JavaScript CheatSheet

Presented By

Code Easy Academy

Fundamentals

1: Syntax Essentials

Variables: Your Data's Nametags

Declaring Variables:

```
let myVariable; // Flexible, block-scoped
const myConstant;// Value can't change
var oldVariable; // Older, avoid in modern code
```

Assigning Values

```
let age = 30;
```

```
const userName = "Sarah";
```

```
var isLoggedIn = true;
```

Data Types: What Kind of Information Are We Storing?

Numbers

```
let quantity = 5;
```

```
const pi = 3.14159;
```

Strings (Text)

```
let greeting = "Hello";
```

```
const message = 'Welcome to JavaScript!';
```

Booleans (True/False)

```
let isComplete = true;
```

```
const hasError = false;
```

null and undefined

null: Intentional absence of value
undefined: Variable declared, but not assigned

Operators: Our Calculation Toolkit

Arithmetic

```
let total = price + tax;
let difference = x - y;
let result = a * b;
let quotient = a / b;
let remainder = a % b; // Modulo
```

Comparison

```
let isEqual = x == y;
let isNotEqual = x != y;
let isGreater = x > y;
let isLess =  x < y;
// Plus: >=, <=,  === (strict equality)
```

Logical

```
let isValid = (age >= 18) && (hasTicket == true);
// AND
let hasAccess = isAdmin || isModerator;
// OR
let isVisible = !isHidden;
// NOT
```

Comments: Notes for Yourself and Other Developers

Single-line: // This is a comment

Multi-line:
```
/* This is a multi-line comment.
It can span several lines. */
```

Gotchas, Tips

Gotchas

Loose Typing: Be careful with == (use === more often)
Naming Conflicts: Avoid reserved words (like let, if, function)

Tips:

Descriptive Variable Names: totalCost is better than x
Semicolons: Optional, but good practice
Code Formatting: Use indentation for readability

Glossary

Variable: A named container for storing data.
Data Type: The classification of data (number, string, etc.)
Operator: A symbol that performs an action.
Comment: Text ignored by the JavaScript engine.
Loose Typing: JavaScript's automatic conversion of data types.
Strict Equality (===): Checks value and type for a match.

2: Control Flow

Making Decisions with if...else

```
if (condition) {
    // Code to run if condition is true
} else {
    // Code to run if condition is false
}
```

Example:

```
if (age >= 18) {
    console.log("You are eligible to vote.");
} else {
    console.log("You are not yet eligible to
vote.");
}
```

More Options with else if

```
if (condition1) {
    // Code to run if condition1 is true
} else if (condition2) {
    // Code to run if condition2 is true
} else {
    // Default code to run
}
```

Example:

```
if (score >= 90) {
   grade = "A";
} else if (score >= 80) {
   grade = "B";
} else {
   grade = "C";
}
```

The switch Statement: Picking from Many Choices

```
switch (expression) {
    case value1:
        // Code to run if expression matches value1
        break;
    case value2:
        // Code to run if expression matches value2
        break;
    // ... more cases
    default:
        // Code to run if no match is found
}
```

Example

```
let day = "Monday";
switch (day) {
    case "Monday":
        console.log("First day of the week");
        break;
    case "Tuesday":
        console.log("Second day of the week");
        break;
    // ... more cases
    default:
        console.log("It's a weekend!");
}
```

Repeating with Loops

for Loop

```
for (initialization; condition; increment) {
    // Code to run in each loop iteration
}
```

while Loop

```
while (condition) {
    // Code to run as long as condition is true
}
```

Tips & Gotchas

Tips

- Use curly braces ({}) even for single-line blocks within control structures – improves readability
- Structure complex conditions with parentheses for clarity

Gotchas

- Be careful of infinite loops if your loop condition never becomes false

Glossary

Conditional Statement: Controls code execution based on a condition.

Loop: Repeats code a specified number of times or until a condition is met.

Iteration: One cycle of a loop's execution.

Infinite Loop: A loop that never ends.

Let me know if you'd like another topic expanded – I'm ready to tackle functions, arrays, or anything else!

3: Functions

Reusable Blocks of Code

Functions package code to do a specific task.
They can be called multiple times, making code organized and efficient.

Function Declarations

```
function functionName(parameter1, parameter2, ...) {
    // Code to be executed
}
```

Example:

```
function calculateArea(width, height) {
    let area = width * height;
    return area;
}
```

Calling a Function

```
let result = calculateArea(10, 5); // result will be 50
console.log(result);
```

Function Expressions

Assigning a function to a variable.

```
const greet = function(name) {
    console.log("Hello " + name + "!");
};

greet("Alice");
```

Arrow Functions (ES6)

Shorter syntax for function expressions.

```javascript
const multiply = (num1, num2) => num1 * num2;

let product = multiply(5, 6); // product will be 30
```

Parameters vs. Arguments

Parameters: Placeholders in the function definition.
Arguments: Actual values passed when calling a function.

Advanced Concepts & Gotchas

Recursion: A function that calls itself.
Example (Calculating Factorial):

```javascript
function factorial(n) {
    if (n === 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Gotchas

Infinite Recursion: Ensure a way to end the recursion.
Scope: Be mindful of variable access inside functions.

Tips

Default Parameters: Provide default values if arguments are missing.

Glossary

Function: Self-contained block of code performing a task.
Parameter: Variable listed in the function's definition.
Argument: Value passed when calling the function.
Return Value: What a function sends back after execution.
Recursion: A function calling itself within its definition.

4: Arrays

Ordered Collections of Data

Store multiple values under a single variable name.
Items are indexed starting from zero (0, 1, 2...).

Creating Arrays

Array Literal Syntax:

```
let fruits = ["apple", "banana", "orange"];
let numbers = [1, 5, 10];
let mixedArray = [true, "hello", 42];
let emptyArray = [];
```

Accessing Array Elements

Using Square Brackets with Index:

```
let firstFruit = fruits[0];  // firstFruit will be
"apple"
let secondNumber = numbers[1]; // secondNumber
will be 5
```

Working with Arrays

Common Array Methods

push() Adds an element to the end.
pop() Removes and returns the last element.
unshift() Adds an element to the beginning.
shift() Removes and returns the first element.
length Property gives the number of elements in the array.

Iteration: Looping through Arrays

for Loop:

JavaScript

```
for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```

for...of Loop (ES6):

```
for (const fruit of fruits) {
   console.log(fruit);
}
```

More Methods, Tips, Gotchas

More Useful Methods

join() Creates a string from array elements.
slice() Extracts a portion of the array.
indexOf() Searches for an element's index.

Tips

Use descriptive array names (e.g., shoppingCart, userAges)

Gotchas

Arrays are mutable (their contents can be changed even with const).
Out-of-bounds index errors (accessing elements that don't exist).

Glossary

Array: An ordered list of values.
Element: Each item in an array.
Index: The numerical position of an element (starting from 0).
Method: A function associated with an object (like an array).
Mutable: The contents of the object can be changed after creation.

5: Objects

Key-Value Collections for Organizing Data

Unlike Arrays: Store data with named properties instead of just numerical indexes.

Real-world Analogy: Think of an object like a contact card with fields like "name", "email", "phone number".

Creating Objects

Object Literal Syntax:

```
let person = {
    name: "Sarah",
    age: 30,
    city: "New York"
};
```

With the new Keyword:

```
let car = new Object();
car.make = "Toyota";
car.model = "Camry";
car.year = 2023;
```

Accessing Object Properties

Dot Notation:

```
console.log(person.name);   // Output: Sarah
console.log(person.age);    // Output: 30
```

Bracket Notation:

```
console.log(person["city"]); // Output: New York

// Using a variable:
let propertyName = "age";
console.log(person[propertyName]); // Output: 30
```

Adding and Modifying Properties

Adding New Properties:

```
person.country = "USA";
car.color = "red";
```

Modifying Existing Properties:

```
person.age = 31; // Updates the existing age
```

Methods: Functions Inside Objects

```
let student = {
    firstName: "John",
    lastName: "Doe",
    getFullName: function() {
        return this.firstName + " " +
this.lastName;
    }
};

console.log(student.getFullName()); // Output: John
Doe
```

The this Keyword: Refers to the object that the method is being called on.

Advanced Concepts & Gotchas

Iteration with for...in

JavaScript

```
for (const prop in person) {
    console.log(prop + ": " + person[prop]);
}
```

Gotchas

Object References: Objects are passed by reference, which means changes made to a copy can affect the original

Glossary

Object: A collection of key-value pairs.
Property: A characteristic of an object.
Value: The data held by a property.
Method: A function defined within an object
this: A keyword referencing the current object (inside methods).

6: The DOM (Document Object Model)

Your Webpage as a Tree

The DOM represents the HTML structure of a webpage as a tree of nodes.
Each tag is an element node, text content has its own node, and so on.

The Root: document

The top-level object representing the entire webpage.
You access everything else through the document object.

Types of Nodes:

Element Node: HTML tags (e.g., , <p>, <img>)
Text Node: The actual text content within elements.
Attribute Node: Attributes within HTML tags (e.g., class, id, src)

Selecting Elements

getElementById(): Gets an element with a specific 'id'.
getElementsByClassName(): Gets a collection of elements with a specific 'class'.
querySelector(): Flexible selection, finds the first matching element (by id, class, tag, or complex CSS selector).

Manipulating the DOM

Modifying Content

textContent: Change plain text inside an element.
innerHTML: Change text and HTML markup.

Modifying Styles

element.style.propertyName: Change inline styles.

```
let heading =
document.getElementById("main-heading");
heading.style.color = "blue";
heading.style.backgroundColor = "yellow";
```

Adding/Removing Classes

element.classList.add()
element.classList.remove()
element.classList.toggle()

Creating Elements & Traversal

Creating Elements

document.createElement()

```
let newListItem = document.createElement("li");
```

Appending Elements

parentNode.appendChild(childNode)

Traversal: (Moving around the tree)

parentNode
childNodes
nextSibling, **previousSibling**

Glossary

DOM: Document Object Model - how the browser represents a webpage.
Node: A single item in the DOM tree.
Element: An HTML tag within the DOM.
Attribute: A property associated with an element.
Traversal: The act of navigating the DOM tree structure.

Intermediate

7: Events

Responding to User Actions

Events are signals that something happened: a button click, a mouse hover, a form submission, etc.
JavaScript can "listen" for them and run code in response.

Adding Event Listeners

addEventListener()

```
let button = document.getElementById("myButton");
button.addEventListener("click", function() {
    console.log("Button clicked!");
});
```

Parts:

Target Element: The element to listen on.
Event Type: "click", "mouseover", "submit", etc.
Handler Function: Code to execute when the event happens.

The event Object

The handler function often receives an event object with details:

event.type (The event type)
event.target (The element that triggered the event)
And many more depending on the event type

Common Events & Handling

Mouse Events

click: A single click
mouseover: Mouse hovers over an element
mouseout: Mouse moves away from an element

Keyboard Events

keydown: A key is pressed
keyup: A key is released

Form Events

submit: A form is submitted
change: A form input value changes

Example: Displaying a message on click

```
let messageBox =
document.getElementById("message");
button.addEventListener("click", function() {
    messageBox.textContent = "You clicked the
button!";
});
```

Advanced Concepts

Preventing Default Behavior

```
let form = document.querySelector("form");
form.addEventListener("submit", function(event) {
    event.preventDefault(); // Prevent normal form
submission
    // Do your own validation and submission logic
here
});
```

Event Propagation: Bubbling & Capturing

Bubbling: Event travels from the target element up the DOM tree to its parents.

Capturing: *(less common)*: Event travels down the tree.

Glossary

Event: A signal representing something that happened on the

webpage.
Event Listener: Code that "listens" for a specific event.
Event Handler: A function that runs in response to an event.
Event Object: Contains details about the event that was triggered.
Propagation: How events travel through the DOM tree.

8: Scope & Hoisting

Scope: Where Variables Live

Scope determines where variables and functions can be accessed within your code.
Think of scope as a series of nested boxes.

Types of Scope in JavaScript

Global Scope: Variables declared outside any function have global scope – they're accessible from anywhere.

Function Scope: Variables declared with let or const inside a function have function scope – they only exist inside that function.

Block Scope (ES6+): Variables declared with let or const within a block of code (like an if statement or a loop) have block scope.

Hoisting

A Confusing JavaScript Behavior

Hoisting seems to "lift" variable and function declarations to the top of their scope.

Key Points:

Only declarations are hoisted, not assignments.
var declarations are hoisted and initialized with undefined.
let and const are hoisted BUT not initialized, so accessing them before the declaration line will cause an error.

Example

```
console.log(myName); // undefined  (var is hoisted, but not initialized)
sayHello();          // Works (function declarations are fully hoisted)

var myName = "Alice";
```

```
function sayHello() {
    console.log("Hello!");
}
```

Best Practices and Gotchas

Best Practices

Use let and const: Avoid var due to hoisting quirks.
Declare variables close to their use: Improves readability.
Mind the Global Scope: Limit global variables to prevent naming conflicts.

Gotchas:

Using variables before declaration with let/const: Causes errors.
Accidental global variables: Happens if you forget let, const, or var.

Glossary

Scope: The visibility and lifetime of variables and functions.

Hoisting: The behavior of seemingly moving declarations to the top of their scope.

Global Scope: The outermost scope accessible from anywhere in the code.

Function Scope: Variables declared within a function are only accessible within that function and its nested functions.
Block Scope: Variables declared with let and const inside a code block ({}) are only accessible within that block.

9: Classes & Object-Oriented Programming (OOP)

Blueprints for Creating Objects

Classes define the properties and methods that objects of that type will have.

Think of a class as a template for creating similar objects.

Class Syntax

```
class Car {
    constructor(make, model, year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
    startEngine() {
        console.log("Vroom!");
    }
}
```

Key Parts

class Keyword: Declares a class.
constructor: Special method called when creating an object.
this: Refers to the current object being created.

Creating Objects and Using Methods

Instantiating Objects

Use the new keyword to create an object from a class.

```
let myCar = new Car("Toyota", "Camry", 2023);
```

Accessing Properties and Methods

```
console.log(myCar.make); // Output: Toyota
```

```
myCar.startEngine();      // Output: Vroom!
```

OOP Principles

Encapsulation: Bundling data and behavior together within a class

Abstraction: Hiding implementation details, exposing a clean interface

Inheritance

Building Class Hierarchies

Inheritance allows one class to inherit properties and methods from another.

Promotes code reusability.

```
class ElectricCar extends Car {
   constructor(make, model, year, range) {
      super(make, model, year); // Call parent
constructor
      this.range = range;
   }
}
```

Glossary

Class: A blueprint for creating objects.

Object: An instance of a class.

OOP: Object-Oriented Programming – paradigm centered around objects.

Constructor: A special method to initialize properties when creating an object.

Inheritance: One class deriving properties and methods from another class.

Let's Keep Going! Should we tackle Topic 10 (Error Handling) next, or do you have a different focus in mind?

10: Error Handling

Preventing JavaScript Meltdowns

Errors happen: typos, bad data, unexpected situations.
Error handling gracefully recovers, giving a better user experience instead of crashes.

The try...catch Block

```
try {
    // Code that might throw an error
    let result = 5 / nonExistentVariable;
} catch (error) {
    // Code to handle the error
    console.error("An error occurred:", error);
}
```

Key Points

Code inside the try block is executed.
If an error occurs, execution jumps to the catch block.
The error object in catch contains error details.

Catching Errors

The error Object

error.message: A description of the error.
error.name: The type of error (e.g., "TypeError", "ReferenceError")

Custom Errors with throw

```
function validateInput(input) {
    if (input.length < 5) {
        throw new Error("Input must be at least 5 characters long.");
    }
}
```

The finally Block

Code in finally always runs, whether or not there was an error:

```
try {
    // ...
} catch (error) {
    // ...
} finally {
    console.log("This code always executes");
}
```

Tips, Gotchas & Common Errors

Tips:

Use try...catch liberally around risky code.
Log errors or display user-friendly messages.

Gotchas

Error handling can't prevent all errors, only manage them.

Common Error Types:

TypeError: Trying to operate on the wrong data type.
ReferenceError: Using an undeclared variable.
SyntaxError: Mistakes in your code's syntax.

Glossary

Error: An object representing something going wrong in code.
Error Handling: The process of catching and responding to errors.
try...catch: The core mechanism for error handling in JavaScript.
throw: Used to create and signal custom errors.
finally: A block that always executes after try and catch.

11: Asynchronous JavaScript

JavaScript Isn't Always Step-by-Step

Some operations (network requests, timers) take time. JavaScript doesn't wait around for them to finish.

Asynchronous programming lets your code continue while these long operations run in the background.

The Event Loop

A behind-the-scenes manager in the browser.

Checks if the main code is done running. If so, it takes the next thing from the queue and executes it.

Callbacks (Older Style)

A function you pass to another function, to be called later when the long operation finishes.

```
setTimeout(function() {
    console.log("This runs after 2 seconds");
}, 2000);
```

Promises

A Cleaner Way to Handle Async

Promises represent the eventual result of an asynchronous operation (success or failure).

They have states: pending, fulfilled (success), rejected (error).

Creating a Promise

```
const myPromise = new Promise((resolve, reject) =>
{
    // Do some async work...
    if (success) {
        resolve("The data you requested");
    } else {
        reject(new Error("Something went wrong"));
    }
```

```
});
```

Using Promises

```
myPromise
    .then(result => console.log(result))  // On
success
    .catch(error => console.error(error)) // On
error
```

Async/Await (Modern Style)

Syntactic Sugar on Top of Promises

Makes async code look more like step-by-step code, without losing the power of asynchronous patterns.

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

Glossary

Asynchronous: Code that doesn't run immediately and sequentially.

Event Loop: The mechanism that handles asynchronous tasks in JavaScript.

Callback: A function passed as an argument to be executed later.

Promise: An object representing the eventual outcome of an asynchronous operation.

Async/Await: Syntax for working with promises in a cleaner way.

12 : AJAX and Fetch API

Making Web Requests without Reloading

AJAX (Asynchronous JavaScript and XML) is a technique to fetch data from a server and update parts of a webpage without a full refresh.

The Fetch API is a modern, promise-based replacement for the older XMLHttpRequest way of doing AJAX.

The Fetch API Basics

```
fetch('https://api.example.com/data')
    .then(response => response.json())  // Parse
the response as JSON
    .then(data => {
        // Update the webpage with the fetched data
    })
    .catch(error => console.error(error));  //
Handle errors
```

Working with Fetch

Understanding the Response Object

response.ok: True if the request was successful (HTTP status code 200-299).

response.status: The HTTP status code.

response.json(): Method to parse the response body as JSON (if applicable).

Common Fetch Options

```
fetch('https://api.example.com/users', {
    method: 'POST',  // GET, POST, PUT, DELETE,
etc.
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(userData) // For sending
```

```
data
})
```

AJAX with XMLHttpRequest

The Older Way (for reference)

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data');
xhr.onload = function() {
    if (xhr.status === 200) {
        const data = JSON.parse(xhr.responseText);
        // Update the webpage with the fetched data
    }
};
xhr.send();
```

Glossary

AJAX: A technique for making web requests and updating webpages asynchronously.

Fetch API: Modern, promise-based way to perform AJAX.

XMLHttpRequest: Older way to perform AJAX.

HTTP Request: A message sent to a web server.

HTTP Response: The message sent back from the server.

Let's Keep Going! Should we dive into modern JavaScript features (Modules, Regular Expressions, etc.), or do you have a particular area of interest?

Advanced

13: Modules

Building Blocks for Your Code

Modules let you organize code into reusable, self-contained units.
Each module has its own scope, preventing naming conflicts.
Think of modules like Lego bricks for building your application.

Exporting Values

Use the export keyword to make variables, functions, or classes available to other modules.

```
// myMath.js
export const PI = 3.14159;

export function calculateArea(radius) {
    return PI * radius * radius;
}
```

Importing Values

Use the import keyword to bring in elements from other modules and use them.

```
import { PI, calculateArea } from './myMath.js';

console.log(PI);
let circleArea = calculateArea(5);
console.log(circleArea);
```

Module Features

Default Exports

A module can have one default export:

```
// message.js
export default "Hello from the module!";
```

Importing a default export:

```
import message from './message.js';
```

Named Exports vs. Default

Named Exports: Explicitly named when exported. Promote clear identification.

Default Export: Only one per module, convenient for a main export.

Module Loaders

Previously required tools like Webpack or Parcel.

Modern browsers natively support modules using <script type="module">

Benefits of Modules

Code Organization: Break down large applications into manageable modules.

Reusability: Share modules across different parts of your project.

Dependency Management: Clearly see what a module imports and what it exports.

Avoid Global Scope Pollution: Variables and functions inside a module aren't global by default.

Glossary

Module: A self-contained unit of JavaScript code.
Export: Making parts of a module available to other modules.
Import: Using values exported from another module.
Default Export: A single, primary export from a module.
Named Exports: Multiple, individually named exports from a module.

14: Closures

Functions that Remember Their Surroundings

A closure is a function that has access to variables from its outer (enclosing) function's scope, even after the outer function has finished executing.
It's like the inner function carries a little backpack of the variables it needs.

Example

```
function outerFunction(name) {
    let message = "Hello, " + name + "!";
    function innerFunction() {
        console.log(message);
    }
    return innerFunction;
}
let greet = outerFunction("Alice");
greet(); // Output: Hello, Alice!
```

Note: innerFunction still accesses message from outerFunction even though outerFunction has finished.

Why Closures Matter

Data Encapsulation

Closures help create functions that have "private" variables.
Example: A counter function:

```
function createCounter() {
    let count = 0;
    return function() {
        count++;
        return count;
    };
}
```

Preserving State Between Calls

The inner function of a closure "remembers" values across multiple function calls.

Module Pattern (Older technique)

Closures were used to simulate private variables and methods before modern modules.

Gotchas & Tips

Gotchas

Closures reference variables, not their values at the time of creation.

Changes to a variable persist in the closure.

Tips:

Use closures to create function factories (functions that generate specialized functions).

Be mindful of how they impact memory usage if they hold onto large variables.

Glossary

Closure: An inner function retaining access to variables from its outer function's scope.

Lexical Scope: The scope defined by where a function is written, not where it's called from.

Encapsulationulation: Hiding data and implementation details within a function or module.

15: Regular Expressions (Regex)

Powerful Pattern Matching for Text

Regex defines a sequence of characters that form a search pattern.
Used for:
Finding specific strings
Validating input format (emails, phone numbers, etc.)
Extracting parts of text

Regex Basics

Literal Characters: Match themselves (e.g., /hello/ matches "hello").
Special Characters (Metacharacters):
. Matches any single character (except a newline).
\d Matches a single digit.
\w Matches a word character (letter, digit, or underscore).
* Zero or more of the preceding element.
+ One or more of the preceding element.

Using Regular Expressions in JavaScript

Creating Regex Patterns

Literal Syntax: /pattern/
RegExp Constructor: new RegExp('pattern')

String Methods with Regex

.search(regexp): Returns the index of the first match, -1 if no match.
.match(regexp): Returns an array of matches, or null if no match.
.replace(regexp, replacement): Replaces matches with a new string.

Example: Email Validation (Simplified)

```
let emailPattern =
/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
function isValidEmail(email) {
    return emailPattern.test(email);
```

```
}
```

Advanced Regex

Character Classes:

[]: Matches a single character from a set. Example: [abc] matches 'a', 'b', or 'c'.

[^] Matches any character NOT in the set.

Quantifiers:

{n}: Exactly 'n' occurrences.
{n, m}: At least 'n', at most 'm' occurrences.

Groups: (): Capture parts of the match for extraction.

Glossary

Regular Expression (Regex): A pattern for defining textual search criteria.

Metacharacter: A character in a regex with special meaning.

Pattern Matching: The process of comparing a string against a regex.

Validation: Using regex to check if input matches a certain format.

16: Web Storage

Storing Data on the Client-Side

Like super-powered cookies, but better.

Two main types:

localStorage: Data persists even after the browser is closed.

sessionStorage: Data is cleared when the browser tab/window is closed.

Simple API

localStorage.setItem('key', 'value');

localStorage.getItem('key');

localStorage.removeItem('key');

localStorage.clear(); // Clears everything

Important Notes

Web storage only stores strings. Convert complex data to JSON for storage.

Using Web Storage

When to Use It

Storing user preferences (like dark mode settings)

Caching data fetched from the server to improve performance

Offline functionality for web apps

Simple shopping cart data persistence

Example: Remembering Theme Preference

```
function loadTheme() {
    const storedTheme =
localStorage.getItem('theme');
    if (storedTheme) {
        document.body.classList.add(storedTheme);
    }
}

function toggleTheme() {
    // ... (code to toggle between light and dark
themes)
    const currentTheme =
```

```
    document.body.classList.contains('dark') ? 'dark' :
    'light';
        localStorage.setItem('theme', currentTheme);
    }
```

Security & Limitations

Security

Don't store highly sensitive data (passwords, etc.) in Web Storage.
Web storage is per domain/origin - other websites can't access its data.

Limitations

Storage size limits (usually around 5MB)
No complex querying - it's a simple key-value store.

Glossary

Web Storage: Mechanism for storing data in the web browser.

localStorage: Storage with no expiration date.

sessionStorage: Storage cleared when the browser tab/window closes.

Key-Value Store: A way to store data using pairs of "keys" and their associated "values".

JSON: JavaScript Object Notation – a format for structured data.

17: JavaScript Design Patterns

Reusable Solutions for Common Problems

Design patterns provide tried-and-tested templates for structuring your code in ways that improve flexibility, readability, and maintainability.

They're not rigid rules, but guidelines to adapt to your project.

Types of Design Patterns

Creational: Patterns that deal with object creation.

Structural: Patterns for organizing relationships between objects.

Behavioral: Patterns for improving communication and responsibility assignment between objects.

Example Patterns

Module Pattern (Structural)

Simulates private variables and methods, used for organization and encapsulation.

```
const calculatorModule = (function() {
    let privateTotal = 0;
    function add(num) {
        privateTotal += num;
    }
    return { // Expose only what we need
        add: add,
        getTotal: function() { return privateTotal;
}
    };
})();
```

Observer Pattern (Behavioral)

One object (subject) maintains a list of dependents (observers) and notifies them of changes.

```
class Subject {
    // ... (code to manage observers)
```

```
    notify() {
        // Notify all observers of a change
    }
}

class Observer {
    // ... (code to receive updates)
}
```

Benefits & Other Patterns

Benefits of Design Patterns

- Proven solutions to common problems
- Improves code reusability
- Makes code easier for others to understand

More Patterns to Explore

- Singleton (Creational)
- Factory Method (Creational)
- Decorator (Structural)
- Facade (Structural)
- And many others!

Glossary

Design Pattern: A reusable solution to a recurring coding problem.
Creational Pattern: Patterns concerned with how objects are created.
Structural Pattern: Patterns focused on how objects are composed and related.
Behavioral Pattern: Patterns describing how objects communicate and interact.
Encapsulation: Keeping data and methods private within an object or module.

18: Functional Programming in JavaScript

A Different Programming Paradigm

FP emphasizes functions as the core building blocks of your code.

Focuses on:

Immutability: Data is not modified in place, new values are created.

Pure Functions: A function's output depends only on its inputs, and it has no side effects.

Higher-Order Functions: Functions that take other functions as arguments or return them.

JavaScript is NOT Purely Functional

But it supports functional techniques that bring many benefits

Core Functional Techniques

map()

Transforms an array by applying a function to each element.

```
const numbers = [1, 2, 3];
const doubledNumbers = numbers.map(number => number
* 2);
console.log(doubledNumbers); // Output: [2, 4, 6]
```

filter()

Creates a new array with elements that pass a test.

```
const ages = [12, 25, 16, 30];
const adults = ages.filter(age => age >= 18);
console.log(adults); // Output: [25, 30]
```

reduce()

Accumulates a single value from an array.

```
const sum = numbers.reduce((total, current) =>
total + current, 0); //0 is the initial value
console.log(sum); // Output: 6
```

Benefits of FP

Immutability & Predictability

Easier to reason about code, less prone to bugs.
Better for parallelism (multithreading) since data isn't shared.

Testability

Pure functions are easy to test in isolation.

Composable Code

Higher-order functions let you chain operations for elegant data processing.

Glossary

Functional Programming: A programming paradigm focused on functions and immutability.

Pure Function: A function with predictable output based only on its input, and no side effects.

Side Effect: When a function changes something outside of its own scope.

Higher-Order Function: A function that operates on other functions.

Immutability: The concept of data not being changed in place.

19: Web Workers API

Running JavaScript in the Background

Web Workers allow you to offload heavy tasks to separate threads, preventing your main UI from freezing.

Ideal for:

- Intensive calculations
- Complex data processing
- Background network requests

How it Works

1. **Create a Worker:**

```
JavaScript
const myWorker = new
Worker('computation-worker.js');
```

2. **Communication via Messages:**

   Main Thread -> Worker: worker.postMessage(data)
   Worker -> Main Thread: Worker uses postMessage inside its own script.

Using Web Workers

Worker Script (computation-worker.js):

```
self.addEventListener('message', function(event) {
    const data = event.data;
    // Perform heavy calculations or tasks with
data
    const result = doSomeCalculation(data);
    self.postMessage(result);
});
```

Main Script:

```
myWorker.postMessage({ task: 'calculatePi',
precision: 10000 });

myWorker.addEventListener('message',
```

```
function(event) {
    const calculatedPi = event.data;
    // Update the UI with the result
});
```

Types & Considerations

Types of Web Workers

Dedicated Workers: Single script, communicate with only their creator.

Shared Workers: A script shared across multiple scripts/windows.

Considerations

Web Workers can't directly manipulate the DOM.
Data is transferred between threads by copying, not by sharing.

Glossary

Web Worker: A way to run JavaScript in a background thread.
Thread: An independent path of code execution within a program.
Main UI Thread: The thread where your page's UI interactions happen.
Message Passing: How the main thread and Web Workers communicate.

20: JavaScript Testing

Why Write Tests?

Catch bugs early in development.
Ensure changes don't break existing functionality (regression).
Provide documentation of how your code is supposed to work.

Types of Tests

Unit Tests: Test individual functions or components in isolation.

Integration Tests: Test how different parts of your application work together.

End-to-End (E2E) Tests: Test the entire application from the user's perspective.

Getting Started with a Testing Framework

Popular Options

Jest: Simple setup, wide feature set.
Mocha: Flexible, many compatible tools.
Jasmine: Focus on behavior-driven development (BDD).

Basic Test Structure (using Jest)

```
test('adds two numbers correctly', () => {
    const result = add(2, 3);
    expect(result).toBe(5);
});
```

Key Concepts

test or it: Defines a single test case.
expect: Make assertions about expected values.
Matchers (like .toBe, .toEqual, etc.): Describe how values should compare.

Advanced Testing Techniques

Test-Driven Development (TDD)

Write tests **before** writing code. Forces thoughtful design.

Mocking

Replace external dependencies with fakes for controlled testing.

Code Coverage
Tools that report on the percentage of your code covered by tests.

Glossary

Testing: The process of verifying code behavior.
Test Case: A single, defined test with expected inputs and outputs.
Test Suite: A collection of related test cases.
Assertion: A statement that checks for an expected result.
Test Runner: A tool that executes and reports on tests.
Mocking: Simulating parts of your code for focused testing.

Ready for More? We've covered a lot of ground! Some possibilities for expansion:

Specific testing scenarios (testing asynchronous code, DOM manipulation)
Setting up a more complex testing project
Debugging techniques to complement testing