DATA STRUCTURES & ALGORITHM IN JAVASCRIPT

Jayson N. Reales

Instructor

What is JavaScript

JavaScript was initially created to "***make web pages alive***".

The programs in this language are called ***scripts***. They can be written right in a web page's HTML and run automatically as the page loads.

Scripts are provided and executed as plain text. They don't need special preparation or compilation to run.

In this aspect, JavaScript is very different from another language called Java.

Why called JavaScript?

When JavaScript was created, it initially had another name: "**LiveScript**". But Java was very popular at that time, so it was decided that positioning a new language as a "**younger brother**" of Java would help.

But as it evolved, JavaScript became a fully independent language with its own specification called **ECMAScript**, and now it has no relation to Java at all.

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

The browser has an embedded engine sometimes called a "**JavaScript virtual machine**".

Different engines have different "codenames". For example:

- **V8** – in Chrome, Opera and Edge.
- **SpiderMonkey** – in Firefox.
- …There are other codenames like "Chakra" for IE, "JavaScriptCore", "Nitro" and "SquirrelFish" for Safari, etc.

The terms above are good to remember because they are used in developer articles on the internet. We'll use them too. For instance, if "a feature X is supported by V8", then it probably works in Chrome, Opera and Edge.

How do engines work?

Engines are complicated. But the basics are easy.

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

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

What can in-browser JavaScript do?

Modern JavaScript is a "**safe**" programming language. It does not provide **low-level access** to memory or the CPU, because it was initially created for browsers which do not require it.

JavaScript's capabilities greatly depend on the environment it's running in. For instance, **Node.js** supports functions that allow JavaScript to read/write arbitrary files, perform network requests, etc.

In-browser JavaScript can do everything related to webpage manipulation, interaction with the user, and the webserver.

For instance, in-browser JavaScript is able to:

- Add new HTML to the page, change the existing content, modify styles.
- React to user actions, run on mouse clicks, pointer movements, key presses.
- Send requests over the network to remote servers, download and upload files (so-called AJAX and COMET technologies).
- Get and set cookies, ask questions to the visitor, show messages.
- Remember the data on the client-side ("local storage").

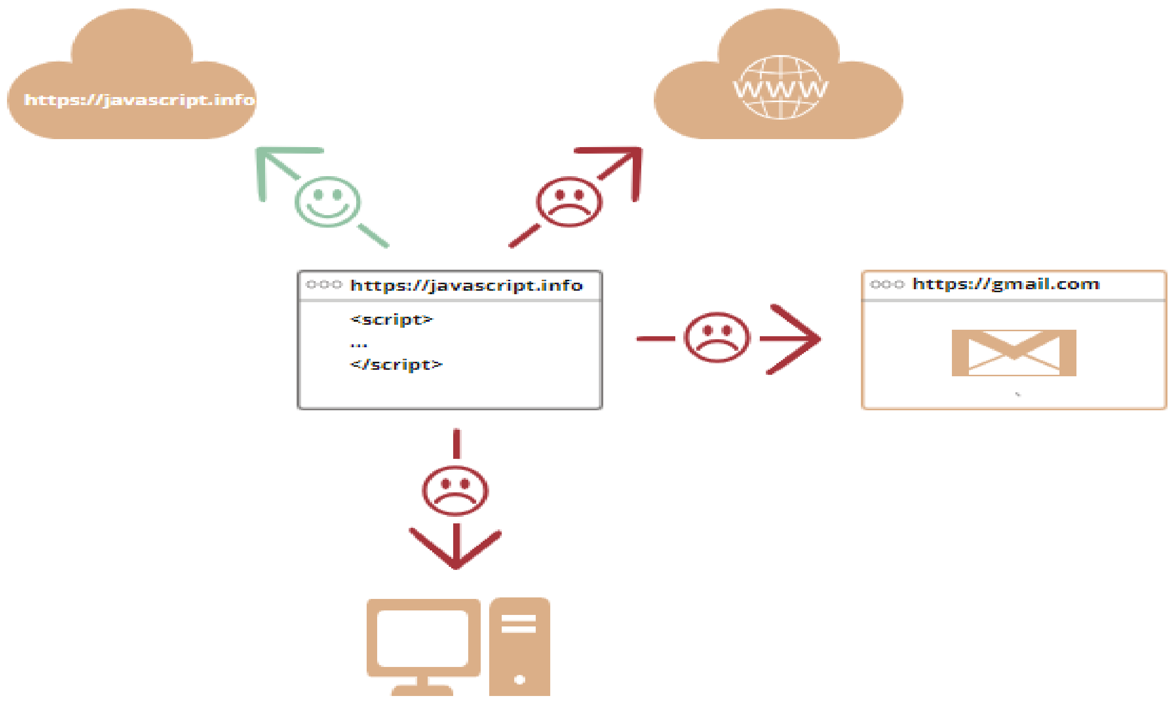
What CAN'T in-browser JavaScript do?

JavaScript's abilities in the browser are limited to protect the user's safety. The aim is to prevent an evil webpage from accessing private information or harming the user's data.

Examples of such restrictions include:

- JavaScript on a webpage may not read/write arbitrary files on the hard disk, copy them or execute programs. It has no direct access to OS functions.
- Modern browsers allow it to work with files, but the access is limited and only provided if the user does certain actions, like "dropping" a file into a browser window or selecting it via an <input> tag.
- There are ways to interact with the camera/microphone and other devices, but they require a user's explicit permission. So a JavaScript-enabled page may not sneakily enable a web-camera, observe the surroundings and send the information to the NSA.

- Different tabs/windows generally do not know about each other. Sometimes they do, for example when one window uses JavaScript to open the other one. But even in this case, JavaScript from one page may not access the other page if they come from different sites (from a different domain, protocol or port).
- This is called the “Same Origin Policy”. To work around that, both pages must agree for data exchange and must contain special JavaScript code that handles it. We’ll cover that in the tutorial.
- This limitation is, again, for the user’s safety. A page from http://anysite.com which a user has opened must not be able to access another browser tab with the URL http://gmail.com, for example, and steal information from there.
- JavaScript can easily communicate over the net to the server where the current page came from. But its ability to receive data from other sites/domains is crippled. Though possible, it requires explicit agreement (expressed in HTTP headers) from the remote side. Once again, that’s a safety limitation.



Such limitations do not exist if JavaScript is used outside of the browser, for example on a server. Modern browsers also allow plugins/extensions which may ask for extended permissions.

What makes JavaScript unique?

There are at least three great things about JavaScript:

- Full integration with HTML/CSS.
- Simple things are done simply.
- Supported by all major browsers and enabled by default.

JavaScript is the only browser technology that combines these three things.

That's what makes JavaScript unique. That's why it's the most widespread tool for creating browser interfaces.

That said, JavaScript can be used to create servers, mobile applications, etc.

Languages "over" JavaScript

The syntax of JavaScript does not suit everyone's needs. Different people want different features.

That's to be expected, because projects and requirements are different for everyone.

So, recently a plethora of new languages appeared, which are transpiled (converted) to JavaScript before they run in the browser.

Modern tools make the transpilation very fast and transparent, actually allowing developers to code in another language and auto-converting it "under the hood".

Examples of such languages:

- **CoffeeScript** is "syntactic sugar" for JavaScript. It introduces shorter syntax, allowing us to write clearer and more precise code. Usually, Ruby devs like it.
- **TypeScript** is concentrated on adding "strict data typing" to simplify the development and support of complex systems. It is developed by Microsoft.
- **Flow** also adds data typing, but in a different way. Developed by Facebook.
- **Dart** is a standalone language that has its own engine that runs in non-browser environments (like mobile apps), but also can be transpiled to JavaScript. Developed by Google.
- **Brython** is a Python transpiler to JavaScript that enables the writing of applications in pure Python without JavaScript.
- **Kotlin** is a modern, concise and safe programming language that can target the browser or Node.

There are more. Of course, even if we use one of these transpiled languages, we should also know JavaScript to really understand what we're doing.

Language Syntax

Language syntax refers to the set of rules that dictate how programs written in a particular programming language must be structured. This can include **rules** for how to **declare variables**, how to **call functions**, how to **structure control flow statements**, and so on. Syntax varies significantly between different programming languages, so it is critical to grasp the specific syntax of the language you are using. It's similar to grammar in human languages - putting words in the wrong order or including extraneous punctuation can make a sentence hard to understand, and the same applies to programming. **Incorrect syntax leads to syntax errors** which prevent your code from executing.

JavaScript Syntax

JavaScript syntax comprises a set of rules that define how to construct a JavaScript code. JavaScript can be implemented using JavaScript statements that are placed within the <script> ... </script> HTML tags in a web page. You can place the <script> tags, containing your JavaScript, anywhere within your web page, but it is normally recommended that you should keep it within the <head> tags. The <script> tag alerts the browser program to start interpreting all the text between these tags as a script. A simple syntax of your JavaScript will appear as follows.

```
<script>
  // This is the part where you put your JavaScript code
</script>
```

JavaScript Values

In JavaScript, you can have two types of values.

- Fixed values (Literals)
- Variables (Dynamic values)

JavaScript Literals

In the below code, 10 is a Number literal and ‘Hello’ is a string literal.

```
<script>
    document.write(10); // Number literal
    document.write("<br />"); // To add line-break
    document.write("Hello"); //String literal
</script>
```

Output ???

JavaScript Variables

In JavaScript, variables are used to store the dynamic data.

You can use the below keyword to define variables in JavaScript.

- var
- let
- const

You can use the assignment operator (equal sign) to assign values to the variable.

In the below code, variable a contains the numeric value, and variable b contains the text (string).

```
<script>
    let a = 5; // Variable Declaration
    document.write(a); // Using variable
    document.write("<br>");
    let b = "One";
    document.write(b);
  </script>
```

Output???

Whitespace and Line Breaks

JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

Semicolons are Optional

Simple statements in JavaScript are generally followed by a semicolon character, just as they are in C, C++, and Java. JavaScript, however, allows you to omit this semicolon if each of your statements are placed on a separate line. For example, the following code could be written without semicolons.

Case Sensitivity

JavaScript is a case-sensitive language. This means that the language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

So the identifiers **Time** and **TIME** will convey different meanings in JavaScript.

JavaScript and Camel Case

- **v** − We can create variables like SmartWatch, MobileDevice, WebDrive, etc. It represents the upper camel case string.
- **Lower Camel Case** − JavaScript allows developers to use variable names and expression names like smartwatch, mobileDevice, webDriver, etc. Here the first character is in lowercase.
- **Underscore** − We can use underscore while declaring variables like smart_watch, mobile_device, web_driver, etc.

Comments in JavaScript

JavaScript supports both C-style and C++-style comments, Thus –

- Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters /* and */ is treated as a comment. This may span multiple lines.
- JavaScript also recognizes the HTML comment opening sequence <!--. JavaScript treats this as a single-line comment, just as it does the // comment.
- The HTML comment closing sequence --> is not recognized by JavaScript so it should be written as //-->.

JAVASCRIPT OPERATORS

Operators in JavaScript

JavaScript contains various arithmetic, logical, bitwise, etc. operators. We can use any operator in JavaScript.

1. Arithmetic Operators
    1. +
    2. –
    3. /
    4. *
    5. %
2. Logical Operator
    1. AND Operator or &&
    2. OR Operator or ||
    3. NOT Operator or !

Example

In this example, we have defined var1 and va2 and initialized them with number values. After that, we use the '*' operator to get the multiplication result of var1 and var2.

```
<script>

    var1 = 10

    var2 = 20

    var3 = var1 * var2;

    var4 = 10 + 20;

    console.log(var3, " " ,var4);

</script>
```

Output???

JAVASCRIPT CONTROL STRUCTURES

Control Structures

Control structures are fundamental elements in most programming languages that facilitate the flow of control through a program. There are three main types of control structures: Sequential, Selection and Iteration. Sequential control structures are the default mode where instructions happen one after another. Selection control structures (often called “conditional” or “decision” structures) allow one set of instructions to be executed if a condition is true and another if it’s false. These typically include if...else statements. Iteration control structures (also known as “loops”) allow a block of code to be repeated multiple times. Common loop structures include for, while, and do...while loops. All these control structures play a vital role in shaping the program logic.

JavaScript Control Structures

- If Statement
- Using If-Else Statement
- Using Switch Statement
- Using the Ternary Operator (Conditional Operator)
- Using For loop

Approach 1: If Statement

In this approach, we are using an if statement to check a specific condition, the code block gets executed when the given condition is satisfied.

Syntax:

```
if ( condition_is_given_here ) {

   // If the condition is met, the code will get executed.

}
```

Approach 2: If-Else Statement

The if-else statement will perform some action for a specific condition. If the condition met, then a particular code of action will be executed otherwise it will execute another code of action that satisfies that particular condition.

Syntax:

```
if (condition1) {
   // Executes when condition1 is true
} else {
  // Executes when condition1 is false
}
```

Approach 3: Switch Statement

The switch case statement in JavaScript is also used for decision-making purposes. In some cases, using the switch case statement is seen to be more convenient than if-else statements.

```
switch (expression) {
   case value1:
      statement1;
      break;
   case value2:
      statement2;
      break;
   .
   .
   case valueN:
      statementN;
      break;
   default:
      statementDefault;
}
```

Approach 4: Using the Ternary Operator (Conditional Operator)

The conditional operator, also referred to as the ternary operator (?:), is a shortcut for expressing conditional statements in JavaScript.

Syntax:

```
condition ? value if true : value if false
```

Approach 5: Using For Loop

In this approach, we are using for loop in which the execution of a set of instructions repeatedly until some condition evaluates and becomes false

Syntax:

```
for (statement 1; statement 2; statement 3) {
   // Code here . . .
}
```

Programming Fundamentals

Programming Fundamentals are the basic concepts and principles that form the foundation of any computer programming language. These include understanding variables, which store data for processing, control structures such as loops and conditional statements that direct the flow of a program, data structures which organize and store data efficiently, and algorithms which step by step instructions to solve specific problems or perform specific tasks.

JAVASCRIPT FUNCTIONS

Functions

Functions in programming are named sections of a program that perform a specific task. They allow us to write a piece of code once and reuse it in different places throughout the program, making our code more modular and easier to maintain. Functions often take in input, do something with it, and return output. Functions can be categorized into four main types: built-in functions (like print(), provided by the programming language), user-defined functions (written by the user for a specific use case), anonymous functions (also known as lambda functions, which are not declared using the standard def keyword), and higher-order functions (functions that take other functions as arguments or return a function).

JavaScript Function

A JavaScript function is a block of code designed to perform a particular task. It encapsulates a set of instructions that can be reused throughout a program. Functions can take parameters, execute statements, and return values, enabling code organization, modularity, and reusability in JavaScript programming. A JavaScript function is executed when "something" invokes it (calls it).

Example: A basic JavaScript function, here we create a function that divides the 1st element by the second element.

```
function myFunction(g1, g2) {
    return g1 / g2;
}
const value = myFunction(8, 2); // Calling the function
console.log(value);
```

Output: 4

Syntax:

```
function functionName(Parameter1, Parameter2, ...) {
    // Function body
}
```

To create a function in JavaScript, we have to first use the keyword function, separated by the name of the function and parameters within parenthesis. The part of the function inside the curly braces {} is the body of the function. In JavaScript, functions can be used in the same way as variables for assignments, or calculations.

Function Invocation:

- Triggered by an event (e.g., a button click by a user).
- When explicitly called from JavaScript code.
- Automatically executed, such as in self-invoking functions.

Function Definition:

Before, using a user-defined function in JavaScript we have to create one. We can use the above syntax to create a function in JavaScript. A function definition is sometimes also termed a function declaration or function statement. Below are the rules for creating a function in JavaScript:

- Every function should begin with the keyword function followed by,
- A user-defined function name that should be unique,
- A list of parameters enclosed within parentheses and separated by commas,
- A list of statements composing the body of the function enclosed within curly braces {}.

Example Declaration

```
function calcAddition(number1, number2) {
    return number1 + number2;
}
console.log(calcAddition(6,9)); // This is where you call the function
```

Output: 15

1. This function accepts two numbers as parameters and returns the addition of these two numbers.
2. Accessing the function with just the function name without () will return the function object instead of the function result.

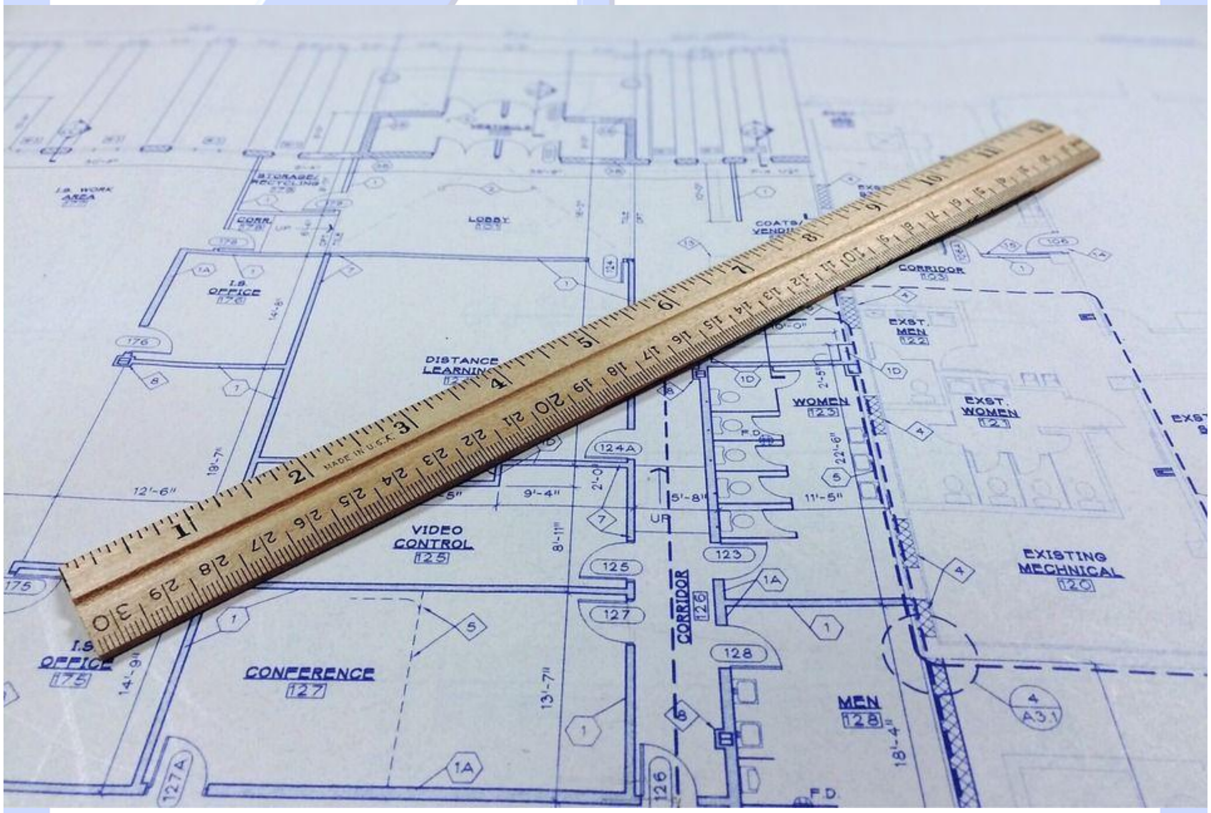
JAVASCRIPT OBJECT-ORIENTED BASICS

What is a Class?

You can think of a class like a blueprint of a house. A class is not a real world object but we can create objects from a class. It is like a template for an object.

We can create classes using the class keyword which is reserved keyword in JavaScript. Classes can have their own properties and methods. We will study how to create a class in detail shortly. This is just a high level overview of a class.

Let's take an example. Below is a blueprint for a house (like a class).



What is an Object?

An object is an instance of a class. Now with the help of the house class we can construct a house. We can construct multiple houses with the help of same house class.

Example of Classes and Objects in Action

Let's take a simple example to understand how classes and objects work.

The below example has nothing to do with JavaScript syntax. It is just to explain classes and objects. We will study the syntax of OOP in JavaScript in a bit.

Consider a Student Class. Student can have properties like **name**, **age**, **standard**, and so on, and functions like **study**, **play**, and do **homework**.

```
class Student{
 // Data (Properties)
 Name
 Age
 Standard

 // Methods (Action)
 study(){
 // Study
 }

 Play(){
 // Play
 }

 doHomeWork(){
 // Do Home Work
 }

}
```

With the help of the above class, we can have multiple students or instances.

Here’s info for Student - 01

```
// Student 1
{
  Name = "John"
  Age = 15
  Standard = 9

  study(){
   // Study
  }

  Play(){
   // Play
  }

  doHomeWork(){
   // Do Home Work
  }
}
```

Here’s info for Student - 01

```
// Student 2
{
 Name = "Gorge"
 Age = 18
 Standard = 12

 study(){
  // Study
 }

 Play(){
  // Play
 }

 doHomeWork(){
  // Do Home Work
 }
}
```

How Do We Actually Design a Class?

There is no perfect answer to this question. But we can get help from some OOP principles when designing our classes.

There are 4 main principles in OOP, and they are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

We will dive deep into these concepts in JavaScript below. But first, let's get a high level overview of these concepts to understand them better.

A **constructor** is a special function that creates and initializes an object instance of a class.

What Does Abstraction Mean in OOP?

Abstraction means hiding certain details that don't matter to the user and only showing essential features or functions.

```
// Define an abstract class Animal

function Animal() {

    if (this.constructor === Animal) {

        throw new Error(`Cannot instantiate

        abstract class Animal`);

    }
    this.makeSound = function () {
        throw new Error(`Cannot call abstract
        method makeSound from Animal`);
    };
}

// Create a concrete class Dog that extends Animal
function Dog(name) {
    Animal.call(this);
    this.name = name;

    this.makeSound = function () {
        console.log(`${this.name} barks`);
    };
}

// Inherit from the abstract class
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Create an instance of the Dog class
let dog = new Dog("Buddy");
dog.makeSound();

// Try to create an instance of
// the abstract class Animal
```

```
try {
    let animal = new Animal();
} catch (error) {
    console.error(error.message);
}
```

What Does Encapsulation Mean in OOP?

Encapsulation is a fundamental concept in object-oriented programming that refers to the practice of hiding the internal details of an object and exposing only the necessary information to the outside world.

```
class BankAccount {
    constructor(accountNumber, accountHolderName, balance) {
        this._accountNumber = accountNumber;
        this._accountHolderName = accountHolderName;
        this._balance = balance;
    }

    showAccountDetails() {
        console.log(`Account Number: ${this._accountNumber}`);
        console.log(`Account Holder Name:
${this._accountHolderName}`);
        console.log(`Balance: ${this._balance}`);
    }

    deposit(amount) {
        this._balance += amount;

        this.showAccountDetails();

    }


    withdraw(amount) {

        if (this._balance >= amount) {

            this._balance -= amount;

            this.showAccountDetails();

        } else {

            console.log("Insufficient Balance");

        }

    }

}


let myBankAccount = new BankAccount("123456", "John Doe", 1000);

myBankAccount.deposit(500);

// Output: Account Number: 123456 Account Holder Name:

//John Doe Balance: 150
```

What Does Inheritance Mean in OOP?

Inheritance makes all properties and methods available to a child class. This allows us to reuse common logic and to model real-world relationships. We will discuss inheritance in further section of this article with practical example.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

let myCar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = myCar.show();
```

What Does Polymorphism Mean in OOP?

Polymorphism is one of the core concepts of object-oriented programming languages where poly means many and morphism means transforming one form into another. Polymorphism means the same function with different signatures is called many times. In real life, for example, a boy at the same time may be a student, a class monitor, etc. So a boy can perform different operations at the same time. This is called polymorphism.

Features of Polymorphism:

- Programmers can use the same method name repeatedly.
- Polymorphism has the effect of reducing the number of functionalities that can be paired together.

Inheritance Polymorphism in JavaScript: In this example, we will create three functions with the same name and different operations. This program shows JavaScript Inheritance polymorphism.

Example: This example perform javascript inheritance polymorphism.

```
class firstClass {
    add() {
        console.log("First Method")
    }
}
class secondClass extends firstClass {
    add() {
        console.log(30 + 40);
    }
}
class thirdClass extends secondClass {
    add() {
        console.log("Last Method")
    }
}
let ob = new firstClass();
let ob2 = new secondClass();

let ob3 = new thirdClass();

ob.add();

ob2.add();

ob3.add();
```

JAVASCRIPT

PSEUDO CODE

What is Pseudocode?

Pseudocode literally means 'fake code'. It is an informal and contrived way of writing programs in which you represent the sequence of actions and instructions (aka algorithms) in a form that humans can easily understand.

You see, computers and human beings are quite different, and therein lies the problem.

The language of a computer is very rigid: you are not allowed to make any mistakes or deviate from the rules. Even with the invention of high-level, human-readable languages like JavaScript and Python, it's still pretty hard for an average human developer to reason and program in those coding languages.

With pseudocode, however, it's the exact opposite. You make the rules. It doesn't matter what language you use to write your pseudocode. All that matters is comprehension.

In pseudocode, you don't have to think about semi-colons, curly braces, the syntax for arrow functions, how to define promises, DOM methods and other core language principles. You just have to be able to explain what you're thinking and doing.

Benefits of Writing Pseudocode

When you're writing code in a programming language, you'll have to battle with strict syntax and rigid coding patterns. But you write pseudocode in a language or form with which you're very familiar. Since pseudocode is an informal method of program design, you don't have to obey any set-out rules. You make the rules yourself.

Pseudocode acts as the bridge between your brain and computer's code executor. It allows you to plan instructions which follow a logical pattern, without including all of the technical details.

Pseudocode is a great way of getting started with software programming as a beginner. You won't have to overwhelm your brain with coding syntax.

In fact, many companies organize programming tests for their interviewees in pseudocode. This is because the importance of problem solving supersedes the ability to 'hack' computer code.

You can get quality code from many platforms online, but you have to learn problem solving and practice it a lot.

Planning computer algorithms with pseudocode makes you meticulous. It helps you explain exactly what each line in a software program should do. This is possible because you are in full control of everything, which is one of the great features of pseudocode.

DATA STRUCTURES

What is a data structure?

In computer science, a data structure is a **format to organize**, **manage** and **store data** in a way that allows efficient access and modification. More precisely, a data structure is a **collection of data values**, the **relationships** among them, and the **functions** or **operations** that can be applied to that data.

Importance of Data Structures

Data structures bring together the data elements in a logical way and facilitate the effective use, persistence and sharing of data. They provide a formal model that describes the way the data elements are organized. Data structures are the building blocks for more sophisticated applications.

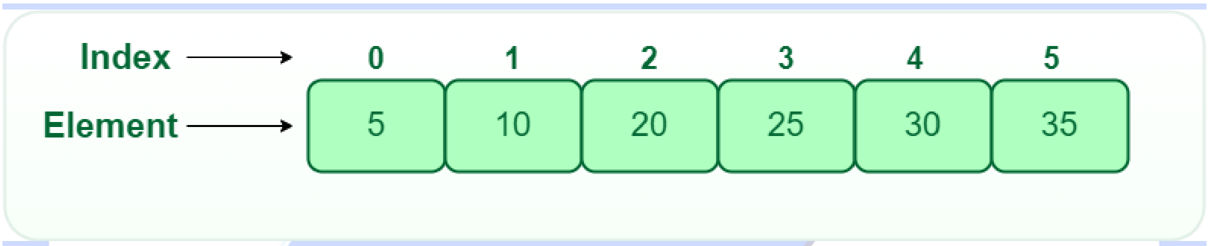
Basic Data Structures

The five main types of basic data structures are: Arrays, Linked Lists, Stacks, Queues, and Hash Tables.

1. **Arrays** - are static data structures that store elements of the same type in contiguous memory locations.
2. **Linked Lists** - are dynamic data structures that store elements in individual nodes, with each node pointing to the next.
3. **Stacks** - follow the Last-In-First-Out principle (LIFO) and primarily assist in function calls in most programming languages.
4. **Queues** - operate on the First-In-First-Out principle (FIFO) and are commonly used in task scheduling.
5. **Hash Tables** - store key-value pairs allowing for fast insertion, deletion, and search operations.

Arrays

Array is a variable that can **store multiple values**. The **values** inside an array is called an **element**. And the **number** where an element is **located** is called the **index**.



Each item can be accessed through its **index** (position) number. Arrays ***always start at index 0***, so in an array of 6 elements we could access the 3rd element using the index number 2.

Declaration of an Array: There are basically two ways to declare an array.

Method 1: let arrayName = [value1, value2, ...];

Method 2: let arrayName = new Array();

Creating Array

```
// Array of Strings
let names = ["Alenere", "David", "Jaymar", "Maye"];

// Array of Number
let numbers = [3, 6.5, 7, 8.6, 9, 10.1];

// Array of Mixed Datatypes
let numbers = ["Alenere", 55, "David", 8.6, 9, "Maye"];

// Empty Array
let dummy = [];
```

You can **create** any array with different **datatypes**.

Types of Array Operations:

1. **Traversal**: Accessing each element of an array one by one.
2. **Insertion**: Inserting or adding a new element in an array.
3. **Deletion**: Deleting or removing element from the array.
4. **Searching**: Search for an element in the array or involves in finding whether a specific element exists in the array.
5. **Sorting**: Involves in arranging the elements of the array in specific order, such as ascending or descending order.

What will be output in the console?

```
const arr = ['a', 'b', 'c', 'd']
console.log(arr[2])
```

The length property of an array is defined as the number of elements it contains. If the array contains 4 elements, we can say the array has a length of 4.

```
const arr = ['a', 'b', 'c', 'd']
console.log(arr.length) // 4
```

In some programming languages, the user can only store values of the same type in one array and the length of the array has to be defined at the moment of its creation and can't be modified afterwards.

In JavaScript that's not the case, as we can store values of any type in the same array and the length of it can be dynamic (it can grow or shrink as much as necessary).

```
const arr = ['store', 1, 'whatever', 2, 'you want', 3]
```

Any data type can be stored in an array, and that includes arrays too. An array that has other arrays within itself is called a multidimensional array.

```
const arr = [
   [1,2,3],
   [4,5,6],
   [7,8,9],
]
```

In JavaScript, arrays come with many built-in properties and methods we can use with different purposes, such as adding or deleting items from the array, sorting it, filtering its values, know its, length and so on. As I mentioned, in arrays, each element has an **index** defined by its position in the array. When we add a new item at the end of the array, it just takes the index number that follows the previous last item in the array.

But when we add/delete a new item at the beginning or the middle of the array, the indexes of all the elements that come after the element added/deleted **have to be changed**. This of course has a computational cost, and is one of the weaknesses of this data structure.

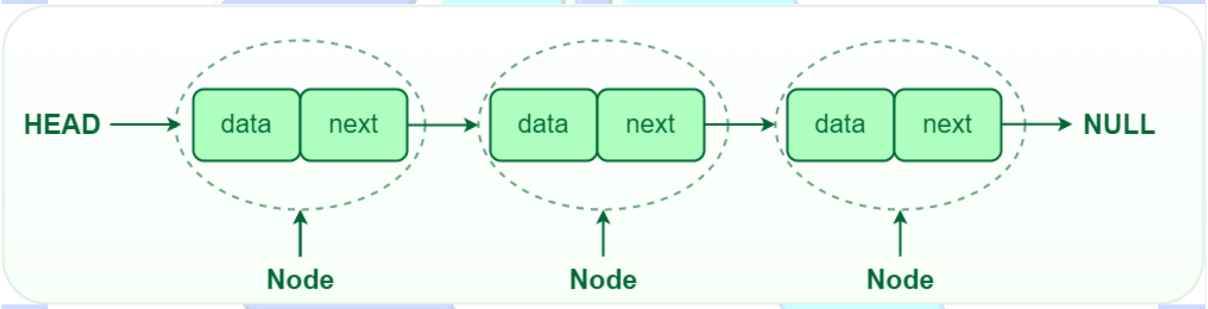
Arrays are useful when we have to store individual values and add/delete values from the end of the data structure. But when we need to add/delete from any part of it, there are other data structures that perform more efficiently.

Types of Array

1. Associative Array
2. Multi-dimensional Array

Linked List

A linked list is a linear data structure, Unlike arrays, linked list elements are not stored at a contiguous location. it is basically chains of nodes, each node contains information such as data and a pointer to the next node in the chain. In the linked list there is a head pointer, which points to the first element of the linked list, and if the list is empty then it simply points to null or nothing.



Types of Linked List operations:

1. Traversal: We can traverse the entire linked list starting from the head node. If there are n nodes then the time complexity for traversal becomes O(n) as we hop through each and every node.
2. Insertion: Insert a key to the linked list. An insertion can be done in **3 different ways**; insert at the beginning of the list, insert at the end of the list and insert in the middle of the list.
3. Deletion: Removes an element x from a given linked list. You cannot delete a node by a single step. A deletion can be done in 3 different ways; delete from the beginning of the list, delete from the end of the list and delete from the middle of the list.
4. Search: Find the first element with the key k in the given linked list by a simple linear search and returns a pointer to this element