JavaScript

Functions

What are functions

Functions are self-contained blocks of code that perform specific tasks

They allow breaking down a program into smaller, more manageable pieces of code

Functions can take input parameters and return output values

Defining functions for repetitive tasks makes the program more modular and easier to understand

Functions can be called from multiple places in the code, making it easier to reuse code and maintain the program.

Why are functions useful

1. Reusability: Functions allow us to reuse code by defining a block of code once and then calling it from multiple places in the program. This reduces code duplication, saves time, and makes the program more maintainable.
2. Modularity: Functions allow us to break down a program into smaller, more manageable pieces of code. This makes the program easier to understand, debug, and modify.
3. Abstraction: Functions can abstract complex code, making it easier to use and understand. By encapsulating complex logic inside a function, we can simplify the code that uses it.
4. Scalability: Functions allow us to scale up the program easily. As the program grows, we can define more functions to handle different tasks, without worrying about how the code inside each function works.
5. Testing: Functions make testing easier. Since each function performs a specific task, we can test each function independently of the rest of the program. This makes it easier to find and fix bugs.

Function Syntax

- Function syntax includes function name, parameters, code block, and return statement.
- Function names should be descriptive and follow naming conventions.
- Parameters are optional input values that the function can take.
- The code block contains the instructions that the function executes.
- The return statement is optional and returns a value from the function.

```
function_name(parameters) {
  // code block
  return value;
}
```

Example:

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

Defining functions

there are several ways to define a function, each with its own advantages and use cases

1. Function Declaration:This function declaration is hoisted to the top of its scope, which means that it can be called anywhere in the code, even before it's defined.

```
function add(a, b) {
  return a + b
}
```

2. Function Expression: This method is useful when you need to assign a function to a variable, pass it as an argument, or use it as a callback.

```
var add = function(a, b) {
  return a + b;
};
```

3. Arrow Function: Arrow functions have a shorter syntax. It is useful for writing more concise and readable code

```
const add = (a, b) => a + b;
```

4. Function Constructor: This method is less common and less recommended than the other methods, as it can be less secure and less performant

```
var add = new Function('a', 'b', 'return a + b');
```

Calling functions

Functions can be called in several ways
1.As standalone functions: They can be defined using the function keyword and called using their name

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

sayHello(); // output: "Hello!"
```

2. As methods: Functions that are attached to an object are called methods.We call functions as methods using the object name followed by the method name and parentheses

```
var person = {
  name: "Alice",
  greet: function() {
    console.log("Hello, " + this.name + "!");
  }
};

person.greet(); // output: "Hello, Alice!"
```

3. Using the apply() or call() method: These methods allow you to call a function with a specific this value and arguments passed as an array or a comma-separated list, respectively.

```
function greetAll(names) {
  names.forEach(function(name) {
    console.log("Hello, " + name + "!");
  });
}

var friends = ["Alice", "Bob", "Charlie"];
greetAll.call(this, friends); // output: "Hello, Alice!", "Hello, Bob!", "Hello, Charlie!"
```

The return keyword

This keyword is used inside a function to specify the value that should be returned to the caller. It stops the execution of the function and returns the value.

```
// Define a function that takes an array and returns the first element

function getFirstElement(array) {
  if (array.length > 0) {
    return array[0];
  }
  return null;
}
```

Parameters and Arguments

What are Parameters

A parameter is a named variable declared in the function definition that represents a value that the function expects to receive when it is called.

Example:

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

What are Arguments

Arguments are the actual values passed into the function when it is called.

Example:

```
addNumbers(3, 4);
```

Default Parameters

Default parameters allow you to specify default values for parameters in a function. If a parameter is not passed a value, it will use the default value instead

Example:

```
function greetUser(firstName = 'Guest', lastName = '') {
  console.log(`Hello, ${firstName} ${lastName}!`);
}

greetUser();
//If a value is not passed in for either parameter when the function is called, the default value will be used instead.

greetUser('John', 'Doe');
//the firstName and lastName parameters, respectively, and override the default values.
```

Rest Parameters

Rest parameters allow you to pass an indefinite number of arguments to a function as an array.They are denoted by three dots ...

```
function sum(...numbers) {
  let result = 0;
  for (let number of numbers) {
    result += number;
  }
  return result;
}

sum(1, 2, 3);
```

Spread syntax

Spread syntax is a syntax in JavaScript that allows an iterable (e.g. an array, a string, or an object that has the iterable property) to be expanded into individual elements

Expanding an array
```
const array = [1, 2, 3];
console.log(...array); // Output: 1 2 3
```

Concatenating arrays
```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const combinedArray = [...array1, ...array2];
console.log(combinedArray); // Output: [1, 2, 3, 4, 5, 6]
```

Copying arrays
```
const array = [1, 2, 3];
const copyArray = [...array];
console.log(copyArray); // Output: [1, 2, 3]
```

Passing arguments to a function:
```
function sum(a, b, c) {
  return a + b + c;
}
const array = [1, 2, 3];
console.log(sum(...array)); // Output: 6
```

Adding properties to an object:
```
const object1 = { a: 1, b: 2 };
const object2 = { ...object1, c: 3 };
console.log(object2); // Output: { a: 1, b: 2, c: 3 }
```

Function Scope

What is function scope?

variables declared inside a function are only accessible within the function, and not outside of it.

scope refers to the area of a program where a variable, function, or object can be accessed

Global scope is the outermost scope of a program, which means that any variable or function declared outside of any function has global scope

Any variables or functions declared inside a function have function scope

```
var globalVariable = "I am a global variable";

function exampleFunction() {
  var functionVariable = "I am a function variable";
  console.log(globalVariable); // Output: I am a global variable
  console.log(functionVariable); // Output: I am a function variable
}

exampleFunction();

console.log(globalVariable); // Output: I am a global variable
console.log(functionVariable); // Output: Uncaught ReferenceError: functionVariable is not defined
```

How does javascript execute codes?

there are two main phases that code goes through before it is executed

- The compilation phase
- The execution phase

During the compilation phase, the JavaScript engine first performs lexical analysis to break the code down into tokens (such as keywords, identifiers, and operators), and then parses those tokens into an abstract syntax tree (AST). The AST represents the structure of the code in a way that the engine can understand and execute. Errors that arise here are called
Syntax errors or parse errors
Next, the engine enters the execution phase, where it actually runs the code. During this phase, it goes through the code line by line, executing each statement in turn. Errors that arise hers are called runtime errors

Function Hoisting

Hoisting is a term used in JavaScript that refers to the behavior of moving variable and function declarations to the top of their respective scopes during the compilation phase, before the code is actually executed. This means that regardless of where in the code a variable or function is declared, it is treated as if it was declared at the beginning of its scope

All function declarations are hoisted to the top of the scope
However, function expressions and arrow functions are not hoisted in the same way that function declarations are
if you try to call it before it's defined, you'll get a ReferenceError.

```
console.log(myVariable); // Output: undefined
var myVariable = "Hello, world!";
console.log(myVariable); // Output: "Hello, world!";
```

Vs

```
// function Declaration
sayHello();

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

```
// Function expression
myOtherFunction(); // Output: Uncaught ReferenceError: myOtherFunction is not defined
var myOtherFunction = function() {
  console.log("Hello, world!");
}
```

Callback functions

A callback function is a function that's passed as an argument to another function and is called by that function when a certain event occurs or when it's needed to perform a specific task.
They allow us to execute code in a specific order, even when some parts of that code take longer to run than others.

Example

```
function greeting(name, callback) {
  console.log('Hello, ' + name + '!');
  callback();
}

function sayGoodbye() {
  console.log('Goodbye!');
}

function sayGoodnight() {
  console.log('Goodnight!');
}

function sayGoodAfternoon() {
  console.log('Good afternoon!');
}

greeting('John', sayGoodbye);
greeting('Jane', sayGoodnight);
greeting('Bob', sayGoodAfternoon);
```

End

Functions

Assignment 1

What is another name for anonymous functions?

What are the types of function in javascript?

Function are defined and called. functions that are defined and immediately called without being assigned to variables are called?

Advanced Function Concepts

Advanced Function Concepts

Some other advanced concepts of JS functions to know about

1. Function Generators: These are functions that can be paused and resumed at any time, allowing for the generation of a sequence of values over time. They are created using the function* syntax.
2. Async Functions: These are functions that allow for the use of await expressions to pause and resume execution until a promise is resolved. They are created using the async keyword.
3. Promises: These are objects that represent the eventual completion or failure of an asynchronous operation and allow for chaining of multiple asynchronous operations. They are created using the Promise constructor.
4. Higher-Order Functions: These are functions that take other functions as arguments or return functions as their results. They are used extensively in functional programming and can be used to compose complex behaviors out of simpler functions.
5. Function Composition: This involves combining two or more functions to create a new function that performs the behavior of both functions in sequence. It is commonly used in functional programming to create reusable and composable functions.
6. Closure: This refers to the ability of a function to access variables in its outer lexical environment, even after the outer function has returned. It is often used to create private variables and functions in JavaScript.
7. Currying: This involves breaking down a function that takes multiple arguments into a series of functions that each take a single argument. It can be used to create specialized functions that can be reused in multiple contexts.
8. Function Composition: This is the process of combining multiple functions to create a new function that performs a complex task.Function Memoization: This is a technique for optimizing function performance by caching the results of expensive function calls and returning the cached result when the same inputs are provided again.