JavaScript

Welcome to the world of JavaScript! Today, having a solid understanding of JavaScript is crucial for anyone who wants to pursue a career in web development. With its versatility and wide-range of applications, JavaScript is one of the most important programming languages to learn. But don't worry, we'll start from the basics and gradually raise the bar. By the end of these tutorials, you'll be well on your way to becoming a JavaScript pro.

What is Programming?

Programming is the art of communicating with computers. Just like some of your friends might understand Hindi, others might understand English, computers understand programming. Think of it as a language that we use to tell computers what to do.

The Dumb Computer

Despite its incredible speed and accuracy, a computer is actually quite "dumb." It can only do exactly what you tell it to do, and nothing more. This means that you have to be very clear and precise in your instructions, or else the computer might do something unexpected. For example, if you asked a person to jump, they would probably know what you meant. But if you asked a computer to jump, it wouldn't have a clue!

This might sound like a hindrance, but it's actually one of the things that makes computers so useful. They can perform complex calculations and processes much faster than a person ever could. For example, if you asked a person to solve the equation 24 + 5, they might take a few seconds to give you the answer of 29. But a computer would solve that equation almost instantly.

ECMAScript: The Standard Behind JavaScript

ECMAScript is the standardized version of JavaScript that helps ensure that code written in JavaScript will work the same way on any computer. ECMAScript has come a long way since its inception, with new features and capabilities added with each new release. In most cases, "JavaScript" and "ECMAScript" can be used interchangeably.

One of the great things about JavaScript is that it's a very forgiving language. It tries to do something with the code you write, even if it's not exactly right, and will only throw an error if you've really messed up.

How to Use JavaScript

There are several ways to use JavaScript, including:

1. In the browser: Most modern browsers have a JavaScript console that you can use to test and run your code.
2. In HTML: You can add JavaScript to your HTML files and run it in the browser. This is a great way to add interactivity to your website.
3. With a runtime environment: One popular runtime environment for JavaScript is Node.js. Node.js was created when a programmer named Ryan Dahl took Google's V8 JavaScript engine and wrapped it in C++. This allows you to run JavaScript code outside of the browser, which is useful for server-side programming.
   And, as a bonus, you can also use JavaScript in Replit, even if your computer's specs are low or you're on the go.

In conclusion, learning JavaScript will open up a world of possibilities for you. With its versatility and wide range of applications, there's no limit to what you can accomplish with this powerful programming language. So let's get started!

Foundation

Introduction to Syntax, Types of Languages, Comments and Variables

Before we dive into the world of programming, we need to understand the basics of syntax, types of languages, and variables. This chapter will cover the basics and lay a strong foundation for your programming journey.

Syntax

Just like there are some set of rules for writing a sentence in English, there are some set of rules for writing a program in a programming language. These rules are called syntax. Syntax is how a program is written and arranged. It defines the combination of symbols that are considered to be correctly structured.

Types of Languages

There are two main types of programming languages: dynamically typed and statically typed.

Dynamically Typed

Dynamically typed languages are the ones where the type of a variable is determined at runtime. This means that the type of a variable can change during the execution of the program. Js is example of a dynamically typed language.

Statically Typed

Statically typed languages are the ones where the type of a variable is determined at compile time. This means that the type of a variable cannot change during the execution of the program. C is example of a statically typed language.

Comments

Comments are used to explain the code. They are not executed by the computer. They are just for the programmer to understand the code. There are two types of comments: single line and multi-line.

Single Line Comments

Single line comments are used to comment a single line of code. They are denoted by //. Everything after // is a comment and will not be executed by the computer.

```
// This is a single line comment
console.log("Hello World"); // This is also a single line comment
```

Multi-Line Comments

Multi-line comments are used to comment multiple lines of code. They are denoted by /* and */. Everything between /* and */ is a comment and will not be executed by the computer.

```
/*
This is
a multi-line
comment.
*/
```

Think of comments as sticky notes that you leave for yourself or others who may read your code in the future. They help make your code easier to understand and maintain.

Variables

Variables are used to store data. They are like containers that hold data.Just like there are different containers in your kitchen like one may be used to store rice, another may be used to store sugar, similarly, variables are used to store different types of data. And since javascript is a dynamically typed language, the type of data that can be stored in a variable can change during the execution of the program.

Variables are declared using the var, let or const keyword (we will discuss this in next lecture). The syntax for declaring a variable is as follows:

```
var name = "Harry";
```

Here, name is the name of the variable and "Harry" is the value that is stored in the variable. The value can be of any type. It can be a string, a number, a boolean, an array, an object, etc. Don't worry if you don't understand what these terms mean. We will cover them in the upcoming lectures.

Now that we know what variables are, let's discuss some rules for naming variables.

Rules for Naming Variables

1. Variable names cannot start with a number.
   ```
   var 1name = "Harry"; // will throw an error because of the 1 at the start
   ```

2. Variable names cannot contain spaces.
   ```
   var my name = "Harry"; // will throw an error because of the space
   ```

3. Variable names cannot contain special characters like !, @, #, $, %, ^, &, *, (,), -, +, =, [,], {, }, |, \, :, ;, <, >, ,, ., /, ?, ~.
   ```
   var my-name = "Harry"; // will throw an error because of the `-` character
   ```

4. Variable names cannot be a reserved keyword. Reserved keywords are the keywords that are already used by the language. For example, var, let, const, if, else, for, while, function, return, etc.
   ```
   var var = "Harry"; // will throw an error because var is a reserved keyword
   ```

5. Variable names are case sensitive. This means that name and Name are two different variables.
   ```
   var name = "Harry";
   var Name = "Ron";
   console.log(name); // will print "Harry"
   console.log(Name); // will print "Ron"
   ```

Variables

var, let and const

In the previous video, we learned about syntax, types of languages, comments and variables. In this chapter, we will learn about the var, let and const keywords and how to declare variables using them.

var, let and const

In javascript, there are three keywords that are used to declare variables: var, let and const. The var keyword was used to declare variables in javascript before the let and const keywords were introduced. The let and const keywords were introduced in ES6 (ECMAScript 6). ES6 is the latest version of javascript. It was released in 2015. ES6 introduced a lot of new features to javascript. We will learn about some of them in this course.

var

The var keyword is used to declare variables in JavaScript. Here's an example:

```
var name = "Harry";
console.log(name); //output: Harry
```

we can also reassign the value of a variable declared using the var keyword:

```
var name = "Harry";
name = "Ron";
console.log(name); //output: Ron
```

One important thing to note about var is that it has block scope. This means that if a variable is declared with var inside a function, it is accessible anywhere within that block.

Note: Block scope will be discussed in the upcoming videos. For now, just remember that anything inside curly braces {} is a block.

Here's an example:

```
var name = "Harry";
{
   var name = "Ron";
   console.log(name); //output: Ron
}
console.log(name); //output: Ron
```

This is the reason why the var keyword is not recommended to be used anymore. It can lead to unexpected results. Instead we make use of the let and const keywords.

let

The let keyword is used to declare variables in JavaScript and has block scope. This means that if a variable is declared with let inside a block, it is only accessible within that block.

Here's an example:

```
let a = 29

let b = " Harry";
{
let b = "this";
console.log(b); //output: this
}
console.log(b); //output: Harry
```

const

The const keyword is used to declare variables in JavaScript and is used when you don't want to reassign the variable. The value of a variable declared with const cannot be changed.

Here's an example:

```
const name = " Harry";
name = "this";
console.log(name); //output: Uncaught TypeError: Assignment to constant variable.
```

Differences

	Block scoped	Hoisting	Reassignment	Initialization
let	Yes	No	Yes	Optional
var	No	Yes	Yes	Optional
const	Yes	No	No	Required

- **Block scoped**: let and const are block-scoped, meaning they are only accessible within the block they were defined in, while var is function-scoped, meaning it is accessible within the entire function it was defined in.
- **Hoisting:** let and const are not hoisted, meaning they cannot be accessed before they are declared, while var is hoisted, meaning it can be accessed before it is declared (although it will have a value of undefined).
- **Reassignment:** let and var can be reassigned to a new value, while const cannot be reassigned.
- **Initialization:** let and var can be declared without being initialized, while const must be initialized with a value when it is declared.

Best Practices

Now we know everything about the variables in JavaScript. Let's look at some best practices that we should follow while declaring variables in JavaScript to make our code more readable and maintainable.

1. Use descriptive and meaningful variable names: Choose variable names that clearly describe the value they hold. This makes your code easier to read and understand.
   ```
   const a = "Harry"; //bad
   const name = "Harry"; //good
   ```

2. Use camelCase to name your variables. This makes your code easier to read and understand.
   ```
   const myName = "Harry"; //good
   const myname = "Harry"; //bad
   ```

3. Use const by default and only use let if you need to reassign the variable. Avoid using var.
   ```
   const name = "Harry"; //good
   let temporary = 29; //good
   ```

```
var name = "Harry"; //bad
```

4. Declare variables at the top of their scope: To make your code more readable, it's best to declare variables at the top of their scope. This makes it easier to see what variables are in scope and what values they hold

```
//good:
{
   const name = "Harry";
   const hobby = "programming";

   console.log("My name is " + name + " and I love " + hobby); // output: My name is Harry and I love programming
}
//bad
{
   console.log("My name is " + name + " and I love " + hobby); // output: My name is undefined and I love undefined
   const name = "Harry";
   const hobby = "programming";
}
```

5. Use const whenever possible: If you know that a variable will not change, use const to declare it. This helps to prevent bugs in your code that could occur if you accidentally reassign a value to a variable that should not change.

```
const name = "Harry"; //good
let name = "Harry"; //fine but const is better because we know that the value of name will not change
var name = "Harry"; //bad
```

Conclusion

Variables are an essential part of JavaScript programming. They allow us to store and manipulate data in our code. In JavaScript, we can declare variables using the var, let, and const keywords. While the var keyword has block scope, it can lead to unexpected results and is no longer recommended. Instead, we should use the let and const keywords, with const being the preferred option whenever possible. When declaring variables, we should follow best practices like using descriptive names, using camelCase, declaring variables at the top of their scope, and using const by default. By following these practices, we can make our code more readable and maintainable.

Primitives and Objects

In JavaScript, there are 7 primitive data types. The primitive data types are fundamental data types that are built into JavaScript. A good trick to remember these 7 data types is "NNSSBBU":

- N - null
- N - number
- S - symbol
- S - string
- B - boolean
- B - bigint
- U - undefined

These data types are used to define the type of variables and the type of content that can be stored in them. For example:

```
let nullVar = null;
let numVar = 29;
let boolVar = true;
let bigIntVar = BigInt("567");
let strVar = "harry";
let symbolVar = Symbol("I'm a nice symbol");
let undefinedVar = undefined;
```

It's important to use descriptive variable names instead of generic ones like a, b, c, etc.

null vs undefined

The null data type represents the intentional absence of any object value, whereas the undefined data type indicates the absence of a defined value. For example:

```
let nullVar = null; // value is intentionally nothing
let undefinedVar; // value is undefined or not defined
```

Using typeof

We can use the typeof operator to determine the data type of a variable. For example:

```
console.log(typeof numVar); // Output: number
console.log(typeof strVar); // Output: string
```

Objects

Other than the 7 primitive data types, the non-primitive data type in JavaScript is the object. Objects are key-value pairs, used to map keys to values. We can create an object like this:

```
const bioData = {
            name: "Harry",
            age: 29,
            likesJS: true,
            secret: undefined,
};
```

We can access an object's values in two ways: by using square brackets [] or by using dot notation . For example:

```
console.log(bioData["name"]); // Output: Harry
console.log(bioData.age); // Output: 29
```

```
console.log(bioData["pet"]); // Output: undefined
```
Note: see how ["pet"] says undefined, this is because the key "pet" doesn't exist in the object bioData.

Conclusion

Remember that variables are like containers, and data types define what type of content can be stored in them. Just like we wouldn't store our cookies in a container meant for pesticides, it's important to use the correct data types in JavaScript.

Another key point to remember is that objects are used to map keys to values and can be accessed using square brackets [] or dot notation .

```
const harryMarks = {
        english: 100,
        maths: 80,
        chemistry: 40,
};

const shubhMarks = {
        english: 70,
        maths: 100,
        chemistry: 60,
};
```
In conclusion, understanding primitive data types and objects is important for writing effective JavaScript code. By using the correct data types, we can improve the efficiency and readability of our code.

Practice Set 1

Now that we've covered the basics of JavaScript, it's time to put our knowledge to the test. In this practice set, we'll cover a range of topics from variables to objects.

Q1: Adding numbers to strings
Create a variable of type string and try to add a number to it.

Answer

```
let name = "Harry";
let age = 29;
console.log(name + age); // "Harry29"
```
In JavaScript, **concatenation** is the process of combining two or more strings into a single string. When you use the + operator with a string and a number, the number is automatically converted to a string and then concatenated with the original string. We'll cover more about string concatenation in the future.

Q2: Using typeof
Use the typeof keyword to see the data type of the variables in the previous question.

Answer

```
console.log(typeof name); // "string"
console.log(typeof age); // "number"
```
When you use the typeof keyword with a variable, it returns a string indicating the data type of that variable. In this case, name is a string, and age is a number.

Q3: Const objects
Create a constant object in JavaScript and then try to change its value to another data type.

Answer

```
const biodata = {
   name: "Harry",
   age: 30
};
biodata = 29; // TypeError: Assignment to constant variable.
```
When you create a constant in JavaScript, its value cannot be changed. If you try to assign a new value to a constant, you'll get a TypeError.

Q4: Modifying const objects
Try to add a new key to the previous const object.

Answer

```
const biodata = {
   name: "Harry",
   age: 30
   };
biodata.address = "123 Main St";
console.log(biodata); // { name: "John", age: 30, address: "123 Main St" }
```

Even though biodata is a constant, we can still modify its properties. This is because the const keyword only prevents us from reassigning the variable to a different value. The object itself is still mutable, which means we can add or change its properties.

Q5: Month dictionary

Create a JavaScript program to make a month dictionary like 1 = January, 2 = February, and so on.

Answer

```
const months = {
  1: "January",
  2: "February",
  3: "March",
  4: "April",
  5: "May",
  6: "June",
  7: "July",
  8: "August",
  9: "September",
  10: "October",
  11: "November",
  12: "December"
};
```

Congratulations, you've completed Practice Set 1! If you got everything correct, great job! If not, don't worry - practice makes perfect. Keep practicing and you'll get there.

Operators and Expressions

In JavaScript, an expression is a combination of values, variables, and operators that can be evaluated to produce a result. An expression can be as simple as a single variable, or it can be complex, involving many different parts. For example, the expression `29` is a valid JavaScript expression, as is `a + b * 3`, which involves two variables (`a`and `b`), a constant (`3`), and two operators (`+` and `*`).

In programming, we often use operators in our computations. An operator is a symbol that tells JavaScript to perform a certain mathematical or logical operation. For example, when we write `20 + 9`, `20` and `9` are our operands, and `+` is our operator, and `29` is our result. It's not always the case that we have one operator and two operands. For example, if we have `!true`, we have one operator (`!`), one operand (`true`), and `false` is our result.

Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations. Here are the most common arithmetic operators in JavaScript:

Operator	Description	Example
+	Addition	`3 + 5` evaluates to `8`
-	Subtraction	`7 - 2` evaluates to `5`
*	Multiplication	`4 * 6` evaluates to `24`
/	Division	`12 / 4` evaluates to `3`
%	Modulus	`10 % 3` evaluates to `1`
**	Exponentiation	`2 ** 3` evaluates to `8`

In the above table, the first column represents the operator symbol, the second column describes what it does, and the third column provides an example of its usage. Here are some examples of arithmetic operators in action:

```
let a = 5;
let b = 3;

// Addition
let c = a + b; // c is now 8

// Subtraction
let d = a - b; // d is now 2

// Multiplication
let e = a * b; // e is now 15

// Division
let f = a / b; // f is now 1.6666666666666667

// Modulus
let g = a % b; // g is now 2

// Exponentiation
let h = a ** b; // h is now 125
```

In the case of the **modulus** operator (`%`), it returns the remainder after division. For example, `10 % 3` evaluates to `1`, since `10` divided by `3` leaves a remainder of `1`.

The **exponential** operator (`**`) raises the first operand to the power of the second operand. For example, `2 ** 3` evaluates to `8`, since `2` raised to the power of `3` is `8`.

In addition to the basic arithmetic operators, JavaScript also has some shorthand operators that combine arithmetic with assignment. Here are some examples:

Operator	Description	Example
+=	Addition assignment	a += b is the same as a = a + b
-=	Subtraction assignment	a -= b is the same as a = a - b
*=	Multiplication assignment	a *= b is the same as a = a * b
/=	Division assignment	a /= b is the same as a = a / b
%=	Modulus assignment	a %= b is the same as a = a % b

```
let x = 5;

// Addition assignment
x += 3; // equivalent to x = x + 3
console.log(x); // Output: 8

// Subtraction assignment
x -= 2; // equivalent to x = x - 2
console.log(x); // Output: 6

// Multiplication assignment
x *= 4; // equivalent to x = x * 4
console.log(x); // Output: 24

// Division assignment
x /= 3; // equivalent to x = x / 3
console.log(x); // Output: 8

// Modulus assignment
x %= 5; // equivalent to x = x % 5
console.log(x); // Output: 3
```

before we study comparison operators, heres an importanf concept to understand, **Post increment and Pre increment**. Have a look at the following code sample:

```
let a = 5;

// Post-increment: returns the value of a, then increments it
console.log(a++); // output: 5
console.log(a);   // output: 6

// Pre-increment: increments the value of a, then returns it
console.log(++a); // output: 7

// Post-decrement: returns the value of a, then decrements it
console.log(a--); // output: 7
console.log(a);   // output: 6

// Pre-decrement: decrements the value of a, then returns it
console.log(--a); // output: 5
```

In the code above, we first initialize the variable `a` with the value `5`. Then we use the post-increment operator (`a++`) to return the value of `a` (which is `5`), and then increment it to `6`. We log the value of `a++` to the console and get the output `5`, and then log the value of a to the console and get the output `6`.

Next, we use the pre-increment operator (`++a`) to increment the value of `a` to `7`, and then return its new value. We log the value of `++a` to the console and get the output `7`.

Then, we use the post-decrement operator (`a--`) to return the value of `a` (which is `7`), and then decrement it to `6`. We log the value of `a--` to the console and get the output `7`, and then log the value of `a` to the console and get the output `6`.

Finally, we use the pre-decrement operator (`--a`) to decrement the value of `a` to `5`, and then return its new value. We log the value of `--a` to the console and get the output `5`.

Comparison Operators

Comparison operators are used to compare two values and return a Boolean value (`true` or `false`) depending on whether the comparison is true or false. Here are the most common comparison operators in JavaScript:

Operator	Description	Example
==	Equal to	5 == 5 evaluates to true
===	Strict equal to	5 === "5" evaluates to false
!=	Not equal to	5 != 4 evaluates to true

Operator	Description	Example
!==	Strict not equal to	5 !== "5" evaluates to true
>	Greater than	6 > 4 evaluates to true
<	Less than	6 < 4 evaluates to false
>=	Greater than or equal to	6 >= 6 evaluates to true
<=	Less than or equal to	6 <= 4 evaluates to false

The first four operators in the table above compare two values for equality. The double equal sign (==) checks if two values are equal to each other, while the triple equal sign (===) checks if they are strictly equal to each other. The difference between the two is that the double equal sign performs type coercion, meaning that it will convert one value to match the other's type before making the comparison, while the triple equal sign requires both values to be of the same type.

The not equal to operator (!=) returns true if the values being compared are not equal, while the strictly not equal to operator (!==) returns true if the values are not only different, but also of a different type.

The last four operators in the table above compare two values to see which is greater, less than, or equal to the other. These operators work on both numbers and strings. When comparing strings, JavaScript compares the characters in the strings based on their Unicode code points.

Here are some examples of comparison operators in action:

```
console.log(5 > 2); // true
console.log(5 == "5"); // true (type coercion is performed)
console.log(5 === "5"); // false (different types)
console.log(5 != "6"); // true
console.log(5 !== "5"); // true (different types)
```

In the above examples, the first comparison returns true because 5 is greater than 2. The second comparison returns true because the double equal sign performs type coercion, and 5 and "5" are considered equal. The third comparison returns false because the triple equal sign requires both values to be of the same type, and 5 and "5" are not of the same type. The fourth comparison returns true because 5 is not equal to 6. The fifth comparison returns true because 5 and "5" are not only different, but also of different types.

Logical Operators

Operator	Description	Example
&&	Logical AND. Returns true if both operands are true, otherwise false.	true && true evaluates to true
||	Logical OR. Returns true if at least one of the operands is true, otherwise false.	true || false evaluates to true
!	Logical NOT. Returns the opposite of the operand. If it's true, it returns false. If it's false, it returns true.	!true evaluates to false

In JavaScript, there are three logical operators: && (logical AND), || (logical OR), and ! (logical NOT).

The && operator returns true if both operands are truthy, and false otherwise. Here's an example:

```
const a = 10;
const b = 20;
const c = 30;

console.log(a < b && b < c); // true
console.log(a < b && b > c); // false
```

In the first console.log() statement, both a < b and b < c are true, so the overall expression evaluates to true. In the second console.log() statement, b > c is false, so the overall expression evaluates to false.

The || operator returns true if at least one of the operands is truthy, and false otherwise. Here's an example:

```
const x = 5;
const y = 10;

console.log(x > 3 || y < 5); // true
console.log(x < 3 || y < 5); // false
```

In the first console.log() statement, x > 3 is true, so the overall expression evaluates to true. In the second console.log() statement, both x < 3 and y < 5 are false, so the overall expression evaluates to false.

The ! operator returns the opposite of the operand's truthiness. If the operand is truthy, ! returns false. If the operand is falsy, ! returns true. Here's an example:

```
const z = 10;

console.log(!(z > 5)); // false
console.log(!(z < 5)); // true
```
In the first console.log() statement, z > 5 is true, so !(z > 5) evaluates to false. In the second console.log() statement, z < 5 is false, so !(z <br 5) evaluates to true.

Phew that was a lot to grasp, if you dindt understand anything, sit back, relax for a while and try to understand it again. we gotta keep our foundations strong.

Before we learn conditional statements, let's learn how to ask the user for input since we will be using that in our programs.

Getting User Input

To get input from the user, we use the prompt() function in JavaScript. This input is stored as a string datatype in a variable. To check the datatype of the variable, we use the typeof operator.

```
let age = prompt("Enter your age");
console.log(typeof age); //age is of string datatype
```
Note: By default, the input is stored as a string datatype. If we want to perform mathematical operations on the input , we need to convert the string datatype into a number datatype. We can do this by using the Number() function.

```
let age = prompt("Enter your age");
console.log(typeof age); //age is of string datatype
age = Number(age);
console.log(typeof age); //age is of number datatype
```
We can also use the parseInt() function to convert the string datatype to a number.

```
let age = prompt("Enter your age");
console.log(typeof age); //age is of string datatype
age = parseInt(age);
console.log(typeof age); //age is of number datatype
```

Conditional Statements

Conditional statements are used to perform different actions based on different conditions. In JavaScript, we have the following conditional statements:

For example, when you log in to Instagram, you can use email or username. If you use email, your program might know that it should search in the place where you stored emails, and if you use a username, it should search in the place where you stored usernames.

- if statement
- if-else statement
- if else-if else statement

The Basic If Statement

The basic syntax for an if statement is as follows:

```
if(condition){
    //code if condition is true
}
```
The if statement checks if the condition is true or false. If the condition is true, the code inside the if statement is executed. If the condition is false, the code inside the if statement is not executed.

The If-Else Statement

The basic syntax for an if-else statement is as follows:

```
if(condition){
    //code if condition is true
}
else{
    //code if condition is false
}
```
The if-else statement checks if the condition is true or false. If the condition is true, the code inside the if statement is executed. If the condition is false, the code inside the else statement is executed.

The If-Else If-Else Statement

The basic syntax for an if-else if-else statement is as follows:

```
if(condition1){
    //code if condition1 is true
}
else if(condition2){
    //code if condition2 is true
}
```

```
else{
    //code if both condition1 and condition2 are false
}
```
The if-else if-else statement checks if the condition1 is true or false. If the condition1 is true, the code inside the if statement is executed. If the condition1 is false, the code inside the else if statement is executed. If both condition1 and condition2 are false, the code inside the else statement is executed.

Example Programs

Program 1
print "Hello World" if the user enters "Hello" as input.

```
let input = prompt("Enter a string");
if(input === "Hello"){
    console.log("Hello World");
}
```
Program 2
print "Hello World" if the user enters "Hello" as input. Otherwise, print "Goodbye World".

```
let input = prompt("Enter a string");
if(input === "Hello"){
    console.log("Hello World");
}
else{
    console.log("Goodbye World");
}
```
Program 3
print "Hello World" if the user enters "Hello" as input. Otherwise, print "Goodbye World" if the user enters "Goodbye" as input. Otherwise, print "Invalid Input".

```
let input = prompt("Enter a string");
if(input === "Hello"){
    console.log("Hello World");
}
else if(input === "Goodbye"){
    console.log("Goodbye World");
}
else{
    console.log("Invalid Input");
}
```
Note that we didn't convert the input to a number datatype. This is because we are not performing any mathematical operations on the input. If we were performing mathematical operations on the input, we would have converted the input to a number datatype.

For Example

```
let input = prompt("Enter a number");
if(input > 10){
    console.log("Number is greater than 10");
}
else if(input < 10){
    console.log("Number is less than 10");
}
else{
    console.log("Number is equal to 10");
}
```

Ternary Operator

The ternary operator is a shorthand for the if-else statement. The basic syntax for the ternary operator is as follows:

```
condition ? expression1 : expression2
```
If the condition is true, the expression1 is executed. If the condition is false, the expression2 is executed.

For Example: print "Hello World" if the user enters "Hello" as input. Otherwise, print "Goodbye World".

```
let input = prompt("Enter a string");
input === "Hello" ? console.log("Hello World") : console.log("Goodbye World");
```

Homework

Just like if else and ternary operator, you can also use switch statements. As a homework, explore the switch statement and try to write a program using the switch statement.

Practice Set 2

Ok everyone, now lets put our knowledge to the test. and try to solve the following questions.

Q1: Using Logical Operators

Create a JavaScript program to check if a age is between 10 and 20.

Answer

```
let age = prompt("Enter your age");
if (age >= 10 && age <= 20) {
  console.log("You are between 10 and 20");
} else {
  console.log("You are not between 10 and 20");
}
```

In this example, we use the && operator to check if the age is between 10 and 20. If the age is between 10 and 20, the condition evaluates to true, and the first console.log statement is executed. If the age is not between 10 and 20, the condition evaluates to false, and the second console.log statement is executed.

Q2: Using Switch Case

Demonsrate the use of switch case in JavaScript.

Answer

```
let day = prompt("Enter a day");
switch (day) {
  case "Monday":
    console.log("Today is Monday");
    break;
  case "Tuesday":
    console.log("Today is Tuesday");
    break;
  case "Wednesday":
    console.log("Today is Wednesday");
    break;
  case "Thursday":
    console.log("Today is Thursday");
    break;
  case "Friday":
    console.log("Today is Friday");
    break;
  case "Saturday":
    console.log("Today is Saturday");
    break;
  case "Sunday":
    console.log("Today is Sunday");
    break;
  default:
    console.log("Invalid day");
}
```

In this example, we use the switch statement to check the value of the day variable. If the value of day is Monday, the first console.log statement is executed. If the value of day is Tuesday, the second console.log statement is executed. If the value of day is not Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday, the default statement is executed.

Also do note that the break keyword is used to prevent the execution of the next case statement. If you don't use the break keyword, the execution will continue to the next case statement, even if the condition is false.

Q3: Divisible by 2 and 3

Create a JavaScript program to check if a number is divisible by 2 and 3.

Answer

```
let num = prompt("Enter a number");
if (num % 2 == 0 && num % 3 == 0) {
  console.log("The number is divisible by 2 and 3");
} else {
  console.log("The number is not divisible by 2 and 3");
}
```

In this example, we use the && operator to check if the number is divisible by 2 and 3. If the number is divisible by 2 and 3, the condition evaluates to true, and the first console.log statement is executed. If the number is not divisible by 2 and 3, the condition evaluates to false, and the second console.log statement is executed.

Q4: Divisible by 2 or 3

Create a JavaScript program to check if a number is divisible by 2 or 3.

Answer

```
let num = prompt("Enter a number");
if (num % 2 == 0 || num % 3 == 0) {
  console.log("The number is divisible by 2 or 3");
} else if (num % 2 == 0) {
  console.log("The number is divisible by 2");
} else if (num % 3 == 0) {
  console.log("The number is divisible by 3");
```

```
} else {
  console.log("The number is not divisible by 2 or 3");
}
```
In this example, we use the `||` operator to check if the number is divisible by `2` or `3`. If the number is divisible by 2 or 3, the condition evaluates to true, and the first `console.log` statement is executed. If the number is not divisible by 2 or 3, the condition evaluates to false, and the second `console.log` statement is executed.

Q5: Using Ternary Operator

Print "you can drive" if the age is greater than or equal to 18, otherwise print "you cannot drive". Use the ternary operator.

Answer

```
let age = prompt("Enter your age");
age >= 18
  ? console.log("You can drive")
  : console.log("You cannot drive");
```
In this example, we use the ternary operator to check if the age is greater than or equal to 18. If the age is greater than or equal to 18, the condition evaluates to true, and the first `console.log` statement is executed. If the age is less than 18, the condition evaluates to false, and the second `console.log` statement is executed.

Congratulations! You have completed the practice set. Now you can move on to the next chapter.

Loops

Loops are used to perform repetitive tasks in programming. For example, if you wanted to print something 20 times, you could do it manually, but what if you needed to print it 20,000 times? That's where loops come in handy.

There are 5 types of loops in JavaScript:

1. `for` loop
2. `for-in` loop
3. `for-of` loop
4. `while` loop
5. `do-while` loop

In this lesson, we'll learn about the first 3 types of loops.

For Loops

The syntax for a for loop is as follows:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```
Statement 1 is executed once before the loop starts. Statement 2 is checked at the beginning of each iteration, and if it's true, the code block is executed. After the code block has been executed, statement 3 is executed, and then the condition is checked again. This process repeats until the condition is no longer true.

Here's an example of using a for loop to print numbers from 1 to 29:

```
for (let i = 1; i <= 29; i++) {
  console.log(i);
}
```
This code will print the numbers from 1 to 29 to the console. As you can see, using a for loop makes it much easier to perform repetitive tasks like this.

Summing Numbers with a for Loop

Here's an example of using a for loop to sum the first n numbers, where n is entered by the user:

```
let n = prompt("Enter a number");
let sum = 0;
for (let i = 1; i <= n; i++) {
  sum = sum + i;
}
console.log("The sum of first " + n + " numbers is " + sum);
```
This code is used to calculate the sum of the first n numbers where `n` is provided by the user through a prompt. The code uses a for loop to iterate over the numbers from 1 to n and add them to a variable called `sum`.

The for loop starts by initializing a loop variable `i` to 1, and checking if it is less than or equal to the value of `n`. If the condition is true, the loop body is executed, which adds the value of `i` to the `sum` variable. The loop then increments the value of `i` by 1 and repeats the process until the condition becomes `false`.

Finally, the code prints out the result of the sum using `console.log()`, which concatenates strings and variables to create a message that displays the value of `n` and the sum of the first `n` numbers.

For-In Loops

A for-in loop is a variant of the for loop that is used to iterate over the properties of an object. The syntax is as follows:

```
for (let property in object) {
  // code block to be executed
}
```
Here's an example of using a for-in loop to iterate over the properties of an object:

```
let person = {
  name: "Harry",
  age: 30,
  gender: "male"
};

for (let x in person) {
  console.log(x + ": " + person[x]);
}
```
This code will print each property of the person object and its corresponding value to the console.

For-Of Loops

A for-of loop is another variant of the for loop that is used to iterate over iterable objects like arrays and strings. The syntax is as follows:

```
for (let value of iterable) {
  // code block to be executed
}
```
Here's an example of using a for-of loop to iterate over the elements of an array:

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

for (let number of numbers) {
  console.log(number);
}
```
This code will print each element of the numbers array to the console.

Note that we will be using for-in and for-of loops extensively when working with arrays, which we will cover later in this course.

Conclusion

Loops are a fundamental programming concept used to perform repetitive tasks efficiently. JavaScript offers five types of loops: **for, for-in, for-of, while,** and **do-while**.

For loops are commonly used for iterating over a sequence of numbers or a list of items. For-in loops are used for iterating over the properties of an object, and for-of loops are used for iterating over iterable objects such as arrays and strings. Understanding how to use loops effectively is essential for writing efficient and readable code in JavaScript.

We will discuss while and do-while loops in our next tutorial.

We discussed the following loops in previous lesson

1. for loop
2. for-in loop
3. for-of loop

In this lesson, we will learn about the remaining 2 types of loops:

1. while loop
2. do-while loop

while Loop

A while loop is a type of loop that repeatedly executes a block of code while a particular condition is true. The loop will keep executing until the condition becomes false.

Syntax

The syntax of the while loop is as follows:

```
while (condition) {
   // code to be executed
}
```
Here's an example of using a while loop to print all the numbers from 0 to the user input:

```
let input = prompt("Enter a number");
input = Number(input);
let i = 0;

while (i < input) {
   console.log(i);  // print the current value of i
   i++;  // increment i by 1
}
```
The code above works as follows:

- Ask the user to enter a number using prompt().
- Convert the user input to a number using Number().
- Initialize the variable i to 0.
- Check if i is less than the user input.

- If `i` is less than the user input, print the current value of `i` using `console.log()`.
- Increment `i` by 1.
- Check the condition again, and repeat the loop until `i` is no longer less than the user input.

Example without i++

What happens if we don't increment `i` by 1? Here's an example of using a while loop to print all the numbers from 0 to the user input without using i++:

```
let input = prompt("Enter a number");
input = Number(input);
let i = 0;

while (i < input) {
   console.log(i);  // print the current value of i
}
```

In the code above, the `i` variable is never incremented, so the loop will continue to execute indefinitely. To stop the loop, you can press `Ctrl+C`.

Do While Loop

A `do-while` loop is a type of loop that executes a block of code at least once, and then repeatedly executes the block of code while a particular condition is true.

Syntax

The syntax for the do-while loop is as follows:

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

The following code asks the user for a number and prints all the numbers from 0 to that number using a do-while loop:

```
let input;
do {
   input = prompt("Enter a number");
   input = Number(input);
} while(isNaN(input));

let i = 0;
do {
   console.log(i);
   i++;
} while(i < input);
```

- The first `do` loop asks the user for a number and assigns it to the `input` variable. If the input is not a number, the loop will run again until a valid number is entered.
- The `let i = 0` line initializes a counter variable `i` to 0.
- The second `do` loop will keep running as long as `i` is less than `input`.
- The `console.log(i)` line will print the value of `i` to the console on each iteration.
- The `i++` statement increments the value of `i` by 1 after each iteration.

Differences between While and Do-While Loop

	while loop	do-while loop
Syntax	while (condition) { statement }	do { statement } while (condition);
First iteration	Condition is checked before the iteration.	The statement is executed before the check.
Number of loops	0 or more	1 or more
Execution order	Condition → Statement	Statement → Condition → Statement
Use case	Use when the number of iterations is unknown or based on a condition.	Use when you want to execute the statement at least once, and then check the condition.

In summary, the main differences between the `while` loop and the `do-while` loop are:

- The `while` loop checks the condition first, then executes the loop if the condition is true, however the `do-while` loop executes the loop first, then checks the condition.
- The `while` loop may execute zero or more times, while the `do-while` loop always executes at least once.
- The `while` loop is often used when the number of iterations is unknown or based on a condition, however the `do-while` loop is useful when you want to execute a statement at least once, and then check the condition.

We understand that the concept of loops can be intimidating and confusing for beginners. However, with practice and a good understanding of the concepts covered in these markdowns, you will be well on your way to writing your own programs. Remember, don't be discouraged if you don't get it right away - keep practicing and you'll get there!

Functions

Functions are a way to group a block of code together and give it a name, which can then be called/invoked at any point in your code. They are important for keeping your code organized and separating different pieces of logic.

Why Are Functions Needed?

Sometimes we have a piece of code that we need to repeat multiple times in our code. Rather than writing it out every time, we can create a function to handle it. Functions can also take arguments as input, allowing them to be used with different values and reducing the amount of code needed.

Syntax of JavaScript Functions

The basic syntax of a function in JavaScript is as follows:

```
function functionName(parameters) {
  // function code
}
```

Example: Simple Function

Here's an example of a simple function that just prints "Hello, world!" when called:

```
function sayHello() {
  console.log("Hello, world!");
}

sayHello(); // calling/invoking the function
```

Note that if you don't call/invoke the function, nothing will happen.

Example: Function with Arguments

Here's an example of a function that takes two numbers, calculates their average, adds 30 to it, and prints the result:

```
function averageWith30(x, y) {
  let average = (x + y) / 2;
  let result = average + 30;
  console.log(`30 + average of ${x} and ${y} is = ${result}`);
}

averageWith30(5, 10); // calling/invoking the function with arguments
```

You can use any variable names instead of `x` and `y`.

One of the advantages of using functions is that if you need to make a change, you only need to do it once in the function code and it will be reflected everywhere you call the function.

Parameter vs. Argument

The terms "parameter" and "argument" are often used interchangeably, but they have slightly different meanings:

Parameter	Argument
Defined in the function declaration	Passed to the function when called
Used within the function	Value of the parameter when the function is called
Example: `function add(x, y) {...}`	Example: `add(2, 3)`

Returning Values

Functions can also return values using the return keyword. Here's an example of a function that calculates the square of a number and returns the result:

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

let result = square(5); // calling/invoking the function and storing the result
console.log(result);
```

Note that when a function returns a value, you need to do something with that value, such as store it in a variable or use it in an expression.

Arrow Functions

Arrow functions are a shorthand way of writing functions in JavaScript. Here's an example of a simple arrow function that just prints "Hello, world!":

```
let sayHello = () => {
  console.log("Hello, world!");
}

sayHello(); // calling/invoking the function
```

Conclusion

Functions are an important part of programming and can greatly simplify your code by allowing you to reuse blocks of code and make changes in one place that are reflected everywhere. They can also take arguments and return values, making them versatile and powerful tools for organizing your code.

Practice set 3

Today, we will be testing our knowledge about loops and functions in JavaScript. We will be solving a few programming questions to help you understand the concepts better. Remember that there are dozens of way of solving a problem, your oesnt have to be the same way as mine, aslong as it give u the desired output, its fine.

Question 1

Write a program to print the marks of a student in an object using for loop: object could be like this:

```
 {
  coco: 80,
  harry: 98,
  rohan: 75
 }
```

Answer

```
let marks = {
  coco: 80,
  harry: 98,
  rohan: 75
}

for (let i = 0; i < Object.keys(marks).length; i++) {
  console.log("The marks of " + Object.keys(marks)[i] + " are " + marks[Object.keys(marks)[i]]);
}
```

This might seem a bit complex its because we havent studied arrays yet, but dont worry, we will learn it soon. For now, just know that arrays are a collection of items. Here is a detailed explaination of the code, dont worry if you dont understand it, we will learn it soon.

The first line creates an object called "marks" that contains three key-value pairs. Each key represents a student's name and the corresponding value represents their marks.

The second line starts a `for` loop that will iterate through each of the keys in the "marks" object.

The `Object.keys()` method is used to get an array of all the keys in the "marks" object. `Object.keys(marks)` will return an array containing ["coco", "harry", "rohan"]. The `.length` property is then used to get the number of items in this array (which is 3 in this case).

The `for` loop runs three times (once for each student) because of the `Object.keys(marks).length` condition in the loop.

The third line of code inside the `for` loop is what prints the student's name and their corresponding marks to the console.

The `Object.keys(marks)[i]` part of the code retrieves the key for the current student that we are looking at in the loop. For example, when `i` is `0`, it will retrieve the key `coco`.

The `marks[Object.keys(marks)[i]]` part of the code retrieves the value for the current key. In other words, it gets the marks for the current student. For example, when `i` is `0`, it will retrieve the value `80`.

Finally, the `console.log()` method is used to print the student's name and marks to the console using the retrieved key and value.

So in summary, this code loops through each key in the "marks" object and prints the name of the student along with their marks to the console.

Question 2

Write a program in Q1 using for in loop:

Answer

```
let marks = {
  coco: 80,
  harry: 98,
  rohan: 75
}
```

```
for (let key in marks) {
  console.log("The marks of " + key + " are " + marks[key]);
}
```
See how much simpler it is to use a `for-in` loop in this scenario? The `for-in` loop automatically iterates through each key in the object, so we don't need to use `Object.keys()` to get an array of the keys.

Question 3

Write a program to print "try again" until the user enters the correct number:

Answer

```
let correctNumber = 9;
let guessedNumber = null
while (guessedNumber != correctNumber){
guessedNumber = prompt("Enter a number: ")
console.log("Try Again!");
}
console.log("congrats, youve guessed the number!")
```
The `while` loop will continue to run as long as the condition `guessedNumber != correctNumber` is true. The `prompt()` method is used to get the user's input and store it in the `guessedNumber` variable. The `console.log()` method is used to print "Try Again!" to the console. However there is a slight issue with this code, even if thse user enters the correct number, the loop will still run once more. To fix this, we can add an `if` statement inside the loop that checks if the user has guessed the correct number. If they have, we can use the `break` keyword to exit the loop.

```
let correctNumber = 9;
let guessedNumber = null
while (guessedNumber != correctNumber){
guessedNumber = prompt("Enter a number: ")
if (guessedNumber == correctNumber){
break;
}
console.log("Try Again!");
}
console.log("congrats, youve guessed the number!")
```
the `break` keyword is used to exit the loop. So when the user enters the correct number, the `break` keyword will be executed and the loop will be exited.

Question 4

Write a function to find the mean of 5 numbers:

Answer

```
function findMean(num1, num2, num3, num4, num5) {
  let sum = num1 + num2 + num3 + num4 + num5;
  let mean = sum / 5;
  return mean;
}

let result = findMean(10, 20, 30, 40, 50);
console.log("The mean of the numbers is " + result);
```
The `function` keyword is used to create a function. The `findMean()` function takes five parameters: `num1`, `num2`, `num3`, `num4`, and `num5`. The `return` keyword is used to return the mean of the five numbers to the caller of the function and finally, the `console.log()` method is used to print the result to the console.

Congratulations on completing the programming questions! Don't worry if you weren't able to complete them. You can always revise the concepts and practice more. Remember, practice makes perfect!

Strings

A string is a collection of characters enclosed in quotes. In JavaScript, you can use either single quotes or double quotes to create a string. However, it's important to remember that once you choose a quote type to start your string, you must use the same type to close it. For example,

```
let name = "Harry";         //correct
let friend = 'Prakash'      //correct
let wrong = 'Harry";        //wrong, never do this!
```
Even a number enclosed in quotes is considered a string, like `"29"`. You can convert a number to a string using the `toString()` method. For example:

```
let age = 29;
let ageAsString = age.toString();
console.log(typeof ageAsString); //string
```
By converting a number to a string, you can no longer perform mathematical operations on it. But you will be able to perform operations on it like concatenation.

Template Literals

You can also use backticks ` (it is the button under your escape key) to create a string, called a template literal. This is useful when you need to embed variables within a string. Without template literals, you would need to concatenate strings and variables using the + operator. Here's an example:

```
const name = "Rayyan";
const favorite = "Pepsi";
const sentence = name + " loves " + favorite;
console.log(sentence); // Output: "Rayyan loves Pepsi"
```

With template literals, you can embed variables within a string without using the + operator. We use the ${variable} syntax to embed a variable within a string. Here's an example:

```
const sentence = `${name} loves ${favorite}`;
console.log(sentence); // Output: "Rayyan loves Pepsi"
```

Notice how the variables are enclosed in ${} within the backticks, and the string is automatically formatted with the values of the variables. This is called string interpolation.

Escape Sequences

Sometimes, you may want to use a character that is already used in JavaScript. For example, you may want to use a single quote within a string. In this case, you can use an escape sequence. An escape sequence is a backslash \ followed by a character that tells the JavaScript interpreter to interpret the following character in a special way. Here are some examples:

```
let sentence = 'She said, "Hello!"';
console.log(sentence);                  // Output: She said, "Hello!"

sentence = "She said, \"Hello!\"";
console.log(sentence);                  // Output: She said, "Hello!"

sentence = "She said, \"Hello!\"";      // Output: She said, "Hello!"
```

- The first example uses a single quote to start the string and a single quote to end the string. Since the string contains a single quote, we can use a double quote without escaping it.
- The second example uses a double quote to start the string and a double quote to end the string. Since the string contains a double quote, we can use a single quote without escaping it.
- The third example uses a single quote to start the string and a single quote to end the string. Since the string contains a single quote, we use a backslash followed by a double quote to escape it.

Escape sequences can also be used to include special characters within a string. For example, \n represents a new line, and \t represents a tab. Here's an example:

```
let sentence = "Hello\nWorld";
console.log(sentence); // Output:   Hello
                       //         World
```

In the above example, the \n escape sequence creates a new line after the word "Hello".

```
let sentence = "Hello\tWorld";
console.log(sentence); // Output: Hello   World
```

In the above example, the \t escape sequence creates a tab after the word "Hello".

We can also use escape sequences together here's an example:

```
let sentence = "Hello\n\tWorld";
console.log(sentence); // Output: Hello
                       //         World
```

In the above example, the \n escape sequence creates a new line after the word "Hello", and the \t escape sequence creates a tab after the word "Hello".

It's important to note that escape sequence characters are not treated as characters of the string. For example, "apple\".length will return 6, not 7, because the \ is treated as an escape character, not a regular character.

Conclusion

In this lesson, we learned about strings and how to create them. We also learned about template literals and escape sequences. Don't forget to practice what you've learned so far!

String Methods

In the last lesson, we discussed what strings are, how to make and use them. Now we will be having a look at some methods of strings by using which you can easily manipulate the strings. Maybe you want to convert your sentence into uppercase, maybe you want to replace a particular word in a sentence, etc. You can do all of this easily with string methods.

String Length

The length property of a string returns the number of characters in the string.

```
let name = "Harry";
console.log(name.length); // Output: 5
```

Note that the length property is not a method. It is a property of the string object. You do not need to use parentheses to access it. Here is a table displaying the difference between a method and a property:

Property	Method
A property is a value that belongs to an object, such	A method is a function that belongs to an object, such as a

Property	Method
as a string's length or a number's value.	string's toUpperCase() or an array's push() method.
You can access a property using dot notation, without using parentheses. For example, string.length returns the length of the string.	You call a method using parentheses, which may or may not take arguments. For example, string.toUpperCase() returns a new string with all characters in uppercase.
Properties cannot be called like a function, they just return a value. For example, string.length() is not valid.	Methods can be called like a function, and they perform an action on the object they belong to. For example, string.toUpperCase() is valid and returns a new string in uppercase.
Properties are used to get information about an object.	Methods are used to perform an action on an object.

toUpperCase() and toLowerCase()

The toUpperCase() method converts all characters in a string to uppercase, while the toLowerCase() method converts all characters to lowercase. Note: If a string is already in uppercase or lowercase, these methods do nothing and return the original string.

```
let name = "Harry";
console.log(name.toUpperCase()); // Output: HARRY
console.log(name.toLowerCase()); // Output: harry
```

slice()

The slice() method returns a section of a string. You can provide two arguments to the method: the starting index and the ending index (not inclusive) of the substring you want to extract.

```
let name = "Harry";
console.log(name.slice(2, 4)); // Output: rr
console.log(name.slice(1, 3)); // Output: ar
console.log(name.slice(2)); // Output: rry
```

If you omit the second argument, slice() will return the remainder of the string starting from the provided index:

```
console.log(name.slice(2)); // Output: rry
```

replace()

The replace() method returns a new string with some or all matches of a pattern replaced by a replacement string.

```
let name = "Harry";
console.log(name.replace("ry", "is")); // Output: Hais
console.log(name.replace("Ry", "is")); // This will not work because "Ry" is not present in the string
```

Note that replace() is case-sensitive.

trim()

The trim() method removes whitespace from both ends of a string.

```
let spacedName = "      Harry      ";
console.log(spacedName.trim()); // Output: Harry
```

indexOf()

The indexOf() method returns the index of the first occurrence of a specified value in a string. This method returns -1 if the value to search for never occurs.

```
let name = "Harry";
console.log(name.indexOf("r")); // Output: 2
console.log(name.indexOf("R")); // This will give -1 because "R" is not present in the string
```

Note that indexOf() is case-sensitive.

Indexing

You can access individual characters in a string using indexing. In JavaScript, indexing starts at 0.

```
let name = "Harry";
console.log(name); // Output: Harry
console.log(name[0]); // Output: H
console.log(name[1]); // Output: a
console.log(name[2]); // Output: r
console.log(name[3]); // Output: r
console.log(name[4]); // Output: y
```

It's important to remember that all string methods return a new string, leaving the original string unchanged. If you want to make changes to a string, you'll need to store the result of the method call in a new variable.

```
let name = "Harry";
let upperCaseName = name.toUpperCase();
console.log(upperCaseName); // Output: HARRY
console.log(name); // Output: Harry (original string is unchanged)
```

These are some of the most important string methods, but there are many others that you can learn and use in your code. Remember that you don't have to memorize all of these methods; instead, focus on understanding how they work and practice using them in your code.

Practice Set 4

Time to practice what you've learned! This practice set will help you get comfortable with the concepts you've learned in previous lessons.

Question 1
What will the following code print in JavaScript?

```
console.log("har\"".length)
```
Answer

The above code will print `4` because the escape sequence `\` is not counted as a character in the string. Therefore, the length of the string `"har\""` is actually 4, not 5.

Question 2
Explore the `includes`, `startsWith`, and `endsWith` functions in a string.

Answer

```
let str = "Hello World";
console.log(str.includes("Hello")); // true
console.log(str.startsWith("Hello")); // true
console.log(str.endsWith("World")); // true

console.log(str.includes("harry")); // false
console.log(str.startsWith("new")); // false
console.log(str.endsWith("new")); // false
```
The `includes` method returns `true` if the string includes the specified substring, otherwise it returns `false`. The `startsWith` method returns `true` if the string starts with the specified substring, otherwise it returns `false`. The `endsWith` method returns `true` if the string ends with the specified substring, otherwise it returns `false`.

Question 3
Write a program to convert a given string into lowercase.

Answer

```
let str = "Hello World";
console.log(str.toLowerCase());
```
The `toLowerCase` method returns the calling string value converted to lower case.

Question 4
Extract the numerical amount from this string "Total: Rupees 2907".

Answer

```
let str = "Total: Rupees 2907";
console.log(str.slice(13));
```
The `slice` method extracts a section of a string and returns it as a new string, without modifying the original string. The `slice` method takes two arguments, the starting index and the ending index (end not included). If the ending index is not specified, it will slice the string till the end.

Question 5
Try to change the 4th character of a given string. Would you be able to do it?

Answer

```
let str = "Hello World";
str[3] = "z"; // 3rd index is 4th character
console.log(str); // Hello World | No change
```
Strings are immutable in JavaScript. This means that strings cannot be changed, only replaced. Therefore, the above code will not change the string.

Thats it for this practice set

Ok so we will finally be discussing arrays now, they are a very cruicial part of javascript and are used in almost every program, so make sure you understand them well.

Arrays

Arrays are a way to store multiple values in a single variable. They are very useful when you want to store a list of values and perform operations on them. Arrays are declared using square brackets `[]` and the values are separated by commas `,`. For example, the following code declares an array named `arr` with 3 values:

```
let arr = [1, 2, 3];
```
The values in an array are called elements. The first element of an array is at index 0, the second element is at index 1, and so on. So, in the above example, the value `1` is at index 0, the value `2` is at index 1, and the value `3` is at index 2.

Creating an Array

We can create an array of marks and store different marks in it as follows:

```
let marks = [90, 85, 95, 80];
console.log(marks); // [90, 85, 95, 80]
```

Arrays can have different data types as well. For example, we can create an array with a string, a number, and a boolean as follows:

```
let arr = ["Hello", 10, true];
console.log(arr); // ["Hello", 10, true]
```

We can also create an empty array as follows:

```
let arr = [];
console.log(arr); // []
```

Why would we want to create an empty array? Well, we can add elements to an array later on. We will see how to do that later in this lesson.

Accessing Elements of an Array

We can access the elements of an array using the index of the element. For example, the following code accesses the first element of the array `marks`:

```
let marks = [90, 85, 95, 80];
console.log(marks[0]); // 90
```

lets create a new array `fruits` and try accessing the elements of the array:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits[0]); // Apple
console.log(fruits[1]); // Orange
console.log(fruits[2]); // Banana
```

We can also access the elements of an array using negative indices. For example, the following code accesses the last element of the array `fruits`:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits[-1]); // Banana
```

The index `-1` refers to the last element of the array, `-2` refers to the second last element, and so on. So, the index `-1` is equivalent to the index `fruits.length - 1`.

If we try to access an index that is not present in the array, it will return undefined.

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits[3]); // undefined
```

Array Length

We can find the length of an array using the `.length` property, which returns the number of elements in the array.

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits.length); // 3
```

Adding and Changing Array Values:

Unlike strings, arrays are mutable, which means we can change or add values to them. To add a new value to an array, we need to specify the index where we want to add the value and assign the new value to it. If the index does not exist in the array, it will create a new element with the assigned value, and the elements in the array after that index will be shifted to the right.

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits); // ["Apple", "Orange", "Banana"]
console.log(fruits.length); // 3
fruits[3] = "Grapes";
console.log(fruits); // ["Apple", "Orange", "Banana", "Grapes"]
console.log(fruits.length); // 4
```

In the above example, we added a new element to the array `fruits` at index 3. The value of the element at index 3 is `"Grapes"`. The length of the array is now 4.

We can also change the value of an existing element in an array. For example, the following code changes the value of the first element of the array `fruits`:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits); // ["Apple", "Orange", "Banana"]
console.log(fruits.length); // 3
fruits[0] = "Pinapple";
console.log(fruits); // ["Pinapple", "Orange", "Banana"]
console.log(fruits.length); // 3
```

In the above example, we changed the value of the first element of the array `fruits` to `"Pinapple"`. Note that the length of the array is still 3, this is because we did not add or remove any elements from the array, we just modified the value of an existing element.

ok so last thing to take away from this lesson is that arrays are not primitive types; they are a type of object in JavaScript. We can check the type of an array using the `typeof` operator.

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(typeof fruits); // object
```

In the next lesson, we will learn about the different methods that can be used to manipulate arrays. It will be a very interesting lesson!

Array Methods

Array methods are functions that are called on an array. They can be used to perform operations on the elements of an array. In this lesson, we will learn about some of the most commonly used array methods.

Arrays come with a bunch of built-in methods that can be used to manipulate the array. Some methods return a new array, while others modify the existing array. It's important to note that these methods do not create a copy of the original array unless explicitly stated. Here are some of the most commonly used array methods:

toString()

The `toString()` method converts an array to a string of comma-separated values. For example, the following code converts the array `fruits` to a string:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits.toString()); // Apple,Orange,Banana
console.log(typeof fruits.toString()); // string
```

join()

The `join()` method also converts an array to a string. However, it takes an argument for the separator. For example, the following code converts the array `fruits` to a string using a hyphen as the separator:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits.join("-")); // Apple-Orange-Banana
console.log(typeof fruits.join("-")); // string
```

or lets try " and " as the separator:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits.join(" and ")); // Apple and Orange and Banana
```

pop()

The `pop()` method removes the last element from an array and returns that element. For example, the following code removes the last element from the array `fruits`:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits.pop()); // Banana
console.log(fruits); // ["Apple", "Orange"]
```

push()

The `push()` method adds one or more elements to the end of an array and returns the new length of the array. For example, the following code adds the string "Pineapple" to the end of the array `fruits`:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits.push("Pineapple")); // 4
console.log(fruits); // ["Apple", "Orange", "Banana", "Pineapple"]
```

shift()

The `shift()` method removes the first element from an array and returns that element. For example, the following code removes the first element from the array `fruits`:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits.shift()); // Apple
console.log(fruits); // ["Orange", "Banana"]
```

unshift()

The `unshift()` method adds one or more elements to the beginning of an array and returns the new length of the array. For example, the following code adds the string "Pineapple" to the beginning of the array `fruits`:

```
let fruits = ["Apple", "Orange", "Banana"];
console.log(fruits.unshift("Pineapple")); // 4
console.log(fruits); // ["Pineapple", "Apple", "Orange", "Banana"]
```

This was a quick overview of some of the most commonly used array methods. In the next lesson, we will explore some more of these.

More Array Methods

In the previous lesson, we learned some common array methods, lets explore some more of the most commonly used array methods in this lesson.

delete()

The `delete()` method removes the element at the specified index from an array. For example, the following code removes the element at index 1 from the array `fruits`:

```
let fruits = ["Apple", "Orange", "Banana"];
delete fruits[1];
console.log(fruits); // ["Apple", <1 empty item>, "Banana"]
```

Note that the `delete()` method does not change the length of the array. It simply removes the element at the specified index and replaces it with an empty item. Also, the `delete()` method does not return the removed element like the `pop()` method.

concat()

The `concat()` method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array. For example, the following code merges the arrays `fruits` and `vegetables`:

```
let fruits = ["Apple", "Orange", "Banana"];
let vegetables = ["Potato", "Tomato", "Onion"];
let food = fruits.concat(vegetables);
console.log(food); // ["Apple", "Orange", "Banana", "Potato", "Tomato", "Onion"]
```

The `concat()` method can also be used to add items to an existing array. For example, the following code adds the string "Pineapple" to the array `fruits`:

```
let fruits = ["Apple", "Orange", "Banana"];
let food = fruits.concat("Pineapple");
console.log(food); // ["Apple", "Orange", "Banana", "Pineapple"]
```

You can concatenate as many arrays as you want and the `concat()` method will merge them all into a single array.

Its also possible to add multiple items to an array using the concat() method. For example, the following code adds the strings "Pineapple" and "Mango" to the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
let food = fruits.concat("Pineapple", "Mango");
console.log(food); // ["Apple", "Orange", "Banana", "Pineapple", "Mango"]
```

sort()

The sort() method sorts the elements of an array in place and returns the sorted array. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values. For example, the following code sorts the array fruits:

```
let fruits = ["Banana", "Orange", "Apple"];
fruits.sort();
console.log(fruits); // ["Apple", "Banana", "Orange"]
```

The sort() method can also be used to sort numbers in an array. For example, the following code sorts the array numbers:

```
let numbers = [40, 100, 1, 5, 29, 10, 2907];
numbers.sort();
console.log(numbers); // [1, 10, 100, 2907, 29, 40, 5]
```

the sort() method modifies the original array and does not create a new array. Also, the sort() method sorts the elements as strings by default. This works well for strings ("Apple" comes before "Banana"). However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1". Because of this, the sort() method will produce incorrect result when sorting numbers.

To sort numbers correctly, you must provide a compare function. The compare function should return a negative, zero, or positive value, depending on the arguments. The following code sorts the array numbers in ascending order:

```
let compare = (a, b) => {
   return a - b;
}
let numbers = [40, 100, 1, 5, 29, 10, 2907];
numbers.sort(compare);
console.log(numbers); // [1, 5, 10, 29, 40, 100, 2907]
```

In the above code, the compare function returns a negative value when a is less than b, zero when a is equal to b, and a positive value when a is greater than b. The sort() method sorts the elements by calling the compare function repeatedly.

reverse()

The reverse() method reverses the order of the elements in an array. For example, the following code reverses the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
fruits.reverse();
console.log(fruits); // ["Banana", "Orange", "Apple"]
```

lets take a look at another example:

```
let numbers = [40, 100, 1, 5, 29, 10, 2907];
numbers.reverse();
console.log(numbers); // [2907, 10, 29, 5, 1, 100, 40]
```

The reverse() method modifies the original array and does not create a new array.

slice()

The slice() method returns a shallow copy of a portion of an array into a new array object. The original array will not be modified. For example, the following code returns a shallow copy of the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
let copy = fruits.slice();
console.log(copy); // ["Apple", "Orange", "Banana"]
```

The slice() method can also be used to return a shallow copy of a portion of an array into a new array object. For example, the following code returns a shallow copy of the first two elements of the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
let copy = fruits.slice(0, 2);
console.log(copy); // ["Apple", "Orange"]
```

The slice() method takes two arguments: the index at which to begin extraction, and the index at which to end extraction (excluded). If the second argument is omitted, the slice() method will extract all elements from the start index to the end of the array. If the first argument is omitted, the slice() method will extract all elements from the beginning of the array to the end index.

splice()

The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. The original array will be modified. For example, the following code removes the first element of the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
fruits.splice(0, 1);
console.log(fruits); // ["Orange", "Banana"]
```

The splice() method takes three arguments: the index at which to begin changing the array, the number of elements to remove, and the elements to add to the array. If the second argument is omitted, the splice() method will remove all elements from the start index to the end of the array. If the third argument is omitted, the splice() method will only remove elements from the array.

indexOf()

The indexOf() method returns the first index at which a given element can be found in the array, or -1 if it is not present. For example, the following code returns the index of the element "Banana" in the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
let index = fruits.indexOf("Banana");
console.log(index); // 2
```

This was a quick overview of the most commonly used array methods. There are many more array methods that you can use to manipulate arrays. You can find a complete list of array methods in the MDN documentation.

Remember, you don't have to memorize these, just understand the concepts. Everything is just a google search away!

In our previous lessons we have learned about loops and arrays. In this lesson, we will learn how to use loops with arrays.

Loops with Arrays

lets start with a simple example. The following code uses a for loop to print all the elements of the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
for (let i = 0; i < fruits.length; i++) {
   console.log(fruits[i]);
}
```

In the above code, the for loop starts at index 0 and keeps incrementing the index until it reaches the end of the array. The length property of the array returns the number of elements in the array. The for loop will stop when the index is equal to the length of the array.

forEach()

The above code can be rewritten using the forEach() method. The forEach() method executes a provided function once for each array element. For example, the following code uses the forEach() method to print all the elements of the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
fruits.forEach((i) => {
   console.log(i);
});
```

In the above code, the forEach() method takes a callback function as an argument. The callback function takes the current element as an argument. The forEach() method will call the callback function once for each element in the array.

from()

The from() method creates a new array from an array-like object. For example, the following code creates a new array from a string:

```
let str = "Hello";
let arr = Array.from(str);
console.log(arr); // ["H", "e", "l", "l", "o"]
```

In the above code, the from() method takes a string as an argument. The from() method creates a new array from the string. The new array contains the characters of the string.

for...of

The for...of statement creates a loop iterating over iterable objects. For example, the following code uses the for...of statement to print all the elements of the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
for (let i of fruits) {
   console.log(i);
}
```

In the above code, the for...of statement iterates over the array fruits. The for...of statement will stop when it reaches the end of the array.

for...in

The for...in statement creates a loop iterating over enumerable properties of an object. For example, the following code uses the for...in statement to print all the elements of the array fruits:

```
let fruits = ["Apple", "Orange", "Banana"];
for (let i in fruits) {
   console.log(fruits[i]);
}
```

In the above code, the for...in statement iterates over the array fruits. The for...in statement will stop when it reaches the end of the array. In objects, the for...in loop gives you the keys of the object and array is an object too. The keys in arrays are the indexes of the elements. So, the for...in loop will iterate over the indexes of the array and print the elements of the array.

and thats a wrap for this lesson. In the next lesson, we will learn about the map(), filter(), and reduce() methods.

Lets continue our journey and keep exploring more about the arrays. In this lesson, we will learn about the map(), filter(), and reduce() methods. These methods are used to transform the elements of an array. The map(), filter(), and reduce() methods are very useful and are used in many real-world applications. Lets start with the map() method.

map()

The map() method creates a new array with the results of calling a provided function on every element in the calling array. For example, the following code uses the map() method to double the elements of the array numbers:

```
let numbers = [1, 2, 3, 4, 5];
let doubledNumbers = numbers.map((i) => {
   return i * 2;
});
console.log(doubledNumbers); // [2, 4, 6, 8, 10]
```

In the above code, the map() method takes a callback function as an argument. The callback function takes the current element as an argument. The map() method will call the callback function once for each element in the array. The callback function returns the new value of the element. The map() method creates a new array with the new values returned by the callback function. The new array is assigned to the variable doubledNumbers.

The map() method is very useful when you want to transform the elements of an array. For example, the following code uses the map() method to convert the elements of the array numbers to strings:

```
let numbers = [1, 2, 3, 4, 5];
let stringNumbers = numbers.map((i) => {
   return i.toString();
```

```
});
console.log(stringNumbers); // ["1", "2", "3", "4", "5"]
```
In the above code, the map() method takes a callback function as an argument. The callback function takes the current element as an argument. The map() method will call the callback function once for each element in the array. The callback function returns the new value of the element. The map() method creates a new array with the new values returned by the callback function. The new array is assigned to the variable stringNumbers.

It is important to note that the map() method does not change the original array. The map() method creates a new array with the new values returned by the callback function. The original array remains unchanged.

filter()

The filter() is a really interesting and easy method, it creates a new array with all elements that pass the test implemented by the provided function. For example, the following code uses the filter() method to filter out the numbers greater than 3 from the array numbers:

```
let numbers = [1, 2, 3, 4, 5];
let filteredNumbers = numbers.filter((i) => {
   return i > 3;
});
console.log(filteredNumbers); // [4, 5]
```
In the above code, the filter() method takes a callback function as an argument. The callback function takes the current element as an argument. The filter() method will call the callback function once for each element in the array. The callback function returns a boolean value. If the callback function returns true, the current element will be added to the new array. If the callback function returns false, the current element will be ignored. The filter() method creates a new array with the elements that passed the test. The new array is assigned to the variable filteredNumbers.

reduce()

The reduce() method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value. For example, the following code uses the reduce() method to sum the elements of the array numbers:

```
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce((first, second) => {
   return first + second;
});
console.log(sum); // 15
```
In the above code, the reduce() method takes a callback function as an argument. The callback function takes two arguments, the first argument is the accumulator and the second argument is the current element. The reduce() method will call the callback function once for each element in the array. The callback function returns the new value of the accumulator. The reduce() method returns the final value of the accumulator. The final value of the accumulator is assigned to the variable sum.

These concepts may seem a bit confusing at first, but they are very useful and are used in many real-world applications. Lets practice these concepts by solving some problems in next lesson.

Practice Set 5

We have learnt a lot of things in the previous chapters. Let's test our knowledge by solving some problems.

Question 1

Create an array of numbers and take input from the user to add numbers to this array.

Answer

```
let numbers = [1, 2, 3, 4, 5];
let input = prompt("Enter a number: ");
input = Number(input);
numbers.push(input);
console.log(numbers);
```
The push() method is used to add an element to the end of an array. The prompt() method is used to get the user's input and the Number() function is used to convert the input to a number.

Question 2

Keep adding numbers to the array in Q1 until 0 is entered.

Answer

```
let numbers = [1, 2, 3, 4, 5];
let input = prompt("Enter a number: ");
input = Number(input);
while (input != 0) {
  numbers.push(input);
  input = prompt("Enter a number: ");
  input = Number(input);
}
console.log(numbers);
```
The while loop will continue to run as long as the condition input != 0 is true. The prompt() method is used to get the user's input and store it in the input variable. The Number() function is used to convert the input to a number. The push() method is used to add an element to the end of an array.

we can also do this with do-while loop:

```
let numbers = [1, 2, 3, 4, 5];
let input = null;
do {
```

```
  input = prompt("Enter a number: ");
  input = Number(input);
  numbers.push(input);
} while (input != 0);
console.log(numbers);
```

The do-while loop will run at least once, even if the condition is false. The prompt() method is used to get the user's input and store it in the input variable. The Number() function is used to convert the input to a number. The push() method is used to add an element to the end of an array.

Question 3

Filter for numbers deivisible by 10 from an array of numbers.

Answer

```
let numbers = [10, 29, 33, 40, 50, 61, 17, 38, 90, 7];
let filtered = numbers.filter(function (number) {
  return number % 10 == 0;
});
console.log(filtered);
```

The filter() method is used to create a new array with all elements that pass the test implemented by the provided function.
The function keyword is used to define a function. The number parameter is used to store the value of each element in the array.
The return keyword is used to return the result of the function. The % operator is used to get the remainder of a division.

Question 4

Create an array of square of given numbers.

Answer

```
let numbers = [1, 2, 3, 4, 5];
let squares = numbers.map(function (number) {
  return number * number;
});
console.log(squares);
```

The map() method creates a new array with the results of calling a provided function on every element in the calling array.
The function keyword is used to define a function. The number parameter is used to store the value of each element in the array.
The return keyword is used to return the result of the function. The * operator is used to multiply two numbers.

Question 5

Use the reduce method to calulate factorial of a given number from an array of first n natural numbers (n being the numbers whose factorial needs to be calculated).

Answer

```
let numbers = [1, 2, 3, 4, 5];
let factorial = numbers.reduce(function (x1, x2) {
  return x1 * x2;
});
console.log(factorial);
```

The reduce() method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value. The function keyword is used to define a function. The x1 and x2 parameters are used to store the value of each element in the array. The return keyword is used to return the result of the function. The * operator is used to multiply two numbers.

This one is a bit tricky. Let's break it down.

1. First, we create an array called numbers that contains the values [1, 2, 3, 4, 5].
2. Next, we call the reduce() method on the numbers array. The reduce() method is used to perform a specific operation on each element of an array and reduce the entire array to a single value.
3. The reduce() method takes a callback function as its argument, which is executed on each element of the array. This function takes two arguments: x1 (the accumulated value) and x2 (the current element of the array being processed).
4. In this example, the callback function multiplies the accumulated value (x1) by the current element of the array being processed (x2). So, for example, the first time the function is called, x1 is 1 (the initial value), and x2 is 2 (the first element of the numbers array).
5. The reduce() method continues to call the callback function for each element of the array, passing in the result of the previous function call as the new x1 value, and the next element of the array as x2. This continues until every element of the array has been processed, and the final value is returned.
6. In this example, the reduce() method returns the factorial of the numbers in the numbers array, which is the product of all the numbers from 1 to 5. So, the value of factorial is 1 * 2 * 3 * 4 * 5, which is 120.
7. Finally, we use the console.log() method to output the value of factorial to the console, which will display the value 120.

Done! You have completed this practice set. You can now move on :))

Guess the number game

Ok so we have covered a lot of stuff so far. Let's put it all together and make a game. The game will be a guessing game where the user has to guess a number between 1 and 100.

Instructions

1. Generate a random number using the Math.random() function and store it in a variable.
2. Take input from the user using the prompt() function and store it in a variable.
3. Compare the user's guess with the generated number using if-else statements and inform the user accordingly.
4. Repeat steps 2 and 3 until the user guesses the correct number.
5. Keep track of the number of turns it took the user to guess the correct number and calculate the score accordingly.
6. Terminate the program and show the final score.

Hint

```
// Generate a random number between 1 and 100
const randomNumber = Math.floor(Math.random() * 100) + 1;
```

Go ahead and try it out on your own. Once you are done, you can check the solution below.

Solution

Code

```
const randomNumber = Math.floor(Math.random() * 100) + 1;
let userGuess = prompt("Guess the number between 1 and 100");
let turns = 0;

while (userGuess != randomNumber) {
 if (userGuess < randomNumber) {
  userGuess = prompt("Your guess was too low. Try again.");
 } else {
  userGuess = prompt("Your guess was too high. Try again.");
 }
 turns++;
}

const score = 100 - turns;
console.log(`Congratulations! You guessed the number in ${turns} turns! Your score is ${score}.`);
```

Explanation

```
const randomNumber = Math.floor(Math.random() * 100) + 1;
```

This line generates a random number between 1 and 100 and stores it in the variable randomNumber.

```
let userGuess = prompt("Guess the number between 1 and 100");
```

This line takes input from the user and stores it in the variable userGuess.

```
let turns = 0;
```

This line initializes the variable turns to 0.

```
while (userGuess != randomNumber) {
 if (userGuess < randomNumber) {
  userGuess = prompt("Your guess was too low. Try again.");
 } else {
  userGuess = prompt("Your guess was too high. Try again.");
 }
 turns++;
}
```

This is a while loop that runs until the user guesses the correct number. It compares the user's guess with the generated number and prompts the user to guess again if the guess is incorrect. It also increments the variable turns by 1.

```
const score = 100 - turns;
```

This line calculates the score of the user. The score is calculated by subtracting number of turns taken to guess the correct number from 100. The score is then stored in the variable score.

```
console.log(`Congratulations! You guessed the number in ${turns} turns! Your score is ${score}.`);
```

This line displays a message to the user informing them of the number of turns it took them to guess the correct number and their score.

JavaScript in the Browser | JavaScript Tutorial in Hindi #23

JavaScript was initially created to **make web pages alive.** JS can be written right in a web page's HTML to make it interactive.
The browser has an embedded engine called the JavaScript engine or the JavaScript runtime.

JavaScript's ability in the browser is very limited to protect the user's safety. For example, a webpage on http://google.com cannot access http://codeswear.com and steal information from there.

Developer tools

Every browser has some developer tasks which makes a developer's life a lot easier.
F12 on chrome opens Dev tools.

Elements (All HTML Elements)	**Console** (All the errors + logs)	**Network** (All network requests)