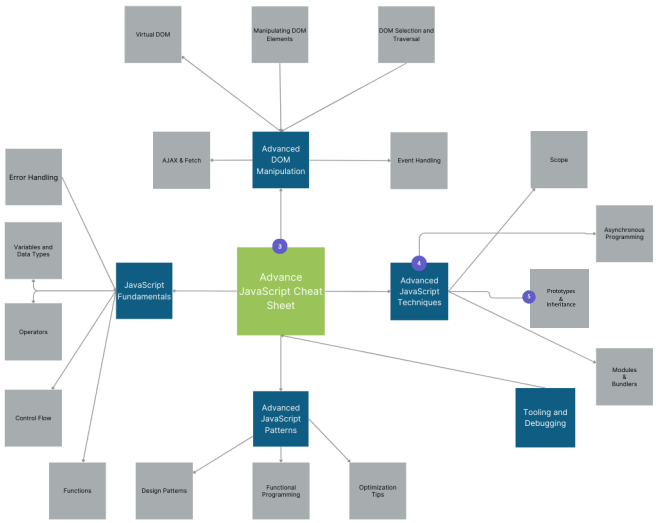
Advanced JavaScript Cheat Sheet

Today, we have got you an "Advanced JavaScript Cheat Sheet". This can be your shortcut to mastering the cool functions in JavaScript! If you're serious about making your code top-notch, this guide is ideal. In web development, knowing JavaScript well can prove to be very lucky. You get many opportunities to earn good using JavaScript. According to an expert, one thing that you will never regret learning is "JavaScript".

This cheat sheet isn't about the basics; it's your map to the exciting world of advanced JavaScript. We'll explore async programming, closures, and prototypes. They are the real game-changers. Whether you're a coding pro or just getting started, this guide gives quick answers, easy examples, and practical tips. Ready to make JavaScript your playground? Let's do this!

Course map



What Are JavaScript Fundamentals?

Our first section for this cheat sheet explores JavaScript fundamentals. It will build your basic understanding of JS fundamentals. Then, we can move to advanced-level concepts after covering the basics. Let's explore them.

Variables and Data Types :

In JavaScript, think of variables as containers for holding values. You create them using `var`, `let`, or `const`, followed by a name. On the other hand, data types let you define which kind of data you are using. It can be a string or a number. There are two main types of data types. These are given below:

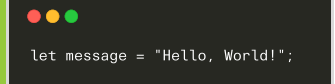
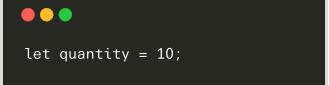
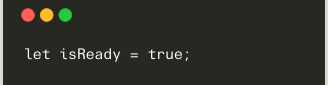
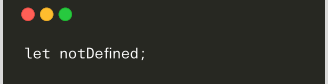
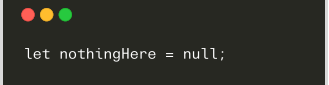
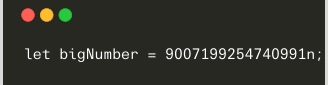
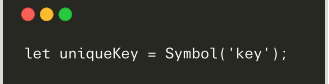
Primitive Data Types :

In JavaScript, primitive data types are the building blocks of your code. Moreover, they represent simple values that can't be broken down further. If you have a variable with a primitive value, you can give it a new value. However, you can't change the original value like you can with more complex things.

Primitives work by copying their values when used. They're like photocopies ✓ you get a copy, and the original stays as it is. There are seven primitive types in JavaScript:

1. String
2. Number
3. Bigint
4. Boolean
5. Null
6. Undefined
7. Symbol

We have created a table explaining their purpose, code example, and why we need this data type for your reference.

Primitive Data Type	Code Example	What Are They?	Why Use This Data Type?	Code Structure and Output
String		Textual informati on.	Used for storing and handling text, like messages or names.	The variable message stores the string "Hello, World!".
Number		Numeric values.	Ideal for representing numbers, like quantities or measurements.	The variable quantity is set to the numeric value 10.
Boolean		True or false values.	Employed for binary conditions, indicating true/false or on/off.	The variable isReady is assigned true, indicating a true condition.
Undefined		When declared but has no value.	Initially set when a variable is declared but not assigned a value.	The variable notDefined is declared but has no assigned value yet.
Null		The intentional absence of a value.	Explicitly used to show that a variable intentionally has no value.	The variable nothingHere is intentionally set to null.
Bigint		Large integers that can't be accurately represented.	Suitable for very large integer values.	The variable bigNumber holds a large integer value.
Symbol		Unique and immutable data type.	Often used as unique identifiers for object properties.	The variable uniqueKey is assigned a unique symbol as a key.

Reference Data Types or Non-Primitive Data Types :

Nonprimitive data types leave us with objects. Objects can be changed, and their features are shared. It means their characteristics aren't stored individually in the memory.

When you make a new variable that points to an object, it doesn't make a duplicate. Instead, it points to where the original object is saved in memory. So, if you modify the second object, it will also change the first one.

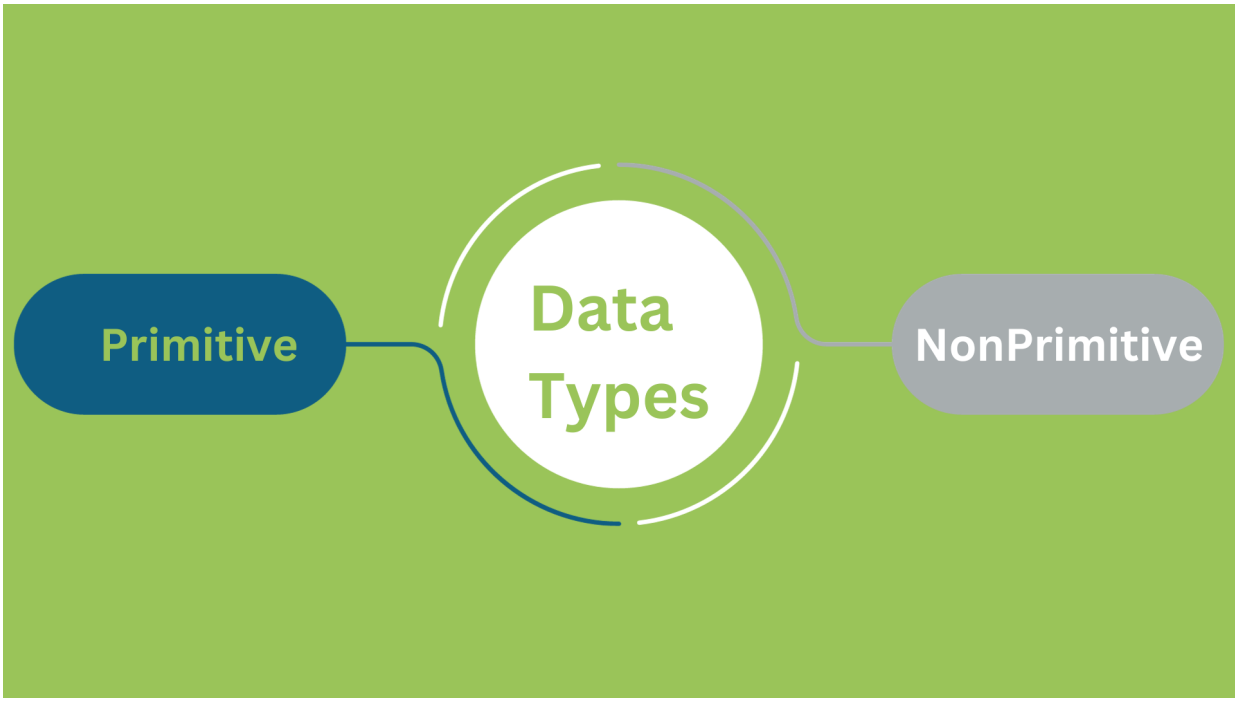
```
// Objects are passed by reference
let cat = {
  breed: "Persian",
  age: 3
};
let newCat = cat; // Points to the same memory location as cat
newCat.age = 4; // Modifies the shared memory
// Since both point to the same place...
console.log(cat); // {breed: "Persian", age: 4}
console.log(newCat); // {breed: "Persian", age: 4}
// They are both modified.

let numbers = [5, 10, 15];
let newNumbers = numbers;
newNumbers.push(20);
console.log(numbers); // [5, 10, 15, 20]
console.log(newNumbers); // [5, 10, 15, 20]
```

To avoid modifying the original object, Object.assign() or the spread operator {...} are helpful. They allow us to create a "shallow copy."

- Shallow copy replicates the object's values, but if it contains references to other objects, only the memory addresses are copied.
- For a complete duplication, including referenced objects, a deep copy is needed.

The shallow copy can be achieved with the spread operator. On the other hand, deep copy often requires using more elaborate methods, like JSON.parse and JSON.stringify. But be cautious with this method as it has limitations, especially when dealing with functions or non-serializable objects. For a more robust deep copy in complex scenarios, libraries like Lodash can be considered.



Operators

Operators are special symbols or words that help us do different operations with numbers, values, or pieces of data. We have different types of math operators. They help with calculations. There are comparison operators which check if things are the same or different. Here are the tables explaining all the operators :

Arithmetic Operators

Code Example	What Are They?	Why Use This Data Type?	Code Structure and Output
`let result = 5 + 3;`	Adds two numbers together.	Used for basic addition in calculations.	result will be 8.
`let difference = 10 ✅ 4;`	Subtracts one number from another.	Helpful for finding the difference between values.	difference will be 6.

`let product = 6 * 7;`	Multiplies two numbers.	Useful for calculations involving multiplication.	product will be 42.
`let quotient = 20 / 4;`	Divides one number by another.	Great for tasks requiring division in your code.	quotient will be 5.
`let remainder = 17 % 3;`	Finds the remainder of a division.	Often used to check for even or odd numbers.	remainder will be 2.

Comparison Operators

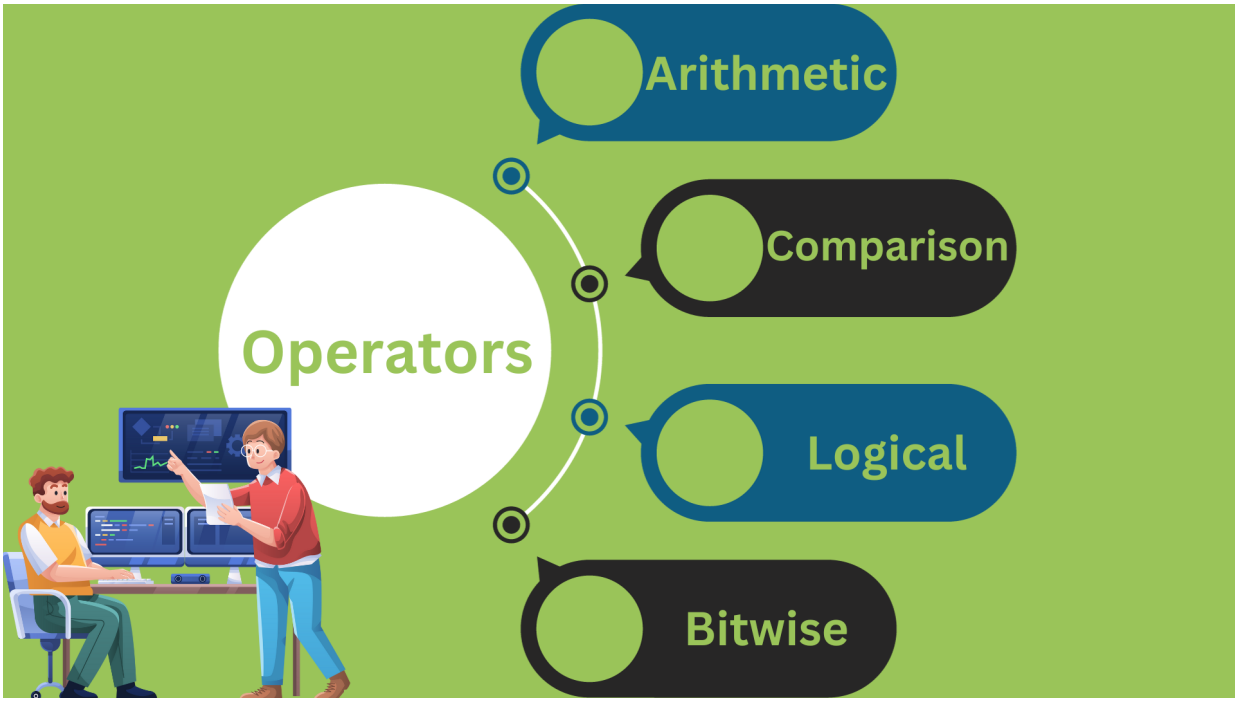
Code Example	What Are They?	Why Use This Data Type?	Code Structure and Output
`let isEqual = 5 === 5;`	Checks if two values are strictly equal.	Ensures that both the value and data type are the same.	isEqual will be true.
`let isNotEqual = 10 !== 5;`	Checks if two values are not equal.	Useful for comparing values when you want them to be different.	isNotEqual will be true.
`let isGreaterThan = 8 >`	Checks if one value is greater than another.	Helps in situations where you need to compare numerical values.	isGreaterThan will be true.
`let isLessThan = 3 < 10;`	Checks if one value is less than another.	Handy for determining the order of numerical values.	isLessThan will be true.

Logical Operators

Code Example	What Are They?	Why Use This Data Type?	Code Structure and Output
`let andResult = true && false;`	Performs a logical AND operation.	Useful when you want two conditions to be true.	andResult will be false.
`let orResult = true`		false;`	Performs a logical OR operation.
`let notResult = !true;`	Performs a logical NOT operation.	Flips the truth value, making true into false and vice versa.	notResult will be false.

Bitwise Operators

Code Example	What Are They?	Why Use This Data Type?	Code Structure and Output
`let andBitwise = 5 & 5 & 3;`	Performs a bitwise AND operation.	Useful for manipulating individual bits in binary representations.	andBitwise result depends on bitwise AND rules.
`let xorBitwise = 53;`	3;`	Performs a bitwise OR operation.	Useful when you want to set bits to 1 if either of them is 1.
`let orBitwise = 5`	Performs a bitwise XOR operation.	Helps in flipping bits when they are different.	xorBitwise result depends on bitwise XOR rules.
`let notBitwise = ~5;`	Performs a bitwise NOT operation.	Inverts the bits of a number.	notBitwise result depends on bitwise NOT rules.



Control Flow

Control flow in JavaScript is like a roadmap for your code. It decides the order in which things happen.

If Statements

You can use them to check if something is true or false and do different things accordingly.

Code Example

```
let temperature = 25;
if (temperature > 30) {
    console.log("It's a hot day!");
} else if (temperature >= 20 && temperature <= 30) {
    console.log("The weather is pleasant.");
} else {
    console.log("It's a bit chilly today.");
}
```

Output

```
The weather is pleasant.
```

Loops

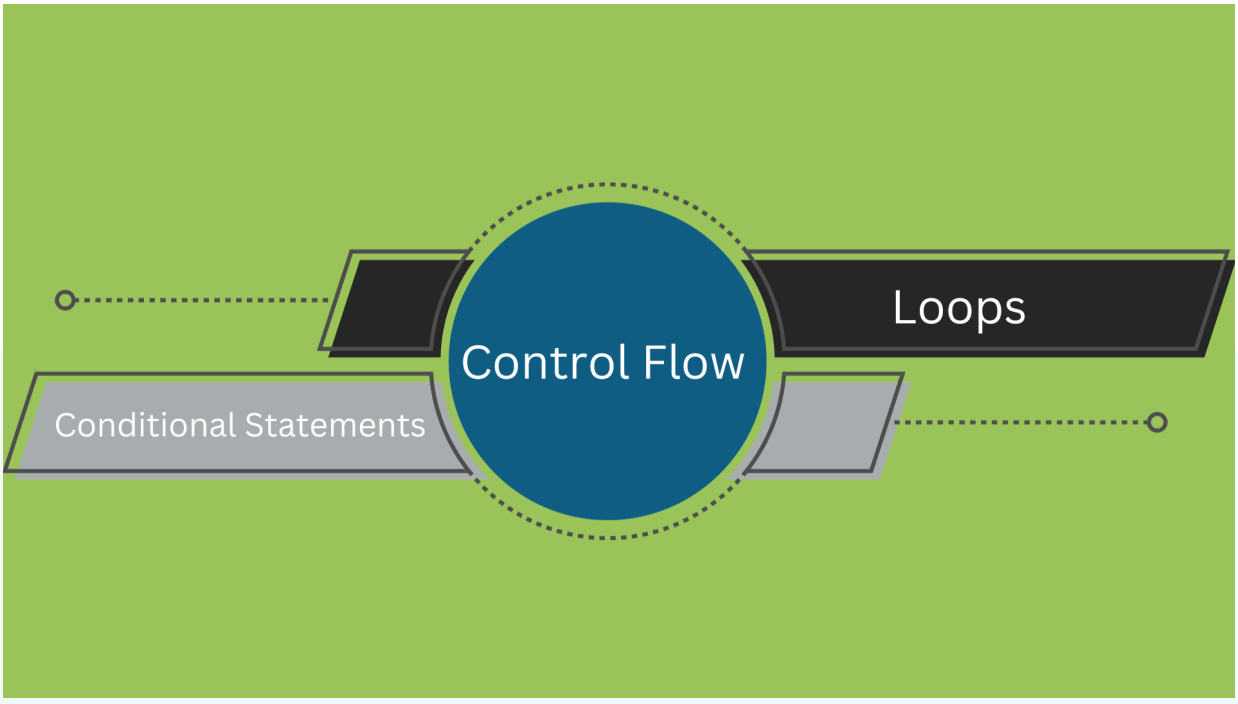
They help you repeat tasks, like going through a list of items.

Code Example

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

Output

```
1
2
3
4
5
```



Switch Statements

Like a multiple-choice decision. It checks a value and goes to the right option.

Code Example

```
// Define a variable
let dayOfWeek = "Monday";

// Use a switch statement to perform different actions based on the day of the week
switch (dayOfWeek) {
  //Case for Monday
  case "Monday":
    console.log("It's Monday! Time to start the week.");
    break;// Don't forget to break to exit the switch once a case is matched

  //Case for Tuesday
  case "Tuesday":
    console.log("Happy Tuesday! Keep going.");
    break;

  //Case for Wednesday
  case "Wednesday":
    console.log("It's Wednesday! Halfway through the week.");
    break;

  //Case for Thursday
  case "Thursday":
    console.log("Almost there, it's Thursday!");
    break;

  //Case for Friday
  case "Friday":
    console.log("Hello Friday! Weekend is near.");
    break;

  //Case for Saturday and Sunday
  case "Saturday":
  case "Sunday":
    console.log("It's the weekend! Enjoy.");
    break;

  //Default case for any day not covered above
  default:
    console.log("Not a valid day of the week.");
    break;
}
```

Output

```
It's Monday! Time to start the week.
```

Functions

When you use a function, your code jumps to that part, does some work, and comes back.

Code Example

```
// Function definition: This function adds two numbers and returns the result.
function addNumbers(a, b) {
  // The parameters 'a' and 'b' represent the numbers to be added.
  // The 'return' statement specifies the result of the function.
  return a + b;
}

// Function call: Using the addNumbers function to add5 and 7.
var sum = addNumbers(5, 7);

// Displaying the result using console.log.
console.log("Sum:", sum);

// Function definition: This function greets a person with the provided name.
function greetPerson(name) {
  //Concatenating the name with a greeting message.
  var greeting = "Hello, " + name + "!";
  //Printing the greeting message to the console.
  console.log(greeting);
}

// Function call: Using the greet Person function to greet someone named
"Alice."
greetPerson("Alice");
```

Output

```
Sum: 12
Hello, Alice!
```

Error Handling

It helps your program deal with mistakes without crashing.

Code Example

```
// Function that performs a division operation
function divideNumbers(a, b) {
    try {
        //Attempt to divide numbers
        if (b === 0) {
            // If the divisor is zero, throw a custom error
            throw new Error("Cannot divide by zero");
        }

        // If the divisor is not zero, perform the division
        let result = a / b;

        //Return the result
        return result;
    } catch (error) {
        //Catch any errors that occurred in the try block
        console.error("Error:", error.message);
        // You can also log more details about the error if needed
        //console.error("Error details:", error);

        //Return a default value or handle the error in some way
        return null;
    } finally {
        // This block is executedwhether an error occurred or not
        console.log("Division operationcompleted, regardless of success or failure.");
    }
}

// Example usage
let result1 = divideNumbers(10, 2); // Should print result and "Division operation completed"
console.log("Result 1:", result1);

let result2 = divideNumbers(5, 0); // Should print error message, then "Division operation completed"
console.log("Result 2:", result2);

let result3 = divideNumbers("abc", 2); // Should print error message, then "Division operation completed"
console.log("Result 3:", result3);
```

Output

```
Division operation completed, regardless of success or failure.
Result 1: 5
Error: Cannot divide by zero
Division operation completed, regardless of success or failure.
Result 2: null
Division operation completed, regardless of success or failure.
Result 3: NaN
```

Understanding control flow helps your code make intelligent decisions and do things in the correct order.

Best Practices for Control Flow Implementation :

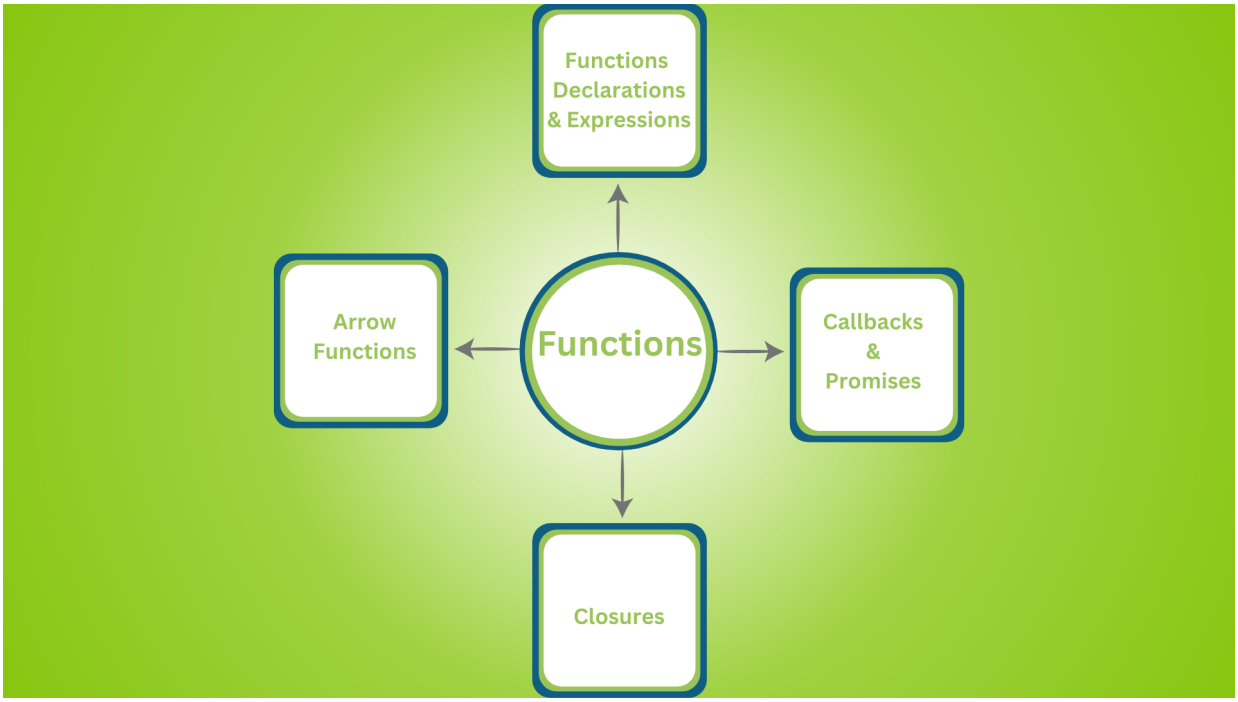
- Use names that clearly describe what a variable or function does. This helps others (or even yourself) understand the purpose without reading the whole code.
- Keep your code less indented. Too many levels of indentation make code harder to follow. Try to flatten your code structure when possible.
- Use a short form called the ternary operator for simple decisions. It's like a shortcut for if-else, useful when the conditions are straightforward.
- Deal with special cases or invalid inputs at the beginning of a function. This simplifies the main part of your code and avoids unnecessary nesting.
- When you have many conditions, use a switch statement. It's a cleaner way to handle multiple possibilities than a series of if-else statements.
- Plan for unexpected problems using try-catch blocks. This ensures that if something goes wrong, your program can respond without crashing.
- Be precise when checking if two things are equal. Use `===` instead of `==` to avoid unexpected results due to type differences.
- Consider ideas from functional programming, like keeping data unchangeable (immutable) when possible. Also, use functions that always give the same output for the same inputs (pure functions).
- Test your code in small pieces to ensure it works correctly. This catches mistakes early and gives you confidence when making changes.

Let's explore functions and error handling in more detail.

Functions :

To avoid modifying the original object, Object.assign() or the spread operator {...} are helpful. They allow us to create a "shallow copy."

- Code reusability
- Code Organization
- Code Maintainability.



Let's explore various aspects of functions in JavaScript :

Function Declarations vs. Expressions

Function declarations are hoisted. In other words, they are moved to the top of the script during compilation. As a result, you can call the function before its declaration.

Code Example

```
//Function Declaration
sayHello("Alice"); // Output: Hello, Alice!

function sayHello(name) {
  console.log('Hello, ${name}!');
}
```

Output

```
Hello, Alice!
```

- **Function expressions,** on the other hand, cannot be called before their definition due to hoisting behavior.

Code Example

```
// Function  Expression
  greet("Bob"); // Output:  TypeError: greet is not a function

  var greet = function(name) {
    console.log(`Greetings, ${name}!`);
  };
```

Output

```
Uncaught TypeError TypeError: greet is not a function
```

Arrow Functions :

Arrow functions provide a concise syntax and lexical scoping of the this keyword. They are especially useful for short, one-line operations.

Traditional Function

```
function add(x, y) {
    return x + y;
  }
  console.log(add(3, 4)); // Output: 7
```

Arrow Function

```
const add = (x, y) => x + y;

console.log(add(3, 4)); // Output: 7
```

Closures :

Closures occur when a function retains access to variables from its outer (enclosing) scope even after that scope has finished executing. In the below example, inner has access to outerVar even after outer has completed execution.

```
function outer() {
    let outerVar = 'I am from the outer function';
    function inner() {
        console.log(outerVar);
    }
    return inner;
  }
  const closureFunc = outer();
  closureFunc(); // Outputs: "I am from the outer function"
```

Callbacks and Promises

Callbacks are functions passed as arguments to another function. Moreover, they are executed after an asynchronous operation is completed.

Code Example

```
function fetchData(callback) {
  setTimeout(() => {
    constdata = 'Some fetched data';
    callback(data);
  }, 1000);
}

function processData(data) {
  console.log(`Processing data:${data}`);
}

fetchData(processData);
```

Output

```
Processing data: Some fetched data
```

- **Promises** provide a more structured way to handle asynchronous operations. As a result, they help us make code more readable and maintainable.

Code Example

```
function fetchData() {
  return new Promise((resolve, reject) =>{
    setTimeout(() => {
      const data = 'Some fetched data';
      resolve(data);
      // or reject('Error occurred'); for handling errors
    }, 1000);
  });
}

fetchData()
  .then((data) => console.log(`Fetched data: ${data}`))
  .catch((error)=> console.error(`Error: ${error}`));
// Output: (After 1 second) Fetched data: Some fetched data
```

Output

```
Fetched data: Some fetched data
```

Error Handling :

Error handling in JavaScript is important when you are dealing with robust applications. Here are the two main types :

- **Try-Catch Blocks**

  The try-catch block is a common mechanism to handle exceptions gracefully.

Code Example

```
try {
  //Risky operation
  throw new Error("This is an example error.");
} catch (error) {
  //Handle the error
  console.error(`Caught an error:${error.message}`);
}
```

Output

```
Caught an error: This is an example error.
```

Custom Error Handling

For custom error handling, developers can create their error classes to provide more context and specificity.

Code Example

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomError";
  }
}

try {
  throw new CustomError("This is a custom error.");
} catch (error) {
  console.error(`${error.name}:${error.message}`);
}
```

Output

```
CustomError: This is a custom error.
```

This was all about JavaScript basics. Let's explore the JavaScript advanced techniques.

What Are Advanced JavaScript Techniques?

Scope

Scope is like the area where a variable is recognized. There are two types :

- Global (anywhere)
- Local (only within a function or block).

Global Scope

```
let globalVar = "I'm global";

    function  exampleFunction() {
      console.log(globalVar); // Can be used here
    }

    exampleFunction();
    console.log(globalVar); // Can be used here
```

Local Scope

```
function exampleFunction() {
    let localVar = "I'm local";
    console.log(localVar); // Can be used here
  }

  exampleFunction();
  //console.log(localVar); // Error: Can't be used here
```

Lexical Scope

Lexical scope is like the neighborhood where a variable lives in a program. It decides where you can use that variable. If a variable is defined inside a certain part of the code, you can only access it there. This helps keep things organized and makes sure variables behave in a predictable way. Here is the code example:

```
function outerFunction() {
  let outerVariable = "I'm outside!";

  function innerFunction() {
    let innerVariable = "I'm inside!";
    console.log(outerVariable); // Can access outerVariable
  }

  innerFunction();
  //console.log(innerVariable); // Uncommenting this line would
result in an error
}

outerFunction();
```

In this example, outerVariable is accessible within innerFunction because of lexical scope. However, trying to access innerVariable from outside innerFunction would result in an error. This is the limitation of lexical scope.

Closure Patterns :

We discussed closures in the above sections. Interestingly, we have some patterns to use closures in JavaScript. Let's explore them.

Module Pattern

Create a box for your code with secrets inside. Only share what you want others to use.

```
const box = (function() {
     let secret = "I'm a secret";

     return{
       show: "I'm okay to share",
       reveal: function() {
         console.log("I'll share a secret:" +secret);
       }
     };
   })();

   console.log(box.show);
   box.reveal();
```

Factory Function

Make a machine that creates things with memories. Each time you use it, it remembers what happened before.

```
function createCounter() {
     let count = 0;

     return function() {
       return ++count;
     };
   }

   const counter = createCounter();
   console.log(counter());
   console.log(counter());
```

Currying

Imagine making a pizza step by step. Currying is like adding ingredients one at a time and getting a tastier result each time.

```
function curry(a) {
     return function(b) {
       return function(c) {
         return a + b + c;
       };
     };
   }

   const result = curry(1)(2)(3);
   console.log(result);
```

Callback Closure

Ask for information and tell the computer what to do when it gets the answer. It's like leaving a note for someone to follow when they find what you're looking for.

```
function fetchData(url, callback) {
     setTimeout(function() {
       const data = "Got this from " + url;
       callback(data);
     }, 1000);
   }

   fetchData("example.com/api", function(response) {
     console.log(response);
   });
```

Asynchronous Programming :

Async programming helps programming tasks happen without waiting.It plays an important role in making things faster.

Async/Await

Async/Await helps us implement asynchronous functionality in JavaScript. It makes writing code that does many things at once easier. In other words, it helps us create an independent program. As a result, we can also enhance the maintainability and readability of the code.

Code Example

```
// Function simulating asynchronous data fetching
function fetchData() {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve("Data fetched!");
      }, 2000);
    });
  }

  //Using Async/Await to handle asynchronous code
  async function fetchDataAsync() {
    console.log("Start fetching data...");

    try {
      const result = await fetchData();
      console.log(result);
    } catch (error) {
      console.error("Error fetching data:", error);
    }

    console.log("Data fetching completed!");
  }

  //Calling the asynchronous function
  fetchDataAsync();
```

Output

```
Start fetching data…
Data fetched!
Data fetching completed!
```

Prototypes and Inheritance :

In JavaScript, prototypes and inheritance are important concepts in object-oriented programming.

Constructor Functions

Constructor functions act as blueprints for creating objects. Furthermore, it helps us defind properties and methods.

Prototypal Inheritance

Prototypal inheritance enables objects to inherit features from other objects. As a result, we can promote code reuse and flexibility.

ES6 Classes

ES6 classes, introduced for cleaner syntax, provide a more structured way to implement inheritance. In other words, we are making it easier to create and manage object-oriented code.

Here is a code example showing implementation of constructor function, prototypal inheritance, and ES6 classes.

```
// Constructor function asa blueprint
function Animal(name) {
  this.name = name;
}

// Creating a method using prototype
Animal.prototype.sayHello = function(){
  console.log(`Hello, I'm ${this.name}`);
};

// Creating an object using the constructor function
const cat = new Animal("Whiskers");

// Using the method from the prototype
cat.sayHello();

// Prototypal Inheritance: Creating a more specific constructor function
function Dog(name, breed) {
  // Calling the parent constructor using 'call'
  Animal.call(this, name);
  this.breed = breed;
}
```

```
// Inheriting from Animal's prototype
Dog.prototype = Object.create(Animal.prototype);
// Setting the constructor to Dog (repairing the constructor property)
Dog.prototype.constructor = Dog;

// Adding a new method to Dog
Dog.prototype.bark = function() {
  console.log("Woof! Woof!");
};

// Creating a Dog object
const goldenRetriever = new Dog("Buddy", "Golden Retriever");

// Using methods from both Animal and Dog
goldenRetriever.sayHello();
goldenRetriever.bark();

// ES6 Classes: A cleaner way to define constructors and methods
class Bird {
  constructor(name, species) {
    this.name = name;
    this.species = species;
  }

  chirp() {
    console.log("Tweet! Tweet!");
  }
}

// Creating a Bird object
const canary = new Bird("Sunny", "Canary");

// Using the method from the class
canary.chirp();
```

Modules and Bundlers

Modules help organize code by breaking it into smaller, manageable pieces. On the other hand, bundlers help us combine those modules for use in browsers.

CommonJS & ES6 Modules

CommonJS and ES6 Modules are ways to structure and share code in JavaScript. CommonJS uses require and module.exports for importing and exporting. On the other hand, the ES6 Modules use import and export. We will see these concepts more clearly in the code example. Let's explore the bundlers first.

```
//CommonJS Example
// File: math.js
const add = (a, b) => a + b;
module.exports = { add };

// File: app.js
const { add } = require('./math');
console.log(add(2, 3));

// ES6 Modules Example
// File: math.js
export const add = (a, b) => a + b;

// File: app.js
import { add } from './math';
console.log(add(2, 3));
```

ES6 Implementation


```
// Implementation using require
const { add } = require('./mathCommonJS');
console.log(add(2, 3));

// implementation using exports
const add = (a, b) => a + b;
module.exports = { add };
```

CommonJS Implementation

Bundlers like Webpack and Rollup :

Webpack and Rollup help combine these modules for use in browsers.

- Webpack bundles modules and assets. It also helps us enhance performance and simplifying deployment.

- Rollup focuses on creating smaller, optimized bundles.

Assuming you have the ES6 Modules example (app.js and math.js) in a folder, you can use Webpack to bundle them.

First, install webpack using the below command:

npm install webpack webpack-cli --save-dev
The next step is to create a configuration file for webpack.

```
const path= require('path');

module.exports = {
    entry: './app.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

- Now, you will have a bundled file (bundle.js) in the 'dist' folder. You can include this single file in your HTML to run the application.

What is Advanced DOM Manipulation?

Advanced DOM manipulation gives us techniques to dynamically modify, update, or interact with the Document Object Model using JavaScript.

The DOM represents the structure of an HTML document as a tree-like structure of objects. Moreover, manipulating it enables developers to create dynamic and interactive web pages.

DOM Selection and Traversal

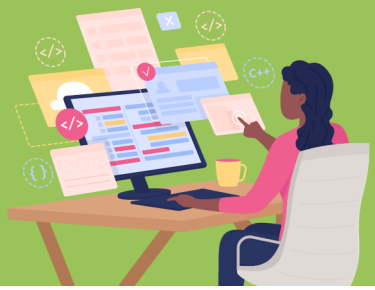
- **Query Selectors**

  Advanced DOM manipulation gives us techniques to dynamically modify, update, or interact with the Document Object Model using JavaScript.

Query Selector

```
// Example: Get the first element with class 'example'
const element = document.querySelector('.example');

// Example: Get all elements with class 'example'
const elements = document.querySelectorAll('.example');
```



Traversal Methods

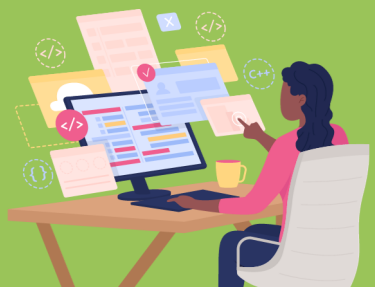
Traversal methods let you move around the DOM tree. You can find parent nodes, child nodes, and sibling nodes of an element.

```
// Example: Find the parent node
const parent = element.parentNode;

// Example: Find the first child node
const firstChild = parent.firstChild;

// Example: Find the next sibling
const nextSibling = element.nextSibling;
```

Traversal Methods



Manipulating DOM Elements

Creating and Appending Elements

Create new elements on the fly using createElement and add them to the DOM with appendChild.

```
// Example: Create a new div element
const newDiv =document.createElement('div');

// Example: Add it to an element with id 'container'
document.getElementById('container').appendChild(newDiv);
```

Modifying Element Properties

Change attributes like text content or values to update what users see.

```
// Example: Change text content
element.textContent = 'New Text';

// Example: Change an input value
document.getElementById('inputField').value = 'New Value';
```

Removing Elements

Delete elements from the DOM to manage content dynamically.

```
// Example: Remove an element
const elementToRemove = document.getElementById('toBeRemoved');
elementToRemove.parentNode.removeChild(elementToRemove);
```

Event Handling

Event handling helps us manage different events occurring in the JavaScript. Let's explore its sub-types to be more clear on event handling.

Event Listeners

Event Listeners are like ears for your webpage. They listen for events, such as clicks or keystrokes, and trigger a response, like running a function.

```
// Example: Add a click event listener
element.addEventListener('click', function(){
    console.log('Element clicked!');
});
```

Event Object

The Event Object provides details about the event. For example, where the mouse is or which key was pressed. It's like a package of information about what just happened.

```
// Example: Access event properties
element.addEventListener('mousemove', function(event){
    console.log('Mouse coordinates:', event.clientX, event.clientY);
});
```

AJAX and Fetch

AJAX is a way for web pages to talk to servers without reloading the whole page. It lets your webpage send requests to a server in the background and update only the necessary parts.

Making Asynchronous Requests

Making asynchronous requests is like asking a server for information without making the webpage wait. The Fetch API is a tool to do this efficiently. Here is an example to fetch data from a server without slowing down the webpage.

```
// Example: Fetch data from an API
fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error=> console.error('Error:', error));
```

Handling Responses

Handling responses involves dealing with the information you get back from the server. This could include showing data on the webpage or handling errors. Here is how to do it:

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

Virtual DOM

The Virtual DOM is a clever way to speed up webpage updates. It keeps a lightweight copy of the real DOM, making changes faster and more efficient. Let's differentiate between the real and virtual DOM using a table.

Aspect	Virtual DOM	Real DOM
Definition	A lightweight copy or representation of the actual DOM.	The actual document structure of a webpage.
Updating Approach	Changes are first made to the virtual representation, and then selectively applied to the real DOM for efficiency.	Directly updates the entire DOM tree when changes occur.
Rendering Process	More complex initial rendering as it involves creating a virtual tree.	Simple initial rendering, directly reflecting the HTML structure.
Performance Impact	Generally more efficient, particularly in applications with frequent updates.	Can be less efficient, especially in applications with frequent updates, as it involves updating the entire DOM tree.
Re-Rendering	Only updates the parts of the DOM that changed, reducing unnecessary work.	Updates the entire DOM, even if only a small part has changed.
Framework Use	Often associated with front-end libraries/frameworks like React.	Traditional approach used in many web development scenarios.

Virtual DOM

- A pretend version of your webpage that React uses.
- React first changes the pretend version and then updates the real webpage only where needed.
- React does a bit more work at the beginning to set up the pretend version.
- Generally makes your webpage faster, especially when things change a lot.

Code Example

```
import React, { useState } from 'react';

  function VirtualDomExample() {
    const [items, setItems] = useState(['Item 1', 'Item 2']);

    const addItem = () => {
      const newItems = [...items, `Item  ${items.length + 1}`];
      setItems(newItems);
    };

    return (
      <div>
        <button  onClick={addItem}>Add Item</button>
        <ul>
          {items.map((item, index) =>  (
            <li  key={index}>{item}</li>
          ))}
        </ul>
      </div>
    );
  }

  export default VirtualDomExample;
```

Real DOM

- The real, actual structure of your webpage.
- Changes go directly to the real webpage, affecting everything.
- Simpler at the start because it just reflects what's in your HTML.
- Can be slower, especially when a lot of changes happen.

Code Example

```
<!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,  initial-scale=1.0">
    <title>Real DOM  Example</title>
  </head>
  <body>
    <script>
      function addItem() {
        var newItem =  document.createElement('li');
        newItem.textContent = 'Item ' + (document.querySelectorAll('li').length + 1);
        document.querySelector('ul').appendChild(newItem);
      }
    </script>

    <button onclick="addItem()">Add Item</button>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
    </ul>
  </body>
  </html>
```

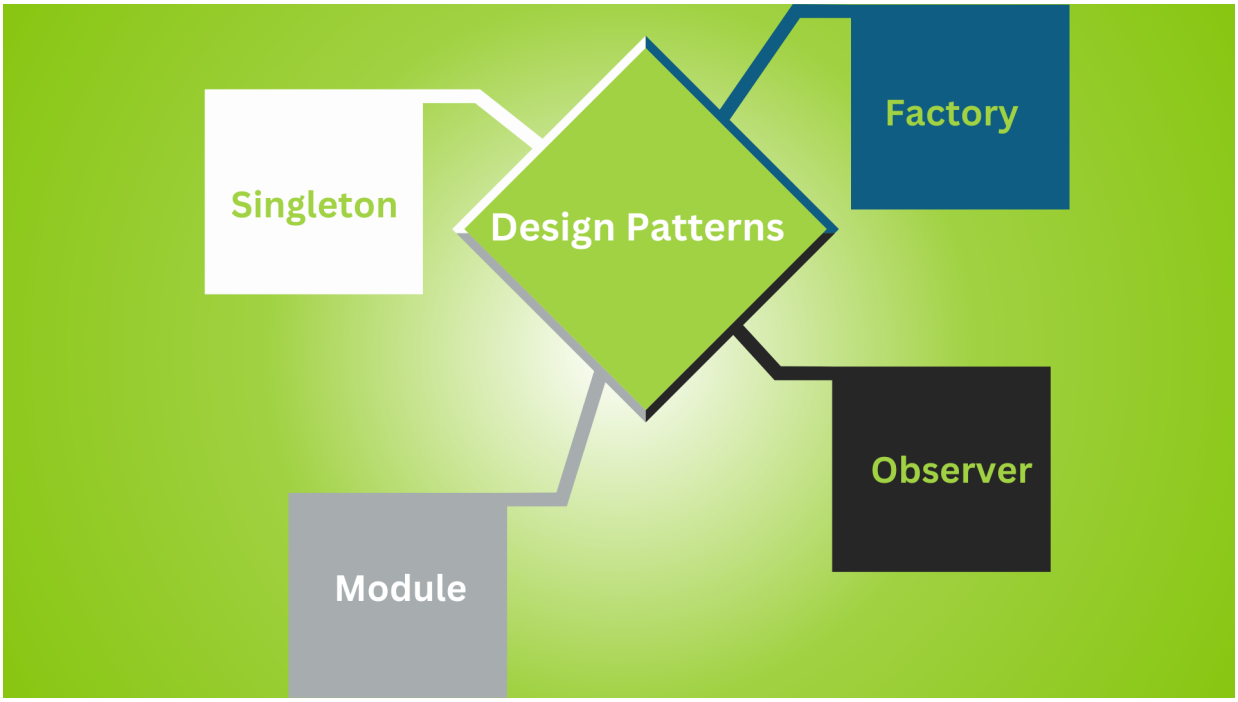
What Are the Advanced JavaScript Patterns?

Advanced JavaScript patterns are essential for writing the code that is:

- Scalable
- Maintainable
- Efficient

Design patterns offer proven solutions to common problems. Besides, they also help developers in promoting code organization and reusability. Here are some advanced JavaScript patterns:

- **Design Patterns**



Singleton

It ensures a single instance of a class. We use it when exactly one object is needed to coordinate actions across the system.

```
const Singleton = (() => {
  let instance;

  function createInstance() {
    return {};
  }

  return {
    getInstance: () => {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    },
  };
})();
```

Module

A module pattern encapsulates functionality, allowing private and public members. Moreover, it is ideal for organizing code into independent, self-contained units, preventing global namespace pollution.

```
const Module = (() => {
  const privateVariable = 'I am
private';

  return {
    publicVariable: 'I am public',
    publicFunction: () => {
      //Your logic here
    },
  };
})();
```

Observer

An observer pattern defines a one-to-many dependency between objects, notifying changes. We use it when an object's state change should notify and update multiple dependents.

```
class Observer {
  constructor() {
    this.observers = [];
  }

  addObserver(observer) {
    this.observers.push(observer);
  }

  notify(data) {
    this.observers.forEach(observer =>observer.update(data));
  }
}
```

Factory

A factory pattern creates objects based on conditions, encapsulating the creation logic. It is suitable when the system needs to be independent of how its objects are created, composed, and represented.

```
class ProductA {
  //Product A specific logic
}

class ProductB {
  //Product B specific logic
}

class ProductFactory {
  createProduct(type) {
    switch (type) {
      case 'A':
        return new ProductA();
      case 'B':
        return new ProductB();
      default:
        throw new Error('Invalid product type');
    }
  }
}
```

Functional Programming

Functional programming is about writing code in a clear way. Here are the two most common methods of functional programming.

Higher-Order Functions

They treat functions as first-class citizens. In other words, they allow them to be passed as arguments or returned from other functions.

Code Example

```
const multiplyBy = (factor) => (number) => number * factor;

const double = multiplyBy(2);
console.log(double(5)); // Output: 10
```

Immutability

Immutability refers to a state where we can't change anything further. It helps us ensure that the data remains constant once it is created. Instead of changing the existing data, we create new data. Here is the code example:

Code Example

```
const originalArray = [1, 2, 3];
const newArray = [...originalArray, 4];

console.log(originalArray); // Output: [1, 2, 3]
console.log(newArray);      // Output: [1, 2, 3, 4]
```

Optimizations and Best Practices

Here are some code optimization tips:

- Use meaningful variable names.
- Avoid unnecessary code complexity.
- Prefer built-in functions for common tasks.
- Optimize loops for better performance.
- Minimize global variables.

Performance Tips

Here are some performance tips for your code:

- Minimize DOM manipulations.
- Use asynchronous operations when possible.
- Optimize algorithms for efficiency.
- Choose efficient data structures (e.g., maps, sets).
- Reduce unnecessary function calls.
- Prioritize lazy loading.
- Optimize images for web applications.

Code Organization

Code organization plays an important role in better quality and understanding of code for others. Here are some tips for code organization:

- Organize code logically.
- Group related functions together.
- Maintain consistent naming conventions.
- Use modular design with smaller, manageable files.
- Prioritize code readability over cleverness.
- Document code effectively.
- Apply version control systems for collaborative development.
- Automate tasks using build tools.

What Are Tooling and Debugging?

Tooling refers to the use of software tools to enhance the development process. On the other hand, debugging refers to identifying and fixing errors in code.

Developer Tools

Developer tools make it easier for developers to build, test, and manage web applications.

Some examples of developer tools are :

- Visual Studio Code (VSCode)
- Git
- Jenkins
- Atlassian Bitbucket
- Prometheus
- Windows Powershell / Command Prompt

Developer tools at Sencha are :

- Visual Studio Code (VSCode)
- Git
- Jenkins
- Atlassian Bitbucket
- Prometheus
- Windows Powershell / Command Prompt

These tools make coding easier, help teams work together, and ensure that apps run smoothly. Developers choose tools based on what fits their needs and makes their work more efficient.

Debugging Techniques

Here are some debugging techniques for your code:

- Insert print statements for variable values and messages.
- Set breakpoints and step through code for controlled analysis.
- Explain code or problems to someone else or an object.
- Get others to review code for fresh insights and improvements.
- Identify the specific code section where issues occur.
- Verify function inputs and ensure expected outputs.
- Record information about program execution.
- Write tests for individual components to identify failures.
- Temporarily remove or comment out code sections to pinpoint issues.
- Use Git or similar tools to track and review recent changes.
- Use profilers to identify memory-related issues.
- Review language, library, or framework documentation.
- Take breaks to return with a fresh perspective.

Linting and Code Quality Tools

Linting and code quality tools are like helpful assistants for programmers. They look at your code to find mistakes and make sure it looks nice and follows the rules.

Linting

- Linting finds common errors and checks if your code looks clean.
- It helps catch mistakes before you run your program.
- Linters also make sure your code follows the style guide, so it looks consistent.

Code Quality Tools

- These tools look at the overall health of your code.
- They check if your code is easy to understand, performant, and secure.
- They give you feedback on how to make your code better.

Examples of these tools include ESLint and Pylint for finding mistakes. Tools like SonarQube or CodeClimate are helpful for checking the overall quality of your code. They're like spell checkers for your programming code!

Testing Frameworks

Testing frameworks are like tools that help developers check if their code works well. Tests are important because they catch mistakes early and make sure the software does what it's supposed to.

Here are some debugging techniques for your code:

1. **JUnit (Java)**
   - Checks if Java code works correctly.
   - Useful for making sure different parts of the code are okay.

2. **pytest (Python)**
   - Tests Python code to see if it's doing what it should.
   - Can handle simple or complex tests.

3. **Mocha (JavaScript - Node.js)**
   - Checks if JavaScript code, especially in Node.js, is working fine.
   - Good for different types of tests, including ones that happen at the same time.

4. **NUnit (C#)**
   - Tests C# code to ensure it works well.
   - Useful for checking different parts of the code.

5. **RSpec (Ruby)**
   - Tests Ruby code and helps describe how it should behave.
   - Often used with Cucumber for special types of testing.

6. **Jest (JavaScript - React, Node.js)**
   - Checks if JavaScript code, especially in React, is working.
   - Quick and easy to set up for testing.

7. **PHPUnit (PHP)**
   - Tests PHP code to make sure it's working properly.
   - Can handle different levels of testing.

8. **Cypress (JavaScript - End-to-end testing)**
    - Tests web applications, making sure they work from start to finish.
    - Especially good for testing how users interact with a website.

9. **Selenium (Cross-browser testing)**
    - Tests web applications across different browsers.
    - Makes sure the website works well for everyone.

Conclusion :

JavaScript cheat sheet explores the basic and advanced JavaScript concepts. We have tried to clear each and every concept with code examples for your understanding. We also shared the best tips for optimized code. Finally, we also talked about some testing tools to ensure that your JS applications work smoothly.

USEFUL RESOURCES

- How to get started with Ext JS
- Download FREE 30 days trial of Ext JS
- View Examples and documentation
- Video Tutorials
- Front-end developers, unlock rapid web app scaling! Dive into our exclusive Developer Skill Sprints series.
- Don't miss out! Learn to build a Gmail-like app step-by-step using Ext JS. Get started now.