HubSpot

An Introduction to JavaScript

A Hands-On Guide

What is JavaScript?

JavaScript is one of the three core web development technologies, alongside Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS). While HTML structures the webpage's content and CSS handles styling and layout, JavaScript adds interactivity, logic, and advanced features, which this guide explores further. Developers combine these technologies to make websites esthetically pleasing and engaging.

Brendan Eich created JavaScript in 1995 for the Netscape Navigator browser. This versatile programming language has become an essential tool for web developers thanks to its ability to enhance websites with interactivity and dynamic content.

The JavaScript language enables web developers to create richer, more interactive user experiences. Some of its typical use cases are:

Form validation: JavaScript can validate data that users have entered into a form before they submit it, ensuring the data is correct and complete.

Animations and transitions: JavaScript creates smooth animations, transitions, and other visual effects to make websites more engaging and visually appealing.

Interactive content: JavaScript helps developers create interactive elements such as image sliders, accordions, and tabbed content areas that respond to user input.

Fetching and displaying data: JavaScript can send and retrieve data from a server and update a webpage's content without reloading the page.

In contrast, developers primarily use HTML and CSS for static content and styling. HTML builds the webpage's fundamental structure, formatting elements such as text and images, while CSS applies styles such as colors, fonts, and layouts to the HTML elements.

JavaScript can handle client-side and server-side scripting. Client-side scripting is JavaScript code that runs in the user's browser, directly interacting with the HTML and CSS to create dynamic content and interactivity. Server-side scripting is JavaScript code that runs on a web server such as Node.JS, processing data and generating HTML that goes to the user's browser.

By learning and mastering JavaScript, you'll be prepared to take on a great variety of web development tasks and create impressive web applications. After finishing this guide, you'll be ready to tackle basic web development projects with a solid foundation to expand your knowledge further.

How to Use This Guide

Throughout this guide, you'll find a comprehensive introduction to the most fundamental concepts in JavaScript. The guide is structured to provide quick references for essential concepts.

Go ahead and experiment with the various code samples throughout the guide. Rather than building entire projects, this guide focuses on providing quick and easily-digestible introductions to operators, functions, and syntax necessary to write JavaScript applications.

Experimenting with JavaScript from Your Web Browser

If you're a beginner to JavaScript and unsure how to use the example code, the easiest way to experiment with these code samples is by embedding them in an HTML file. Use the <script> tag to include your JavaScript code directly within your HTML.

For example, you might open a blank text file and paste the following HTML and JavaScript:

```
<!DOCTYPE html>
<html>
<head>
   <title>My First JavaScript</title>
</head>
<body>

<h2>My First JavaScript Program</h2>

<button onclick=”displayMessage()”>Click
me
</button>

<p id=”demo”></p>

<script>
function displayMessage() {
  document.querySelector(“#demo”).
textContent = “Welcome to your first
JavaScript program.”;
}
</script>

</body>
</html>
</html>
```

This example is an HTML document with a header, button, and paragraph.

A best practice is to always place JavaScript at the bottom of an HTML page. This positioning allows the HTML to load and render completely before any JavaScript code begins to execute. It also enables the page to function as a static HTML page, even if JavaScript is disabled.

In this example, the <script> tag at the end of the body section includes the JavaScript code. It defines a function displayMessage in JavaScript, which changes the paragraph element's text with the ID demo to Welcome to your first JavaScript program when a user clicks the button.

The onclick attribute in the button tag is an event handler that executes the displayMessage function when somebody clicks the button. This code is a simple example of how JavaScript can interact with HTML elements to create dynamic content.

To run this sample code, simply save it in an HTML file, then open the file in a web browser to see it in action.

Throughout this guide, we encourage you to copy-paste the example code snippets into a similar HTML file, enclosing your JavaScript in <script></script> tags and experimenting with the example code to create your own projects.

Variables, Data Types, and Operators

Variables, data types, and operators form the foundation of JavaScript. Variables allow you to store and manipulate data, data types define the nature of this data, and operators enable operations and calculations.

Declaring Variables

There are three ways to declare a variable: var, let, and const.

let

let declares variables that you can reassign later:

```
let counter = 0;
counter = 1; //
Reassigning the
variable is allowed
```

const

const declares variables you cannot reassign:

```
const pi = 3.14159;
pi = 3.14; //
Reassigning is not
allowed and will
cause an error
```

var

var is the older way to declare variables:

```
var myVar =
‘Hello’;
```

It’s generally not recommended to use var in modern JavaScript code. Web developers prefer to use let and const for their benefits, including block scoping, safer hoisting, disallowance of re-declaration in the same scope, and const’s support for immutability.

Data Types

Data types are categories that help define the kind of data a variable can hold. Understanding data types is crucial for novice developers to learn how to work with and manipulate data in JavaScript.

Strings

Strings represent text or characters. You can enclose them in single (‘ ’) or double (“ ”) quotes:

```
let greeting = “Hello, World!”;
let name = ‘Alice’;
```

String templates, introduced in JavaScript standard ECMAScript 6 (ES6), provide an easy way to embed expressions in string literals. You define them using backticks (`) instead of quotes (‘ or “):

```
let name = “John”;
let greeting = `Hello, ${name}!`; // Hello, John!
```

The ${expression} inserts the result of the expression into the string, making concatenating strings and values a clean and simple process.

Numbers

You can write numbers in JavaScript with or without decimal points. Unlike some other programming languages, JavaScript doesn’t have separate data types for integers (whole numbers) and floating-point numbers (numbers with decimals).

```
let age = 30; // An integer
let price = 19.99; // A floating-point number
```

Booleans

Booleans represent true or false values, often in conditional statements.

```
let isRaining = true;
let isSunny = false;
```

Symbols

Symbols in JavaScript represent a unique value that can never repeat. This attribute makes them ideal identifiers for an object's properties, ensuring there won't be any accidental naming conflicts.

To create a symbol, use the Symbol function:

```
let symbol1 = Symbol('mySymbol');
console.log(typeof symbol1); // "symbol"
```

The real magic of symbols is that each is unique. Even if you create two symbols with the same description, they aren't equal, as demonstrated below:

```
let symbol1 = Symbol('mySymbol');
let symbol2 = Symbol('mySymbol');

console.log(symbol1 === symbol2); //
Output: false
```

Although 'mySymbol' describes both symbol1 and symbol2, they're not the same symbol.

Objects

Objects are like containers for values, letting you group related data and functions together. Think of an object like a backpack: You can stuff it with related items (values) and label each pocket (property) to find what you need easily.

As an example, create an object below to represent a person:

```
let person = {
  name: 'Alice',
  age: 30
};
```

In this person object, name and age are properties (sometimes informally called keys), and 'Alice' and 30 are their corresponding values.

Now that you have an object, you can retrieve its stored data. Go back to your person object. If you want to find out the person's name, do so using the code below:

```
let personName = person.name;
console.log(personName); //
Output: Alice
```

In this example, name is a property of the person object. When you say person. name, you're asking JavaScript to give you the value associated with the property 'name' in the person object.

Similarly, you can get the person's age using person.age:

```
let personAge = person.age;
console.log(personAge); //
Output: 30
```

So, by using the property, you can access the corresponding value in an object. This approach makes objects a handy tool for grouping and managing related data.

Null

Null is a special value indicating “no value” or “no object.” It has two main purposes in JavaScript. First, you can assign it a variable that you plan to use later as an object, demonstrating an intent to use the variable for an object in the future. Second, you can use it to indicate or clarify an object reference, implying that the object is no longer needed and can be discarded. This use case aids the process of memory management, called garbage collection.

```
let emptyValue = null;
```

Undefined

The undefined value indicates that the code has declared a variable but hasn’t assigned it a value. In most cases, JavaScript developers don’t intentionally set a variable to undefined because variables are automatically initialized as undefined when they’re declared.

```
let notAssigned;
console.log(notAssigned); // Output: undefined
```

Working with Operators

Operator symbols perform actions on values and variables. This section outlines the primary operators in JavaScript.

Arithmetic Operators

Arithmetic operators perform basic arithmetic operations.

```
let a = 10;
let b = 5;

let sum = a + b; // Addition
let difference = a - b; // Subtraction
let product = a * b; // Multiplication
let quotient = a / b; // Division
let remainder = a % b; // Modulus (remainder)
```

Logical Operators

Logical operators determine the logic between variables or values, returning boolean values as either true or false. JavaScript has three logical operators:

- **Logical AND (&&):** This operator returns true if both operands (values or variables) are true.
- **Logical OR (||):** This operator returns true if at least one operand is true.
- **Logical NOT (!):** This operator returns true if the operand is not true (that is, it returns the opposite Boolean value of the operand).

```
// Logical AND (&&)
console.log(true && false);  // Output: false

// Logical OR (||)
console.log(true || false);  // Output: true

// Logical NOT (!)
console.log(!true);  // Output: false
```

Comparison Operators

Comparison operators compare values and return a boolean result:

```
let num1 = 10;
let num2 = 20;

let isEqual = num1 == num2; // Equality: false (checks if num1 and num2 are equal)
let isNotEqual = num1 != num2; // Inequality: true (checks if num1 and num2 are not equal)
let isGreater = num1 > num2; // Greater than: false
let isLess = num1 < num2; // Less than: true
let isGreaterOrEqual = num1 >= num2; // Greater than or equal: false
let isLessOrEqual = num1 <= num2; // Less than or equal: true
```

You can use either == or === when comparing values. These two operators look similar but work differently. While == compares values and ignores their type, === compares values and types, making it a stricter comparison.

For example, 5 == '5' is true because JavaScript changes the string '5' into a number before checking:

```
let num = 5;
let str = '5';
console.log(num == str);  // Outputs: true
```

=== checks if the values and types are equal. There's no conversion. For example, 5 === '5' is false because one is a number and the other is a string, so they aren't the same:

```
let num = 5;
let str = '5';
console.log(num === str);  // Outputs: false
```

This makes === a stricter check than ==. It's usually safer to use === to avoid inaccurate results.

Logical Operators

Compound assignments combine an assignment with another operation.

```
let count = 5;

count += 10; // Equivalent to count = count + 10;
count -= 3; // Equivalent to count = count - 3;
count *= 2; // Equivalent to count = count * 2;
count /= 4; // Equivalent to count = count / 4;
```

Order of Precedence

When an expression uses multiple operators, the order of precedence determines the sequence to perform them. Generally, the order is parentheses, exponentiation, multiplication and division, and addition and subtraction. You can use parentheses to control the order of execution:

```
let count = 5;

count += 10; // Equivalent to count = count + 10;
count -= 3; // Equivalent to count = count - 3;
count *= 2; // Equivalent to count = count * 2;
count /= 4; // Equivalent to count = count / 4;
```

Control Flow

The control flow is the precise order in which JavaScript statements are executed. Think of a roadmap that tells JavaScript how to go through the code, allowing you to control the sequence of actions and instructions in your programs.

Conditional Statements

Conditional statements perform decisions. They allow your program to run differently based on whether a certain condition is true or false. They also help control the program's execution flow.

To write clean, maintainable code with conditional statements, you must use clear and concise conditions, avoid deeply nested if statements, and use the ternary operator for simple cases.

if/else Statements

if/else statements execute a block of code if a specified condition is true and another block of code if the condition is false:

```
let temperature = 25;

if (temperature > 30) {
  console.log("It's hot outside!");
} else {
  console.log("It's not too hot.");
}
```

Nested if Statements

Nested if statements are inside another if statement. They help check multiple conditions in a sequence:

```
let score = 85;

if (score >= 90) {
  console.log(“You got an A!”);
} else if (score >= 80) {
  console.log(“You got a B!”);
} else if (score >= 70) {
  console.log(“You got a C!”);
} else {
  console.log(“You did not pass.”);
}
```

Ternary Operator

A ternary operator is a shorthand for an if/else statement. It takes three operands: a condition, a value to return if the condition is true, and a value to return if the condition is false:

```
let age = 18;
let canVote = age >= 18 ? “You can vote!” : “You cannot vote.”;
console.log(canVote);
```

Loops

Loops repeat a block of code a specified number of times or until it meets a certain condition.

while Loop

The while loop executes a block of code as long as the specified condition is true. The example below runs until the variable reaches the value 5, increasing the value each loop iteration:

```
let i = 0;

while (i < 5) {
  console.log(i);
  i++; // Increment the value of i
}
```

for Loop

The for loop has an initialization, condition, and increment/decrement expression. It's useful when you know how many times you want to repeat the code:

```
for (let i = 0; i < 5; i++) {
  console.log(i); //Outputs numbers 0-4
```

for..of Loop

The for..of loop iterates over the elements of an iterable object, such as arrays and strings. It helps you navigate all elements in a collection without needing an index:

```
let fruits = ["apple", "banana", "orange"];

for (let fruit of fruits) {
  console.log(fruit); //First output will be "apple", second "banana", third "orange"
}
```

The order of iteration for a for..of loop over an iterable object is not technically guaranteed.

For example:

```
let fruits = ["apple", "banana", "orange"];

for (let fruit of fruits) {
   console.log(fruit);
   // Always prints:
   // apple
   // banana
   // orange
}
```

The fruits array iterates in index order, so you're guaranteed that output order.

But for a generic object, the iteration order is engine-dependent:

```
let obj = {
  a: 1,
  b: 2,
  c: 3
}

for (let key in obj) {
  console.log(key);
  // Could print a b c or c b a or any other order
}
```

For built-in objects like arrays, strings, maps, and sets, for..of always iterates properties in ascending index order. It guarantees ordered iteration for these objects. If ordered iteration is important, use a map or array instead of a plain object.

Whenever you need ordered iteration over a data structure in JavaScript, double-check that the object you're looping over guarantees order. If it doesn't have a guaranteed order, the output could vary between environments and engine versions.

Infinite Loops

It's important to break out of loops to avoid infinite loops that can cause the program to freeze or crash. You can use the break keyword to exit a loop and the continue keyword to skip the rest of the current iteration and move on to the next:

```
let score = 85;

if (score >= 90) {
  console.log("You got an A!");
} else if (score >= 80) {
  console.log("You got a B!");
} else if (score >= 70) {
  console.log("You got a C!");
} else {
  console.log("You did not pass.");
}
```

Understanding control flow with conditional statements and loops is essential for writing effective JavaScript code. As you practice, apply these concepts to create more complex programs and improve your programming skills.

Object-Oriented Programming

Object-oriented programming (OOP) organizes code using objects, which can represent real-world entities like people, cars, or anything else. Objects have properties that store data and methods that perform actions.

OOP helps organize code more intuitively, making it easier to understand, maintain, and reuse. Using classes and objects, you can create modular code representing real-world entities and their behaviors. This approach improves productivity and makes it easier to collaborate with other developers.

With a basic understanding of object-oriented programming in JavaScript, you can create more complex, organized, and maintainable code. Practice defining classes, creating objects, and using inheritance to develop your programming skills further.

Defining Classes

In OOP, a class is a blueprint for creating objects. It defines the properties and methods that objects have. In JavaScript, you can create classes using the class keyword, followed by the class name (typically in PascalCase, with the first letter of every word capitalized and no spaces). Then, you define properties and methods inside the class.

```
class Dog {
  // Properties and methods go here
}
```

A constructor is a special method that runs when you create a new object from a class. It sets the initial values of the object's properties. To define a constructor, use the constructor keyword inside the class, followed by parentheses containing any needed parameters:

```
class Dog {
  constructor(name, breed) {
    this.name = name;
    this.breed = breed;
  }
}
```

Inheritance allows a class to inherit properties and methods from another class, reducing code duplication and promoting reusability. Use the extends keyword to create a subclass that inherits from a parent class. To call the parent class's constructor, use the super function:

```
class ServiceDog extends Dog {
  constructor(name, breed, job) {
    super(name, breed);
    this.job = job;
  }
}
```

To create an object from a class, use the new keyword followed by the class name and any arguments needed for the constructor:

```
const myDog = new Dog("Fred", "Rottweiler");
```

Methods

Methods are actions that the code can perform on objects. They are functions associated with an object.

For instance, consider the Dog class. This dog can have properties such as name, breed, and age. It can also have methods that represent actions the dog can perform, such as bark, eat, and sleep.

```
class Dog {
  constructor(name, breed) {
        this.name = name;
        this.breed = breed;
  }

  // Method for the Dog class
  bark() {
        console.log(this.name + ' says Woof! Woof!');
  }
}
```

In the Dog class, you have defined a method called bark, which allows a dog object to "bark." The this keyword refers to the instance of the class. So, this.name refers to the name of the specific barking dog.

Now, create an object of the Dog class and call the bark method:

```
const myDog = new Dog("Buddy", "Chocolate Labrador");
myDog.bark(); // logs 'Buddy says Woof! Woof!' to the console
```

Inheritance lets you create a subclass with all the properties and methods of the parent class, along with its own unique properties and methods.

Create a ServiceDog subclass that inherits from the Dog class:

```
class ServiceDog extends Dog {
  constructor(name, breed, job) {
        super(name, breed);
        this.job = job;
  }

  // Method for the ServiceDog class
  performJob() {
        console.log(this.name + ' is performing his job as a ' + this.job + '.');
  }
}
```

The ServiceDog class has a performJob method that allows a service dog to perform its job.

Now, create an object of the ServiceDog class and call both the bark and performJob methods:

```
const myServiceDog = new ServiceDog("Max", "Golden Retriever", "guide dog");
myServiceDog.bark(); // logs 'Max says Woof! Woof!' to the console
myServiceDog.performJob(); // logs 'Max is performing his job as a guide dog.' to the console
```

In the example above, although we define bark in the Dog class, myServiceDog can still call the bark method because ServiceDog inherits from Dog. A powerful feature of object-oriented programming is its ability to reuse and build upon existing code.

Functions

Functions are reusable pieces of code that perform a specific task. You can call them multiple times with different inputs called arguments.

Defining Functions

There are several ways to define functions in JavaScript: function declarations, function expressions, and arrow functions. Each has its own syntax and characteristics.

To declare a function, use the function keyword, followed by the function name, parentheses containing any parameters, and curly braces containing the function body:

```
function add(a, b) {
  return a + b;
}
```

Function expressions are similar to function declarations, but you assign them to a variable. They can be anonymous (no name) or named:

```
const multiply = function(a, b) {
  return a * b;
};
```

Arrow functions are a shorter way to define functions. They use the “fat arrow” syntax (=>) and don’t require the function keyword. Arrow functions can have implicit returns for single-expression functions:

```
const subtract = (a, b) => a - b;
```

Functions can return multiple values using an object or an array, allowing you to package several variables into a single return statement. This functionality is helpful in specific scenarios, such as the following example:

```
const calculateAreaAndPerimeter = function(length, width) {
  let area = length * width;
  let perimeter = 2 * (length + width);

  return { area: area, perimeter: perimeter };
};

let rectangle = calculateAreaAndPerimeter(4, 6);
console.log("Area: " + rectangle.area); // "Area: 24"
console.log("Perimeter: " + rectangle.perimeter); // "Perimeter: 20"
```

In this code, the calculateAreaAndPerimeter function takes two parameters, length and width, to calculate a rectangle's area and perimeter. The function returns an object containing both the area and perimeter. The rectangle variable then stores the returned object, so dot notation can access the values. This is a robust technique that allows a function to produce more than one result, which you can use elsewhere in your code.

Scope

In JavaScript, scope refers to the area of the code where a variable is accessible. Understanding scope is essential for managing variables and preventing errors. There are four types of scope in JavaScript: global scope, function scope, block scope, and lexical scope.

Global Scope

Variables declared outside any function or block have global scope. Any part of the code can access them, leading to unintended consequences and making debugging difficult:

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

function exampleFunction() {
  console.log(globalVar);
}
exampleFunction(); // Logs "I'm global!"
```

Using too many global variables can pollute the global namespace, causing naming conflicts, unintended variable changes, and hard-to-debug code.

To avoid global scope issues, declare variables within functions or blocks whenever possible. Limit your use of global variables and use more specific scopes when needed.

Function Scope

Variables declared within a function are only accessible within that function. This approach is known as function scope:

```
function exampleFunction() {
  let functionVar = “I’m local to this function!”;
  console.log(functionVar);
}

exampleFunction(); // Logs “I’m local to this function!”
console.log(functionVar); // Error: functionVar is not defined
```

Block Scope

Variables declared with the let and const keywords have block scope, so they’re only accessible within the block (the code between curly braces) where they’re declared:

```
if (true) {
  let blockVar = “I’m local to this block!”;
  console.log(blockVar);
}

console.log(blockVar); // Error: blockVar is not defined
```

Block scope is generally preferable to function scope because it offers better control over where variables are accessible.

Lexical Scope

Lexical scope means that variables are accessible within the scope they're declared in and any nested scopes (functions or blocks inside the declaring scope.) Understanding lexical scope is vital to manage variables effectively, avoid code errors, and write clean, maintainable code.

```
function outerFunction() {
  let outerVar = "I'm in the outer function!";

  function innerFunction() {
    console.log(outerVar); // Has access to outerVar
  }

  innerFunction(); // Logs "I'm in the outer function!"
}

outerFunction();
```

In this example, the code declares outerVar inside outerFunction, and outerVar is accessible within outerFunction and the nested innerFunction.

Data Structures

There are two common data structures in JavaScript: Arrays and Maps/Sets. This section shows you how to create and manipulate these data structures and explores some best practices for working with them.

Arrays

Arrays are a fundamental data structure in JavaScript that store multiple values in a single variable. Arrays are ordered so that elements have a specific sequence, and they're indexed, allowing you to access elements by their position in the array.

Creating an Array

Use the following code to create an array:

```
let fruits = [‘apple’, ‘banana’, ‘cherry’];
```

In this example, an array called fruits contains three elements: ‘apple,’ ‘banana,’ and ‘cherry.’

Adding Elements to an Array

Here’s how to add elements to an array:

```
fruits.push(‘orange’);
```

This code uses the push method to add ‘orange’ to the end of the fruits array. The fruits array now contains four elements.

Removing Elements from an Array

Use the following code to remove elements from an array:

```
fruits.pop();
```

The pop method removes the last element from the array. In this case, it removes ‘orange’ from the fruits array.

The pop method is just one way of removing elements from an array. You can also delete an element from the array based on its index number:

```
delete fruits[0]; // Removes ‘apple’ by deleting index 0
```

Or, shift the contents of the array, removing the first element:

```
fruits.shift(); // Removes ‘apple’ by shifting all elements left
```

You can dissect your array to remove specific elements using the slice method:

```
fruits.slice(1); // Returns ['banana', 'orange'] - removes 'apple'
```

Finally, you can redefine your array, removing any unwanted elements from the new definition:

```
fruits = ['banana', 'orange']; // Redefines fruits, effectively removing 'apple'
```

While this may seem like a lot of different ways to accomplish the same goal, there are a few important distinctions between the various methods:

- pop and shift mutate the original array by removing elements.
- slice returns a new array with the elements removed but leaves the original unmodified.
- delete and redefinition don't change the array length. The deleted spots become empty slots.

Accessing Elements in an Array

Use square brackets to access an array element by its index (position). Array indices are zero-based, so fruits[1] refers to the second element in the array, 'banana' below:

```
console.log(fruits[1]); // Outputs "banana"
```

Maps and Sets

The JavaScript standard ECMAScript 6 (ES6) introduced two data structures, Maps and Sets, that offer new ways to store and manage data.

Maps are similar to objects but allow you to use any data type as a key, whereas objects only support strings and symbols as keys. Maps are helpful when you need to associate values with keys that aren't strings or when you need to maintain the insertion order of key-value pairs.

Creating and Adding Elements to a Map

Use the following code to create and add elements to a map:

```
let myMap = new Map();
myMap.set('name', 'John Doe');
myMap.set('age', 30);
```

Here, you created a new Map called myMap and added two key-value pairs using the set method.

Accessing Values in a Map

To access a Map's value, use the get method with the key as the argument:

```
console.log(myMap.get('name')); // Outputs "John Doe"
console.log(myMap.get('age')); // Outputs "30"
```

A set is a collection of unique values, so each value can only appear once in a Set. Sets help store a collection of distinct elements and ensure no duplicates.

Creating a Set and Adding Elements

The following example creates a new Set called mySet and adds three unique elements using the add method:

```
let mySet = new Set();
mySet.add(1);
mySet.add(2);
mySet.add(3);
```

Removing Elements from a Set

To remove an element from a Set, use the delete method with the value as the argument:

```
mySet.delete(2);
```

JavaScript Libraries and Framework Overview

The Role of Libraries and Frameworks

Libraries and frameworks are collections of pre-written code that developers can use to streamline their work, reduce development time, and improve code maintainability. These libraries and frameworks offer useful functions and solve typical problems, enabling you to focus on writing application-specific code.

Advantages of Using Libraries and Frameworks

Using libraries and frameworks in JavaScript development offers several benefits:

- **Reduced development time:** Libraries and frameworks often have built-in solutions for everyday tasks, eliminating the need to write code from scratch.
- **Improved code maintainability:** By following standardized conventions and best practices, libraries and frameworks can help you write clean, modular, and maintainable code.
- **Access to a community of developers and resources:** Popular libraries and frameworks often have large communities of developers who contribute to their development, provide support, and share resources.

Popular JavaScript Libraries and Frameworks

Listed below are some popular JavaScript libraries and frameworks:

- **jQuery:** This popular, lightweight library simplifies tasks like HTML document traversal (finding documents), manipulation, event handling, and animation, making it easier for you to create dynamic web applications.
- **React:** Facebook developed this JavaScript library for building user interfaces. React focuses on creating reusable user interface (UI) components and efficiently updating the UI when the underlying data changes. Developers typically use React to build single-page applications.
- **Angular:** Google developed this robust, feature-rich framework for building dynamic web applications. Angular uses a declarative approach to building UI components and provides comprehensive tools and features for creating complex applications.
- **Node.js:** This runtime environment lets you run JavaScript on the server side. Node.js builds on the V8 JavaScript engine and uses an event-driven, non-blocking I/O model, making it well-suited for scalable and high-performance applications.

Next Steps

This guide covered fundamental JavaScript concepts, including data types, variables, operators, control flow, object-oriented programming, functions, scope, and data structures, and provided a high-level overview of popular libraries and frameworks. These foundational topics offer you a solid basis for understanding JavaScript and its role in web development.

As a beginner, it's essential to continue learning and practicing JavaScript. This robust programming language is an invaluable skill for web development and can open many career opportunities. We encourage you to explore additional resources, tutorials, and courses to deepen your understanding of JavaScript. Remember, practice makes perfect, and the more you work with JavaScript, the more comfortable and proficient you'll become.

Ready to put your new JavaScript skills to the test? Explore some of HubSpot's other JavaScript resources, like **How to use JavaScript frameworks and libraries on HubSpot** and create your own website to showcase your new talents.