JavaScript

Agenda

- Introduction
- Variables
- Datatypes
- Type conversion
- Alert
- Prompt
- Confirm
- Basic operations and Maths
- Comparisons
- Conditional Statements
- Logical operators

- Loops
- Switch
- Function
- Arrow function
- Objects
- Function Closures
- Classes
- Prototyping
- promises
- Error handling
- Modules

Why and Where??

Think about it before moving to next slide……

Introduction to JavaScript

- JavaScript was initially created to “make web pages alive”.
- The programs in this language are called scripts. They can be written right in a web page’s HTML and run automatically as the page loads.
- Scripts are provided and executed as plain text. They don’t need special preparation or compilation to run.
- When JavaScript was created, it initially had another name: “LiveScript”. But Java was very popular at that time, so it was decided that positioning a new language as a “younger brother” of Java would help.
- But as it evolved, JavaScript became a fully independent language with its own specification called ECMAScript, and now it has no relation to Java at all.

- Today, JavaScript can execute not only in the browser, but also on the server, or actually on any device that has a special program called the JavaScript engine.

- The browser has an embedded engine sometimes called a “JavaScript virtual machine”.

Engine

- Engines are complicated. But the basics are easy.
  1. The engine (embedded if it's a browser) reads ("parses") the script.
  2. Then it converts ("compiles") the script to the machine language.
  3. And then the machine code runs, pretty fast.

- The engine applies optimizations at each step of the process. It even watches the compiled script as it runs, analyzes the data that flows through it, and further optimizes the machine code based on that knowledge.

What can in-browser JavaScript do?

- It does not provide low-level access to memory or CPU, because it was initially created for browsers which do not require it.
- Add new HTML to the page, change the existing content, modify styles.
- React to user actions, run on mouse clicks, pointer movements, key presses.
- Send requests over the network to remote servers, download and upload files (so-called AJAX and COMET technologies).
- Get and set cookies, ask questions to the visitor, show messages.
- Remember the data on the client-side (“local storage”).

What CAN’T in-browser JavaScript do?

- JavaScript does not allow the reading or writing of files on client machines. That's a good thing, because you certainly don't want a Web page to be able to read files off of your hard disk, or be able to write viruses onto your disk, or be able to manipulate the files on your computer. The only exception is that JavaScript can write to the browser's cookie file, and even then there are limitations.

- JavaScript does not allow the writing of files on server machines. There are a number of ways in which this would be handy (such as storing page hit counts or filled-out form data), but JavaScript isn't allowed to do that. Instead, you'll need to have a program on your server to handle and store this data. This can be, for example, a CGI written in a language such as Perl or PHP, or a Java program.

- JavaScript cannot close a window that it hasn't opened. This is to avoid a situation where a site takes over your browser, closing windows from any other sites.
- JavaScript cannot read information from an opened Web page that came from another server. In other words, a Web page can't read any information from other open windows and find out what else a surfer visiting the site is up to

Code Structure

- Statements
- Semicolon
- Comments

Statements

- JavaScript statements are composed of:
  - Values,
  - Operators,
  - Expressions,
  - Keywords,
  - Comments

Keywords

Keyword	Description
break	Terminates a switch or a loop
continue	Jumps out of a loop and starts at the top
debugger	Stops the execution of JavaScript, and calls (if available) the debugging function
do ... while	Executes a block of statements, and repeats the block, while a condition is true
for	Marks a block of statements to be executed, as long as a condition is true
function	Declares a function
if ... else	Marks a block of statements to be executed, depending on a condition
return	Exits a function
switch	Marks a block of statements to be executed, depending on different cases
try ... catch	Implements error handling to a block of statements
var	Declares a variable

Variables

- JavaScript variables are containers for storing data values.
- A variable is a “named storage” for data. We can use variables to store goodies, visitors, and other data.
- A JavaScript variable is simply a name of storage location. There are two types of variables in JavaScript : local variable and global variable.
- Before you use a variable in a JavaScript program, you must declare it.
- To create a variable in JavaScript, use the let and var keyword.

- There are some rules while declaring a JavaScript variable (also known as identifiers):
    1. Name must start with a letter (a to z or A to Z), underscore(_), or dollar($) sign.
    2. After first letter we can use digits (0 to 9), for example value1.
    3. JavaScript variables are case sensitive, for example x and X are different variables.
- Local and global Variables:
    - Var
    - Let
    - const

- **Global Variables** − A global variable has global scope which means it can be defined anywhere in your JavaScript code.
- **Local Variables** − A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

Using Var

- Using var we can declare a variable.
- In JavaScript, variables are initialized with the value of undefined when they are created. What that means is if we try to log the declaration variable, we'll get undefined.
- Example:-

```
var x = 5;
var y = 6;
var z = x + y;
```

- Before the advent of ES6, var declarations ruled. There are issues associated with variables declared with var, though. That is why it was necessary for new ways to declare variables to emerge.

Scope of Var

- Scope essentially means where these variables are available for use. var declarations are globally scoped or function/locally scoped.

- The scope is global when a var variable is declared outside a function. This means that any variable that is declared with var outside a function block is available for use in the whole window.

- var is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

- var variables can be re-declared and updated
- Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution
- Problem with Var:-

```
var greeter = “First";
 var times = 4;
 if (times > 3) {
     var greeter = “Last";
 }
 console.log(greeter)
```

Let

- let is now preferred for variable declaration. It's no surprise as it comes as an improvement to var declarations. It also solves the problem with var that we just covered.
- Let is Block Scoped
- A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block.
- A variable declared in a block with let is only available for use within that block.
- Let can be updated but not re-declared.

- Just like var, let declarations are hoisted to the top.
- Unlike var which is initialized as undefined, the let keyword is not initialized.
- if you try to use a let variable before declaration, you'll get a Reference Error.

Const

- Variables declared with the const maintain constant values. const declarations share some similarities with let declarations.
- const declarations are block scoped
- const cannot be updated or re-declared
- Every const declaration, therefore, must be initialized at the time of declaration.
- This behavior is somehow different when it comes to objects declared with const.
- While a const object cannot be updated, the properties of this objects can be updated.
- Just like let, const declarations are hoisted to the top but are not initialized.

KEYWORD	SCOPE	CAN BE REDECLARED	CAN BE REASSIGNED	HOISTING BEHAVIOR
var	function & global	yes	yes	initialized with undefined
let	block	no	yes	unintialized
const	block	no	no	unintialized

Data Type

- A value in JavaScript is always of a certain type. For example, a string or a number.
- JavaScript variables can hold many **data types**: numbers, strings, objects and more
- JavaScript provides different **data types** to hold different types of values. There are two types of data types in JavaScript.
    1. Primitive data type
    2. Non-primitive (reference) data type
- JavaScript is a **dynamic type language**, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine.

JavaScript primitive data types

Data Type	Description
String	represents sequence of characters e.g. "hello"
Number	represents numeric values e.g. 100
Boolean	represents boolean value either false or true
Undefined	represents undefined value
Null	represents null i.e. no value at all

JavaScript non-primitive data types

Data Type	Description
Object	represents instance through which we can access members
Array	represents group of similar values
RegExp	represents regular expression

Type conversion

- The process of converting one data type to another data type is called type conversion. There are two types of type conversion in JavaScript.
  - Implicit Conversion
  - Explicit Conversion

Implicit Conversion

- JavaScript is loosely typed language and most of the time operators automatically convert a value to the right type but there are also cases when we need to explicitly do type conversions.

- While JavaScript provides numerous ways to convert data from one type to another but there are two most common data conversions :
  - Converting Values to String
  - Converting Values to Numbers

- In certain situations, JavaScript automatically converts one data type to another (to the right type). This is known as implicit conversion.

- **Implicit Conversion to String**
- **Implicit Conversion to Number**
- **Non-numeric String Results to NaN**
- **Implicit Boolean Conversion to Number**
- **null Conversion to Number**
- **undefined used with number, boolean or null**

- To convert numeric strings and boolean values to numbers, you can use Number().
- In JavaScript, empty strings and null values return 0
- If a string is an invalid number, the result will be NaN.
- You can also generate numbers from strings using parseInt(), parseFloat(), unary operator + and Math.floor().
- To convert other data types to strings, you can use either String() or toString().
- String() takes null and undefined and converts them to string.
- To convert other data types to a boolean, you can use Boolean().
- In JavaScript, undefined, null, 0, NaN, " converts to false.

Explicit Conversion

- You can also convert one data type to another as per your needs. The type conversion that you do manually is known as explicit type conversion.
- In JavaScript, explicit type conversions are done using built-in methods.
- To convert numeric strings and boolean values to numbers, you can use Number().
- To convert other data types to strings, you can use either String() or toString().
- To convert other data types to a boolean, you can use Boolean().

- Convert to Number Explicitly: Number(),ParseInt,ParseFloat,Math.floor,+
- Convert to String Explicitly
- Convert to Boolean Explicitly

➢There are 8 basic data types in JavaScript.

- number for numbers of any kind: integer or floating-point, integers are limited by ±(253-1).
- bigint is for integer numbers of arbitrary length.
- string for strings. A string may have zero or more characters, there’s no separate single-character type.
- boolean for true/false.
- null for unknown values – a standalone type that has a single value null.
- undefined for unassigned values – a standalone type that has a single value undefined.
- object for more complex data structures.
- symbol for unique identifiers.

- You can also convert one data type to another as per your needs. The type conversion that you do manually is known as explicit type conversion.
- In JavaScript, explicit type conversions are done using built-in methods.
- Here are some common methods of explicit conversions.
  - Convert to Number Explicitly
  - Convert to String Explicitly
  - Convert to Boolean Explicitly

Interaction

alert, prompt, confirm

JavaScript Input Output

- JavaScript can "display" data in different ways:
  - Writing into an HTML element, using innerHTML.
  - Writing into the HTML output using document.write().
  - Writing into an alert box, using window.alert().
  - Writing into the browser console, using console.log().

- **Readline Module** in Node.js allows the reading of input stream line by line.

- Using alert()
- Using console.log()
- Prompting for Input
- Greeting the User with alert()

Alert

- It shows a message and waits for the user to press “OK”.
- Example:-

```
alert("Hello");
```

- The mini-window with the message is called a *modal window*.
- The word “modal” means that the visitor can’t interact with the rest of the page, press other buttons, etc, until they have dealt with the window. In this case – until they press “OK”.

Prompt

- It shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel.
- The visitor can type something in the prompt input field and press OK. Then we get that text in the result. Or they can cancel the input by pressing Cancel or hitting the Esc key, then we get null as the result.
- The call to prompt returns the text from the input field or null if the input was cancelled.
- Example:-

```
let age = prompt('How old are you?', 100);
```

Confirm

- The function confirm shows a modal window with a question and two buttons: OK and Cancel.

- The result is true if OK is pressed and false otherwise.

- Example:-

```
let isBoss = confirm("Are you the boss?");
```

- We covered 3 browser-specific functions to interact with visitors:
- **Alert**, shows a message.
- **Prompt**, shows a message asking the user to input text. It returns the text or, if Cancel button or Esc is clicked, null.
- **Confirm**, shows a message and waits for the user to press “OK” or “Cancel”. It returns true for OK and false for Cancel/Esc.
- All these methods are modal: they pause script execution and don’t allow the visitor to interact with the rest of the page until the window has been dismissed.
- There are two limitations shared by all the methods above:
  - The exact location of the modal window is determined by the browser. Usually, it’s in the center.
  - The exact look of the window also depends on the browser. We can’t modify it.

Basic operators

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Bitwise Operators
4. Logical Operators
5. Assignment Operators
6. Special Operators

Arithmetic Operators

Operator	Description	Example
+	Addition	10+20 = 30
-	Subtraction	20-10 = 10
*	Multiplication	10*20 = 200
/	Division	20/10 = 2
%	Modulus (Remainder)	20%10 = 0
++	Increment	var a=10; a++; Now a = 11
--	Decrement	var a=10; a--; Now a = 9

Lets Code

Write a program to make calculator with the help of HTML, CSS, Javascript

Comparison Operators

Operator	Description	Example
==	Is equal to	10==20 = false
===	Identical (equal and of same type)	10==20 = false
!=	Not equal to	10!=20 = true
!==	Not Identical	20!==20 = false
>	Greater than	20>10 = true
>=	Greater than or equal to	20>=10 = true
<	Less than	20<10 = false
<=	Less than or equal to	20<=10 = false

Bitwise Operators

Operator	Description	Example
&	Bitwise AND	(10==20 & 20==33) = false
|	Bitwise OR	(10==20 | 20==33) = false
^	Bitwise XOR	(10==20 ^ 20==33) = false
~	Bitwise NOT	(~10) = -10
<<	Bitwise Left Shift	(10<<2) = 40
>>	Bitwise Right Shift	(10>>2) = 2
>>>	Bitwise Right Shift with Zero	(10>>>2) = 2

Logical Operators

Operator	Description	Example
&&	Logical AND	(10==20 && 20==33) = false
||	Logical OR	(10==20 || 20==33) = false
!	Logical Not	!(10==20) = true

Assignment Operators

Operator	Description	Example
=	Assign	10+10 = 20
+=	Add and assign	var a=10; a+=20; Now a = 30
-=	Subtract and assign	var a=20; a-=10; Now a = 10
*=	Multiply and assign	var a=10; a*=20; Now a = 200
/=	Divide and assign	var a=10; a/=2; Now a = 5
%=	Modulus and assign	var a=10; a%=2; Now a = 0

Special Operators

Operator	Description
(?:)	Conditional Operator returns value based on the condition. It is like if-else.
,	Comma Operator allows multiple expressions to be evaluated as single statement.
delete	Delete Operator deletes a property from the object.
in	In Operator checks if object has the given property
instanceof	checks if the object is an instance of given type
new	creates an instance (object)
typeof	checks the type of object.
void	it discards the expression's return value.
yield	checks what is returned in a generator by the generator's iterator.

Conditional Branching

- If
- Else
- Else if
- Conditional operators
- Multiple ?
- Non-traditional use of ‘?’

Switch

- The JavaScript switch statement is used to execute one code from multiple expressions. It is just like else if statement that we have learned in previous page. But it is convenient than if..else..if because it can be used with numbers, characters etc.

- Lets see syntax and practice …..

```
switch(expression){
case value1:
 code to be executed;
 break;
case value2:
 code to be executed;
 break;
......

default:
 code to be executed if above values are not matched;
}
```

Loops

- The **JavaScript loops** are used *to iterate the piece of code* using for, while, do while or for-in loops. It makes the code compact. It is mostly used in array.

- There are four types of loops in JavaScript.
  - for loop
  - while loop
  - do-while loop
  - for-in loop

For Loop

- The JavaScript for loop iterates the elements for the fixed number of times. It should be used if number of iteration is known. The syntax of for loop is given below.

```
for (initialization; condition; increment)
{
    code to be executed
}
```

while loop

- The JavaScript while loop iterates the elements for the infinite number of times. It should be used if number of iteration is not known. The syntax of while loop is given below.

```
while (condition)
{
    code to be executed
}
```

Do While

- The JavaScript do while loop iterates the elements for the infinite number of times like while loop. But, code is executed at least once whether condition is true or false. The syntax of do while loop is given below.

```
do{
    code to be executed
}while (condition);
```

JavaScript for in loop

- The JavaScript for in loop is used to iterate the properties of an object.

Example:-

```
var person = {fname:"John", lname:"Doe", age:25};
var text = "";
var x;
for (x in person) {
  text += person[x] + " ";
}
```

Nullish coalescing operator '??'

- The nullish coalescing operator is written as two question marks ??.
- it treats null and undefined similarly.
- We'll say that an expression is "defined" when it's neither null nor undefined.
- The result of a ?? b is:
  - if a is defined, then a,
  - if a isn't defined, then b.
- ?? returns the first argument if it's not null/undefined. Otherwise, the second one.
- The nullish coalescing operator isn't anything completely new. It's just a nice syntax to get the first "defined" value of the two.

Did you thought???????????

- What is the difference between ?? And ||
- Due to safety reasons, JavaScript forbids using ?? together with && and || operators, unless the precedence is explicitly specified with parentheses.

- The nullish coalescing operator ?? provides a short way to choose the first “defined” value from a list.

- It’s used to assign default values to variables:

- The operator ?? has a very low precedence, only a bit higher than ? and =, so consider adding parentheses when using it in an expression.

- It’s forbidden to use it with || or && without explicit parentheses.

Functions

- Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

```
function name(parameters) {
  alert( message);
}
```

- The function keyword goes first, then goes the name of the function, then a list of parameters between the parentheses and finally the code of the function, also named “the function body”, between curly braces.

- **JavaScript functions** are used to perform operations. We can call JavaScript function many times to reuse the code.
- Advantage of JavaScript function
- There are mainly two advantages of JavaScript functions.
  1. **Code reusability**: We can call a function several times so it save coding.
  2. **Less coding**: It makes our program compact. We don’t need to write many lines of code each time to perform a common task.

- Local Variables
- Outer variables
- Parameters
- Default values
- Returning a value

- A function declaration looks like this:

```
function name(parameters, delimited, by, comma) {
  /* code */
}
```

- Values passed to a function as parameters are copied to its local variables.
- A function may access outer variables. But it works only from inside out. The code outside of the function doesn’t see its local variables.
- A function can return a value. If it doesn’t, then its result is undefined.
- To make the code clean and easy to understand, it’s recommended to use mainly local variables and parameters in the function, not outer variables.

- It is always easier to understand a function which gets parameters, works with them and returns a result than a function which gets no parameters, but modifies outer variables as a side-effect.

Function naming:

- A name should clearly describe what the function does. When we see a function call in the code, a good name instantly gives us an understanding what it does and returns.

- A function is an action, so function names are usually verbal.

- There exist many well-known function prefixes like create…, show…, get…, check… and so on. Use them to hint what a function does.

- Functions are the main building blocks of scripts. Now we've covered the basics, so we actually can start creating and using them. But that's only the beginning of the path. We are going to return to them many times, going more deeply into their advanced features.

JavaScript Function Object

- The purpose of **Function constructor** is to create a new Function object. It executes the code globally. However, if we call the constructor directly, a function is created dynamically but in an unsecured way.

  new Function ([arg1[, arg2[, ...argn]],] functionBody)

- arg1, arg2, ... , argn - It represents the argument used by function.

- functionBody - It represents the function definition.

JavaScript Function Methods

Method	Description
apply()	It is used to call a function contains this value and a single array of arguments.
bind()	It is used to create a new function.
call()	It is used to call a function contains this value and an argument list.
toString()	It returns the result in a form of a string.

Advance Features of JavaScript +ES6

- Arrow function
- Objects
- Advance datatypes
- Classes
- Prototyping
- promises
- Error handling
- Modules

Arrow function

- Arrow functions, introduced in ES6, provides a concise way to write functions in JavaScript.

- Another significant advantage it offers is the fact that it does not bind its own this. In other words, the context inside arrow functions is lexically or statically defined.

- Unlike other functions, the value of this inside arrow functions is not dependent on how they are invoked or how they are defined. It depends only on its enclosing context.

- Syntax:

```
let func = (arg1, arg2, ..., argN) => expression
```

Multiline arrow functions

- Sometimes we need something a little bit more complex, like multiple expressions or statements. It is also possible, but we should enclose them in curly braces. Then use a normal return within them.

```
let sum = (a, b) => {
  let result = a + b;
  return result;
};
alert( sum(1, 2) );
```

- Arrow functions are handy for one-liners. They come in two flavors:
  - Without curly braces: (...args) => expression – the right side is an expression: the function evaluates it and returns the result.
  - With curly braces: (...args) => { body } – brackets allow us to write multiple statements inside the function, but we need an explicit return to return something.

Example:-

```
hello = function() {
  return "Hello World!";
}
```

Objects:-

- Objects are more complex and each object may contain any combination of these primitive data-types as well as reference data-types.
- An object, is a reference data type. Variables that are assigned a reference value are given a reference or a pointer to that value. That reference or pointer points to the location in memory where the object is stored. The variables don’t actually store the value.
- Loosely speaking, objects in JavaScript may be defined as an unordered collection of related data, of primitive or reference types, in the form of “key: value” pairs. These keys can be variables or functions and are called properties and methods, respectively, in the context of an object.

- An object can be created with figure brackets {…} with an optional list of properties. A property is a “key: value” pair, where a key is a string (also called a “property name”), and value can be anything.

- There are 3 ways to create objects.

1. By object literal:

   object={property1:value1,property2:value2.....propertyN:valueN}

2. By creating instance of Object directly (using new keyword)

   var objectname=new Object();

3. By using an object constructor (using new keyword)

   e=new emp(103,"Vimal Jaiswal",30000);

Closures

- Whenever a function is invoked, a new scope is created for that call. The local variable declared inside the function belong to that scope – they can only be accessed from that function -. It’s very important to understand that before moving further.

- The function scope is created for a function call, not for the function itself

- Every function call creates a new scope

- When the function has finished the execution, the scope is usually destroyed.

- A **closure** is a function which has access to the variable from another function's scope. This is accomplished by creating a function inside a function. Of course, the outer function does not have access to the inner scope.
- Closure are nested function which has access to the outer scope
- After the outer function is returned, by keeping a reference to the inner function (the closures) we prevent the outer scope to be destroyed.

```
var add = (function () {
var counter = 0;
return function () {counter += 1; return counter}
})();
```

Classes

- n JavaScript, classes are the special type of functions. We can define the class just like function declarations and function expressions.
- The JavaScript class contains various class members within a body including methods or constructor. The class is executed in strict mode. So, the code containing the silent error or mistake throws an error.
- The class syntax contains two components:
    - Class declarations
    - Class expressions

- Syntax:-

```
class ClassName {
    constructor() { ... }
}
```

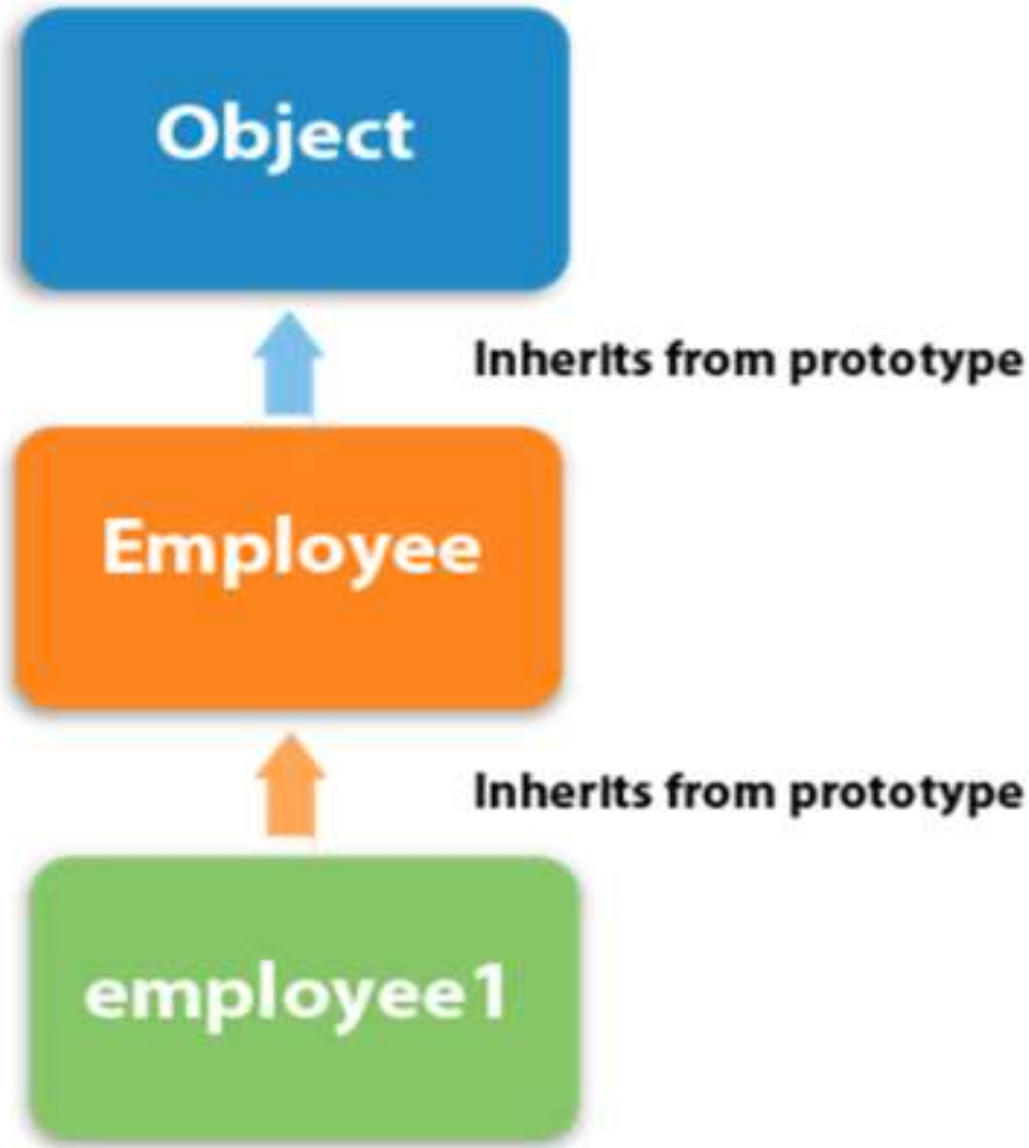
- Lets Practically code and see few more concepts related to classes…

Prototyping

- JavaScript is a prototype-based language that facilitates the objects to acquire properties and features from one another. Here, each object contains a prototype object.

- In JavaScript, whenever a function is created the prototype property is added to that function automatically. This property is a prototype object that holds a constructor property.

- ClassName.prototype.methodName


Object
Inherits from prototype
Employee
Inherits from prototype
employee1

- In JavaScript, all objects have a hidden [[Prototype]] property that's either another object or null.
- We can use obj.__proto__ to access it (a historical getter/setter, there are other ways, to be covered soon).
- The object referenced by [[Prototype]] is called a "prototype".
- If we want to read a property of obj or call a method, and it doesn't exist, then JavaScript tries to find it in the prototype.
- Write/delete operations act directly on the object, they don't use the prototype (assuming it's a data property, not a setter).
- If we call obj.method(), and the method is taken from the prototype, this still references obj. So methods always work with the current object even if they are inherited.
- The for..in loop iterates over both its own and its inherited properties. All other key/value-getting methods only operate on the object itself.

Promises

- Promises are one of the ways we can deal with asynchronous operations in JavaScript.
- A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.
- Promises are eager, meaning that a promise will start doing whatever task you give it as soon as the promise constructor is invoked.

➢**Syntax:**

```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Modules

- Good authors divide their books into chapters and sections; good programmers divide their programs into modules.

- A module is just a file. One script is one module. As simple as that.

- Modules can load each other and use special directives export and import to interchange functionality, call functions of one module from another one:

- export keyword labels variables and functions that should be accessible from outside the current module.
- import allows the import of functionality from other modules.

- A module is a file. To make import/export work, browsers need <script type="module">. Modules have several differences:
- Deferred by default.
- Async works on inline scripts.
- To load external scripts from another origin (domain/protocol/port), CORS headers are needed.
- Duplicate external scripts are ignored.
- Modules have their own, local top-level scope and interchange functionality via import/export.
- Modules always use strict.
- Module code is executed only once. Exports are created once and shared between importers.
- When we use modules, each module implements the functionality and exports it. Then we use import to directly import it where it’s needed. The browser loads and evaluates the scripts automatically.

Error Handling

- The try statement lets you test a block of code for errors.
- The catch statement lets you handle the error.
- The throw statement lets you create custom errors.
- The finally statement lets you execute code, after try and catch, regardless of the result.

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of the try / catch result
}
```

- Lets talk more about the Errors and practical examples…….

Thanks