JavaScript

Introduction

JavaScript:

JavaScript is the programming language of HTML and the Web.
JavaScript (often shortened to JS) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it's used in many non-browser environments as well such as Node.js. It is a prototype-based, multi-paradigm scripting language that is dynamic, and supports object-oriented, imperative, and functional programming styles.

JavaScript Can Do:

1. **JavaScript Can Change HTML Content:**
   One of many JavaScript HTML methods is **getElementById().** This method uses for find HTML element by its id attribute and JavaScript can change HTML Content by **innerHTML()** method. Such as-

   ```
   <p id="demo">JavaScript can change HTML content.</p>
   <button type="button" onclick='document.getElementById("demo").innerHTML = "Hello
   JavaScript!"'>Click Me!</button>
   ```

   Here, if we click on "**Click Me!**" button **.innerHTML** will change the value of **<p>** tag. We get **<p>** tag by its **id** attribute with help of **.getElementById()**.
2. **JavaScript Can Change HTML Attribute Values:**
   JavaScript can change html attribute values. To do this we need get the html element by its id attribute with help of **ducoment.getElementById()** method. After get the element we need to call the attribute which value will be change.

   ```
   <button onclick="document.getElementById('myImage').src='pic_bulbon.gif'">Turn on the
   light</button>
   <img id="myImage" src="pic_bulboff.gif" style="width:100px">
   <button onclick="document.getElementById('myImage').src='pic_bulboff.gif'">Turn off the
   light</button>
   ```

   Here, if we click on "**Turn on the light**" button it will change the **scr** attribute value of **<img>** tag.
3. **JavaScript Can Change HTML Styles (CSS):**
   JavaScript can change html style values also. To do this we need get the html element by its id attribute with help of **ducoment.getElementById()** method. After get the element we need to call html style attribute and after that need to call the css property which value will be change.

   ```
   <p id="demo">JavaScript can change the style of an HTML element.</p>
   <button type="button" onclick="document.getElementById('demo').style.fontSize='35px'">Click
   Me!</button>
   ```

   Here, if we click on "**Click Me!**" button, it will change <p> tag value's font size.
4. **JavaScript Can Hide/Show HTML Elements:**
   Hiding HTML elements can be done by changing the **display** style. Showing hidden HTML elements can also be done by changing the **display** style.

```
<p id="demo">JavaScript can hide/show HTML elements.</p>
<button type="button" onclick="document.getElementById('demo').style.display='none'">Click
Me!</button>
<button type="button" onclick="document.getElementById('demo').style.display='block'">Click
Me!</button>
```

JavaScript Where to Place:

In HTML, JavaScript code must be inserted between **<script>** and **</script>** tags. We can place any number of scripts in an HTML document. Scripts can be placed in the **<body>**, or in the **<head>** section of an HTML page, or in both.
Scripts can also be placed in external files which declare with **.js** extension. External scripts are practical when the same code is used in many different web pages. JavaScript files have the file extension **.js**. To use an external script, put the name of the script file in the **src** (source) attribute of a **<script>** tag.

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

We can place an external script reference in **<head>** or **<body>** as we like. The script will behave as if it was located exactly where the <script> tag is located.
External scripts cannot contain **<script>** tags.

JavaScript Output (Print):

JavaScript can "display" data in different ways:

- **Writing into an HTML element, using *innerHTML***:
  To access an HTML element, JavaScript can use the **document.getElementById(id)** method. The **id** attribute defines the HTML element. The **innerHTML** property defines the HTML **content**.

  ```
  <p id="demo"></p>
  <script>
  document.getElementById("demo").innerHTML = 5 + 6;
  </script>
  ```

  Changing the **innerHTML** property of an HTML element is a common way to display data in HTML.
- **Writing into the HTML output using *document.write()*:**
  For testing purposes, it is convenient to use document.write().

  ```
  <script>
  document.write(5 + 6);
  </script>
  ```

  Using **document.write()** after an HTML document is loaded, will **delete all existing HTML.**

  ```
  <p>My first paragraph.</p>
  <button type="button" onclick="document.write(5 + 6)">Try it</button>
  ```

  When we will click on “Try it” button it will show 11 but delete all other html content.
- **Writing into an alert box, using *window.alert()*:**
  We can use an alert box to display data. Such as-

  ```
  <script>window.alert(5 + 6);</script>
  ```

- **Writing into the browser console, using *console.log()*:**
  For debugging purposes, we can use the **console.log()** method to display data. Such as-

```
<script>console.log(5 + 6);</script>
```

JavaScript Statements:

- A **computer program** is a list of "instructions" to be "executed" by a computer. In a programming language, these programming instructions are called **statements**. A **JavaScript program** is a list of programming **statements**. Most JavaScript programs contain many JavaScript statements. The statements are executed, one by one, in the same order as they are written. JavaScript programs (and JavaScript statements) are often called JavaScript code. In HTML, JavaScript programs are executed by the web browser.
- JavaScript statements are composed of: Values, Operators, Expressions, Keywords, and Comments.

```
document.getElementById("demo").innerHTML = "Hello Dolly.";
```

  This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo"
- Semicolons separate JavaScript statements.

```
var a;
a = 5;
b = 6;
c = a + b;
a = 5; b = 6; c = a + b;
```

  In JavaScript Ending statements with semicolon is not required, but highly recommended.
- JavaScript ignores multiple spaces. We can add white space to script to make it more readable. A good practice is to put spaces around operators (= + - * /).

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

- For best readability, programmers often like to avoid code lines longer than 80 characters. If a JavaScript statement does not fit on one line, the best place to break it is after an operator.

```
document.getElementById("demo").innerHTML =
"Hello Dolly!";
```

- JavaScript statements can be grouped together in code blocks, inside curly brackets {...}. The purpose of code blocks is to define statements to be executed together. One place will find statements grouped together in blocks, is in JavaScript functions.

```
function myFunction() {
  document.getElementById("demo1").innerHTML = "Hello Dolly!";
  document.getElementById("demo2").innerHTML = "How are you?";
}
```

- JavaScript statements often start with a **keyword** to identify the JavaScript action to be performed. JavaScript keywords are reserved words. Reserved words cannot be used as names for variables. Some keyword: debugger, function, continue etc.

Syntax

- JavaScript syntax is the set of rules, how JavaScript programs are constructed.
- The JavaScript syntax defines two types of values: Fixed values and variable values. Fixed values are called **literals**. Variable values are called **variables**.
- The most important rules for writing fixed values are:
    - **Numbers** are written with or without decimals:

      ```
      10.50
      1001
      ```

    - **Strings** are text, written within double or single quotes:

      ```
      "John Doe"
      'John Doe'
      ```

- In a programming language, **variables** are used to **store** data values. JavaScript uses the **var** keyword to declare variables. An **equal sign** is used to **assign values** to variables.

  ```
  var x;
  x = 6;
  ```

- JavaScript uses **arithmetic operators** (+ - * /) to **compute** values:

  ```
  (5 + 6) * 10
  ```

- JavaScript uses an **assignment operator** (=) to **assign** values to variables:

  ```
  var x, y;
  x = 5;
  y = 6;
  ```

- An expression is a combination of values, variables, and operators, which computes to a value. The computation is called an evaluation.

  ```
  5 * 10
  ```

  Expressions can also contain variable values:

  ```
  x * 10
  ```

  The values can be of various types, such as numbers and strings.

  ```
  "John" + " " + "Doe"
  ```

- JavaScript **keywords** are used to identify actions to be performed. The `var` keyword tells the browser to create variables:

```
var x, y;
```

- Not all JavaScript statements are "executed". Code after double slashes `//` or between `/*` and `*/` is treated as a **comment**. Comments are ignored, and will not be executed:

```
// var x = 6;   I will NOT be executed
```

- Identifiers are names. In JavaScript, identifiers are used to name variables (and keywords, and functions, and labels). The rules for legal names are much the same in most programming languages. In JavaScript, the first character must be a letter, or an underscore (_), or a dollar sign ($).Subsequent characters may be letters, digits, underscores, or dollar signs. Numbers are not allowed as the first character.
- All JavaScript identifiers are **case sensitive**. The variables **lastName** and **lastname**, are two different variables. JavaScript does not interpret **VAR** or **Var** as the keyword **var**.
- JavaScript programmers tend to use camel case that starts with a lowercase letter: **firstName**, **lastName**.
- JavaScript uses the **Unicode** character set. Unicode covers (almost) all the characters, punctuations, and symbols in the world.

Comment:

JavaScript comments can be used to explain JavaScript code, and to make it more readable. JavaScript comments can also be used to prevent execution, when testing alternative code. There are two type of comment in JavaScript.

- **Single Line:** A single line comment will comment out a single line and is denoted with two forward slashes // preceding it.

```
// Prints 5 to the console
console.log(5);
```

We can also use a single line comment to comment after a line of code.

```
console.log(5);  // Prints 5
```

- **Multiline:** A multi-line comment will comment out multiple lines and is denoted with /* to begin the comment, and */ to end the comment.

```
/*
It is most common to use single line comments.
Block comments are often used for formal documentation.
*/
```

We can also use this syntax to comment something out in the middle of a line of code.

```
console.log(/*IGNORED!*/ 5);  // Still just prints 5
```

- It is most common to use single line comments. Block comments are often used for formal documentation.

Variables

- In programming, variables are containers for storing data values. Variables label and store data in memory. It is important to distinguish that variables are not values; they contain values and represent them with a name.
- It's a good programming practice to declare all variables at the beginning of a script.
- If you re-declare a JavaScript variable, it will not lose its value.

```
var carName = "Volvo";
var carName;
```

- If you put a number in quotes, the rest of the numbers will be treated as strings, and concatenated.

```
var x = "5" + 2 + 3;
```

- We can create a variable using three way. Such as-

var:

- In JavaScript, variables are created with **var** keyword. The **var** statement declares a variable, optionally initializing it to a value.

```
var myName = 'Arya';
```

- When we declare a variable with **var** keyword in globally, it’s accessible in everywhere.
- If we declare a variable with **var** keyword within a function (locally) then it only accessible from the function body part only. If we want access it outside of the function then it will throw undefined error.

```
var tester = "hey hi";
function newFunction() {
    var hello = "hello";
}
console.log(hello); // error: hello is not defined
```

But if declare it within a code block {} (such as conditional block) we can access the variable outside of the block.

```
var times = 4;
 if (times > 3) {
    var greeter = "say Hello instead";
 }
 console.log(greeter) //"say Hello instead"
```

- Variable names cannot start with numbers.
- Variable names are **case sensitive**, so **myName** and **myname** would be different variables. It is bad practice to create two variables that have the same name using different cases.
- Variable names cannot be the same as **keywords**. JavaScript keyword list: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Keywords

Let:

Let keyword also use to declare variable in JavaScript. The **let** keyword was introduced in ES6. Its act as like as **var** keyword. There is only one different between them. **Let** keyword solved **var** keywords code block problem. We can’t access a variable which declare within a code block with let keyword from outside of the code block. It will create undefined error.

```
let price;
console.log(price); // Output: undefined
price = 350;
console.log(price); // Output: 350
```

Const:

- The **const** keyword was also introduced in ES6, and is short for the word **constant**. Just like with **var** and **let** we can store any value in a **const** variable.

```
const myName = 'Gilberto';
console.log(myName);          // Output: Gilberto
```

- A **const** variable cannot be reassigned because it is constant. If we try to reassign a **const** variable, we will get a **TypeError**.

```
const entree = 'Enchiladas'
console.log(entree);
entree = 'Tacos'        //TypeError: Assignment to constant
```

- Constant variables must be assigned a value when declared. If you try to declare a **const** variable without a value, you'll get a **SyntaxError**.

```
const entree;   //SyntaxError: Missing initializer in const
```

- If we need to reassign the variable we should use **let**, otherwise, **const**.

Operator

JavaScript has multiple operator for different purpose. Such as-

Arithmetic Operators:

JavaScript has several built-in in *arithmetic operators* that allow us to perform mathematical calculations on numbers. These include the following operators and their corresponding symbols:

1. **Add: +**
2. **Subtract: -**
3. **Multiply: ***
4. **Divide: /**
5. **Remainder/Modulus: %**
6. **Exponentiation: ****
7. **Increment: ++**
8. **Decrement: --**

In arithmetic, the division of two integers produces a **quotient** and a **remainder**. In mathematics, the result of a **modulo operation** is the **remainder** of an arithmetic division.
The **exponentiation** operator (**) raises the first operand to the power of the second operand. **x ** y** produces the same result as **Math.pow(x,y)**.

Assignment Operators:

Assignment operators assign values to JavaScript variables.

- The = assignment operator assigns a value to a variable.

```
var x = 10;
```

- The `+=, -=, *=, and /=` assignment operator adds, subtracts, multiplies, divides a value to a variable.

```
var x = 10;
x += 5;
```

String Operators:

The + operator can also be used to add (concatenate) strings.

```
var txt1 = "John";
var txt2 = "Doe";
var txt3 = txt1 + " " + txt2;
```

The += assignment operator can also be used to add (concatenate) strings:

```
var txt1 = "What a very ";
txt1 += "nice day";
```

Adding two numbers, will return the sum, but adding a number and a string will return a string:

```
var x = 5 + 5;
var y = "5" + 5;
var z = "Hello" + 5;
```

When used on strings, the + operator is called the concatenation operator.

Comparison Operators:

Comparison and Logical operators are used to test for **true** or **false**. Comparison operators are used in logical statements to determine equality or difference between variables or values.

- **== (equal to)**: compare value only.
- **=== (equal value and equal type)**: compare value and type.
- **!= (not equal)**
- **!== (not equal value and not equal type)**
- **> (greater than)**
- **< (less than)**
- **>= (greater than or equal to)**
- **<= (less than or equal to)**

Logical Operators:

Logical operators are used to determine the logic between variables or values.

- && (Logical And)
- ||(Logical or)
- ! (Logical not)

Conditional (Ternary) Operator:

JavaScript also contains a conditional operator that assigns a value to a variable based on some condition.

```
variablename = (condition) ? value1:value2
var voteable = (age < 18) ? "Too young":"Old enough";
```

Type Operators:

JavaScript has two type operators by using them we can check type of an object or variable.

- **typeof**: Returns the type of a variable
- **instanceof**: Returns true if an object is an instance of an object type

Bitwise Operators:

JavaScript stores numbers as 64 bits floating point numbers, but all bitwise operations are performed on 32 bits binary numbers. Before a bitwise operation is performed, JavaScript converts numbers to 32 bits signed integers. After the bitwise operation is performed, the result is converted back to 64 bits JavaScript numbers.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

Example-

Operation	Result	Same as	Result
5 & 1	1	0101 & 0001	0001
5 | 1	5	0101 | 0001	0101
~ 5	10	~0101	1010
5 << 1	10	0101 << 1	1010
5 ^ 1	4	0101 ^ 0001	0100
5 >> 1	2	0101 >> 1	0010
5 >>> 1	2	0101 >>> 1	0010

Data Type

JavaScript variables can hold many **data types**: numbers, strings, objects and more. JavaScript has dynamic types. This means that the same variable can be used to hold different data types.

```
var x;          // Now x is undefined
x = 5;          // Now x is a Number
x = "John";     // Now x is a String
```

In JavaScript, there are seven fundamental data types:

1. ***Number***
2. ***String***
3. ***Boolean***
4. ***Null***
5. ***Undefined***
6. ***Symbol*** (A newer feature to the language, symbols are unique identifiers, useful in more complex coding.)
7. ***Object***

There are another data type in JavaScript.

8. ***Arrays***

String, Number, Boolean and Undefined are treat as primitive type and Object, Arrays and Null as Object type.

Null:

In JavaScript null is "nothing". It is supposed to be something that doesn't exist. This data type represents the intentional absence of a value, and is represented by the keyword **null** (without quotes).
Unfortunately, in JavaScript, the data type of `null` is an object.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;   // Now value is null, but type is still an object
```

Undefined:

In JavaScript, a variable without a value, has the value **undefined**. The type is also **undefined**.

```
var car;   // Value is undefined, type is undefined
```

Any variable can be **emptied**, by setting the value to **undefined**. The type will also be **undefined**.

```
car = undefined;   // Value is undefined, type is undefined
```

String

A string (or a text string) is a series of characters like "John Doe". Strings are used for storing and manipulating text.
Strings are written with quotes. We can use single or double quotes.

```
var carName1 = "Volvo XC60";  // Using double quotes
var carName2 = 'Volvo XC60';  // Using single quotes
```

We can use quotes inside a string, as long as they don't match the quotes surrounding the string:

```
var answer1 = "It's alright";           // Single quote inside double quotes
var answer2 = "He is called 'Johnny'";  // Single quotes inside double quotes
var answer3 = 'He is called "Johnny"';  // Double quotes inside single quotes
```

Special Characters:

Because strings must be written within quotes, JavaScript will misunderstand this string:

```
var x = "We are the so-called "Vikings" from the north.";
```

The string will be chopped to "We are the so-called ". The solution to avoid this problem, is to use the **backslash escape character**. The backslash (\) escape character turns special characters into string characters:

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

The sequence `\"` inserts a double quote in a string:

```
var x = "We are the so-called \"Vikings\" from the north.";  // We are the so-called "Vikings" from the north.
```

Six other escape sequences are valid in JavaScript.

Code	Result
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tabulator
\v	Vertical Tabulator

The 6 escape characters above were originally designed to control typewriters, teletypes, and fax machines. They do not make any sense in HTML.

Breaking Long Code Lines:

For best readability, programmers often like to avoid code lines longer than 80 characters. If a JavaScript statement does not fit on one line, the best place to break it is after an operator. We can also break up a code line **within a text string** with a single backslash.

```
document.getElementById("demo").innerHTML = "Hello \
Dolly!";
```

The \ method is not the preferred method. It might not have universal support. Some browsers do not allow spaces behind the \ character.
We cannot break up a code line with a backslash:

```
document.getElementById("demo").innerHTML = \
"Hello Dolly!";
```

A safer way to break up a string, is to use string addition:

```
document.getElementById("demo").innerHTML = "Hello " +
"Dolly!";
```

Strings Can be Objects:

Normally, JavaScript strings are primitive values. But strings can also be defined as objects with the keyword **new**.

```
var firstName = new String("John");
```

But we should not create strings as objects. It slows down execution speed. The **new** keyword complicates the code. This can produce some unexpected results.

String Length:

The **length** property returns the length of a string:

```
var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
var sln = txt.length;
```

Finding String Position:

The **indexOf()** and **search()** method returns the index of (the position of) the first occurrence of a specified text in a string:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.indexOf("locate"); //7

var str = "Please locate where 'locate' occurs!";
var pos = str.search("locate"); //7
```

But the two methods are **NOT** equal. These are the differences:

- The **search()** method cannot take a second start position argument.
- The **indexOf()** method cannot take powerful search values (regular expressions).

The **`lastIndexOf()`** method returns the index of the **last** occurrence of a specified text in a string:

```
var str = "Please locate where 'locate' occurs!";
var pos = str.lastIndexOf("locate");
```

Both **`indexOf()`**, and **`lastIndexOf()`** return -1 if the text is not found.

```
var str = "Please locate where 'locate' occurs!";
var pos = str.lastIndexOf("John");
```

Both methods accept a second parameter as the starting position for the search. But The **lastIndexOf()** methods searches **backwards** (from the end to the beginning), meaning: if the second parameter is 15, the search starts at position 15, and searches to the beginning of the string.

```
var str = "Please locate where 'locate' occurs!";
var pos = str.indexOf("locate", 15);   //21

var str = "Please locate where 'locate' occurs!";
var pos = str.lastIndexOf("locate", 15);
```

Extracting String Parts:

There are 3 methods for extracting a part of a string:

- **slice(start, end)**
- **substring(start, end)**
- **substr(start, length)**

slice() extracts a part of a string and returns the extracted part in a new string. The method takes 2 parameters: the start position, and the end position (end not included).

```
var str = "Apple, Banana, Kiwi";
var res = str.slice(7, 13); //Banana
```

If a parameter is negative, the position is counted from the end of the string.

```
var str = "Apple, Banana, Kiwi";
var res = str.slice(-12, -6); //Banana
```

If omit the second parameter, the method will slice out the rest of the string. If negative count from back.

```
var res = str.slice(-12);
```

substring() is similar to **slice()**.The difference is that **substring()** cannot accept negative indexes.

```
var str = "Apple, Banana, Kiwi";
var res = str.substring(7, 13);
```

If omit the second parameter, **substring()** will slice out the rest of the string.

substr() is similar to **slice()**.The difference is that the second parameter specifies the **length** of the extracted part. If you omit the second parameter, **substr()** will slice out the rest of the string. If the first parameter is negative, the position counts from the end of the string.

```
var str = "Apple, Banana, Kiwi";
var res = str.substr(-4); //Kiwi
```

Replacing String Content:

The **replace()** method replaces a specified value with another value in a string. It does not change the string it is called on. It returns a new string. By default, the `replace()` method replaces **only the first** match.

```
str = "Please visit Microsoft and Microsoft!";
var n = str.replace("Microsoft", "W3Schools");
```

By default, the **replace()** method is case sensitive. Writing **MICROSOFT** (with upper-case) will not work:

```
str = "Please visit Microsoft!";
var n = str.replace("MICROSOFT", "W3Schools");
```

To replace case **insensitive**, use a regular expression with an **/i flag** (insensitive):

```
str = "Please visit Microsoft!";
var n = str.replace(/MICROSOFT/i, "W3Schools");
```

To replace all matches, use a **regular expression** with a `/g` flag (global match):

```
str = "Please visit Microsoft and Microsoft!";
var n = str.replace(/Microsoft/g, "W3Schools");
```

Converting to Upper and Lower Case:

A string is converted to upper case with `toUpperCase()` and a string is converted to lower case with `toLowerCase()`.

```
var text1 = "Hello World!";       // String
var text2 = text1.toLowerCase();  // text2 is text1 converted to lower
```

The concat() Method:

concat() joins two or more strings. It can be used instead of the plus operator.

```
var text = "Hello" + " " + "World!";
var text = "Hello".concat(" ", "World!");
```

All string methods return a new string. They don't modify the original string. Formally said: Strings are immutable: Strings cannot be changed, only replaced.

Remove whitespace:

The **trim()** method removes whitespace from both sides of a string:

```
var str = "       Hello World!        ";
```

Extracting String Characters:

There are 3 methods for extracting string characters:

- **charAt(position)**
- **charCodeAt(position)**
- **Property access [ ]**

The **charAt()** method returns the character at a specified **index (position)** in a string.

```
var str = "HELLO WORLD";
str.charAt(0);          // returns H
```

The **charCodeAt()** method returns the **unicode** of the character at a specified index in a string.

```
var str = "HELLO WORLD";
str.charCodeAt(0);       // returns 72
```

ECMAScript 5 (2009) allows property access [ ] on strings.

```
var str = "HELLO WORLD";
str[0];                  // returns H
```

Property access might be a little **unpredictable:**

- It does not work in Internet Explorer 7 or earlier
- It makes strings look like arrays (but they are not)
- If no character is found, [ ] returns undefined, while charAt() returns an empty string.
- It is read only. str[0] = "A" gives no error (but does not work!)

```
var str = "HELLO WORLD";
str[0] = "A";           // Gives no error, but does not work
str[0];                 // returns H
```

Converting a String to an Array:

A string can be converted to an array with the `split()` method:

```
var txt = "a,b,c,d,e";   // String
txt.split(",");        // Split on commas
txt.split(" ");        // Split on spaces
txt.split("|");        // Split on pipe
```

If the separator is omitted, the returned array will contain the whole string in index [0]. If the separator is "", the returned array will be an array of single characters.

String Interpolation:

In the ES6 version of JavaScript, we can insert, or interpolate, variables into strings using template literals.

```
const myPet = 'armadillo';
console.log(`I own a pet ${myPet}.`);     // Output: I own a pet armadillo.
```

- **A template literal is wrapped by backticks `(this key is usually located on the top of your keyboard, left of the 1 key)**.
- Inside the template literal, you'll see a placeholder, ${myPet}. The value of myPet is inserted into the template literal.
- When we interpolate `I own a pet ${myPet}.`, the output we print is the string: 'I own a pet armadillo.'

One of the biggest benefits to using template literals is the readability of the code. Using template literals, we can more easily tell what the new string will be. We also don't have to worry about escaping double quotes or single quotes.

Number

JavaScript has only one type of numbers. Numbers can be written with, or without decimals.

```
var x = 3.14;    // A number with decimals
var y = 3;      // A number without decimals
```

Extra-large or extra small numbers can be written with scientific (exponent) notation.

```
var x = 123e5;    // 12300000
var y = 123e-5;   // 0.00123
```

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc. JavaScript numbers are always stored as double precision floating point numbers.

This format stores numbers in 64 bits, where the number (the fraction (decimal)) is stored in bits 0 to 51, the exponent (power) in bits 52 to 62, and the sign (+, -) in bit 63

Integers (numbers without a period or exponent notation) are accurate up to 15 digits.

```
var x = 999999999999999;   // x will be 999999999999999
var y = 9999999999999999;  // y will be 10000000000000000
```

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate:

```
var x = 0.2 + 0.1;         // x will be 0.30000000000000004
```

To solve the problem above, it helps to multiply and divide:

```
var x = (0.2 * 10 + 0.1 * 10) / 10;      // x will be 0.3
```

Adding Numbers and Strings:

JavaScript uses the + operator for both addition and concatenation. Numbers are added. Strings are concatenated.
If we add two numbers, the result will be a number. If we add two strings, the result will be a string concatenation. If you add a string and a number, the result will be a string concatenation. If we add a number and a string, the result will be a string concatenation.

```
var x = 10;
var y = "20";
var z = x + y;          // z will be 1020 (a string)
```

If first two are numbers and last one string and we add all of them, first two will added first then will concatenation string. Such as-

```
var x = 10;
var y = 20;
var z = "30";
var result = x + y + z; //3030
```

Because of this, The JavaScript interpreter works from left to right. First 10 + 20 is added because x and y are both numbers. Then 30 + "30" is concatenated because z is a string.

Numeric Strings:

JavaScript strings can have numeric content. JavaScript will try to convert strings to numbers in all numeric operations. *, / and – operation work with numeric string. But + operation is not work with numeric string. Because it work like as concatenation.

```
var x = "100";
var y = "10";
var z = x - y;      // z will be 90

var x = "100";
var y = "10";
var z = x + y;      // z will not be 110 (It will be 10010)
```

NaN - Not a Number:

NaN is a JavaScript reserved word indicating that a number is not a legal number. Trying to do arithmetic with a non-numeric string will result in `NaN` (Not a Number). However, if the string contains a numeric value, the result will be a number.

```
var x = 100 / "Apple";  // x will be NaN (Not a Number)
var x = 100 / "10";     // x will be 10
```

We can use the global JavaScript function **isNaN()** to find out if a value is a number or not. If the value is number it will be return false.

```
var x = 100 / "Apple";
isNaN(x);             // returns true because x is Not a Number
```

If you use **NaN** in a mathematical operation, the result will also be **NaN** or might be a concatenation.

```
var x = NaN;
var y = 5;
var z = x + y;        // z will be NaN

var x = NaN;
var y = "5";
var z = x + y;        // z will be NaN5
```

NaN is a number**: typeof NaN** returns number.

```
typeof NaN;           // returns "number"
```

Infinity:

`Infinity` (or `-Infinity`) is the value JavaScript will return if you calculate a number outside the largest possible number.

```
var myNumber = 2;
while (myNumber != Infinity) {   // Execute until Infinity
  myNumber = myNumber * myNumber;
}

var x =  2 / 0;       // x will be Infinity
var y = -2 / 0;       // y will be -Infinity
```

`Infinity` is a number: `typeof Infinity` returns `number`.

Number Conversion:

By default, JavaScript displays numbers as **base 10** decimals. But we can use the `toString()` method to output numbers from **base 2** to **base 36**. Hexadecimal is **base 16**. Decimal is **base 10**. Octal is **base 8**. Binary is **base 2**.

```
var myNumber = 32;
var conNumber1 = myNumber.toString(10);  // returns 32
var conNumber2 = myNumber.toString(32);  // returns 10
var conNumber3 = myNumber.toString(16);  // returns 20
var conNumber4 = myNumber.toString(8);   // returns 40
var conNumber5 = myNumber.toString(2);   // returns 100000
```

And reverse the process with:

```
var revNumber1 = parseInt (conNumber1 , 10);  // returns 32
var revNumber2 = myNumber.toString(conNumber2, 32);  // returns 32
var revNumber3 = myNumber.toString(conNumber3, 16);  // returns 32
var revNumber4 = myNumber.toString(conNumber4, 8);   // returns 32
var revNumber5 = myNumber.toString(conNumber5, 2);   // returns 32
```

Numbers Can be Objects:

Normally JavaScript numbers are primitive values created from literals. But numbers can also be defined as objects with the keyword **`new`**. It slows down execution speed.

Numbers Methods:

- The `toString()` method returns a number as a string.

  ```
  var x = 123;
  x.toString();          // returns 123 from variable x
  (123).toString();      // returns 123 from literal 123
  (100 + 23).toString(); // returns 123 from expression 100 + 23
  ```

- `toExponential()` returns a string, with a number rounded and written using exponential notation. A parameter defines the number of characters behind the decimal point.

  ```
  var x = 9.656;
  x.toExponential(2);    // returns 9.66e+0
  x.toExponential(4);    // returns 9.6560e+0
  ```

  The parameter is optional. If don't specify it, JavaScript will not round the number.
- `toFixed()` returns a string, with the number written with a specified number of decimals.

  ```
  var x = 9.656;
  x.toFixed(0);          // returns 10
  x.toFixed(2);          // returns 9.66
  x.toFixed(4);          // returns 9.6560
  ```

- `toPrecision()` returns a string, with a number written with a specified length.

  ```
  var x = 9.656;
  x.toPrecision();       // returns 9.656
  x.toPrecision(2);      // returns 9.7
  x.toPrecision(4);      // returns 9.656
  ```

- `valueOf()` returns a number as a number. In JavaScript, a number can be a primitive value (typeof = number) or an object (typeof = object). The `valueOf()` method is used internally in JavaScript to convert Number objects to primitive values.

```
var x = 123;
x.valueOf();          // returns 123 from variable x
(123).valueOf();      // returns 123 from literal 123
(100 + 23).valueOf(); // returns 123 from expression 100 + 23
```

All JavaScript data types have a `valueOf()` and a `toString()` method.

Converting Variables to Numbers:

There are 3 JavaScript methods that can be used to convert variables to numbers.

Method	**Description**
Number()	Returns a number, converted from its argument.
parseFloat()	Parses its argument and returns a floating point number
parseInt()	Parses its argument and returns an integer

These methods are not **number** methods, but **global** JavaScript methods. JavaScript global methods can be used on all JavaScript data types.

- `Number()` can be used to convert JavaScript variables to numbers. If the number cannot be converted, `NaN` (Not a Number) is returned. It can also convert a date to a number.

```
Number(true);        // returns 1
Number(false);       // returns 0
Number("10");        // returns 10
Number("  10");      // returns 10
Number("10  ");      // returns 10
Number(" 10  ");     // returns 10
Number("10.33");     // returns 10.33
Number("10,33");     // returns NaN
Number("10 33");     // returns NaN
Number("John");      // returns NaN
Number(new Date("2017-09-30"));   // returns 1506729600000
```

The `Number()` method returns the number of milliseconds since 1.1.1970 for date.

- `parseInt()` parses a string and returns a whole number. Spaces are allowed. Only the first number is returned.

```
parseInt("10");        // returns 10
parseInt("10.33");     // returns 10
parseInt("10 20 30");  // returns 10
parseInt("10 years");  // returns 10
parseInt("years 10");  // returns NaN
```

- `parseFloat()` parses a string and returns a number. Spaces are allowed. Only the first number is returned.

```
parseFloat("10");        // returns 10
parseFloat("10.33");     // returns 10.33
parseFloat("10 20 30");  // returns 10
parseFloat("10 years");  // returns 10
parseFloat("years 10");  // returns NaN
```

Number Properties:

Property	Description
MAX_VALUE	Returns the largest number possible in JavaScript
MIN_VALUE	Returns the smallest number possible in JavaScript
POSITIVE_INFINITY	Represents infinity (returned on overflow)
NEGATIVE_INFINITY	Represents negative infinity (returned on overflow)
NaN	Represents a "Not-a-Number" value

```
var x = Number.MAX_VALUE; //1.7976931348623157e+308
var x = Number.NEGATIVE_INFINITY; //-Infinity
```

Number properties belongs to the JavaScript's number object wrapper called **Number**. These properties can only be accessed as `Number.MAX_VALUE`. Using *myNumber*.MAX_VALUE, where *myNumber* is a variable, expression, or value, will return `undefined`.

```
var x = 6;
var y = x.MAX_VALUE;    // y becomes undefined
```

Booleans

A JavaScript Boolean represents one of two values: **true** or **false**. You can use the `Boolean()` function to find out if an expression (or a variable) is true.

```
Boolean(10 > 9)       // returns true
```

Or even easier:

```
(10 > 9)          // also returns true
10 > 9            // also returns true
```

Everything with a "Value" is true and Everything Without a "Value" is False. The Boolean value of **-0** (minus zero) is **false.** The Boolean value of **""** (empty string) is **false.** The Boolean value of **undefined** is **false.** The Boolean value of **null** is **false.** The Boolean value of **false** is (you guessed it) **false.** The Boolean value of **NaN** is **false.**

```
var b1 = Boolean(100); //true
var x;
Boolean(x);      // returns false
```

Objects

JavaScript objects are written with curly braces `{}`. Object properties are written as name:value pairs, separated by commas. Objects are variables too. But objects can contain many values.

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

Accessing Object Properties:

We can access object properties in two ways.

- *objectName.propertyName*
- *objectName["propertyName"]*

```
person.lastName;
person["lastName"];
```

Object Methods:

Objects can also have **methods**.

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

You access an object method with the following syntax:

objectName.methodName()

Arrays

JavaScript arrays are used to store multiple values in a single variable. Arrays are written with square brackets. Array items are separated by commas.

An array can hold many values under a single name, and can access the values by referring to an index number. Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

Arrays can store any data types (including strings, numbers, and booleans). Like lists, arrays are ordered, meaning each item has a numbered position.

Creating Arrays:

- Using an array literal is the easiest way to create a JavaScript Array.
  var *array_name* = [*item1*, *item2*, ...];

  ```
  var cars = ["Saab", "Volvo", "BMW"];
  ```

- Using new keyword-

  ```
  var cars = new Array("Saab", "Volvo", "BMW");
  ```

Array Length:

The `length` property of an array returns the length of an array (the number of array elements).

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length;   // the length of fruits is 4
```

Access the Elements of an Array:

We access an array element by referring to the **index number**.

```
var name = cars[0];
```

Access the Full Array:

With JavaScript, the full array can be accessed by referring to the array name:

```
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars; // Saab,Volvo,BMW
```

Adding Element:

- The easiest way to add a new element to an array is using the `push()` method. The **`push()`** method adds a new element to an array (at the end) and returns the new array length.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.push("Kiwi");   // adds a new element (Kiwi) to fruits and the value of x is 5
```

- The **`unshift()`** method adds a new element to an array at the beginning, and returns the new array length. unshifting is equivalent to pushing, working on the first element instead of the last.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");    // Adds a new element "Lemon" to fruits and Returns 5
```

- New element can also be added to an array using the `length` property:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length] = "Lemon";    // adds a new element (Lemon) to fruits
```

- We can add elements in an empty array using index position also.

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;     // person.length will return 3
var y = person[0];        // person[0] will return "John"
```

Changing an Array Element:

Array elements are accessed using their **index number.** So, we can change array element using their index number.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[0] = "Kiwi";       // Changes the first element of fruits to "Kiwi"
```

Removing Array Element:

- The `pop()` method removes the last element from an array and returns the value that was "popped out".

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.pop();   // Removes the last element ("Mango") from fruits and the value of x is "Mango"
```

- The `shift()` method removes the first array element and "shifts" all other elements to a lower index and returns the string that was "shifted out". Shifting is equivalent to popping, working on the first element instead of the last.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var x = fruits.shift();  // Removes the first element "Banana" from fruits and the value of x is "Banana"
```

- Since JavaScript arrays are objects, elements can be deleted by using the JavaScript operator `delete`. It leaves **undefined** holes in the array.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
delete fruits[0];  // Changes the first element in fruits to undefined
```

Splicing an Array:

The `splice()` method used to add new items to an array and remove items from array.
In splice() method we can specify how many items should be remove from which position and which items should be add in that position.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 2, "Lemon", "Kiwi");
```

We can add items only by specified remove item 0.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
```

And we can remove items only by don’t specify any item.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(0, 1);       // Removes the first element of fruits
```

In splice() method The first parameter defines the position **where** new elements should be **added** (spliced in). The second parameter defines **how many** elements should be **removed**. The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**. The `splice()` method returns an array with the deleted items.

Merging (Concatenating) Arrays:

The **`concat()`** method creates a new array by merging (concatenating) existing arrays.

```
var myGirls = ["Cecilie", "Lone"];
var myBoys = ["Emil", "Tobias", "Linus"];
var myChildren = myGirls.concat(myBoys);   // Concatenates (joins) myGirls and myBoys
```

The `concat()` method does not change the existing arrays. It always returns a new array. The `concat()` method can take any number of array arguments.

```
var arr1 = ["Cecilie", "Lone"];
var arr2 = ["Emil", "Tobias", "Linus"];
var arr3 = ["Robin", "Morgan"];
var myChildren = arr1.concat(arr2, arr3);   // Concatenates arr1 with arr2 and arr3
```

The `concat()` method can also take values as arguments.

```
var arr1 = ["Cecilie", "Lone"];
var myChildren = arr1.concat(["Emil", "Tobias", "Linus"]);
```

Slicing an Array:

The **`slice()`** method slices out a piece of an array into a new array. The creates a new array. It does not remove any elements from the source array. It can take two arguments like `slice(1, 3)`. The method then selects elements from the start argument, and up to the end argument.

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
var citrus = fruits.slice(1, 3); // Orange,Lemon
```

If the end argument is omitted, the `slice()` method slices out the rest of the array.

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
var citrus = fruits.slice(1); // Orange,Lemon,Apple,Mango
```

Converting Array to String:

- JavaScript automatically converts an array to a comma separated string when a primitive value is expected. This is always the case when you try to output an array.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits; // Banana,Orange,Apple,Mango
```

- The JavaScript method `toString()` converts an array to a string of (comma separated) array values.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString(); // Banana,Orange,Apple,Mango
```

- The `join()` method also joins all array elements into a string. It behaves just like `toString()`, but in addition can specify the separator:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
// Banana * Orange * Apple * Mango
```

Sorting an Array:

- The **`sort()`** method sorts an array alphabetically.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();       // Sorts the elements of fruits
```

- By default, the `sort()` function sorts values as **strings**. If numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1". We can fix this by providing a **compare function.**

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b}); //1,5,10,25,40,100
```

  Using the same trick we can sort an array descending.

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a}); //100,40,25,10,5,1
```

  The compare function should return a negative, zero, or positive value, depending on the arguments:

```
function(a, b){return a - b}
```

  When the `sort()` function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.
  - If the result is **negative** `a` is sorted before `b`.
  - If the result is **positive** `b` is sorted before `a`.
  - If the result is **0** no changes are done with the sort order of the two values.
- JavaScript arrays often contain objects.

```
var cars = [
  {type:"Volvo", year:2016},
  {type:"Saab", year:2001},
  {type:"BMW", year:2010}
];
```

  Even if objects have properties of different data types, the `sort()` method can be used to sort the array. The solution is to write a compare function to compare the property values.

```
cars.sort(function(a, b){return a.year - b.year});
```

  Comparing string properties is a little more complex.

```
cars.sort(function(a, b){
  var x = a.type.toLowerCase();
  var y = b.type.toLowerCase();
  if (x < y) {return -1;}
  if (x > y) {return 1;}
  return 0;
});
```

Reversing an Array:

The `reverse()` method reverses the elements in an array.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.reverse();    // Reverse the order of the elements
```

We can use it to sort an array in descending order.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();       // First sort the elements of fruits
fruits.reverse();    // Then reverse the order of the elements
```

Arrays Iteration:

- **Array.forEach():** The `forEach()` method calls a function (a callback function) once for each array element. The callback function takes 3 arguments:
  - The item value
  - The item index
  - The array itself

```
var txt = "";
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(myFunction); //45, 4, 9, 16, 25
function myFunction(value, index, array) {
  txt = txt + value+', ';
}
```

  But only one argument is mandatory to store iterate value.

```
var txt = "";
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(myFunction); //45, 4, 9, 16, 25
function myFunction(value) {
  txt = txt + value+', ';
}
```

  We can write the callback function within forEach(). This is the best way.

```
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(function(element) {
  console.log(element);
});
```

- **Array.map():** The `map()` method creates a new array by performing a function on each array element. The `map()` method does not execute the function for array elements without values. The `map()` method does not change the original array. the function takes 3 arguments:
  - The item value
  - The item index
  - The array itself

  But only value is mandatory.

```
var numbers1 = [45, 4, 9, 16, 25];
var numbers2 = numbers1.map(myFunction); //[90, 8, 18, 32, 50]
function myFunction(value) {
  return value*2;
}
```

- **Array.filter():** The `filter()` method creates a new array with array elements that passes a test. The function takes 3 arguments:
    - The item value
    - The item index
    - The array itself

  But only value is mandatory.

```
var numbers = [45, 4, 9, 16, 25];
var over18 = numbers.filter(myFunction); //[45, 25]
function myFunction(value) {
  return value > 18;
}
```

- **Array.reduce():** The `reduce()` method runs a function on each array element to produce (reduce it to) a single value by performing an arithmetic operation. The `reduce()` method works from left-to-right in the array. It does not reduce the original array. The function takes 4 arguments:
    - The total (the initial value / previously returned value)
    - The item value
    - The item index
    - The array itself

  But two are mandatory. One for initial value and another iterate item.

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduce(myFunction); //99
function myFunction(total, value) {
  return total + value;
}
```

  The `reduce()` method can accept an initial value.

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduce(myFunction, 100); //199
function myFunction(total, value) {
  return total + value;
}
```

- **Array.reduceRight():** reduceRight() work as like as reduce() method. The only different between two method is reduce() method works from left-to-right in the array and reduceRight() method works from right-to-left in the array.

```
var numbers1 = [45, 4, 9, 16, 25];
var sum = numbers1.reduceRight(myFunction); //99
function myFunction(total, value) {
  return total + value;
}
```

- **Array.every():** The `every()` method check if all array values pass a test. `every()` will return true if *all* predicate is `true`. The function takes 3 arguments:

    - The item value
    - The item index
    - The array itself

  But only one parameter (value) are mandatory.

```
var numbers = [45, 4, 9, 16, 25];
var allOver18 = numbers.every(myFunction); //false
function myFunction(value) {
  return value > 18;
}
```

- **Array.some():** The `some()` method check if some array values pass a test. **some()** will return true if any predicate is true.

```
var numbers = [45, 4, 9, 16, 25];
var someOver18 = numbers.some(myFunction); //true
function myFunction(value, index, array) {
  return value > 18;
}
```

- **Array.find():** The `find()` method returns the value of the first array element that passes a test function.

```
var numbers = [4, 9, 16, 25, 29];
var first = numbers.find(myFunction);//25
function myFunction(value, index, array) {
  return value > 18;
}
```

- **Array.findIndex():** The `findIndex()` method returns the index of the first array element that passes a test function.

```
var numbers = [4, 9, 16, 25, 29];
var first = numbers.findIndex(myFunction);//3
function myFunction(value, index, array) {
  return value > 18;
}
```

- **Array.indexOf():** The `indexOf()` method searches an array for an element value and returns its position. `Array.indexOf()` returns -1 if the item is not found. If the item is present more than once, it returns the position of the first occurrence.

```
var fruits = ["Apple", "Orange", "Apple", "Mango"];
var a = fruits.indexOf("Apple"); //0
```

  We can specify the searching position within the method with another parameter. If the second parameter value is negative than search start at the given position counting from the end, and search to the end.

- **Array.lastIndexOf():** `Array.lastIndexOf()` is the same as `Array.indexOf()`, but searches from the end of the array.

```
var fruits = ["Apple", "Orange", "Apple", "Mango"];
var a = fruits.lastIndexOf("Apple");
```

Function

A **function** is a reusable block of code that groups together a sequence of statements to perform a specific task. 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).

Syntax:

A JavaScript function is defined with the `function` keyword, followed by a **name**, followed by parentheses **()**.Function names can contain letters, digits, underscores, and dollar signs (same rules as variables). The parentheses may include parameter names separated by commas: **(*parameter1, parameter2 ...*).** The code to be executed, by the function, is placed inside curly brackets: **{}.**

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

Function Return:

When JavaScript reaches a `return` statement, the function will stop executing. Functions often compute a **return value**. The return value is "returned" back to the "caller":

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

Function Expressions:

A JavaScript function can also be defined using an **expression**. To define a function inside an **expression**, we can use the **function** keyword. **In a function expression, the function name is usually omitted**. A function with no name is called an **anonymous function**. A function expression is often stored in a variable in order to refer to it. The variable should be declare with **const** and when we call the function we need to call it by variable name with parenthesis ().

```
const plantNeedsWater = function(day){
  if(day==='Wednesday'){
        return true;
   }else{
    return false;
   }
}
console.log(plantNeedsWater('Tuesday')); //false
```

Arrow Functions:

Arrow functions allows a short syntax for writing function expressions. You don't need the `function` keyword, the `return` keyword, and the **curly brackets**.

```
const x = (x, y) => x * y;
```

Arrow functions do not have their own `this`. They are not well suited for defining **object methods**. Arrow functions are not hoisted. They must be defined **before** they are used. Using `const` is safer than using `var`, because a function expression is always constant value. We can only omit the `return` keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them.

```
const x = (x, y) => { return x * y };
```

Arrow functions that take only a **single parameter do not need that parameter to be enclosed in parentheses**.

```
const functionName = paramOne => paramOne * 2;      //one parameter
```

However, if a function takes **zero or multiple parameters, parentheses are required**. So, that it is good practice to keep them.

JavaScript Function Parameters:

JavaScript function definitions do not specify data types for parameters. JavaScript functions do not perform type checking on the passed arguments. JavaScript functions do not check the number of arguments received.
If a function is called with **missing arguments** (less than declared), the missing values are set to: **undefined.** Sometimes this is acceptable, but sometimes it is better to assign a default value to the parameter.

```
function (a=1, b=1) { // function code }
```

Arguments Object:

JavaScript functions have a built-in object called the arguments object. The argument object contains an array of the arguments used when the function was called (invoked).

```
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
  var i;
  var sum = 0;
  for (i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
}
```

Function Invocation:

The code inside a function is not executed when the function is **defined**. The code inside a function is executed when the function is **invoked (call)**.

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self-invoked)

We need to call function with (). If we do this it will return the result. But if we call the function without () it will return the definition of the function. Such as –

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
document.getElementById("demo").innerHTML = toCelsius; // function toCelsius(f) { return (5/9) * (f-32); }
```

When a function is called, the computer will run through the function's code and evaluate the result of calling the function. By default that resulting value is **undefined**.

```
function rectangleArea(width, height) {
  let area = width * height
}
console.log(rectangleArea(5, 7)) // Prints undefined
```

In JavaScript we can define functions as object methods.

```
var myObject = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}
```

The **fullName** method is a function. The function belongs to the object. **myObject** is the owner of the function. The thing called `this`, is the object that "owns" the JavaScript code. In this case the value of `this` is **myObject**. So, to invoke the object method we need access with object name. Such as-

```
myObject.fullName();
```

Methods:

Methods are actions we can perform. JavaScript provides a number of string methods.
We call, or use, these methods by appending an instance with a period (the **dot operator**), the name of the method, and opening and closing parentheses: i.e. 'example string'.methodName().

```
console.log('hello'.toUpperCase()); // Prints 'HELLO'
console.log('Hey'.startsWith('H')); // Prints true
```

Built-in methods of String-
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/prototype

Built-in Objects:

In addition to console, there are other objects built into JavaScript. Down the line, you'll build your own objects, but for now these "built-in" objects are full of useful functionality.
For example, if you wanted to perform more complex mathematical operations than arithmetic, JavaScript has the built-in **Math** object.

The great thing about objects is that they have methods! Let's call the **.random()** method from the built-in **Math** object.

```
console.log(Math.random()); // Prints a random number between 0 and 1
```

To generate a random number between 0 and 50, we could multiply this result by 50, like so:

```
Math.random() * 50;
```

The example above will likely evaluate to a **decimal**. To ensure the answer is a **whole number**, we can take advantage of another useful **Math** method called **Math.floor()**.

```
Math.floor(Math.random() * 50);
```

Built-in methods of Math-
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Built-in methods of Number-
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

Kelvin Weather:

```
const kelvin = 296;        //The forecast today is 293 Kelvin.
const celsius = kelvin - 273;        //convert Kelvin to Celsius
let fahrenheit = celsius * (9/5)+32;        //convert Celsius to Fahrenheit
fahrenheit = Math.floor(fahrenheit);     //round down the Fahrenheit temperature
console.log(`The temperature is ${fahrenheit} degrees Fahrenheit.`);
```

Dog Years:

```
const myAge = 29;        //My Age
let earlyYears = 2;        // First two years of human years
earlyYears *= 10.5;        //first two human years converted to dog years
let laterYears = myAge - 2;        // remaining years of human years
laterYears *= 4;        //remaining human years converted to dog years
console.log(earlyYears);
console.log(laterYears);
const myAgeInDogYears = earlyYears + laterYears;        // Convert my age human years to dog years
const myName = 'Golam Mostafa'.toLowerCase();        //convert my name to lower case
console.log(`My name is ${myName}. I am ${myAge} years old in human years which is ${myAgeInDogYears}
years old in dog years.`);
```

Conditional Statement (Control Flow)

A conditional statement checks specific condition(s) and performs a task based on the condition(s).

Conditional Statement:

As like Java, we can use if…else if…else statement in JavaScript.

```
let stopLight = 'yellow';
if (stopLight === 'red') {
  console.log('Stop!');
} else if (stopLight === 'yellow') {
  console.log('Slow down.');
} else if (stopLight === 'green') {
  console.log('Go!');
} else {
  console.log('Caution, unknown!');
}
```

Comparison Operators:

When writing conditional statements, sometimes we need to use different types of operators to compare values. These operators are called **comparison operators**. Here is a list of some handy comparison operators: -

- Less than: **<**
- Greater than: **>**
- Less than or equal to: **<=**
- Greater than or equal to: **>=**
- Is equal to: **=== (identity operator)**
- Is NOT equal to: **!==**

Comparison operators compare the value on the left with the value on the right. It can be helpful to think of comparison statements as questions. When the answer is "yes", the statement evaluates to **true**, and when the answer is "no", the statement evaluates to **false**.

```
'apples' === 'oranges' // false
```

Switch Case:

We can use switch case statement as alternative of if…else statement.

```
let groceryItem = 'papaya';
switch (groceryItem) {
  case 'tomato':
    console.log('Tomatoes are $0.49');
    break;
  case 'lime':
    console.log('Limes are $1.49');
    break;
  case 'papaya':
    console.log('Papayas are $1.29');
    break;
  default:
    console.log('Invalid item');
    break;
}          // Prints 'Papayas are $1.29'
```

Magic Eight Ball:

```
const userName = 'Mostafa';
userName!==''?console.log(`Hello, ${userName}`):console.log('Hello!');
```

```
const userQuestion = 'Will I become a werewolf tonight?';
console.log(`The ${userName} asked: ${userQuestion}`);
const randomNumber = Math.floor(Math.random() * 8 );
let eightBall = '';
if(randomNumber===0){
   eightBall = 'It is certain';
} else if(randomNumber===1){
   eightBall = '‘It is decidedly so';
} else if(randomNumber===2){
   eightBall = 'Reply hazy try again';
}else if(randomNumber===3){
   eightBall = 'Cannot predict now';
}else if(randomNumber===4){
   eightBall = 'Do not count on it';
}else if(randomNumber===5){
   eightBall = 'My sources say no';
}else if(randomNumber===6){
   eightBall = 'Outlook not so good';
}else if(randomNumber===7){
   eightBall = 'Signs point to yes';
}
console.log(eightBall);
```

Scope

- Scope is the idea in programming that some variables are accessible/inaccessible from other parts of the program.
- A block is the code found inside a set of curly braces {}. Blocks help us group one or more statements together and serve as an important structural marker for our code.
  A block of code could be a function, like this:

```
const logSkyColor = () => {
        let color = 'blue';
        console.log(color); // blue
};
```

  Observe the block in an `if` statement:

```
if (dusk) {
        let color = 'pink';
         console.log(color); // pink
};
```

- In global scope, variables are declared outside of blocks. These variables are called global variables. Because global variables are not bound inside a block, they can be accessed by any code in the program, including code in blocks.

```
const color = 'blue'
const returnSkyColor = () => {
  return color; // blue
};
console.log(returnSkyColor()); // blue
```

- When a variable is defined inside a block, it is only accessible to the code within the curly braces {}. We say that variable has block scope because it is only accessible to the lines of code within that block.
  Variables that are declared with block scope are known as local variables because they are only available to the code that is part of the same block.

```
const logSkyColor = () => {
        let color = 'blue';
        console.log(color); // blue
};
logSkyColor(); // blue
console.log(color); // ReferenceError
```

- When you declare global variables, they go to the global namespace. The global namespace allows the variables to be accessible from anywhere in the program. These variables remain there until the program finishes which means our global namespace can fill up really quickly.
  Scope pollution is when we have too many global variables that exist in the global namespace, or when we reuse variables across different scopes. Scope pollution makes it difficult to keep track of our different variables and sets us up for potential accidents. For example, globally scoped variables can collide with other variables that are more locally scoped, causing unexpected behavior in our code.
- We can declare global and local variable as same name. Like Java it didn't create ambiguity problem. If local variable have a declaration part then it is a new variable and it only accessible from the block. But global one accessible from outside of the block and other blocks. Even before the local variable declaration.

Loops

- A loop is a programming tool that repeats a set of instructions until a specified condition, called a stopping condition is reached. When we need to reuse a task in our code, we often bundle that action in a function. Similarly, when we see that a process has to repeat multiple times in a row, we write a loop. **Iterate** simply means "to repeat". As illustrated in the diagram, loops iterate or repeat an action until a specific condition is met. When the condition is met, the loop stops and the computer moves on to the next part of the program.
- Instead of writing out the same code over and over, loops allow us to tell computers to repeat a given block of code on its own. One way to give computers these instructions is with a **for** loop. The typical for loop includes an iterator variable that usually appears in all three expressions. The iterator variable is initialized, checked against the stopping condition, and assigned a new value on each loop iteration. Iterator variables can have any name, but it's best practice to use a descriptive variable name. A for loop contains three expressions separated by ; inside the parentheses:
  - An initialization starts the loop and can also be used to declare the iterator variable.
  - A stopping condition is the condition that the iterator variable is evaluated against— if the condition evaluates to true the code block will run, and if it evaluates to false the code will stop.
  - An iteration statement is used to update the iterator variable on each loop.

```
for (let counter = 0; counter < 4; counter++) {
  console.log(counter);
}
```

```
for (let counter = 4; counter >=0; counter--) {
  console.log(counter);
}
```

- How for loops iterate on arrays:

```
const animals = ['Grizzly Bear', 'Sloth', 'Sea Lion'];
for (let i = 0; i < animals.length; i++){
  console.log(animals[i]);
}
```

- When we have a loop running inside another loop, we call that a nested loop. One use for a nested for loop is to compare the elements in two arrays. **For each round of the outer for loop, the inner for loop will run completely**.

```
const myArray = [6, 19, 20];
const yourArray = [19, 81, 2];
for (let i = 0; i < myArray.length; i++) {
  for (let j = 0; j < yourArray.length; j++) {
    if (myArray[i] === yourArray[j]) {
      console.log('Both loops have the number: ' + yourArray[j])
    }
  }
};
```

- **While** loop syntax:

```
let counterTwo = 1;
while (counterTwo < 4) {
  console.log(counterTwo);
  counterTwo++;
}
```

- In some cases, we want a piece of code to run at least once and then loop based on a specific condition after its initial run. This is where the **do...while** statement comes in.
  A **do...while** statement says to do a task once and then keep doing it until a specified condition is no longer met. The **syntax** for a **do...while** statement looks like this:

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

- Imagine we're looking to adopt a dog. We plan to go to the shelter every day for a year and then give up. But what if we meet our dream dog on day 65? We don't want to keep going to the shelter for the next 300 days just because our original plan was to go for a whole year. In our code, when we want to stop a loop from continuing to execute even though the original stopping condition we wrote for our loop hasn't been met, we can use the keyword **break**. The **break** keyword allows programs to "break" out of the loop from within the loop's block.

```
for (let i = 0; i < 99; i++) {
  if (i > 2 ) {
     break;
  }
  console.log('Banana.');
}
console.log('Orange you glad I broke out the loop!');
```

break statements can be especially helpful when we're looping through large data structures! With breaks, we can add test conditions besides the stopping condition, and exit the loop when they're met.

High-Order Function

Functions as Data:

JavaScript functions behave like any other data type in the language; we can assign functions to variables, and we can reassign them to new variables.

```
const announceThatIAmDoingImportantWork = () => {
console.log("I'm doing very important work!");
};
```

What if we wanted to rename this function without sacrificing the source code? We can re-assign the function to a variable with a suitably short name:

```
const busy = announceThatIAmDoingImportantWork;
busy();  // This function call barely takes any space!
```

busy is a variable that holds a reference to our original function. If we could look up the address in memory of busy and the address in memory of announceThatIAmDoingImportantWork they would point to the same place.

In JavaScript, functions are **first class objects**. This means that, like other objects you've encountered, JavaScript **functions** can **have properties and methods**. Since functions are a type of object, they have properties such as .length and .name and methods such as .toString().
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function

Functions as Parameter:

Since functions can behave like any other type of data in JavaScript, it might not surprise you to learn that we can also pass functions (into other functions) as parameters. A higher-order function is a function that either accepts functions as parameters, returns a function, or both! We call the functions that get passed in as parameters and invoked callback functions because they get called during the execution of the higher-order function.
When we pass a function in as an argument to another function, we don't invoke it. Invoking the function would evaluate to the return value of that function call. With callbacks, we pass in the function itself by typing the function name without the parentheses (that would evaluate to the result of calling the function):

```
const timeFuncRuntime = funcParameter => {
  let t1 = Date.now();
  funcParameter();
  let t2 = Date.now();
  return t2 - t1;
}
const addOneToOne = () => 1 + 1;
timeFuncRuntime(addOneToOne);
```

Iterators

If we want to iterator or scan a list or array and wanted to know what each item was, we can do this by loops. However, we also have access to built-in array methods which make looping easier. The built-in JavaScript array methods that help us iterate are called iteration methods, at times referred to as iterators. Iterators are methods called on arrays to manipulate elements and return values.

The .forEach() Method:

Aptly named, .forEach() will execute the same code for each element of an array.

```
const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];
artists.forEach(function(artist) {
        console.log(artist + ' is one of my favorite artists.');
});
```

Let's explore the syntax of invoking .forEach():

- **artists.forEach()** calls the **forEach** method on the **artists** array.
- **.forEach()** takes an **argument of callback function**. Remember, a **callback function** is a function passed as an argument into another function.
- **.forEach()** loops through the array and executes the **callback function** for each element. During each execution, the current element is passed as an argument to the callback function.
- The **return** value for .forEach() will always be **undefined**.

Another way to pass a callback for **.forEach()** is to use **arrow function syntax**:

```
const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];
var a = artists.forEach(artist => {
        console.log(artist + ' is one of my favorite artists.');
});
console.log(a);  // undefined
```

We can also define a function beforehand to be used as the **callback function**:

```
var printArtists = artist => {
        console.log(`${artist} is one of my favorite artists.`);
}
const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];
artists.forEach(printArtists);
```

The .map() Method:

The second iterator we're going to cover is **.map()**. When **.map()** is called on an array, it takes an argument of a **callback function** and **returns a new array**!
.map() works in a similar manner to **.forEach()**— the major difference is that **.map() returns a new array** and **.forEach() returns undefined**.

```
const numbers = [1, 2, 3, 4, 5];
const bigNumbers = numbers.map(number => {
  return number * 10;
});
console.log(numbers); // Output: [1, 2, 3, 4, 5]
console.log(bigNumbers); // Output: [10, 20, 30, 40, 50]
```

In the example above:

- **numbers** is an array of numbers.
- **bigNumbers** will store the return value of calling **.map()** on **numbers**.
- **numbers.map** will iterate through each element in the **numbers** array and pass the element into the **callback function**.
- **return number * 10** is the code we wish to execute upon each element in the array. This will save each value from the **numbers** array, **multiplied by 10**, **to a new array**.
- Notice that the elements in **numbers** were **not altered** and **bigNumbers** is a **new array**.

```
const animals = ['Hen', 'elephant', 'llama', 'leopard', 'ostrich', 'Whale', 'octopus', 'rabbit', 'lion', 'dog'];
const secretMessage = animals.map(animal => {
  return animal[0];
});
console.log(secretMessage.join(''));
```

The .filter() Method:

Another useful iterator method is **.filter()**. Like **.map()**, **.filter() returns a new array**. However, **.filter() returns an array of elements after filtering out certain elements from the original array**. The **callback function for the .filter()** method **should return true or false** depending on the element that is passed to it. **The elements** that cause the callback function to **return true are added to the new array**.

```
const words = ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];
const shortWords = words.filter(word => {
  return word.length < 6;
});
console.log(words); // Output: ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];
console.log(shortWords); // Output: ['chair', 'music', 'brick', 'pen', 'door']
```

Here,

- **words** is an array that contains string elements.
- **const shortWords =** declares a new variable that will store the **returned array** from invoking **.filter()**.
- The **callback function** is an arrow function has a **single parameter**, **word**. Each element in the words array will be passed to this function as an argument.
- **word.length < 6**; is the **condition in the callback function**. Any word from the words array that has **fewer than 6 characters** will be **added to the shortWords array**.
- Observe how **words** was **not mutated**, i.e. changed, and **shortWords** is **a new array**.

The .findIndex() Method:

We sometimes want to find the location of an element in an array. That's where the **.findIndex()** method comes in! Calling **.findIndex()** on an array will **return the index of the first element that evaluates to true in the callback function**.

```
const jumbledNums = [123, 25, 78, 5, 9];
const lessThanTen = jumbledNums.findIndex(num => {
  return num < 10;
});
console.log(lessThanTen); // Output: 3
console.log(jumbledNums[3]); // Output: 5
```

- **jumbledNums** is an array that contains elements that are numbers.
- **const lessThanTen =** declares a new variable that stores the returned **index number** from invoking **.findIndex().**
- The **callback function** is an arrow function has a **single parameter, num**. Each element in the **jumbledNums** array will be passed to this function as an argument.
- **num < 10**; is the condition that elements are **checked against. .findIndex()** will **return the index of the first element which evaluates to true for that condition**.

If there isn't a single element in the array that satisfies the condition in the callback, then .findIndex() will return -1.

```
const greaterThan1000 = jumbledNums.findIndex(num => {
        return num === 1000;
});
console.log(greaterThan1000); // Output: -1
```

The .reduce() Method:

Another widely used iteration method is **.reduce()**. The **.reduce()** method **returns a single value after iterating through the elements of an array**, thereby reducing the array.

```
const numbers = [1, 2, 4, 10];
const summedNums = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue
})
console.log(summedNums) // Output: 17
```

- **numbers** is an array that contains numbers.
- **summedNums** is a variable that stores the **returned value of invoking .reduce()** on numbers.
- **numbers.reduce() calls the .reduce()** method on the numbers array and takes in a callback function as argument.
- The **callback function has two parameters, accumulator and currentValue**. The value of **accumulator starts off as the value of the first element in the array and the currentValue starts as the second element**. To see the value of accumulator and currentValue change, review the chart above.
- As **.reduce()** iterates through the array, **the return value of the callback function becomes the accumulator value for the next iteration, currentValue takes on the value of the current element in the looping process**.

The **.reduce()** method can also take an **optional second parameter** to set an **initial value for accumulator** (remember, the **first argument is the callback function**!).

```
const numbers = [1, 2, 4, 10];
const summedNums = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue
}, 100)  // <- Second argument for .reduce()
console.log(summedNums); // Output: 117
```

Iterator Documentation:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Iteration_methods

Objects

There are only **seven fundamental data types in JavaScript**, and **six** of those are the **primitive** data types: **string, number, boolean, null, undefined, and symbol**. With the **seventh** type, **objects**, we open our code to more complex possibilities. We can use JavaScript objects to model real-world things, like a basketball, or we can use objects to build the data structures that make the web possible.

Creating Object Literals:

Objects can be assigned to variables just like any JavaScript type. We use **curly braces**, `{}`, to designate an ***object literal***:

```
let spaceship = {}; // spaceship is an empty object
```

We fill an object with unordered data. This data is organized into **key-value pairs**. A key is like a variable name that points to a location in memory that holds a value.
A key's value can be of any data type in the language including functions or other objects.
We make a key-value pair by writing the key's name, or identifier, followed by a colon and then the value. We separate each key-value pair in an object literal with a comma (,). Keys are strings, but when we have a key that does not have any special characters in it, JavaScript allows us to omit the quotation marks:

```
let spaceship = {
 'Fuel Type': 'diesel',
  color: 'silver'
};
```

The **spaceship** object has two properties **Fuel Type** and **color**. **'Fuel Type'** has **quotation marks** because it contains a **space character**.

Accessing Properties:

There are two ways we can access an object's property.

- **dot notation:** With property dot notation, we write the object's name, followed by the dot operator and then the property name (key). It will return the value of the property. If we try to access a property that does not exist on that object, **undefined** will be returned.

```
let spaceship = {
  homePlanet: 'Earth',
  color: 'silver'
};
spaceship.homePlanet; // Returns 'Earth',
spaceship.color; // Returns 'silver',
spaceship.favoriteIcecream; // Returns undefined
```

- **Bracket Notation:** The second way to access a key's value is by using **bracket notation,** `[]`.To use **bracket notation** to access an object's property, we pass in the **property name (key) as a string**. We *must* use bracket notation when accessing keys that have numbers, spaces, or special characters in them. Without bracket notation in these situations, our code would throw an error.

```
let spaceship = {
  'Fuel Type': 'Turbo Fuel',
  'Active Duty': true,
  homePlanet: 'Earth',
  numCrew: 5
};
spaceship['Active Duty'];   // Returns true
spaceship['Fuel Type'];   // Returns  'Turbo Fuel'
spaceship['numCrew'];   // Returns 5
spaceship['!!!!!!!!!!!!!!!'];   // Returns undefined
```

With **bracket notation** you can also use a **variable inside the brackets** to select the keys of an object. This can be especially helpful when working with functions:

```
let spaceship = {
        'Fuel Type': 'Turbo Fuel',
         'Active Duty': true,
         homePlanet: 'Earth',
         numCrew: 5
};
let returnAnyProp = (objectName, propName) => {
        objectName[propName];
}
returnAnyProp(spaceship, 'homePlanet'); // Returns 'Earth'
```

If we tried to write our **returnAnyProp()** function with **dot notation** (**objectName.propName**) the computer would look for a key of **'propName'** on our object and **not the value** of the **propName** parameter.

Property Assignment:

Once we've defined an object, we're not stuck with all the properties we wrote. **Objects are mutable** meaning we can update them after we create them!
We can use either **dot notation**, or **bracket notation [],** and the **assignment operator, =** to add new key-value pairs to an object or change an existing property.
One of two things can happen with property assignment:

- If the property already exists on the object, whatever value it held before will be replaced with the newly assigned value.
- If there was no property with that name, a new property will be added to the object.

It's important to know that although we **can't reassign an object** declared with **const**, we can **still mutate it**, meaning we can add new properties and change the properties that are there.

```
const spaceship = {
        type: 'shuttle'
};
spaceship = {
        type: 'alien'
}; // TypeError: Assignment to constant variable.
spaceship.type = 'alien'; // Changes the value of the type property
spaceship.speed = 'Mach 5'; // Creates a new key of 'speed' with a value of 'Mach 5'
spaceship[ 'mission'] = 'Explore the universe'; // Creates a new key of 'mission' with a value of 'Explore the
universe'
```

You can delete a property from an object with the **delete** operator.

```
const spaceship = {
  'Fuel Type': 'Turbo Fuel',
  homePlanet: 'Earth',
  mission: 'Explore the universe'
};
delete spaceship.mission;  // Removes the mission property
```

Methods:

When the data stored on an object is a function we call that a method. A property is what an object has, while a method is what an object does. For example **console** is a **global javascript object** and **.log()** is a **method** on that object. **Math** is also a **global javascript object** and **.floor()** is a **method** on it.
We can include methods in our object literals by creating ordinary, comma-separated key-value pairs. The key serves as our method's name, while the value is an anonymous function expression.

```
const alienShip = {
  invade: function () {
    console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New
Xaculon.')
  }
};
```

With the new method syntax introduced in **ES6** we can **omit** the **colon** and the **function** keyword.

```
const alienShip = {
  invade () {
    console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New
Xaculon.')
  }
};
```

Object methods are invoked by appending the **object's name with the dot operator followed by the method name and parentheses**:

```
alienShip.invade(); // Prints 'Hello! We have come to dominate your planet. Instead of Earth, it shall be called
New Xaculon.'
```

Nested Objects:

In application code, objects are often nested— an object might have another object as a property which in turn could have a property that's an array of even more objects!

```
const spaceship = {
     telescope: {
        yearBuilt: 2018,
        model: '91031-XLT',
        focalLength: 2032
     },
    crew: {
        captain: {
            name: 'Sandra',
            degree: 'Computer Engineering',
            encourageTeam() { console.log('We got this!') }
         }
    },
    engine: {
        model: 'Nimbus2000'
     },
     nanoelectronics: {
         computer: {
            terabytes: 100,
            monitors: 'HD'
         },
        'back-up': {
           battery: 'Lithium',
           terabytes: 50
         }
    }
};
```

We can chain operators to access nested properties. We'll have to pay attention to which operator makes sense to use in each layer. It can be helpful to pretend you are the computer and evaluate each expression from left to right so that each operation starts to feel a little more manageable.

```
spaceship.nanoelectronics['back-up'].battery; // Returns 'Lithium'
```

Pass By Reference:

Objects are passed by reference. This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a **const** variable).

```
const spaceship = {
  homePlanet : 'Earth',
  color : 'silver'
};
let paintIt = obj => {
  obj.color = 'glorious gold'
};
paintIt(spaceship);
spaceship.color // Returns 'glorious gold'
```

Our function **paintIt()** permanently changed the **color** of our **spaceship** object. However, reassignment of the **spaceship** variable wouldn't work in the same way:

```
let spaceship = {
  homePlanet : 'Earth',
  color : 'red'
};
let tryReassignment = obj => {
  obj = {
    identified : false,
    'transport type' : 'flying'
  }
  console.log(obj) // Prints {'identified': false, 'transport type': 'flying'}
};
tryReassignment(spaceship) // The attempt at reassignment does not work.
spaceship // Still returns {homePlanet : 'Earth', color : 'red'};
spaceship = {
  identified : false,
  'transport type': 'flying'
}; // Regular reassignment still works.
```

Let's look at what happened in the code example:

- We declared this **spaceship** object with **let**. This allowed us to reassign it to a new object with **identified** and **'transport type'** properties with no problems.
- When we tried the same thing using a function designed to reassign the **object passed into it**, the reassignment **didn't stick** (even though calling **console.log()** on the object produced the expected result).
- When we passed **spaceship** into that function, **obj** became a reference to the **memory location** of the **spaceship** object, but not to the **spaceship** variable. This is because the **obj** parameter of the **tryReassignment() function** is a variable in its own right. The body of **tryReassignment()** has no knowledge of the **spaceship** variable at all!
- When we did the reassignment in the body of **tryReassignment()**, the **obj** variable came to refer to the memory location of the object {'identified' : false, 'transport type' : 'flying'}, while the **spaceship** variable was completely unchanged from its earlier value.

Looping Through Objects:

Loops are programming tools that repeat a block of code until a condition is met. We learned how to iterate through arrays using their numerical indexing, but the **key-value pairs in objects aren't ordered**! JavaScript has given us alternative solution for iterating through objects with the **for...in syntax** .
for...in will execute a given block of code for each property in an object.

```
let spaceship = {
   crew: {
   captain: {
      name: 'Lily',
      degree: 'Computer Engineering',
      cheerTeam() { console.log('You got this!') }
      },
   'chief officer': {
      name: 'Dan',
      degree: 'Aerospace Engineering',
      agree() { console.log('I agree, captain!') }
      },
   medic: {
      name: 'Clementine',
      degree: 'Physics',
      announce() { console.log(`Jets on!`) } },
   translator: {
      name: 'Shauna',
      degree: 'Conservation Science',
      powerFuel() { console.log('The tank is full!') }
      }
   }
};
for (let crewMember in spaceship.crew) {
   console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`)
};
//captain: Lily
//chief officer: Dan
//medic: Clementine
//translator: Shauna
for (let crewMember in spaceship.crew) { console.log(`${spaceship.crew[crewMember].name}:
${spaceship.crew[crewMember].degree}`)
};
//Lily: Computer Engineering
//Dan: Aerospace Engineering
//Clementine: Physics
//Shauna: Conservation Science
```

<u>The this Keyword:</u>

Objects are collections of related data and functionality. We store that functionality in methods on our objects:

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  }
};
goat.makeSound(); // Prints baaa
```

If we want to access **object properties** from **object method** it will throw **ReferenceError.**

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(dietType);
  }
};
goat.diet(); // Output will be "ReferenceError: dietType is not defined"
```

Because inside the scope of the **.diet()** method, we don't automatically have access to other properties of the **goat** object.
Here's where the **this** keyword comes to the rescue.
If we change the **.diet()** method to use the **this**, the **.diet()** works! :

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(this.dietType);
  }
};
goat.diet(); // Output: herbivore
```

The **this** keyword references the ***calling object*** which provides access to the **calling object's properties**. In the example above, the calling object is **goat** and by using **this** we're accessing the **goat** object itself, and then the **dietType** property of **goat** by using property dot notation.

```
const robot = {
 model : '1E78V2',
 energyLevel : 100,
 provideInfo () {
  return `I am ${this.model} and my current energy level is ${this.energyLevel}.`
 }
};
console.log(robot.provideInfo());         // I am 1E78V2 and my current energy level is 100.
```

Arrow Functions and this:

In **Arrow Function** if we want to access object properties from object method without **this** it will throw **ReferenceError.** But if we want to access object properties with **this** keyword it will show **undefined**.

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet: () => {
    console.log(this.dietType);
  }
};
goat.diet(); // Prints undefined
```

Arrow functions inherently **bind**, or **tie**, an **already defined this value to the function itself** that is **NOT the calling object**. In the code snippet above, the value of **this** is the **global object** (such as **window object** which have no **dietType** property), or **an object that exists in the global scope**, **which doesn't have a dietType property** and therefore **returns undefined**.
We need to access the value of dietType from diet() function with goat object instead of this-

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet: () => {
    console.log(goat.dietType);
  }
};
goat.diet();
```

Privacy:

Accessing and updating properties is fundamental in working with objects. However, there are cases in which we don't want other code simply accessing and updating an object's properties. When discussing privacy in objects, we define it as the idea that only certain properties should be mutable or able to change in value.
Certain languages have privacy built-in for objects, but JavaScript does not have this feature. Rather, JavaScript developers follow naming conventions that signal to other developers how to interact with a property. One common convention is to place an underscore _ before the name of a property to mean that the property should not be altered. Here's an example of using _ to prepend a property:

```
const bankAccount = {
  _amount: 1000
}
```

Even so, it is still possible to reassign _amount:

```
bankAccount._amount = 1000000;
```

Here is another example:

```
const robot = {
 _energyLevel: 100,
 recharge(){
   this._energyLevel += 30;
   console.log(`Recharged! Energy is currently at ${this._energyLevel}%.`)
 }
};
robot._energyLevel = 500;
robot.recharge(); // Recharged! Energy is currently at 530%.
```

Getters:

Getters are methods that **get** and **return** the internal **properties of an object**. But they can do more than just retrieve the value of a property! Let's take a look at a getter method:

```
const person = {
  _firstName: 'John',
  _lastName: 'Doe',
  get fullName() {
    if (this._firstName && this._lastName){
      return `${this._firstName} ${this._lastName}`;
    } else {
      return 'Missing a first name or a last name.';
    }
  }
}

// To call the getter method:
person.fullName; // 'John Doe'
```

Notice that in the getter method above:

- We use the **get** keyword followed by a function.
- We use an **if...else conditional** to check if both **_firstName** and **_lastName** exist (by making sure they both return **truthy** values) and then return a different value depending on the result.
- We can access the calling **object's internal properties** using **this**. In **fullName**, we're accessing both **this._firstName** and **this._lastName**.
- In the last line we call **fullName** on **person**. In general, getter methods **do not need** to be called **with** a set of **parentheses**. Syntactically, it looks like we're accessing a property.

Some notable advantages of using getter methods:

- Getters can perform an action on the data when getting a property.
- Getters can return different values using conditionals.
- In a getter, we can access the properties of the calling object using this.
- The functionality of our code is easier for other developers to understand.

Another thing to keep in mind when using getter (and setter) methods is that properties cannot share the same name as the getter/setter function. If we do so, then calling the method will result in an infinite call stack error. One workaround is to add an underscore before the property name like we did in the example above.

Setters:

Along with getter methods, we can also create setter methods which reassign values of existing properties within an object. Let's see an example of a setter method:

```
const person = {
  _age: 37,
  set age(newAge){
    if (typeof newAge === 'number'){
      this._age = newAge;
    } else {
      console.log('You must assign a number to age');
    }
  }
};
person.age = 40;
console.log(person._age); // Logs: 40
person.age = '40'; // Logs: You must assign a number to age
```

Notice that in the example above:

- We can perform a check for what value is being assigned to **this._age**.
- When we use the setter method, only values that are numbers will reassign **this._age**
- There are different outputs depending on what values are used to reassign **this._age**.
- Setter methods like **age** do not need to be called with a set of **parentheses**. Syntactically, it looks like we're reassigning the value of a property.

Like getter methods, there are similar advantages to using setter methods that include checking input, performing actions on properties, and displaying a clear intention for how the object is supposed to be used.
Nonetheless, even with a setter method, it is still possible to directly reassign properties. For example, in the example above, we can still set **._age** directly:

```
person._age = 'forty-five'
console.log(person._age); // Prints forty-five
```

Factory Functions:

A factory function is a function that returns an object and can be reused to make multiple object instances. Factory functions can also have parameters allowing us to customize the object that gets returned.

```
const monsterFactory = (name, age, energySource, catchPhrase) => {
  return {
    name: name,
    age: age,
    energySource: energySource,
    scare() {
      console.log(catchPhrase);
    }
  }
};
```

In the **`monsterFactory`** function above, it has four parameters and returns an object that has the properties: **`name`**, **`age`**, **`energySource`**, and **`scare()`**. To make an object that represents a specific monster like a **ghost**, we can call **`monsterFactory`** with the necessary arguments and assign the return value to a variable:

```
const ghost = monsterFactory('Ghouly', 251, 'ectoplasm', 'BOO!');
ghost.scare(); // 'BOO!'
```

Now we have a ghost object as a result of calling monsterFactory() with the needed arguments. With monsterFactory in place, we don't have to create an object literal every time we need a new monster. Instead, we can invoke the monsterFactory function with the necessary arguments to take over the world make a monster for us!
Here is another example:

```
const robotFactory = (model, mobile) =>{
 return {
  model: model,
  mobile: mobile,
  beep () {
   console.log('Beep Boop');
  }
 }
};
const tinCan = robotFactory('P-500', true);
tinCan.beep();  // Beep Boop
```

Property Value Shorthand:

ES6 introduced some new shortcuts for assigning properties to variables known as ***destructuring***.
In the previous exercise, we created a factory function that helped us create objects. We had to assign each property a key and value even though the key name was the same as the parameter name we assigned to it. To remind ourselves, here's a truncated version of the factory function:

```
const monsterFactory = (name, age) => {
  return {
    name: name,
    age: age
  }
};
```

Imagine if we had to include more properties, that process would quickly become tedious! But we can use a **destructuring** technique, called ***property value shorthand***, to save ourselves some keystrokes. The example below works exactly like the example above:

```
const monsterFactory = (name, age) => {
  return {
    name,
    age
  }
};
```

Here is another example-

```
function robotFactory(model, mobile){
 return {
   model,
   mobile,
   beep() {
     console.log('Beep Boop');
   }
 }
}

// To check that the property value shorthand technique worked:
const newRobot = robotFactory('P-501', false)
console.log(newRobot.model); //P-501
console.log(newRobot.mobile);//false
```

Destructured Assignment:

We often want to extract key-value pairs from objects and save them as variables.

```
const vampire = {
  name: 'Dracula',
  residence: 'Transylvania',
  preferences: {
    day: 'stay inside',
    night: 'satisfy appetite'
  }
};
```

If we wanted to extract the `residence` property as a variable, we could using the following code:

```
const residence = vampire.residence;
console.log(residence); // Prints 'Transylvania
```

However, we can also take advantage of a **destructuring** technique called ***destructured** assignment* to save ourselves some keystrokes. In **destructured** assignment we create a variable with the name of an object's key that is wrapped in curly braces `{ }` and assign to it the object. Take a look at the example below:

```
const { residence } = vampire;
console.log(residence); // Prints 'Transylvania'
```

We can even use **destructured** assignment to grab nested properties of an object:

```
const { day } = vampire.preferences;
console.log(day); // Prints 'stay inside'
```

Built-in Object Methods:

In the previous exercises we've been creating instances of objects that have their own methods. But, we can also take advantage of built-in methods for Objects!
For example, we have access to object instance methods like: **.hasOwnProperty()**, **.valueOf()**, and many more!
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#Methods

There are also useful Object class methods such as **`Object.assign()`**, **`Object.entries()`**, and **`Object.keys()`** just to name a few.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#Methods_of_the_Object_constructor

Class

JavaScript is an *object-oriented programming* (OOP) language we can use to model real-world items.
Classes are a tool that developers use to quickly produce similar objects.
Take, for example, an object representing a dog named **`halley`**. This dog's **`name`** (a key) is **`"Halley"`** (a value) and has an **`age`** (another key) of **`3`** (another value). We create the **`halley`** object below:

```
let halley = {
  _name: 'Halley',
  _behavior: 0,
  get name() {
    return this._name;
  },
  get behavior() {
    return this._behavior;
  },
  incrementBehavior() {
    this._behavior++;
  }
}
```

Now, imagine you own a dog daycare and want to create a catalog of all the dogs who belong to the daycare. Instead of using the syntax above for every dog that joins the daycare, we can create a `Dog` class that serves as a template for creating new `Dog` objects. For each new dog, you can provide a value for their name.
As you can see, classes are a great way to reduce duplicate code and debugging time.

Constructor:

We see similarities between class and object syntax, there is one important method that sets them apart. It's called the **constructor** method. JavaScript calls the **constructor()** method every time it creates a new **instance** of a class.

```
class Dog {
  constructor(name) {
    this.name = name;
    this.behavior = 0;
  }
}
```

- **Dog** is the name of our class. By convention, we capitalize and **CamelCase** class names.
- JavaScript will invoke the **constructor()** method every time we create a new **instance** of our **Dog class**.
- This **constructor()** method accepts one argument, **name**.
- Inside of the **constructor()** method, we use the **this** keyword. In the context of a class, **this** refers to an **instance** of that **class**. In the **Dog class**, we use **this** to set the value of the **Dog instance's name** property to the **name** argument.
- Under **this.name**, we create a property called **behavior**, which will keep track of the number of times a dog misbehaves. The **behavior** property is always initialized to **zero**.

Instance:

Now, we're ready to create class **instances**. An **instance** is an object that contains the property names and methods of a class, but with unique property values.

```
class Dog {
  constructor(name) {
    this.name = name;
    this.behavior = 0;
  }
}
const halley = new Dog('Halley'); // Create new Dog instance
console.log(halley.name); // Log the name value saved to halley
// Output: 'Halley'
```

- We use the **new** keyword to create an instance of our **Dog** class.
- We create a new variable named **halley** that will store an **instance** of our **Dog** class.
- We use the **new** keyword to generate a **new instance** of the **Dog** class. The **new** keyword calls the **constructor(),** runs the code inside of it, and then **returns** the **new instance**.
- We pass the **'Halley'** string to the **Dog constructor**, which sets the **name** property to **'Halley'**.
- Finally, we log the value saved to the **name** key in our **halley** object, which logs **'Halley'** to the console.

If we print instance of class it will print the constructor properties with the class name.

```
class Surgeon {
  constructor(name, department) {
    this._name = name;
    this._department = department;
    this._remainingVacationDays = 20;
  }
}
const surgeonCurry = new Surgeon('Curry', 'Cardiovascular');
console.log(surgeonCurry);
//Surgeon {
//  _name: 'Curry',
//  _department: 'Cardiovascular',
//  _remainingVacationDays: 20 }
```

Methods:

We can declare getter and methods within **class**. Class method and getter syntax is the same as it is for objects **except we can't include commas between methods**.

```
class Dog {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }
  get name() {
    return this._name;
  }
  get behavior() {
    return this._behavior;
  }
  incrementBehavior() {
    this._behavior++;
  }
}
```

In the example above, we add **getter** methods for `name` and `behavior`. Notice, we also prepended our property names with **underscores** (`_name` and `_behavior`), which indicate these **properties should not be accessed directly**. Under the **getters**, we add a method named `.incrementBehavior()`. When you call `.incrementBehavior()` on a **Dog** instance, it adds `1` to the `_behavior` property. Between each of our methods, we did ***not*** **include commas**.

Method Calls:

To call getter or method of class firstly we need to create instance of the class. Then using the instance with a period we can access or call the method of the class as like as object method call.

```
class Dog {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }
  get name() {
    return this._name;
  }
  get behavior() {
    return this._behavior;
  }
  incrementBehavior() {
    this._behavior++;
  }
}
let nikko = new Dog('Nikko'); // Create dog named Nikko
nikko.incrementBehavior(); // Add 1 to nikko instance's behavior
let bradford = new Dog('Bradford'); // Create dog name Bradford
console.log(nikko.behavior); // Logs 1 to the console
console.log(bradford.behavior); // Logs 0 to the console
```

In the example above, we create two new `Dog` **instances**, `nikko` and `bradford`. Because we increment the **behavior** of our `nikko` instance, but not `bradford`, accessing `nikko.behavior` returns `1` and accessing `bradford.behavior` returns `0`.

Inheritance:

Note

- JavaScript properties need to call with **functionVariableOrObjectName.propertyName** without parenthesis. Such as – **arr.length**. Method need to call with **functionVariableOrObjectName.methodName()** with parenthesis. Such as- **Math.random();** and function need to call with **functionName();**
-