# Chapter 3.1: JavaScript Basics

Welcome to this beginner-friendly JavaScript tutorial! JavaScript (often called JS) is a fun and powerful language that makes websites interactive, like adding buttons that do things when clicked. This guide starts with the basics and builds up to more exciting concepts, combining related topics (like variables and data types) to make learning smooth and clear. Each section has explanations and code you can try in your browser's console or a simple HTML file.

To run JavaScript code, you can:

- Open your browser's developer tools (press F12 or Ctrl+Shift+I, then click the Console tab).
- Use an online editor like JSFiddle or CodePen.
- Create a local HTML file with a `<script>` tag.

Let’s get started and have fun coding!

## 1. Including JavaScript in Your Web Page

Before we write any JavaScript, we need to know how to add it to a web page. There are three ways to do this, and each has its own use. Let’s explore them!

### Inline JavaScript

You can add JavaScript right inside HTML tags using event attributes (we’ll learn more about events later).

```html
<button onclick="alert('Hello!')">Click me</button>
```

**Explanation**: This code creates a button that shows a popup saying "Hello!" when you click it. The `onclick` part is like telling the button, "Hey, when someone clicks you, run this `alert` code!" Try this in an HTML file: save it, open it in a browser, and click the button to see the popup. It’s a quick way to add small bits of JS, but we’ll see better ways for bigger projects.

### Internal JavaScript

You can put JavaScript inside a `<script>` tag in your HTML file, usually in the `<head>` or at the end of the `<body>`.

```html
<!DOCTYPE html>
<html>
<head>
    <script>
        alert('Hello from internal JS!');
    </script>
</head>
<body></body>
</html>
```

**Explanation**: This code puts JavaScript inside a `<script>` tag in the `<head>` of an HTML page. When you load the page, a popup says "Hello from internal JS!" Try copying this into a file (e.g., `test.html`), open it in a browser, and see the alert. This method is great for small scripts because it keeps everything in one file, but it can get messy for lots of code.

### External JavaScript

For bigger projects, you can put your JavaScript in a separate `.js` file and link it to your HTML.

```
<!DOCTYPE html>
<html>
<head>
    <script src="script.js"></script>  <!-- script.js contains your code --
>
</head>
<body></body>
</html>
```

**Explanation**: This HTML links to a file called `script.js` using the `<script src="script.js">` tag. The browser will load and run the code in `script.js` when the page opens. Create a file named `script.js` in the same folder as your HTML, add some code (like the next example), and try it out! This keeps your HTML clean and makes your JavaScript reusable.

In `script.js`:

```
alert('Hello from external JS!');
```

**Explanation**: This code goes in `script.js` and shows a popup saying "Hello from external JS!" when the HTML page loads it. Save this in `script.js`, make sure your HTML file links to it, and open the HTML in a browser to see the alert. Using external files is awesome for organizing bigger projects, and you'll use this a lot as you learn more.

**Tip:** External JS is great for reusability and keeps your HTML clean.

## 2. Variables and Data Types

Variables are like boxes where you store information, like numbers or text. JavaScript has three ways to create them: `var`, `let`, and `const`.

- `var`: The old way to make variables. It's flexible but can cause confusion because it's function-scoped (we'll talk about scope later).
- `let`: The modern way. It's block-scoped (only works inside `{}` where it's created) and you can change its value but not redeclare it in the same scope.
- `const`: Like `let`, but you can't change its value after setting it (though you can tweak objects or arrays).

JavaScript figures out the type of data (like number or text) automatically, so you don't need to tell it upfront.

### Common Data Types

- **Number**: Whole numbers (like `5`) or decimals (like `3.14`).
- **String**: Text, like `'Hello'` or `"World"`, in single or double quotes.
- **Boolean**: Just `true` or `false`.
- **Null**: Means "nothing" on purpose, like `null`.
- **Undefined**: When a variable exists but has no value yet.

### Examples

```
var oldWay = 10;  // Can be redeclared
let age = 25;     // Block-scoped, reassignable
const name = 'Alice';  // Can't be reassigned

console.log(typeof age);  // Outputs: "number"
console.log(typeof name); // Outputs: "string"

let isAdult = true;       // Boolean
let nothing = null;       // Null
let notDefined;           // Undefined
console.log(notDefined);  // Outputs: undefined
```

**Explanation**: This code shows how to create variables with `var`, `let`, and `const`. We store a number (10), another number (25), and a string ('Alice'). The `typeof` command tells us the data type: `age` is a "number", and `name` is a "string". We also create a boolean (`true`), a null value, and a variable with no value (`undefined`). Open your browser's console (F12), type this code, and see what it logs. Try changing `age = 30;`—it works! But try `name = 'Bob';`—it'll give an error because `const` can't change. Play around to see how each type behaves!

**Try it:** Change `age = 30;` and see it works. But `name = 'Bob';` will throw an error.

## 3. Output Methods: console.log, document.write, alert, prompt, confirm

JavaScript lets you show messages or get input from users in different ways. Here are the main ones:

- `console.log()`: Shows messages in the browser's console (perfect for testing).
- `document.write()`: Writes text directly onto the web page (be careful—it can erase other content).
- `alert()`: Pops up a message box.
- `prompt()`: Asks the user to type something in a popup.
- `confirm()`: Shows a yes/no popup and gives you `true` or `false`.

### Examples

```
console.log('This goes to the console');
```

```
document.write('This writes to the page');

alert('Popup alert!');

let userInput = prompt('Enter your name:');
console.log('Hello, ' + userInput);

let ok = confirm('Are you sure?');
if (ok) {
    console.log('User said yes!');
}
```

**Explanation**: This code is your toolkit for talking to users! First, `console.log` sends a message to the console—open your browser's console (F12) and try it to see the message. `document.write` adds text to the page, but use it carefully in an HTML file. The `alert` shows a popup—try it and see! `prompt` asks for the user's name and stores it in `userInput`, then we log a greeting with it. Finally, `confirm` asks for a yes/no answer and logs if the user clicks "OK". Put this in an HTML `<script>` tag or console, run it, and try typing different names or clicking "OK"/"Cancel" to see what happens.

**Tip:** Use `console.log` for testing; avoid `document.write` in real projects as it can mess up the page.

## 4. Strings and Concatenation

Strings are bits of text, like names or messages, and you can join them together using `+` (called concatenation).

### Examples

```
let first = 'Hello';
let second = 'World';
let combined = first + ' ' + second;  // 'Hello World'
console.log(combined);

let num = 42;
let mixed = 'The answer is ' + num;  // 'The answer is 42' (num is
converted to string)
```

**Explanation**: Here, we're playing with text! We create two strings, `first` ("Hello") and `second` ("World"), then use `+` to join them with a space to make "Hello World" and log it. Next, we mix a string with a number (42), and JavaScript automatically turns the number into a string to create "The answer is 42". Try this in your console: change the strings or number and see what you get. Want a cooler way? Try template literals like `` `The answer is ${num}` ``—we'll mention that next!

**Modern Tip:** Use template literals for easier concatenation: `` `The answer is ${num}` ``.

## 5. Conditions: if-else and switch

Conditions let your code make decisions, like choosing what to do based on a value.

### if-else Statement

Checks if something is true and runs different code based on that.

```
let age = 18;

if (age >= 18) {
    console.log('Adult');
} else if (age >= 13) {
    console.log('Teen');
} else {
    console.log('Child');
}
```

**Explanation**: This code decides what to call someone based on their `age`. If `age` is 18 or more, it logs "Adult". If it's 13–17, it logs "Teen". Otherwise, it logs "Child". Try this in your console: change `age` to 15 or 10 and see what happens. It's like telling your code, "If this is true, do this; if not, try this other thing!" Run it and experiment with different ages to understand how it picks.

### switch Statement

Great when you have several specific values to check.

```
let day = 'Monday';

switch (day) {
    case 'Monday':
        console.log('Start of the week');
        break;
    case 'Friday':
        console.log('End of the week');
        break;
    default:
        console.log('Midweek');
}
```

**Explanation**: This code checks the value of `day`. If it's "Monday", it logs "Start of the week". If it's "Friday", it logs "End of the week". For any other day, it logs "Midweek". The `break` stops the code from running the next case. Try this in your console: change `day` to "Friday" or "Sunday" and see what it logs. It's like a menu where you pick one option and get a specific result!

## 6. Loops: for, while, forEach

Loops let you repeat code, like printing numbers multiple times.

### for Loop

Best when you know how many times to repeat.

```
for (let i = 0; i < 5; i++) {
    console.log(i);  // 0,1,2,3,4
}
```

**Explanation**: This `for` loop counts from 0 to 4 and logs each number. The `let i = 0` starts the counter, `i < 5` says "keep going until i is 5", and `i++` adds 1 each time. Try this in your console to see it print 0, 1, 2, 3, 4. Change `i < 5` to `i < 10` and see what happens—it's like telling your code to count higher! Play with it to get the hang of loops.

### while Loop

Repeats as long as a condition is true.

```
let count = 0;
while (count < 3) {
    console.log(count);  // 0,1,2
    count++;
}
```

**Explanation**: This `while` loop keeps running as long as `count` is less than 3, logging 0, 1, and 2. The `count++` adds 1 each time to avoid an endless loop. Try this in your console and change `count < 3` to `count < 5` to see more numbers. It's like saying, "Keep doing this until I tell you to stop!" Be careful—if you forget `count++`, it'll loop forever!

### forEach (for Arrays)

Loops over items in an array (we'll cover arrays next).

```
let fruits = ['apple', 'banana', 'cherry'];
fruits.forEach(function(fruit) {
    console.log(fruit);
});
```

**Explanation**: This code takes an array of fruits and uses `forEach` to log each one: "apple", "banana", "cherry". The `function(fruit)` runs for each item in the array. Try this in your console: add a new fruit like `'orange'` to the array and see it print. It's a super easy way to loop through lists without counting manually!

## 7. Arrays

Arrays are like lists that hold multiple values, like numbers or text. JavaScript arrays can even mix different types!

### Syntax and Examples

```
let numbers = [1, 2, 3];  // Array syntax
let mixed = [1, 'two', true, null];  // Mixed types

console.log(numbers[0]);  // 1 (zero-indexed)

numbers.push(4);  // Add to end: [1,2,3,4]
numbers.pop();    // Remove last: [1,2,3]

for (let num of numbers) {  // Loop with for-of
    console.log(num);
}
```

**Explanation**: This code creates two arrays: `numbers` with 1, 2, 3, and `mixed` with different types. We log the first number (1) using `numbers[0]` (arrays start at 0). `push(4)` adds 4 to the end, and `pop()` removes it. The `for...of` loop prints each number in `numbers`. Try this in your console: add a number with `push`, remove one with `pop`, or change the loop to print `mixed`. Arrays are like your shopping list—you can add, remove, or check items easily!

## 8. Objects: Creation, Key-Value Pairs, and JSON

Objects are like labeled storage boxes, holding pairs of keys (names) and values (data).

### Object Creation

```
let person = {
    name: 'Alice',     // Key-value pair
    age: 25,
    isAdult: true
};

console.log(person.name);  // 'Alice'
person.age = 26;           // Modify
person.gpa = 3;            // add new key-value pair to existing object.
```

**Explanation**: This code creates a `person` object with keys (`name`, `age`, `isAdult`) and values. We log the `name` ("Alice") using dot notation (`person.name`). Then we change `age` to 26 and add a new `gpa` key with value 3. Try this in your console: log `person.isAdult`, change `gpa`, or add a new key like `city: 'New York'`. Think of objects like a student ID card with labeled info you can update!

### Objects with Functions and 'this'

You can add functions to objects, and `this` refers to the object itself.

```
let car = {
    brand: 'Tesla',
    speed: 0,
    accelerate: function() {
        this.speed += 10;
        console.log('Speed: ' + this.speed);
    }
};

car.accelerate();  // Speed: 10
```

**Explanation**: This creates a `car` object with a brand, speed, and an `accelerate` function. When we call `car.accelerate()`, it adds 10 to `speed` (using `this.speed`) and logs the new speed (10). Try this in your console: call `car.accelerate()` again to see the speed go to 20. The `this` keyword is like saying, "Hey, use the speed from this car!" Play with changing the increment or logging `car.brand`.

### JSON Format

JSON (JavaScript Object Notation) is a way to store or send data as a string.

```
let jsonString = '{"name": "Bob", "age": 30}';  // JSON string

let obj = JSON.parse(jsonString);  // Parse to object
console.log(obj.name);  // 'Bob'

let backToJson = JSON.stringify(obj);  // Convert back to JSON
console.log(backToJson);  // '{"name":"Bob","age":30}'
```

**Explanation**: This code starts with a JSON string (like a text version of an object). `JSON.parse` turns it into a real object so we can log `name` ("Bob"). Then `JSON.stringify` turns it back into a string. Try this in your console: change the JSON string to include a new key, like `"city": "Paris"`, and parse it to see the new object. JSON is like mailing a package—you pack it as a string to send, then unpack it to use!

## 9. Functions: Basics, Parameters, Defaults, Arguments, Callbacks

Functions are reusable chunks of code, like recipes you can follow again and again.

### Basic Function

```
function greet() {
    console.log('Hello!');
}
greet();
```

**Explanation**: This defines a simple `greet` function that logs "Hello!" when called. We run it with `greet()`. Try this in your console: call `greet()` a few times or change the message to "Hi there!". Functions are like shortcuts—you write the code once and use it whenever you want.

### Parameters and Defaults

```
function add(a, b = 0) {  // Default value for b
    return a + b;
}
console.log(add(5, 3));  // 8
console.log(add(5));     // 5 (b defaults to 0)
```

**Explanation**: This `add` function takes two numbers, `a` and `b`, and returns their sum. If you don't give `b`, it uses 0 as a default. We log `add(5, 3)` (8) and `add(5)` (5). Try this in your console: test `add(10, 20)` or `add(7)` to see how defaults work. It's like a calculator that assumes 0 if you only give one number!

### Multiple Parameters with ...args (Rest Parameters)

```
function sum(...args) {  // Collects all args into an array
    let total = 0;
    for (let num of args) {
        total += num;
    }
    return total;
}
console.log(sum(1, 2, 3, 4));  // 10
```

**Explanation**: The `sum` function uses `...args` to grab any number of inputs as an array. It loops through them, adds them up, and logs the total (10 for 1+2+3+4). Try this in your console: call `sum(5, 5)` or `sum(1, 2, 3, 4, 5)` to see different totals. It's like a magic bag that adds up whatever numbers you throw in!

### Callback Functions

A function passed to another function to run later.

```
function process(callback) {
    console.log('Processing...');
    callback();
}

function done() {
    console.log('Done!');
}

process(done);  // Processing... Done!
```

**Explanation**: Here, `process` is a function that logs "Processing..." and then runs a callback function. We define `done` to log "Done!" and pass it to `process`. When you run `process(done)`, it logs both messages. Try this in your console: create a new callback function, like `function sayHi() { console.log('Hi!'); }`, and pass it to `process`. It's like telling a friend, "Do this, then do that extra thing I gave you!"

## 10. Function Constructors (Like Classes)

Before modern JavaScript classes, we used functions to create objects, like blueprints for building things.

```
function Person(name, age) {  // Constructor function
    this.name = name;
    this.age = age;
    this.greet = function() {
        console.log('Hi, I\'m ' + this.name);
    };
}

let alice = new Person('Alice', 25);  // Create object
alice.greet();  // Hi, I'm Alice
```

**Explanation**: This `Person` function is a constructor that makes objects with a `name`, `age`, and `greet` function. Using `new Person('Alice', 25)` creates an object for Alice, and `alice.greet()` logs her greeting. Try this in your console: make a new person, like `let bob = new Person('Bob', 30);`, and call `bob.greet()`. It's like a factory that builds custom objects for you!

**Note:** Modern JS uses `class` syntax, but this is the classic way.

## 11. Scope: Global vs Local

Scope is about where your variables can be used.

- **Global**: Variables outside functions, available everywhere.
- **Local**: Variables inside functions or `{}` blocks, only usable there.

### Examples

```
let globalVar = 'Global';  // Global scope

function test() {
    let localVar = 'Local';  // Local to function
    console.log(globalVar);  // Accessible
}

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

**Explanation**: This code shows a `globalVar` that's available everywhere and a `localVar` inside the `test` function. When we call `test()`, it logs `globalVar` because it's global, but trying to log `localVar` outside the function causes an error. Try this in your console: run `test()`, then try logging `localVar`. Now add a new `console.log(localVar)` inside the function to see it work. Scope is like a fence—local variables stay inside their function!

**Tip:** Use `let`/`const` to avoid global pollution.

## 12. Events: Introduction to onclick and onmouseover

Events let your page respond to user actions, like clicks or hovers.

- `onclick`: Runs code when something is clicked.
- `onmouseover`: Runs code when the mouse hovers over something.

### Examples (in HTML)

```
<button onclick="alert('Clicked!')">Click</button>
<div onmouseover="this.style.backgroundColor='red'">Hover me</div>
```

**Explanation**: This HTML creates a button that shows a "Clicked!" popup when you click it and a `<div>` that turns red when you hover over it. Save this in an HTML file, open it in a browser, and try clicking and hovering. The `onclick` and `onmouseover` are like telling the page, "When this happens, do that!" It's a great way to make your page interactive.

### In JS (Better for Separation)

```
document.getElementById('myButton').onclick = function() {
    alert('Clicked!');
};
```

**Explanation**: This JavaScript adds a click event to a button with `id="myButton"` (you need `<button id="myButton">Click</button>` in your HTML). When clicked, it shows a "Clicked!" popup. Try this: create an HTML file with the button, add this code in a `<script>` tag, and click it. Using JavaScript like this keeps your HTML cleaner and is better for bigger projects. Try changing the alert message to see what happens!

Assume HTML: `<button id="myButton">Click</button>`

## Cheat Sheet

| Topic | Description | Key Points/Examples |
|---|---|---|
| **Including JS** | Methods to add JavaScript to HTML for execution. | Inline: `onclick="code"`<br>Internal: `<script>code</script>`<br>External: `<script src="file.js"></script>` |
| **Variables** | Ways to declare and store data in JavaScript. | `var x = 1;` (old, function-scoped)<br>`let y = 2;` (block-scoped, reassignable)<br>`const z = 3;` (constant) |

| Topic | Description | Key Points/Examples |
|---|---|---|
| **Data Types** | Basic types of data JavaScript handles. | Number: `42`<br>String: `'text'`<br>Boolean: `true/false`<br>Null: `null`<br>Undefined: (unassigned var) |
| **Output** | Methods to display data or interact with users. | `console.log('msg')`<br>`alert('popup')`<br>`prompt('input?')`<br>`confirm('yes/no?')`<br>`document.write('to page')` |
| **Concatenation** | Joining strings or combining with other types. | `'A' + 'B' = 'AB'`<br>Template: `` `A ${var}` `` |
| **Conditions** | Control program flow based on conditions. | `if (cond) { } else { }`<br>`switch (val) { case 'x': ... break; default: ... }` |
| **Loops** | Repeat code execution for iteration. | `for (i=0; i<5; i++) { }`<br>`while (cond) { }`<br>`array.forEach(func(item) { })` |
| **Arrays** | Store multiple values in a single variable. | `let arr = [1, 'two', true];`<br>`arr.push(3); arr.pop(); arr[0]` |
| **Objects** | Store key-value pairs, like dictionaries. | `let obj = {key: 'value'}; obj.key;`<br>With method: `obj.method = function() { this.key }` |
| **JSON** | Format for data exchange using strings. | `JSON.parse('{"k":"v"}')` → object<br>`JSON.stringify(obj)` → string |
| **Functions** | Reusable code blocks, with parameters and callbacks. | `function name(params=def) { return; }`<br>Rest: `...args`<br>Callback: Pass func as param |
| **Constructor** | Functions to create objects (pre-ES6 classes). | `function Class() { this.prop = val; }`<br>`new Class()` |
| **Scope** | Rules for variable accessibility. | Global: Outside funcs<br>Local: Inside funcs/blocks |
| **Events** | Triggers for user interactions like clicks. | `onclick="code"`<br>`onmouseover="code"` |