# JavaScript Introduction:

**JavaScript (JS)** is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat.

## Interpreted versus compiled languages:

In interpreted languages, the code is run from top to bottom and the result of running the code is immediately returned.

Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer. For example, C/C++ are compiled into machine code that is then run by the computer.

## Server-side versus client-side:

Client-side code is code that is run on the user's computer. Server-side code on the other hand is run on the server.

## Variables:

### let:

Variables are basically names for values (such as numbers, or strings of text). You create a variable with the keyword **let** followed by a name for your variable.

### Const:

Constants are also used to name values, but unlike variables, you can't change the value once set.

## Variable Types

### Functions:

Functions are reusable blocks of code that you can write once and run again and again, saving the need to keep repeating code all the time.

### Operators:

JavaScript operators allow us to perform tests, do math, join strings together, and other such things.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| + | Addition | 6 + 9 |
| - | Subtraction | 20 - 15 |
| \* | Multiplication | 3 \* 7 |
| / | Division | 10 / 5 |

## Problem 1:

Write program to print the kth digit from last. E.g. input 23617 and k=4 output 3.

## Problem 2:

Write program to find sum of all digits. Input 23617 output 2+3+6+1+8=20

(Hint: convert string to array and then use reduce function)

## Problem 3:

Write program to find sum of even digits. Input 23617 output 2+6=8.

(Hint: Convert string to array and use map function)

## Problem 4:

Write a JavaScript that calculates the squares and cubes of the numbers from 0 to 10.

## Problem 5:

Develop and demonstrate JavaScript script that uses functions for the following problems:

1. Parameter: A string
2. Output: The position in the string of the left-most vowel
3. Parameter: A number
4. Output: The number with its digits in the reverse order

## Problem 6:

Write a JavaScript program where the program takes a random integer between 1 to 10, the user is then prompted to input a guess number. If the user input matches with guess number, the program will display a message "You WIN" otherwise display a message "Not matched".

## Problem 7:

Write a JavaScript program to check whether 10 appears in first or last position of a given array of integers. The array length must be greater or equal to 2.

## Problem 8:

Write a JavaScript function that returns a passed string with letters in alphabetical order.

**Example string: 'comsats'**

**Expected Output: 'acmost'**

# Lab 2: ES6 Concepts

# Let key word

In ES5, when you declare a variable using the **var** keyword, the scope of the variable is either global or local. If you declare a variable outside of a function, the scope of the variable is global. When you declare a variable inside a function, the scope of the variable is local.

ES6 provides a new way of declaring a variable by using the **let** keyword. The let keyword is similar to the **var** keyword, except that these variables are blocked-scope. For example:

let variable\_name;

In JavaScript, blocks are denoted by curly braces {} , for example, the if else, for, do while, while, try catch and so on:

|  |
| --- |
| if(condition) {  // inside a block  } |

## Exercise:

|  |
| --- |
| let x = 10;  if (x == 10) {  let x = 20;  console.log(x); // 20: reference x inside the block  }  console.log(x); // 10: reference at the begining of the script |

How the script works: First, declare a variable x and initialize its value to 10.

Second, declare a new variable with the same name x inside the if block but with an initial value of 20.

Third, output the value of the variable x inside and after the if block.

Because the let keyword declares a block-scoped variable, the x variable inside the if block is a new variable and it shadows the x variable declared at the top of the script. Therefore, the value of x in the console is 20.

When the JavaScript engine completes executing the if block, the x variable inside the if block is out of scope. Therefore, the value of the x variable that following the if block is 10.

## JavaScript let and global object

When you declare a global variable using the var keyword, you add that variable to the property list of the global object. In the case of the web browser, the global object is the window. For example:

|  |  |
| --- | --- |
| var a = 10;  console.log(window.a); // 10 | let b = 20;  console.log(window.b); // undefined |

## Redeclaration

The var keyword allows you to redeclare a variable without any issue:

|  |
| --- |
| var counter = 0;  var counter;  console.log(counter); // 0 |

However, redeclaring a variable using the let keyword will result in an error:

|  |
| --- |
| let counter = 0;  let counter;  console.log(counter);  //Uncaught SyntaxError: Identifier 'counter' has already been declared |

## JavaScript let variables and hoisting

Let’s examine the following example:

|  |
| --- |
| {  console.log(counter); //  let counter = 10;  }  //Uncaught ReferenceError: Cannot access 'counter' before initialization |

In this example, accessing the counter variable before declaring it causes a ReferenceError. You may think that a variable declaration using the let keyword does not hoist, but it does.

In fact, the JavaScript engine will hoist a variable declared by the let keyword to the top of the block. However, the JavaScript engine does not initialize the variable. Therefore, when you reference an uninitialized variable, you’ll get a ReferenceError.

## const keyword

ES6 provides a new way of declaring a constant by using the const keyword. The const keyword creates a read-only reference to a value.

|  |
| --- |
| const CONSTANT\_NAME = value; |

## JavaScript arrow functions

ES6 arrow functions provides an alternative way to write a shorter syntax compared to the function expression.

The following example defines a function expression that returns the sum of two numbers:

|  |
| --- |
| let add = function (x, y) {  return x + y;  };  console.log(add(10, 20)); // 30 |

The following example is equivalent to the above add() function expression but use an arrow function instead:

|  |
| --- |
| let add = (x, y) => x + y;  console.log(add(10, 20)); // 30; |

In this example, the arrow function has one expression x + y so it returns the result of the expression.

However, if you use the block syntax, you need to specify the return keyword:

|  |
| --- |
| let add = (x, y) => { return x + y; }; |

### Arrow functions with multiple parameters

If an arrow function has two or more parameters, you use the following syntax:

|  |
| --- |
| (p1, p2, ..., pn) => expression; |

For example, to sort an array of numbers in the descending order, you use the sort() method of the array object as follows:

|  |
| --- |
| let numbers = [4,2,6];  numbers.sort(function(a,b){  return b - a;  });  console.log(numbers); // [6,4,2] |

### arrow functions with a single parameter

If an arrow function takes a single parameter, you use the following syntax:

|  |
| --- |
| (p1) => { statements } |

Note that you can omit the parentheses as follows:

|  |
| --- |
| p => { statements } |

The following example uses an arrow function as an argument of the map() method that transforms an array of strings into an array of the string’s lengths.

|  |
| --- |
| let names = ['John', 'Mac', 'Peter'];  let lengths = names.map(name => name.length);  console.log(lengths); |

### arrow functions with no parameter

If the arrow function has no parameter, you need to use parentheses, like this:

|  |
| --- |
| () => { statements } |

For example:

|  |
| --- |
| let logDoc = () => console.log(window.document);  logDoc(); |

### arrow functions and object literal

Consider the following example:

|  |
| --- |
| let setColor = function (color) {  return {value: color}  };  let backgroundColor = setColor('Red');  console.log(backgroundColor.value); // "Red" |

The setColor() function expression returns an object that has the value property set to the color argument.

If you use the following syntax to return an object literal from an arrow function, you will get an error.

|  |
| --- |
| p => {object:literal} |

For example, the following code causes an error.

|  |
| --- |
| let setColor = color => {value: color }; |

Since both block and object literal use curly brackets, the JavasScript engine cannot distinguish between a block and an object.

To fix this, you need to wrap the object literal in parentheses as follows:

|  |
| --- |
| let setColor = color => ({value: color }); |

## Classes

A JavaScript class is a blueprint for creating objects. A class encapsulates data and functions that manipulate data.

Unlike other programming languages such as Java and C#, JavaScript classes are syntactic sugar over the prototypal inheritance. In other words, ES6 classes are just special functions.

### Classes prior to ES6 revisited

Prior to ES6, JavaScript had no concepts of classes. To mimic a class, you often use the constructor/prototype pattern as shown in the following example:

|  |
| --- |
| function Person(name) {  this.name = name;  }  Person.prototype.getName = function () {  return this.name;  };  var john = new Person("John Doe");  console.log(john.getName()); |

**How it works.**

First, create the Person as a constructor function that has a property name called name. The getName() function is assigned to the prototype so that it can be shared by all instances of the Person type.

Then, create a new instance of the Person type using the new operator. The john object, hence, is an instance of the Person and Object through prototypal inheritance.

### ES6 class declaration

ES6 introduced a new syntax for declaring a class as shown in this example:

|  |
| --- |
| class Person {  constructor(name) {  this.name = name;  }  getName() {  return this.name;  }  } |

This Person class behaves like the Person type in the previous example. However, instead of using a constructor/prototype pattern, it uses the class keyword.

In the Person class, the constructor() is where you can initialize the properties of an instance. JavaScript automatically calls the constructor() method when you instantiate an object of the class.

The following creates a new Person object, which will automatically call the constructor() of the Person class:

|  |
| --- |
| let john = new Person("John Doe"); |

### Class vs. Custom type

Despite the similarities between a class and a custom type defined via a constructor function, there are some important differences.

First, class declarations are not hoisted like function declarations.

For example, if you place the following code above the Person class declaration section, you will get a ReferenceError.

|  |
| --- |
| let john = new Person("John Doe"); |

Second, all the code inside a class automatically executes in the strict mode. And you cannot change this behavior.

Third, class methods are non-enumerable. If you use a constructor/prototype pattern, you have to use the Object.defineProperty() method to make a property non-enumerable.

Finally, calling the class constructor without the new operator will result in an error as shown in the following example.

|  |
| --- |
| let john = Person("John Doe"); |

## Exercises:

### Problem 1:

Write a JavaScript program to list the properties of a JavaScript object. Go to the editor

Sample object:

|  |
| --- |
| var student = {  name : "David Rayy",  sclass : "VI",  rollno : 12 }; |

### Problem 2:

Write a JavaScript program to delete the rollno property from the following object. Also print the object before or after deleting the property. Go to the editor

Sample object:

|  |
| --- |
| var student = {  name : "David Rayy",  sclass : "VI",  rollno : 12 }; |

### Problem 3:

Write a JavaScript program to display the reading status (i.e. display book name, author name and reading status) of the following books. Go to the editor

|  |
| --- |
| var library = [  {  author: 'Bill Gates',  title: 'The Road Ahead',  readingStatus: true  },  {  author: 'Steve Jobs',  title: 'Walter Isaacson',  readingStatus: true  },  {  author: 'Suzanne Collins',  title: 'Mockingjay: The Final Book of The Hunger Games',  readingStatus: false  }]; |

Problem 3:

1. Write a function called sandwich calculator. This should accept one value: bread
2. The function should return the total number of possible sandwiches based on the amount of breads available. I need 2 breads to make one sandwich, so if there are 10 breads, it should return 5. Test your function with console.log.
3. Extend your function so it accepts two values, bread and cheese.
4. It takes two sbreads and one cheese to make a sandwich. The function should return the total number of possible sandwiches, so if there are breads, but only 1 cheese, it should return 1.

# LAB 3

## Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

Imagine a function, createAudioFileAsync(), which asynchronously generates a sound file given a configuration record and two callback functions, one called if the audio file is successfully created, and the other called if an error occurs.

Here's some code that uses createAudioFileAsync():

|  |
| --- |
| function successCallback(result) {  console.log("Audio file ready at URL: " + result);  }  function failureCallback(error) {  console.error("Error generating audio file: " + error);  }  createAudioFileAsync(audioSettings, successCallback, failureCallback); |

If createAudioFileAsync() were rewritten to return a promise, you would attach your callbacks to it instead:

|  |
| --- |
| createAudioFileAsync(audioSettings).then(successCallback, failureCallback); |