Introduction to JavaScript (continuation)

URL:https://codefinity.com/courses/v2/2cd68d01-3aa7-4429-969c-7eea851507a7/d58c77b6-5684-4962-84c3-f763b181394c/462e1f19-4be6-4530-935c-e8407cb72f5e

Lecture plan:
- Conditional Statements
- Mastering Functions
- Discovering Loops

1. Nested Conditional Statements

We can have conditional statements inside other conditional statements. This lets us perform multiple checks before executing some code.

We can check if a number is in a specific range using this kind of an arrangement:

```
let number = 17;

if(number >= 10)
{
  if(number <= 20)
  {
    console.log("The number is in the range 10-20");
  }
}
```

It's generally recommended to avoid nesting many conditional statements, as it makes the code harder to read and manage.

There are better methods for performing multiple checks, which we will explore in the upcoming chapters.

2. The `else` Clause

Conditional Statements have an optional else clause which can be used to specify a block of code to be executed in case the **boolean expression** evaluates to false.

The syntax is as follows:

```
if(boolean_expression)
{
  // Code to execute if boolean_expression is true
}
else
{
  // Code to execute if boolean_expression is false
}
```

This way we don't need to write multiple **if-statements** to check for opposite conditions, making the code shorter and neater.

```
let age = 18;

if(age >= 18) {
  console.log("The user is old enough to drive.");
} else {
  console.log("The user is not old enough to drive");
}
```

3. The `else if` Clause

In addition to the else clause, conditional statements support an else if clause, which can be used to define alternative conditions if the initial if condition is false.

The general syntax is as follows:

```
if(expression) {
  // Code … (executed if the expression is true)
} else if(expression) {
  // Fallback Code …
  //(executed if the previous condition is false, and this one is true)
}
```

As seen in the general syntax, the else if clause takes a boolean expression, which is evaluated when the condition before it turns out to be false.

We can chain multiple else if clauses to form an if-else if chain:

```
if(expression) {
  // … (executed if the first condition is true)
} else if(expression) {
  // … (executed if the first condition is false and this is true)
} else if(expression) {
  // … (executed if previous conditions are false and this is true)
} else {
  // … (executed if all previous conditions are false)
}
```

As shown in the code above, we can optionally add the else clause at the end. This block is executed only when all the previous conditions evaluate to false.

The following example demonstrates the usage of this syntax:

```
let number = 50;

if (number < 20) {
  console.log("The number is less than 20.");
} else if (number === 20) {
  console.log("The number is exactly 20.");
```

```
} else {
  console.log("The number is greater than 20.");
}
```

4. ` AND` Logical Operator

Definition. **Logical operators** allow us to combine multiple **boolean expressions** into a single, more complex condition.

The **AND operator** (&&) is a **logical operator**, which joins two or more expressions and returns true only if all the expressions evaluate to true.

For example, we can combine the conditions a >= 10 and a <= 20 using the && operator:

```
a >= 10 && a <= 20
```

This expression means: "**a is greater than or equal to 10 AND a is less than or equal to 20.**"

5 OR Logical Operator

The || (OR) operator is used for defining the "**or**" relation between two or more expressions.

It returns true if **any** of the expressions evaluate to true.

Example:

```
let a = 9;

if(a < 10 || a > 30) {
  console.log("Condition Matched");
}
```

The above code output "Condition Matched" when a is less than 10 **OR** a is greater than 30.

6. Mastering Functions

A powerful programming feature called a function allows the execution of a block of code multiple times with minimal repetition. The section introduces core concepts and syntax for defining and using functions effectively.

What Are Functions?

Definition. **Functions** are a feature in programming that lets us **reserve** a block of code to be executed later. This also enables us to execute that block of code multiple times with ease.

The basic syntax of **defining** a function is as follows:

```
function funcName() {
```

```
  // Code here
}
```

Here function is the **keyword** used to define a function, and funcName represents the name of the function that we want to create.

Creating a function is more accurately referred to as "**defining**" a function. The code which defines a function is referred to as the "**function definition**" code.

Note. The **DRY (Don't Repeat Yourself)** principle is a core programming concept that emphasizes minimizing code duplication. It encourages writing each piece of logic once and reusing it whenever necessary. This improves the code's **readability** and **efficiency**. Functions play an important role in adhering to this principle, as they allow us to eliminate any redundant code.

Following is an example of a function which draws a triangle in the console:

```
function drawTriangle() {
  console.log("*");
  console.log("* *");
  console.log("* * *");
  console.log("* * * *");
  console.log("* * * * *");
}

drawTriangle();
```

It's possible to **execute** a function more than once:

```
function drawTriangle() {
  console.log("*");
  console.log("* *");
  console.log("* * *");
  console.log("* * * *");
  console.log("* * * * *");
}

drawTriangle();
drawTriangle();
drawTriangle();
```

Note. **Executing** a function is also sometimes referred to as **calling** a function. Similarly, a statement that executes a function (for example: myFunc()) is referred to as a **Function Call**.

It is recommended to name the functions meaningfully such that the name of the function accurately reflects the operation it performs.

7. Scopes

Definition. A Scope is simply an area in the code where a variable can be accessed or used.

There are two types of scopes:

1. **Global Scope**;
2. **Local Scope**.

If a variable is defined inside a block of code (between curly brackets {}), it is said to have a **local scope**. This means that it can only be accessed from **inside** that function or code block, or any **nested** blocks:

```
function exampleFunc() {
  let exampleVariable = 10;

  console.log(exampleVariable); // Valid

  if(10 + 10 == 20) {
    console.log(exampleVariable); // Valid
  }
}

exampleFunc();
console.log(exampleVariable); // Shows error
```

A variable that is defined outside of any code block is said to have a **Global Scope**, and it can be accessed from anywhere:

```
let exampleVariable = 10;

function exampleFunc() {
  console.log(exampleVariable); // Valid

  if(10 + 10 == 20) {
    console.log(exampleVariable); // Valid
  }
}

exampleFunc();
console.log(exampleVariable); // Valid
```

A variable defined in a lower (nested) scope cannot be accessed from a higher (parent) scope:

```
function exampleFunc() {
  if(10 + 10 == 20) {
    let exampleVariable = 10;
```

```
    console.log(exampleVariable); // Valid
  }

  console.log(exampleVariable); // Shows error
}

exampleFunc();
console.log(exampleVariable); // Shows error
```

8. Passing Data into Functions

We can pass data into functions using **parameters**:

```
function funcName(parameter1, parameter2, ...) {

  // your code here…
}
```

A **parameter** acts like a **local variable** inside the function. However, it takes up the value of the corresponding **argument** that is passed into the function at the **function call**.

```
function sum(a, b) {
  console.log(a + b);
}

sum(1, 2); // Output: 3
sum(5, 10); // Output: 15
sum(20, 30); // Output: 50
sum(33, 37); // Output: 70
```

- **An argument** is the actual value passed into a function when calling it:
  - Example: In `sum(1, 2)`, the values `1` and `2` are arguments;
- **A parameter** is a placeholder inside the function definition that receives the argument's value:
  - Example: In `function sum(a, b)`, `a` and `b` are parameters.

Study More. A function in which other functions are **called** is known as a **compound function**. A **compound function** combines smaller functions to accomplish a larger task.

9. Returning Data from Functions

We can **return** any kind of value from a function using a `return` statement.

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

```
let result = sum(10, 15);
console.log(result); // Output: 25
```

General Syntax. The general syntax of a return statement is

```
return <value>;
```

Where `<value>` is optional. If no value is provided, the function returns `undefined`:

```
function exampleFunc() {
  console.log("In the Function");
  return;
}

let result = exampleFunc();
console.log("Return value if nothing is returned:", result); // Output: undefined
```

How It Works?

The return statement **stops the execution** of the function, and returns to the point in code the function where it was called. Therefore, any code after `return` is ignored:

```
function exampleFunc() {
  console.log(1);
  console.log(2);
  return true; // Execution stops here
  console.log(4); // Ignored
  console.log(5); // Ignored
}

console.log("Before Function Call");
console.log(exampleFunc()); // Output: true
console.log("After Function Call");
```

10. Default Values

Definition. Default values are values that are taken up by the parameters of a function when no corresponding argument is passed into the function call.

We can assign **default values** to parameters using the syntax:

```
function funcName(parameter1 = value1, parameter2 = value2, …) {
  // your code here …
}
```

Example:

```
function sum(a = 5, b = 10) {
  return a + b;
}
```

```
console.log(sum()); // Output: 15
```

11. The `for` Loop

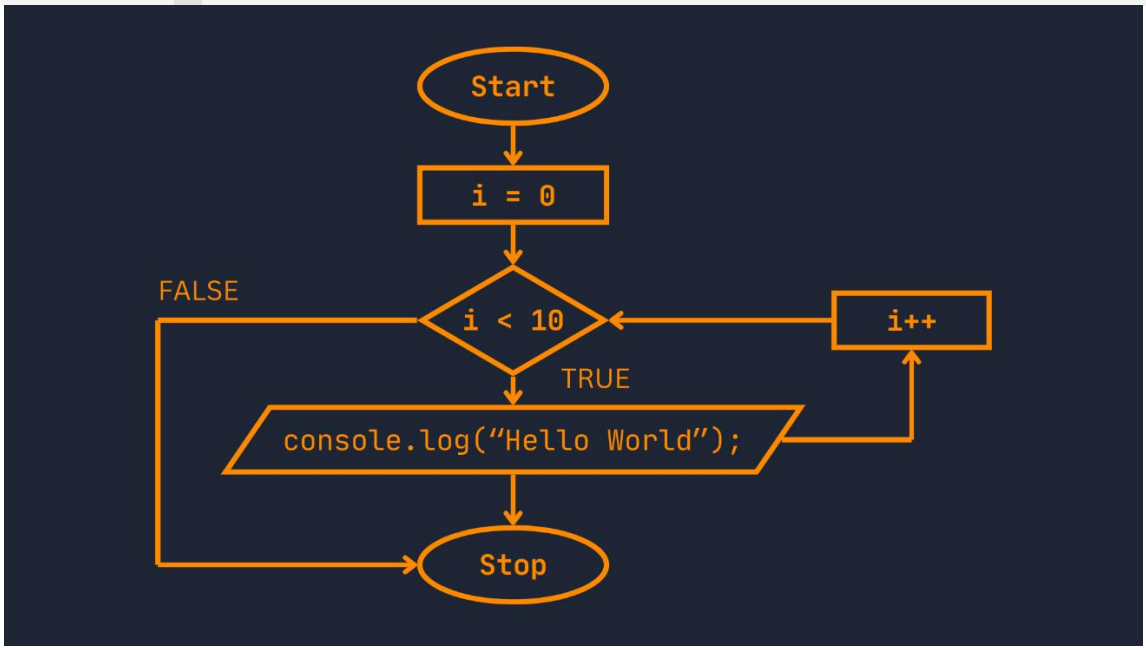
One of the most popular loops, which is available in almost every programming language, is known as the `for` loop. The `for` loop is primarily used for executing a block of code a **specific number of times**.

The general syntax of the `for` loop is as follows:

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

The `initialization` part is where we declare & initialize a variable, this part is executed only once - at the start of the loop. The **boolean expression** in place of the `condition` defines the **loop condition** - the loop executes as long as that condition is `true`. The `operation` can be any expression which is evaluated or executed after every successful iteration.

The flow of a `for` loop can be better understood through a flowchart:



Following is an example of a simple `for` loop which executes a block of code 15 times:

```
for(let i = 0; i < 15; i++) {
  console.log("This is the " + i + "th iteration");
}
```

12. The `while` Loop

The while loop is another kind of loop supported by most programming languages **including JavaScript**.

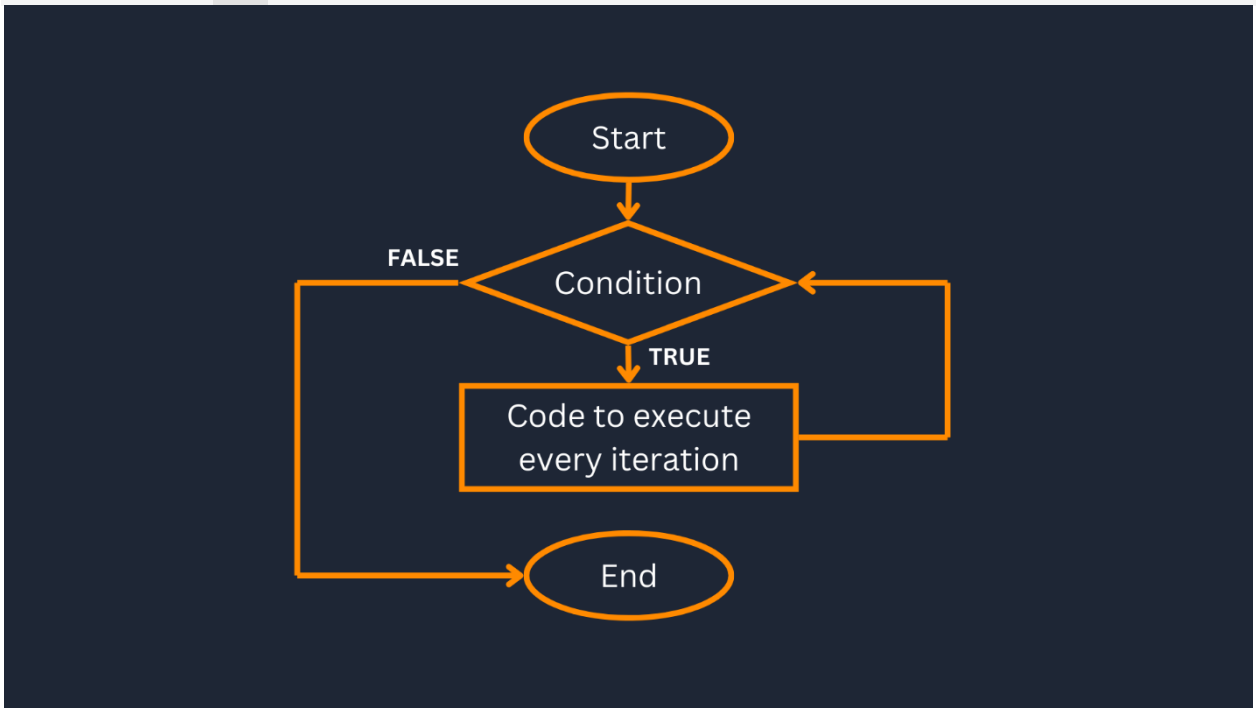
The while loop is mainly used when we want to execute a block of code as long as a condition is true. Although it may behave similar to a for loop in certain circumstances, it is mainly used in cases where we don't know exactly how many times a block of code needs to be executed.

The general syntax of a while loop is:

```
while(boolean_expression) {
  // code to execute
}
```

Note. If the **condition** of a while loop is always true, it will execute forever. Such a loop is known as **an infinite loop**.

The process of a while loop can be better understood by looking at it's flowchart:



Following is an example program which utilizes a while loop to find the first number that is divisible by both 11 and 12:

```
let i = 13;
while(i % 11 != 0 && i % 12 != 0) {
  i += 1;
}

console.log("The first number divisible by both 11 and 12 is:", i);
```

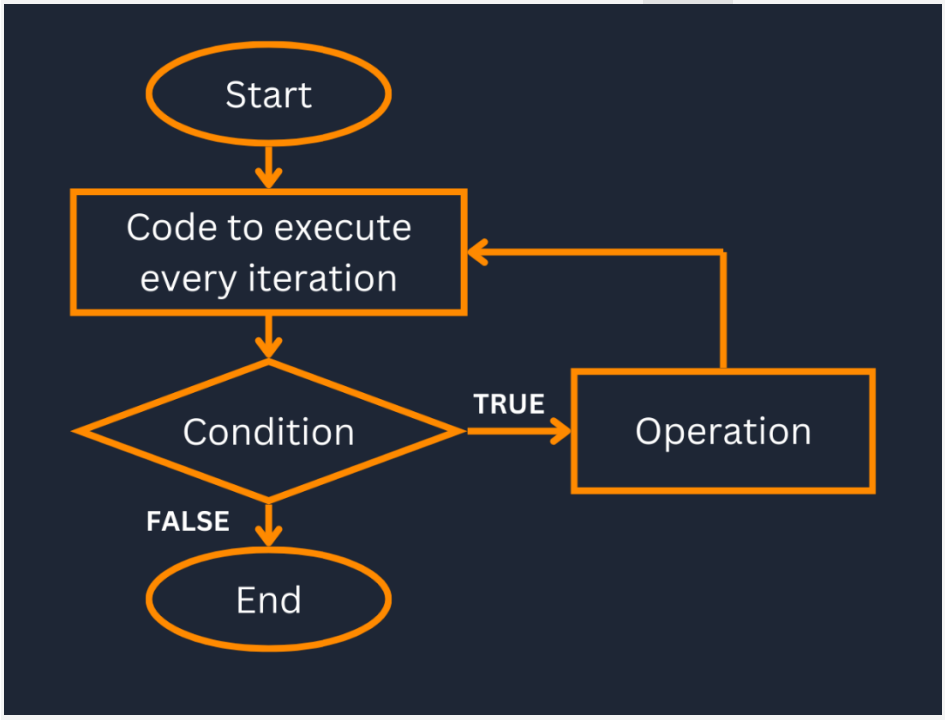
13 The `do-while` Loop

The do-while is very similar to a while loop, except it is always **executed at least once**, even if the **loop condition** is false.
Another difference is that the code block is executed **before** the loop condition is checked.

The general syntax of a do-while loop is the following:

```
do {
  // code to execute
} while(boolean_expression);
```

The flowchart describes the execution process of a do-while loop:



For example, following is a program which uses a do-while loop to print the first ten even numbers:

```
let i = 1;

do {
  console.log(i * 2);
  i += 1;
} while (i <= 10);
```

Even if we change the value of i, such that the condition becomes false, the code block will still execute at least once:

```
let i = 11;

do {
  console.log(i * 2);
  i += 1;
} while (i <= 10);
```