1. Stack and Heap memory in JavaScript:

In JavaScript, memory is divided into two main areas: the stack and the heap. The stack is a small, organised section of memory used to keep track of function calls and local variables. It works on a Last-In-First-Out (LIFO) principle, similar to a stack of plates. When a function is called, a new frame is added to the stack, and when a function completes, its frame is removed from the stack.

On the other hand, the heap is a larger, unstructured section of memory used for dynamic memory allocation. It is where objects, arrays, and other complex data structures are stored. Unlike the stack, memory allocation and deallocation on the heap can happen in any order and is managed by the JavaScript engine.

Here's an example to illustrate how stack and heap memory work in JavaScript:

```
function foo() {
  var x = 10;  // x is stored in the stack
  var obj = {};  // obj is stored in the heap
}

foo();
```

In this example, when the `foo` function is called, a new frame is added to the stack. Inside the function, the variable `x` is allocated in the stack, while the `obj` variable is allocated in the heap.

- Article: Understanding Memory Management in JavaScript
- Diagrams and explanations: JavaScript Visualizer

2. JS Hashing:

Hashing in JavaScript refers to the process of generating a unique identifier (hash) for a given input data. Hash functions take an input, perform some calculations, and produce a fixed-size output, typically a string of characters.

Hashing is commonly used for data retrieval, encryption, and data integrity checking. In JavaScript, you can use built-in hash functions like `hashCode()` or libraries like `crypto-js` for more advanced hashing algorithms.

Here's an example using the `crypto-js` library to hash a string with the SHA-256 algorithm:

```
const CryptoJS = require("crypto-js");

function hashString(input) {
  return CryptoJS.SHA256(input).toString();
}

const hashedString = hashString("Hello, world!");
console.log(hashedString);
```

In this example, the `hashString` function takes an input string and uses the SHA-256 algorithm from `crypto-js` to compute the hash. The resulting hash is converted to a string and stored in the `hashedString` variable.

- Article with examples: Understanding JavaScript Hash Functions
- Visualization and examples: MD5 Hashing Visualization

3. String Padding:

String padding in JavaScript refers to the process of adding characters to the beginning or end of a string to achieve a desired length. This is often useful when formatting or aligning text.

JavaScript provides the `padStart()` and `padEnd()` methods to perform string padding. The `padStart()` method adds characters to the beginning of a string, while `padEnd()` adds characters to the end.

Here's an example that demonstrates string padding using the `padStart()` method:

```
const originalString = "Hello";
const paddedString = originalString.padStart(10, " ");
console.log(paddedString);  // Output: "     Hello"
```

In this example, the `originalString` is padded with spaces (`" "`) using the `padStart()` method. The first argument specifies the desired length, and the second argument specifies the character(s) to pad with.

- Article with examples: String Padding in JavaScript
- Visual explanation: Visualizing JavaScript String Padding

4. Local Storage and Session Storage:

Local Storage and Session Storage are two web storage mechanisms provided by modern browsers to store data persistently on the client-side.

Local Storage:

- Local Storage allows you to store key-value pairs in the browser's storage, which persists even after the browser is closed and reopened.
- The data stored in Local Storage is available to all pages from the same origin (domain).
- You can access Local Storage using the `localStorage` object in JavaScript.

Session Storage:

- Session Storage is similar to Local Storage but has a different lifespan. It persists data for the duration of the browser session. When the session ends (e.g., the browser is closed), the stored data is cleared.
- Like Local Storage, Session Storage is accessible to all pages from the same origin.
- You can access Session Storage using the `sessionStorage` object in JavaScript.

Here's an example that demonstrates storing and retrieving data using Local Storage:

```
// Storing data in Local Storage
localStorage.setItem("username", "John");

// Retrieving data from Local Storage
const username = localStorage.getItem("username");
console.log(username);  // Output: "John"
```

In this example, we store the value "John" with the key "username" in Local Storage using the `setItem()` method. We then retrieve the value associated with the "username" key using the `getItem()` method.

- Article with examples: Web Storage API: Local Storage and Session Storage
- Visual explanation: Visualizing Local Storage

5. Optional Chaining:

Optional Chaining, introduced in ECMAScript 2020, is a feature in JavaScript that allows you to safely access nested properties or methods of an object without worrying about null or undefined values. It helps to avoid error-prone situations where accessing a property or method on a null or undefined value would result in a runtime error.

Here's an example to illustrate how optional chaining works:

```
const user = {
  name: "John",
  address: {
    street: "123 Main St",
    city: "New York",
  },
};

const city = user.address?.city;
console.log(city);  // Output: "New York"
```

In this example, the `address` object is nested inside the `user` object. By using the optional chaining operator (`?.`), we can safely access the `city` property of `user.address`. If `user.address` is null or undefined, the expression evaluates to undefined instead of throwing an error.

- Article with examples: Optional Chaining in JavaScript
- GIFs and code examples: Optional Chaining Visualized

6. Working with Dates and Times:

Working with dates and times in JavaScript involves manipulating and formatting date objects using built-in methods and libraries like Moment.js.

Here's an example of basic date and time operations in JavaScript:

```
// Creating a new Date object with the current date and time
const currentDate = new Date();

// Getting the current year
```

```
const year = currentDate.getFullYear();
console.log(year);  // Output: 2023

// Formatting the date and time
const formattedDate = currentDate.toLocaleString("en-US", {
  weekday: "long",
  year: "numeric",
  month: "long",
  day: "numeric",
  hour: "numeric",
  minute: "numeric",
  second: "numeric",
});
console.log(formattedDate);  // Output: "Tuesday, July 13, 2023, 12:34:56 PM"
```

In this example, we create a new `Date` object representing the current date and time. We then use various methods like `getFullYear()` to extract specific components of the date. Finally, we format the date and time using `toLocaleString()` and specify the desired options for the formatting.

- Article with examples: Working with Dates and Times in JavaScript
- Diagrams and examples: JavaScript Date Object Explained

7. Pattern Matching and Searching:

Pattern matching and searching in JavaScript involve using regular expressions (regex) to find and manipulate strings based on specific patterns or criteria.

Here's an example that demonstrates pattern matching and searching using regex:

```
const sentence = "The quick brown fox jumps over the lazy dog.";

// Searching for the word "fox" in the sentence
const regex = /fox/;
const found = regex.test(sentence);
console.log(found);  // Output: true
```

In this example, we define a regular expression `/fox/` to match the word "fox". We then use the `test()` method of the regular expression object to check if the word "fox" exists in the `sentence`. The `test()` method returns `true` if a match is found.

- Article with examples: A Beginner's Guide to Regular Expressions in JavaScript
- Visual explanation: Regexper

8. String Manipulation with Regex:

String manipulation with regex involves using regular expressions to perform various operations like replacing parts of a string, extracting specific patterns, or validating string formats.

Here's an example that demonstrates string manipulation using regex:

```
const email = "john@example.com";

// Validating an email address format
const regex = /^\w+@\w+\.\w+$/;
const isValidEmail = regex.test(email);
console.log(isValidEmail);  // Output: true

// Replacing part of a string
const replacedEmail = email.replace(/example/, "test");
console.log(replacedEmail);  // Output: "john@test.com"
```

In this example, we first use a regular expression `/^\w+@\w+\.\w+$/` to validate if the `email` string matches a typical email address format. The `test()` method returns `true` if the format is valid.

Next, we use the `replace()` method with the regular expression `/example/` to replace the word "example" in the `email` string with "test". The resulting string is stored in the `replacedEmail` variable.

- Article with examples: Mastering Regular Expressions in JavaScript
- GIFs and code examples: Regex101

9. Lazy Loading of Assets:

Lazy loading of assets is a technique used to defer the loading of non-critical resources (such as images or scripts) until they are needed. This can improve the initial page load time and overall performance of a website.

Here's an example of lazy loading images using the `loading="lazy"` attribute in HTML:

```
<img src="image.jpg" alt="Lazy-loaded image" loading="lazy">
```

In this example, the `loading="lazy"` attribute is added to the `img` tag. When the browser encounters this attribute, it defers the loading of the image until it enters the viewport or becomes visible on the screen. This way, images below the fold or not immediately visible don't affect the initial page load.

- Article with examples: Lazy Loading Images Complete Guide
- Diagrams and explanations: Lazy Loading Images Explained

10. Browser Caching:

Browser caching is a mechanism that allows web browsers to store (cache) static resources, such as images, CSS files, and JavaScript files, locally on the user's device. This helps reduce server load and improves page load times for subsequent visits or requests.

When a browser requests a resource from a website, it checks if it has a cached version. If it does, and the cached version is still valid (according to the caching headers sent by the server), the browser uses the cached version instead of making a new request to the server.

Configuring browser caching involves setting appropriate caching headers on the server-side. For example, the `Cache-Control` header can be used to specify how long the resource should be cached by the browser.

By leveraging browser caching, subsequent visits or requests for the same resource can be served directly from the browser's cache, resulting in faster page loads and reduced bandwidth usage.

- Article with examples: A Beginner's Guide to HTTP Cache Headers
- Diagrams and explanations: Caching Explained

Prepared By :
Divyansh Singh (LinkedIn : rgndunes)