: For one-line arrow functions, the return is implicit.
const square = x => x * x;

//Returning an object literal: To implicitly return an object, you must wrap the curly braces in parentheses to avoid a syntax error.
const createObject = () => ({ message: "Hello" });
```

- If the function body has multiple lines, you must use curly braces and an explicit return statement.

## Higher Order Functions

- A higher-order function (HOF) is a function that either takes one or more functions as arguments, returns a function, or both.
- This is possible in JavaScript because functions are "first-class citizens," meaning they can be treated like any other value (such as a string or number).
- A function that is passed as an argument to a higher-order function is known as a callback function.
- This is a common pattern used to handle asynchronous operations and create reusable code.

```
//executer function
// Higher Order functions
function executer(n1, n2, fn) {
    const res = fn(n1, n2)
    console.log('res - '+res)
}

const add = (n1,n2)=>{
    const res = n1+n2;
    return res;
}
// Here add is called as callback function
executer(10,20,add)

// we can pass the callback function by directly declaring it in the argument
executer(10,20,(n1,n2)=>n1-n2)
executer(10,20,(n1,n2)=>n1*n2)
```