JavaScript

Session 3

Introduction to JavaScript Local Storage

- JavaScript Local Storage allows web applications to store data persistently in a web browser with no expiration time.
- Key Points:
  - Enables data storage across browser sessions.
  - Stores data as key-value pairs.
  - Has a maximum storage limit of about 5MB per domain.
  - Data is stored in a string format but can be converted using JSON methods.

Example

```
// Storing data
localStorage.setItem('username', 'CodeLover');

// Retrieving data
let user = localStorage.getItem('username');

// Removing data
localStorage.removeItem('username');

// Clearing all data
localStorage.clear();
```

Use-Cases of JavaScript Local Storage

- **Persist User Data:** Maintain user data, settings, and states even after a browser is closed or refreshed.
- **Save Preferences:** Store user interface preferences like themes or languages.
- **Reduce Server Load:** Cache data client-side to minimize constant server requests.
- **Form Data Retrieval:** Prevent data loss by saving form inputs in case of accidental page reloads.

```
// Save form data to prevent loss on page reload
document.getElementById('userForm').addEventListener('input', function(e) {
    localStorage.setItem(e.target.id, e.target.value);
});
```

Storing Complex Data with JSON.stringify() in Local Storage

- **Understanding JSON.stringify():**
  - Converts JavaScript objects into a string format.
  - Essential for storing non-string data (like objects and arrays) in local storage since it only accepts string key-value pairs.
- **Key Points:**
  - Allows the storage of more complex data types (e.g., arrays, objects) by converting them into a string format.
  - Upon retrieval, JSON.parse() is used to convert the string back into its original format.

```
// Storing an object
const user = {
    name: 'CodeLover',
    preferences: {
        theme: 'dark',
        language: 'English'
    }
};

localStorage.setItem('user', JSON.stringify(user));

// Retrieving and parsing the object
let retrievedUser = localStorage.getItem('user');
retrievedUser = JSON.parse(retrievedUser);

// Accessing properties
alert(`Welcome back, ${retrievedUser.name}!`);
```

Exercise: Build chrome extension!

- Build a bookmark Extension!

Chrome Extensions and User Customization

- **Definition:** Chrome extensions are small software programs that customize the browsing experience.
- **User Customization:**
  - Users can alter the functionality and behavior of extensions according to their needs.
  - Customizations can range from changing themes and layouts to modifying functionality using user scripts.
  - Some extensions offer built-in options for customization while others may be modified using third-party tools or additional coding.

```
// Sample manifest.json of a Chrome extension allowing custom themes
{
   "name": "My Custom Extension",
   "version": "1.0",
   "description": "An extension with customizable theme options",
   "permissions": ["storage"],
   "options_page": "options.html",
   "manifest_version": 2
}
```

Dynamic Image Rendering with JavaScript

- **Issue with Static HTML:**
  - Hardcoding image tags (<img>) in HTML requires manual entry and updates, becoming unscalable and cumbersome with numerous images.
- **Leveraging JavaScript:**
  - Dynamically render images from a folder to promote ease of update and scalability.
  - Accommodate for dynamically added or removed images without modifying the HTML.
- **Key Benefits:**
  - Scalability: Effortlessly manage large collections of images.
  - Maintainability: Simplify updates, additions, or removals.
  - Efficiency: Save time and mitigate errors related to manual entries.

example

```
// Assume `imageNames` is an array of image file names in your folder
let imageNames = ["image1.jpg", "image2.png", "image3.jpeg"];

// Targeting the container where images will be rendered
let imageContainer = document.getElementById('imageContainer');

// Looping through image names and appending them to the HTML
imageNames.forEach(name => {
    let imgElement = document.createElement('img');
    imgElement.src = `path/to/images/${name}`;
    imgElement.alt = `Image - ${name}`;
    imageContainer.appendChild(imgElement);
});
```

Enhancing User Experience & Performance

- **Lazy Loading:**
  - Load images only when they enter the viewport to optimize initial page load times.
  - Use the loading="lazy" attribute in your <img> tag or implement a JavaScript intersection observer.
- **Image Optimization:**
  - Ensure images are appropriately sized and compressed to avoid unnecessarily large files.
  - Consider using modern formats like WebP for quality at significantly smaller file sizes.
- **Interactive UI:**
  - Implement features like lightboxes, sliders, or galleries for an enhanced user experience.

Example cont.,

```
// ... (Previous code)

// Looping through image names and appending them to the HTML
imageNames.forEach(name => {
    let imgElement = document.createElement('img');
    imgElement.src = `path/to/images/${name}`;
    imgElement.alt = `Image - ${name}`;
    imgElement.loading = "lazy";  // Enabling native lazy loading
    imageContainer.appendChild(imgElement);
});
```

Extra topics

- **Exception handling**: try, catch!
- **Drag and drop** events in JS
- **Modules** in JS: export, import
- **Forms** event listener – (preventdeafult() option)
- **classList** property: get, add, delete
- **Attribute** methods: get, this, custom attribute
- **Classes** in JS
- **REST API**