CSS Animations

CSS Animations help to animate elements from one frame to another. Controlling various CSS properties helps in changing the animation style and time without having the need to use JavaScript for the same.

The most basic way to do so is by using “from” and “to”.

Eg:

```
@keyframes demo {    from {        background-color: red;    }    to
{        background-color: blue;    }}
```

The animation here would start with the background color as red and will turn blue gradually. To use this animation we simply add it to the div stylesheet rule.

This is the most basic way to do CSS animations, but now let’s look into the properties for advanced animation styling.

1. Animation Name

To identify the animation we assign it with the name. In the above example the name was demo. So, we will add the attribute animation name in the div element.

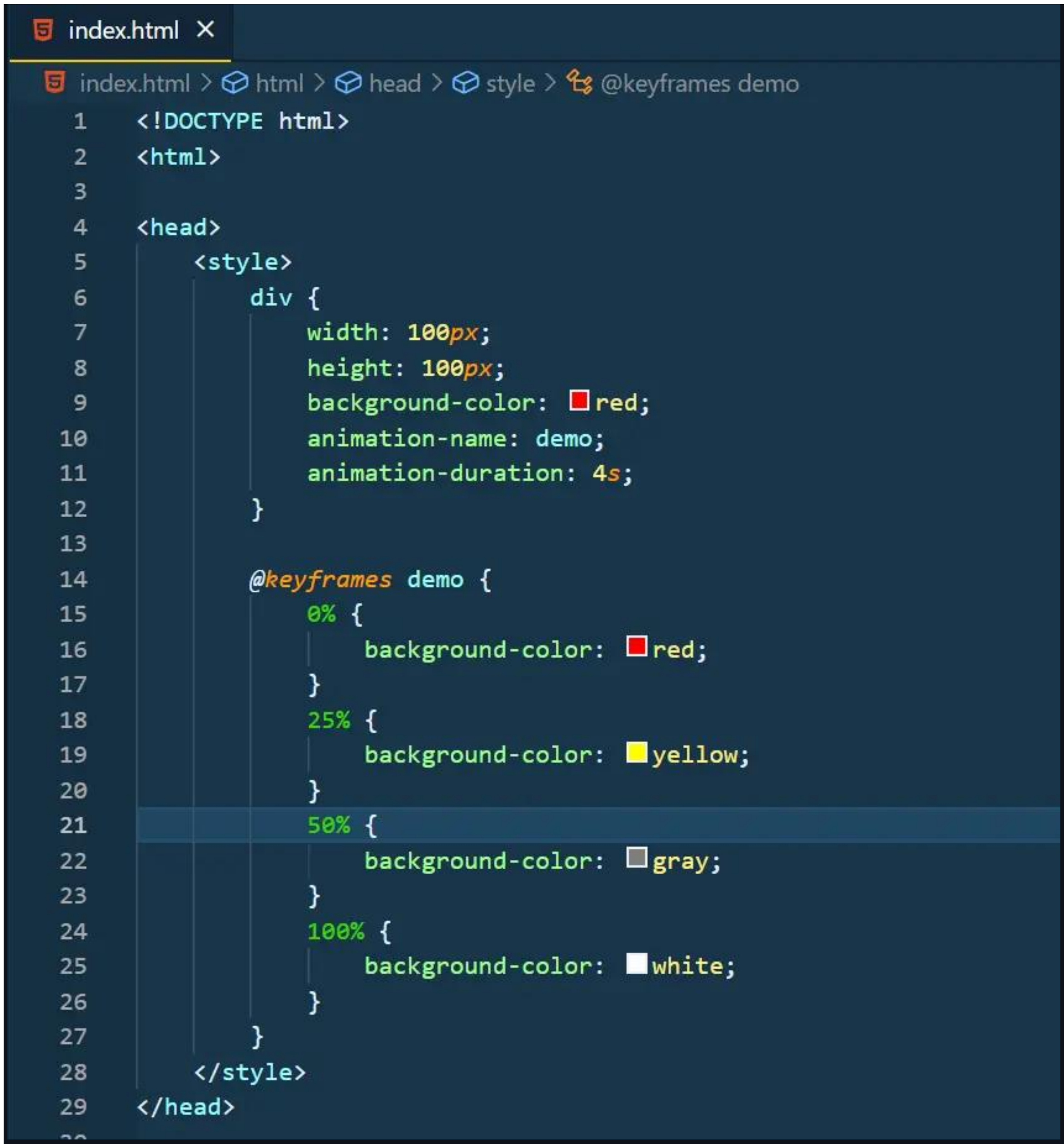
```
Syntax: div { animation-name: demo; }
```

2. Key Frames

Keyframes define at what % of the animation, which stylesheet rules should be used. The value of keyframes varies from 0% to 100%.

To add keyframes use the % operator with the value inside the keyframes. More keyframes mean smoother animation.

Eg:

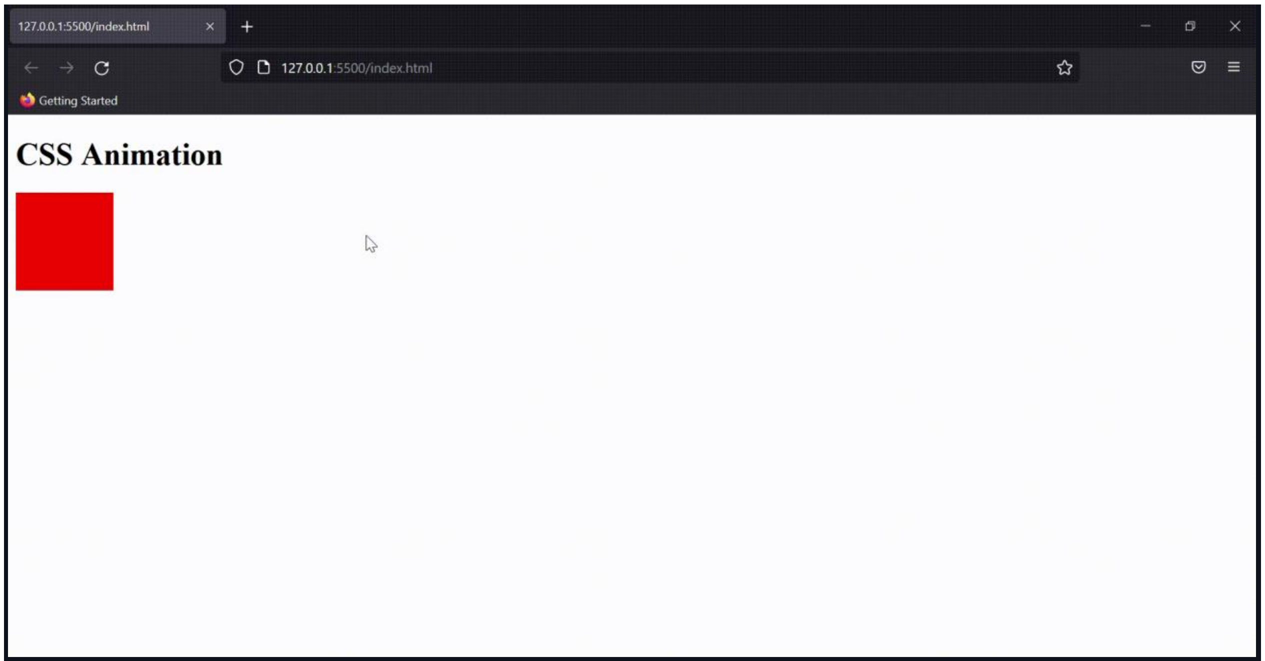


```
<!DOCTYPE html>
<html>

<head>
    <style>
        div {
            width: 100px;
            height: 100px;
            background-color: red;
            animation-name: demo;
            animation-duration: 4s;
        }

        @keyframes demo {
            0% {
                background-color: red;
            }
            25% {
                background-color: yellow;
            }
            50% {
                background-color: gray;
            }
            100% {
                background-color: white;
            }
        }
    </style>
</head>
```

Output:



3. Looping Animations

By default, the animation runs only one time on the website but you can choose to loop the animation by using the “animation-iteration-count” property.

```
Syntax: { animation-iteration-count: value; }
```

Just assign the values to the property and the number of times the animation will run.

To keep the animation running continuously we keep the value as “infinite”.

4. Animation Durations

Animation Duration is the key property to start the animations. By default, the animation duration is 0 sec, which means the animation won’t start.

So, to start the animation we will assign the value to the animation duration.

```
Syntax: animation-duration: value(in sec);
```

5. Other Properties

Animation Delay

Delays the start of animation on the website. Adding a negative value to delay will make the animation run even before the website is loaded.

Eg:

```
{ animation-delay: -2s; }
```

Animation Direction

This property defines the order of the animation. Whether it should be played forward, reverse, or alternatively. The values it takes are:

- **normal**: the default state of forward animation.
- **reverse**: plays the animation in the reverse direction.
- **alternate**: forward animation and then backward.
- **alternate-reverse**: reverse animation and then forward.

```
Syntax: { animation-direction: value; }
```

Animation Timing Function

It defines the speed curve for the animation. This property adds ease to our animation and can have the following values: ease, linear, ease-in, ease-out, and ease-in-out. As the properties are self-explanatory, you can see the output accordingly.

```
Syntax: animation-timing-function: value;
```

Animation Fill Mode

This property defines the style of the element when the animation is either stopped or about to start. The value it takes is none (default state), forwards (value set by the last keyframe), backwards (value of the first keyframe), or both.

```
Syntax: animation-fill-mode: value;
```

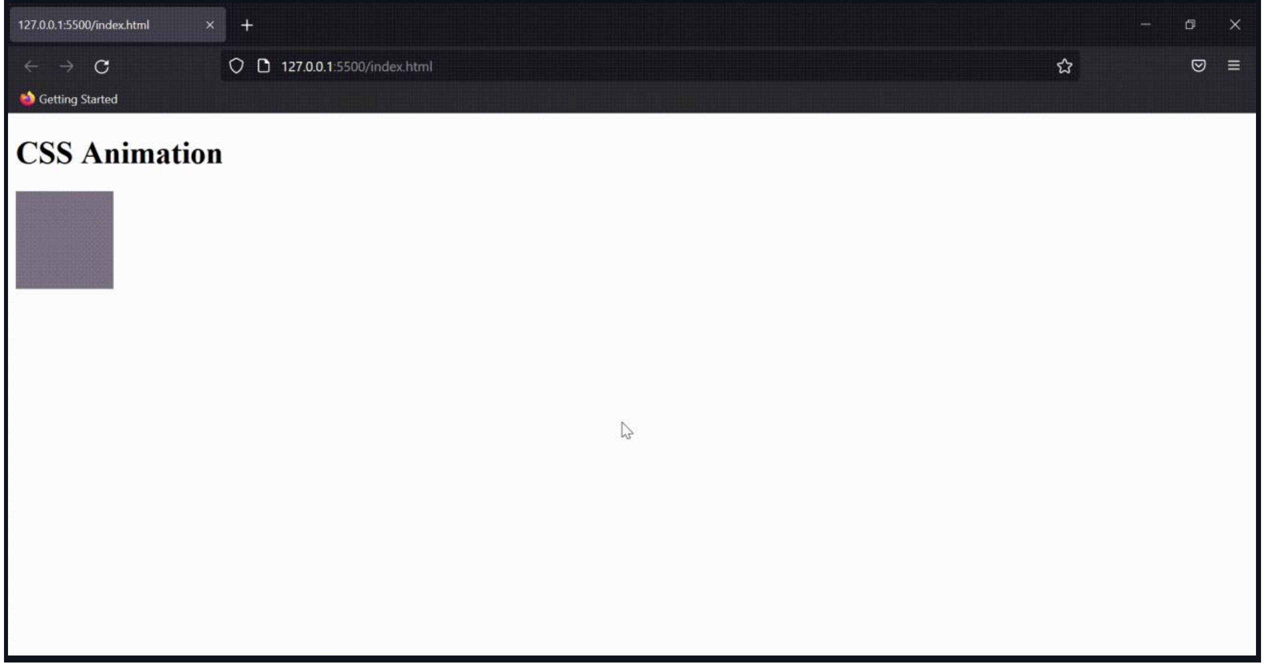
Animation Shorthand

We’ve seen shorthand properties for other CSS properties and the same can be applied to Animation too.

Eg:

```
animation: demo 5s ease-in 1s infinite normal both;
```

Output:



CSS Animation

JAVASCRIPT NOTES:

What are Variables?

In JavaScript, variables are used to store data. They are an essential part of any programming language, as they allow you to store, retrieve, and manipulate data in your programs.

There are a few different ways to declare variables in JavaScript, each with its own syntax and rules. In this blog post, we'll take a look at the different types of variables available in JavaScript and how to use them.

Declaring Variables

To declare a variable in JavaScript, you use the `var` keyword followed by the name of the variable. For example:

```
var x;
```

This declares a variable called `x` but does not assign any value to it. You can also assign a value to a variable when you declare it, like this:

```
var x = 10;
```

In JavaScript, you can also use the `let` and `const` keywords to declare variables. The `let` keyword is used to declare a variable that can be reassigned later, while the `const` keyword is used to declare a variable that cannot be reassigned. For example:

```
let y = 20;const z = 30;
```

In this example, `y` is a variable that can be reassigned, while `z` is a constant that cannot be reassigned.

Data Types

In JavaScript, there are several data types that you can use to store different types of data. Some common data types include:

- Numbers (e.g. 10, 3.14)
- Strings (e.g. "hello", 'world')
- Booleans (e.g. true, false)
- Arrays (e.g. [1, 2, 3])
- Objects (e.g. { name: "John", age: 30 })
- **Variable Naming Rules**
- JavaScript is a dynamically-typed language, which means that you don't have to specify the data type of a variable when you declare it. The data type of a variable is determined by the value that is assigned to it. For example:

- `var x = 10; // x is a numbervar y = "hello"; // y is a stringvar z = [1, 2, 3]; // z is an array`

- **Variable Naming Rules**

- There are a few rules that you need to follow when naming variables in JavaScript:
- Variable names can only contain letters, digits, underscores, and dollar signs.
- Variable names cannot start with a digit.
- Variable names are case-sensitive.
- It is also a good practice to use descriptive and meaningful names for your variables, as this makes your code easier to read and understand.

- **Using Variables**

- Once you have declared a variable, you can use it to store and retrieve data in your program. For example:
- `var x = 10;console.log(x); // prints 10 x = "hello";console.log(x); // prints "hello"`
- You can also perform various operations on variables, such as mathematical calculations, string concatenation, and more. For example:
- `var x = 10;var y = 20;var z = x + y; // z is 30 var str1 = "hello";var str2 = "world";var str3 = str1 + " " + str2; // str3 is "hello world"`

Primitives and Objects

In JavaScript, there are two main types of data: primitives and objects.

Primitives

Primitives are the simplest and most basic data types in JavaScript. They include:

- Numbers (e.g. 10, 3.14)
- Strings (e.g. "hello", 'world')
- Booleans (e.g. true, false)

- Null (a special value that represents an absence of value)
- Undefined (a special value that represents an uninitialized variable)

Primitives are immutable, which means that once they are created, they cannot be changed. For example:

```
let x = 10;x = 20; // x is now 20
```

In this example, the value of "x" is changed from 10 to 20. However, this does not change the value of the primitive itself, but rather creates a new primitive with the value of 20.

Objects

Objects are more complex data types in JavaScript and are used to represent real-world objects or abstract concepts. They are composed of key-value pairs, where the keys are strings and the values can be any data type (including primitives and other objects).

Objects are mutable, which means that they can be changed after they are created. For example:

```
let obj = { name: "John", age: 30 };obj.age = 31; // the age property
of obj is now 31
```

In this example, the "age" property of the "obj" object is changed from 30 to 31. This changes the value of the object itself, rather than creating a new object.

There are several other data types in JavaScript that are classified as objects, including arrays, functions, and dates. These data types behave similarly to objects in that they are mutable and can be modified after they are created.

Operators and Expressions

Operators in JavaScript are symbols that perform specific operations on one or more operands (values or variables). For example, the addition operator (+) adds two operands together and the assignment operator (=) assigns a value to a variable.

There are several types of operators in JavaScript, including:

- Arithmetic operators (e.g. `+`, `-`, `*`, `/`, `%`)
- Comparison operators (e.g. `>`, `<`, `>=`, `<=`, `==`, `!=`)
- Logical operators (e.g. `&&`, `||`, `!`)
- Assignment operators (e.g. `=`, `+=`, `-=`, `*=`, `/=`)
- Conditional (ternary) operator (e.g. `?:`)

Expressions are combinations of values, variables, and operators that produce a result. For example:

```
let x = 10;let y = 20;let z = x + y; // z is 30
```

In this example, the expression `x + y` is evaluated to 30 and assigned to the `z` variable.

Operator precedence determines the order in which operators are applied when an expression has multiple operators. For example:

```
let x = 10 + 5 * 3; // x is 25
```

In this example, the multiplication operator (`*`) has a higher precedence than the addition operator (`+`), so the multiplication is performed before the addition. As a result, the expression is evaluated as `10 + (5 * 3) = 25`.

You can use parentheses to specify the order of operations in an expression. For example:

```
let x = (10 + 5) * 3; // x is 45
```

In this example, the parentheses indicate that the addition should be performed before the multiplication. As a result, the expression is evaluated as `(10 + 5) * 3 = 45`.

var

The `var` keyword is used to declare variables in JavaScript. It was introduced in the early days of the language and was the only way to declare variables for a long time. However, the `var` keyword has some limitations and has been largely replaced by the `let` and `const` keywords in modern JavaScript.

One of the main issues with `var` is that it is function-scoped, rather than block-scoped. This means that variables declared with `var` are accessible within the entire function in which they are declared, rather than just within the block of code in which they appear. This can lead to unexpected behavior and can make it difficult to reason about the scope of variables in your code.

let

The `let` keyword was introduced in ECMAScript 6 (also known as ES6) and is used to declare variables that can be reassigned later. `let` variables are block-scoped, which means that they are only accessible within the block of code in which they are declared. This makes them more predictable and easier to reason about than `var` variables.

For example:

```
if (x > 10) {  let y = 20;  console.log(y); //
20}console.log(y); // ReferenceError: y is not defined
```

In this example, the `y` variable is declared with the `let` keyword and is only accessible within the block of the if statement. If you try to access it outside of the block, you will get a `ReferenceError` because `y` is not defined in that scope.

const

The `const` keyword was also introduced in ES6 and is used to declare variables that cannot be reassigned later. `const` variables are also block-scoped and behave similarly to `let` variables in that respect. However, the main difference is that `const` variables must be initialized with a value when they are declared and cannot be reassigned later.

For example:

```
const PI = 3.14;PI = 3.14159; // TypeError: Assignment to constant variable.
```

In this example, the `PI` variable is declared with the `const` keyword and is assigned the value of 3.14. If you try to reassign a new value to `PI`, you will get a `TypeError` because `PI` is a constant variable and cannot be changed.

If else conditionals

The "if" statement in JavaScript is used to execute a block of code if a certain condition is met. The "else" clause is used to execute a block of code if the condition is not met.

Here is the basic syntax for an "if" statement:

```
if (condition) {  // code to be executed if condition is true}
```

Here is the syntax for an "if" statement with an "else" clause:

```
if (condition) {  // code to be executed if condition is true} else {  // code to be executed if condition is false}
```

The condition is a boolean expression that evaluates to either true or false. If the condition is true, the code in the "if" block is executed. If the condition is false, the code in the "else" block is executed (if present).

For example:

```
let x = 10;if (x > 5) {  console.log("x is greater than 5");} else {  console.log("x is not greater than 5");}
```

If else ladder

The "if-else ladder" is a control structure in JavaScript that allows you to execute a different block of code depending on multiple conditions. It is called a ladder because it consists of multiple "if" and "else" statements arranged in a ladder-like fashion.

Here is the syntax for an "if-else ladder":

```
if (condition1) {  // code to be executed if condition1 is true} else
if (condition2) {  // code to be executed if condition1 is false and
condition2 is true} else if (condition3) {  // code to be executed if
condition1 and condition2 are false and condition3 is true} ...else
{  // code to be executed if all conditions are false}
```

In this structure, each "if" statement is followed by an optional "else" statement. If the first "if" condition is true, the code in the corresponding block is executed and the rest of the ladder is skipped. If the first "if" condition is false, the second "if" condition is evaluated, and so on. If none of the conditions are true, the code in the "else" block is executed.

For example:

```
let x = 10;if (x > 15) {  console.log("x is greater than 15");} else
if (x > 10) {  console.log("x is greater than 10 but less than or
equal to 15");} else if (x > 5) {  console.log("x is greater than 5
but less than or equal to 10");} else {  console.log("x is less than
or equal to 5");}
```

Switch case

The "switch" statement in JavaScript is another control structure that allows you to execute a different block of code depending on a specific value. It is often used as an alternative to the "if-else ladder" when you have multiple conditions to check against a single value.

Here is the syntax for a "switch" statement:

```
switch (expression) {  case value1:    // code to be executed if expression == value1    break;  case value2:    // code to be executed if expression == value2    break;  ...  default:    // code to be executed if expression does not match any of the values}
```

In this structure, the "expression" is evaluated and compared to each of the "case" values. If the "expression" matches a "case" value, the corresponding block of code is executed. The "break" statement is used to exit the "switch" statement and prevent the code in the following cases from being executed. The "default" case is optional and is executed if the "expression" does not match any of the "case" values.

For example:

```
let x = "apple";switch (x) {  case "apple":    console.log("x is an apple");    break;  case "banana":    console.log("x is a banana");    break;  case "orange":    console.log("x is an orange");    break;  default:    console.log("x is something else");}
```

Ternary Operator

The ternary operator is a shorthand way to write an `if-else` statement in JavaScript. It takes the form of `condition ? value1 : value2`, where `condition` is a boolean expression, and `value1` and `value2` are expressions of any type. If `condition` is `true`, the ternary operator returns `value1`; if `condition` is `false`, it returns `value2`.

Here's an example of how you can use the ternary operator to assign a value to a variable based on a condition:

```
let x = 10;let y = 20;let max; max = (x > y) ? x : y;
console.log(max); // Outputs: 20
```

In this example, the ternary operator checks whether `x` is greater than `y`. If it is, `max` is assigned the value of `x`; otherwise, it is assigned the value of `y`.

For Loops

For loops are a common control flow structure in programming that allows you to repeat a block of code a specific number of times. In JavaScript, there are three types of for loops: the standard for loop, the for-in loop, and the for-of loop.

Standard for loop

The standard for loop has the following syntax:

```
for (initialization; condition; increment/decrement) {  // code to be
executed}
```

The `initialization` statement is executed before the loop starts and is typically used to initialize a counter variable. The `condition` is checked at the beginning of each iteration and if it is `true`, the loop continues. If it is `false`, the loop exits. The `increment/decrement` statement is executed at the end of each iteration and is used to update the counter variable.

Here's an example of a standard for loop that counts from 1 to 10:

```
for (let i = 1; i <= 10; i++) {  console.log(i);}
```

This loop will print the numbers 1 through 10 to the console.

For-in loop

The for-in loop is used to iterate over the properties of an object. It has the following syntax:

```
for (variable in object) {  // code to be executed}
```

The `variable` is assigned the name of each property in the object as the loop iterates over them.

Here's an example of a for-in loop that iterates over the properties of an object:

```
let person = {  name: "John",  age: 30,  job: "developer"}; for (let
key in person) {  console.log(key + ": " + person[key]);}
```

This loop will print the following to the console:

```
name: John
age: 30
job: developer
```

For-of loop

The for-of loop is used to iterate over the values of an iterable object, such as an array or a string. It has the following syntax:

```
for (variable of object) {  // code to be executed}
```

The `variable` is assigned the value of each element in the object as the loop iterates over them.

Here's an example of a for-of loop that iterates over the elements of an array:

```
let numbers = [1, 2, 3, 4, 5]; for (let number of numbers)
{  console.log(number);}
```

This loop will print the numbers 1 through 5 to the console.

While Loop

While loops are a control flow structure in programming that allow you to repeat a block of code while a certain condition is true. In JavaScript, the syntax for a while loop is:

```
while (condition) {  // code to be executed}
```

The `condition` is checked at the beginning of each iteration and if it is `true`, the code block is executed. If it is `false`, the loop exits.

Here's an example of a while loop that counts from 1 to 10:

```
let i = 1; while (i <= 10) {  console.log(i);  i++;}
```

This loop will print the numbers 1 through 10 to the console.

It's important to include a way to update the `condition` within the loop, otherwise it will become an infinite loop and will run forever. In the example above, the `i++` statement increments the value of `i` by 1 at the end of each iteration, which eventually causes the `condition` to be `false` and the loop to exit.

While loops can be useful when you don't know exactly how many times you need to execute a block of code. For example, you might use a while loop to keep prompting a user for input until they provide a valid response.

```
let input = ""; while (input !== "yes" && input !== "no") {  input = prompt("Please enter 'yes' or 'no':");}
```

This loop will keep prompting the user for input until they enter either "yes" or "no".

Functions

JavaScript functions are blocks of code that can be defined and executed whenever needed. They are a crucial part of JavaScript programming and are used to perform specific tasks or actions.

Functions are often referred to as "first-class citizens" in JavaScript because they can be treated like any other value, such as a number or a string. This means that they can be assigned to variables, passed as arguments to other functions, and returned as values from functions.

Here's the basic syntax for defining a function in JavaScript:

```
function functionName(parameters) {  // code to be executed}
```

The `functionName` is a unique identifier for the function, and the `parameters` are the variables that are passed to the function when it is called. These parameters act as placeholders for the actual values that are passed to the function when it is executed.

Here's an example of a simple function that takes a single parameter and returns the square of that number:

```
function square(x) {  return x * x;}
```

To call this function, you would simply use the function name followed by the arguments in parentheses:

```
let result = square(5);  // returns 25
```

Functions can also have multiple parameters, like this:

```
function add(x, y) {  return x + y;}
```

In this case, the `add` function takes two parameters, `x` and `y`, and returns their sum.

JavaScript also has a special type of function called an "arrow function," which uses a shorter syntax. Here's the same `square` function defined using an arrow function:

```
const square = (x) => {  return x * x;};
```

Arrow functions are often used when you want to create a small, one-line function that doesn't require a separate `function` keyword.

Functions can be defined inside other functions, which is known as "nesting." This is useful for creating smaller, reusable blocks of code that can be called from within the larger function.

```
function outerFunction(x) {  function innerFunction() {    // code to be executed  }  // more code}
```

In this example, the `innerFunction` is defined inside the `outerFunction` and can only be called from within that function.

Strings

One of the most important aspects of JavaScript is its ability to manipulate strings, which are sequences of characters. In this blog post, we will explore the basics of JavaScript strings and the various string methods that can be used to manipulate them.

A string in JavaScript is a sequence of characters enclosed in either single or double quotes. For example, the following are valid strings in JavaScript:

```
"Hello World"'Hello World'
```

JavaScript provides a number of built-in methods for manipulating strings. Some of the most commonly used string methods are:

length - This method returns the number of characters in a string. For example, the following code will return 11:

```
var str = "Hello World";console.log(str.length);
```

concat - This method is used to concatenate (combine) two or more strings. For example, the following code will return "Hello World":

```
var str1 = "Hello";var str2 = " World";console.log(str1.concat(str2));
```

indexOf - This method is used to find the index of a specific character or substring in a string. For example, the following code will return 6:

```
var str = "Hello World";console.log(str.indexOf("W"));
```

slice - This method is used to extract a portion of a string. For example, the following code will return "World":

```
var str = "Hello World";console.log(str.slice(6));
```

replace - This method is used to replace a specific character or substring in a string. For example, the following code will return "Hello Universe":

```
var str = "Hello World";console.log(str.replace("World", "Universe"));
```

toUpperCase and **toLowerCase** - These methods are used to convert a string to uppercase or lowercase letters. For example, the following code will return "HELLO WORLD" and "hello world" respectively:

```
var str = "Hello
World";console.log(str.toUpperCase());console.log(str.toLowerCase());
```

These are just a few of the many string methods available in JavaScript. By understanding the basics of strings and the various methods that can be used to manipulate them, you can create more dynamic and interactive web pages. So, start experimenting with different string methods and see what you can create!

Arrays and Array Methods

One of the most important data structures in JavaScript is the array, which is a collection of elements. In this blog post, we will explore the basics of JavaScript arrays and the various array methods that can be used to manipulate them.

An array in JavaScript is a collection of elements enclosed in square brackets. Elements can be of any data type, including numbers, strings, and other arrays. For example, the following is a valid array in JavaScript:

```
var myArray = [1, "Hello", [2, 3]];
```

JavaScript provides a number of built-in methods for manipulating arrays. Some of the most commonly used array methods are:

length - This method returns the number of elements in an array. For example, the following code will return 3:

```
var myArray = [1, "Hello", [2, 3]];console.log(myArray.length);
```

push - This method is used to add an element to the end of an array. For example, the following code will add the element "World" to the end of the array:

```
var myArray = [1, "Hello", [2, 3]];myArray.push("World");console.log(myArray); // [1, "Hello", [2, 3], "World"]
```

pop - This method is used to remove the last element of an array. For example, the following code will remove the last element ("World") from the array:

```
var myArray = [1, "Hello", [2, 3], "World"];myArray.pop();console.log(myArray); // [1, "Hello", [2, 3]]
```

shift - This method is used to remove the first element of an array. For example, the following code will remove the first element (1) from the array:

```
var myArray = [1, "Hello", [2, 3]];myArray.shift();console.log(myArray); // ["Hello", [2, 3]]
```

unshift - This method is used to add an element to the beginning of an array. For example, the following code will add the element 0 to the beginning of the array:

```
var myArray = [1, "Hello", [2, 3]];myArray.unshift(0);console.log(myArray); // [0, 1, "Hello", [2, 3]]
```

slice - This method is used to extract a portion of an array. For example, the following code will extract the elements from index 1 to 2 (exclusive):

```
var myArray = [1, "Hello", [2, 3]];console.log(myArray.slice(1, 2));
// ["Hello"]
```

splice - This method is used to add or remove elements from an array. For example, the following code will remove the element at index 1 and add the elements "Hello World" and [4, 5] at index 1:

```
var myArray = [1, "Hello", [2, 3]];myArray.splice(1, 1, "Hello World",
[4, 5]);console.log(myArray); // [1, "Hello World", [4, 5], [2, 3]]
```

Loops with Arrays

One of the most important data structures in JavaScript is the array, which is a collection of elements. When working with arrays, it is often necessary to iterate through each element in the array, which is where loops come in. In this blog post, we will explore how to use loops with arrays in JavaScript.

JavaScript provides several ways to iterate through an array, including the **for** loop, **forEach** method, and **for...of** loop.

for loop - This is the most basic way to iterate through an array. The for loop uses a counter variable that is incremented on each iteration. For example, the following code will print out each element in the array:

```
var myArray = [1, 2, 3, 4, 5];for (var i = 0; i < myArray.length; i++)
{    console.log(myArray[i]);}
```

forEach method - This method is a more concise way to iterate through an array. The forEach method takes a callback function as its argument, which is called on each element in the array. For example, the following code will print out each element in the array:

```
var myArray = [1, 2, 3, 4, 5];myArray.forEach(function(element)
{    console.log(element);});
```

for...of loop - This is a more recent addition to JavaScript, and it is the most concise way to iterate through an array. It allows you to iterate through the elements of an array without having to access the index, and it works with any iterable object, not just arrays. For example, the following code will print out each element in the array:

```
var myArray = [1, 2, 3, 4, 5];for (var element of myArray)
{    console.log(element);}
```

It is important to note that when you are iterating through an array using a for loop and you plan to change the array during iteration, you should use a for loop with a separate counter variable.