# Advanced JavaScript Guide

## 5. Advanced Concepts

### 1. Advanced Concepts

#### 1.1 Session Storage & LocalStorage

* **Session Storage**: Stores data for the duration of a page session (data is deleted when the page or browser is closed).
* sessionStorage.setItem("username", "JohnDoe");  
  console.log(sessionStorage.getItem("username")); *// Outputs: JohnDoe*  
  sessionStorage.removeItem("username");
* **LocalStorage**: Stores data with no expiration date (persists even when the browser is closed and reopened).
* localStorage.setItem("theme", "dark");  
  console.log(localStorage.getItem("theme")); *// Outputs: dark*  
  localStorage.clear(); *// Clears all localStorage data*

#### 1.2 Basics of Cookies

* **Cookies**: Small pieces of data stored on the user’s computer, often used for tracking or remembering information.
* **Setting a Cookie**:
* document.cookie =  
   "username=JohnDoe; expires=Fri, 31 Dec 2024 12:00:00 UTC; path=/";
* **Reading Cookies**:
* console.log(document.cookie); *// Outputs all cookies as a string*
* **Deleting a Cookie**: Set its expiration date to a past date.

#### 1.3 Browser Debugging

##### 1.3.1 Inspect Element Window

* **Inspect Element**: A developer tool to examine and modify the DOM and CSS of a webpage in real-time.
* **Shortcut**: Right-click on the page → Click on “Inspect”.

##### 1.3.2 Detailed Knowledge of Different Tabs in Inspect Element

* **Elements**: View and modify HTML and CSS.
* **Console**: Execute JavaScript code snippets and see error logs.
* **Sources**: View the source files and debug JavaScript.
* **Network**: Monitor network requests (e.g., API calls).
* **Performance**: Analyze page load and runtime performance.
* **Application**: Manage storage (LocalStorage, Session Storage, Cookies, etc.).
* **Memory**: Diagnose memory leaks and optimize memory usage.
* **Lighthouse**: Generate performance, accessibility, and SEO reports.

##### 1.3.3 Caching

* **Browser Caching**: Temporarily storing resources (like images and scripts) to improve performance.
* **Clearing Cache**: Can be done manually from the browser settings or programmatically using cache control headers.

## 2. Object-Oriented JavaScript (OOJS) Study

### 2.1 What is OOJS?

* **Object-Oriented JavaScript**: Programming paradigm based on the concept of objects, which can contain data (properties) and functions (methods).
* **Example**:
* **let** car = {  
   brand: "Tesla",  
   model: "Model S",  
   drive: **function** () {  
   console.log("Driving...");  
   },  
  };  
  car.drive(); *// Outputs: Driving...*

### 2.2 Possible Ways to Implement Classes

* **Using Constructor Functions**:
* **function** Animal(name) {  
   **this**.name = name;  
  }  
  Animal.prototype.speak = **function** () {  
   console.log(`${**this**.name} makes a sound.`);  
  };  
  **let** dog = **new** Animal("Dog");  
  dog.speak(); *// Outputs: Dog makes a sound.*
* **Using ES6 Classes**:
* **class** Animal {  
   constructor(name) {  
   **this**.name = name;  
   }  
   speak() {  
   console.log(`${**this**.name} makes a sound.`);  
   }  
  }  
  **let** cat = **new** Animal("Cat");  
  cat.speak(); *// Outputs: Cat makes a sound.*

### 2.3 Static Class & Properties Declaration

* **Static Methods and Properties**: Belong to the class itself rather than an instance.
* **class** MathHelper {  
   **static** pi = 3.14159;  
   **static** calculateCircumference(radius) {  
   **return** 2 \* MathHelper.pi \* radius;  
   }  
  }  
  console.log(MathHelper.calculateCircumference(5)); *// Outputs: 31.4159*

## 3. ECMAScript6 (ES6) Documentation

### 3.1 Difference Between let, var, and const

* **var**: Function-scoped, can be redeclared.
* **let**: Block-scoped, cannot be redeclared in the same scope.
* **const**: Block-scoped, used for constants that cannot be reassigned.
* **Example**:
* **var** a = 5; *// Function-scoped*  
  **let** b = 10; *// Block-scoped*  
  **const** c = 15; *// Block-scoped, constant*

### 3.2 JavaScript Classes

* **Classes**: Templates for creating objects, introduced in ES6.
* **class** Rectangle {  
   constructor(height, width) {  
   **this**.height = height;  
   **this**.width = width;  
   }  
   area() {  
   **return** **this**.height \* **this**.width;  
   }  
  }  
  **let** rect = **new** Rectangle(10, 20);  
  console.log(rect.area()); *// Outputs: 200*

### 3.3 Arrow Functions

* **Arrow Function Syntax**: A concise way to write functions using the => syntax.
* **const** add = (a, b) **=>** a + b;  
  console.log(add(5, 10)); *// Outputs: 15*

### 3.4 Import, Export, Async, Await Functions

* **Import/Export**: Used to modularize code.
* **Export**:
* **export** **const** pi = 3.14;
* **Import**:
* **import** { pi } **from** "./math.js";
* **Async/Await**: Simplifies asynchronous code, making it more readable.
* **async** **function** fetchData() {  
   **try** {  
   **let** response = **await** fetch("https://api.example.com/data");  
   **let** data = **await** response.json();  
   console.log(data);  
   } **catch** (error) {  
   console.error("Error fetching data", error);  
   }  
  }  
  fetchData();

## 4. Extra Points

### 4.1 Difference Between == & ===, != & !==

* **==**: Loose equality, compares values after type conversion.
* console.log(5 == "5"); *// Outputs: true*
* **===**: Strict equality, compares both value and type.
* console.log(5 === "5"); *// Outputs: false*
* **!=**: Loose inequality.
* console.log(5 != "5"); *// Outputs: false*
* **!==**: Strict inequality.
* console.log(5 !== "5"); *// Outputs: true*