INTRODUCTION TO JAVASCRIPT:

JavaScript is a widely-used, high-level, and versatile programming language primarily known for its role in web development. It was created by Brendan Eich in 1995 and has since become an integral part of building interactive and dynamic websites and web applications.

Key points about JavaScript:

1. **Client-Side Scripting:** JavaScript is primarily used for client-side scripting, which means it runs in the user's web browser. It allows developers to create interactive and responsive user interfaces directly within web pages.
2. **Dynamic and Event-Driven:** JavaScript enables developers to make web pages come to life by responding to user actions (e.g., clicks, form submissions) and dynamically updating content without the need for a page reload.
3. **Cross-Platform:** JavaScript is supported by all major web browsers, making it a cross-platform language that works on various devices and operating systems.
4. **Versatile:** While JavaScript is most commonly associated with web development, it can also be used for server-side scripting (Node.js), desktop application development (Electron), game development (using libraries like Phaser), and more.
5. **Syntax:** JavaScript has a C-style syntax, making it relatively easy for developers who are familiar with languages like C++, Java, or C# to learn. It uses variables, functions, conditionals, loops, and objects to create robust and maintainable code.
6. **Libraries and Frameworks:** There are numerous libraries and frameworks built on top of JavaScript, such as React, Angular, and Vue.js for building web interfaces, and Express.js for building server-side applications. These tools simplify development and enhance productivity.
7. **ECMAScript:** JavaScript follows the ECMAScript standard, which defines the language's core features and capabilities. As of my last knowledge update in September 2021, ECMAScript 2021 (ES12) was the latest version. New features and improvements are regularly introduced to enhance the language.

HOW JAVASCRIPT MAKES HTML BUILD WEBSITE BETTER?

1. **Interactivity**: JavaScript allows developers to create interactive web pages. Users can click buttons, submit forms, drag and drop elements, and perform other actions that trigger responses in real-time. This interactivity enhances user engagement and provides a more dynamic experience.
2. **Dynamic Content**: JavaScript can dynamically update and modify content on a webpage without requiring a full page reload. This ability to change text, images, and other elements on the fly is vital for building responsive and modern web applications.

3. **Form Validation**: JavaScript can validate form inputs in real-time, providing immediate feedback to users if they enter invalid data. This helps ensure data integrity and improves the user experience by preventing submission errors.
4. **Client-Side Data Manipulation**: JavaScript can manipulate data on the client side before sending it to the server. This reduces server load and latency, leading to a faster and more efficient user experience.
5. **User Interface Enhancements**: JavaScript frameworks and libraries (e.g., React, Angular, Vue.js) simplify the development of rich and responsive user interfaces. They offer components, templates, and tools for building complex UIs, reducing development time and effort.
6. **AJAX (Asynchronous JavaScript and XML)**: JavaScript enables AJAX, which allows websites to retrieve data from a server without refreshing the entire page. This asynchronous data loading enhances website performance and user satisfaction.
7. **Animations**: JavaScript facilitates animations and transitions, making websites more visually appealing and engaging. CSS and JavaScript libraries like jQuery and GSAP are commonly used for creating animations.
8. **Cross-Browser Compatibility**: JavaScript libraries and frameworks often handle browser-specific inconsistencies and compatibility issues, ensuring that web applications work consistently across various browsers and devices.
9. **Third-Party Integrations**: JavaScript can be used to integrate third-party services and APIs into a website, such as social media sharing, payment gateways, mapping services, and more.
10. **User Experience Optimization**: JavaScript can be used to implement features like infinite scrolling, lazy loading of images, and smooth scrolling, which improve the overall user experience by reducing page load times and enhancing navigation.
11. **Data Visualization**: JavaScript libraries like D3.js and Chart.js enable the creation of interactive and visually appealing data visualizations, charts, and graphs.
12. **Real-Time Updates**: JavaScript can establish WebSocket connections to enable real-time updates in applications like chat applications, collaborative tools, and online gaming.
13. **Progressive Web Apps (PWAs)**: JavaScript is essential for building PWAs, which offer native app-like experiences in web browsers, including offline capabilities, push notifications, and home screen installation.

JAVASCRIPT VERSIONS

1. **JavaScript 1.0 (1995):** This was the initial release of JavaScript, introduced by Netscape. It provided basic scripting capabilities for web pages, allowing developers to interact with HTML elements and handle user events.
2. **JavaScript 1.1 (1996):** Also known as "LiveScript," this version added some new features like exceptions and the ability to create custom objects.
3. **JavaScript 1.2 (1997):** This version, introduced by Netscape Navigator 3.0, included support for regular expressions, enhanced string handling, and better support for manipulating HTML elements.
4. **JavaScript 1.3 (1998):** Introduced by Netscape Communicator 4.0, this version brought improvements to object-oriented programming, added new methods to built-in objects, and introduced the **try...catch** statement for better error handling.
5. **JavaScript 1.4 (1999):** This version introduced support for ECMAScript's prototype-based inheritance and included a **link** tag for including external scripts.
6. **JavaScript 1.5 (2000):** Introduced by Netscape 6.0, this version included several important features such as the **forEach** method for arrays, native support for JSON (JavaScript Object Notation), and improvements in regular expression handling.
7. **ECMAScript 3 (ES3, 1999):** Although not a JavaScript version per se, ES3 was a significant milestone. It formalized the language specification and introduced features like try...catch exception handling and strict mode.
8. **ECMAScript 4 (ES4, Abandoned):** ES4 was an attempt to significantly update the language, but it faced considerable controversy and was ultimately abandoned in favor of a more incremental approach to language development.
9. **ECMAScript 5 (ES5, 2009):** ES5 was a major revision of the language and introduced important features like **Object.defineProperty** for property manipulation, **JSON.parse** and **JSON.stringify** for working with JSON data, and the **bind** method for functions.
10. **ECMAScript 5.1 (ES5.1, 2011):** This was a minor update to ES5 to fix some issues in the specification.
11. **ECMAScript 6 (ES6, or ECMAScript 2015):** ES6 introduced a substantial number of new features and syntax enhancements, including arrow functions, classes, modules, the **let** and **const** keywords for variable declarations, and many others.
12. **ECMAScript 2016 (ES2016 or ES7):** This version included features like the **Array.prototype.includes** method for checking if an array includes a particular value.
13. **ECMAScript 2017 (ES2017 or ES8):** ES2017 introduced features like asynchronous functions (**async/await**) and shared memory and atomics for better multi-threading support.

14. **ECMAScript 2018 (ES2018 or ES9):** This version introduced features like asynchronous iteration, **Promise.prototype.finally**, and rest/spread properties for objects.
15. **ECMAScript 2019 (ES2019 or ES10):** ES2019 introduced features like **Array.prototype.flat** and **Array.prototype.flatMap**, **Object.fromEntries**, and more.
16. **ECMAScript 2020 (ES2020 or ES11):** This version introduced features like the optional chaining operator (**?.**), the nullish coalescing operator (**??**), and **BigInt** for handling arbitrarily large integers.
17. **ECMAScript 2021 (ES2021 or ES12):** ES2021 introduced features like the **String.prototype.replaceAll** method, numeric separators (e.g., **1_000_000**), and **Promise.any** for handling promises.

APPLICATION OF JAVASCRIPT:

- Client-side validation,
- Dynamic drop-down menus,
- Displaying date and time,
- Displaying pop-up windows and dialog boxes (like an alert dialog box, confirm dialog box and prompt dialog box),
- Displaying clocks etc.

PLACES TO PUT JAVASCRIPT CODE

1. Between the body tag of html
2. Between the head tag of html
3. In .js file (external javaScript)

1. Code Between The Body Tag

In the above example, we have displayed the dynamic content using JavaScript. Let's see the simple example of JavaScript that displays alert dialog box.

```
<script type="text/javascript">
        alert("Hello Javatpoint");
</script>
```

2. Code Between The Head Tag

In this example, we are creating a function msg(). To create function in JavaScript, you need to write function with function_name as given below.

To call function, you need to work on event. Here we are using onclick event to call msg() function.

```
<html>
<head>
<script type="text/javascript">
function msg(){
alert("Hello ");
}
</script>
</head>
<body>
<p>Welcome to JavaScript</p>
<form>
<input type="button" value="click" onclick="msg()"/>
</form>
</body>
</html>
```

3. **External JavaScript file:**
   We can create external JavaScript file and embed it in many html page.

   It provides code re usability because single JavaScript file can be used in several html pages.

   An external JavaScript file must be saved by .js extension. It is recommended to embed all JavaScript files into a single file. It increases the speed of the webpage.

   message.js

```
function msg(){
alert("Hello ");
}
```

   index.html

```
<html>
<head>
<script type="text/javascript" src="message.js"></script>
</head>
<body>
<p>Welcome to JavaScript</p>
```

```
<form>
<input type="button" value="click" onclick="msg()"/>
</form>
</body>
</html>
```

Keywords in JavaScript:

Keywords in JavaScript are reserved words that have special meanings and purposes in the language.

1. **break**: Used to exit a loop or switch statement.
2. **case**: Used in a switch statement to specify different code to execute based on a particular value.
3. **catch**: Used to catch and handle exceptions in a try-catch block.
4. **class**: Introduced in ECMAScript 6 (ES6), used to define classes and create objects using the class-based syntax.
5. **const**: Used to declare a constant variable with a value that cannot be reassigned.
6. **continue**: Used to skip the current iteration of a loop and move to the next one.
7. **debugger**: Used to set a breakpoint in your code for debugging purposes.
8. **default**: Used in a switch statement to specify the code to execute when no other case matches.
9. **delete**: Used to remove a property from an object.
10. **do**: Used to create a do-while loop.
11. **else**: Used to specify the code to execute if a conditional statement is false.
12. **export**: Used to export variables, functions, or classes from a module in ES6 modules.
13. **extends**: Used to create a subclass in class inheritance.

14. **false**: A boolean value representing the concept of "false."
15. **finally**: Used in a try-catch-finally block to specify code that always executes, whether an exception is thrown or not.
16. **for**: Used to create a for loop.
17. **function**: Used to declare a function.
18. **if**: Used to create a conditional statement.
19. **import**: Used to import variables, functions, or classes from a module in ES6 modules.
20. **in**: Used to check if an object has a certain property.
21. **instanceof**: Used to check if an object is an instance of a particular class or constructor function.
22. **new**: Used to create an instance of a constructor function or a class.
23. **null**: A special value representing the absence of an object.
24. **return**: Used to specify the value to return from a function.
25. **super**: Used to call a method on the superclass in class inheritance.
26. **switch**: Used to create a switch statement for multi-case branching.
27. **this**: Refers to the current context or object.
28. **throw**: Used to throw an exception.
29. **true**: A boolean value representing the concept of "true."
30. **try**: Used to create a try-catch block for exception handling.
31. **typeof**: Used to check the data type of an expression.
32. **var**: Used to declare a variable (older way, avoid using **var** in favor of **let** and **const** in modern JavaScript).
33. **void**: Used to evaluate an expression and return **undefined**.
34. **while**: Used to create a while loop.
35. **with**: Used to create a block in which you can access object properties without specifying the object explicitly (not recommended and discouraged in modern JavaScript).

Naming Conventions for JavaScript Variables

- A variable name must start with a letter, underscore (_), or dollar sign ($).
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters (A-z, 0-9) and underscores.
- A variable name cannot contain spaces.
- A variable name cannot be a JavaScript keyword or a JavaScript reserved word.

var, let and const keywords in javascript:

In JavaScript, var, let, and const are keywords used to declare variables.

var:

- Variables declared with **var** are function-scoped. This means they are only accessible within the function in which they are defined.
- Variables declared with **var** can be reassigned, and multiple declarations with the same name within the same scope are allowed.

```
function example() {
  var x = 10;
  if (true) {
    var x = 20; // This reassigns the outer 'x'
  }
  console.log(x); // Outputs 20
}
```

let:

- Variables declared with **let** are block-scoped, meaning they are only accessible within the nearest enclosing pair of curly braces (block) where they are defined.
- Variables declared with **let** can be reassigned, but they cannot be re-declared within the same scope.

```
function example() {
  let x = 10;
  if (true) {
    let x = 20; // This creates a new 'x' in the inner block
  }
  console.log(x); // Outputs 10
}
```

const:

- Variables declared with **const** are also block-scoped.
- **const** variables must be initialized when declared, and their values cannot be changed after initialization. They are used for defining constants.
- You cannot reassign or redeclare a **const** variable within the same scope.

```
const pi = 3.14159;
pi = 3.14; // This will result in an error

const x; // This will result in an error; 'const' variables must be initialized
```

Operators In Javascript

JavaScript has 8 types of operators that allow you to perform operations on values and variables. Here are some common types of operators with examples:

Arithmetic Operators:

- These operators perform basic mathematical operations.

```
let x = 10;
let y = 5;
```

```
let addition = x + y;    // Addition
let subtraction = x - y; // Subtraction
let multiplication = x * y; // Multiplication
let division = x / y; // Division
let remainder = x % y; // Modulo (Remainder after division)
```

Comparison Operators:

These operators compare two values and return a Boolean result.

```
let a = 5;
let b = 10;

let isEqual = a === b; // Equality
let isNotEqual = a !== b; // Inequality
let isGreaterThan = a > b; // Greater than
let isLessThan = a < b; // Less than
let isGreaterOrEqual = a >= b; // Greater than or equal to
let isLessOrEqual = a <= b; // Less than or equal to
```

Logical Operators:

- These operators are used for logical operations and work with Boolean values.

  ```
  let isTrue = true;
  let isFalse = false;

  let andResult = isTrue && isFalse; // Logical AND
  let orResult = isTrue || isFalse; // Logical OR
  let notResult = !isTrue; // Logical NOT
  ```

Assignment Operators:

- These operators assign values to variables.

```
let num = 10;
num += 5; // Equivalent to num = num + 5
num -= 3; // Equivalent to num = num - 3
num *= 2; // Equivalent to num = num * 2
num /= 4; // Equivalent to num = num / 4
```

Increment and Decrement Operators:

- These operators are used to increase or decrease the value of a variable by 1.

```
let counter = 5;
counter++; // Increment by 1 (post-increment)
++counter; // Increment by 1 (pre-increment)
counter--; // Decrement by 1 (post-decrement)
--counter; // Decrement by 1 (pre-decrement)
```

String Concatenation Operator:

- The **+** operator can be used for concatenating strings.

```
let firstName = 'Rajat';
let lastName = 'Kumar';
let fullName = firstName + ' ' + lastName; // Concatenation
```

Ternary (Conditional) Operator:

- It's a shorthand way of writing conditional statements.

```
let age = 18;
let canVote = age >= 18 ? 'Yes' : 'No'; // If-else in a single line
```

Typeof Operator:

- It's used to check the data type of a value.

```
let value = 42;
let valueType = typeof value; // Returns 'number'
```

Difference Between “console.Log()” And “document.Write()” In Javascript

console.log:

- **console.log** is used for debugging and logging messages to the console for developers.
- It is primarily used during development to inspect and log the state of variables and objects, trace code execution, and find and fix issues.
- It does not affect the actual content of the web page; it's only visible in the browser's developer tools console.
- It's the preferred way to log information during development because it doesn't modify the document's content.

```
let message = "This is a log message.";

console.log(message); // Logs the message to the console
```

document.write:

- **document.write** is used to write content directly to the document (web page).
- It adds content to the HTML document at the exact place where the script is executed, and it can modify the page's structure in real-time.
- While it can be useful for simple demonstrations and quick testing, it's generally discouraged in modern web development because it can cause issues with page rendering and structure.

```
document.write("This text is written to the document.");
```

Conditional Statement In Javascript

Conditional statements in JavaScript allow you to make decisions in your code based on conditions. The most common types of conditional statements are the **if** statement, the **if-else** statement, and the **switch** statement. Here are examples of each:

if Statement: The **if** statement is used to execute a block of code only if a specified condition is true.

```
let age = 18;

if (age >= 18) {
  console.log("You are an adult.");
}
```

if-else Statement: The **if-else** statement allows you to execute one block of code if a condition is true and another block of code if the condition is false.

```
let time = 10;

if (time < 12) {
```

```
  console.log("Good morning!");
} else {
  console.log("Good afternoon!");
}
```

else if Statement: You can use multiple **else if** clauses to check multiple conditions.

```
let time = 18;
if (time < 12) {
  console.log("Good morning!");
} else if (time < 18) {
  console.log("Good afternoon!");
} else {
  console.log("Good evening!");}
```

switch Statement: The **switch** statement is used to select one of many code blocks to be executed. It's often used when you have multiple conditions to check against a single value

```
let day = "Monday";

switch (day) {
  case "Monday":
    console.log("It's the start of the workweek.");
    break;
  case "Friday":
    console.log("It's almost the weekend!");
    break;
  default:
    console.log("It's an ordinary day.");
}
```

Ternary Operator: The ternary operator (**? :**) allows you to write a condensed form of an **if-else** statement.

```
let isRaining = true;
```

```
let weather = isRaining ? "Take an umbrella" : "Enjoy the sunshine";
console.log(weather);
```

JAVASCRIPTS LOOPS:

For Loop

While Loop

Do…while Loop

For….in Loop

For…..of Loop

for Loop:

The for loop is used to execute a block of code a specified number of times.

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

- The **for** loop consists of three parts: initialization, condition, and iteration.
- It's widely used for iterating over arrays, performing repetitive tasks, and counting operations.

while Loop: The **while** loop is used to execute a block of code as long as a specified condition is true.

```
let count = 0;
while (count < 5) {
  console.log(count);
  count++;
}
```

- The condition is checked before each iteration, and the loop continues as long as the condition is true.
- It's used when you don't know in advance how many times the loop should run.

do...while Loop: The **do...while** loop is similar to the **while** loop, but it always executes the block of code at least once, and then it continues as long as the condition is true.

```
let count = 0;
do {
  console.log(count);
  count++;
} while (count < 5);
```

- The condition is checked after each iteration.
- It's used when you want to ensure that a block of code is executed at least once.

for...in Loop: The **for...in** loop is used to iterate over the properties of an object.

```
const person = { name: "Alice", age: 30 };
for (let key in person) {
  console.log(key + ": " + person[key]);
}
```

- It's primarily used for iterating over the properties of objects, not for arrays.
- Use it with caution, as it may also iterate over properties in the object's prototype chain.

for...of Loop: The **for...of** loop is used to iterate over the values of iterable objects, such as arrays and strings.

```
const fruits = ["apple", "banana", "cherry"];
for (let fruit of fruits) {
  console.log(fruit);
}
```

JAVASCRIPT POP UP BOXES

Alert Box (alert): The **alert** box is used to display a simple message to the user. It only has an OK button and is often used for informational purposes.

```
alert("This is an alert box.");
```

Prompt Box (prompt): The **prompt** box allows you to prompt the user for input. It displays a message, an input field, and OK/Cancel buttons.

```
let name = prompt("Please enter your name:", "John Doe");

if (name !== null) {

  alert("Hello, " + name + "!");

} else {

  alert("You canceled the prompt.");

}
```

Confirm Box (confirm): The **confirm** box is used to get a yes/no response from the user. It displays a message and OK/Cancel buttons.

```
let result = confirm("Are you sure you want to delete this item?");

if (result) {

  alert("Item deleted.");

} else {

  alert("Deletion canceled.");

}
```

JavaScript Events

The change in the state of an object is known as an **Event**.

When javascript code is included in HTML, js react over these events and allow the execution. This process of reacting over the events is called **Event Handling**.

Thus, js handles the HTML events via **Event Handlers.**

Mouse events:

onclick: When mouse click on an element

onmouseover: When the cursor of the mouse comes over the element

onmouseout: When the cursor of the mouse leaves an element

onmousedown: When the mouse button is pressed over the element

onmouseup: When the mouse button is released over the element

onmousemove: When the mouse movement takes place.

Keyboard events:

onkeydown & onkeyup: When the user press and then release the key

Form events:

onfocus: When the user focuses on an element

onsubmit: When the user submits the form

onblur: When the focus is away from a form element

onchange: When the user modifies or changes the value of a form element

Window/Document events:

onload : When the browser finishes the loading of the page

onunload: When the visitor leaves the current webpage, the browser unloads it

onresize: When the visitor resizes the window of the browser

DOCUMENT OBJECT MODEL

The **document object** represents the whole html document.

the object of window.

window.document Is same as **document**

Methods of document object:-

write("string"): writes the given string on the doucment.

writeln("string"): writes the given string on the doucment with newline character at the end**.**

getElementById(): returns the element having the given id value.

getElementsByTagName(): returns all the elements having the given tag name.

getElementsByClassName(): returns all the elements having the given class name.

JAVASCRIPT ARRAY

JavaScript array is an object that represents a collection of similar type of elements.

There are 3 ways to construct array in JavaScript

1. By array literal
2. By creating instance of Array directly (using new keyword)
3. By using an Array constructor (using new keyword)

1) JavaScript array literal: var arrayname=[value1,value2.....valueN];

```
<script>
var emp=["Aditya","Vimal","Sumit"];
for (i=0;i<emp.length;i++){
document.write(emp[i] + "<br/>");
}
</script>
```

2) JavaScript Array directly (new keyword): var arrayname=new Array();

```
<script>
var i;
var emp = new Array();
emp[0] = "Aditya";
emp[1] = "Vimal";
emp[2] = "Sumit";
for (i=0;i<emp.length;i++){
document.write(emp[i] + "<br>");
}
</script>
```

3) JavaScript array constructor (new keyword):

Here, you need to create instance of array by passing arguments in constructor so that we don't have to provide value explicitly.

```
<script>
var emp=new Array("Aditya","Vimal","Sumit");
for (i=0;i<emp.length;i++){
document.write(emp[i] + "<br>");
}
</script>
```

JAVASCRIPT ARRAY METHOD:

1. **push()**: Adds one or more elements to the end of an array and returns the new length of the array.

   let arr = [1, 2, 3]; arr.push(4); // arr is now [1, 2, 3, 4]

2. **pop()**: Removes the last element from an array and returns that element.

   let arr = [1, 2, 3, 4]; let removed = arr.pop(); // arr is now [1, 2, 3], removed is 4

3. **unshift()**: Adds one or more elements to the beginning of an array and returns the new length of the array.

   let arr = [2, 3, 4]; arr.unshift(1); // arr is now [1, 2, 3, 4]

4. **shift()**: Removes the first element from an array and returns that element.

   let arr = [1, 2, 3, 4]; let removed = arr.shift(); // arr is now [2, 3, 4], removed is 1

5. **concat()**: Combines two or more arrays and returns a new array.

   let arr1 = [1, 2]; let arr2 = [3, 4]; let combined = arr1.concat(arr2); // combined is [1, 2, 3, 4]

6. **slice()**: Returns a shallow copy of a portion of an array into a new array.

   let arr = [1, 2, 3, 4, 5]; let subArray = arr.slice(1, 4); // subArray is [2, 3, 4]

7. **splice()**: Changes the contents of an array by removing, replacing, or adding elements in-place.

   let arr = [1, 2, 3, 4, 5]; arr.splice(2, 1, 'a', 'b'); // arr is now [1, 2, 'a', 'b', 4, 5]

8. **forEach()**: Executes a provided function once for each array element.

   let arr = [1, 2, 3]; arr.forEach((element) => { console.log(element); }); // This will log: // 1 // 2 // 3

9. **map()**: Creates a new array with the results of calling a provided function on every element in the array.

   let arr = [1, 2, 3]; let doubled = arr.map((element) => element * 2); // doubled is [2, 4, 6]

10. **filter()**: Creates a new array with all elements that pass a provided test.

let arr = [1, 2, 3, 4, 5]; let evenNumbers = arr.filter((element) => element % 2 === 0); // evenNumbers is [2,4]

JavaScript Objects

In real life, a car is an **object**.

A car has **properties** like weight and color, and **methods** like start and stop:

Properties	Methods
car.name = Fiat	car.start()
car.model = 500	car.drive()
car.weight = 850kg	car.brake()
car.color = white	car.stop()

This code assigns a simple value (Fiat) to a variable named car:

let car = "Fiat";

Objects are variables too. But objects can contain many values.

This code assigns many values (Fiat, 500, white) to a variable named car:

const car = {type:"Fiat", model:"500", color:"white"};

The values are written as **name:value** pairs (name and value separated by a colon).

Object Definition

You define (and create) a JavaScript object with an object literal:

const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

Spaces and line breaks are not important. An object definition can span multiple lines:

```
const person = {
  firstName: Rajat",
  lastName: "Kumar",
  age: 28,
  weight: "76"
};
```

Accessing Object Properties

You can access object properties in two ways:

objectName.propertyName *OR*

objectName["propertyName"]

Object Methods

Objects can also have **methods**.

Methods are **actions** that can be performed on objects.

Methods are stored in properties as **function definitions**.

function() {

return this.firstName + " " + this.lastName;

}

A method is a function stored as a property.

Ex: const person = {

```
 firstName: "Rajat",
 lastName : "Kumar",
 id: 5566,
 fullName : function() {
   return this.firstName + " " + this.lastName;
 }
};
```

In the example above, this refers to the person object.

i.e. this.firstName means the firstName property of this.

i.e. this.firstName means the firstName property of person.

What is this?

In JavaScript, the this keyword refers to an **object**.

Which object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:

In an object method, **this** refers to the **object**.

Alone, **this** refers to the **global object**.

In a function, **this** refers to the **global object.**

In a function, in strict mode, **this** is **undefined.**

In an event, this refers to the **element** that received the event.

Methods like **call()**, **apply()**, and **bind()** can refer this to any object.

this is not a variable. It is a keyword. You cannot change the value of **this**

The this Keyword

In a function definition, this refers to the "owner" of the function.

In the example above, this is the **person object** that "owns" the fullName function.

In other words, this.firstName means the firstName property of **this object**.

Accessing Object Methods:

objectName.methodName()

name = person.fullName(); OR

name = person.fullName;

Do Not Declare Strings, Numbers, and Booleans as Objects!

When a JavaScript variable is declared with the keyword "**new**", the variable is created as an object:

x = new String(); // Declares x as a String object

y = new Number(); // Declares y as a Number object

z = new Boolean(); // Declares z as a Boolean object

Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

JAVASCRIPT FUNCTIONS

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

Ex: // Function to compute the product of p1 and p2

```
function myFunction(p1, p2) {
  return p1 * p2;
}
```

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

Function Invocation:

The code inside the function will execute when "something" invokes (calls) the function:

When an event occurs (when a user clicks a button)

When it is invoked (called) from JavaScript code

Automatically (self invoked)

Function Return

When JavaScript reaches a return statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

Ex:

```
// Function is called, the return value will end up in x
let x = myFunction(4, 3);
function myFunction(a, b) {
// Function returns the product of a and b
  return a * b;
}
```

JavaScript Form Validation

HTML form validation can be done by JavaScript.

If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

Ex:

```
function validateForm() {
  let x = document.forms["myForm"]["fname"].value;
  if (x == "") {
    alert("Name must be filled out");
    return false;
  }
}
```

The function can be called when the form is submitted:

```
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

Automatic HTML Form Validation

If a form field (fname) is empty, the required attribute prevents this form from being submitted:

```
<form action="/action_page.php" method="post">
  <input type="text" name="fname" required>
  <input type="submit" value="Submit">
</form>
```

Data Validation

Data validation is the process of ensuring that user input is clean, correct, and useful.

Typical validation tasks are:

- has the user filled in all required fields?
- has the user entered a valid date?
- has the user entered text in a numeric field?

Most often, the purpose of data validation is to ensure correct user input.

Validation can be defined by many different methods, and deployed in many different ways.

Server side validation is performed by a web server, after input has been sent to the server.

Client side validation is performed by a web browser, before input is sent to a web server.

HTML Constraint Validation

HTML5 introduced a new HTML validation concept called constraint validation.

HTML constraint validation is based on:

Constraint validation HTML Input Attributes

Constraint validation CSS Pseudo Selectors

Constraint validation DOM Properties and Methods

Attribute	Description
disabled	Specifies that the input element should be disabled
max	Specifies the maximum value of an input element
min	Specifies the minimum value of an input element
pattern	Specifies the value pattern of an input element
required	Specifies that the input field requires an element
type	Specifies the type of an input element

Constraint Validation CSS Pseudo Selectors:

Selector	Description
:disabled	Selects input elements with the "disabled" attribute specified
:invalid	Selects input elements with invalid values
:optional	Selects input elements with no "required" attribute specified
:required	Selects input elements with the "required" attribute specified
:valid	Selects input elements with valid values

Changing HTML Style

To change the style of an HTML element, use this syntax:

document.getElementById(id).style.property = new style

The following example changes the style of a **<p>** element:

```
<html>
<body>
<p id="p2">This is paragraph</p>
<script>
document.getElementById("p2").style.color = "blue";
</script>
</body>
</html>
```

Using Events

The HTML DOM allows you to execute code when an event occurs.

Events are generated by the browser when "things happen" to HTML elements:

An element is clicked on

The page has loaded

Input fields are changed

Ex:

```
<!DOCTYPE html>
<html>
<body>
<h1 id="id1">My Heading 1</h1>
<button type="button"
onclick="document.getElementById('id1').style.color = 'red'">
Click Me!</button>
</body>
</html>
```

ARROW FUNCTION

Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax:

let myFunction = (a, b) => a * b;

Traditional Method:

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

New Method:

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

Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
```

Arrow Function with Parameters:

```
hello = (val) => "Hello " + val;
```

Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

JAVASCRIPT STRINGS

JavaScript strings are for storing and manipulating text.

```
let text = "Rajat";


let answer1 = "It's alright";
let answer2 = "He is called 'PM' ";
let answer3 = 'He is called "President "';
```

String Length:

To find the length of a string, use the built-in **length** property:

Ex:

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

let length = text.length;
```

Escape Character:

Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

JAVASCRIPT CALLBACKS

Function Sequence:

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

This example will end up displaying "Goodbye":

Ex:

```
function myFirst() {
  myDisplayer("Hello");
}
function mySecond() {
  myDisplayer("Goodbye");
}
myFirst();
mySecond();
```

Sequence Control:

Sometimes you would like to have better control over when to execute a function.

Suppose you want to do a calculation, and then display the result.

You could call a calculator function (myCalculator), save the result, and then call another function (myDisplayer) to display the result:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2) {
  let sum = num1 + num2;
  return sum;
}
let result = myCalculator(5, 5);
myDisplayer(result);
```

JavaScript Callbacks:

A callback is a function passed as an argument to another function.

Using a callback, you could call the calculator function (myCalculator) with a callback (myCallback), and let the calculator function run the callback after the calculation is finished:

Ex:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
```

In the example above, **myDisplayer** is a called a callback function.

It is passed to **myCalculator()** as an argument.

EX:

```
// Create an Array
const myNumbers = [4, 1, -20, -7, 5, 9, -6];
// Call removeNeg with a callback
const posNumbers = removeNeg(myNumbers, (x) => x >= 0);
// Display Result
document.getElementById("demo").innerHTML = posNumbers;
// Keep only positive numbers
function removeNeg(numbers, callback) {
  const myArray = [];
  for (const x of numbers) {
    if (callback(x)) {
      myArray.push(x);
    }
  }
  return myArray;
}
```

Asynchronous JavaScript

Functions running in parallel with other functions are called asynchronous.

syntax of callback functions:

```
function myDisplayer(something) {
  document.getElementById("demo").innerHTML = something;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
```

Waiting for a Timeout

When using the JavaScript function setTimeout(), you can specify a callback function to be executed on time-out:

Ex:

```
setTimeout(myFunction, 3000);

function myFunction() {
  document.getElementById("demo").innerHTML = "Hello World";
}
```

3000 is the number of milliseconds before time-out, so myFunction() will be called after 3 seconds.

Waiting for Intervals:

When using the JavaScript function **setInterval(),** you can specify a callback function to be executed for each interval:

Ex:

```
setInterval(myFunction, 1000);

function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
  d.getHours() + ":" +
  d.getMinutes() + ":" +
  d.getSeconds();
}
```

In the example above, myFunction is used as a callback.

myFunction is passed to setInterval() as an argument.

1000 is the number of milliseconds between intervals, so myFunction() will be called every second.

JavaScript Promise Object:

A JavaScript Promise object contains both the producing code and calls to the consuming code

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 */ }
);
```

Promise Object Properties:

A JavaScript Promise object can be:

- Pending
- Fulfilled
- Rejected

The Promise object supports two properties: **state** and **result**.

While a Promise object is "pending" (working), the result is undefined.

When a Promise object is "fulfilled", the result is a value.

When a Promise object is "rejected", the result is an error object.

Promise Examples:

To demonstrate the use of promises, use these callback examples:

- Waiting for a Timeout
- Waiting for a File

Waiting for a Timeout (Using Callback)

```
setTimeout(function() { myFunction("Hello World!!!"); }, 3000);

function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
```

(Using Promise)

```
let myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function() { myResolve("I love You !!"); }, 3000);
});
myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
```

Waiting for a file (using Callback)

```
function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myCallback(req.responseText);
    } else {
      myCallback("Error: " + req.status);
    }
  }
  req.send();
}
getFile(myDisplayer);
```

(using Promise)

```
let myPromise = new Promise(function(myResolve, myReject) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.htm");
  req.onload = function() {
    if (req.status == 200) {
      myResolve(req.response);
    } else {
      myReject("File not Found");
    }
  };
  req.send();
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
```

TYPESCRIPT FUNDAMENTALS: