**How Can You Test if a String Contains a Substring?**

When working with strings in JavaScript, there are many cases where you might need to check whether a string contains a specific substring, which is a smaller part of that string.

For example, you might want to check if a user's input includes a specific word or character before performing some action. One way to achieve this is by using the includes() method.

The includes() method is used to check if a string contains a specific substring. If the substring is found within the string, the method returns true otherwise, it returns false.

Here's the basic syntax:

string.includes(searchValue);

For the syntax, the searchValue is the substring you want to look for within the string. And here's an example:

let phrase = "JavaScript is awesome!";

let result = phrase.includes("awesome");

console.log(result); // true

In this example, the word awesome is found within the string JavaScript is awesome!, so the includes() method returns true.

It's important to note that the includes() method is case-sensitive. This means that the exact match of the characters is required, including their case.

For example:

let phrase = "JavaScript is awesome!";

let result = phrase.includes("Awesome");

console.log(result); // Output: false

Since Awesome (with an uppercase A) does not match awesome (with a lowercase a), the result is false.

You can also use the includes() method to check for a substring starting at a specific index in the string by providing a second parameter:

let text = "Hello, JavaScript world!";

let result = text.includes("JavaScript", 7);

console.log(result); // true

Here, the search for the substring JavaScript starts from the 7th position in the string, ensuring it skips any characters before this position.

The includes() method only returns a true or false result. It does not provide information on where the substring is located in the string or how many times it occurs. If you need that level of detail, other methods, such as the indexOf() method might be more suitable.

**How Can You Extract a Substring from a String?**

When working with strings in JavaScript, you often need to extract a portion or substring from a larger string.

For example, you may want to extract part of a word, a specific character sequence, or just a fragment of a sentence.

JavaScript provides several methods for this task, one of the most commonly used being the slice() method.

The slice() method allows you to extract a portion of a string and returns a new string, without modifying the original string. It takes two parameters: the starting index and the optional ending index.

Here's the basic syntax:

string.slice(startIndex, endIndex);

startIndex is the position where the extraction starts. endIndex is where the extraction ends. If not provided, slice() extracts until the end of the string.

Let's look at a simple example of extracting part of a string:

let message = "Hello, world!";

let greeting = message.slice(0, 5);

console.log(greeting); // Output: Hello

In this example, slice(0, 5) extracts characters starting from index 0 up to but not including index 5. As a result, the word Hello is extracted.

If you omit the second parameter, slice() will extract everything from the start index to the end of the string:

let message = "Hello, world!";

let world = message.slice(7);

console.log(world); // Output: world!

Here, slice(7) extracts the string from index 7 to the end of the string, resulting in world!.

You can also use negative numbers as indexes. When you use a negative number, it counts backward from the end of the string:

let message = "JavaScript is fun!";

let lastWord = message.slice(-4);

console.log(lastWord); // Output: fun!

In this case, slice(-4) extracts the last four characters from the string, giving us fun!.

Let's say you want to extract a section from the middle of a string. You can provide both the starting and ending indexes to precisely control which part of the string you want:

let message = "I love JavaScript!";

let language = message.slice(7, 17);

console.log(language); // Output: JavaScript

Here, slice(7, 17) extracts the substring starting at index 7 and ending right before index 17, which is the word JavaScript.

The slice() method is a powerful tool for extracting parts of a string in JavaScript.

You specify the start and end indexes, and the method returns a new string that contains the extracted portion.

With options for positive, negative, and omitted indexes, you can adapt it to various situations without altering the original string.

**What Are Unary Operators, and How Do They Work?**

Unary operators act on a single operand to perform operations like type conversion, value manipulation, or checking certain conditions. Let's look at some common unary operators and how they work.

The unary plus operator converts its operand into a number. If the operand is already a number, it remains unchanged.

const str = '42';

const strToNum = +str;

console.log(strToNum); // 42

console.log(typeof str); // string

console.log(typeof strToNum); // number

Unary plus is handy when you want to make sure you're working with a numeric value. As you might guess, there's a unary negation operator. It negates the value of the operand. It works similarly to the unary plus, except it flips the sign.

const str = '42';

const strToNegativeNum = -str;

console.log(strToNegativeNum); // -42

console.log(typeof str); // string

console.log(typeof strToNegativeNum); // number

The logical NOT operator, represented by an exclamation mark (!), is another unary operator. It flips the boolean value of its operand. So, if the operand is true, it becomes false, and if it's false, it becomes true.

let isOnline = true;

console.log(!isOnline); // false

let isOffline = false;

console.log(!isOffline); // true

The bitwise NOT operator is a less commonly used unary operator. Represented by a tilde, ~, it inverts the binary representation of a number. Computers store numbers in binary format (1s and 0s). The ~ operator flips every bit, meaning it changes all 1s to 0s and all 0s to 1s. You will learn more about binary and bits in a future lesson.

const num = 5; // The binary for 5 is 00000101

console.log(~num); // -6

In this example, 5 became -6 because by applying the ~ operator to 5, you get - (5 + 1), which equals -6 due to two's complement representation. Two's complement is a way computers represent negative numbers in binary. You probably won't use the bitwise NOT often unless you're working with low-level programming tasks like manipulating bits directly.

The void keyword is a unary operator that evaluates an expression and returns undefined.

const result = void (2 + 2);

console.log(result); // undefined

void is also commonly used in hyperlinks to prevent navigation:

<a href="javascript:void(0);">Click Me</a>

Finally, there is the typeof operator which you learned about in previous lessons. This returns the type of its operand as a string.

const value = 'Hello world';

console.log(typeof value); // string

**What Are Bitwise Operators, and How Do They Work?**

Bitwise operators in JavaScript are special operators that work on the binary representations of numbers. To understand bitwise operators, we first need to grasp the concept of bits and binary numbers. In computing, a bit is the most basic unit of information. It can have only two values: 0 or 1. Binary is a number system that uses only these two digits to represent all numbers.

For example, the binary representation of the decimal number 10 is 1010. In this system, each digit represents a power of 2, starting from the rightmost digit and increasing as we move left.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 1 | 0 |
| 1 · 23 | 0 · 22 | 1 · 21 | 0 · 20 |
| 8 | 0 | 2 | 0 |

In the table above, the first row shows the binary number 1010, the second row shows the power of 2 represented by each binary position, and the third row shows the result of each multiplication. If you add all the values in the third row, they total 10.

Now, let's dive into bitwise operators. These operators perform operations on the binary representations of numbers. JavaScript provides several bitwise operators, including AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>).

The bitwise AND (&) operator returns a 1 in each bit position for which the corresponding bits of both operands are 1. Here's an example:

let a = 5; // Binary: 101

let b = 3; // Binary: 011

console.log(a & b); // Output: 1 (Binary: 001)

In this example, we perform a bitwise AND operation on 5 (101 in binary) and 3 (011 in binary). The result is 1 (001 in binary) because only the rightmost bit is 1 in both numbers.

The bitwise OR (|) operator returns a 1 in each bit position for which the corresponding bits of either or both operands are 1. For example:

let a = 5; // Binary: 101

let b = 3; // Binary: 011

console.log(a | b); // Output: 7 (Binary: 111)

Here, the result is 7 (111 in binary) because at least one of the bits is 1 in each position.

The bitwise XOR (^) operator returns a 1 in each bit position for which the corresponding bits of either, but not both, operands are 1. For instance:

let a = 5; // Binary: 101

let b = 3; // Binary: 011

console.log(a ^ b); // Output: 6 (Binary: 110)

The result is 6 (110 in binary) because the first and second bits from the right are different in the two numbers.

The bitwise NOT (~) operator inverts all the bits of its operand. For example:

let a = 5; // Binary: 101

console.log(~a); // Output: -6

This might seem surprising, but it's because of how negative numbers are represented in binary using two's complement.

The left shift (<<) operator shifts all bits to the left by a specified number of positions. For example:

let a = 5; // Binary: 101

console.log(a << 1); // Output: 10 (Binary: 1010)

Here, all bits are shifted one position to the left, effectively multiplying the number by 2.

The right shift (>>) operator shifts all bits to the right. For example:

let a = 5; // Binary: 101

console.log(a >> 1); // Output: 2 (Binary: 10)

Here, all bits are shifted one position to the right, effectively dividing the number by 2 and rounding down.

Bitwise operators are often used in low-level programming and cryptography. While they may not be as commonly used in everyday JavaScript programming, understanding them can be beneficial for certain specialized tasks and can deepen your understanding of how computers work at a fundamental level.

**What Are Conditional Statements, and How Do If/Else If/Else Statements Work?**

Conditional statements let you make decisions in your JavaScript code. They allow your program to flow in a particular way based on certain conditions. Let's take a look at how if, else if, else, and the ternary operator work to let you control the flow of your code.

An if statement takes a condition and runs a block of code if that condition is truthy. Truthy values are any values that result in true when evaluated in a Boolean context like an if statement. Here are examples of truthy values:

* non-empty strings, for example, hello
* any number other than 0 and -0, for example, 4, -5, and others
* arrays
* objects
* the boolean true

On the other hand, falsy values are values that evaluate to false in a boolean context. JavaScript has few falsy values, which makes them easy to remember. Here are a few falsy values:

* boolean false
* 0 (zero)
* "" (empty string)
* null
* undefined
* NaN (Not a Number)

Now, that we have a basic understanding of truthy and falsy values, let's see how it works with if statements. In this first example, we are using a couple of if statements to check against truthy and falsy values:

if (null) {

console.log("This will not run.");

}

if ("freeCodeCamp") {

console.log("This will run.");

}

Since null is a falsy value, the message inside the block will never be logged to the console. But for the second if statement, the string freeCodeCamp is a truthy value, and will be considered true in this boolean context of the if statement. As a result, the message This will run. will be logged to the console.

Let's take a look at a few more examples on how if statements work with different comparison operators. Here is an example of using an if statement to check if the user is eligible to vote:

const age = 22;

if (age >= 18) {

console.log("You're eligible to vote"); // You're eligible to vote

}

In this example, since age is currently 22, this means the condition will evaluate to true because 22 is greater than or equal to 18. So the message You're eligible to vote will be logged to the console. If we change the example so age is now 15, then the condition will evaluate to false and the message will not be logged to the console:

const age = 15;

if (age >= 18) {

console.log("You're eligible to vote"); // Code not running because age is less than 18

}

When a condition is false, then you can use an else clause:

const age = 15;

if (age >= 18) {

console.log("You're eligible to vote");

} else {

console.log("You're not eligible to vote"); // You're not eligible to vote

}

In this example, 15 is not greater than or equal to 18, so the condition would be false. The code inside the else block will run in this case.

If you want to check multiple conditions, you can use an else if block. This allows your program to choose between more than two paths.

const score = 87;

if (score >= 90) {

console.log('You got an A');

} else if (score >= 80) {

console.log('You got a B'); // You got a B

} else if (score >= 70) {

console.log('You got a C');

} else {

console.log('You failed! You need to study more!');

}

Since the score is currently 87, then the message of You got a B would be logged to the console.

The ternary operator is a compact way to write simple if/else statements. It has three parts: a condition, a result if the condition is true, and a result if it is false. Here's the basic syntax:

condition ? expressionIfTrue : expressionIfFalse;

Here's an example dealing with weather temperatures in Celsius:

const weather = temperature > 25 ? 'sunny' : 'cool';

console.log(`It's a ${weather} day!`);

If temperature is greater than 25, the code above logs It's a sunny day!. If temperature is ever less than 25, it logs It's a cool day!.

So, which should you use between an if statement and a ternary? Use a ternary while dealing with a single condition or single expressions, or when you want a compact syntax for simple logic. Use if/else statements when you're dealing with complex conditions and multiple statements, as things become unreadable if you nest ternaries.