JavaScript Tutorial: Basics to Advanced Concepts

1. JavaScript Fundamentals

- **Writing JavaScript code in an HTML page**: We must enclose the JavaScript code within the <script> tag in order to include it on an HTML page just as the example shown below:

```
<script type = "text/javascript">
//JavaScript coding can be done inside this tag
</script>
```

With this information, the browser can correctly make out that the code is written in JavaScript and execute the code.

- **Inclusion of external JavaScript files in an HTML file**: An external JavaScript file can also be written separately and included within our HTML file. That way, different types of code can be kept isolated from one another, resulting in better-organised files. For instance, if our JavaScript code is written in the file script.js, we can include it in our HTML file in the following way:

```
<script src="script.js"></script>
```

- **Usage of Comments in JavaScript coding:** Comments are extremely useful in programming because they can assist others to understand what's going on in your code or they can help you if you have forgotten something. Keep in mind that they must be correctly identified so that the browser does not attempt to execute them. There are two alternatives available in JavaScript in which we can add comments:
  - **Single-line comments:** If you want to include a single line comment, start it with "//".
  - **Multi-line comments**: If you want to write a comment that spans multiple lines, you can wrap it in /* and */ to prevent it from being executed.

2. Javascript Variables

Variables in JavaScript are simply names of storage locations. In other words, they can be considered as stand-in values that we can use to perform various operations in our JavaScript codes. JavaScript allows the usage of variables in the following three ways:

- **var**: The most commonly used variable in JavaScript is var. It can be redeclared and its value can be reassigned, but only inside the context of a function. When the JavaScript code is run, variables defined using var are moved to the top. An example of a variable declared using the "var" keyword in JavaScript is shown below:

```
var x = 140; //  variable x can be reassigned a new value and also redeclared
```

- **const**: const variables in JavaScript cannot be used before they appear in the code. They can neither be reassigned values, that is, their value remains fixed throughout the execution of the code, nor can they be redeclared. An example of a variable declared using the "const" keyword in JavaScript is shown below:

```
const x = 5; // variable x cannot be reassigned a new value or redeclared
```

- **let**: The let variable, like const, cannot be redeclared. But they can be reassigned a value. An example of a variable declared using the "let" keyword in JavaScript is shown below:

```
let x = 202; // variable x cannot be redeclared but can be reassigned a new value
```

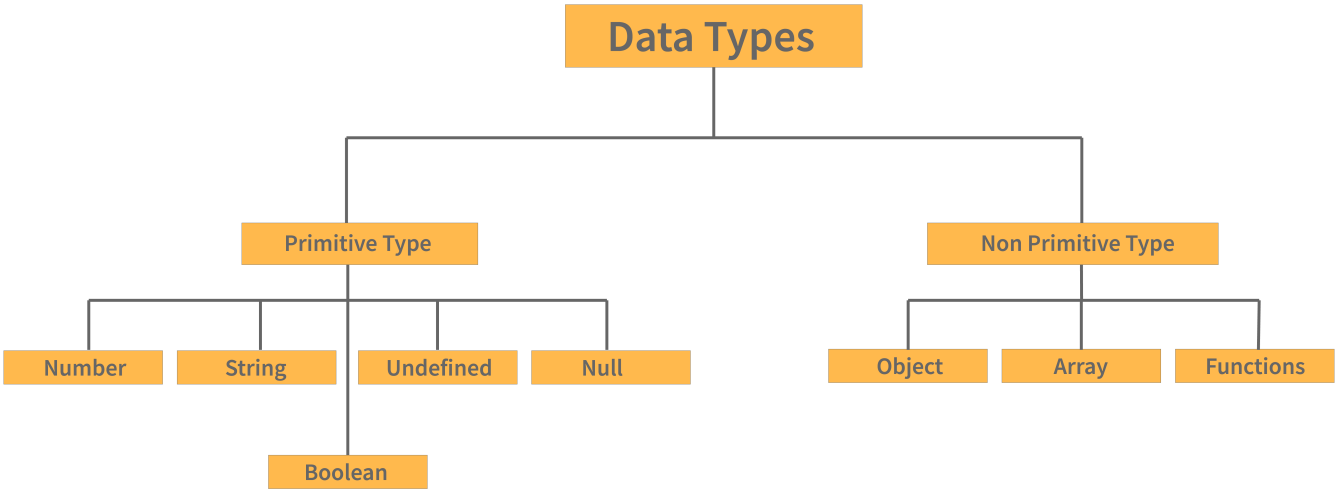
3. Javascript Data Types

Different types of values and data can be stored in JavaScript variables. To assign values to JavaScript variables, you use the equals to "=" sign operator. The various data types in JavaScript are as follows:

- Numbers: These are just numerical values. They can be real numbers or integers. An example of the numbers data type is shown below: `var id = 100`
- Variables: The variable data type, as the name suggests, does not have a fixed value. An example of the variables data type

is shown below: `var y`

- Text (strings): String data types of JavaScript are a combination of multiple characters. An example of the string data type is shown below: `var demoString = "Hello World"`
- Operations: Operations in JavaScript can also be assigned to JavaScript variables. An example of these is shown below: `var sum = 20 + 30 + 29`
- Boolean values: Boolean values can be true or false. An example of the boolean data type is shown below: `var booleanValue = true`
- Constant numbers: As the name suggests, these data types have a fixed value. An example of the constant data type is shown below: `const g = 9.8`
- Objects: JavaScript objects are containers for named values called properties. They possess their own data members and methods. An example of the objects data type is shown below:
`var name = {name:"Jon Snow", id:"AS123"}`



InterviewBit

> *Note: It is important to remember that variables in JavaScript are case-sensitive. As a result, "small" and "Small" will be treated as separate variables in JavaScript.*

You can download a PDF version of Javascript Cheat Sheet. **Download PDF**

4. JavaScript Operators

You can use variables to conduct a variety of tasks using JavaScript operators. The various types of operators in JavaScript are as follows:

- **Fundamental Operators**: These operators are used to perform basic operations like addition, multiplication, etc. in JavaScript. A list of all the Fundamental operators in JavaScript is as follows:
  - **+**: The Addition Operator is used to add two numbers
  - **-**: The Subtraction Operator is used to subtract two numbers
  - *****: The MultiplicationOperator is used to multiply two numbers
  - **/**: The Division Operator is used to divide two numbers
  - **(...)**: In general, operations within brackets are executed earlier than that outside. This grouping operator surrounds an expression or sub-expression with a pair of parentheses to override the conventional operator precedence, allowing operators with lower precedence to be evaluated before operators with higher precedence. It does exactly what it says: it groups the contents of the parentheses.
  - **%**: The Modulus operator is used to get the remainder when an integer number is divided by another integer number.
  - **++**: The Increment operator is used to increase the value of numbers by one.

    - **--**: The Decrement operator is used to decrease the value of numbers by one.
- **Bitwise Operators**: All the operations dealing with the bits of the numbers can be performed by the bitwise operators in JavaScript. A list of all the Bitwise operators in JavaScript is as follows:
    - **&**: The bitwise AND operator returns a 1 in every bit position where both operands' corresponding bits are 1.
    - **|**: The bitwise OR operator (|) returns a 1 in each bit position where either or both operands' corresponding bits are 1.
    - **~**: The bitwise NOT operator reverses the operand's bits. It turns the operand into a 32-bit signed integer, just like other bitwise operators.
    - **^**: The bitwise XOR operator () returns a 1 in each bit position where the corresponding bits of both operands are 1s but not both.
    - **<<**: The left shift operator shifts the first operand to the left by the provided number of bits. Extra bits that have been relocated to the left are discarded. From the right, zero bits are shifted in.
    - **>>**: The right shift operator (>>) moves the first operand to the right by the provided number of bits. Extra bits that have been relocated to the right are discarded. The leftmost bit's copies are shifted in from the left. The sign bit (the leftmost bit) does not change since the new leftmost bit has the same value as the old leftmost bit. As a result, the term "sign-propagating" was coined.
- **Comparison Operators:** A list of all the Comparison operators in JavaScript is as follows:
    - **==**: The equality operator (==) returns a Boolean value when its two operands are equal. It tries to convert and compare operands of different kinds, unlike the rigorous equivalent operator.
    - **===**: The equivalent operator (===) returns a Boolean value when its two operands are equal and they have the same type. It tries to convert and compare operands of the same kinds, unlike the equality operator.
    - **!=**: The inequality operator (!=) returns a Boolean value if the two operands are not equal. It tries to convert and compare operands of different kinds, unlike the rigorous inequivalent operator.
    - **!==**: The inequivalent operator (!=) returns a Boolean value if the two operands are not equal or they are not of the same type. It tries to convert and compare operands of the same kinds, unlike the inequality operator.
    - **?**: The conditional (ternary) operator is the only one in JavaScript that takes three operands: a condition followed by a question mark (?), an expression to execute if the condition is true followed by a colon (:), and lastly an expression to execute if the condition is false. As an alternative to an if...else statement, this operator is commonly used.>: The Greater than operator returns true if the operand to its left is greater in value than the operand to its right.
    - **<**: The Lesser than operator returns true if the operand to its left is lesser in value than the operand to its right.
    - **>=**: The Greater than equals to operator returns true if the operand to its left is greater in value or equal in value than the operand to its right.
    - **<=**: The Lesser than equals to operator returns true if the operand to its left is lesser in value or equal in value than the operand to its right.
- **Logical Operators**: A list of all the Logical operators in JavaScript is as follows:
    - **&&**: If and only if all of the operands are true, the logical AND operator (logical conjunction) for a set of Boolean operands is true. It will be false if it is not. When evaluating from left to right, the operator returns the value of the first falsy operand encountered, or the value of the last operand if all operands are truth.
    - **||**: If and only if one or more of its operands are true, the logical OR operator (logical disjunction) is true for a set of operands. It's most often used with logical (Boolean) values. It returns a Boolean value when this is the case. The || operator, on the other hand, returns the value of one of the provided operands, hence using it with non-Boolean values will result in an error.
    - **!**: The logical NOT operator (logical complement, negation) converts truth to falsity. It's most commonly used with logical (Boolean) values. It returns false if its sole operand can be transformed to true when used with non-Boolean values; otherwise, it returns true.

5. JavaScript If-Else Statements

The if-else statements are simple to comprehend. You can use them to set conditions for when your code runs. If specific requirements are met, something is done; if they are not met, another action is taken. The switch statement is a concept that is comparable to if-else. The switch however allows you to choose which of several code blocks to run. The syntax of if-else statements in JavaScript is given below:

```
if (check condition) {
   // block of code to be executed if the given condition is satisfied
} else {
   // block of code to be executed if the given condition is not satisfied
}
```

Loops in JavaScript:

Most programming languages include loops. They let you run code blocks as many times as you like with different values. Loops can be created in a variety of ways in JavaScript:

- **the for loop**: The most frequent method of creating a loop in JavaScript. Its syntax is shown below:

```
for (initialization of the loop variable; condition checking for the loop; updation after the loop) {
   // code to be executed in loop
}
```

- **the while loop**: Establishes the conditions under which a loop will run. Its syntax is shown below:

```
// Initialization of the loop variable is done before the while loop begins
while(condition checking for the loop){
// 1. code to be executed in loop
// 2. updation of the loop variable
}
```

- **the do-while loop**: Similar to the while loop, but it runs at least once and checks at the end to see whether the condition is met to run again. Its syntax is shown below:

```
// Initialization of the loop variable is done before the do-while loop begins
do{
// 1. code to be executed in loop
// 2. updation of the loop variable
}while(condition checking for the loop);
```

There are two statements that are important in the context of loops:

- **the continue statement**: Skip parts of the loop if certain conditions are met.
- **break statement:** Used to stop and exit the cycle when specific conditions are met.

6. JavaScript Arrays

Arrays are the next item on our JavaScript cheat sheet. Arrays are used in a variety of programming languages. They are a method of categorising variables and attributes. Arrays can be defined as a collection of objects of the same type. In JavaScript, here's how one can make an array of cars:

```
var cars = {"Mercedes", "Tesla","Volvo"};
```

Now that we understand how to make arrays, we can perform a bunch of operations on them. Let us take a look at some JavaScript methods which can be used to perform various types of operations on arrays:

- **pop():** This method is used for removing the last element of an array.
- **push():** This method is used for adding a new element at the very end of an array.
- **concat():** This method is used for joining various arrays into a single array.
- **reverse():** This method is used for reversing the order of the elements in an array.
- **shift():** This method is used for removing the first element of an array.
- **slice():** This method is used for pulling a copy of a part of an array into a new array.
- **splice()**: This method is used for adding elements in a particular way and position.
- **toString():** This method is used for converting the array elements into strings.
- **unshift():** This method is used for adding new elements at the beginning of the array.

- **valueOf():** This method is used for returning the primitive value of the given object.
- **indexOf():** This method is used for returning the first index at which a given element is found in an array.
- **lastIndexOf():** This method is used for returning the final index at which a given element appears in an array.
- **join():** This method is used for combining elements of an array into one single string and then returning it.
- **sort():** This method is used for sorting the array elements based on some condition.

7. JavaScript Functions

JavaScript Functions can be defined as chunks of code written in JavaScript to perform a single task. A function in JavaScript looks like this:

```
function nameOfTheFunction(parameterOne, parameterTwo, parameterThree, parameterFour,....,parameterN) {
   // Job or Task of the function
}
```

The code above consists of the "function" keyword and a name, as you can see. The parameters of the function are enclosed in brackets, while the function's task code and output is enclosed in curly brackets. You can make your own, but there are a few default functions to make your life easier. Although we will be discussing various methods throughout this cheat sheet, let us discuss in brief two important types of JavaScript functions in this section:

- **Functions For Throwing Data As Output**: The output of data is a common application for functions. You have the following options for outputting data:
  - **prompt()**: This function is used for creating a dialogue box for taking input from the user.
  - **alert()**: This function is used for outputting information in an alert box in the browser window
  - **console.log()**: This function is used for writing data to the browser's console and is used for the purpose of debugging code by developers.
  - **document.write()**: This function is used for writing straight to our HTML document
  - **confirm()**: This function is used for opening up a yes or no dialogue box and for returning a boolean value depending upon the user's click
- **Global Functions:** Every browser that can run JavaScript has a set of global functions built-in. Some of them are as follows:
  - **parseFloat():** This function is used for parsing the argument passed to it and returning a floating-point number.
  - **parseInt():** This function is used for parsing the argument passed to it and returning an integral number.
  - **encodeURI():** This function is used for encoding a URI into a UTF-8 encoding scheme.
  - **decodeURI():** This function is used for decoding a Uniform Resource Identifier (URI) made by encodeURI() function or similar functions.
  - **encodeURIComponent():** This function is used for the same purpose as encodeURI() only for URI components.
  - **decodeURIComponent():** This function is used for decoding a URI component.
  - **isNaN():** This function is used for determining if a given value is Not a Number or not.
  - **Number():** This function is used for returning a number converted from what is passed as an argument to it.
  - **eval():** This function is used for evaluating JavaScript programs presented as strings.
  - **isFinite():** This function is used for determining if a passed value is finite or not.

8. Scope and Scope Chain in JavaScript

1. Scope: The accessibility or visibility of variables in JavaScript is referred to as scope. That is, which sections of the program can access a given variable and where the variable can be seen. There are usually three types of scopes:

- **Global Scope:** The global scope includes any variable that is not contained within a function or block (a pair of curly braces). Global scope variables can be accessed from anywhere in the program. An example showing the global scope of a variable is given below:

```
var hello = 'Hello!';
function sayHello() {
 console.log(hello);
}
// 'Hello!' gets logged
sayHello();
```

- **Local or Function Scope:** Variables declared inside a function are local variables. They can only be accessed from within that function; they are not accessible from outside code. An example showing local scope of a variable is given below:

```
function sayHello() {
 var hello = 'Hello!';
 console.log(hello);
}
// 'Hello!' gets logged
sayHello();
```

console.log(hello); // Uncaught ReferenceError: hello is not defined

- **Block Scope**: Unlike var variables, let and const variables can be scoped to the nearest pair of curly brackets in ES6. They can't be reached from outside that pair of curly braces, which means they can't be accessed from the outside. An example showing the block scope of a variable is given below:

```
{
 let hello = 'Hello!';
 var language = 'Hindi';
 console.log(hello); // 'Hello!' gets logged
}
console.log(language); // 'Hindi!' gets logged
console.log(hello); // Uncaught ReferenceError: hello is not defined
```

2. Scope Chain: When a variable is used in JavaScript, the JavaScript engine searches the current scope for the variable's value. If it can't find the variable in the inner scope, it will look into the outer scope until it finds it or reaches the global scope.

If it still can't identify the variable, it will either return an error or implicitly declare the variable in the global scope (if not in strict mode). Let us take into consideration the following example:

```
let a = 'a';
function foo() {
 let b = 'b';
 console.log(b); // 'b' gets logged
 console.log(a); // 'a' gets logged
 randomNumber = 33;
 console.log(randomNumber);  // 33 gets logged
}
foo();
```

When the function foo() is called, the JavaScript engine searches for the 'b' variable in the current scope and finds it. Then it looks for the 'a' variable in the current scope, which it can't find, so it moves on to the outer scope, where it finds it (i.e global scope).

After that, we assign 33 to the 'randomNumber' variable, causing the JavaScript engine to search for it first in the current scope, then in the outer scope.

If the script isn't in strict mode, the engine will either create a new variable called randomNumber and assign 33 to it, or it will return an error (if not in strict mode).

As a result, the engine will traverse the scope chain till the time when a variable is found.

9. JavaScript Hoisting

Prior to executing the code, the interpreter appears to relocate the declarations of functions, variables, and classes to the top of

their scope using a process known as Hoisting in JavaScript. Functions can be securely utilised in code before they have been declared thanks to hoisting. Variable and class declarations are likewise hoisted, allowing them to be referenced prior to declaration. It should be noted that doing so can result in unforeseen mistakes and is not recommended. There are usually two types of Hoisting:

- **Function Hoisting**: Hoisting has the advantage of allowing you to use a function before declaring it in your code as shown in the code snippet given below. Without function hoisting, we would have to first write down the function display and only then can we call it.

```
display("Lion");
function display(inputString) {
 console.log(inputString); // 'Lion' gets logged
}
```

- **Variable Hoisting:** You can use a variable in code before it is defined and/or initialised because hoisting works with variables as well. JavaScript, however, only hoists declarations, not initializations! Even if the variable was initially initialised then defined, or declared and initialised on the same line, initialization does not occur until the associated line of code is run. The variable has its default initialization till that point in the execution is reached (undefined for a variable declared using var, otherwise uninitialized). An example of variable hoisting is shown below:

```
console.log(x) // 'undefined' is logged from hoisted var declaration (instead of 7)
var x // Declaration of variable x
x = 7; // Initialization of variable x to a value 7
console.log(x); // 7 is logged post the line with initialization's execution.
```

10. JavaScript Closures

A closure is a function that has been bundled together (enclosed) with references to its surroundings (the lexical environment). In other words, a closure allows an inner function to access the scope of an outside function. Closures are formed every time a function is created in JavaScript, during function creation time. An example of closures in Javascript is given below:

```
function subtractor(subtractingInteger) {
 return function(a) {
   return a - subtractingInteger;
 };
}
var subtract2 = subtractor(2);
var subtract5 = subtractor(5);
console.log(subtract2(5));  // 3 is logged
console.log(subtract5(5)); // 0 is logged
```

In this example, we have developed a function subtractor(subtractingInteger) that takes a single parameter subtractingInteger and returns a new function. Its return function accepts only one input, a, and returns the difference of a and subtractingInteger. The function 'subtractor' is essentially a function factory. It creates functions that have the ability to subtract a specified value from their arguments. The function factory creates two new functions in the example above: one that subtracts 2 from its argument and one that subtracts 5 from its arguments. Both subtract2 and subtract5 are closures. They have the same function body definition, but they hold lexical surroundings that are distinct. subtractingInteger is 2 in subtract2's lexical environment, but subtractingInteger is 5 in subtract5's lexical environment.

11. JavaScript Strings

As mentioned earlier, Strings are nothing but a combination of characters that can be used to perform a variety of tasks. JavaScript provides so many methods for Strings alone that it makes sense to cover Strings as a standalone topic in this cheat sheet. Let us now look at the various escape sequences in JavaScript and the methods which JavaScript provides for strings:

- **Escape Sequences or Escape Characters:** An escape character is a character in computers and telecommunications that causes the following characters in a character sequence to take on a different meaning. Metacharacters include escape characters, which are a subset of metacharacters. In general, whether something is an escape character or not is determined by the context. For instance, Strings in JavaScript are delimited by single or double-quotes. You must use special characters in a string if you want to utilise quote marks. A few of the escape characters allowed by JavaScript are as follows:

    - **\'** — Single quotes
    - **\"** — Double quotes
    - **\t** — Horizontal tab
    - **\v** — Vertical tab
    - **** — Backslash
    - **\b** — Backspace
    - **\f** — Form feed
    - **\n** — Newline
    - **\r** — Carriage return
- **String methods:** As mentioned earlier, JavaScript provides a lot of methods to manipulate its Strings. Let us take a look at some of them:
    - **toLowerCase()** — This method is used for converting strings to lower case
    - **toUpperCase()** — This method is used for converting strings to upper case
    - **charAt()** — This method is used for returning the character at a particular index of a string
    - **charCodeAt()** — This method is used for returning to us the Unicode of the character at a given index
    - **fromCharCode()** — This method is used for returning a string made from a particular sequence of UTF-16 code units
    - **concat()** — This method is used for concatenating or joining multiple strings into a single string
    - **match()** — This method is used for retrieving the matches of a string against a pattern string which is provided
    - **replace()** — This method is used for finding and replacing a given text in the string
    - **indexOf()** — This method is used for providing the index of the first appearance of a given text inside the string
    - **lastIndexOf()** — This method is similar to the indexOf() methods and only differs in the fact that it searches for the last occurrence of the character and searches backwards
    - **search()** — This method is used for executing a search for a matching text and returning the index of the searched string
    - **substr()** — This method is pretty much the same as the slice() method but the extraction of a substring in it depends on a given number of characters
    - **slice()** — This method is used for extracting an area of the string and returning it
    - **split()** — This method is used for splitting a string object into an array of strings at a particular index
    - **substring()** — Even this method is almost the same as the slice() method but it does not allow negative positions
    - **valueOf()** — This method is used for returning the primitive value (one without any properties or methods) of a string object

12. Document Object Model (DOM) in JavaScript

The Document Object Model (DOM) is the structure of a webpage's code. There are many different ways to build and alter HTML elements with JavaScript (called nodes).

- **Node Properties:** Let us first take a look at some of the properties of a JavaScript DOM node:
    - **attributes** — Gets a live list of all the characteristics associated with an element.
    - **baseURI** — Returns an HTML element's absolute base URL.
    - **childNodes** — Returns a list of the child nodes of an element.
    - **firstChild** — Returns the element's first child node.
    - **lastChild** — An element's final child node
    - **nextSibling** — Returns the next node in the same node tree level as the current node.
    - **nodeName** —Returns a node's name.
    - **nodeType** — Returns the node's type.
    - **nodeValue** — Sets or returns a node's value.
    - **ownerDocument** — This node's top-level document object.
    - **parentNode** — Returns the element's parent node.
    - **previousSibling** — Gets the node that comes before the current one.
    - **textContent** — Sets or returns a node's and its descendants' textual content.

- **Node Methods:** Let us now take a look at some of the methods provided by JavaScript to manipulate these nodes in the DOM:
  - **appendChild()** — Adds a new child node as the last child node to an element.
  - **cloneNode()** is a function that duplicates an HTML element.
  - **compareDocumentPosition()** — Compares two elements' document positions.
  - **getFeature()** returns an object that implements the APIs of a feature.
  - **hasAttributes()** — If an element has any attributes, it returns true; otherwise, it returns false.
  - **hasChildNodes()** — If an element has any child nodes, it returns true; otherwise, it returns false.
  - **insertBefore()** — Adds a new child node to the left of an existing child node.
  - **isDefaultNamespace()** returns true if a given namespaceURI is the default, and false otherwise.
  - **isEqualNode()** — Determines whether two elements are the same.
  - **isSameNode()** — Determines whether two elements belong to the same node.
  - **isSupported()** — Returns true if the element supports the provided feature.
  - **lookupNamespaceURI()** — Returns the namespace URI for a specific node.
  - **lookupPrefix** — If the prefix for a given namespace URI is present, lookupPrefix() returns a DOMString containing it.
  - **normalise()** — In an element, joins neighbouring text nodes and removes empty text nodes.
  - **removeChild()** — Removes a child node from an element using the Child() method.
  - **replaceChild()** — In an element, this function replaces a child node.
- **Element Methods:** Given below are some of the element methods provided by JavaScript:
  - **getAttribute()** — Returns the value of an element node's provided attribute.
  - **getAttributeNS()** returns the string value of an attribute with the namespace and name supplied.
  - **getAttributeNode()** — Returns the attribute node supplied.
  - **getAttributeNodeNS()** — Returns the attribute node for the specified namespace and name for the attribute.
  - **getElementsByTagName()** — Returns a list of all child elements whose tag name is supplied.
  - **getElementsByTagNameNS()** — Returns a live HTMLCollection of items belonging to the provided namespace with a certain tag name.
  - **hasAttribute()** — If an element has any attributes, it returns true; otherwise, it returns false.
  - **hasAttributeNS()** returns true or false depending on whether the current element in a particular namespace has the supplied attribute.
  - **removeAttribute()** — Removes an element's supplied attribute.
  - **removeAttributeNS()** — Removes an attribute from an element in a specific namespace.
  - **setAttributeNode()** — Sets or modifies an attribute node.
  - **setAttributeNodeNS()** — Sets a new namespaced attribute node to an element with setAttributeNodeNS().

13. JavaScript Data Transformation

Data Transformation in JavaScript can be done with the usage of higher-order functions. Higher-order functions are those functions in JavaScript which can accept one or more functions as inputs and return a function as the result. All higher-order functions that take a function as input are map(), filter(), and reduce(). Let us now take a look at how these functions can be used. One thing to note over here is that because all of these functions are part of the JavaScript Array prototype, they can be used directly on an array.

map() method: The map method applies a function to each array element. The callback function receives each element of the array and returns a new array of the same length. This method can be used to conduct the same operation/transformation on each element of an array and return a new array with the modified values of the same length. An example of the usage of the map() method is given below:

```
var arr = [10,20,30];
var triple  = arr.map(x => x * 3);
triple; // [30,60,90]
```

filter() method: Using the filter() method, items that do not meet a criterion are removed from the array. The callback function

receives every element of the array. If the callback returns true on each iteration, the element will be added to the new array; otherwise, it will not be added. An example of the usage of the filter() method is given below:

```
var arr = [13,40,47];
var odd = arr.filter(x => x % 2);
odd; // [13,47]
```

reduce() method: The reduce() method can combine the items of an array into a single value. When using reduce, we must declare the accumulator's beginning value (final result). Each iteration, we do some operation inside the callback, which is then added to the accumulator. An example of the usage of the reduce() method is given below:

```
var arr = [10,20,30];
var counter = 0;
let answer = arr.reduce((accumulator, value) => value + accumulator, counter);
console.log(answer) // answer = 10 + 20 + 30 = 60
```

14. JavaScript Regular Expressions

Regular expressions can be defined as search patterns that can be used to match string character combinations. Text search and text to replace procedures can both benefit from the search pattern. Let us look at how JavaScript allows Regular Expressions:

- **Pattern Modifiers**: Following are the pattern modifiers that are allowed in JavaScript:
  - **e** — This is used for evaluating replacement
  - **i** — This is used for performing case-insensitive matching
  - **U** — This is used for ungreedy pattern
  - **g** — This is used for performing global matching
  - **m** — This is used for performing multiple line matching
  - **s** — This is used for treating strings as a single line
  - **x** — This is used for allowing comments and whitespace in the pattern
- **Metacharacters:** Following are the metacharacters that are allowed in JavaScript:
  - **.** — This is used for finding a single character, except newline or line terminator
  - **\w** — This is used for finding Word characters
  - **\W** — This is used for finding Non-word characters
  - **\s** — This is used for finding Whitespace characters
  - **\S** — This is used for finding Non-whitespace characters
  - **\b** — This is used for finding matches at the beginning or at the end of a word
  - **\B** — This is used for finding matches not at the beginning or at the end of a word
  - **\0** — This is used for finding NULL characters
  - **\n** — This is used for finding a new line character
  - **\f** — This is used for finding a Form feed character
  - **\r** — This is used for finding a Carriage return character
  - **\t** — This is used for finding a Tab character
  - **\v** — This is used for finding a Vertical tab character
  - **\d** — This is used for finding digits
  - **\D** — This is used for finding non-digit characters
  - **\xxx** — This is used for finding characters given by an octal number xxx
  - **\xdd** — This is used for finding characters given by a hexadecimal number dd
  - **\uxxxx** — This is used for finding the Unicode character given by a hexadecimal number XXXX
- **Brackets:** You can group parts of a regular expression together by putting them inside round brackets or parentheses. You can use this to apply a quantifier to the entire group or to limit the alternation to a specific area of the regex. For grouping, only parenthesis can be used. A character class is defined by square brackets, while a quantifier with precise bounds is defined by curly braces. Let us look at some of the brackets which JavaScript allows:

- **[abc]** — This is used for finding all the characters between the brackets
- **(a|b|c)** — This is used for finding all of the alternatives separated with |
- **[^abc]** — This is used for finding every character that is not in the brackets
- **[0-9]** — This is used for finding each digit from 0 to 9
- **[A-z]** — This is used for finding each character from uppercase A to lowercase z

- **Quantifiers:** Quantifiers provide the minimum number of instances of a character, group, or character class in the input required to find a match. The quantifiers supported by JavaScript are listed in the table below.
  - **n+** — This is used for matching each string which is having one or more n
  - **n*** — This is used for matching any string which is having zero or more occurrences of n
  - **n?** — This is used for matching strings which are having zero or one occurrence of n
  - **^n** — This is used for matching strings with n in the first place
  - **?=n** — This is used for matching all strings which are followed by a particular string n
  - **?!n** — This is used for matching strings that are not followed by a particular string ni
  - **n{X}** — This is used for matching strings that contain a sequence of X n's
  - **n{X,Y}** — This is used for matching a string that contains a sequence of X to Y n's
  - **n{X,}** — This is used for matching all strings which are having a sequence of X or more n's
  - **n$** — This is used for matching all strings having n at the end.

15. Numbers and Mathematics in JavaScript

JavaScript provides various properties and methods to deal with Numbers and Maths. Let us have a quick look at those:

- **Numbers Properties:**
  - **MAX VALUE** — The maximum numeric value that JavaScript can represent.
  - **NaN** — The "Not-a-Number" value is NaN.
  - **NEGATIVE INFINITY** – The value of Infinity is negative.
  - **POSITIVE INFINITY** – Infinity value that is positive.
  - **MIN VALUE** — The smallest positive numeric value that JavaScript can represent.
- **Numbers Methods:**
  - **toString()** — Returns a string representation of an integer.
  - **toFixed()** — Returns a number's string with a specified number of decimals.
  - **toPrecision()** — Converts a number to a string of a specified length.
  - **valueOf()** — Returns a number in its original form.
  - **toExponential()** — Returns a rounded number written in exponential notation as a string.
- **Maths Properties:**
  - **E** — Euler's number is E.
  - **SQRT1_2** — 1/2 square root
  - **SQRT2** stands for square root of two.
  - **LOG2E** — E's base 2 logarithm
  - **LN2** — The natural logarithm of 2 is LN2.
  - **LN10** — The natural logarithm of ten is LN10.
  - **LOG10E** — E's base ten logarithm
  - **PI** — PI stands for Pianistic Integer.
- **Maths Methods:**
  - **exp(x)** — Ex's value
  - **floor(x)** — x's value rounded to the nearest integer.
  - **log(x)** — The natural logarithm (base E) of x is log(x).
  - **abs(x)** — Returns the value of x in its absolute (positive) form.
  - **acos(x)** — In radians, the arccosine of x.
  - **asin(x)** — In radians, the arcsine of x.

- **pow(x,y)** — x to the power of y
- **random()** — Returns a number between 0 and 1 at random.
- **round(x)** — Rounds the value of x to the nearest integer.
- **sin(x)** — The sine of x is sin(x) (x is in radians)
- **sqrt(x)** — x's square root
- **tan(x)** — The angle's tangent
- **atan(x)** is the numeric value of the arctangent of x.
- **atan2(y,x)** — Arctangent of its arguments' quotient
- **ceil(x)** — x's value rounded to the next integer
- **cos(x)** – The cosine of x is cos(x) (x is in radians)
- **max(x,y,z,...,n)** — Returns the highest-valued number.
- **min(x,y,z,...,n)** — The number with the lowest value is the same as the number with the highest value.

16. JavaScript Date Objects

Dates are extremely important to deal with while creating most types of applications. JavaScript also allows you to deal with and change dates and times. The next section of the JavaScript cheat sheet is how to work on dates:

- **Setting Dates:** Dates can be set using the following three ways:
  - **Date()** — Returns a new date object that contains the current date and time.
  - **Date(1993, 6, 19, 5, 12, 50, 32)** — We can create a custom date object with the pattern as Year, month, day, hour, minutes, seconds, and milliseconds are represented by the numbers. Except for the year and month, we can omit anything we like.
  - **Date("1999-12-22")** — Date as a string declaration
- **Getting the values of Time and Date:** The following methods can be used for getting the date and time values in JavaScript:
  - **getDate()** returns the month's day as a number (1-31)
  - **getTime()** — Get the milliseconds since January 1, 1970
  - **getUTCDate()** returns the month's day (day) in the supplied date in universal time (also available for day, month, full year, hours, minutes etc.)
  - **getMilliseconds()** — This function returns the milliseconds (0-999)
  - **getMinutes()** — Returns the current minute (0-59)
  - **getMonth()** returns the current month as a number (0-11)
  - **parse** — It returns the number of milliseconds since January 1, 1970 from a string representation of a date.
  - **getDay()** returns a number representing the weekday (0-6)
  - **getFullYear()** returns the current year as a four-digit value (yyyy)
  - **getHours()** — Returns the current hour (0-23)
  - **getSeconds()** — Returns the second number (0-59)
- **Setting a Part of the Dates:** We can set a part of the dates in JavaScript using the following methods:
  - **setDate()** — Returns the current date as a number (1-31)
  - **setFullYear()** — This function sets the year (optionally month and day)
  - **setMonth()** — This function sets the month (0-11)
  - **setSeconds()** — This function sets the seconds (0-59)
  - **setTime()** — This function is used to set the time (milliseconds since January 1, 1970)
  - **setMinutes()** — This function sets the minutes (0-59)
  - **setUTCDate()** — According to universal time, sets the day of the month for a given date (also available for day, month, full-year, hours, minutes etc.)
  - **setHours()** — Changes the time (0-23)
  - **setMilliseconds()** — This function sets the milliseconds (0-999)

17. JavaScript Browser Objects

JavaScript is also capable of taking note of the user's browser activity and incorporating its properties into the code, in addition to HTML elements.

- **Given below is a list of Window properties that JavaScript can take into account:**
  - **history** — Provides the window's History object.
  - **innerHeight** — The content area of a window's inner height.
  - **innerWidth** — The content area's inner width.
  - **closed** — Returns true or false depending on whether or not a window has been closed.
  - **pageXOffset** — The number of pixels offset from the centre of the page. The current document has been horizontally scrolled.
  - **pageYOffset** — The number of pixels offset from the centre of the page. The document has been vertically scrolled.
  - **navigator** — Returns the window's Navigator object.
  - **opener** — Returns a reference to the window that created the window.
  - **outerHeight** — A window's total height, including toolbars and scrollbars.
  - **outerWidth** — A window's outside width, including toolbars and scrollbars.
  - **defaultStatus** — Changes or restores the default text in a window's status bar.
  - **document** — Returns the window's document object.
  - **frames** — All <iframe> elements in the current window are returned by frames.
  - **length** — Determine how many iframe> elements are in the window.
  - **location** — Returns the window's location object.
  - **name** — Sets or retrieves a window's name.
  - **parent** — The current window's parent window is called parent.
  - **screen** — Returns the window's Screen object.
  - **screenLeft** — The window's horizontal coordinate (relative to the screen)
  - **screenTop** — The window's vertical coordinate.
  - **self** — Returns the window that is currently open.
  - **status** — Changes or restores the text in a window's status bar.
  - **top** — Returns the browser window that is currently at the top of the screen.
  - **screenX** — Identical to screenLeft, but required by some browsers
  - **screenY** — Identical to screenTop, but required by some browsers
- **Given below are the JavaScript methods which can work on the user's browser window:**
  - **alert()** — Shows a message and an OK button in an alert box.
  - **setInterval()** — Calls a function or evaluates an expression at intervals defined by the user.
  - **setTimeout()** — After a specified interval, calls a function or evaluates an expression.
  - **clearInterval()** — Removes a timer that was started with setInterval() ()
  - **clearTimeout()** — Removes the timer that was set with setTimeout() ()
  - **open()** — This method creates a new browser window.
  - **print()** — Prints the current window's content.
  - **blur()** — Removes the current window's focus.
  - **moveBy()** — Repositions a window with respect to its present position.
  - **moveTo()** — This function moves a window to a specific location.
  - **close()** — This function closes the currently open window.
  - **confirm()** — Shows a dialogue box with a message and buttons to OK and Cancel.
  - **focus()** — Sets the current window's focus.
  - **scrollBy()** — Scrolls the document by a certain amount of pixels.
  - **scrollTo()** — Scrolls the document to the supplied coordinates with the scrollTo() method.
  - **prompt()** — Shows a conversation window asking for feedback from the visitor.
  - **resizeBy()** — Resizes the window by the number of pixels supplied.
  - **resizeTo()** — Resizes the window to the width and height supplied.

- **stop()** — This function prevents the window from loading.

- **Given below is a list of Screen properties that JavaScript can take into account:**
  - **height** — The screen's entire height.
  - **pixelDepth** — The screen's colour resolution in bits per pixel.
  - **width** — The screen's entire width.
  - **colorDepth** — Gets the colour palette's bit depth for showing images.
  - **availableHeight** — Returns the screen's height (excluding the Windows Taskbar).
  - **availableWidth** — Returns the screen's width (excluding the Windows Taskbar).

18. JavaScript Events

Things that can happen to HTML components and are executed by the user in JavaScript are called events. These events can be detected by the programming language, which can then be used to activate code actions. Without them, no JavaScript cheat sheet would be complete. Let us take a look at some of the events supported by JavaScript:

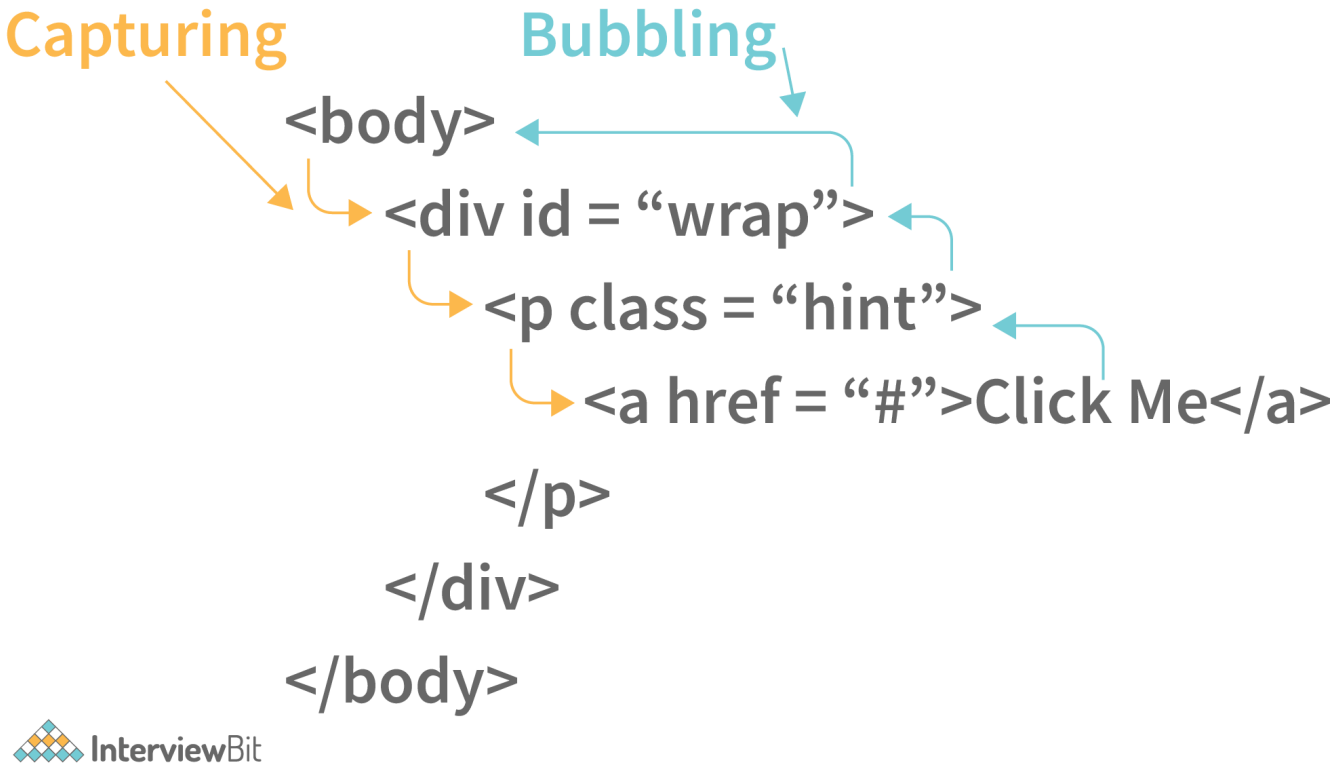
- **Mouse Events:**
  - **onclick** – When a user clicks on an element, an event is triggered.
  - **oncontextmenu** — When a user right-clicks on an element, a context menu appears.
  - **ondblclick** — When a user double-clicks on an element, it is called ondblclick.
  - **onmousedown** — When a user moves their mouse over an element, it is called onmousedown.
  - **onmouseenter** — The mouse pointer is moved to a certain element.
  - **onmouseleave** — The pointer leaves an element.
  - **onmousemove** — When the pointer is over an element, it moves.
  - **onmouseover** — When the cursor is moved onto an element or one of its descendants, the term onmouseover is used.
  - **onmouseout** — When the user moves the mouse cursor away from an element or one of its descendants, it is called onmouseout.
  - **onmouseup** — When a user releases a mouse button while hovering over an element, it is known as onmouseup.
- **Form Events:**
  - **onblur** — When an element loses focus, it is called onblur.
  - **onchange** — A form element's content changes. (for the input>, select>, and textarea> elements)
  - **onfocus** – An aspect is brought into focus.
  - **onfocusin** — When an element is ready to become the centre of attention.
  - **onfocusout** —The element is about to lose focus.
  - **oninput** — When a user inputs something into an element, it's called oninput.
  - **oninvalid** — If an element isn't valid, it's called oninvalid.
  - **onreset** — The state of a form is reset.
  - **onsearch** — A user enters text into a search field (for input="search">).
  - **onselect** — The user chooses some text (input> and textarea>).
  - **onsubmit** — A form is filled out and submitted.
- **Drag Events**:
  - **ondrag** — When an element is dragged, it is called ondrag.
  - **ondragend** — The element has been dragged to its final position.
  - **ondragenter** — When a dragged element enters a drop target, it is called ondragenter.
  - **ondragleave** — When a dragged element departs the drop target, it is called ondragleave.
  - **ondragover** — The dropped element is on top of the dragged element.
  - **ondragstart** — The user begins dragging an element.
  - **ondrop** — When a dragged element is dropped on a drop target, it is called ondrop.
- **Keyboard Events**:
  - **onkeydown** — When the user presses a key down
  - **onkeypress** — When the user begins to press a key.

    - **onkeyup** — A key is released by the user.
- **Frame Events**:
    - **onabort** — The loading of a media is aborted with onabort.
    - **onerror** — When loading an external file, an error occurs.
    - **onpagehide** – When a user leaves a webpage, it is called onpagehide.
    - **onpageshow** — When the user navigates to a webpage
    - **onhashchange** — The anchor component of a URL has been changed.
    - **onload** — When an object has loaded
    - **onresize** — The document view gets resized when onresize is called.
    - **onscroll** — The scrollbar of an element is being scrolled.onbeforeunload — Before the document is due to be emptied, an event occurs.
    - **onunload** — When a page is emptied, this event occurs.
- **Animation Events**:
    - **animationstart** — The animation in CSS has begun.
    - **animationend** — When a CSS animation is finished, it is called animationend.
    - **animationiteration** — CSS animation is repeated using animationiteration.
- **Clipboard Events:**
    - **oncut** — The content of an element is cut by the user.
    - **onpaste** — When a user pastes material into an element, it is called onpaste.
    - **oncopy** — The content of an element is copied by the user.
- **Media Events:**
    - **onloadeddata** — Media data has been loaded
    - **onloadedmetadata** — Metadata is loaded (such as size and duration).
    - **onloadstart** — The browser begins looking for the media that has been specified.
    - **onabort** — The loading of media has been halted.
    - **onerror** — When an error occurs while loading an external file, the event onerror is triggered.
    - **onpause** — Media is paused, either manually or automatically, by the user.
    - **onplay** — The video or audio has begun or is no longer paused.
    - **onstalled** — The browser is attempting to load the media, but it is not currently accessible.
    - **oncanplay** — The browser has the ability to begin playing media (e.g. a file has buffered enough)
    - **oncanplaythrough** — The browser is capable of playing media without pausing.
    - **ondurationchange** — The media's duration changes.
    - **onended** — The media's time has come to an end.
    - **onsuspend** — The browser is not loading media on purpose.
    - **ontimeupdate** — The situation has shifted (e.g. because of fast forward)
    - **onvolumechange** — The volume of the media has changed (including mute)
    - **onwaiting** — The media has taken a break, but it is anticipated to resume soon (for example, buffering)
    - **onplaying** — Media that has been paused or halted for buffering is now playing.
    - **onprogress** — The media is being downloaded by the browser.
    - **onratechange** — The media's playback speed changes.
    - **onseeking** — The user begins to move/skip.
- **Miscellaneous Events:**
    - **transitionend** — When a CSS transition is complete, this event is triggered.
    - **onmessage** — The event source has received a message.
    - **onpopstate** — When the history of the window changes
    - **onshow** — A <menu> element is shown as a context menu when it is onshow.
    - **onoffline** — The browser switches to offline mode.
    - **ononline** — The browser enters the online mode.
    - **ontouchcancel** — The user's ability to touch the screen has been halted.

- **ontouchstart** — The touch-screen is activated by placing a finger on it.
- **onstorage** — An part of Web Storage has been upgraded.
- **ontoggle** — The user toggles the details> element open or closed.
- **onwheel** — The mouse wheel moves up and down when it passes over an element.
- **ontouchend** — A touch-screen user's finger is withdrawn.
- **ontouchmove** — When a finger is dragged over the screen, it is called ontouchmove.

19. JavaScript Event Propagation

Event propagation is a technique that governs how events propagate or travel through the DOM tree to reach their destination, as well as what happens to them once they arrive. Consider the following scenario: you have been given a click event handler to a hyperlink (i.e. <a> element) that's nested inside a paragraph (i.e. <p> element). The handler will now be executed if you click on that link. However, if you set the click event handler to the paragraph containing the link instead of the link, the handler will be triggered regardless of whether the link is clicked. Because events go up and down the DOM tree to reach their target, they don't merely affect the target element that triggered the event. This is known as event propagation.



When an event is fired on an element with parent elements, the above picture shows how the event travels through the DOM tree at different stages of the event propagation. Event propagation in current browsers is divided into two phases: capturing and bubbling.

- **The Capturing Phase**: In the capturing phase, events propagate from the Window down through the DOM tree to the target node. For example, if the user clicks a hyperlink, that click event would pass through the <html> element, the <body> element, and the <p> element containing the link. Also if any ancestor (i.e. parent, grandparent, etc.) of the target element and the target itself has a specially registered capturing event listener for that type of event, those listeners are executed during this phase.
- **The Bubbling Phase:** From the target element up to the Window, the DOM tree visits all of the target element's ancestors one by one. When a user hits a hyperlink, the click event passes via the <p> element containing the link, the <body> element, the <html> element, and the document node, for example. Additionally, if the target element or any of its ancestors have event handlers for that sort of event, those handlers are run during this phase. By default, all event handlers in current browsers are registered at the bubbling phase.

20. Asynchronous JavaScript

A number of Web API features now use asynchronous code for running, especially those that access or fetch a resource from

external devices, for instance, retrieving files from the network, accessing some database and returning data to it, accessing a video stream from a webcam, or broadcasting the display to a VR headset. There are two ways in which asynchronous coding can be done in JavaScript:

- **Async Callbacks:** When invoking a function, async callbacks are functions that are passed as arguments and begin executing code in the background. When the background code is finished, it runs the callback function to notify you that the work is complete or that anything interesting has occurred. Callbacks are a little out of date these days, but they're still utilised in a lot of older but still popular APIs. The second parameter of the addEventListener() method is an example of an async callback:

```
buton.addEventListener('click', () => {
 alert('Button has been clicked!');
 let paraElement = document.createElement('p');
 paraElement.textContent = 'A new paragraph.';
 document.body.appendChild(paraElement);
});
```

The first parameter specifies the type of event to be listened for, while the second specifies a callback function to be called when the event occurs. When we give a callback function as an input to another function, we are merely passing the function's reference; the callback function isn't immediately performed. It is asynchronously "called back" (thus the name) somewhere within the containing function's body. When the time comes, the contained function is in charge of calling the callback function.

- **Promises:** Promises are a new async programming paradigm that you'll see in current Web APIs. The get() API, which is essentially a newer, more efficient version of XMLHttpRequest, is a nice example. Let's take a look at an example from our post Fetching data from the server:

```
fetch(items.json').then(function(res) {
 return res.json();
}).then(function(json) {
 let item = json;
 initialise(item);
}).catch(function(e) {
 console.log('Fetch error: ' + e.message);
});
```

Here, fetch() takes a single argument: the URL of a network resource we wish to fetch and a promise is returned. A promise is an object that represents whether the async operation succeeded or failed. In a sense, it reflects a transitional condition. In essence, it's the browser's way of saying, "I promise to respond as quickly as I can," hence the name "promise."

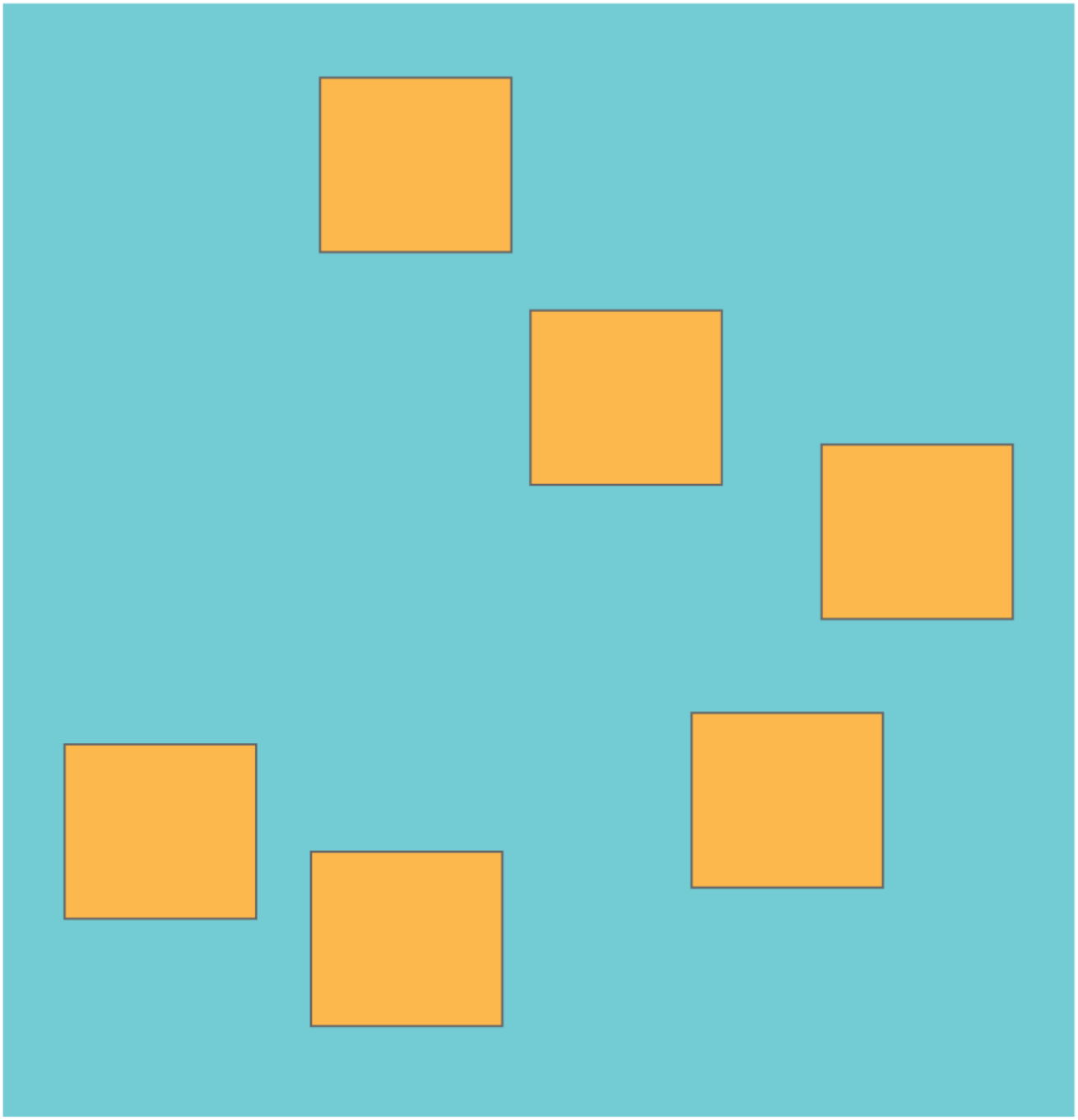
> *Note: Pyramid of Doom, also known as Callback Hell, is an anti-pattern found in asynchronous computer code. It is a slang phrase for a large number of interconnected "if" statements or functions. A few callbacks appear harmless if you do not expect your application logic to become too sophisticated. However, as your project's requirements grow, you will quickly find yourself with layers upon layers of nested callbacks. The usage of callbacks makes writing and maintaining code more challenging. It also makes it more difficult to identify the application's flow, which is a debugging roadblock, hence the problem's well-known nickname: Callback Hell. An example of callback hell is given below:*

```
fs.readdir(source, function (error, file) {
 if (error) {
   console.log(Problem occurred while finding file: ' + error)
 } else {
   file.forEach(function (filename, fileIndex) {
     console.log(filename)
     gm(source + filename).size(function (error, value) {
       if (error) {
         console.log(Problem occurred while identifying file size: ' + error)
       } else {
         console.log(filename + ' : ' + value)
         aspect = (values.width / values.height)
         widths.forEach(function (width, widthIndex) {
           height = Math.round(width / aspect)
           console.log('resizing ' + filename + 'to ' + height + 'x' + height)
           this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(error) {
             if (error) console.log('Problem occurred while writing file: ' + error)
           })
         }.bind(this))
       }
     })
   })
 }
})
```

21. JavaScript Memory Allocation and Event Loop

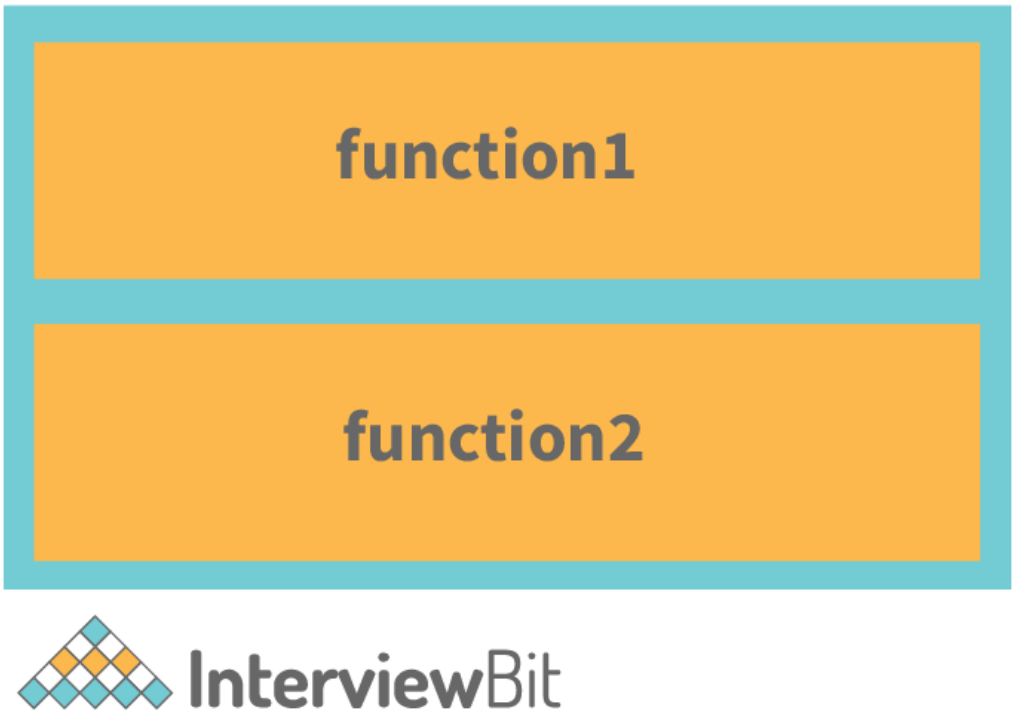
In JavaScript, memory allocation is done in the following regions:

- **Heap memory:** Data is stored in random order and memory is allocated accordingly.



InterviewBit

- **Stack memory:** Memory that is allocated in stacks. The majority of the time, it's employed for functions.



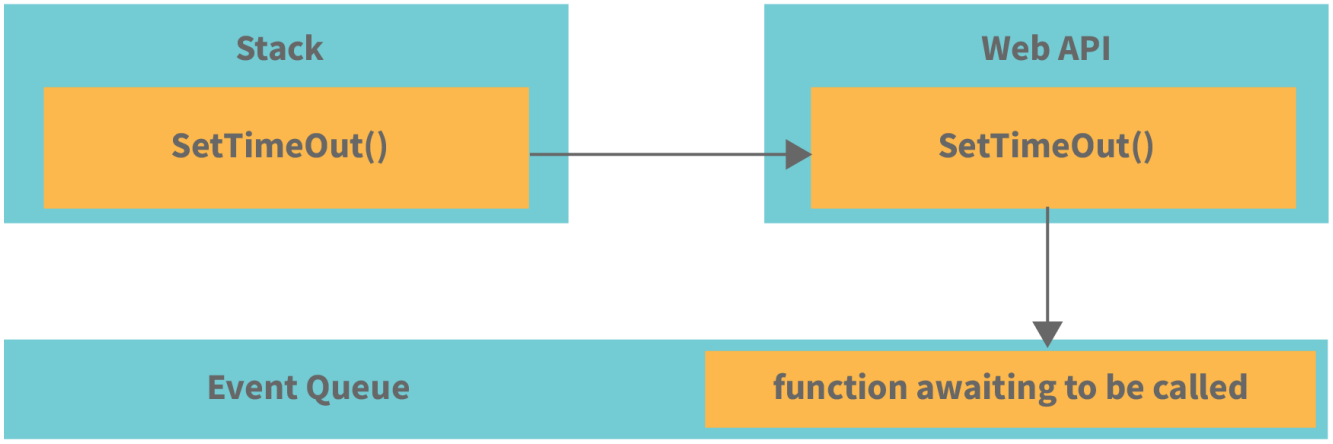
The function stack is a function that maintains track of all other functions that are running at the same time. An example to illustrate it is as follows:

```
function second() {
console.log("Second")
}
function First() {
second()
}
function foo() {
first()
}
foo()
```

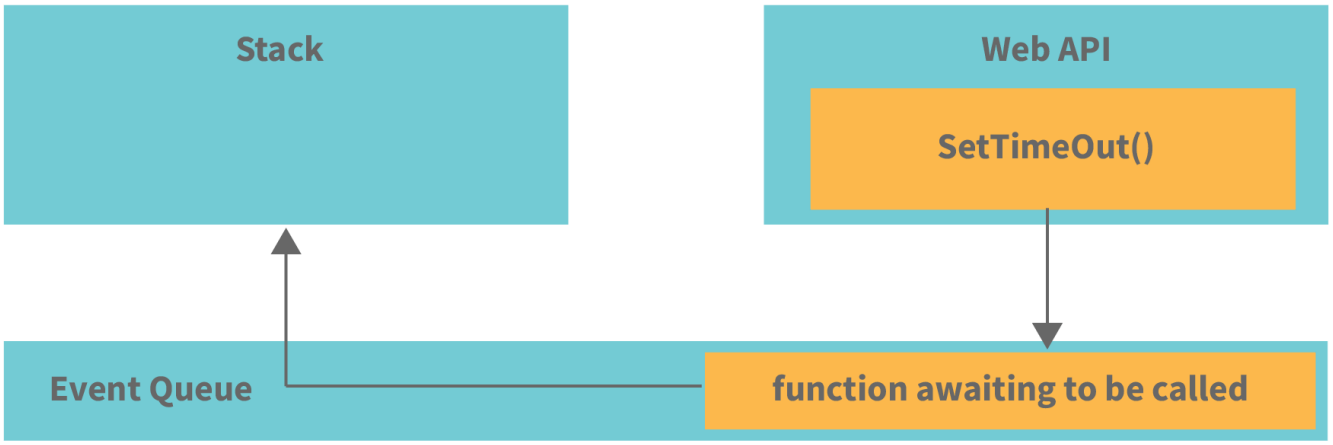
The order in which functions are executed, that is. when they are popped out of the stack once their purpose is completed, is as follows:

1. console.log
2. second
3. first
4. foo

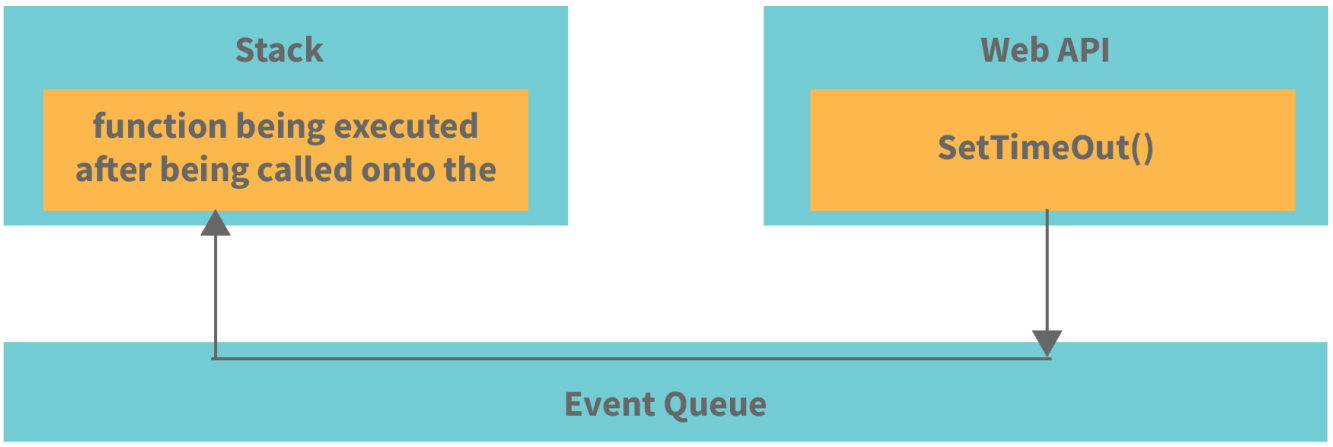
- **Event loop:** An event loop is something that pulls various things like methods, etc. out of the queue and places it onto the function execution stack whenever the function stack becomes empty. The event loop is the trick to making JavaScript appear multithreaded even if it is only single-threaded. The following illusion clearly explains how the event loop works:



The callback function in the event queue has not yet started and is waiting for its time to be added to the stack when SetTimeOut() is called and the Web API waits. The function is loaded onto the stack when the function stack becomes empty, as seen below:



The event loop is used to take the first event from the Event Queue and place it on the stack, which in this case is the callback function. If this function is called from here, it will call other functions within it.



22. JavaScript Error Handling

Various types of errors occur when we are coding in JavaScript. There are a few options for dealing with them:

- **try** — We can define a code block for testing errors using the try block.
- **catch** — We can set up a block of code to execute in the event of an error using the catch statement.
- **throw** — Instead of the typical JavaScript errors, we can also create custom error messages using the throw statement.
- **finally** — JavaScript also allows us to run our code regardless of the outcome of try and catch.

JavaScript possesses its own inbuilt error object which has the following properties:

- **name** — It is used for setting or returning an error name.
- **message** — It is used for setting or returning the error message as a string.

There are six types of ways in which the error property can return its name. They are as follows:

- **EvalError** — It indicates that an error has occurred within the eval() method.
- **RangeError** — It indicates that some number is “out of range”.
- **ReferenceError** — It indicates that an illegal reference was occurring.
- **SyntaxError** — It indicates that a syntax error was occurring.
- **TypeError** — It indicates that a type error was occurring.
- **URIError** — It indicates that an encodeURI() error was occurring.

23. ES5 Vs ES6: What is the Difference

ECMAScript 5 (ES5P): Because it was launched in 2009, ES5 is also known as ECMAScript 2009. It has functions where developers concentrate on how items are created. In ES5, you must use the function keyword and return to define the function, just as you would in any other JavaScript language.

ECMAScript 6 (ES6): Because it was launched in 2015, ES6 is also known as ECMAScript 2015. Its class allows developers to create an object with the new operator and an arrow function if they don't need to use the function keyword to specify the function, and they can also avoid using the return keyword to get the computer value.

The key differences between ES5 and ES6 are as shown below:

ES5	ES6
Its year of release is 2009.	Its year of release is 2015.
It accepts string, integer, boolean, null, and undefined primitive data types.	JavaScript data types have some new features in ES6. It introduced the 'symbol' basic data type to support unique values.
Using the var keyword, there is only one way to define variables.	The keywords let and const are two new ways to define variables.
ES5 has a lower performance compared to ES6.	ES6 has a better performance compared to ES5.
Object manipulation in ES5 takes a longer time than ES6.	Object manipulation in ES6 takes a lesser time than ES5.

Conclusion:

As a programming language, JavaScript is gaining popularity. Because of its proven track record and benefits, it is becoming the language of choice for developing web properties. JavaScript based libraries like React, Angular, Nest, etc. are becoming popular day by day because of the ease with which they allow users to use JavaScript to build applications. We have compiled several of the most basic and important operators, functions, principles, and methods in the JavaScript cheat sheet above. It gives both experienced developers and beginners a good introduction of the language and serves as a reference. We hope you found it informative.