



JavaScript



Overview

Introduction

Advanced Functions

Fundamentals

Arrays and Collections

Fundamentals

Generators and Iterators

Objects, Prototype and Classes

Promises and Async Programming



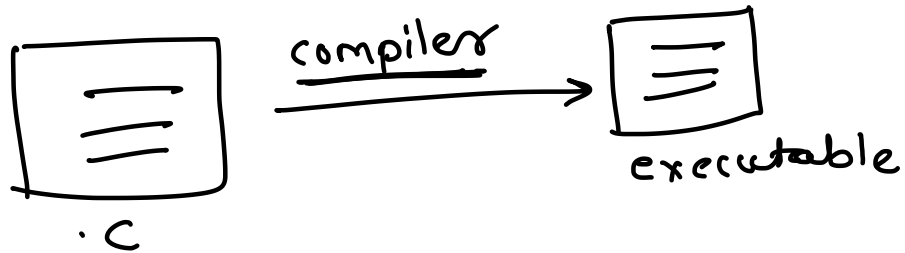
Introduction



Overview

- JavaScript was initially created Brendan Eich to make the web pages alive
- The earlier name of JS was LiveScript
- The programs written in this language are called as scripts
- They can be written in the web page to execute at the time of page load
- Browser can execute the JS code because it has a built-in engine called a “JavaScript Virtual Engine”

JS Execution



JavaScript Execution Engine

- It is a program or an interpreter which executes JS code
- A JS engine can be implemented as a standard interpreter or just-in-time compiler that compiles the JS code to bytecode in some form
- There are many implementations of JS engines
 - **V8** – Open source developed by Google and written in C++
 - SpiderMonkey – developed by Mozilla Foundation and today powers Firefox
 - Chakra – developed by MS for IE (JScript) and Edge (JS)
 - JavaScriptCore – also known as Nitro, developed by Apple for Safari
 - Rhino – managed by Mozilla and written in Java ↳ WebKit
 - KJS – developed for KDE's Konqueror browser
 - JerryScript – is a lightweight engine for IoT



JavaScript Today

- Today JS programs can be executed not only in the browsers but also on the server or on any device that has a JavaScript engine running
- Applications
 - Used in Web pages
 - Business applications (using TypeScript in Angular, React, Vue)
 - Utility Applications (console scripting)
 - Game development (Unity)
 - Mobile application development (Cordova, React-Native, Nativescript)
 - Server side development (Express)



Fundamentals



Features

- Case sensitive name (Name)
- Semicolon is optional if every statement is written on separate line
- Code can be commented using // (one line) or /* */ (multiline)



Variables

- A variable is a named storage for data
- We can store any value of any type in a variable
- Variables are mutable → changed
- JavaScript provides a keyword **let** to declare a variable
- Syntax
 - let <varname> = <value>
- e.g.
 - ✓ ■ let age = 30
 - ✓ ■ let firstName = 'steve'
 - ✓ ■ let email = "person@test.com"
 - ✓ ■ let address = `A-100, Near Sunbeam,
Pune, 411000`



↳ block scope



Constants

- Variables which do not change the values
- Constants are immutable , read only
- Preferred over variables
- JS provides a keyword **const** to declare a constant
- Syntax
 - const <name> = <value>
- e.g.
 - const pi = 3.14
 - const gst = 18



Data Types

- In JavaScript, all data types are inferred
- Automatically set by JS by checking the "current" value in the const or variable
- JS provides a operator typeof to check the variables datatype [can be used as a function as well]

`int num = 100;`
↓
`num = 100`



Data Types

▪ Number

- both whole and decimal numbers
- Can not represent integer values larger than 2^{53} or less than -2^{53}

▪ BigInt

- Represent big number
- Uses 'n' at the end

▪ String

- Represents string values
- Can use single quote, double quote or back quote

▪ Boolean

- Represents logical value
- Can have only true or false

▪ null *↪ object*

- Special value

▪ undefined

- Special value represents a variable which does not have any value

▪ Objects

- Objects represents collection of key value pairs

NaN

* Not a Number

* number → datatype

* special value



Type Conversions

- Used for converting value from one type to another
- String conversion
 - Uses String() to convert to string
- Number conversion
 - Number() to convert to number
 - parseInt() to covert to whole number
 - parseFloat() to convert to decimal number
- Boolean conversion
 - Uses Boolean() to covert to boolean



Comparison

- ✓ Equality (==) ↗ avoided
 - Checks only value and ignores the data type
 - E.g.
 - "20" == 20 ⇒ true
- ✓ Identity/strict equality (===)
 - Checks both value and data type
 - E.g. "20" === 20 ⇒ false



Array

- Array represents collection of values
- e.g.
 - `const numbers = [10, 20, 30, 40, 50]`
 - `const countries = ["india", "usa", "uk"]`



Functions

- Block of organized, reusable code that is used to perform single, related action
- JavaScript provides **function** keyword to declare a function
- Syntax

```
function greet() {  
    console.log("welcome to JS")  
}
```
- Types
 - Parameter less function
 - Parameterized function



Objects



Overview

- Object represents collection of properties and methods
Key-value pairs
- In JS, object can be created and modified dynamically [properties can be added dynamically]
functions inside the object
- JS provides following ways to create objects
 - ✓ Object Literals
 - ✓ Constructor functions
 - ✓ Classes

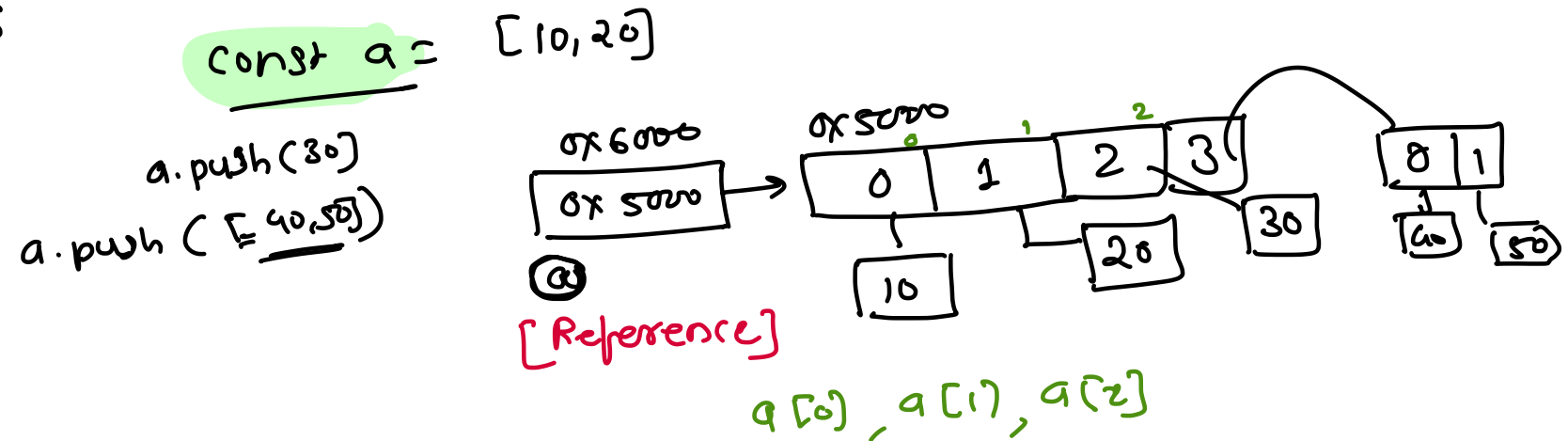
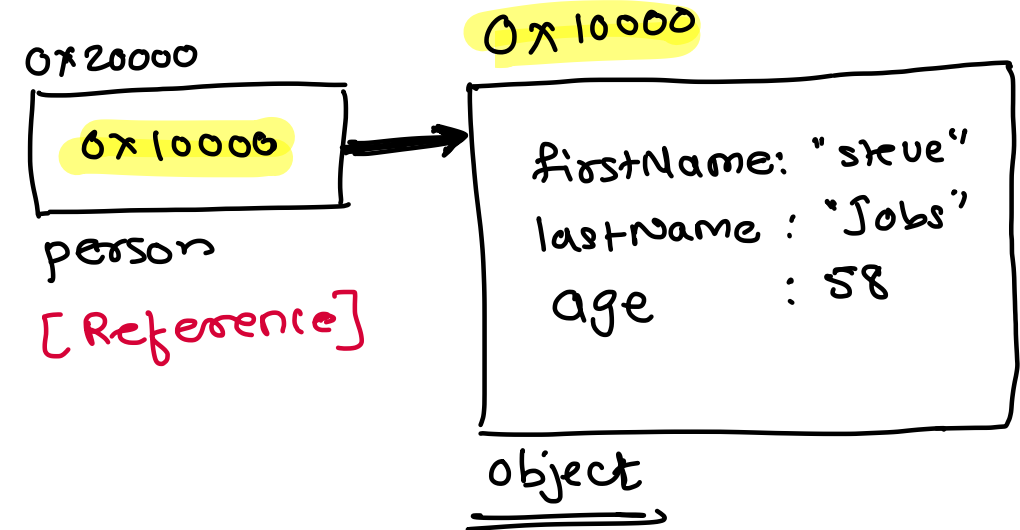


Object Literals

- Simplest way to create objects
- Uses `{}` to enclose the key-value pairs
- E.g.

```
const person = {  
  firstName: 'Steve',  
  lastName: 'Jobs'  
}
```

`person.age = 58`



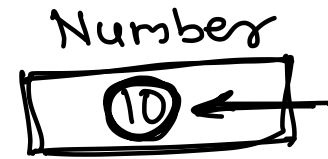
Object Literals

- Object literals can use property shorthand
- E.g.

```
function registerUser(firstName, lastName) {  
  const person = {  
    firstName, lastName  
  }  
}
```

```
registerUser('steve', 'jobs')
```

num = 10
↳

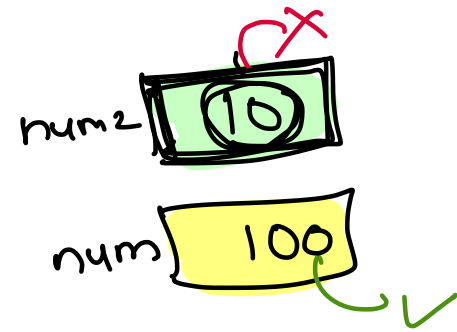


Number,
String,
Boolean } primitive

const num2 = 10

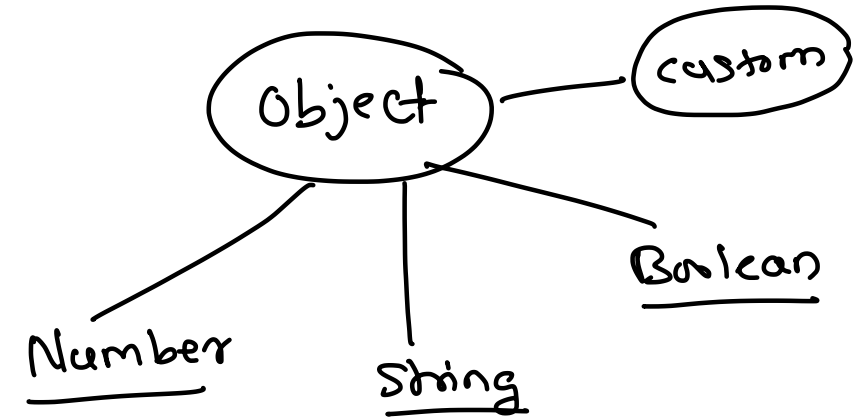
let num = 10

num = 100



Object Properties

- Properties are used to store data inside object
- Properties are key-value pairs
- Properties can be added by using
 - ✓ [] syntax
 - ✓ Dot (.) syntax
- E.g.
 - `person['age'] = 30`
 - `person['email'] = 'person@test.com'`



Inspecting Object properties

- JS provides Object.keys() to inspect the object properties
- You can also use for..in loop to get all the properties of an object
- E.g
 - `const p = {name: 'person1', address: 'pune'}`
 - `console.log(Object.keys(p))`
 - `for (let property in p) {`
 - `console.log(property)`
 - `}`



Property Descriptor

- Property descriptor can be used to get the information about the property
- E.g.
 - `const p = {name: 'person1', address: 'pune'}`
 - `console.log(Object. getOwnPropertyDescriptor(p, 'name'))`
- Modifying property
 - Property can be modified by using property descriptor
 - To make the property read only immutable
 - ~~`Object.getOwnPropertyDescriptor(person, 'firstName', {writable: false})`~~
 - To skip the property from enumerating
 - ~~`Object.getOwnPropertyDescriptor(person, 'firstName', {enumerable: false})`~~
 - Non-enumerable properties will not be serialized
 - To lock a property from modification
 - ~~`Object.getOwnPropertyDescriptor(person, 'firstName', {configurable: false})`~~



Property Getter and Setter

■ Getter

- Also known as inspector
- Method used to return value
- E.g.

```
Object.defineProperty(p, 'fullname', {  
  
  get: function() {  
    return  
      `${this.firstName} ${this.lastName}`  
  }  
  
})
```

■ Setter

- Also known as mutator
- Method used to modify value
- E.g.

```
Object.defineProperty(p, 'fullname', {  
  
  set: function(value) {  
    const parts = value.split(' ')  
    this.firstName = parts[0]  
    this.lastName = parts[1]  
  }  
  
})
```



Using Constructor function

- Generally preferred when multiple objects are required of similar type
- Constructor function is the function which is used for creating new objects
- Generally written starting with capital letter
- Should be executed using new keyword
- e.g.

```
// using Object constructor function  
const person = new Object()
```

```
// using custom function  
function Person(name, address) {  
    this.name = name  
    this.address = address  
}
```



Function Prototype

- A function prototype is the object instance that will become the prototype for all the objects created using the function as a constructor
- To access the function prototype use prototype property on the function
- E.g.

```
function Car(model, company) {  
    this.model = model  
    this.company = company  
}
```

```
console.log(Car.prototype)
```



Object Prototype

- An object prototype is the object instance from which the object is inherited
- To access the object prototype use `__proto__` prototype
- E.g.

```
function Car(model, company) {  
    this.model = model  
    this.company = company  
}
```

```
const c1 = new Car('i20', 'Hyundai')  
console.log(c1.__proto__)
```



Prototype under the hood

```
function Car(model, company) {  
  this.model = model  
  this.company = company  
}
```

Car.prototype.price = 5

const c1 = new Car('i20', 'hyundai') ←

const c2 = new Car('nano', 'tata') ←

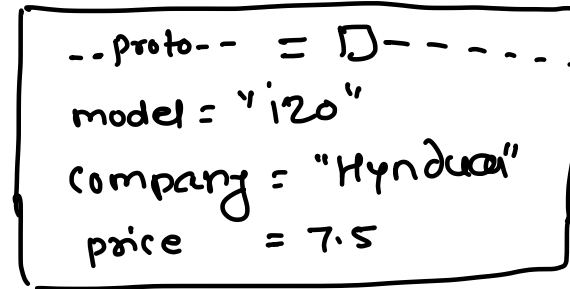
c1.price = 7.5 ←

✓ console.log(c1.price) = 7.5

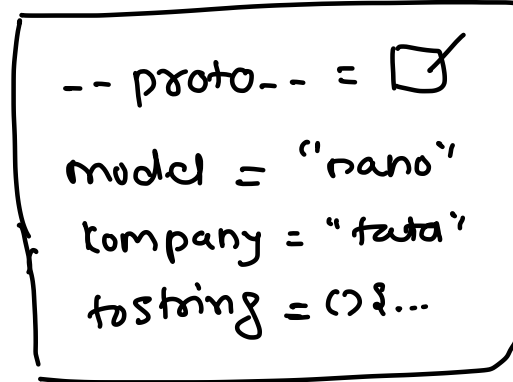
✓ console.log(c2.price) = 5

Car.prototype.price = 3

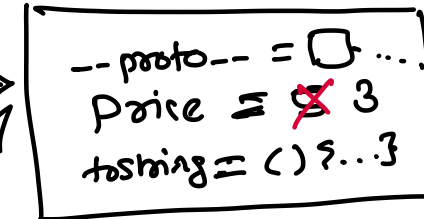
c1 object



c2 object

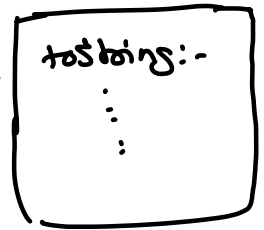


Car function Prototype



Object

Object



① c1.price = 5

② c2.price = 5

✗

③ c1.price = 7.5

④ c2.price = ⑤

✗

⑤ c1.price = 7.5

⑥ c2.price = ③

✗

⑦ c1.toString() ⑧ c2.toString()

Class

- JS supports classes in the latest version of ECMA
- You need latest versions of all the browsers to run this code
- NOTE: IE does not support JS classes
- Benefits of using classes
 - ✓ ■ Makes the coding simpler
 - ✓ ■ Similar to other languages
 - ✓ ■ Preferred using constructor function or object literals
- Syntax

```
class <name> {  
    // class body  
}
```



Adding constructor

- No matter which class you are writing, a constructor in JS always have same name **constructor**
- Like other languages it will be called automatically
- Unlike other language you can have only one constructor in the class
- E.g.

```
Class Person {  
    constructor() {  
        // constructor body  
    }  
}
```



Adding methods

- Method is a function inside a class
- A method does not require any function keyword for declaration
- A method can be parameterless or parameterized
- Unlike other languages
 - JS does not provide any kind of access specifiers
 - JS does not support method overloading
- E.g.

```
class Person {  
    ....  
    ✓ canVote() {  
        return this.age >= 18  
    }  
}
```



Adding getter and setter

- Getters and setters are used to set or get values of properties
- get and set keywords are used to declare them
- E.g.

```
Class Person {  
    ...  
  
    get fullName() {  
        return `${this.firstName} ${this.lastName}`  
    }  
  
    set fullName(name) {  
        ....  
    }  
}
```



Inheritance

- JS uses extends keyword to extend one class [subclass] from another [super]
- Like other languages, subclass inherits all the properties and methods from super class
- E.g.

```
class Student extends Person {  
    ....  
}
```



Method overriding

- Changing the behavior in the subclass
- To override a method simply use the same name as that of the super class method
- e.g.

```
class Person {  
    printInfo() { .... }  
}
```

```
class Student extends Person {  
    printInfo() { .... }  
}
```



Equality

- Equality
 - Should be avoided
- Identity equality
 - Type safe
 - Convenient and concise
 - +0 is equal to -0
- Object.is()
 - Type safe
 - Verbose
 - +0 is not equal to -0



Advanced Functions

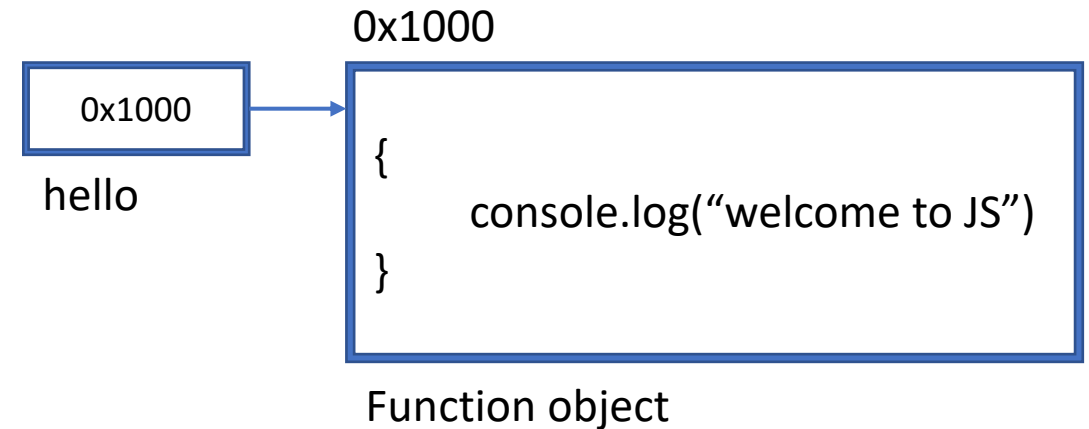


How functions are handled?

- Internally the a function object gets created for every function and function name is used as its reference
- The function object contains the function body
- JS uses call() method to call the function
- E.g.

```
function hello() {  
    console.log("welcome to JS")  
}
```

- The hello function can be called by
 - `hello.call(<context>)`



Function Context

- Functions in JavaScript run in a specific context, and using the **this** variable we have access to it
- All standard functions in the browser run under the Window context
- All functions use Module as a context when run using Node
- Functions defined under an object or a class (another function) will use the context of the object it was created in
- However, we can also change the context of a function at runtime, either before or while executing the function
- E.g.
 - `const person = {name: 'person1', address: 'pune'}`
 - `function printInfo() {`
 - `console.log(`name = ${this.name}, address = ${this.address}`)`
 - `}`
 - `printInfo.call(person) // way 1`
 - `const boundPrintInfo = printInfo.bind(person) // way 2`
 - `boundPrintInfo()`



Function Alias

- Another name given to an existing function
- Similar to function pointer in other languages
- E.g.

```
function helloFunction() {  
    console.log("inside hello")  
}
```

```
helloFunction()
```

```
const myHelloFunction = helloFunction  
myHelloFunction()
```



Arrow functions

- Simpler way to write a function expression
- Why to use them
 - Shorter syntax
 - this derives its value from enclosing lexical scope
- E.g.

```
const greet = () => {  
    console.log("welcome to JS")  
}
```

```
greet()
```



Default Parameters

- Function can assign default values to the parameters
- Such parameters can be optionally passed while calling the function
- E.g.

```
function sayHi(person = "person1") {  
    console.log(`hello ${person}`)  
}
```

```
sayHi()           // hello person1  
sayHi("steve")    // hello steve
```



Variable length arguments function

- A function accepting variable length of arguments
- In JavaScript, every function accepts a hidden parameter arguments which can be used to receive all the arguments
- E.g.

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

```
add(10, 20)           // 30  
add(10, 20, 30)       // 60  
add(10, 20, 30, 40)   // 100
```



Rest Parameters

- The rest parameter syntax allows us to represent an indefinite number of arguments as an array
- Think about it as the remaining parameters
- If present, the rest parameter must be the last parameter
- E.g.

```
function printInfo(firstName, lastName, ...details) {  
    console.log(` firstName: ${firstName}, lastName: ${lastName}`)  
    for (let arg of details) {  
        console.log(arg)  
    }  
}
```

```
printInfo('steve', 'jobs')  
printInfo('steve', 'jobs', 'apple', 'CEO')
```



Spread Operator

- Exactly opposite of rest parameters
- **Spread syntax** allows an iterable such as
 - an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected or
 - an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected
- E.g.
 - `const dateFields = [2020, 3, 25]`
 - `const d = new Date(...dateFields)`



Functional programming

- Functional programming is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data
- Features
 - Pure functions
 - First class and higher order functions
 - Recursion
 - Type system
 - Referential transparency

