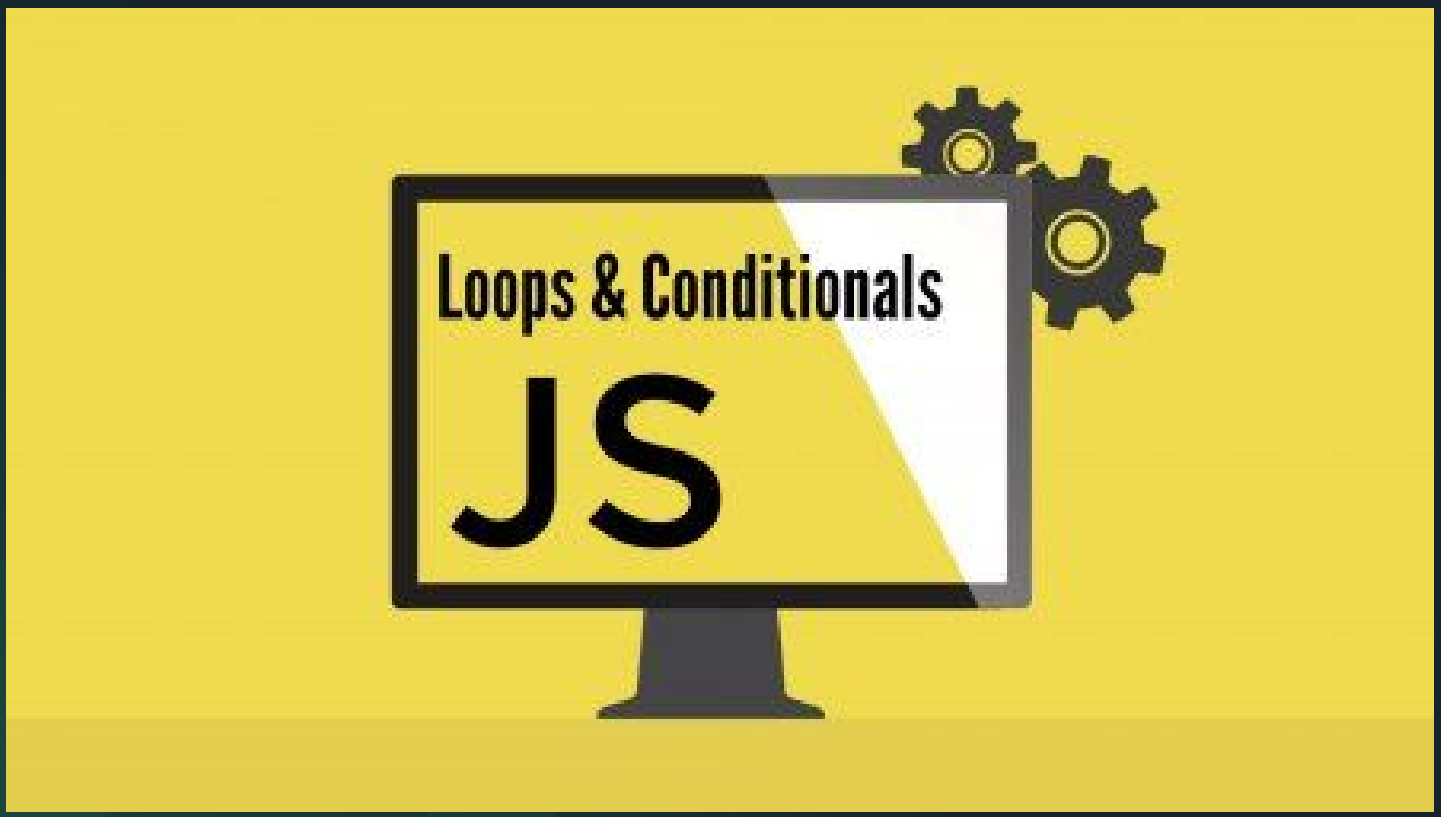
Conditionals and loops



Conditionals

Conditional statements are used for make decisions based on different conditions. By default , statements in JavaScript script executed sequentially from top to bottom. If the processing logic require so, the sequential flow of execution can be altered in two ways:

Conditionals contd..

- Conditional execution: a block of one or more statements will be executed if a certain expression is true

- Repetitive execution: a block of one or more statements will be repetitively executed as long as a certain expression is true. In this section, we will cover if, else , else if statements. The comparison and logical operators we learned in the previous sections will be useful in here.

Conditions can be implementing using the following ways:

- **if**
- **if else**
- **switch**
- **ternary operator**

If Condition

In JavaScript and other programming languages the key word if is to used check if a condition is true and to execute the block code. To create an if condition, we need if keyword, condition inside a parenthesis and block of code inside a curly bracket({}).

```
let num = 3;

if (num > 0) {

  console.log(`${num} is a positive

number`);

}

//  3 is a positive number
```

If Else

If condition is true the first block will be executed, if not the else condition will be executed.

```
let num = 3;

if (num > 0) {

  console.log(`${num} is a positive number`);

} else {

  console.log(`${num} is a negative number`);

}

//  3 is a positive number
```

Switch

- Switch is an alternative for if else if else else. The switch statement starts with a switch keyword followed by a parenthesis and code block. Inside the code block we will have different cases. Case block runs if the value in the switch statement parenthesis matches with the case value

- The break statement is to terminate execution so the code execution does not go down after the condition is satisfied. The default block runs if all the cases don't satisfy the condition.

Switch Example

```
let num = prompt("Enter number");

switch (true) {
  case num > 0:
    console.log("Number is positive");
    break;
  case num == 0:
    console.log("Numbers is zero");
    break;
  case num < 0:
    console.log("Number is negative");
    break;
  default:
    console.log("Entered value was not a number");
}
```

Ternary Operator:

In JavaScript, the ternary operator is a shorthand way of writing an if-else statement. It allows you to make a quick decision between two values based on a condition. The syntax of the ternary operator is as follows:

```
condition ? expressionIfTrue : expressionIfFalse;
```

Here's how it works:

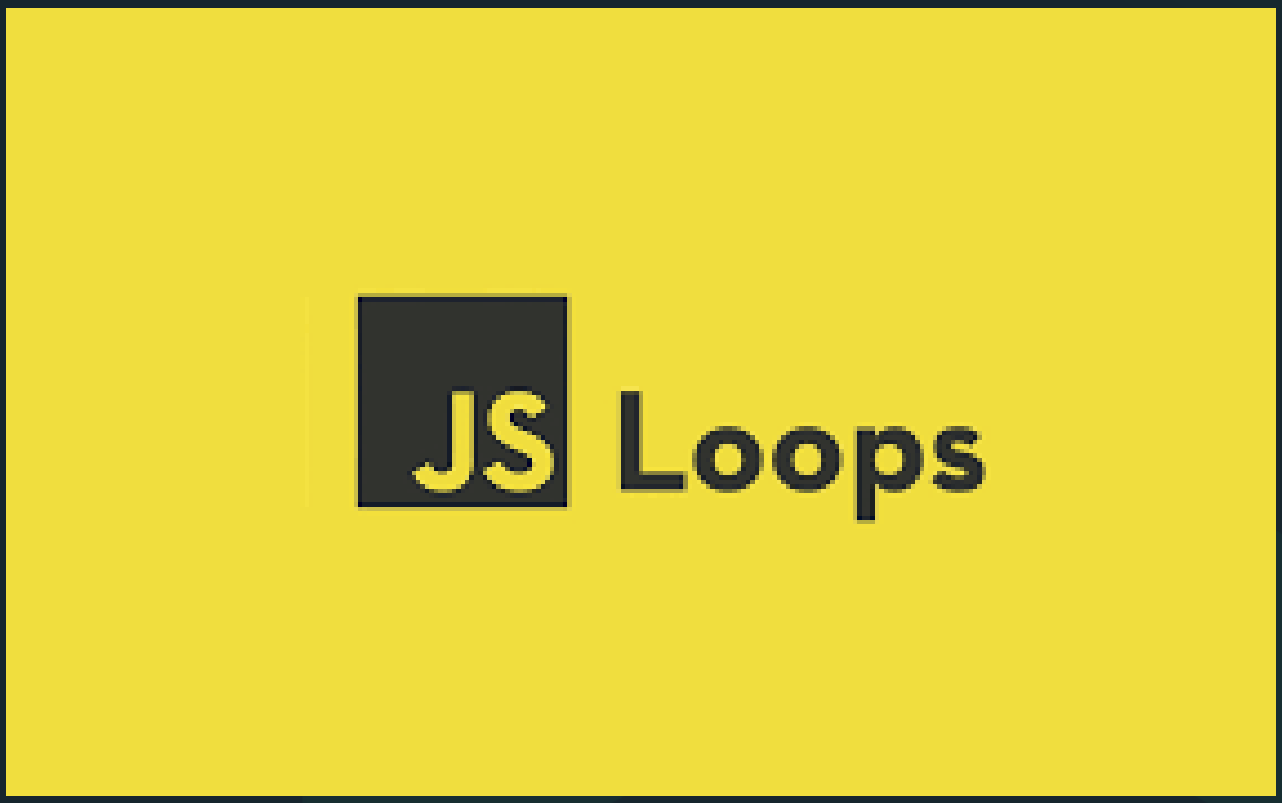
- The condition is an expression that evaluates to either true or false.
- If the condition is true, the value of expressionIfTrue is returned.
- If the condition is false, the value of expressionIfFalse is returned.

Example:

```
const age = 25;
const message = age >= 18 ? "You are an adult." : "You
are a minor.";
console.log(message); // Output: "You are an adult."
```

In this example, the condition age >= 18 is evaluated. Since age is 25, which is greater than or equal to 18, the expression "You are an adult." is assigned to the message variable.

Loops



loops are used to execute a block of code repeatedly until a certain condition is met. There are three types of loops in JavaScript: **for**, **while**, and **do-while**.

For loop

The for loop is the most common loop used in JavaScript. It consists of three parts:

Initialization of a variable (usually an index variable) to a starting value A condition that is checked before each iteration of the loop An update to the index variable at the end of each iteration The syntax for a for loop is as follows:

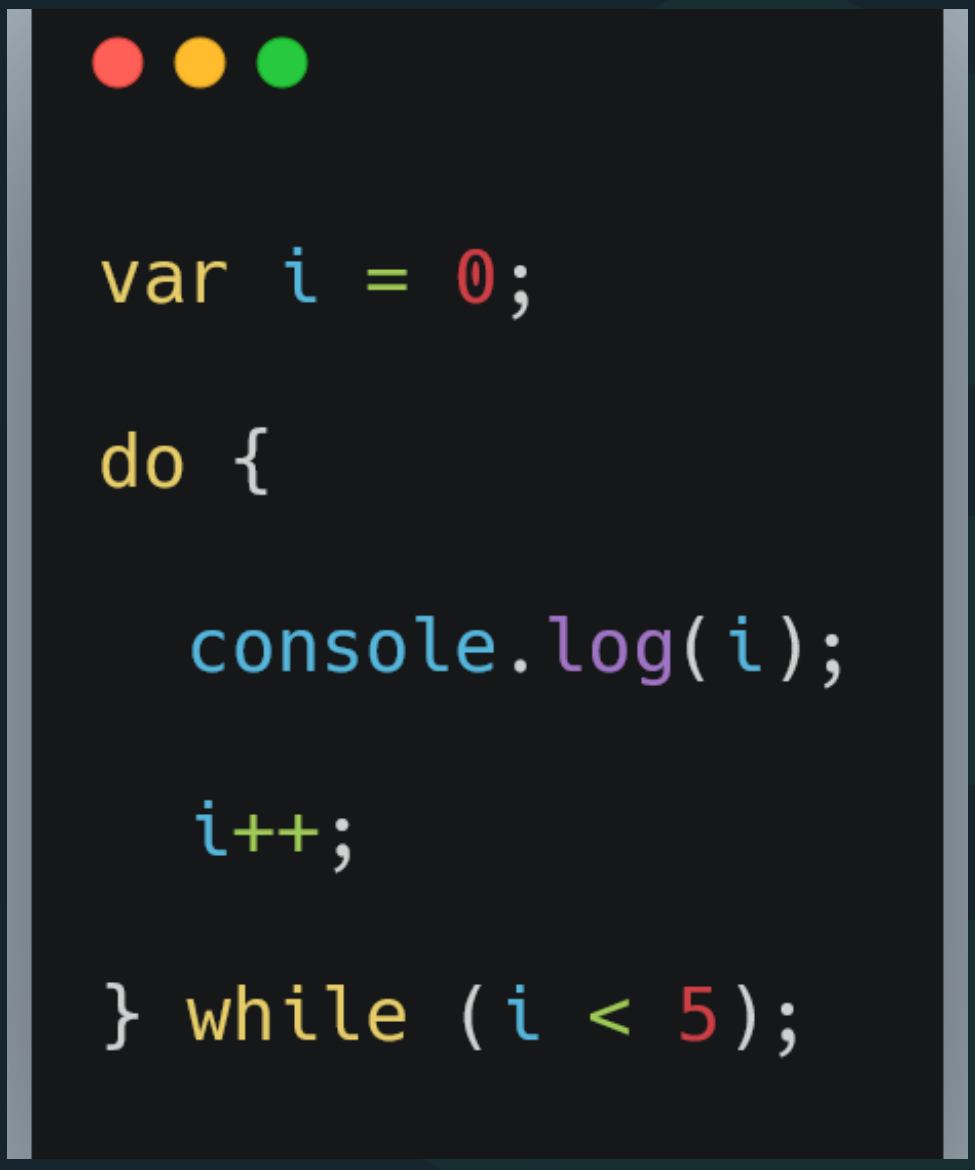
while loop

The while loop is used when the number of iterations is unknown, and the loop continues as long as a specified condition is true. The syntax for a while loop is as follows:

```
var i = 0;

while (i < 5) {

  console.log(i);

  i++;

}
```

do-while loop

The do-while loop is similar to the while loop, but it executes the code block at least once before checking the condition. The syntax for a do-while loop is as follows:



Why Should we need function?

Imagine you have a specific task you need to perform multiple times in your program. It could be something like calculating the area of different shapes, converting units, processing data in a similar way, or solving equations repeatedly.

Without functions, you would have to write the same set of instructions or calculations every time you encounter that task. This not only makes your code longer and harder to read but also increases the chance of errors. Additionally, if you need to make a change to how that task is performed, you'd have to go through every occurrence in the code and update it.

With functions, you can package those instructions or calculations into a single unit. You give that unit a name (the function name) and define how it should operate. Whenever you need to perform that task, you simply call the function by its name. This way, you avoid repeating the same code, and your program becomes more organized and efficient.

Functions

A function is a reusable block of code or programming statements designed to perform a certain task.

Ways to declare function:

Function Declaration:

The function declaration defines a named function with the function keyword, followed by the function name, a list of parameters in parentheses, and the function body enclosed in curly braces. Function declarations are hoisted, which means they can be used before the declaration in the code.

```
function functionName(parameter1, parameter2)
{ // Function body
  // ...
}
```

Function Expression:

A function expression is similar to a function declaration, but it involves assigning a function to a variable. The function is defined anonymously (without a name) or with a name.

Anonymous Function Expression:

```
const anonymousFunction = function(parameter1, parameter2)
{ // Function body
  // ...
};
```

Named Function Expression:

```
const namedFunction = function functionName(parameter1, parameter2)
{
  // Function body
  // ...
};
```

Arrow Function Expression:

A function expression is similar to a function declaration, but it involves assigning a function to a variable. The function is defined anonymously (without a name) or with a name.

```
const arrowFunction = (parameter1, parameter2) => {
  // Function body
  // ...
};
```

Function Scope

In JavaScript, functions have their own scope, which means that variables declared inside a function are only accessible within that function (and any nested functions). This concept is known as "function scope." Variables declared outside of any function (i.e., in the global scope) are accessible throughout the entire script.

```
function myFunction() {
  // Variable declared inside the function
  const localVar = "I am a local variable";
  console.log(localVar);
}

myFunction(); // Output: "I am a local variable"

// Attempting to access the local variable outside the function will
result in an error
// console.log(localVar); // This would throw a ReferenceError
```

Function Scope

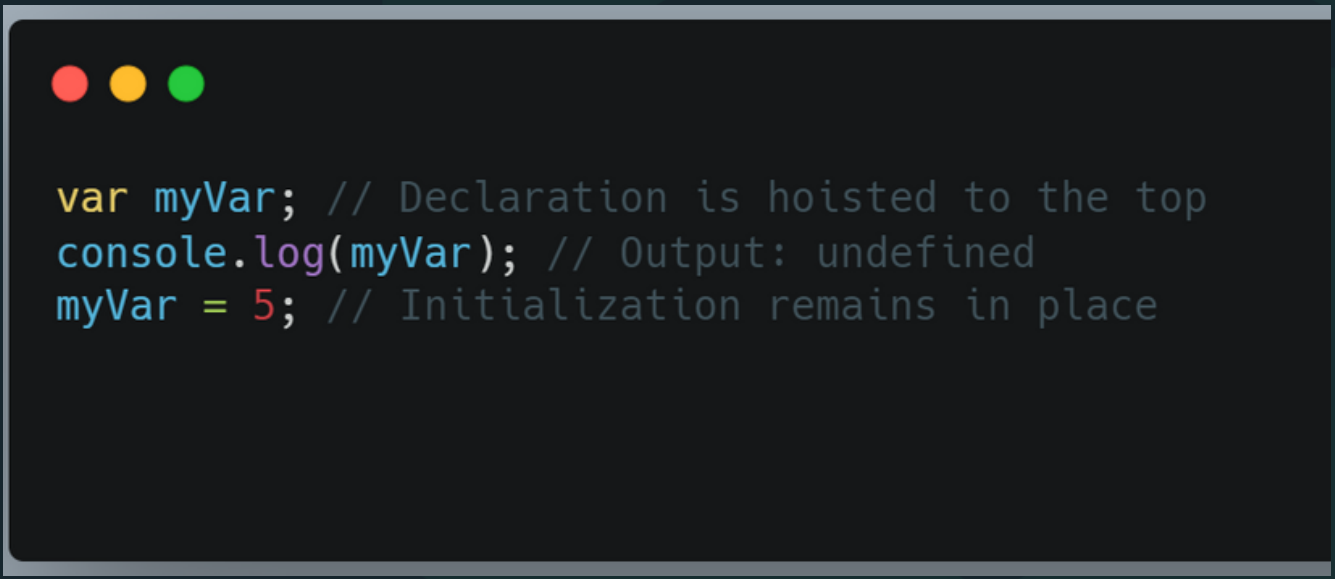
variables declared outside any function (in the global scope) are accessible from any part of the script, including inside functions:

```
const globalVar = "I am a global variable";

function myFunction() {
  console.log(globalVar); // Accessing the global variable inside the function
}

myFunction(); // Output: "I am a global variable"

console.log(globalVar); // Output: "I am a global variable"
```

In this example, globalVar is declared outside any function, making it a global variable. It can be accessed both inside the myFunction() and outside the function without any issues.

Hoisting

Hoisting is a JavaScript behavior where variable declarations and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed. This means that you can use variables and functions before they are actually declared in the code, which might seem counterintuitive at first.



```
var myVar; // Declaration is hoisted to the top
console.log(myVar); // Output: undefined
myVar = 5; // Initialization remains in place
```