Computer Language

- definitions
    - set of instructions (algorithm)
    - implementation of algorithm
    - helps us to interact with hardware
    - medium of communication with hardware
- types
    - based on the level
        - **low level**
            - binary (0s and 1s)
        - **middle level**
            - interacts with CPU
            - Assembly language
            - opcodes: operation code => binary
            - e.g. ADD A, B
        - **high level**
            - developer can write human understable code
            - compiler or interprter convers the human understandable to machine (CPU) understandable (ASM)
            - e.g. C++, Java, Python
    - based on how the application gets generated
        - **compiled language**
            - compile: converting human understable to machine (CPU) understandale
            - compiler: program which does compilation
            - executable:
                - program which contains only ASM instructions (machine understandable)
                - native applications
                - always platform (OS) dependent
                - faster than interpreted program
            - requires compiler
            - the entire program gets converted into executable
            - if program contains error, compiler detects these error at compilation time
            - e.g. C, C++
        - **interperted language**
            - interpretation: which converts the human understandable to machine (CPU) understandable line by line
            - interpreter: program which does interpretation
            - no executable gets generated
            - if there is any error, it will get detected at the run time

        - program will be always platform (OS) independent
        - programs will be always slower than native applications
        - e.g. html/CSS, JS, bash scripting
    - **mixed language**
        - shows behavior from both (compiled as well as interptreted)
        - uses compiler as well as interpreter
        - e.g. Java, **Python**

JavaScript

- is a scripting language
- is an object oriented programming language
- is functional programming language
- is used for adding dynamic behavior (e.g. clicking a button) in the website
- is loosely typed language
    - there is not type checking
    - the data types are inferred
    - the data types are dynamically assigned
- does NOT support pointers

JS Fundamentals

- **rules**
    - the semicolon is optional when one statement is written on one line

```
console.log('this is JS tag in head section')
console.error('this is an error')
```

    - semicolon is required when multiple statements are written on one line

```
console.log('this is JS tag in head section')
console.error('this is an error')
```

    - conventions
        - use camel case when declaring function, variables or classes
        - i.e. start the name with lower case and use upper case for first letter of meaningful word
        - e.g.
            - firstName
            - printPersonInfo()
- **variable**
    - is a placeholder to store a value in memory

- it is a mutable
- to declare a variable use **let** keyword
- the variable must be declared without a data type
- e.g.

```
let firstName = 'steve'
let salary = 10.6

let age = 40

// can update the value
age = 41
```

- **constant**
  - is a placeholder whose value **CAN NOT** be changed
  - this is immutable (readonly)
  - to declare a constant use **const** keyword
  - the constant must be declared without a data type
  - e.g.

```
const pi = 3.14

// can not update the value of a constant
// pi = 100
```

> ALWAYS TRY TO PREFER CONSTANT THAN A VARIABLE

- **pre-defined objects**
  - console
    - object that represents the browser console
    - methods
      - log() : used to print debugging messages on the browser's console
      - info() : used to print information on the browser's console
      - warn() : used to print warning messages on the browser's console
      - error() : used to print error on the browser's console
  - window
    - object that represents the browser's window (UI)
    - this is a default object used when calling a method
    - methods
      - alert() :

        - prompt() :
        - confirm() :

- **Pop ups**
    - alert
        - used to show a message to the user on web browser's window
        - e.g.

```
// window.alert('this is an alert')
alert('this is an alert')
```

    - prompt
        - used to take an input from user
        - e.g.

```
const username = window.prompt('Enter your name')
console.log('user name = ' + username)
```

    - confirm
        - used to get input in terms of boolean answer to a question
        - e.g.

```
const answer = window.confirm('Do you want to have break?')
if (answer) {
  console.log('lets take a break of 10 minutes')
} else {
  console.log('lets continue')
}
```

- **pre-defined values**
    - **undefined**
    - **NaN**
        - Not a Number
        - NaN has data type as number
        - does not represent a valid number
        - e.g.

```
// NaN
console.log(10 * 'test1')
```

- **Infinity**
  - has a data type as number
  - e.g.

```
// Infinity
console.log(`10 / 0 = ${10 / 0}`)
```

- **data types**
  - all data types in javascript will be inferred
  - the data type will be decided by JavaScript by inspecting the current value in the variable
  - types
    - **number**
      - represents whole numbers and floating point (decimal) numbers
      - e.g.

```
// number
let num = 100
console.log('data type of num = ' + typeof num) // number

// number
let salary = 10.5
console.log('data type of salary = ' + typeof salary) //
number
```

    - **string**
      - collection of characters
      - string can be declared using
        - single quotes (')
        - double quotes (")
        - back quotes (`)
      - e.g.

```

```

```
// string
let firstName = 'steve'
console.log('data type of firstName = ' + typeof
firstName) // string

// string
let lastName = 'Jobs'
console.log('data type of lastName = ' + typeof
lastName) // string

// string
let address = `
      address line 1,
      address line 2,
  `
console.log('data type of address = ' + typeof address)
// string
```

- **boolean**
  - represents only true or false values
  - e.g.

```
// boolean
let canVote = false
console.log('data type of canVote = ' + typeof canVote)
// boolean
```

- **undefined**
  - in JS, undefined is both: data type as well as pre-defined value
  - prepresents a variable without having initial value
  - e.g.

```
// undefined
let myvar
console.log('myvar = ' + myvar) // undefined
console.log('data type of myvar = ' + typeof myvar) //
undefined
```

- **object**

- **statements**
  - the smallest unit that executes
  - types

        - assignment
        - declaration
        - comment

- **operators**
    - mathematical operators
        - addition (+)
            - the plus operator works as
                - mathematical addition when bothe params are numbers

```
// 30
console.log(10 + 20)
```

                - string concatination opetator when one of the operands is a string

```
// test1test2
console.log('test1' + 'test2')
```

                - when one of the operands is a string and other is not a string, then all the params get converted to string data type

```
// 1020
console.log(10 + '20')
```

        - division (/)
            - mathematical division
            - e.g.

```

```

        - multiplication (*)
            - mathematical multiplication of two operands
            - the answer will be always a number
            - e.g.

```

```

```
// 200
console.log(10 * 20)

// NaN
console.log('test1' * 'test2')

// 400
console.log('10' * 40)

// 400
console.log('10' * '40')
```

- modulo (%)
- subtraction (-) :

- comparison operators
  - double equals to (==)
    - also known as value equality operator
    - checks ONLY the values of operands
    - e.g.

```
// true
console.log(50 == 50)

// true
// '50' is having a value of 50
console.log(50 == '50')
```

  - triple equals to (===)
    - also known as identity equality operator
    - checks both the value as well as the data types of the operands
    - always prefer === over ==
    - e.g.

```
// true
console.log(50 === 50)

// false
// 50 is a number while '50' is a string
console.log(50 === '50')
```

        - not equals to (!=) :
        - not equals to (!==) :
        - less than (<) :
        - greater than (>) :
        - less than equals to (<=) :
        - greater than equals to (>=) :
    - logical operators
        - and (&&) :
        - or (||) :

- **type conversion**
    - anything to string
        - any data type can be converted into string by concatinating with plus operator
        - e.g.

```
// '10'
console.log(10 + '')
```

    - string to number
        - parseInt
            - convert string to integer number (by discarding the decimal precision)
            - e.g.

```
// 10
console.log(parseInt('10'))

// 10
console.log(parseInt('10.60'))

// 10
console.log(parseInt('10test'))

// NaN
console.log(parseInt('test10'))
```

        - parseFloat

        - convert a string to a decimal number (by keeping the decimal precision)
        - e.g.

```
// 10
console.log(parseFloat('10'))

// 10.60
console.log(parseFloat('10.60'))

// 10
console.log(parseFloat('10test'))

// NaN
console.log(parseFloat('test10'))
```

- **function**
    - a block of code which can be reused
    - reusable block of code having a name
    - types
        - empty function
            - function with no code in the body
            - e.g.

```
// empty function
function function0() {}
```

        - parameterless function
            - which does not accept any parameter
            - e.g.

```
// parameterless function declaration
function function1() {
  console.log('inside function1')
}

// function call
function1()
```

        - parameterized function

        - which accepts at least one parameter
        - e.g.

```
function function2(param) {
  // param = true
  console.log(`inside the function2`)
  console.log(`param = ${param}, type of param =
${typeof param}`)
}

function2(20)
function2('test1')
function2(true)
```

    - **variable length argument function**
        - a function which can accept variable length of arguments
        - every function in JS receives a hidden parameter named arguments
            - which contains a list of all the arguments passed while the function call
        - e.g.

```
function add() {
  // console.log('inside add')
  console.log(arguments)

  let sum = 0
  for (let index = 0; index < arguments.length;
index++) {
    sum += arguments[index]
  }
  console.log(`addition = ${sum}`)
}

add(10, 20)
add(10, 20, 30)
add(10, 20, 30, 40)
```

- parameters
    - the parameter(s) of a function can not have static data type(s)
    - the number and type of parameters will be controlled by the caller instead of the function
    - caller can pass

    - same number of parameters
    - less number of parameters than expected
        - the missing arguments will be treated as undefined
    - more number of parameters than expected
        -
        - the extra parameters will be discarded

- every function in JS receives two hidden parameters
    - **arguments**: used to get all the arguments in an array
    - **this**: used to point the current object
        - **this** points to window in a function inside javascript in html
        - **this** porint to Object if called oustide the html

- e.g.

```
function function1(n1, n2) {
  console.log('inside function1')
}

// n1 = 10, n2 = 20
function1(10, 20, 30)

// n1 = 10, n2 = 20
function1(10, 20)

// n1 = 10, n2 = undefined
function1(10)

// n1 = undefined, n2 = undefined
function1()
```

- default parameters
    - also known as optonal parameters as you may not pass the value to these parameters
    - one or more paramters may be declared with a default value
    - the default value will be used if the caller has not passed the argument for the optional paramters
    - e.g.

```
function multiply(num1, num2 = 10) {
  const multiplication = num1 * num2
  console.log(`multiplication = ${multiplication}`)
}

// num1 = 40, num2 = 100
multiply(40, 100)
```

```
// num1 = 40, num2 = 10
multiply(40)
```

- function alias
  - giving a function another name
  - same function can be called with original function or the function alias
  - e.g.

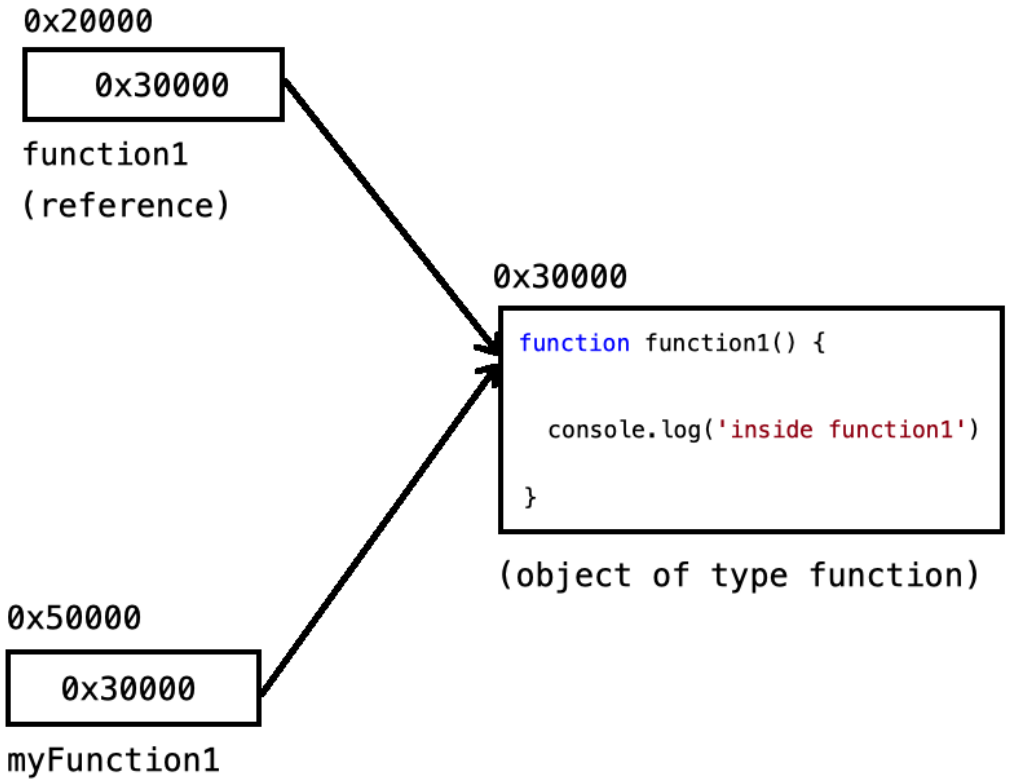
```
function function1() {
  console.log('inside function1')
}

// calling the function1 by its orginal name
function1()

// function alias
const myFunction1 = function1

// calling the function1 by its function alias
myFunction1()
```

```
function function1() {
    console.log('inside function1')
}
```




```
const myFunction1 = function1
```

- **collection**
  - collection of values (similar to array of values in C and C++)
  - properties
    - **length**
  - methods
    - **push**
      - used to append a value at the end of the collection
      - e.g.

```
const numbers = [10, 20, 30]

// [10, 20, 30, 40]
numbers.push(40)
```

- **html + JS**
- **predefined functions**
    - typeof() : used to get the data type of a variable

JS as functional programming language

- in JS, function is considered as first class citizen
    - function can be called by passing another function as an argument
    - a variable can be created for a function (function alias)
        - a function can be considered as a variable
    - one function can be considered as a return value of another function

OOP JS

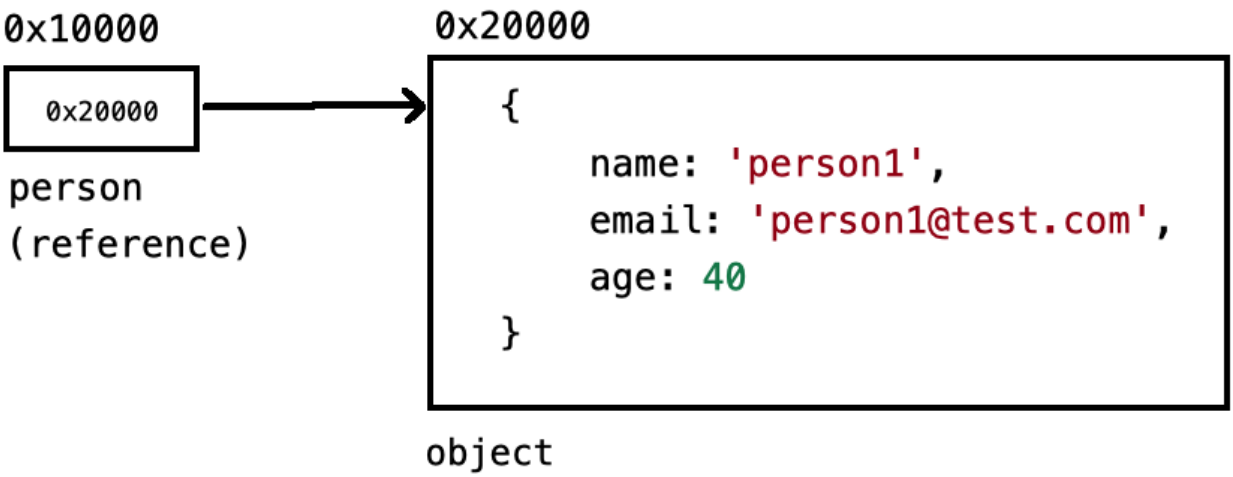
- JS is object oriented programming language
- object can be created by
    - using JSON
    - using Object
    - using constructor functions
    - using keyword class

object

- similar to instance of a class in other language
- collection of key-value pairs
- keys are also known as properties
- e.g.

```
const person = {
  name: 'person1',
  email: 'person1@test.com',
  age: 40,
}

// properties - name, email, age
// values     - person1, person1@test.com , 40
```



- to access value of a property
  - use subscript ([] syntax)
    - used when a property is having special character like space
    - e.g.

```
console.log(`name: ${p1['name']}`)
console.log(`email: ${p1['email']}`)
console.log(`age: ${p1['age']}`)
```

```
const person = {
  'first name': 'steve',
  'last name': 'jobs',
}

console.log(`first name = ${person['first name']}`)
console.log(`last name = ${person['last name']}`)
```

  - use dot (.) syntax
    - e.g.

```
console.log(`name: ${p1.name}`)
console.log(`email: ${p1.email}`)
console.log(`age: ${p1.age}`)
```

    - can not be used when a property has a special character like space

JSON

- JavaScript Object Notation

- way to create an object
- JSON supports
    - object
        - collection of key-value pairs
        - e.g.

```
const person = {
  name: 'person1',
  email: 'person1@test.com',
  age: 40,
}
```

    - array
        - collection of objects
        - e.g.

```
const persons = [
  { name: 'person1', email: 'person1@test.com' },
  { name: 'person2', email: 'person2@test.com' },
  { name: 'person3', email: 'person3@test.com' },
  { name: 'person4', email: 'person4@test.com' },
]
```

creating object using Object

- Object is a root function provided by JS
- everything in JS is an Object

node

module

- any javascript file having an extension .js
- the NodeJs embeds an object named module in every module to present the current module
- the module object has following properties
    - id: the identification of the module
    - path: the file which is representing this module
    - exports:
        - contains the list of functions/variables/constant which can be exported from the current module