



**JR. WEB DEVELOPER  
CERTIFICATION KIT**

<b>Level 1-1—Unlocking the Power of Code To Change Your World</b>	<b>6</b>
Everyone Can Code, Including You!	7
No One Gets Left Behind.	7
Learn to Think Like A Coder.	8
Javascript is everywhere.	9
The Back-end	11
Developer Tools: IDEs and editors	12
Developer Tools: Developer Console	12
Variables	13
Basic Data Types	13
Reference Data Types	14
Let's Review!	15
Preview Next Lesson	16
Your Turn Now.	16
Homework Assignment—Building Confidence Through Practice	16
Homework #1	16
Homework #2	17
Homework #3	18
You did it!	18
<b>Level 1-2: Building Your Programming Fundamentals with Tic-Tac-Toe</b>	<b>20</b>
Art vs. Science	21
The Science of Coding	21
The Art of Coding	21
Bringing Art and Science Together	22
Why This Matters for Developers	22
More about variables	23
Declaring Variables with let	24
Initializing Variables with let	24
Declaring and Initializing in One Step	25
Understanding const	25
Use Cases	26
Best Practices	26
You did it!	26
Introduction to var	27
Understanding Where Your Variables Can Be Used (Introducing "Scope")	27
Redeclaration with var	28

Understanding Redeclaration with let	28
Understanding Hoisting (Moving Variables To The Top of the Scope)	29
Hoisting with var vs. let and const	30
Why This Matters	31
Understanding Functions in JavaScript with Simple Examples	31
Example #1: A Function to Make a Sandwich	31
Defining Functions	31
Using the makeSandwich Function	33
Example 2: A Function to Calculate the Area of a Rectangle	33
Defining a calculateArea Function	33
Using the Function	34
The console Object and the log Method	34
The console Object	34
The log Method	34
How console.log Relates to Defining Functions	35
Why This Matters	36
Introducing Hoisting and Function Scope with Simple Functions	36
Do You Remember What Hoisting Is?	36
A Function to Make Coffee	37
Example: A Function to Calculate Discount	37
The var Quirk	38
Arrays and Indexed Collections	39
Loops	40
Using a for Loop	41
Understanding the for Loop and Array Length	41
How the Loop Works	42
Introduction to Comparison	43
The Less Than Operator in Action	43
What's Going On Here?	43
Connecting the Dots	44
From < to Equality Comparisons	44
Loose Equality (==)	44
What's Happening Here?	45
Strict Equality (===)	45
What's Happening Here?	45
Why It Matters	45
Practical Example: Checking User Input	46
Bringing It All Together: The .every Array Method	47
What is the .every Method?	47
Example: Checking if All Numbers are Positive	47
How It Works	48
Why Use .every?	48
Practical Example: Validating User Input	48

How It Works	49
Connecting the Dots	49
You've Come a Long Way: Let's Make A Tic-Tac-Toe Game	50
How Other Developers Did It	50
Example 1: A Detailed Solution on GeeksforGeeks	50
Example 2: A Simpler Approach on Medium	51
What These Solutions Have in Common	52
Unlocking the if Statement Superpower	52
Understanding if Statements	52
The Basics of if Statements	52
Example: Checking a Number	52
Why if Statements Are So Powerful	53
Connecting The Dots	53
You're Ready!	54
Design Thinking	54
We Can Code This!	54
Representing Winning Patterns	55
Example: Vertical Win in the Middle	55
Example: Diagonal Win from Upper-Right to Lower-Left	56
Putting It All Together	57
Remember the for loop?	59
Let's Walk Through the Code Together, Line by Line	61
1. Function Definition	61
2. Define Winning Combinations	61
3. Loop Through Each Winning Combination	62
4. Accessing the Current Combination	62
5. Checking If the Player Occupies All Three Spots	63
6. Return true if a Winning Combination is Found	63
7. End of the Loop and Return `false`	63
8. Example Board State and Function Call	64
Wow! You've Come So Far	64
What You've Accomplished	64
Keep Going	65
Homework: Grow Your Superpowers	65
1. Create Your Own Tic-Tac-Toe Board	65
2. Check for a Winner	66
3. Experiment with New Winning Combinations	66
4. Reflection	66
Extra Challenge: Make It Interactive	67
What's Next?	67

# Level 1-1—Unlocking the Power of Code To Change Your World

Welcome developers to the Code Accelerator **Junior Web Developer Certification** kit!

Code Accelerator focuses on making beginners comfortable and confident with computer science fundamentals to help you build up a strong understanding of how web applications function on both the inside and outside, so that you can:

- Improve your skills outside of the pressure of a work environment
- Enhance your career opportunities through your improved skills
- Reduce the frustration and anxiety associated with gaps in understanding

You're in the right place if you've ever felt like the world of coding was a closed door because of a lack of experience. We're going to help you open that door!

# Everyone Can Code, Including You!

Code Accelerator knows that everyone, from passionate beginners to seasoned thinkers, can unlock this door—with the right key.

And that key is understanding—really understanding the code you're looking at—and not just copying, pasting, and memorizing code.

Code connects to everything we do in the world.

Code touches almost every aspect of our modern lives—from the apps that wake us up, to the cars that drive us home.

And learning the fundamentals of coding is not just for web developers and computer scientists anymore; it's for anyone with the curiosity to learn.

That's why no matter who you are or where you've started, we ask you to build a foundation with us on the basics and fundamentals, as we work our way up step by step with other developers in the community.

## No One Gets Left Behind.

Code Accelerator expects you to collaborate with other developers to strengthen everyone's understanding.

No one is left behind.

Just like in the movie 'The Karate Kid' where simple, everyday actions build up confidence and power, Code Accelerator's step by step approach builds your coding journey one computer science concept at a time using ways of thinking about things that can be familiar to everyone.

In the Code Accelerator camp, understanding both the front and back end of applications makes you an active participant, a creator, and an innovator in the tech space whether you're a back-end web developer, UX designer, team lead, or startup founder.

You can unlock Code Accelerator certifications only when you're ready to be prepared to think, to solve problems, and to design solutions.

And each of us brings a unique perspective to coding. Code Accelerator is designed to start from whatever knowledge point you're at right now.

Hands-on projects, supportive mentors, and a community of peers, can help you find the confidence to not only understand coding challenges, but to excel at them.

## Learn to Think Like A Coder.

Code Accelerator doesn't just help you learn how to copy and paste code.

Code Accelerator helps you learn how to think like a confident coder.

As we move through today's Level 1-1 session, remember that every expert was once a beginner and every teacher was once a student.

Your journey might have started with a simple line of code, but where it goes is up to you—and that's limitless.

The Level 1-1 session is an overview of what we'll study more closely in the Level 1-2 learning session named "Building Your Programming Fundamentals with Tic-Tac-Toe", where we'll build or rebuild your fundamental understandings of common computer science concepts like variables, functions, and data types.

Code Accelerator certification kits are especially designed to teach you the things you need to learn in the order you need to learn them so that you grow into a more powerful understanding of web development and web applications.

Level 1-3's learning session will take a detour to cover HTML and CSS to create and style the web pages that will support our web applications.

Having spent the proper time to build up your understanding of the foundation of our web applications, Level 1-4 unlocks even more of your JavaScript superpowers, especially handling events and manipulating the DOM within your HTML structure.

By the end of Level 1-5 (“*Understanding Reactive Frameworks*”) & Level 1-6 (“*Understanding Websockets and Real-Time Connections*”) you'll be using the understanding that you've built up thus far to wrap your mind around how front-end and back-end tech stacks work together and are used everyday to fuel the apps that show up in our everyday lives.

There are even more cool things we can learn from there, but before we get ahead of ourselves, we're going to start today's learning session: “Unlocking the Power of Code To Change Your World”.

## Javascript is everywhere.

As of July 2024, ZipRecruiter lists the average “JavaScript Developer” salary as \$106,583.

This shows just how important Javascript skills are when it comes to building and maintaining anything that lives on the web.

Support from big companies like Facebook and Google has made JavaScript the most popular language on the Internet.

JavaScript is also the most widely used client-side coding language.

So, what exactly is a client-side code?

Client-side code is just the code that is stored and running right now on your home computer or in your hand on your mobile device!

Javascript is everywhere in our everyday lives, and that's why every single internet browser on the planet has a built-in way to process JavaScript!



Browsers use special software called a **JavaScript engine** to read and use JavaScript code to power the websites that make our lives easier. There are different JavaScript engines for every browser; V8 for Chrome, Gecko for Firefox, and JavaScriptCore for Safari.



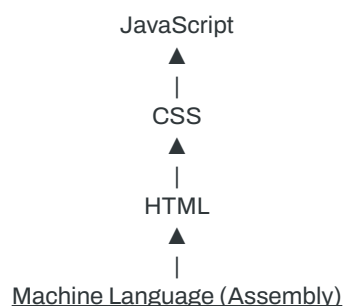
V8 (logo left), Gecko (logo center) and JavaScriptCore (right) are popular JavaScript engines.

## The Front-end

JavaScript, CSS (Cascading Style Sheets), and HTML (Hypertext Markup Language) files are really just text files inside of folders on a computer.

And those code files stack up layer by layer to display everything that you see in your web app. Everything that appears in a web browser is referred to as happening on the **client side**.

Web developers call this client-side mix of JavaScript, HTML, and CSS files the **front-end** of our web applications.



1. HTML rests at the bottom of the stack, creating the basic structure of our web application.

2. CSS sits on top of the HTML. It adds style to the HTML structure by adding colors, layouts, and animations that communicate importance to human users.
3. Javascript floats at the top of the stack, patiently waiting to instantly respond to user behavior and updated information that need to be sent to both the CSS and HTML layers.

This is how the front-end works for browsers on the Internet.

## The Back-end

From its birth in 1995 up until 2008, JavaScript remained trapped inside the client-side of the browser.

But in 2009, a software named Node.js was released that broke JavaScript out from the front-end!

Node.js is a C++ desktop application that allows us to use Javascript to quickly build web servers and work with databases.

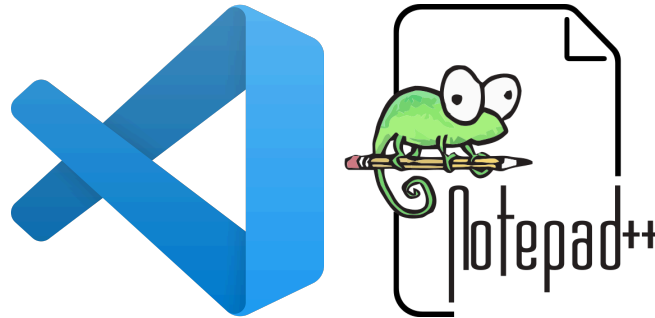
The term **server** describes computers that run software like Node.js to communicate with other computers and databases and send **.js**, **.html** and **.css** files to and from your browser.

The combination of files and folders that sit on servers and make them work is called the **back-end**, because these files quietly work in the *background* to make our web applications do all the wonderful things they do.

Now that developers like us have access to Node.js, we can use JavaScript to create both the front-end and the back-end for powerful applications that live on the World Wide Web.

# Developer Tools: IDEs and editors

Integrated development environments like **Visual Studio Code** are software applications that help programmers develop code more effectively and quickly.



Visual Studio Code (logo left) and Notepad++ (logo right) are important tools for any junior web developer.

Lightweight code editors like Notepad++ are simple text editors with less features than IDEs.

While IDEs are project-focused, lightweight editors are faster for single-file tasks.

The key is finding the right balance between power and simplicity for efficient coding.

## Developer Tools: Developer Console

The **developer console** is a super useful tool used inside your browser that lets you play around and see what happens when you change parts of websites.

You can open it by pressing **Ctrl+Shift+J** on Windows or **Cmd+Option+J** on Mac.

You'll be using this console to try out small bits of code, called **code snippets**. It's great because you can see your code working in real-time without affecting anything big—just like practicing in a sandbox!

# Variables

In Javascript, information is usually held in containers. These data containers are called **variables**.

The keywords **let** and **const** are part of the syntax that allow us to declare and initialize a variable.

**Keywords** are the words in your code that are part of the **syntax** of the programming language.

The term “syntax of a programming language” just means *the specific things you must type to correctly communicate* with the Javascript engine.

To **declare** (create) a variable:

```
JavaScript
// declare a variable named myBootcamp
let myBootcamp;
```

To **initialize** a variable you assign a **value**.

```
JavaScript
// initialize a variable named myBootcamp
let myBootcamp = "Code Accelerator";
```

# Basic Data Types

The seven basic (or **primitive**) types of data you can assign to variables are:

```
JavaScript
let myVariable;
myVariable = "Code Accelerator"; // String (text)
myVariable = 2024; // Number
myVariable = BigInt(2^53); // BigInt (for big numbers)
myVariable = true; // Boolean (true or false)
myVariable = Symbol("Code Accelerator"); // Symbol
```

```
myVariable = null; // null(empty value)
myVariable = ""; // Undefined (no value yet)
```

These primitive data types are used to create other data structures in JavaScript.

## Reference Data Types

JavaScript also has special data structures called **reference data types**, which include objects, arrays, and functions.

An **array** is a way of storing an organized list of items inside of one variable. Arrays help us deal with multiple pieces of data that belong to one larger container.

An **object** in JavaScript is just like any object in the real world. It has properties (attributes or qualities) and actions (**functions**).

```
JavaScript
// initialize an array
const myArray = ["Javascript", "HTML", "CSS"];

// initialize an object
const myEyes = { vision: "20/20" };

// declare a function
function lookAtSomething (subject) {
  return subject;
};
```

For example: my eyes (object) can look at something (function) with 20/20 vision (quality).

And just like in the real world, objects are *everywhere* in JavaScript, so if you've ever used JavaScript you've probably also used objects already.

When a function lives inside of an object, developers call it a **method**.

When a variable lives inside of an object, developers call it a **property**.

The properties and methods (variables and functions) that live in an object should be separated by commas.

```
JavaScript
// initialize an object
const myEyes = {
  vision: "20/20",
  lookAtSomething: function (subject) {
    return subject;
  }
};

// object property
console.log(myEyes.vision) // Output: "20/20"

// object method
console.log(myEyes.lookAtSomething("thing")) // Output: "thing"
```

## Let's Review!

As we wrap up today's session, let's quickly review the key concepts we covered:

1. **Introduction to JavaScript:** We discussed why JavaScript is a fundamental language in web development and how it's used both on the front-end (in the browser) and back-end (with Node.js).
2. **The Front-End Stack:** You learned about the layers of the front-end stack—HTML for structure, CSS for styling, and JavaScript for interactivity.
3. **Developer Tools:** We touched on the importance of tools like IDEs and code editors, as well as the developer console, which is essential

for testing and debugging your code.

4. **Variables and Data Types:** We explored the concept of variables as containers for data and discussed the basic (primitive) data types like strings, numbers, and booleans, as well as reference data types like arrays and objects.

This foundation sets you up perfectly for our next session, where we'll dive deeper into the programming fundamentals that will power your journey through the world of coding.

## Preview Next Lesson

In our next session, “Building Your Programming Fundamentals with Tic-Tac-Toe,” we'll explore essential concepts like functions, loops, and conditionals—tools that will allow you to start building your own web applications. We'll also start looking at how these fundamentals come together to solve real-world problems.

## Your Turn Now.

### Homework Assignment—Building Confidence Through Practice

In line with Code Accelerator's mission to build your confidence and understanding step by step, here's your first homework assignment:

#### Homework #1

##### **Object Creation:**

- Write a simple JavaScript object that represents something in your everyday life (e.g., your phone, a favorite book, or even your coffee mug). Your object should have at least two properties (like brand, size, or flavor) and one method (like `makeCall`, `readBook`, or `sipCoffee`).

JavaScript

// Example:

```
const coffeeMug = {  
  color: "blue",  
  size: "large",  
  sipCoffee: function() {  
    console.log("Sipping coffee...");  
  }  
};
```

```
console.log(coffeeMug.color); // Output: "blue"  
coffeeMug.sipCoffee(); // Output: "Sipping coffee..."
```

## Homework #2

### Data Type Exploration:

- Use the developer console to declare variables of different data types—string, number, boolean, etc.
- HINT: use `typeof` in front of a variable or piece of data.
- EXTRA CREDIT: Try adding two numbers or two strings.

JavaScript

// Example:

```
let age = 25;  
let isAdult = age > 18;  
console.log(isAdult); // Output: true  
console.log(typeof age); // Output: ??  
console.log(typeof isAdult); // Output: ??  
let greeting = "Hello";  
let to = "World";  
let combinedGreeting = greeting + " " + to; console.log(combinedGreeting); //  
Output: "Hello World"
```

## Homework #3

### Sharing and Feedback:



- Once you've completed the exercises, share your code in our online forum. Don't worry if you get stuck—post your questions, and either I or your peers will be there to help you out!

## You did it!

As we end today's session, I want to remind you that coding is more than just a technical skill—it's a way to change your world. Every line of code you write brings you closer to creating something meaningful, whether it's a simple web page or a complex application.

Remember, every expert coder started where you are right now—with curiosity and a willingness to learn. The challenges you face today will become the building blocks of your success tomorrow.

So keep pushing forward, keep experimenting, and most importantly, keep believing in your ability to learn and grow.

Your journey in coding is just beginning, and I'm excited to see where it takes you. The skills you're developing here can open doors to new opportunities, so take pride in each step you take. The world of code is vast and full of possibilities—let's unlock those possibilities together.



**JR. WEB DEVELOPER**  
**CERTIFICATION KIT**

# Level 1-2: Building Your Programming Fundamentals with Tic-Tac-Toe

In this lesson, we will use the Tic-Tac-Toe game to explore fundamental JavaScript concepts. Through this, we'll not only learn a little about how writing JavaScript games works, but also deepen our understanding of core programming principles.

By the end of this lesson, you'll have a better understanding of how to structure your code effectively while embracing the creativity that comes with coding.

# Art vs. Science

The debate of whether coding is an art or a science rages on, but the truth is, it's both. Coding, like science, requires logic, precision, and a strong understanding of fundamentals.

However, like art, coding also requires creativity, intuition, and personal expression. The beauty of coding lies in its ability to combine elements, allowing developers to solve problems in unique and innovative ways.

## The Science of Coding

- **Core Understanding:** Just as scientists rely on established theories and principles, coders rely on algorithms, data structures, and programming paradigms to create functional and efficient software.
- **Logic:** Coding requires a deep understanding of syntax, logic, and computational thinking. Much like in science, a small mistake in code can lead to significant errors, making attention to detail crucial.
- **Predictability:** When you write code, you're essentially giving a set of instructions to a computer which, when executed correctly, produces predictable outcomes—just like conducting a scientific experiment.

## The Art of Coding

- **Creative Problem-Solving:** While coding requires logical thinking, it also allows for a wide range of creative solutions. There's no one "correct" way to solve a problem, which means developers can approach challenges in a way that reflects their unique perspective.
- **Personal Expression:** Just like an artist leaves their mark on a canvas, a coder leaves their signature on their code. This could be through their choice of variable names, the structure of their functions, or the overall design of their software.

- **Different Approaches:** Two developers can tackle the same problem and come up with completely different solutions, both of which are valid. This diversity in approach is what makes coding so fun, powerful and flexible.

## Bringing Art and Science Together

- **Balancing Structure and Flexibility:** Good code often strikes a balance between the rigid structure needed for efficiency and the flexibility required for innovation. It's about knowing the rules (science) and knowing when and how to bend them (art).
- **Continuing to Improve:** Like an artist perfecting their work or a scientist proving their hypothesis, coding is an ongoing process. Developers write, test, and optimize their code continuously, blending analytical thinking with creative intuition.

## Why This Matters for Developers

- **Embrace Your Creativity:** As you learn to code, remember that there's room for your creativity and unique perspective. Don't be afraid to experiment and try new things, even if they don't work out the first time. Coding is as much about exploring possibilities as it is about following rules.
- **Learn the Fundamentals, But Make Them Your Own:** Understanding the core principles of coding is essential, but how you apply them is up to you. Your coding journey is a personal one, and the way you solve problems will evolve as you gain more experience.

As you progress in your coding journey, remember that your ability to think creatively and approach problems from different angles is just as valuable as your technical skills.

Coding is an art, and you are the artist—your creativity and unique perspective are your superpowers.

# More about variables

In Level 1-1, we discussed variables as containers for data and explored the seven basic data types in JavaScript: strings, numbers, booleans, null, undefined, symbols, and BigInt.

JavaScript

```
let myVariable;  
myVariable = "Code Accelerator"; // String (text)  
myVariable = 2024; // Number  
myVariable = BigInt(2^53); // BigInt (for big numbers)  
myVariable = true; // Boolean (true or false)  
myVariable = Symbol("Code Accelerator"); // Symbol  
myVariable = null; // null(empty value)  
myVariable = ""; // Undefined (no value yet)
```

In Level 1-2, understanding the concepts behind variables—especially those declared with `var`, `let`, and `const`—is going to be incredibly important as we move forward.

In the last lesson, we talked about how the keyword `let` can be used to declare or initialize variables.

JavaScript

```
// declare a variable named myBootcamp  
let myBootcamp;
```

JavaScript

```
// initialize a variable named myBootcamp  
  
let myBootcamp = "Code Accelerator";
```

In Level 1-2, we'll deepen our understanding of variables by focusing on two important aspects: declaring and initializing variables using `let` and `const`, and understanding the differences between them.

Then we'll talk about another keyword for creating variables called `var`.

# Declaring Variables with `let`

**What Does Declaration Mean?** When you declare a variable, you are essentially telling JavaScript to create a space in memory to store data. However, at this point, the variable doesn't have a value yet—it's just an empty container.

```
JavaScript  
// declare a variable named myBootcamp  
let myBootcamp;
```

Here, `myBootcamp` is declared but not yet initialized, meaning it exists in memory but doesn't hold any value.

# Initializing Variables with `let`

**What Does Initialization Mean?** Initialization occurs when you assign a value to a previously declared variable. This is the point where the empty container (the variable) actually starts holding something.

```
JavaScript  
// initializing a variable named myBootcamp  
myBootcamp = "Code Accelerator";
```

Now, `myBootcamp` holds the value "Code Accelerator". We've taken our declared variable and given it a purpose.

# Declaring and Initializing in One Step

**Combined Declaration and Initialization:** Oftentimes, you'll just declare and initialize a variable in a single step for convenience and clarity.

```
JavaScript
// declaring AND initializing a variable named myBootcamp
let myBootcamp = "Code Accelerator";
```

This is a common practice, especially when you already know the value you want the variable to hold.

## Understanding `const`

**Introduction to `const`:** While `let` allows you to declare variables that can be reassigned later, `const` is used for variables that should not be reassigned after their initial value is set. This makes `const` ideal for values that should remain constant throughout your code no matter what.

```
JavaScript
// initializing a constant variable and trying to reassign it
const pi = 3.14159;
pi = 3.14; // Output: TypeError: Assignment to constant variable.
```

Here, `pi` is a constant, meaning you cannot reassign `pi` to another value later in your code. If you try to do so, JavaScript will throw an error.

```
JavaScript
// declaring a constant variable without a value
const pi; // Output: SyntaxError: Missing initializer in const declaration
```

You also can't declare a constant without giving it a value in the declaration.

## Use Cases

- Use `let` when you expect the value of a variable to change over time.



- Use `const` when you know that a variable's value should remain constant and unchanged.

## Best Practices

- **Use `const` first.** As a best practice, many developers recommend using `const` when declaring variables and only using `let` when you know the value will need to change.

This approach helps prevent errors (called *bugs*) that would be caused by unintentional reassignment.

- **Clarity and Intent.** Using `const` by default makes your code clearer and more readable, as it clearly indicates which variables should remain unchanged.

## You did it!

Understanding how and when to use `let` and `const` is your new superpower as we move deeper into JavaScript. These concepts help you control the flow of your code and manage data effectively, especially as your programs become more complex.

In Level 1-2, we'll explore how these principles apply in practice through our Tic-Tac-Toe game. You'll see how careful management of variables can lead to cleaner, more reliable code, and you'll gain a deeper appreciation for the role of variables in programming.

Next up, we'll dig into the concept of hoisting, which is closely related to how variables are declared and initialized. Understanding hoisting will give you further understanding of how JavaScript handles your code under the hood.

## Introduction to `var`

In early versions of JavaScript, the only way to declare variables was by using the `var` keyword.

But `var` behaves differently from `let` and `const` in ways that can be confusing, especially for beginners.

Understanding these differences is the next coding superpower you'll unlock as you progress in your coding journey.

## Understanding Where Your Variables Can Be Used (Introducing "Scope")

When you create a variable using `var`, where you place it in your code determines where you can use it. This is known as the variable's **scope**.

- If you create the variable outside of any function, it's said to be in the **global scope**. This means it can be used anywhere in your entire program. Think of it as being available to everyone, everywhere.
- If you create the variable inside a function, it's in the **function scope**. This means it can only be used within that specific function. Imagine it as a special tool that only works inside one room (the function) and can't be taken outside.

Unlike `let` and `const`, which are **block-scoped** (they only work within **blocks** which are smaller sections of your code, like loops or `if` statements), `var` is more flexible—it works throughout the entire function or your entire program (depending on where you declare it).

## Redeclaration with `var`

`var` allows you to redeclare a variable within the same scope without causing an error. This can lead to unexpected behavior and bugs if you're not careful.

JavaScript

```
var myVariable = "Code Accelerator";  
var myVariable = "New"; // No error, `myVariable` is now "New"  
console.log(name); // Output: "New"
```

## Understanding Redeclaration with `let`

When you use `let` to create a variable, there's an important rule to remember: **You can't declare the same variable more than once in the same area of your code.**

- **No Redeclaration in the Same Scope.** If you've already created a variable with `let` in a certain scope (like inside a function or a loop), trying to create another variable with the same name in that same scope will cause an error.
- **Why Is This Useful?** This rule helps prevent mistakes. It stops you from accidentally overwriting a variable that you've already created, which can make your code more reliable and easier to understand.

JavaScript

```
let myVariable = "Code Accelerator";  
let myVariable = "New"; // SyntaxError: Identifier 'myVariable' has already been  
declared
```

Unlike `var`, which allows you to create (or redeclare) the same variable name multiple times in the same scope (which can lead to confusion and bugs), `let` is stricter.

That's why using `let` and `const` helps you keep your variables organized and avoids unintended bugs.

# Understanding Hoisting (Moving Variables To The Top of the Scope)

In JavaScript, there's a behind-the-scenes behavior called **hoisting** that affects how your code runs.

Imagine that JavaScript reads through your code before it actually runs it.

During this reading process, it finds all the places where you declare variables and functions, and it moves those declarations to the top of their scope or section of code (like to the top of a function or your entire program).

This doesn't change the order of your code, but it does change when JavaScript becomes aware of your variables and functions.

**How This Affects `var`.** With `var`, this means you can use a variable before it's actually declared in your code. However, because only the *declaration* is moved to the top, not the assignment, the variable won't have its value yet.

The variable will read as `undefined` until we get to the line where you actually assigned it a value.

```
JavaScript
console.log(x); // Output: undefined
var x = 5;
console.log(x); // Output: 5
```

In this example, even though `var x = 5;` appears after the first `console.log(x);`, JavaScript knows about `x` when the code runs because it moved the declaration to the top. But since the value assignment happens later, `x` is `undefined` at first.

## Hoisting with `var` vs. `let` and `const`

We've talked about how JavaScript moves variable declarations to the top of their section of code before running the code. This works a bit differently depending on whether you use `var`, `let`, or `const`.

**`var`** As we mentioned, `var` lets you use a variable before it's declared in your code because JavaScript moves the declaration to the top. However, since

only the declaration gets moved and not the assignment, the variable will be undefined until you actually give it a value.

```
JavaScript
// JavaScript moves the declaration here: var x;
console.log(x); // Output: undefined
var x = 5; // behaves like a redeclaration
console.log(x); // Output: 5
```

**let** and **const**: When **let** and **const** are also hoisted, they behave differently. Even though their declarations are also moved to the top, you can't use these variables until the code actually reaches the line where they are declared. This creates what's called a **temporal dead zone**—a fancy way of saying you can't use the variable before it's properly declared.

```
JavaScript
console.log(y); // Error: Cannot access 'y' before initialization
let y = 10;
console.log(y); // Output: 10
console.log(z); // Error: Cannot access 'z' before initialization
const z = 20;
console.log(z); // Output: 20
```

## Why This Matters

So, here's what we want you do during Code Accelerator:

- Use **const** first!
- Only use **let** when you expect the value of a variable to change over time.
- Do not use **var** at all.

`var` is more forgiving but can lead to bugs if you're not careful, because it allows you to accidentally use a variable before it's given a value.

`let` and `const` help prevent mistakes by making sure the code can't use the variable until it's actually declared in the code.

Using `let` and `const` is safer and helps you write more reliable code because you can be sure that the variable has been properly set up before your code tries to use it.

## Understanding Functions in JavaScript with Simple Examples

Before we get into building our Tic-Tac-Toe game, let's explore the concept of functions with some everyday examples. These examples will help you understand how functions work and why they are so useful in coding.

### Example #1: A Function to Make a Sandwich

Imagine you're in the kitchen, and you want to make a sandwich. You can think of a function as a set of instructions for making that sandwich. Instead of giving each instruction every time you want to make a sandwich, you can create a function that performs all the steps for you.

## Defining Functions

Here is the format you use to write any function:

```
JavaScript
```

```
function functionName(parameter1, parameter2, parameter3) {  
    return parameter1 + parameter2;  
}
```

And here's how you might write a function to make a sandwich:

JavaScript

```
function makeSandwich(breadType, filling) {  
    return "Here is your " + filling + " sandwich on " + breadType + " bread!";  
}
```

- **Function Name:** `makeSandwich` is the ***function name***. In a perfect world, the name of the function tells you what the function is supposed to do.
- **Parameters:** `breadType` and `filling` are the ***parameters***, or data values that the function needs to do its work. With the function `makeSandwich` the parameters are the ingredients you need to make the sandwich. These are like inputs you give to the function.
- **Return Value:** A function sends information up to your code. When a function sends information up to the code it is called a ***return value***.

The `makeSandwich` function combines the bread and filling and then returns a string data type telling you what kind of sandwich you made.

# Using the `makeSandwich` Function

Now, whenever you want to make a sandwich, you just call the function and give it the ingredients:

```
JavaScript
let myLunch = makeSandwich("whole wheat", "turkey");

console.log(myLunch); // Output: "Here is your turkey sandwich on whole wheat bread!"
```

This is much easier than writing out all those steps every time we want to describe the process of making a sandwich!

## Example 2: A Function to Calculate the Area of a Rectangle

Functions are also great for performing calculations. Let's say you need to calculate the area of a rectangle in different situations. Instead of writing the formula over and over, you can create a function to do it.

# Defining a `calculateArea` Function

Here's a simple function to calculate the area of a rectangle:

```
JavaScript
function calculateArea(length, width) {
  return length * width;
}
```

- **Function Name:** `calculateArea` tells you what the function does—it calculates the area.
- **Parameters:** `length` and `width` are the inputs you need to perform the calculation.



- **Return Value:** The function multiplies the length by the width and returns the area.

## Using the Function

Now, you can easily calculate the area of any rectangle in one step:

```
JavaScript  
let area = calculateArea(5, 3);  
console.log(area); // Output: 15
```

This function makes your code shorter, clearer, and easier to update if you need to change the way you calculate the area.

## The `console` Object and the `log` Method

Before we dive deeper into understanding functions, let's briefly revisit what a function is by looking at something you've already been using: the `console.log` statement.

This is a great example of how functions work in JavaScript.

## The `console` Object

In JavaScript, the `console` is a built-in object that provides access to the browser's debugging console. It has several methods that allow you to interact with the console, and one of the most commonly used methods is `log`.

## The `log` Method

- **What is `log`?** `log` is a method (or function) of the `console` object. When you call `console.log()`, you're using this method to print messages, values, or other information to the console.

- **How Does It Work?** Whenever you write `console.log("Hello, world!");` you're basically saying: "Hey JavaScript engine, run this `log` function and pass it the message **Hello, world!** to display in the console."

## How `console.log` Relates to Defining Functions

- **Function Name:** In this case, `log` is the name of the function.
- **Parameters:** The message or value you want to print is the parameter you pass into the function. `"Hello, world!"` is the parameter or input that the `console.log` function uses to perform its task of printing to the console.
- **Return Value:** While `console.log` doesn't return a value that you can use later, it performs an action—outputting text to the console.

Here's a breakdown of how it looks:

```
JavaScript  
console.log("Hello, world!"); // Output: ??
```

- **`console`:** The object that holds the `log` method.
- **`log`:** The method or function that performs the action.
- **`"Hello, world!"`:** The parameter that gets passed into the function.

## Why This Matters

By using `console.log`, you've already been working with a function in JavaScript. This example shows how functions take inputs (parameters), do something with them, and (sometimes) give something back.

Functions like `console.log` are incredibly powerful because they let you

reuse the same code with different inputs, making your code more efficient and easier to manage.

Understanding how `console.log` works gives you a solid foundation to explore more complex functions, like the ones we'll encounter in the Tic-Tac-Toe game.

Functions are everywhere in JavaScript, and learning about them is an important step to becoming a confident coder.

Functions will be your go-to tool for solving problems in a structured, repeatable way as you continue learning JavaScript.

## Introducing Hoisting and Function Scope with Simple Functions

It's important to revisit our understanding of how JavaScript handles functions, especially in terms of **hoisting** and **function scope**. These concepts will help you understand why certain things happen when you run your code.

### Do You Remember What Hoisting Is?

Hoisting is the behavior in JavaScript where function and variable declarations are moved to the top of their containing scope (like the top of a script or a function) before the code is actually executed. This means you can call a function before you've even written it in your code, and it will still work!

### A Function to Make Coffee

Let's say you want to make a cup of coffee. Here's a function that does that:

```
JavaScript  
console.log(makeCoffee()); // Output: "Your coffee is ready!"
```

```
function makeCoffee() {  
  return "Your coffee is ready!";  
}
```

**How It Works:** Even though the `console.log` comes *before* the `makeCoffee` function in the code, JavaScript “hoists” the function declaration to the top.

So, when the code runs, JavaScript already knows about the `makeCoffee` function, and everything works fine.

### Do You Remember What Scope Is?

Scope says which section in your code a variable or function is accessible.

When we talk about **function scope**, it means that variables or functions declared inside a function are only accessible within that function—they are hidden from the rest of the program.

### Example: A Function to Calculate Discount

Imagine you’re writing a function to calculate a discount on a product:

```
JavaScript  
function calculateDiscount(price) {  
  let discount = 0.1; // 10% discount  
  return price - (price * discount);  
}  
  
console.log(calculateDiscount(100)); // Output: 90  
console.log(discount); // Error: discount is not defined
```

### How It Works:

- The `discount` variable is declared inside the `calculateDiscount` function. This means it only exists within that function—it has **function scope**.

- If you try to access `discount` outside the function, like in the second `console.log`, you'll get an error because `discount` is not visible outside of `calculateDiscount`. This is a good thing!

## The `var` Quirk

Now, here's one of the many interesting quirks about JavaScript coding:

- **What If We Had Used `var` Instead?**

If we had used `var` instead of `let` to declare `discount`, it would have function scope, so the `discount` variable would still not be accessible outside the function.

However, in some other cases, `var` behaves differently. For example, if `var` was used in a loop inside the function, the variable might accidentally "leak" outside of the loop, leading to unexpected behavior.

- **Why We Avoid `var`:**

This is one of the reasons why developers should use `let` and `const`—they help avoid these kinds of mistakes by clearly defining the scope of function variables, making your code more predictable and easier to understand.

## Arrays and Indexed Collections

In Level 1-1, we introduced the idea of arrays as a way to group related items together in an organized list.

Think of an array like a bookshelf, where each book (or item) has its own specific spot on the shelf.

Arrays are one of the most basic tools you'll use in JavaScript, but they are also incredibly powerful.

They help you keep your data organized, make your code more efficient, and allow you to do more with less effort. Understanding how to use arrays is a JavaScript superpower that you're going to want to unlock.

Arrays keep everything in a specific order.

Imagine you have a list of your favorite fruits or a list of scores from a game. Instead of writing down each fruit or score separately, you can put them all together in an array.

Each item in an array has a number, called an *index*, which tells you its position in the list.

JavaScript is a *0-index programming language*. That means the first item in an array is at *index 0*, the second in an array is at *index 1*, and so on.

Arrays make it easy to:

- **Store a List of Items:** Whether it's names, numbers, or anything else, you can keep everything in one organized list.
- **Access Items Quickly:** By using the index number, you can easily find or change any item in the list.
- **Loop Through Items:** If you want to do something with each item (like printing them out or adding them up), arrays let you go through the list automatically.

Here's an example to make it clearer:

```
JavaScript  
let fruits = ["apple", "banana", "cherry"];
```

In this array:

- "apple" is the first item, so it's at index [0].
- "banana" is the second item, so it's at index [1].
- "cherry" is the third item, so it's at index [2].

You can easily get any item by its index:

JavaScript

```
console.log(fruits[1]); // This will print "banana"
```

## Loops

Arrays are used with **loops** to perform the same operation on each **element** (the term “element” is the developer term for any item in an array).

Let's say you want to print out every element in the **fruits** variable.

# Using a **for** Loop

Here's an example to make it clearer:

```
JavaScript
let fruits = ["apple", "banana", "cherry"];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
/*
Output:
apple
banana
cherry
*/
```

## Understanding the **for** Loop and Array Length

Before we dive into what's happening in the code, let's break down the key parts:

- **Array Length.** Every array in JavaScript has a **length** property, which tells you how many items are in the array. In our case, **fruits.length** is 3 because there are three fruits in the list: "apple", "banana", and "cherry".
- **Loop Structure.** A **for** loop is made up of three parts:
  1. **initializer:** The initializer section of the **for loop** can be used to count the turns in a loop. Here, our loop starts by setting **i** to 0.
  2. **condition:** The loop keeps running as long as whatever is in the condition section equals **true**. In this case, the loop will continue to run—turn after turn—as long as **i < fruits.length** (which means the loop will continue to loop over and over again if **i** is less than



the 3 items in the array length).

3. **increment:** The increment section of the **for loop** can be used to count the turns in a loop. `i++` means `i = i + 1`. So after each run of the loop, `i` increases by 1, then we move on to the next item in the array.

## How the Loop Works

Now, let's talk about what happens when the loop runs:

The loop kicks off with `i` starting at 0. It keeps going as long as `i` is less than the total length of the `fruits` array (which has 3 items in it).

Each time the loop runs (and it runs three times in this case), `fruits[i]` grabs the fruit at the current spot in the list, and `console.log(fruits[i])` prints that fruit to the console for you to see.

So, as `i` goes from 0 to 2, the loop prints each fruit in the order they appear in the array.

On the fourth turn, `i` becomes 3, which is equal to the length of the array.

At this point, the loop checks the condition (`i < fruits.length`), sees that it's no longer true, and stops running.

That's why the loop only prints three fruits—one for each time it runs.

# Introduction to Comparison

In coding, comparing values is an important concept that helps your code make decisions.

It's like asking your program a question: "Is this value smaller than that one?" or "Are these two things the same?" These comparisons guide your code to take different actions based on the answers.

One of the simplest and most straightforward comparisons is the `<` or **less than operator**.

This operator is like asking your code, "Is this value smaller than that one?" It's a crucial part of how your code makes decisions and controls what happens next.

You've already seen this operator in action when we used it in our `for` loop. Let's take a moment to see how it works and why it's so important.

## The Less Than Operator in Action

Here's an example to make it clearer:

```
JavaScript
let fruits = ["apple", "banana", "cherry"];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

## What's Going On Here?

- **`i < fruits.length`**: This part of the loop checks if `i` (our loop counter) is still less than the total number of items in the `fruits` array.
- **Why It Matters**. This comparison keeps the loop running the right number of times. As long as `i` is less than the length of the `fruits`, the

loop continues. Once `i` reaches the array's length, the loop stops, making sure we don't try to access items that aren't there.

## Connecting the Dots

Understanding operators is a basic but powerful superpower that keeps your code in check.

It helps guide your loops, ensuring they do exactly what you need without going too far.

In our `for` loop, this simple comparison made sure we processed every fruit in the array, one by one, and then stopped when we were done.

This simple check is a great introduction to how programming lets you evaluate and compare values to make decisions. And just like you can check if one value is less than another, you can also check if two values are equal. That's where ***equality comparisons*** come in.

## From `<` to Equality Comparisons

Just like the `<` operator helps you determine if a value is smaller, equality comparisons let you check if two values are the same.

In JavaScript, there are two ways to compare for equality: ***loose equality*** (`==`) and ***strict equality*** (`===`). Each one has its own special way of working, depending on how precise you want to be.

## Loose Equality (`==`)

Loose equality is like asking, "Are these values equal, even if they're not exactly the same type?" JavaScript might try to convert the values to the same type before making the comparison.

JavaScript

```
console.log(5 == "5"); // Output: true
```

## What's Happening Here?

- JavaScript sees that one value is a **number** and the other is a **string**, so it converts the string "5" to the number 5 before comparing.
- Since both values are now 5, the comparison returns **true**.

## Strict Equality (===)

Strict equality asks a slightly different question: "Are these values exactly the same, with no type conversion?" With strict equality, JavaScript compares both the value and the type as they are.

JavaScript

```
console.log(5 === "5"); // Output: false
```

## What's Happening Here?

- This time, JavaScript doesn't try to change anything. It sees that one value is a number and the other is a string, so it says they're not the same.
- The comparison returns **false** because the types don't match.

## Why It Matters

Understanding these different types of comparisons helps you write code that does exactly what you expect. Just like the `<` operator helps you figure out differences in value, equality operators help you check if values match up the way you want them to.

## Practical Example: Checking User Input

Imagine you're building a simple login system. You want to check if the user's input matches the correct password.

### Loose Equality:

```
JavaScript
let correctPassword = "1234";
let userInput = 1234;

console.log(userInput == correctPassword); // Output: true
```

Even though the user entered a number and the password is stored as a string, loose equality says they match because JavaScript converts the number to a string data type.

### Strict Equality:

```
JavaScript
let correctPassword = "1234";
let userInput = 1234;

console.log(userInput === correctPassword); // Output: false
```

Strict equality, however, sees that one is a string and the other is a number, so it says they don't match.

In most cases, especially with something important like passwords, you'd want to use strict equality to make sure everything matches up perfectly.

Understanding the difference between loose and strict equality in JavaScript is key to writing reliable code.

While loose equality can be handy in some situations, it's often better to use strict equality to avoid unexpected results. By being clear about what you're comparing, you'll make your code more predictable and easier to debug.

Remember, in most cases, it's safer to use `===` to make sure that both the value and the type match exactly.

# Bringing It All Together: The `.every` Array Method

Now that you have a solid understanding of loops, comparisons, and arrays, it's time to bring these concepts together by exploring a powerful tool in JavaScript: the `.every` array method.

This method allows you to check if every item in an array meets a certain condition, making your code more efficient and easier to read.

## What is the `.every` Method?

The `.every` method is a built-in function that you can use on arrays. It tests whether all elements in the array pass a test (that you define in a function) and returns `true` if they do, or `false` if *even one element fails the test*.

### Example: Checking if All Numbers are Positive

Let's say you have an array of numbers, and you want to check if every number in the array is positive.

```
JavaScript
let numbers = [1, 2, 3, 4, 5];

let allPositive = numbers.every(function(number) {
  return number > 0;
});

console.log(allPositive); // Output: true
```

```
JavaScript
let numbers = [1, 2, 3, -4, 5];

let allPositive = numbers.every(function(number) {
```

```
return number > 0;
});

console.log(allPositive); // Output: false
```

## How It Works

- **The Array:** We start with an array of numbers: [1, 2, 3, 4, 5].
- **The .every Method:** We use the .every method to check if every number in the array is greater than 0.
- **The Function:** Inside .every, we define a function that takes each number and checks if it's greater than 0. If all numbers pass this test, .every returns true; if any number fails, it returns false.

## Why Use .every?

Using .every simplifies your code, especially when you need to check all items in an array against the same condition. Instead of writing a for loop and manually checking each element, .every does it all for you in a single line.

### Practical Example: Validating User Input

Imagine you're writing a program that checks if every input in a form is filled out correctly. You can use .every to make sure every field in an array of form inputs is not empty.

```
JavaScript
let formInputs = ["Code", "Accelerator", "learn@codeaccelerator.org"];

let allFilled = formInputs.every(function(input) {
  return input !== "";
});
```

```
console.log(allFilled); // Output: true
```

## How It Works

- We have an array of form inputs (e.g., a first name, last name, and email).
- The `.every` method checks if every input is not an empty string.
- If all inputs are filled out, `.every` returns `true`; otherwise, it returns `false`.

## Connecting the Dots

Let's bring everything together with a recap of how the `.every` method ties into what you've already learned:

1. **Arrays:** You understand that arrays are organized collections of items.
2. **Loops:** You know how to loop through an array with a `for` loop to check each item.
3. **Comparisons:** You've learned how to compare values, like checking if a number is positive or if a string is not empty.

The `.every` method is a function that we can use on arrays that combines all of these skills into one powerful tool that simplifies your code and makes it more readable.

## You've Come a Long Way: Let's Make A Tic-Tac-Toe Game

Take a moment to think about how much you've learned. At first, things like arrays, loops, and comparisons might have seemed a bit tricky, but now you're using them like a pro.

You've been building your skills step by step, and now we're ready to use everything we've learned to tackle a fun challenge: coding a tic-tac-toe game. So far:



- **Arrays:** You've learned how to organize data in lists, and how to use an array to represent something.
- **Loops:** You know how to go through each item in an array, making sure nothing gets missed.
- **Comparisons:** You can check if values are what you expect, and use those checks to decide what your code should do next.

## How Other Developers Did It

One of the best ways to become a confident coder is to study how other developers solved problems. Let's take a look at how some developers approached the challenge of creating a Tic-Tac-Toe game and the different ways they went about it.

### Example 1: A Detailed Solution on GeeksforGeeks

On GeeksforGeeks, there's a solution to the Tic-Tac-Toe game that spans 427 lines of JavaScript. This developer took a detailed approach, writing a lot of code to handle all the different parts of the game. Here's a small snippet of their solution:

```
JavaScript
function myfunc_4() {
  if (flag == 1) {
    document.getElementById("b2").value = "X";
    document.getElementById("b2").disabled = true;
    flag = 0;
  }
  else {
    document.getElementById("b2").value = "O";
    document.getElementById("b2").disabled = true;
    flag = 1;
  }
}
```

You can check out the full solution here: [GeeksforGeeks Tic-Tac-Toe Game](#)

### Example 2: A Simpler Approach on Medium

Another web developer, Canan Korkut, wrote a solution with only 70 lines of JavaScript. Her approach is more streamlined, focusing on the essentials of the game. Here's a piece of her code:

JavaScript

```
function checkTie(){
  for(let i = 0; i < squares.length; i++) {
    if(squares[i].textContent === "") {
      return false;
    }
  }
  return true;
}
```

You can read her full tutorial here: [Medium Tic-Tac-Toe Tutorial](#)

# What These Solutions Have in Common

If you explore these tutorials, you'll notice that while these developers took very different approaches, there's something they both have in common: they use a JavaScript control structure called the `if` statement.

`if` lets your code make decisions, which is important in games like Tic-Tac-Toe.

## Unlocking the `if` Statement Superpower

Both of these solutions rely on the `if` statement to check conditions and decide what the game should do next—whether it's placing an "X" or an "O", or checking if someone has won. As you continue learning, you'll see just how powerful the `if` statement is, and you'll be using it to make your own games and projects come to life.

## Understanding `if` Statements

The `if` statement is one of the most important tools in your programming toolbox. It allows your code to make decisions, which is key to building anything interactive, like a game or a dynamic website.

With `if` statements, you can tell your program, "If this condition is `true`, then do this. Otherwise, do something `else`."

## The Basics of `if` Statements

At its core, an `if` statement checks whether something is true or false. If the condition you're checking is true, the code inside the `if` block runs. If it's false, the code is skipped, and your program moves on.

### Example: Checking a Number

```
JavaScript  
let number = 10;
```

```
if (number > 5) {  
  console.log("The number is greater than 5");  
}
```

## What's Happening Here?

- We've got a number, **10**, and we want to check if it's greater than **5**.
- The **if** statement checks this condition: **number > 5**.
- Since **10** is indeed greater than **5**, the code inside the **if** block runs, and you see the message "**The number is greater than 5**" printed to the console.

## Why **if** Statements Are So Powerful

The **if** statement is like a fork in the road for your code. It lets your program decide which path to take based on the conditions you set. This is what makes interactive programs possible—your code can react differently depending on the input it gets or the state of the game.

## Connecting The Dots

Understanding **if** statements is a big step forward in your coding journey. They allow you to control the flow of your program and make it do different things based on different conditions. Whether you're checking if a player has won a game of Tic-Tac-Toe or deciding which message to show on a website, **if** statements give you the power to make your code smart and responsive to changing conditions.

As you continue learning, you'll see how important your **if** superpowers become in bringing your projects to life.

## You're Ready!

Now that we've discussed `if` statements, loops, comparisons, and arrays, you've got all the tools you need to understand the first version of our Tic-Tac-Toe coding challenge solution. Let's bring everything together by thinking through the problem with a bit of design thinking.

## Design Thinking

Let's start by reviewing what we know about Tic-Tac-Toe:

1. **Two Players:** The game is played by two players, X and O.
2. **Nine Spaces:** There are nine spaces on the board where a player can place their mark.
3. **Player X Goes First:** The game always starts with Player X making the first move.
4. **Eight Ways to Win:** There are only eight different ways that a player can win (three rows, three columns, and two diagonals).

### We Can Code This!

We know that there are nine cells where players can place their marks, so let's represent the Tic-Tac-Toe board using an array with nine elements:

```
JavaScript  
let board = [null, null, null, null, null, null, null, null, null];
```

Each element in this array lines up to a cell on the board, with the first element (`board[0]`) representing the top-left corner and the last element (`board[8]`) representing the bottom-right corner.

Remember, arrays in JavaScript start with an index of 0, so the cells are indexed from 0 to 8.

0	1	2
3	4	5
6	7	8

## Representing Winning Patterns

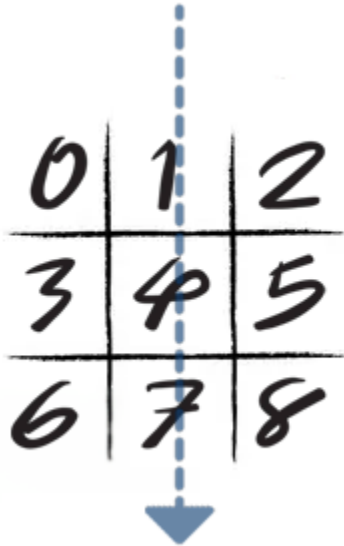
Now that we've set up the board, we can think about how a player wins. A player wins by getting three of their marks in a row, either horizontally, vertically, or diagonally.

### Example: Vertical Win in the Middle

One way to win is by having three marks in a column. For example, if a player places their marks in the middle column:

	○	×
×	○	
	○	×

They would occupy the cells with indexes 1, 4, and 7:



0	1	2
3	4	5
6	7	8

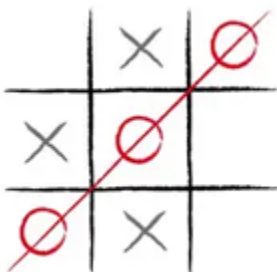
JavaScript

```
// Middle column winning combination
```

```
[1, 4, 7]
```

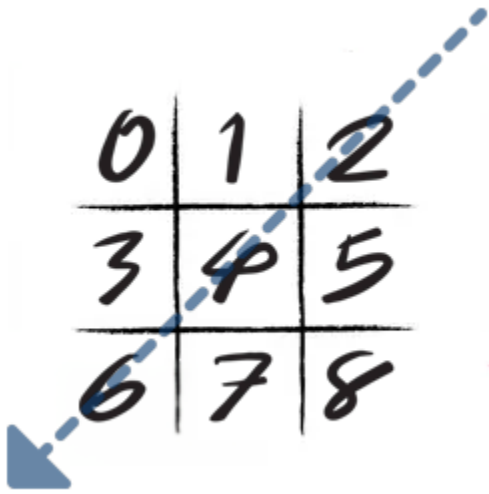
### Example: Diagonal Win from Upper-Right to Lower-Left

Another way to win is by getting three marks diagonally. For example, if a player places their marks from the upper-right corner to the lower-left corner:



	X	○
X	○	
○	X	

They would occupy the cells with indexes 2, 4, and 6:



JavaScript

```
// Diagonal winning combination from upper-right to lower-left  
[2, 4, 6]
```

## Putting It All Together

By representing the board as an array and understanding the indexes of each cell, we can easily track where each player places their marks.

Now, we can define all winning combinations by listing the indexes that need to be occupied by the same player's mark:

JavaScript

```
const allWinningCombinations = [  
  [0, 1, 2], // Top row  
  [3, 4, 5], // Middle row  
  [6, 7, 8], // Bottom row  
  [0, 3, 6], // Left column  
  [1, 4, 7], // Middle column
```



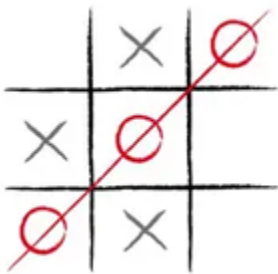
```
[2, 5, 8], // Right column
[0, 4, 8], // Diagonal from top-left to bottom-right
[2, 4, 6] // Diagonal from top-right to bottom-left
];
```

Believe it or not, we have just built a large part of our solution.

The cool thing about design thinking is that you can solve a lot of coding problems in your mind *before you start coding*.

Now that we know all of the winning combinations, we can design the code that we use to check for a win.

First, we'll set up a board that has a winning combination. Let's set up a board where Player O wins diagonally:



JavaScript

```
let board = [null, null, null, null, null, null, null, null, null];
```

```
// Example board state (Xs and Os):
```

```
// null X O
```

```
// X O null
```

```
// O X null
```

```
board = [
  null, 'X', 'O',
  'X', 'O', null,
  'O', 'X', null
];
```

## Remember the **for** loop?

With all the pieces in place, we can now use a **for** loop to go through each of the winning combinations and check if any of them match the current board state. This is where everything you've learned about loops and comparisons comes together to complete the Tic-Tac-Toe solution!

JavaScript

```
function checkForWin(player) {
  const allWinningCombinations = [
    [0, 1, 2], // Top row
    [3, 4, 5], // Middle row
    [6, 7, 8], // Bottom row
    [0, 3, 6], // Left column
    [1, 4, 7], // Middle column
    [2, 5, 8], // Right column
    [0, 4, 8], // Diagonal from top-left to bottom-right
    [2, 4, 6] // Diagonal from top-right to bottom-left
  ];

  // Loop through each winning combination
  for (let i = 0; i < allWinningCombinations.length; i++) {
    const combination = allWinningCombinations[i];
    const a = combination[0];
    const b = combination[1];
    const c = combination[2];

    // Check if the player occupies all three spots in the current winning combination
    if (board[a] === player && board[b] === player && board[c] === player) {
      return true; // Player O has a winning combination
    }
  }

  return false; // No winning combination found
};
```

```
// Example board state where Player O wins diagonally
let board = [
  null, 'X', 'O',
  'X', 'O', null,
  'O', 'X', null
];

// Check if Player O has won
if (checkForWin('O')) {
  console.log("Player O wins!");
} else {
  console.log("No winner yet.");
};
```

## Let's Walk Through the Code Together, Line by Line

Now that we've written the function to check if Player O has a winning combination, let's break it down line by line. This will help you understand how each part of the code works together to solve the Tic-Tac-Toe challenge.

### 1. Function Definition

```
JavaScript
function checkForWin(player) {
```

**What's Happening?:** We're defining a new function called `checkForWin`. This function will take one argument, `player`, which will be the player we're checking for a win (in our case, Player O, represented as `'O'`).

## 2. Define Winning Combinations

```
JavaScript
const allWinningCombinations = [
  [0, 1, 2], // Top row
  [3, 4, 5], // Middle row
  [6, 7, 8], // Bottom row
  [0, 3, 6], // Left column
  [1, 4, 7], // Middle column
  [2, 5, 8], // Right column
  [0, 4, 8], // Diagonal from top-left to bottom-right
  [2, 4, 6] // Diagonal from top-right to bottom-left
];
```

**What's Happening?:** We're creating a list (an array) of all possible winning combinations on the Tic-Tac-Toe board. Each combination is represented by a smaller array of three numbers, where each number corresponds to a position on the board. For example, `[0, 1, 2]` represents the top row.

## 3. Loop Through Each Winning Combination

```
JavaScript
for (let i = 0; i < allWinningCombinations.length; i++) {
```

**What's Happening?:** We start a `for` loop that will go through each winning combination one by one. The loop will run as many times as there are winning combinations, which is 8 in this case. The variable `i` will keep track of which combination we're currently checking.

## 4. Accessing the Current Combination

JavaScript

```
const combination = allWinningCombinations[i];  
const a = combination[0];  
const b = combination[1];  
const c = combination[2];
```

### What's Happening?:

**combination = allWinningCombinations[i];** We're taking the current winning combination from our list and storing it in a variable called **combination**.

**a = combination[0]; b = combination[1]; c = combination[2];** We're breaking down this combination into three separate variables, **a**, **b**, and **c**, which represent the three positions on the board that make up this winning combination.

## 5. Checking If the Player Occupies All Three Spots

JavaScript

```
if (board[a] === player && board[b] === player && board[c] === player) {
```

**What's Happening?:** Here, we're using an `if` statement to check if the player has their mark ('O' in this case) in all three positions of the current combination (**a**, **b**, and **c**). The `&&` operator means "and," so we're asking, "Does the player have their mark in position **a** **and** position **b** **and** position **c**?"

## 6. Return `true` if a Winning Combination is Found

```
JavaScript  
return true;
```

**What's Happening?:** If the player occupies all three positions in this combination, we return `true` from the function, meaning the player has won.

## 7. End of the Loop and Return `false`

```
JavaScript  
}  
return false;
```

**What's Happening?:** If the loop finishes checking all the combinations and doesn't find a winning one, we return `false`, meaning there is no winner yet.

## 8. Example Board State and Function Call

```
JavaScript  
let board = [  
  null, 'X', 'O',  
  'X', 'O', null,  
  'O', 'X', null  
];  
  
if (checkForWin('O')) {  
  console.log("Player O wins!");  
} else {  
  console.log("No winner yet.");  
}
```

**What's Happening?:** We set up a sample board where Player O has a diagonal win, then we call `checkForWin('O')` to see if Player O has won. Depending on the result, we print either "Player O wins!" or "No winner yet."

## Wow! You've Come So Far

Take a moment to appreciate how much progress you've made. When you started, concepts like arrays, loops, and comparisons might have felt new and challenging, but look at where you are now! You've just walked through the process of building a key part of a Tic-Tac-Toe game, breaking down the code step by step, and understanding how it all fits together.

## What You've Accomplished

**Mastered Arrays:** You've learned how to represent complex structures, like a Tic-Tac-Toe board, using arrays.

**Controlled Logic with Loops:** You've used loops to efficiently check through possibilities, like finding a winning combination on the board.

**Made Decisions with `if` Statements:** You've seen how `if` statements help your code make decisions, checking conditions and responding accordingly.

## Keep Going

Coding is a journey, and you're on a great path. Keep challenging yourself with new projects, explore different ways to solve problems, and most importantly, have fun with it! Every line of code you write is another step forward in your journey to becoming a confident and skilled developer.

You've come so far, and there's so much more you can achieve. Keep up the amazing work, and remember—there's no limit to what you can create with code!

# Homework: Grow Your Superpowers

Now that you've made incredible progress, it's time to put your knowledge to the test with some hands-on practice. This homework will help you reinforce what you've learned and give you the confidence to tackle coding challenges on your own.

## 1. Create Your Own Tic-Tac-Toe Board

**Task:** Write a JavaScript program that sets up a Tic-Tac-Toe board as an array and allows two players (X and O) to take turns placing their marks.

**Goal:** The program should display the board in the console after each move, showing the current state of the game.

**Hint:** Use a loop to keep the game going until all spots are filled or a player wins.

## 2. Check for a Winner

**Task:** Expand your Tic-Tac-Toe program by adding a function that checks for a winner after each move. Use the `checkForWin` function you learned about, but try writing it yourself without looking back at the notes.

**Goal:** Your program should announce the winner or declare a tie if all spots are filled and no one has won.

**Hint:** Remember the winning combinations and use a loop to check each one.

## 3. Experiment with New Winning Combinations

**Task:** Modify the game to allow a larger board, like a 4x4 grid. Adjust your `checkForWin` function to find winning combinations on this bigger board.

**Goal:** The game should still correctly identify winners, even with the new board size.

**Hint:** You'll need to add more winning combinations to your list and think about how to check them.



#### **4. Reflection**

**Task:** Write a short paragraph reflecting on what you found easy or challenging about building the Tic-Tac-Toe game. What did you learn? What would you like to explore next?

**Goal:** This reflection will help you identify your strengths and areas where you might want to focus more attention.

## Extra Challenge: Make It Interactive

If you're feeling confident, try making your Tic-Tac-Toe game interactive by allowing players to enter their moves via prompt inputs or even building a simple web interface using HTML and CSS. This is a great way to see your code in action and start thinking about how to bring your projects to life on the web.

This homework is designed to help you build on everything you've learned so far. By working through these tasks, you'll reinforce your understanding and gain the confidence to take on even more complex challenges.

Remember, practice is key to mastering coding, so dive in and have fun with these exercises!

## What's Next?

Now that you've got the basics down, the sky's the limit. Whether you're looking to build more games, create interactive web pages, or solve complex coding challenges, you have the tools you need to get started. Remember, every coder, no matter how advanced, started where you are now—with the basics. The more you practice, the more these concepts will become second nature, and the more creative and powerful your coding will become.

In the next learning session we're going to pivot to talking about HTML, CSS, and how we can add user input to the Tic-Tac-Toe game we're building.

See you there!