MODULE 1

Introduction to programming concepts

1. Introduction

JavaScript is a popular programming language used for both front-end and back-end development.

It is a lightweight interpreted language with first-class functions, meaning functions can be treated like any other variable in the language.

JavaScript also supports Object-Oriented Programming (OOP) concepts such as classes, inheritance, and polymorphism.

Node.js is a runtime environment for executing JavaScript outside the browser, and is often used for building back-end services such as APIs. Asynchronous programming is also a key feature of JavaScript and Node.js, allowing for efficient and non-blocking I/O operations. Overall, JavaScript is a versatile language with a wide range of capabilities and applications.

Lightweight and Interpreted Language

A lightweight language refers to the fact that Javascript has a relatively small and simple syntax compared to other popular programming languages like Java or Python. This makes it easy to learn and use, especially for beginners.

Interpreted, on the other hand, refers to the way Javascript code is executed. In an interpreted language, the source code is interpreted line by line and executed by an interpreter, rather than compiled into machine code before execution.

This means that there is no need to compile your code before you can run it, and you can see the results of your code as soon as you execute it.

This makes it a quick and efficient development process. Another benefit of an interpreted language is that you can easily debug your code by running it line by line and checking for errors. The downside is that interpreted languages tend to run slower than compiled languages since the interpreter has to translate the code into machine language as it runs.
Overall, the lightweight and interpreted nature of Javascript makes it a quick and easy-to-use language, suitable for a wide range of applications.

First-class functions

In JavaScript, functions are considered "first-class citizens" or "first-class objects." This means that functions can be treated like any other variable in the language and can be passed as arguments to other functions, returned from functions, and assigned to variables.

For example, you can define a function and assign it to a variable:

```
const multiply = function(x, y) {
  return x * y;
}
```

You can also pass a function as an argument to another function:

```
const doMath = function(operation, x, y) {
  return operation(x, y);
}
doMath(multiply, 2, 3); // returns 6
```

In this example, `multiply` is passed as an argument to `doMath` along with two numeric values. `doMath` then calls `multiply` with the two numeric values as arguments and returns the result.
By taking advantage of first-class functions, JavaScript allows for more modular and reusable code. Functions can be written and used in a way that is independent of the rest of the code, making it easy to modify and extend your codebase without affecting the behavior of other functions.

Classes

Classes, inheritance, and polymorphism are important concepts in object-oriented programming (OOP) that are also applicable to JavaScript.
A class is a blueprint for creating objects that share the same properties and methods. In JavaScript, classes can be defined using the class keyword. For example:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
  }
}
```

In this example, we define a class called `Person` with a constructor method that sets the name and age properties. The `greet` method is also defined to log a greeting to the console.

Inheritance

Inheritance is the concept of creating new classes based on existing classes, inheriting their properties and methods. In JavaScript, inheritance can be achieved through the use of the `extends` keyword. For example:

```
class Developer extends Person {
  constructor(name, age, language) {
    super(name, age);
    this.language = language;
  }
  code() {
    console.log(`I'm coding in ${this.language}!`);
  }
}
```

Here, we define a `Developer` class that extends the `Person` class. The `super` keyword is used to call the parent constructor, passing in the name and age properties, and the `language` property is added. The `code` method is also defined to log a message to the console.

Polymorphism

Polymorphism in JavaScript is the ability of objects of different types to be accessed through the same interface or method. It allows you to use objects that share the same method or property name interchangeably.

For example, let's say we have a Shape class with a method called draw. We can create other classes like Circle and Rectangle that inherit from the Shape class and also have a draw method. These subclasses can implement the draw method in their own specific way.

```
class Shape {
  draw() {
    console.log("Drawing a shape");
  }
}
class Circle extends Shape {
  draw() {
    console.log("Drawing a circle");
  }
}
class Rectangle extends Shape {
  draw() {
    console.log("Drawing a rectangle");
  }
}
```

Now, we can create instances of both Circle and Rectangle, and call the draw method on both of them, even though they have different implementations of the method.

```
const c = new Circle();
const r = new Rectangle();

c.draw(); // logs "Drawing a circle"
r.draw(); // logs "Drawing a rectangle"
```

In this example, we are using the same `draw` method to draw two different shapes. This is an example of polymorphism in JavaScript, as the same interface (`draw`) is being used to interact with different objects (`Circle` and `Rectangle`) that have different implementations of the method.

Overall, polymorphism is a powerful concept in object-oriented programming that allows you to write more flexible and reusable code, and it is applicable to JavaScript as well.

Node.js is a JavaScript runtime environment

In computer science, a "runtime" is the period of time when a program is running. In the context of Node.js, it is a runtime environment - a software platform that allows you to run JavaScript code outside the web browser.

So, when we say that Node.js is a runtime environment for executing JavaScript outside the browser, it means that it is a platform or a tool that provides the necessary environment for JavaScript code to run on a computer/server, independent of the web browser. It involves the necessary libraries, APIs, and tools to execute JavaScript code on the server-side.

Asynchronous programming is also a key feature of JavaScript and Node.js

Asynchronous programming is a programming paradigm that allows the program to execute multiple tasks in parallel and not wait for one task to finish before starting another.

In the context of Node.js, which is an asynchronous, event-driven JavaScript runtime, asynchronous programming allows you to execute non-blocking I/O operations and concurrent requests efficiently.

The Node.js event loop is at the heart of asynchronous programming, and it manages asynchronous requests by registering callbacks for each operation and executing them when the operation is completed.

This means that while one operation is being processed, other operations can keep running without blocking the program.

Callbacks are a core component of asynchronous programming in Node.js and are functions that are passed as arguments to another function and called later when the operation is completed. In Node.js, when an I/O operation is requested, a callback function is passed to this function, which is executed once the I/O operation is complete.

Asynchronous programming in Node.js can also be achieved with the use of Promises and async/await. Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and allow us to write cleaner and more maintainable asynchronous code. Async/await is a more recent addition to JavaScript, and it allows developers to write asynchronous code that looks like synchronous code, making it easier to read and understand.

Overall, asynchronous programming is essential to writing efficient and scalable Node.js applications, and it is an important concept for any Node.js developer to understand.

2. Variables and Data Types

Variables and data types are fundamental concepts in programming, including JavaScript. A variable is a container that holds a value, which can be of various types. A data type is a classification of data based on the kind of value that it represents.

JavaScript is a dynamically or loosely typed language, meaning variables can store different types of data at different times throughout the program.

In JavaScript, there are **primitive data types**, and **Object or reference data types**.

The primitive data types in JavaScript include numbers, strings, booleans, null, and undefined.

The reference data types include arrays, functions, and objects which are collections of key/value pairs.

In JavaScript, primitive types are passed by value, while objects (and arrays) are passed by reference. When an object is assigned to a variable, it actually stores a reference to that object's memory location, instead of storing the actual object value. This is why they are called reference types in JavaScript.

For example, when you create an object, like this:

```
var obj = {x: 10, y: 20};
```

And then assign it to another variable:

```
var newObj = obj;
```

Both variables now reference the same object in memory, so any changes made to the object using one variable will also be reflected when using the other variable.

Primitive types, however, are stored directly in memory when assigned to a variable, and when they are passed as arguments to a function, a copy of the value is created.

This means that, if changes are made to the value inside the function, the original variable still holds the same value.

So, in summary, object types are called reference types in JavaScript because they are stored in memory and referenced by variables, while primitive types are not stored in memory by reference and are passed by value instead.

When declaring a variable in JavaScript, the keyword "var" is used, and the variable name is followed by an equal sign, and then the value. For example:

```
var myNumber = 42;
var myString = "Hello World!";
var myBoolean = true;
```

In addition to "var," there are two other keywords that can be used to declare variables in JavaScript - "let" and "const." "Let" is used to declare a block-scoped variable. "Const," on the other hand, creates a read-only reference to a value. Once initialized, the value cannot be changed.

Understanding variables and data types is critical in programming, and it allows developers to assign values, perform calculations and store data dynamically in their programs.

In summary, variables and data types are crucial parts of the JavaScript language as they allow developers to process data efficiently, which is key to building successful and efficient programs.

- Numbers - The number data type in JavaScript represents both integer and floating-point numbers. JavaScript stores numbers as 64-bit floating-point values, also known as "double-precision" values. This means that, as with any other programming language, there are some rounding errors and limitations on the range of numbers that can be represented.

- Strings - A string is a sequence of characters, such as "Hello, World!". In JavaScript, strings are represented as a series of 16-bit Unicode characters.

- Booleans - A boolean represents a logical value, which can be either true or false. Boolean values are often used for performing conditional operations, such as if/else statements and loops.

- Null - The null value in JavaScript represents the intentional absence of any object value. If a variable is assigned to null, it means that it does not currently have a value. Null is considered a primitive value in JavaScript.

- Undefined - The undefined value in JavaScript represents a variable that has been declared but has not yet been assigned a value. It is also used to represent missing function arguments. Undefined is considered a primitive value in JavaScript.

The "double" in double-precision stands for the fact that these numbers are stored using 64 bits, which is twice the size of a "single-precision" 32-bit number.

This extra space allows for greater precision when working with floating-point numbers. A floating-point number represents a value that may have a fractional component, such as 3.14, and can be expressed in scientific notation as a mantissa and exponent.

"Double-precision" is a term used to describe how JavaScript stores numbers in computer memory. Essentially, it means that all numbers in JavaScript are represented as floating-point numbers, even if they appear to be integers.

This allows for greater precision when working with numbers that have a fractional component. Double-precision means that these numbers are stored using 64 bits, which is twice the size of a "single-precision" number.

This extra space allows for greater precision when working with floating-point numbers. When we use big numbers or very small numbers, there might be some rounding errors and limitations on the range of numbers that can be represented and JavaScript uses the same storage architecture as many other programming languages

A 64-bit binary number is a number written in base-2 that contains 64 digits, each of which can be either 0 or 1. The decimal equivalent of a 64-bit binary number can be calculated by adding the decimal values of each of its digits.
For example, the binary number 0101010101010101010101010101010101010101010101010101010101010101 has 64 bits,

2. Operators in JavaScript

JavaScript operators are used to perform various operations on data values such as variables, literals, or expressions. There are various types of operators available in JavaScript but we can categorize them into the following categories:

- Arithmetic Operators: JavaScript provides arithmetic operators for basic mathematical calculations. There are several arithmetic operators available in JavaScript such as addition (+), subtraction (-), multiplication (*), division (/), modulus (%), and increment/decrement (++, --).

- Assignment Operators: There are several assignment operators in JavaScript, including the equal operator (=), plus equal operator (+=), minus equal operator (-=), and so on. These operators are used to assign a value to a variable

```
let x = 10; x += 5;
```

- Comparison Operators: JavaScript provides comparison operators to compare two values. These operators return true or false based on the comparison result. There are several comparison operators available in JavaScript such as equal to (==), not equal to (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

```
let a = 10;
let b = 20;
console.log(a == b); // false
console.log(a != b); // true
console.log(a > b); // false
console.log(a < b); // true
console.log(a >= b); // false
console.log(a <= b); // true
```

- Logical Operators: JavaScript provides logical operators to combine multiple conditions. There are three logical operators available in JavaScript such as the AND (&&), OR (||), and NOT (!) operators.

```
let a = 10;
let b = 20;
let c = 30;
console.log((a < b) && (b < c)); // true
console.log((a < b) || (b > c)); // true
console.log(!(a < b)); // false
```

String Operators: String operators are used to concatenate two or more strings together. In JavaScript, the plus (+) operator is commonly used for string concatenation. Here is an example:

```
let firstName = "John";
let lastName = "Doe";
let fullName = firstName + " " + lastName;
console.log(fullName); // Output: "John Doe"
```

In the above example, we initialize two variables firstName and lastName, then concatenate them together using the plus operator and assign the concatenated string to a variable named fullName. The output will be John Doe.

Bitwise Operators: Bitwise operators are used to perform operations on the binary representation of numbers. These operators convert numbers to their binary format, perform the operation, and then convert back to decimal format. In JavaScript, there are six bitwise operators available: AND (&), OR (|), NOT (~), XOR (^), left shift (<<), and right shift (>>). Here's an example:

```
let a = 5;  // 101
let b = 3;  // 011
let c = a & b;  // 001
console.log(c);  // Output: 1
```

In the above example, we perform a bitwise AND operation between the numbers 5 and 3. We first convert them to binary format, then perform the operation, and then convert back to decimal format. The output will be 1.

Special Operators: Special operators in JavaScript include the ternary operator (?:), comma operator (,), and typeof operator (typeof). Here's an example of each:
The ternary operator is a shorthand way of writing an if...else statement. Here's an example:

```
let age = 25;
let eligibility = (age >= 18) ? "Eligible" : "Not Eligible";
console.log(eligibility); // Output: "Eligible"
```

In the above example, we use the ternary operator to check if a person's age is equal to or greater than 18. If it is, we assign the string "Eligible" to a variable named eligibility. Otherwise, we assign the string "Not Eligible". The output will be Eligible.

The comma operator in JavaScript is a binary operator that is mainly used to execute multiple expressions sequentially and return the value of the rightmost expression. It evaluates each of its operands from left to right and discards all the values except for the last one. It is used to join expressions where multiple expressions are expected but in places where only a single expression is allowed. The comma operator can be used in loops to update multiple variables in the same expression.

Here is an example of the comma operator being used to execute multiple expressions sequentially and return the value of the rightmost expression:

```
let x = 0;
let y = (x += 1, x += 2, x += 3);
console.log(x); // Output: 6
console.log(y); // Output: 6
```

In the above example , we first initialize x to 0. Then, we use the comma operator to execute three expressions sequentially, which update the value of x. The value of y is then assigned the value of x. The output will be 6 for both x and y.

Note that while the comma operator can be useful in certain situations, it can also make the code less readable and harder to understand, so it should be used sparingly and only when necessary.

Arithmetic operators in JavaScript

In JavaScript, arithmetic operators are used to perform basic arithmetic calculations on numerical values **1**. Some of the basic arithmetic operators that are used in JavaScript are:

- Addition (+): The addition operator is used to add two or more numerical values.

```
let a = 5;
 let b = 10;
let c = a + b;
console.log(c); // Output: 15
```

- Subtraction (-): The subtraction operator is used to subtract one numerical value from another.

```
let a = 10;
let b = 5;
let c = a - b;
console.log(c); // Output: 5
```

- Multiplication (*): The multiplication operator is used to multiply two or more numerical values.

let a = 5; let b = 10; let c = a * b; console.log(c); *// Output: 50*

- Division (/): The division operator is used to divide one numerical value by another.

let a = 10; let b = 2; let c = a / b; console.log(c); *// Output: 5*

- Modulus (%): The modulus operator is used to find the remainder of dividing one numerical value by another.

let a = 10; let b = 3; let c = a % b; console.log(c); *// Output: 1*

It is important to note that the order of operations follows the same rules from mathematics. Parentheses can be used to indicate the order of operations as well.

```
let a = 10;
let b = 2;
let c = (a + b) * 2;
console.log(c); // Output: 24
```

In addition to these basic arithmetic operators, JavaScript also provides shorthand operators such as
+=,
-=,
*=,
and /=

which allow you to perform an operation and assignment in one step.

```
let a = 5;
a += 10;
console.log(a); // Output: 15
```

In summary, arithmetic operators in JavaScript are used to perform basic arithmetic calculations on numerical values and are an important part of writing JavaScript programs.