JAVASCRIPT NOTES

BY SHEETAL GUPTA

TABLE OF CONTENTS

VARIABLES AND DATA TYPES

Variables are containers for storing data values in JavaScript. JavaScript has several data types, including strings, numbers, booleans, objects, arrays, and undefined/null.

Example:

```
// Variable declaration

let name = "John";

let age = 30;

let isStudent = true;

let person = { firstName: "John", lastName: "Doe" };

let numbers = [1, 2, 3, 4, 5];

let emptyVariable;
```

Operators

Operators are symbols used to perform operations on operands in JavaScript. JavaScript has various types of operators, including arithmetic, assignment, comparison, logical, and bitwise operators.

code

```
// Arithmetic operators

let sum = 5 + 3;
```

```
let difference = 10 - 5;

let product = 3 * 4;

let quotient = 20 / 5;


// Assignment operators

let x = 5;

x += 3; // Equivalent to x = x + 3;


// Comparison operators

let isEqual = 5 === "5"; // false

let isGreaterThan = 10 > 5; // true


// Logical operators

let isValid = true && false; // false

let isTrue = true || false; // true
```

Control Flow

Control flow statements allow you to control the flow of execution in your JavaScript code. Common control flow statements include if...else, switch, and loops like for, while, and do...while.

code

```
// if...else statement
let age = 18;
if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}

// switch statement
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("It's Monday.");
    break;
  case "Tuesday":
    console.log("It's Tuesday.");
    break;
  default:
    console.log("It's another day.");
```

```
7
}

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

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

Functions

Functions are reusable blocks of code that perform a specific task in JavaScript. They can take parameters and return values.

code

```
// Function declaration
```

```
function greet(name) {
  return "Hello, " + name + "!";
}
// Function expression
let add = function (a, b) {
  return a + b;
};
// Arrow function
let subtract = (a, b) => a - b;
console.log(greet("John")); // Output: Hello, John!
console.log(add(5, 3)); // Output: 8
console.log(subtract(10, 3)); // Output: 7
```

Arrays

Arrays are used to store multiple values in a single variable in JavaScript. They can hold various data types and are indexed starting from 0.

code

```
// Array declaration
```

```
let colors = ["red", "green", "blue"];


// Accessing elements
console.log(colors[0]); // Output: "red"
// Array methods
colors.push("yellow"); // Adds "yellow" to the end
colors.pop(); // Removes the last element
colors.splice(1, 1, "orange"); // Removes one element at index 1 and adds "orange"
```

Objects

Objects are collections of key-value pairs in JavaScript. They are used to store related data and functions.

code

```
// Object declaration
let person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  hobbies: ["reading", "coding"],
```

```
  greet: function () {
    return "Hello, " + this.firstName + "!";
  },
};
// Accessing properties
console.log(person.firstName); // Output: "John"
console.log(person["lastName"]); // Output: "Doe"
// Calling methods
console.log(person.greet()); // Output: "Hello, John!"
```

Classes and Prototypes

Classes are blueprints for creating objects with predefined properties and methods in JavaScript. Prototypes are JavaScript's way of implementing inheritance.

code

```
// Class declaration
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
```

```
11

  }

  greet() {

    return "Hello, " + this.firstName + "!";

  }

}

// Creating instances

let john = new Person("John", "Doe");

console.log(john.greet()); // Output: "Hello, John!"
```

Error Handling

Error handling in JavaScript allows you to gracefully handle runtime errors and exceptions using try...catch blocks.

code

```
try {

  // Code that may throw an error

  throw new Error("Something went wrong!");

}

catch (error)
```

```
12
{
  // Handling the error
  console.error(error.message);
}
 finally
{
  // Optional cleanup code
  console.log("Cleanup code executed.");
}
```

DOM Manipulation

DOM manipulation allows you to interact with HTML elements dynamically using JavaScript. You can select elements, modify content, and handle events.

code

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Manipulation</title>
```

```
</head>
<body>
<div id="container">
  <h1>Hello, World!</h1>
  <button id="btn">Click Me</button>
</div>
<script>
  // Selecting elements
  let container = document.getElementById("container");
  let button = document.getElementById("btn");
  // Modifying content
  container.innerHTML = "<h2>Hello, JavaScript!</h2>";
  // Handling events
  button.addEventListener("click", function() {
    alert("Button clicked!");
  });
</script>
</body>
</html>
```

Events

Events in JavaScript represent actions or occurrences that happen in the browser, such as mouse clicks, key presses, or page loads. Event handling allows you to respond to these events and execute code accordingly.

code

```
<!DOCTYPE html>
<html>
<head>
  <title>Event Handling</title>
</head>
<body>
<button id="btn">Click Me</button>
<script>
  // Selecting the button element
  let button = document.getElementById("btn");
  // Adding an event listener
  button.addEventListener("click", function() {
    alert("Button clicked!");
```

```
  });
</script>
</body>
</html>
```

Asynchronous JavaScript

Asynchronous JavaScript allows you to execute code without blocking other operations. This is commonly used for tasks such as fetching data from a server, handling user input, or performing animations.

code

```
// Asynchronous function using setTimeout
console.log("Start");
setTimeout(function() {
  console.log("Inside timeout");
}, 2000);
console.log("End");
```

AJAX and Fetch API

AJAX (Asynchronous JavaScript and XML) and the Fetch API are used for making asynchronous HTTP requests in JavaScript to fetch data from a server.

code

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

JSON

JSON (JavaScript Object Notation) is a lightweight data interchange format. It is commonly used for sending and receiving data between a client and server.

code

```
// Parsing JSON
let json = '{"name": "John", "age": 30}';
let obj = JSON.parse(json);
console.log(obj.name); // Output: John
```

```
// Stringifying an object
let person = { name: "Jane", age: 25 };
let jsonString = JSON.stringify(person);
console.log(jsonString); // Output: {"name":"Jane","age":25}
```

ES6+ Features

ES6 (ECMAScript 2015) introduced many new features and syntax enhancements to JavaScript, including arrow functions, template literals, destructuring, and spread/rest operators.

code

```
// ES5 function
function add(a, b) {
  return a + b;
}
// ES6 arrow function
const add = (a, b) => a + b;
```

Modules

JavaScript modules allow you to organize code into separate files and import/export functionality between them, improving code maintainability and reusability.

code

```
// math.js (module)

export const add = (a, b) => a + b;

// main.js

import { add } from './math.js';

console.log(add(2, 3)); // Output: 5
```

Scope and Closures

Scope defines the accessibility of variables in JavaScript. Closures are functions that have access to variables from their containing scope even after the parent function has finished executing.

code

```
function outer() {

  let name = "John";

  function inner() {
```

```
    console.log(name);
  }
  return inner;
}
let innerFunc = outer();
innerFunc(); // Output: John
```

Promises and Async/Await

Promises are a way to handle asynchronous operations in JavaScript. Async/await is a syntactic sugar built on top of promises, making asynchronous code more readable and easier to understand.

code

```
// Using async/await with a promise
async function fetchData()
{
  Try
 {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
```

```
    const data = await response.json();
    console.log(data);
  }
catch (error)
{
    console.error('Error:', error);
  }
}
fetchData();
```

Regular Expressions

Regular expressions (regex) are patterns used to match character combinations in strings. They are powerful tools for string manipulation and validation.

code

```
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
const isValidEmail = emailRegex.test("example@example.com");
console.log(isValidEmail); // Output: true
```

Browser Storage

Browser storage mechanisms like localStorage and sessionStorage allow you to store data locally in the browser, persisting even after the browser is closed.

code

```
// Storing data
localStorage.setItem("name", "John");
// Retrieving data
const name = localStorage.getItem("name");
console.log(name); // Output: John
// Removing data
localStorage.removeItem("name");
```

Web APIs

Web APIs are interfaces that browsers expose to developers, providing access to various features like geolocation, web storage, and canvas drawing.

code

```
// Getting user's current location
navigator.geolocation.getCurrentPosition(position => {
  console.log("Latitude:", position.coords.latitude);
  console.log("Longitude:", position.coords.longitude);
});
```

Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.

code

```
const numbers = [1, 2, 3, 4, 5];
// Using map to double each number
const doubledNumbers = numbers.map(num => num * 2);
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
// Using filter to get even numbers
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]
```

Testing

Notes: Testing in JavaScript involves writing automated tests to verify that your code behaves as expected. Popular testing frameworks include Jest, Mocha, and Jasmine.

code

```
// test.js

function add(a, b) {
  return a + b;
}

test('adds 1 + 2 to equal 3', () => {
  expect(add(1, 2)).toBe(3);
});
```

Debugging

Debugging is the process of finding and fixing errors or bugs in your code. Developers often use browser developer tools or integrated development environments (IDEs) for debugging.

code

```
let x = 5;
```

```
console.log('The value of x is:', x);
```

Performance Optimization:

Performance optimization involves improving the speed and efficiency of your JavaScript code. Techniques include minimizing DOM manipulation, reducing network requests, and optimizing algorithms

code

```
// Inefficient
for (let i = 0; i < 1000; i++) {
  document.body.innerHTML += '<div>' + i + '</div>';
}
// More efficient
let fragment = document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
  let div = document.createElement('div');
  div.textContent = i;
  fragment.appendChild(div);
}
```

```
document.body.appendChild(fragment);
```

Security

Security in JavaScript involves protecting your applications from various threats, such as cross-site scripting (XSS), cross-site request forgery (CSRF), and injection attacks. Best practices include input validation, using secure communication protocols (HTTPS), and implementing proper authentication and authorization mechanisms.

code

```
// Escaping user input

function escapeHTML(str) {

  return str.replace(/</g, '&lt;').replace(/>/g, '&gt;');

}

// Usage

let userInput = '<script>alert("XSS attack!")</script>';

let escapedInput = escapeHTML(userInput);

document.getElementById('output').innerHTML = escapedInput;
```

Authentication and Authorization

Authentication verifies the identity of users, while authorization determines what actions they are allowed to perform. Common techniques include using JSON Web Tokens (JWT) for authentication and role-based access control (RBAC) for authorization.

code

```
// Server-side code (Node.js)

const jwt = require('jsonwebtoken');

const generateToken = (user) => {

  return jwt.sign({ userId: user.id }, 'secretKey', { expiresIn: '1h' });

};

const verifyToken = (token) => {

  return jwt.verify(token, 'secretKey');

};

// Client-side code

const token = generateToken({ id: 123 });

console.log(token);

const decodedToken = verifyToken(token);

console.log(decodedToken);
```

Single Page Applications (SPA)

Single Page Applications (SPAs) are web applications that load a single HTML page and dynamically update the content as the user interacts with the application. SPAs use client-side routing and AJAX to provide a seamless user experience.

code

```
// App.js

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Home from './Home';

import About from './About';

import Contact from './Contact';

function App() {

 return (

  <Router>

   <Switch>

    <Route path="/" exact component={Home} />

    <Route path="/about" component={About} />

    <Route path="/contact" component={Contact} />

   </Switch>

  </Router>
```

```
  );
}
export default App;
```

Server-Side JavaScript

Server-Side JavaScript involves running JavaScript code on the server instead of the client. Common server-side JavaScript environments include Node.js and Deno.

code

```
// server.js
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\n');
});
server.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

WebSockets

WebSockets is a communication protocol that provides full-duplex communication channels over a single TCP connection. It allows for real-time data transfer between a client and a server.

code

```
// Client-side code
const socket = new WebSocket('ws://localhost:3000');
socket.addEventListener('open', () => {
  console.log('WebSocket connection established');
});
socket.addEventListener('message', (event) => {
  console.log('Message from server:', event.data);
});
// Server-side code (Node.js)
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 3000 });

wss.on('connection', (ws) => {
  ws.send('Welcome to the WebSocket server!');
  ws.on('message', (message) => {
```

```
    console.log('Received message from client:', message);
  });
});
```

GraphQL

GraphQL is a query language for APIs and a runtime for executing those queries with existing data. It enables clients to request only the data they need, reducing over-fetching and under-fetching of data.

code

```
type Query {
  hello: String
}
schema {
  query: Query
}
```

Data Structures

Data structures are collections of data organized in a particular way to perform operations efficiently. Common data structures include arrays, linked lists, stacks, queues, trees, and graphs.

code

```
class Stack {
 constructor() {
  this.items = [];
 }
 push(element) {
  this.items.push(element);
 }
 pop() {
  if (this.items.length === 0) {
   return "Underflow";
  }
  return this.items.pop();
 }
 peek() {
  return this.items[this.items.length - 1];
 }
 isEmpty() {
```

```
    return this.items.length === 0;
  }
}
// Usage
let stack = new Stack();
stack.push(10);
stack.push(20);
console.log(stack.peek()); // Output: 20
stack.pop();
console.log(stack.peek()); // Output: 10
```

Algorithms

Algorithms are step-by-step procedures or formulas for solving a problem. They are the foundation of computer science and are used to perform tasks efficiently.

code

```
function bubbleSort(arr) {
  let len = arr.length;
  for (let i = 0; i < len; i++) {
    for (let j = 0; j < len - 1; j++) {
```

```
      if (arr[j] > arr[j + 1]) {
        [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
      }
    }
  }
  return arr;
}

// Usage
let numbers = [5, 3, 8, 4, 2];
console.log(bubbleSort(numbers)); // Output: [2, 3, 4, 5, 8]
```

Design Patterns

Design patterns are reusable solutions to common problems encountered in software design. They provide a template for solving issues and improving the structure of your code.

code

```
class Singleton {
```

```
  constructor() {
    if (!Singleton.instance) {
      Singleton.instance = this;
    }
    return Singleton.instance;
  }
}

// Usage
let instance1 = new Singleton();
let instance2 = new Singleton();
console.log(instance1 === instance2); // Output: true
```

Code Organization and Best Practices

Code organization and best practices involve structuring your codebase in a logical and maintainable way, following industry best practices such as clean code principles, modularization, and consistent coding styles.

Example organizing code into modules in Node.js:

code

```
// math.js
export function add(a, b) {
  return a + b;
}

// main.js
import { add } from './math.js';
console.log(add(2, 3)); // Output: 5
```

Deployment and Hosting

Deployment and hosting involve making your web application accessible to users on the internet. This includes deploying your application to servers or cloud platforms and configuring domain names and DNS settings.

Example deploying a Node.js application to Heroku:

```
code
# Install Heroku CLI
$ npm install -g heroku
```

```
# Log in to your Heroku account

$ heroku login


# Create a new Heroku app

$ heroku create


# Deploy your application

$ git push heroku master
```