Advance JavaScript

1. Advance Concepts

1.1 Session storage & Local storage: -

Session Storage and Local Storage are two web storage mechanisms available in JavaScript to store data on the client-side, i.e., within the user's web browser. They are used for persisting data across page reloads and even when the browser is closed and reopened. However, they have some differences in terms of scope and lifetime.

- getItem(): Gets the value for a given key.
- setItem(): Sets the value for a given key.
- removeItem(): Removes the value for a given key.
- clear(): Clears all data from storage.

- **Session Storage:**
  - Data stored in session storage is available only for the duration of a page session. A session typically ends when the user closes the tab or browser.
  - It provides a simple key-value store and can store data as strings.
  - Data stored in session storage is limited to the tab or window from which it was set. It cannot be accessed by other tabs or windows from the same website.
  - To use session storage, you can use the 'sessionStorage' object in JavaScript:
    - **Storing:** sessionStorage.setItem('key', 'value');
    - **Retrieving:** const data = sessionStorage.getItem('key');
    - **Remove:** sessionStorage.removeItem('key');

- **Local Storage:**
  - Data stored in local storage has a longer lifetime and is available even after the browser is closed and reopened. It persists until explicitly removed by the user or the website.
  - Like session storage, it provides a simple key-value store and can store data as strings.
  - Data stored in local storage is shared across all tabs and windows from the same website, as it has a broader scope.
  - To use local storage, you can use the 'localStorage' object in JavaScript:
    - **Storing:** localStorage.setItem('key', 'value');
    - **Retrieving:** const data = localStorage.getItem('key');
    - **Remove:** localStorage.removeItem('key');

1.2 Basics of Cookies: -

Cookies are small text files that are stored on the user's computer by a web browser. Cookies can be used to store a variety of data, such as user preferences, login information, and shopping cart items.

To use cookies in JavaScript, developers can use the 'document.cookie' property. This property is a string that contains all the cookies that are currently set for the current domain.

- **Methods**

The document.cookie property has the following methods:

- getItem(): Gets the value for a given key.
- setItem(): Sets the value for a given key.
- removeItem(): Removes the value for a given key.
- clear(): Clears all cookies for the current domain.

- **Snippet**

```
// Set a cookie.
document.cookie = 'name=John Doe';
// Get the value of a cookie.
const name = document.cookie.split(';')[0];
// Remove a cookie.
document.cookie = 'name=; expires=Thu, 01 Jan 1970 00:00:00 GMT';
// Clear all cookies.
document.cookie = '';
```

- **Limitations**
  - Cookies have a maximum size limit (usually a few kilobytes), so they are not suitable for storing large amounts of data.
  - Cookies are sent with every HTTP request made to the same domain, which can impact performance if too many cookies are set.
  - Users can configure their browsers to block or delete cookies, so you should not rely on cookies for critical data storage or security.

1.3 Browser Debugging:

1.3.1 Inspect Element Window: -

The Inspect Element window in JavaScript is a powerful tool that allows developers to inspect and modify the HTML, CSS, and JavaScript of a web page. It is a separate window that is displayed alongside the web page itself.

The Inspect Element window can be opened by right-clicking on any element on a web page and selecting "Inspect". Alternatively, you can press the F12 key on your keyboard.

Developers use the Inspect Element window to:

- **Debug JavaScript code:** The Console tab can be used to step through JavaScript code line by line and view the values of variables. This can be helpful for tracking down errors in your code.
- **Troubleshoot CSS problems:** The Styles tab can be used to inspect the CSS styles that are applied to an element and to edit the CSS styles directly. This can be helpful for troubleshooting CSS problems, such as elements that are not displaying correctly.
- **Learn how web pages are built:** The Elements tab shows the HTML structure of the web page. This can be helpful for understanding how web pages are organized and how different elements are related to each other.

Here are some specific examples of how the Inspect Element window can be used in JavaScript:

- To inspect the HTML structure of a web page, you can use the Elements panel. This can be helpful for understanding how a web page is organized and how different elements are related to each other.
- To modify the CSS styles of a web page, you can use the Styles panel. This can be helpful for making quick changes to the appearance of a web page without having to edit the CSS code directly.
- To debug JavaScript code, you can use the Console panel. This can be helpful for tracking down errors in your code and understanding how it is executing.
- To troubleshoot CSS problems, you can use the Network and Timeline panels. These panels can help you to identify which CSS files are being loaded and when they are being executed.

1.3.2 Detail knowledge of diffrent tabs in inspect element window: -

- **Elements Tab: -**

  The Elements tab shows the HTML structure of the web page. You can click on any element in the Elements tab to select it on the web page. The Elements tab also includes a number of features that can be used to inspect and modify the HTML code, such as the ability to edit the HTML code directly, add and remove elements, and search for specific elements.

- **Styles Tab: -**

  The Styles tab shows the CSS styles that are applied to the selected element. You can click on any style in the Styles tab to edit it. The Styles tab also includes a number of features that can be used to inspect and modify the CSS code, such as the ability to see the inheritance chain for a style and to edit CSS rules directly.

- **Console Tab: -**

  The Console tab shows the JavaScript console, which can be used to execute JavaScript code and view the output. The Console tab is a powerful tool for debugging JavaScript code. You can use the Console tab to step through your code line by line, view the values of variables, and catch errors.

- **Network Tab: -**

  The Network tab shows the network activity of the web page. You can use the Network tab to see which resources are being loaded by the web page, how long they are taking to load, and the size of each resource. The Network tab can be helpful for diagnosing performance problems and troubleshooting network connectivity issues.

- **Timeline Tab: -**

  The Timeline tab shows the execution timeline of the JavaScript code on the web page. You can use the Timeline tab to see how long each JavaScript function is taking to execute and to identify bottlenecks in your code. The Timeline tab can be helpful for optimizing your JavaScript code for performance.

In addition to these four main tabs, the Inspect Element window also includes a number of other tabs, such as the Sources tab, which shows the source code of the web page, and the Accessibility tab, which shows information about the accessibility of the web page.

1.3.3 Caching: -

Caching in JavaScript is the process of storing frequently accessed data in a fast-access storage medium, such as memory, so that subsequent requests for the same data can be served quickly without having to go through the full process of retrieving it from the source.

There are a number of different ways to cache data in JavaScript. One common way is to use the localStorage object. localStorage is a persistent storage mechanism that can be used to store data between browser sessions.

- **Methods for Caching: -**
  - **Using the localStorage object:** localStorage is a persistent storage mechanism that can be used to store data between browser sessions. This means that data cached in localStorage will still be available even after the user closes their browser and reopens it.
  - **Using the Cache API:** The Cache API provides a more flexible way to cache data than localStorage. For example, you can use the Cache API to cache data for a specific amount of time or to cache data conditionally.

- **Benefits of caching**
  - **Improved performance:** Caching can reduce the number of times that your application needs to make network requests. This can result in faster loading times and a better user experience.
  - **Reduced load on servers**: Caching can reduce the load on your server by serving cached data instead of generating it on the fly. This can improve the performance of your server and reduce costs.
  - **Increased reliability:** Caching can help to improve the reliability of your application by reducing the number of network requests that your application needs to make. This can help to prevent your application from crashing if there is a problem with your network connection.

2. OOJS study

2.1 What is OOJS?

OOjs (Object-oriented JavaScript) is a JavaScript library that provides object-oriented programming features to JavaScript. It is based on the principles of inheritance, encapsulation, polymorphism, and abstraction.

- **Classes:** OOjs provides a class system that allows you to create reusable and extensible code.
- **Inheritance:** OOjs supports inheritance, which allows you to create new classes that inherit the properties and methods of existing classes.
- **Encapsulation:** OOjs supports encapsulation, which allows you to hide the implementation details of your classes from other parts of your code.
- **Polymorphism:** OOjs supports polymorphism, which allows you to treat objects of different classes in a similar way.
- **Abstraction:** OOjs supports abstraction, which allows you to hide the implementation details of your classes and expose only the functionality that is needed by other parts of your code.

➢ Code snippet for above topics: -

- **Object:**

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  sayHello: function () {
    console.log(`Hello, ${this.firstName} ${this.lastName}!`);
  },
};
```

- **Class:**

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;  }
```

```
  sayHello() {
    console.log(`Hello, ${this.firstName} ${this.lastName}!`);
  }
}
const person = new Person('John', 'Doe');
```

- **Inheritance:**

```
class Student extends Person {
  constructor(firstName, lastName, studentId) {
    super(firstName, lastName);
    this.studentId = studentId;
  }

  study() {
    console.log(`${this.firstName} is studying.`);
  }
}
```

- **Encapsulation:**

```
class Person {
  constructor(name) {
    this.name = name;
  }
  private greet() {
    return `Hello, my name is ${this.name}!`;
  }
  get greet() {
    return this._greet();}
}
```

2.2 Possible ways to implement class: -

There is two main way to implement class in JavaScript:

- **Class declaration:** This is the standard way to implement classes in JavaScript. It uses the class keyword to define a new class.

```
class Person {
 constructor(name) {
  this.name = name;
 }

 greet() {
  return `Hello, my name is ${this.name}!`;
 }
}

const person = new Person('John Doe');
console.log(person.greet()); // "Hello, my name is John Doe!"
```

- **Class expression:** This is a more concise way to implement classes in JavaScript. It uses the function keyword to define a new class.

```
const Person = function(name) {
 this.name = name;
};
Person.prototype.greet = function() {
 return `Hello, my name is ${this.name}!`;
};

const person = Person('John Doe');
console.log(person.greet()); // "Hello, my name is John Doe!"
```

2.3 Static class, Properties declaration: -

- **Static class**

A static class in JavaScript is a class that does not have any instances. It can only be used to define static members, which are members that are shared by all instances of the class. Static classes are often used to define constants, utility functions, and helper classes.

```
class MathUtils {
 static PI = 3.141592653589793;

 static calculateCircleArea(radius) {
  return Math.PI * radius ** 2;
 }
}
const circleArea = MathUtils.calculateCircleArea(10);
console.log(circleArea); // 314.1592653589793
```

- **Properties declaration**

Properties in JavaScript can be declared as either instance properties or static properties.

Instance properties are properties that are unique to each instance of a class. Static properties are properties that are shared by all instances of a class.

To declare a property as an instance property, we use the 'this' keyword. To declare a property as a static property, we use the 'static' keyword.

```
class Person {
 constructor(name) {
  this.name = name;
 }
 static population = 0;
```

```
  greet() {
    return `Hello, my name is ${this.name}!`;
  }
}
const johnDoe = new Person('John Doe');

const name = johnDoe.name;

console.log(name); // "John Doe"
const population = Person.population;

console.log(population); // 0
```

3. ECMAScript6

3.1 Difference between let, var and const: -

- **let: -**

  Block scope: Variables declared with let have block scope, which means that they are only accessible within the block in which they are declared.

  Mutable: Variables declared with let can be reassigned.

```
function example() {
  let x = 10;
  if (true) {
    let x = 20; // This creates a new x variable within the block
  }
  console.log(x); // Output: 10
}
```

- **var: -**

  Function scope: Variables declared with var have function scope, which means that they are accessible within the function in which they are declared and within any nested functions.

  Mutable: Variables declared with var can be reassigned.

```
function example() {
  var x = 10;
  if (true) {
    var x = 20; // This modifies the outer variable x
  }
  console.log(x); // Output: 20
}
```

- **const: -**

  Block scope: Variables declared with const have block scope, which means that they are only accessible within the block in which they are declared.

  Immutable: Variables declared with const cannot be reassigned.

```
const pi = 3.14159;

pi = 3.14; // This will result in an error because you can't reassign a const variable.


const colors = ['red', 'green', 'blue'];

colors.push('yellow'); // This is allowed because it's modifying the array, not reassigning the variable.
```

3.2 JavaScript Classes: -

JavaScript classes are a way to create reusable and extensible code. They allow you to define new types of objects and define methods that can be used on those objects.

Classes can be used to model real-world entities, such as people, cars, and computers. They can also be used to create more abstract entities, such as mathematical objects or business rules.

ES6 classes are a powerful new feature in JavaScript that can be used to create more concise, readable, and extensible code.

Here are some additional features of ES6 classes:

- **Static methods:** Static methods are methods that are not attached to any specific instance of a class. They can be used to define methods that are shared by all instances of a class.
- **Class properties:** Class properties are properties that are shared by all instances of a class. They can be used to store data that is common to all instances of a class.
- **Constructor overloading**: Constructor overloading allows you to create constructors with different signatures. This can be useful for creating constructors that can be used to create different types of objects.
- **Decorators:** Decorators are a way to modify the behavior of classes and their methods. They can be used to add new functionality, or to change the way that existing functionality works.

3.3 Arrow function: -

Arrow functions are a new type of function that was introduced in ES6. They are a more concise and simplified way to write functions.

- Benefits of Arrow function: -
  - **Conciseness:** Arrow functions are more concise than traditional function declarations. This can make your code more readable and easier to maintain.
  - **Readability:** Arrow functions are easier to read than traditional function declarations. This is because they are more concise and they do not have their own this keyword.
  - **Maintainability:** Arrow functions are easier to maintain than traditional function declarations. This is because they are more concise and they do not have their own this keyword.
- Syntax: -

```
const arrowFunction = (parameters) => {
  // body of the function
};
```

- Example: -

```
const greetPerson = (name) => {
  console.log(`Hello, ${name}!`);
};
greetPerson('John Doe'); // "Hello, John Doe!"
```

```
const button = document.querySelector('button');
button.addEventListener('click', () => {
  console.log('Button clicked!');
});
```

3.4 Import, Export, async, await Functions: -

- **Import and Export: -**

Import and export are two new features in JavaScript that were introduced in ES6. They allow you to modularize your code and make it easier to reuse code. Import is used to import code from another file. Export is used to export code from a file so that it can be imported by other files.

To import code from another file, you use the import keyword. For example, the following code imports the greet() function from the greetings.js file:

```
import greet from './greetings.js';

greet('John Doe'); // "Hello, John Doe!"
```

To export code from a file, you use the export keyword. For example, the following code exports the greet() function from the greetings.js file:

```
export function greet(name) {
  console.log(`Hello, ${name}!`);
}
```

- **Async and Await Functions: -**

Async and await are two new features in JavaScript that were introduced in ES8. They allow you to write asynchronous code in a more concise and readable way. Async functions are functions that can return a promise. Await is a keyword that can be used to wait for a promise to resolve before continuing execution.

For example, the following code uses an async function to fetch data from a server and then print the data to the console:

```
async function fetchData() {
  const response = await fetch('https://example.com/api/data');
  const data = await response.json();
  console.log(data);
}
fetchData();
```

3.5 Difference between == & === , != & !=== :-

In JavaScript, '==', '===', '!=', and '!==' are comparison operators used to compare values for equality or inequality. These operators behave differently based on whether they perform strict or type-coercive comparisons.

- **'==' (Equality Operator - Type Coercion):**
  - The '==' operator performs a loose or type-coercive equality comparison.
  - It compares values for equality after performing type conversion (coercion) if necessary. In other words, it attempts to make both operands of the same type before comparison.
  - If the types are different, JavaScript will implicitly convert one or both values to a common type before making the comparison.
  - Use '==' when you want to compare values while allowing for type conversion.

```
5 == '5' // true (values are equal after type coercion)
```

- **'===' (Strict Equality Operator - No Type Coercion):**
  - The '===' operator performs a strict equality comparison.
  - It compares values for equality without performing type conversion. Both the value and the type must be the same for the comparison to return true.
  - Use '===' when you want to ensure both the value and the type are the same for equality.

```
5 === '5' // false (values are not equal because of different types)
```

- **'!=' (Inequality Operator - Type Coercion):**
  - The '!=' operator performs a loose or type-coercive inequality comparison.
  - It compares values for inequality after performing type conversion if necessary.
  - Like '==', '!=' attempts to make both operands of the same type before comparison.
  - Use '!=' when you want to compare values while allowing for type conversion.

```
5 != '5' // false (values are considered equal after type coercion)
```

- **‘!==’ (Strict Inequality Operator - No Type Coercion):**
  - The ‘!==’ operator performs a strict inequality comparison.
  - It compares values for inequality without performing type conversion. Both the value and the type must be different for the comparison to return true.
  - Use ‘!==’ when you want to ensure either the value or the type (or both) are different for inequality.

```
5 !== '5' // true (values are not equal due to different types)
```