



Javascript

Introduction

- Started as the programming language of the web
- Only way to execute Javascript was via a web browser
- Used primarily to make the web pages dynamic
 - Respond to user actions
 - Implement frontend logic and so on
- So ...
 - HTML defines the structure of web apps
 - CSS adds design to web apps
 - Javascript makes them dynamic



Introduction (Cont.)

- With time it has changed into a much more powerful programming language
- Advanced concepts like Object Oriented Programming have been introduced
- With the advent of Node JS it has transformed from a mere frontend or browser based scripting language into a general purpose programming language



Frontend capabilities

- It can change the content of web pages
- It can change the attribute values
- It can change CSS content
- It can hide/ show elements
- And much more



Linking Javascript with HTML

- We can link Javascript with HTML in 2 ways - Internal and External
- Internal
 - By placing the Javascript code inside `<script>` element either inside head or body elements
- External
 - Writing Javascript code in a separate file with extension `.js` and link it using `src` attribute of `<script>` element



Linking Javascript with HTML (Cont.)

Internal

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script>
    function showVals(){
      alert(document.getElementById("uname").value);
    }
  </script>
</head>
<body>
  <form>
    <label>Username:</label><br>
    <input type="text" id="uname"><br>
    <input type="button" value="Show" onclick="showVals()">
  </form>
</body>
</html>
```

Linking Javascript with HTML (Cont.)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <script src="myscript.js"></script>
</head>
<body>
  <form>
    <label>Username:</label><br>
    <input type="text" id="uname"><br>
    <input type="button" value="Show" onclick="showVals()">
  </form>
</body>
</html>
```

External

```
function showVals(){
  alert(document.getElementById("uname").value);
}
```

Variables and Literals

- Variables in Javascript, or any language for that matter are used for storing data values
- Literals are the fixed values we put into the variables
- ```
var price1 = 5;
var price2 = 6;
var total = price1 + price2;
```





# Identifiers

- Variables must be uniquely identified using names
- These names are called Identifiers



# Rules for Identifiers

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter
- Names are case sensitive (y and Y are different variables)
- Reserved words (like JavaScript keywords) cannot be used as names



# Declaring vs Assigning

- Declaration
  - `var carName;`
- Assignment
  - `carName = "Volvo"`
- If we just declare a variable but do not assign a value to it, it is treated as "undefined"



# Data types

- Primitive
  - Boolean
  - Undefined
  - Null
  - Number
  - BigInt
  - String
- Objects



# Javascript Arithmetic

- We can perform arithmetic operations on Javascript variables
  - `var x = 5, y = 6`
  - `var z = x + 5`
- We can add strings as well
- String addition has a special name – concatenation



# Operators - Arithmetic

| Operator | Description                  |
|----------|------------------------------|
| +        | Addition                     |
| -        | Subtraction                  |
| *        | Multiplication               |
| **       | Exponentiation               |
| /        | Division                     |
| %        | Modulus (Division Remainder) |
| ++       | Increment                    |
| --       | Decrement                    |



# Operators - Assignment

| Operator | Example              | Same As                 |
|----------|----------------------|-------------------------|
| =        | <code>x = y</code>   | <code>x = y</code>      |
| +=       | <code>x += y</code>  | <code>x = x + y</code>  |
| -=       | <code>x -= y</code>  | <code>x = x - y</code>  |
| *=       | <code>x *= y</code>  | <code>x = x * y</code>  |
| /=       | <code>x /= y</code>  | <code>x = x / y</code>  |
| %=       | <code>x %= y</code>  | <code>x = x % y</code>  |
| **=      | <code>x **= y</code> | <code>x = x ** y</code> |



# Operators - Precedence

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)





# Conditionals

- If statement

```
if(condition){
```

```
 //code to be executed when condition is evaluated to true
```

```
}
```



# Conditionals Hands On

- If else hands on
  - A school has following rules for grading system:
    - a. Below 35 - D
    - b. 35 to 50 - C
    - c. 50 to 80 - B
    - d. 80 and above - A
- Create a page with text box to accept a student's marks in English, Maths, Science subjects



# Conditionals Hands On

- Calculate the average marks of the student
- Use the above logic to derive his grade
- Display the grade in another text box on the page



# Switch Case Hands On

- Switch case hands on
  - Extend the above code to display appreciation message according to the grade
    - A - Excellent. Keep it up.
    - B - Well done. Can do better.
    - C - Need to work hard.
    - D - Uh oh. Better luck next time.
    - Default - Thats looks strange to me.



# While Hands On

- While loop hands on
  - Using while loop find the sum of first “n” natural numbers.
  - If  $n = 3$ , sum should be 6
  - If  $n = 5$ , sum should be 15 and so on




# For Loop Hands On

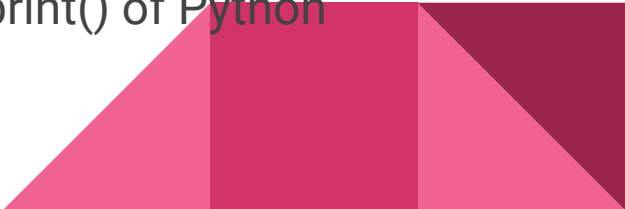
- For loop hands on
  - Do the same using for loop now
  - Using while loop find the sum of first “n” natural numbers.
  - If  $n = 3$ , sum should be 6
  - If  $n = 5$ , sum should be 15 and so on



# Using Console

- A quick way to start writing Javascript code is using Console under browser dev tools
  - Right click anywhere in the page and select Inspect
  - You are taken to a tab called Elements
  - Next to Elements tab we have Console tab
  - Select that and see that it resembles our command prompt
  - Here we can start typing any javascript code
  - Have fun!
- 

# Using Console (Cont.)

- Why does Console matter?
  - Well it can be used to quickly debug your code
  - Also while working with Javascript from HTML, it is easier to send debug statements to the console rather than using them in alert
  - So if you have done some complex calculation and want to check if its right
  - You can do `console.log(result)` instead of `alert(result)`
  - This is equivalent of `System.out.println()` of Java or `print()` of Python
- 



# Strings

- String is a very special data type in Javascript
- The simple string variable created below is actually an object of string class
- `my_str = "hello javascript"`



# Strings (Cont.)

- Since the string created above is actually an object, it has some useful properties and methods
  - length – property
    - `str_length = my_str.length`
- Different characters of the string can be extracted using the index based notation
  - `my_str[0]` – returns the first character
  - `my_str[str_length - 1]` returns the last character



# String (Cont.)

- Substring within a string can be identified using indexOf method
  - `my_str.indexOf("java")`
  - `my_str.indexOf("script")`
- `indexOf` returns the position of the substring within the string
- If the substring is not found it returns -1



# String (Cont.)

- Another useful method is slice()
- Can be used to slice or extract a piece of a string
  - `my_str.slice(0,3)`
    - Returns the substring starting at position 0 and upto but not including position 3
  - `my_str.slice(2)`
    - Returns substring starting from position 2 till end of the string



# String (Cont.)

- Changing case of letters in a string can be done using:
  - `my_str.toLowerCase()`
  - `my_str.toUpperCase()`
- We can use replace method to update a part of a string
  - `my_str.replace(' ', '*****')`

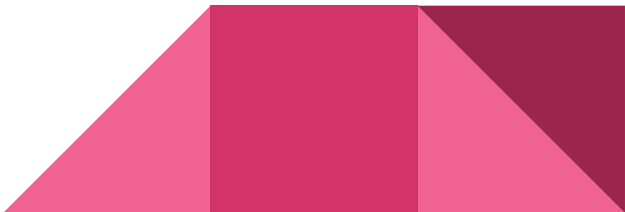


# String Methods At a Glance

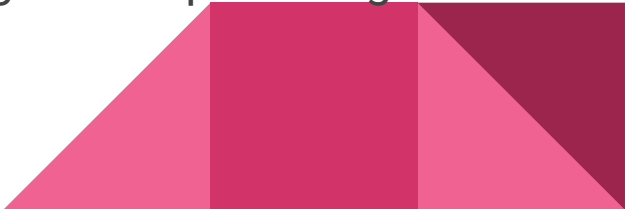
- String Methods

- `indexOf()`
- `lastIndexOf()`
- `search()`
- `slice(start, end)`
- `substring(start, end)`
- `substr(start, length)`
- `replace()`

- String Methods

- `toUpperCase()`
  - `toLowerCase()`
  - `concat()`
  - `trim()`
  - `charAt()`
  - `charCodeAt()`
  - `split()`
- 

# String Template Literals

- When it comes to working with Strings, ES6 introduced a very useful concept called Template Literals
  - Let us say we have 3 string variables name, age and occupation
  - We want to display them in a nice sentence like:
    - He is John, a 40 year old trainer
  - We have to do this something like this in Javascript
    - `console.log("He is " + name + ", a " + age + " year old " + occupation;`
  - Not very intuitive right? Too cumbersome and we might end up missing a space or putting extra spaces etc.
- 

# String Template Literals (Cont.)

- What if we have a more intuitive way to do the same?
- Just write something like:
  - `Let msg = "He is <name>, a <age> year old <occupation>.";`
  - `console.log(msg);`
- And let Javascript figure out how to interpret the variables and substitute them
- That is what precisely done by Template Literals
- Except that the symbols used are slightly different





# String Template Literals (Cont.)

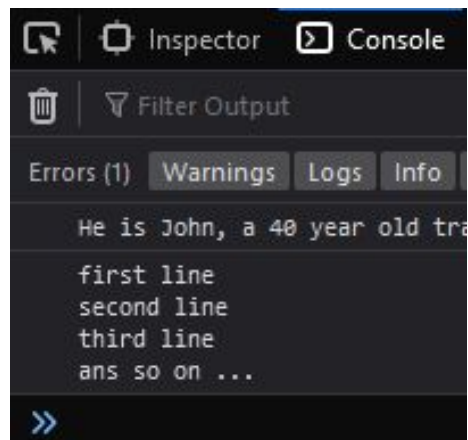
- Using Template Literal notation we can write it as follows:
  - `let msg = `He is ${name} a ${age} year old ${occupation}.`;`
  - `console.log(msg);`
- We just need to use tick (`) symbol to wrap our sentence instead of usual single or double quotes (' or ")
- And variables should be written inside the template literal as `${var_name}`



# String Template Literals (Cont.)

- Using ticks instead of quotes is also very useful when we want to create a multi line string

```
const multiLineMsg = `first line
second line
third line
ans so on ...`;
console.log(multiLineMsg);
```



# String Template Literals (Cont.)

- Example about writing strings in multi lines is not too convincing right?
- Wait till we have to start creating dynamic HTML code snippets from Javascript...



# Functions - Basics

- Functions are blocks of code that can be reused and that achieve a piece of functionality
- We have been using writing and using them already

```
function findProduct(p1, p2) {
 return p1 * p2;
 // returns the product of p1 and p2
}
```

P1 and p2 here are called parameters



# Functions - Basics (Cont.)

- Function invocation
  - Calling a function
  - On occurrence of an event
  - From within javascript code
  - Through automatic or self invocation
- Function return
  - Returning something from the function



# Functions - Basics (Cont.)

- Can be used as variables
  - `document.write("product of the values is " + findProduct(p1, p2))`
- Variables declared within a function are called local variables and their scope is limited to the function
- They get created when the function is invoked and deleted when the functions is exited



# Elements and Events

- All the HTML elements have events associated with them which get triggered on some action like:
  - When a user clicks the mouse
  - When a web page has loaded
  - When an image has been loaded
  - When the mouse moves over an element
  - When an input field is changed
  - When an HTML form is submitted
  - When a user strokes a key



# Elements and Events (Cont.)

- All the HTML elements have events associated with them which get triggered on some action like:
  - onload, onunload
  - onchange
  - onmouseover, onmouseout
  - onmousedown, onmouseup, onclick





# Arrays

- Kind of sequence data types
- String and number types store single values
- Arrays can be used to store a list of objects
- Two ways to create an array
  - `var array_name = [item1, item2, ...];`
  - `var array_name = new Array(item1, item2, ...);`



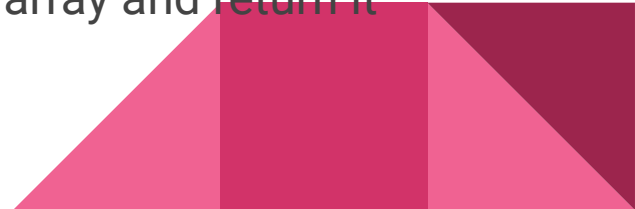
# Arrays (Cont.)

- Similar to strings elements of array are indexed
- So we can access each element of array using index
- Or we can access whole array just by using its name
- Length of an array can be found using `array.length`
- First element of array can be accessed using `array[0]`
- Last element of array can be accessed using `array[array.length - 1]`



# Arrays Methods

- We have lot of useful methods to play around with the arrays
- We can use instanceof operator to check if a variable is of type array or not
  - `cars instanceof Array` `□` true if cars is an array
- `array.toString()` will return the items of an array as a string
- Items can be added to and removed from array using push and pop methods
- `array.push(item)` will add item to the array
- `item = array.pop()` will remove the last item from the array and return it



# Arrays Methods (Cont.)

- `array.shift()` is similar to `pop` but instead of last it will remove the first and shift the indices of the remaining elements
- So second element will now become first
- `Push` and `pop` will add and remove element to the end of array
- If we want to add and remove elements from the beginning of the array we can use `unshift` and `shift`
- `array.unshift(item)` will add item to the beginning of the array



# Arrays Methods (Cont.)

- `Item = array.shift()` will remove the element from the beginning of the array and shift the indices of the remaining elements
- `sort()` method can be used to sort an array
- An array element at any position can be deleted using delete operator
  - `delete cars[1]`



# Arrays Hands On

- Create and print an array
- Sort the array using sort method and print sorted array
- Use array indexing to print elements of array from
  - First to last
  - Last to first
- Try the splice() method
- Try concat() method to merge arrays




# Type Conversion and Coercion

- Often we see a need to convert from one data type to the other
- For instance a value entered in an HTML textbox is always read as a string
- But to be able to do some calculation using it, we need to convert it into a number
- If we do this manually, it is called type conversion
- If Javascript implicitly does it in the background it is called type coercion



# Truthy and Falsy Values

- In the above slide we discussed about conversion or coercion of other data types
  - Truthiness and Falsiness is about boolean values
  - A value of any other type which gets converted or coerced to false is called falsy value and vice versa
  - There are only 5 falsy values in Javascript
  - They are 0, "", undefined, null, NaN
  - Rest all are truthy values
- 



# Equality Operators

- In Javascript we have 2 types of equality operators
  - `===` → Strict equality operator
  - `==` → Loose or just equality operator
- We have seen that in certain situations Javascript does type coercion or auto type conversion
- So the difference between the above operators is that
  - `===` does not do type coercion and returns true only if both sides are exactly the same
  - `==` does type coercion
- To avoid unwanted defects its recommended to use `===`

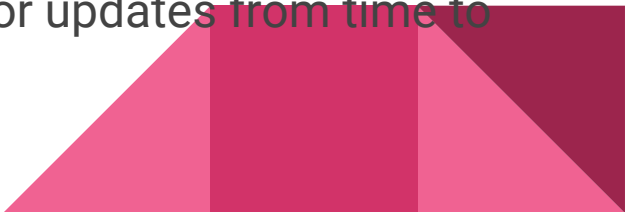


# Conditional or Ternary Operator

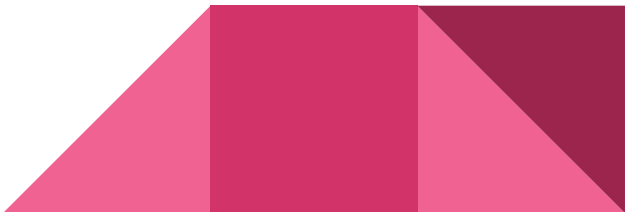
- Let us say we want to assign a value to a variable based on a condition
- Usual way of doing it is using if statement
- But that would be waste to write in statement just to assign a value to variable
- We have a shorter way to do it i.e. using ternary or conditional operator
  - `let salType = empType === "permanent" ? "salary" : "stipend";`



# Javascript History

- Was first developed in 1995 by Netscape and another similar variant by Microsoft in 1996
  - In 1997 it was standardized by ECMA and was released as ECMA Script which works on all browsers
  - In 2009 ECMA Script 5 or ES5 was released as a major update
  - Again in 2015 ES6 was released with even major updates
  - Now ES6 is standard and we will only be getting minor updates from time to time
- 

# Javascript History (Cont.)

- So Javascript is completely backward compatible
  - As a UI developer why should you worry about it?
  - Because the UI that we develop should execute on any browser that the customer might be using
  - In Dev env it is fine as we can use latest browsers
  - But we do not have control on which browser the customer might be using
  - They might be using old browsers
  - And Javascript is not forward compatible
- 

# Javascript History (Cont.)

- To solve this we break our UI development into
  - During development
  - During Production
- During development, use the latest version of Chrome and we have access to latest Javascript features
- During Production we convert latest Javascript back to the standard ES5 using a process called Transpiling and Polyfilling

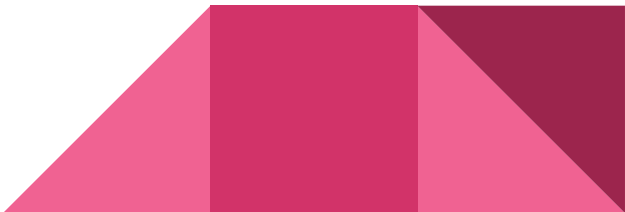


# Javascript History (Cont.)

- ES5 can be safely used as it supports even old browsers
- ES6/ ES2015, ES2020 etc. commonly called as ES6+ also work on all browsers with transpiling and polyfilling



# Variable Declaration

- The old way to declare variables using **var** keyword has certain limitations
  - These are overcome by introducing new way to declare variables using **let** and **const** keywords
  - Variables declared using **var** can be redeclared
  - Variables declared using **let** cannot be
  - Keywords **let** and **const** introduce new scope called block scope in addition to existing global and function scope
  - More on scope later
- 

# Functions Revisited





# Function Declaration vs Expression

- There are different ways in which functions can be written and invoked in Javascript
- 2 of them are function declaration and function expression
- Function declaration we have already seen in the previous section
- We just declare a function along with its definition associating with a name
- Later we can just call the function using its name wherever needed



# Function Declaration vs Expression (Cont.)

- Function Declaration

```
function calcAvgMarks(sc, math, eng) {
 return (sc + math + eng) / 3;
}
```

- Function Expression

```
const calcAvgMarksAgain = function (social, hindi, comps) {
 return (sc + math + eng) / 3;
}
```

- There will be certain situations in which function expressions are useful
- Which one to use? Your choice, unless there is a specific need to use one

# Arrow Functions

- Now a days Arrow functions are becoming popular in all programming languages
- It simply is a shorter way to write functions

The diagram illustrates the components of an arrow function. A code block contains two lines of JavaScript code. Above the code, three labels in light blue boxes with pointers identify parts of the first line: 'Function Name' points to 'calcAge', 'Function Parameter' points to 'birthYear', and 'Function Body' points to '2022 - birthYear'. Below the code, a label 'Parameter goes here' points to the value '1980' inside the function call.

```
const calcAge = birthYear => 2022 - birthYear;
console.log(calcAge(1980))
```

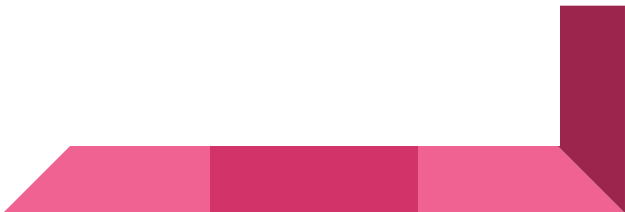
Function Name

Function Parameter


Function Body

Parameter goes here


# Arrow Functions (Cont.)

- More useful for function with one param and one statement
  - In the above case we can write it in single line and do not have to explicitly add a return statement
  - But for functions which need more than one statement we need to use curly braces and add explicit return statement
  - Again - When should you use arrow functions?
  - Depends on our convenience for now
- 


# Functions Calling Other Functions

- We write functions so that we can reuse the code as per DRY (Do not repeat yourself) principle
  - So it might very well happen that we will end up calling one function inside another function which in turn might be calling some other function and so on
  - We need to keep in mind not to make these call too complicated and intuitive
  - Remember that the parameters passed to the function are like variable which are local to that function
- 

# Functions Calling Other Functions (Cont.)

- So if we are expecting them to be available outside the functions, we must return them from the called function to the calling function
  - Beware of cyclic references - Function A calls Function B while Function B calls Function C which in turn calls Function A
  - Return statement should always be the last line in a function
  - Any code present after the return statement will never be reached
- 

# Objects

- Remember that Arrays data structure can be used to store multiple values (of different data types) in Javascript
  - Suppose we also want to store what each of the variables contain, then we can define a new data structure called Object
  - So in Objects we associate a 'key' with each 'value' that specifies what does that value signify
  - So Javascript objects are denoted as key value pairs
  - This is our first step towards object oriented programming
- 


# Objects (Cont.)

- While Arrays are ordered, Objects are unordered
- What does that mean?
- It simply means that the elements in an Array appear and can be retrieved in the same order in which they are stored
- Hence the elements in an Array have an index (ordering) associated with them
- But Objects are different





# Objects (Cont.)

- Create an Object and print it to console to see how are Objects different
  - We will see that elements are sorted by their keys
  - We do not need numbered indices to access elements of an object
  - We can simply retrieve a value from an Object using its key
  - Keys of an Objects are often referred to as Properties
  - So essentially Javascript Objects are used to define certain properties along with their values
- 

# Objects (Cont.)

- Values of an Objects are accessed using the key or property names
- There are two notations in which this can be done
- Dot notation and bracket notation
- Dot notation
  - `object.key`
  - `person.experience`
- Bracket Notation
  - `object['key']`
  - `Person['experience']` - key goes in as a string

# Objects (Cont.)

- Since we specify the key as a string in the Bracket notation, it gives us the flexibility
- We can specify part string concatenated with a string variable
- Or we can use a regular expression in case we do not exactly know what we are looking for and so on...
  - `const partNameKey = "Name";`
  - `person['first' + partNameKey]`
- Clearly each notation has its own advantages

# Objects (Cont.)

- We can use any option based on the need
- Remember we are just talking about Javascript Objects which is a very useful data structure but not Object Oriented Programming yet
- Now we can use the same dot or bracket notation to update an Object with a non existing property and its value
  - `object.newProperty = newPropertyValue`
  - `object['newProperty'] = newPropertyValue`

# Objects Methods

- We can also associate methods with objects
- These methods act on the properties of the object
- This is our next step towards object oriented programming
- These methods are essentially Javascripts functions
- Only function expressions are allowed in objects and not function declarations



# Objects Methods

- Every method associated with an object will have access to the object via the ***this*** keyword
- Using this keyword, the method can access any of the other properties associated with the object
- To understand how ***this*** keyword works, just print it in any method and see what it refers to



# Types of for loops

- **for**
  - Regular for loop
- **for in**
  - Used with objects to get a value associated with properties or keys
  - Can be used with Arrays also, but returns index instead of value
- **for of**
  - Used with Arrays to get values from the Array
- **forEach**
  - Used for function extrapolation



# Setting up Dev Env



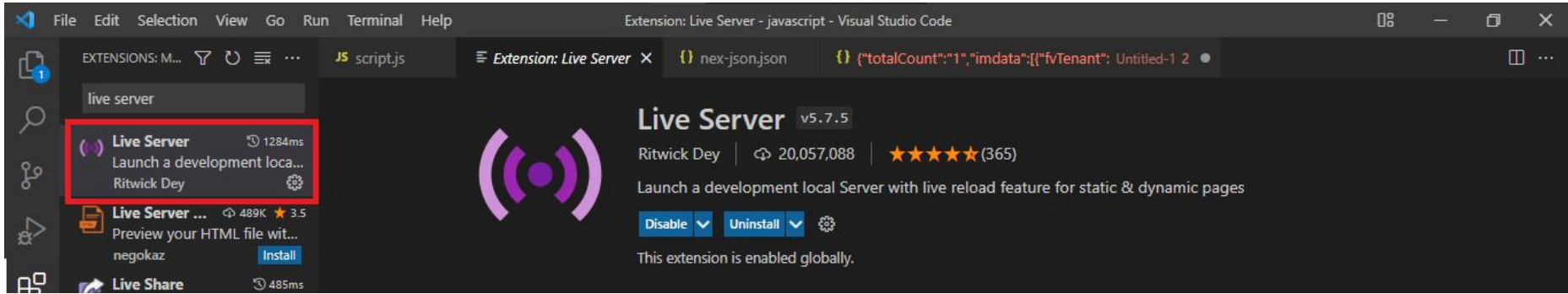


# Code Editor - Liver Server Plugin

- Recommended code editor is VS Code which is widely popular and all powerful
- Install a plugin called Liver Server to avoid reloading the page after making any changes to our script
- Go to extensions and search “Live Server”
- Install it and click “Go Live” button in the task bar



# Code Editor (Cont.)



# Code Editor - Prettier Plugin

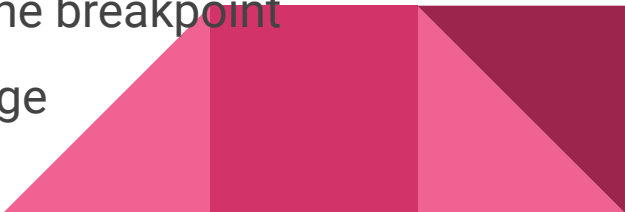


# Debugging - Using Console

- We have been using the following command to print something on console as part of debugging
  - `console.log`
- We can also use the following to write something as a warning or error using the below commands
  - `console.warn`
  - `console.error`
- We can also print an object in a nice format using
  - `console.table(object)`



# Debugging - Using Breakpoints

- We are used to placing breakpoints to in code editors to stop execution at some point as part of debugging
  - We can do this from the browser dev tools itself
  - Click the source tab next to console tab in dev tools
  - Shows the list of files loaded into the browser
  - Select the script we want to debug
  - Click next to the line number where we want to add the breakpoint
  - Now the execution stop there when we reload the page
- 

# Debugging - Using debugger

- Another way to debug is by placing the debugger command in the code
  - `debugger;`
- This will add a breakpoint automatically at this line and open up the debugger under dev tools



# DOM and DOM Manipulation



# DOM

- Document Object Model
- Defines the structure of HTML pages
- Usually represented as a tree with the document node as root

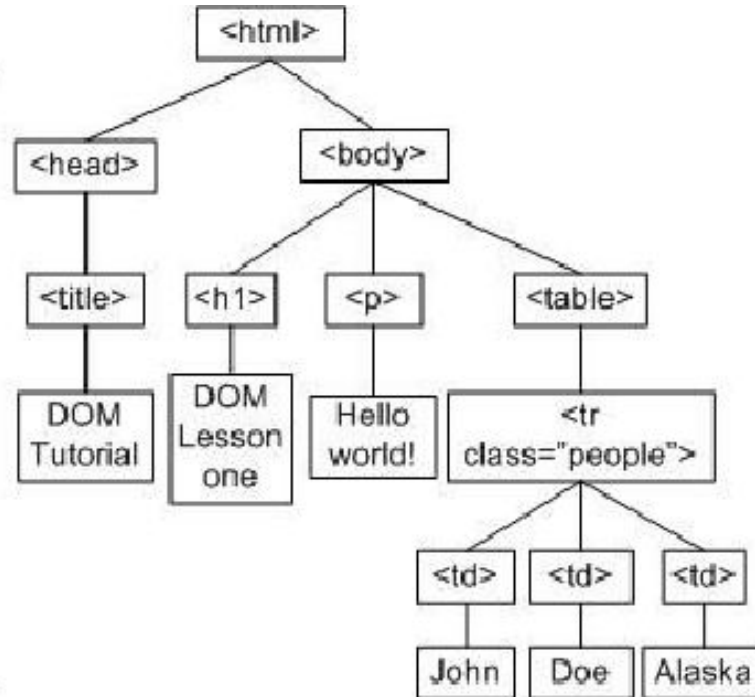




# DOM (Cont.)

```
<html>
 <head>
 <title>DOM Tutorial</title>
 </head>
 <body>
 <h1>DOM Lesson one</h1>
 <p>Hello world!</p>
 <table>
 <tr class="people">
 <td>John</td>
 <td>Doe</td>
 <td>Alaska</td>
 </tr>
 </table>
 </body>
</html>
```

test.html



DOM Tree

# DOM Manipulation

- We can manipulate the DOM dynamically from Javascript
- To be able to manipulate or change the DOM elements we must first select them
- Similar to how CSS selects elements and applies styles to them
- `document.querySelector()` is a very useful method to do this



# DOM Manipulation (Cont.)

- `document.querySelector()` is not part of Javascript
- Instead it is made available to Javascript by Web APIs which are available in all the standard browsers
- Since our Javascript code executed inside browsers, it has access to these Web APIs



# DOM Manipulation (Cont.)

- Handling Events
  - One of the important aspect of DOM Manipulation is the capability to handle events
  - We have seen this by calling a function on occurrence of events like click, mouse over etc.  
(onclick, onmouseover)
- Working with Classes



# Javascript Scope

- Scoping defines how the variables in our code are organized and accessed
- Scoping mechanism used in Javascript is called Lexical Scoping
- Meaning scope of variables is controlled by the placement of functions or blocks in the code
- Scope of a variable is the region of the code where it is accessible
- Scope is applicable to both variable and functions in Javascript



# Javascript Scope (Cont.)

- Javascript has the following scopes available
  - Global Scope
  - Function Scope
  - Block Scope
- Let us look at them in details



# Javascript Scope (Cont.)

- Global Scope
  - Variables and functions declared in the main area of the code
  - They are accessible throughout the code
- Function Scope
  - Variables and functions declared inside a function
  - Also known as local scope
  - Accessible only within that function and not anywhere else outside



# Javascript Scope (Cont.)

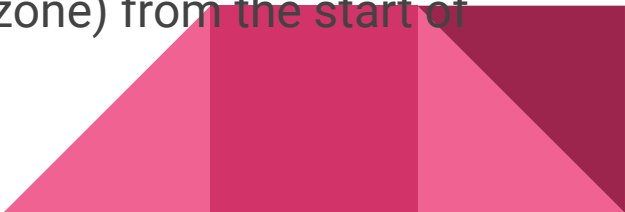
- Block Scope

- In Javascript any code written inside curly braces (“{ }”) is a block
- Variables created inside a block are local only to that block
- They are accessible only inside that block and not anywhere else outside
- The above condition is applicable only to the variables declared as “let” or “const” and not as “var”
- That is because block scope is a new concept in ES6 as well as the keywords **let** and **const**
- So we say var is function scoped





# Javascript Hoisting

- Hoisting is JavaScript's default behavior of moving declarations to the top
  - Javascript declarations are hoisted
  - A variable can be used before it has been declared
  - Variables defined with `let` and `const` are hoisted to the top of the block, but not initialized
  - Hence trying to use them gives an error
  - These variables are said to be in TDZ (temporal dead zone) from the start of the block until it is declared
- 

# Javascript Hoisting (Cont.)

- Function Declarations
  - Hoisted
  - Initial value will be function definition itself
- var variables
  - Hoisted
  - Initialized with ***undefined***



# Javascript Hoisting (Cont.)

- let and const variables
  - Technically hoisted but not practically
  - Not initialized and stay in TDZ until declared
- Function expressions and arrows
  - These are just like variables
  - So same rules apply as above
- Sounds pointless right
- Keep all declarations on top
- Avoid declaring variables or functions with ***var***



# this Variable

- The **this** variable is available only to function declarations and expressions
- It is not available for arrow functions
- Hence we cannot use arrow functions as object methods
- **this** keyword in case of arrow functions refers to the parent scope




# Primitives vs Objects

- The way primitives and objects are handled in Javascript is significantly different
- Javascript runtime which is responsible for executing the code is called Javascript Engine
- It consists of 2 primary components called as
  - Call Stack
  - Heap



# Primitives vs Objects (Cont.)

- Callstack
    - Responsible for maintaining the order in which the code is to be executed
    - Stores variables
  - Heap
    - Temporary memory to store object references
  - So primitives types are stored in stack as values
  - Object types are stored in heap as references
  - So when we copy an object into another just by assigning, it actually refers to same object in heap
- 

# Primitives vs Objects (Cont.)

- So proper way to copy objects is using
  - `Object.assign({}, oldObject)`
- As you can see, there is a new empty object being created and old object being copied into it
- Now both objects can be modified independent of each other



# Welcome to Modern Javascript





# Array Destructuring

- Refers to extracting elements of an array into separate variables
- Provides an easy way to extract elements of array without using index notation
- E.g. Original array → `const arr = [2, 3, 4];`
- Destructuring → `const [x, y, z] = arr;`
- We can now use `x`, `y`, `z` as we please
- Original array stays intact in the process



# Object Destructuring

- Refers to extracting elements of an object into separate variables
- Provides an easy way to extract elements of object
- E.g. Original object → `const obj = { mon: { open: 12, close: 23 } };`
- Destructuring → `const {mon} = obj;`
- `console.log(mon);`
- `{ open: 12, close: 23 }`



# Spread Operator

- Refers to extracting elements of an object into separate variables
- Provides an easy way to extract elements of object
- E.g. Original object → `const obj = { mon: { open: 12, close: 23 } };`
- Destructuring → `const {mon} = obj;`
- `console.log(mon);`
- `{ open: 12, close: 23 }`



# Sets

- Sets are data structures introduced in ES6
- Similar to Sets in other programming languages, these can contain a list or group of elements
- But the restriction is that sets contain only unique elements
- Sets are created using new Set() method and passing an iterable to it
- `let setOfChars = new Set('angular')`



# Sets (Cont.)

- Sets have the following methods to work with them
  - `add()`
  - `delete()`
  - `has()`
  - `values()`
  - `forEach()`
  - `clear()`
- Sets also have the **size** property to find the number of elements



# Sets (Cont.)

- We cannot extract elements of set like we do for arrays
- That is because sets are unordered and hence do not have index
- But we can loop over sets and extract elements
- We can use the for - of loop to loop over sets




# Maps

- Map is another data structure introduced in ES6
- This is similar to Objects but with key differences shown below
- A Map holds key-value pairs where the keys can be any datatype
- A Map remembers the original insertion order of the keys



# Dates

- By default Javascript uses Browser's time zone
  - Full text string form of date looks like
    - Thu Apr 07 2022 21:41:37 GMT+0530 (India Standard Time)
  - Date is created using the new keyword
  - Can use any of the following ways
    - `new Date()`
    - `new Date(year, month, day, hours, minutes, seconds, milliseconds)`
    - `new Date(milliseconds)`
    - `new Date(date string)`
- 



## Dates (cont.)

- Specifying a month higher than 11, will not result in an error but add the overflow to the next year
- Same holds good for all other values as well
- Internally stores dates as number of milliseconds since January 01, 1970



# Date Methods

Method	Description
setDate()	Set the day as a number (1-31)
setFullYear()	Set the year (optionally month and day)
setHours()	Set the hour (0-23)
setMilliseconds()	Set the milliseconds (0-999)
setMinutes()	Set the minutes (0-59)
setMonth()	Set the month (0-11)
setSeconds()	Set the seconds (0-59)
setTime()	Set the time (milliseconds since January 1, 1970)

# Date Methods (Cont.)

Method	Description
getFullYear()	Get the year as a four digit number (yyyy)
getMonth()	Get the month as a number (0-11)
getDate()	Get the day as a number (1-31)
getHours()	Get the hour (0-23)
getMinutes()	Get the minute (0-59)
getSeconds()	Get the second (0-59)
getMilliseconds()	Get the millisecond (0-999)
getTime()	Get the time (milliseconds since January 1, 1970)
getDay()	Get the weekday as a number (0-6)
Date.now()	Get the time. ECMAScript 5.

# More on Functions



# Functions - Default Parameters

- We can now create functions with default parameters
- When we do not pass a value for any parameter, it would be set to the default value
- This was done in ES5 using a concept called short circuiting
- In ES6 there is a more straightforward way to do it
- By just setting them in the function signature itself



# Functions - First Class Functions

- Just means that functions in Javascript are treated as objects or simply as values
- Write a function and check its type using ***typeof***
- Since they are just objects we can:
  - Store functions in variables or properties (arrow functions, object methods)
  - Pass a function to another function as an argument
  - Return functions from other functions



# Functions - Higher Order Functions

- A higher order function is a function that:
  - Receives another function as an argument
  - Returns a new function
  - Does both of the above
- Possible to write higher order functions because Javascript functions are first class functions
- Example of a higher order function is `addEventListener` we had seen previously



# Functions - Call, Apply, Bind

- We have seen that while working with objects and object methods, **this** keyword plays a significant role
- However the behavior of **this** changes depending on the way functions are used
- Inside `object.objectMethod()` this refers to the current object
- We can also copy this method for reuse as follows
  - `const copyMethod = object.objectMethod`



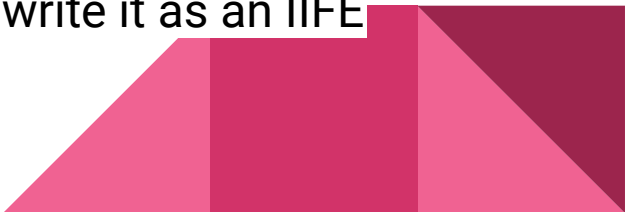


# Functions - Call, Apply, Bind

- The copied method will not have access to the object via **this**
- Also when we pass a function as a callback argument, we lose access to **this**
- To resolve this problem, Javascript provided 3 special methods
  - Call
  - Apply
  - Bind
- These methods help us to manually set the **this** reference and to borrow methods from one object in another object



# Functions - IIFE

- IIFE or Immediately Invoked Function Expression is a way to achieve abstraction in Javascript
  - Also to write functions that will be called just ones and need not be stored for later use
  - If we have any sensitive data to be protected from accidental modification or misuse we can limit its scope by wrapping it in an IIFE
  - If we need a function that is called just once, we can write it as an IIFE
- 

# Closures

- Closure is the combination of a function bundled together with references to its surrounding state
- It just means that Closure is a special function in Javascript which has access to its parent scope even after the parent function completes execution
- Sounds confusing right?.
- To appreciate the concept of Closure, we need to revisit concepts like functions execution context and variable scope chain



# Arrays Revisited

- As we said earlier, with each new version of Javascript, new ways of working with things get introduced
- Accordingly we now have a bunch of new array methods available
  - `array.at(index)` - to get an element at an index
  - `array.forEach(callbackFunction)`
    - `callbackFunction` accepts 3 arguments
      - `arrayItem`, `index` and `array object` itself
- Same works for Maps and Sets as well



# Arrays - Map, Filter and Reduce

- Similar to forEach, we have 3 more useful methods to work with arrays
- Map
  - Can be called on an array
  - Accepts a callback function which is applied to each element of the array
  - Map returns a new array where each element is the result applying the callback function on each element of the original array



# Arrays - Map, Filter and Reduce (Cont.)

- Filter
  - Can be called on an array
  - Accepts a callback function with a condition which is applied to each element of the array
  - Filters returns a new array which is a subset of the original array including only those elements that satisfy the condition



# Arrays - Map, Filter and Reduce (Cont.)

- Reduce
  - Can be called on an array
  - Accepts a callback function with a cumulative operation which is applied to each element of the array
  - Reduce returns a single value which is the result of applying the callback function to each element of the original array and reducing it to a single value



# Arrays - Find

- Find
  - Can be called on an array
  - Accepts a callback function with a condition which is applied to each element of the array
  - Find method returns the first item of the original array that satisfies the condition mentioned in the callback function





# Arrays - Find Index

- Find Index
  - Can be called on an array
  - Accepts a callback function with a condition which is applied to each element of the array
  - `findIndex` method returns the index of the first item of the original array that satisfies the condition mentioned in the callback function



# Arrays - Includes

- Includes
  - Can be called on an array
  - Accepts an element which we want to check whether it is included in the array
  - Returns true or false



# Arrays - Some and Every

- Some
  - Similar to Includes, instead of checking equality, we can specify a condition to be checked which returns true or false
  - Somewhat confusing name as it actually behaves like if any element satisfies the specified condition then return true
- Every
  - Similar to some
  - Accepts a condition to be checked
  - Returns true only if all elements satisfy the condition



# Arrays - Flat and Flat Map

- Flat
  - Used to flatten a nested array and put elements directly in outer array
  - Works with only first level of nesting
- Flat Map
  - Similar to flat used in combination with map



# Object Oriented Programming (OOP)




# What is OOP?

- A programming paradigm or style of programming based on objects
- Just means that we use Javascript objects to model real life scenarios and solve real life problems
- We already know that object contain data (attributes or properties) and code (methods) to manipulate that data together
- So data and behavior is bundled together into a single block called object



# What is OOP? (Cont.)

- So in other words objects are self contained pieces or blocks of code to store and manipulate data
  - Objects are building blocks of the apps we are going to build
  - Different objects that are used to create apps, need to interact with each other
  - These interactions happen through public interfaces called APIs
  - Via the APIs the methods or the behavior of one object is accessible to other objects thereby enabling interaction between different objects
- 

# Why is OOP needed?

- Easy to organize code
- Easy to change and maintain
- Facilitates code reuse





# Concept of Class

- Objects are based on classes
- Class is a blueprint for creating objects
- This is similar to a plan or blueprint diagram of a house based on which multiple houses can be built
- From a class we can create multiple objects
- These objects can have different data but behavior will be the same



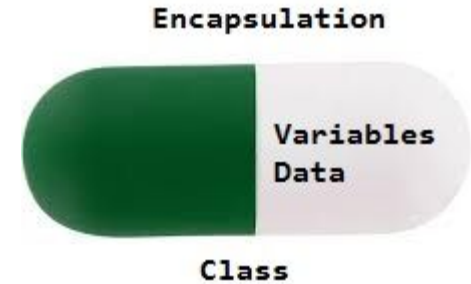
# 4 Fundamental Principles

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism



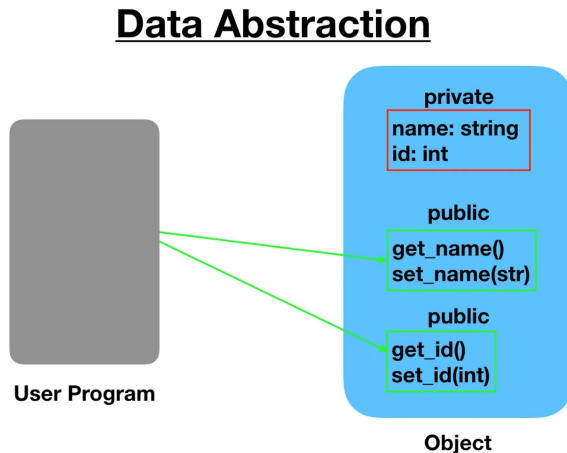
# Encapsulation

- Bundling the data and the behavior of objects together
- In other words the attributes and the methods or code to access the attributes is bundled together in the object itself
- This helps us achieve the next principle called Abstraction



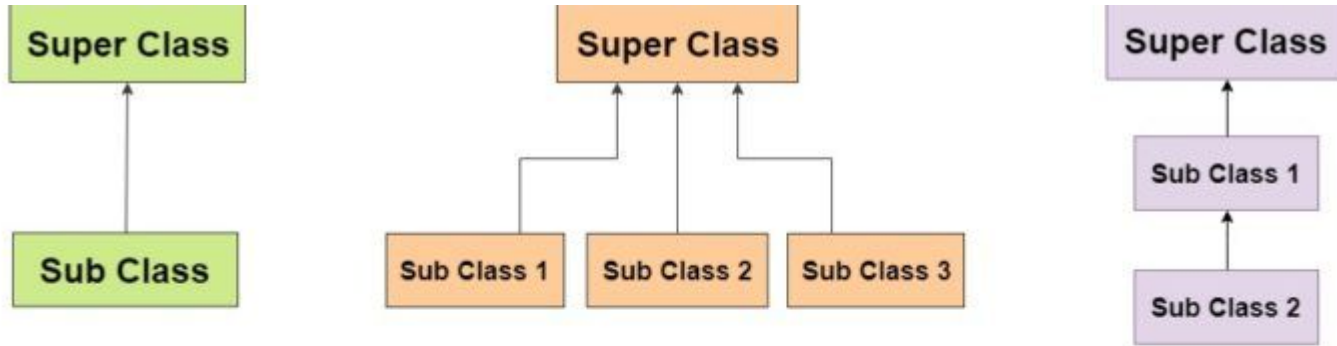
# Abstraction

- Hiding the internal details from the outside world
- Helps avoid accidental modification
- So sensitive data is made private with restricted access
- Access is provided only via the methods which are publicly accessible as APIs
- Also helps to change internal code independent of external world as long as public interface or API does not change



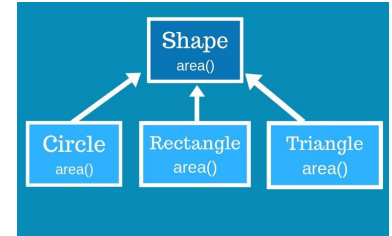
# Inheritance

- Parent - Child relationship between classes
- Child class inherits all the properties and methods of parent class
- On top of the inherited things, child class can add its own properties and methods



# Polymorphism

- Something that takes multiple forms
- A method inherited from the parent class can be overridden by the child class

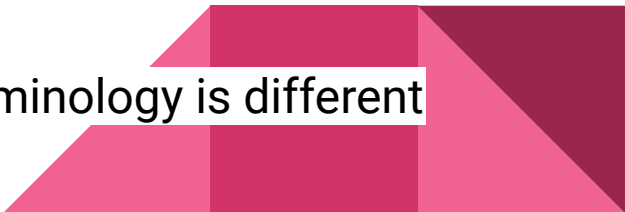


# OOP in Javascript

- Javascript supports OOP through something called as Prototype
- Each object has a prototype associated with it
- So we say each object has a prototype
- Prototype itself is an object and has some properties and methods associated with it



# OOP in Javascript (Cont.)

- Each object of that prototype has access to these properties and methods
  - This is called Prototypal Inheritance
  - This is different from the class inheritance that we talked about earlier
  - Class inheritance is a subclass inheriting properties and behavior from a parent class
  - Prototypal inheritance is object instances inheriting the properties and behavior from the prototype
  - Concept is same if you think about it, but just the terminology is different
- 

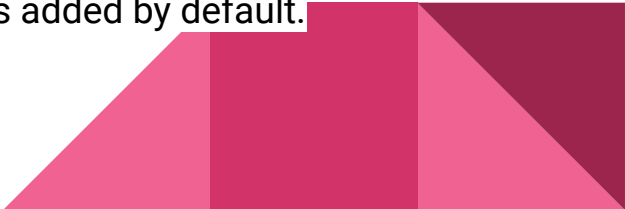


# OOP in Javascript (Cont.)

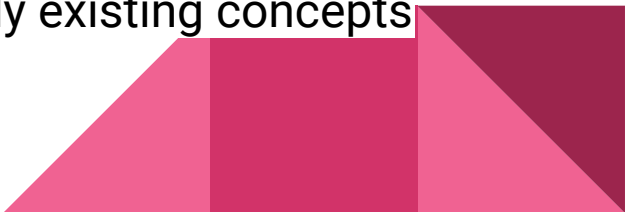
- OOP in Javascript is all about being able to create and manipulate objects programmatically
- Objects can be created using one of the following ways
  - Constructor functions
  - ES6 classes
  - `Object.create()` method



# Constructor Functions

- Using this technique objects are created from a function using the ***new*** keyword
  - When we create an object using the new keyword:
    - A new empty object {} is created
    - Sets the objects invisible prototype property to the constructor function's visible and accessible prototype property
    - Binds the object with ***this*** keyword
    - Returns the object. If return statement is missing, ***return this*** is added by default.
- 


# ES6 Classes

- Since OOP using the concept of Prototype is somewhat different from the way it is done in other languages, ES6 introduced the concept of classes
  - ES6 classes just provide the standard syntax similar to other languages, but underneath they still use Constructor methods to achieve OOP in Javascript
  - Inheritance is also provided by ES6 using the same old concept of prototypal inheritance that was already in use in Javascript
  - So it is just a layer of abstraction on top of the already existing concepts
- 

# Object.create()

- This is the easiest and most straightforward way to create an object and link it to an underlying prototype object
- But it is not used as much as the other two ways

Irrespective of which of the above methods are used to create objects, the primary objective is to achieve the 4 principles of OOP namely:

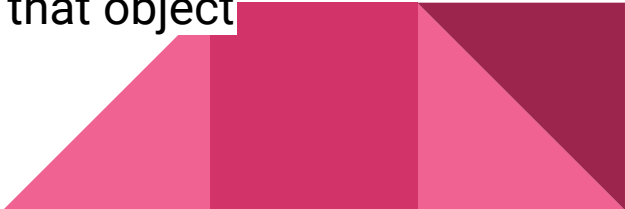
- Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism
- 

# Creating Objects with Constructor

- Constructor function is just like any other normal function in Javascript
- It is just called with a new keyword while creating a new object
- So all the magic is done here by the new keyword
- The constructor function can contain both properties and methods
- But adding methods in the constructor function is not recommended as
  - It gets duplicated to each object created
  - Since the methods are common and only properties change for different objects we should not add methods to the constructor
- So the solution is to add methods to the prototype



# Prototype in Javascript

- Prototype can be thought of as a model associated with functions and objects in Javascript
  - They can be compared in some way to the classes in other programming languages
  - Javascript is a dynamic language
  - We can add additional properties/ methods to an object at any time
  - However the new property/ method is limited only to that object
- 

# Prototype in Javascript (Cont.)

- If we want the new property/ method to be available for other objects also which will be created from the constructor function, then we need to add it to the prototype
- In a sense this is similar to extending the base case and adding the new property to the child class
- This way any object created from the child class will have the new property/ method



# Prototypal Inheritance

- We saw that objects have properties associated with them directly
- We can have methods and additional properties associated with the object prototype
- When we access a property or a method on an object, Javascript will first search the object properties or methods
- If it find there it is used, If not, the objects prototype is searched
- This goes on and finally it is executed or error is shown





# Prototype Chain

- An object created using a constructor function has a prototype of that function
- Prototype itself is an object, so that is also created using some constructor and has a prototype associated with it
- This goes on hierarchically, till we reach the super parent Object
- The Object is final element in this chain and its prototype is set to null
- This is called Prototype Chain
- This pretty similar to scope chain we saw for variables



# ES6 Classes

- As discussed earlier, Javascript classes are just an abstraction on top of the things that we discussed so far
- This is done to make things easier for people coming to Javascript from other programming languages
- In the background Javascript continues to use Prototypal Inheritance and Prototype chains etc.

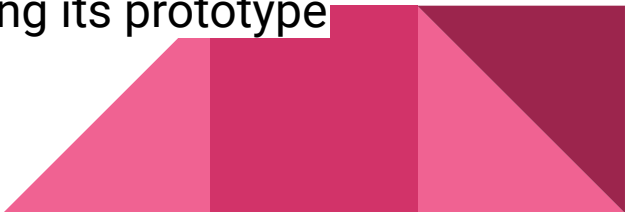


# ES6 Classes (Cont.)


- Classes are not hoisted
- Classes are also first class citizens, meaning we can pass classes and return classes from functions
- This is simply because behind the scenes classes are just like functions
- Classes are always executed in strict mode




# ES6 Classes (Cont.)

- Javascript class is created using class keyword followed by the class name
  - As a convention we write class name with first letter in uppercase
  - Class definition contains a constructor function which is used to initialize the instance variables
  - Class methods are similar to object methods and they follow the constructor method
  - We can add additional methods to the class later using its prototype
- 


# Getters and Setters

- All objects in javascript have special properties called getter and setter properties
  - These are termed as accessor properties while the others are called data properties
  - These special properties are used to get and set data properties of the object
  - Same applies to classes as well
  - We can define getters and setters on classes also which work exactly same as object literal getters and setters
- 


# Static Methods

- Instance methods are associated with each object or instance
  - Static methods are associated with either the Constructor or the Class
  - In case Constructor function we saw that, if we want the new methods to be available to all objects, we need to add them to the prototype
  - If we add a method directly to the Constructor instead of the prototype, then it is available only for the constructor and not to the objects created using this constructor
  - These methods are called static
- 

# Static Methods (Cont.)

- Same way while creating objects using ES6 classes, we can create static methods using the keyword ***static***
  - Methods created as static are available only to the class and not to the individual objects created using the class
  - Typically if we have a method that is common for all the objects, then we make it as static
  - E.g. `Array.from()` method which is used to create an array from an array like object
- 

# Data Privacy

- So far we have seen that anyone can access or modify any of the instance attributes
  - This can lead to unintended use and modification of sensitive data
  - So we need a way to prevent such unintended modification
  - That is where the concept of private fields (properties) and private methods comes in
  - We can define certain properties or attributes as private there by restricting access to them outside the class
- 

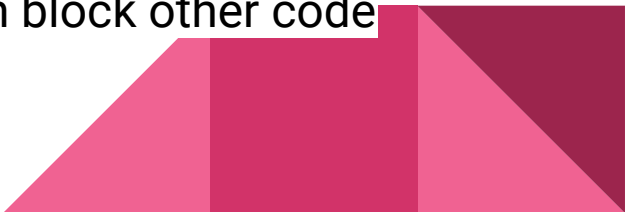


# Data Privacy (Cont.)

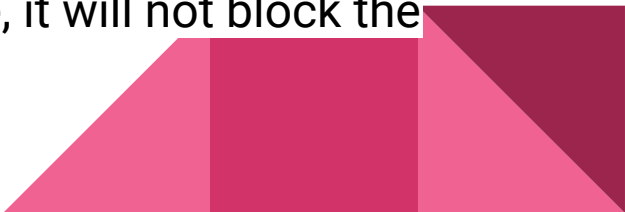
- We can make any attributes or methods as private by prefixing them with #
- When we make some attributes as private as a convention we call them as private fields
- When we try to access private fields outside the class, an appropriate error is shown



# Async Javascript

- The code execution we have see so far can be termed as synchronous
  - It just means that execution will move to next line only after finishing the current line
  - This is fine, if the previous line finishes execution in a timely fashion
  - E.g. A browser alert makes further code to wait till it is dismissed by clicking the Ok button
  - So long running operations in synchronous execution block other code
- 

# Async Javascript

- In contrast, asynchronous execution is non-blocking
  - The statement that would have blocked, is deferred to make the code non blocking
  - After the blocking code finishes execution, it will return and resume execution as part of the main code
  - E.g. Setting src attribute of img tag is asynchronous
  - In case the image to be loaded is large and take time, it will not block the remaining execution
- 

# AJAX Calls

- We can implement asynchronous code in Javascript using AJAX
- Stands for Asynchronous Javascript And XML
- Used to communicate with remote web servers asynchronously
- Used when we have to fetch data by calling an API and render it on web page
- These APIs are also known as “Web APIs”



# APIs

- Now a days APIs are everywhere
  - Weather data
  - Stocks data
  - Flights data
  - Maps data
  - Temperature data
  - Currency data
- Above are some examples of data we can fetch from web or online APIs



# XMLHttpRequest

- Previously we were using XMLHttpRequest to get data from an API via AJAX call
- Since this is an async operation, we had to associate a callback function on a load event
- Meaning, once the response comes back, a load event gets fired
- This in turn calls the callback function in which we handle the AJAX call response

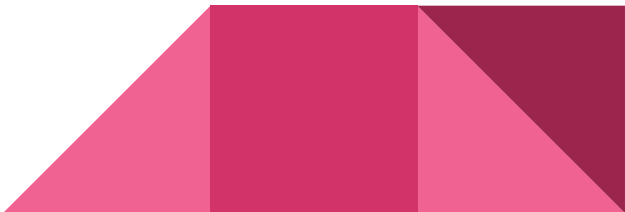


# Fetch

- Now we can simply use `fetch(url)` method with the URL
- This is also an async call
- So it immediately returns something called as a Promise object which is initially in pending state
- Once the response comes back, the status of the Promise object status changes to



# Promises

- In case of async code, we have 2 pieces
  - Producing code - Async code that can take some time
  - Consuming code - code that waits for producing code to provide result
  - Promise is an object that can be used to wrap both producing and consuming code to handle async situation
  - Promise object also accepts a callback function in which we can decide how to deal with async result
- 



# Promises (Cont.)

- The callback function we pass to the promise accepts 2 methods
  - One to process the result if async code is successful
  - One to show error if async code fails
- The producing code calls one of the above methods based on whether the result is success or failure
- Accordingly Promise object will be in one of the below states
  - Pending
  - Fulfilled
  - Rejected



# Promise

- An object that acts like a placeholder for future response of fetch
- Like a container waiting to be filled
- Will be filled once fetch gets the response back
- No more waiting for events and callbacks



# Promise (Cont.)

- Promise life cycle
  - Pending
  - Settled
    - Fulfilled
      - Success. We get the response back.
    - Rejected
      - Failure. An error occurred.
- Promise is settled only once
- Meaning, once it is settled it will remain in that state



# Async/ Await

- Async/ Await statements are used to simplify working with promises while dealing with asynchronous Javascript
- A function defined using async keyword returns a promise
- Inside this function we use await keyword to wait for the promise to complete and return either resolve or reject
- Accordingly we can deal with resolve or reject in next statements

