Functions in JavaScript

Agenda :

1. **Functions**
2. **Function Declaration**
3. **Calling a Function**
4. **Parameters and Arguments**
5. **Default Parameters**
6. **Function Return**
7. **Helper Functions**
8. **Function Expressions**
9. **Types of Functions**
10. **Function execution**
11. **Call Stack**
12. **JavaScript Hoisting**
13. **JavaScript Recursion**

Functions

```
// Defining the function:
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function:
sum(3, 6); // 9
```

- Functions are one of the fundamental building blocks in JavaScript.
- A *function* is a reusable set of statements to perform a task or calculate a value.
- Functions can be passed one or more values and can return a value at the end of their execution.
- In order to use a function, you must define it somewhere in the scope where you wish to call it.
- The example code provided contains a function that takes in 2 values and returns the sum of those numbers.

JavaScript Function

A function is a block of code that performs a specific task.

Suppose you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- a function to draw the circle
- a function to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

JavaScript also has a huge number of inbuilt functions. For example, `Math.sqrt()` is a function to calculate the square root of a number.

Function Declarations

- In JavaScript, there are many ways to create a function. One way to create a function is by using a *function declaration.*
- Just like how a variable declaration binds a value to a variable name, a function declaration binds a function to a name, or an *identifier.*

The syntax to declare a function is:

```
function nameOfFunction () {
    // function body
}
```

- A function is declared using the `function` keyword.
- The basic rules of naming a function are similar to naming a variable. It is better to write a descriptive name for your function. For example, if a function is used to add two numbers, you could name the function `add` or `addNumbers`.
- The body of function is written within `{}`.

For example,

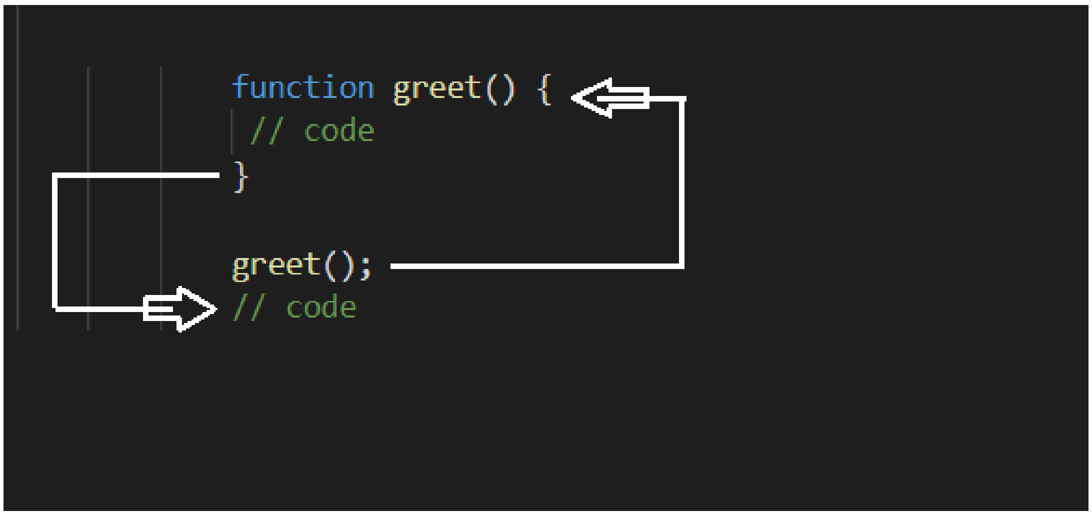
```
// declaring a function named greet()
function greet() {
    console.log("Hello there");
}
```

Calling a Function

In the above program, we have declared a function named `greet()`. To use that function, we need to call it.

Here's how you can call the above `greet()` function.

```
// function call
greet();
```



```
function greet() {
 // code
}

greet();
// code
```

working of a function in JavaScript

Example 1: Display a Text

```
// program to print a text
// declaring a function
function greet() {
    console.log("Hello there!");
}

// calling the function
greet();
```

Output:

```
Hello there!
```

Exercise :

Imagine that you manage an online store. When a customer places an order, you send them a thank you note. Let’s create a function to complete th task:

- Define a function called `sayThanks()` as a function declaration.
- In the function body of `sayThanks()` , add code such that the function writes the following thank you message to the console when called: `'Thank you for your purchase! We appreciate your business.'`

Code :

```
function sayThanks(){
  console.log ('Thank you for your purchase! We appreciate your business.') ;
}
sayThanks() ;
```

Functions can be called as many times as you need them.

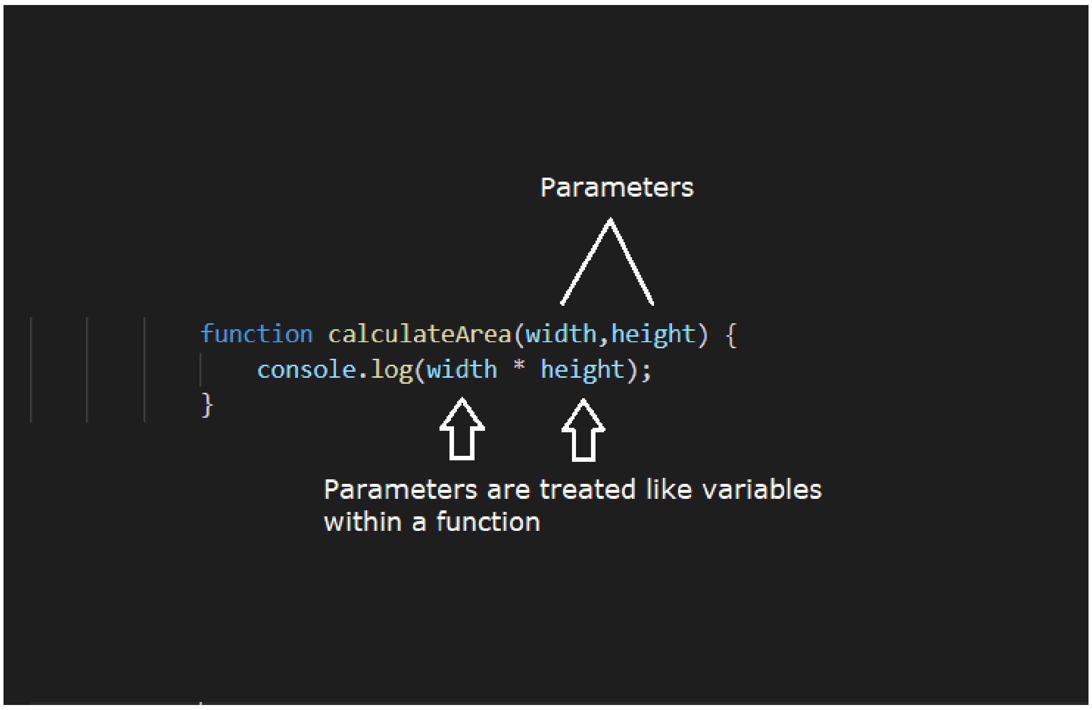
Imagine that three customers placed an order and you wanted to send each of them a thank you message. Update your code call `sayThanks()` three times.

```
function sayThanks(){
  console.log ('Thank you for your purchase! We appreciate your business.') ;
}
sayThanks() ;
sayThanks() ;
sayThanks() ;
```

Parameters and Arguments

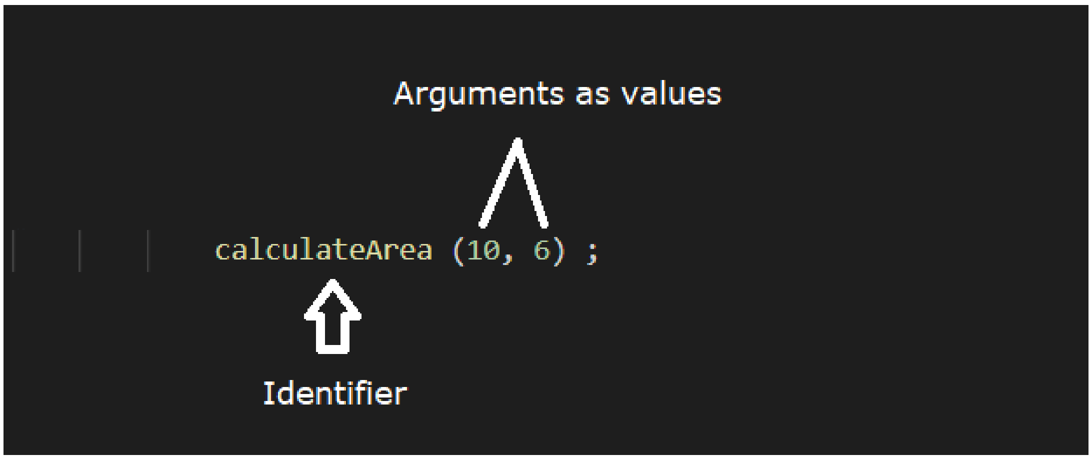
So far, the functions we’ve created execute a task without an input. However, some functions can take inputs and use the inputs to perform a task. Whe declaring a function, we can specify its *parameters*. Parameters allow functions to accept input(s) and perform a task using the input(s). We us parameters as placeholders for information that will be passed to the function when it is called.

Let’s observe how to specify parameters in our function declaration:



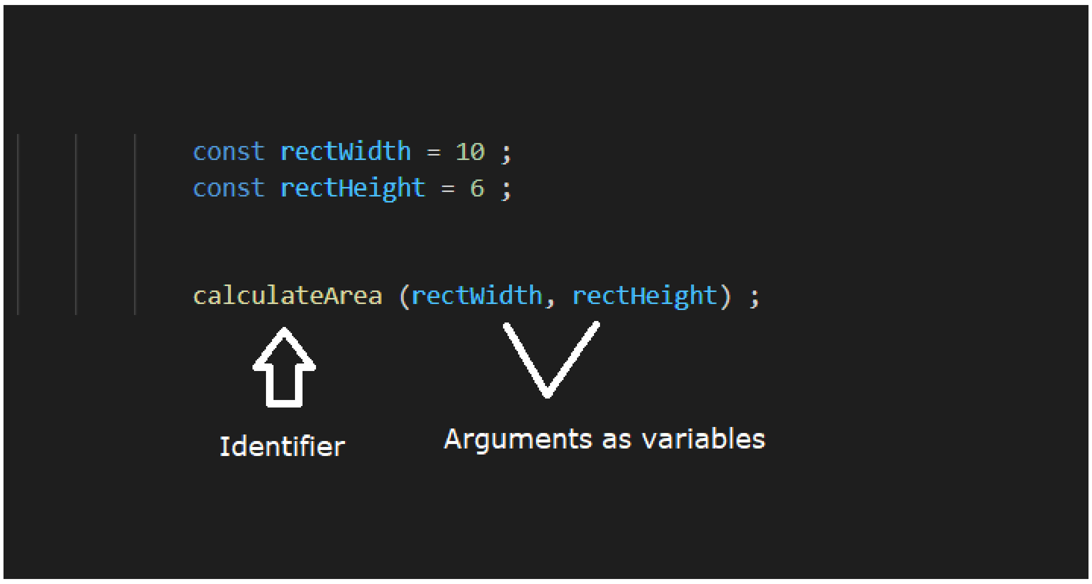
```
function calculateArea(width,height) {
    console.log(width * height);
}
```

- In the diagram above, `calculateArea()` , computes the area of a rectangle, based on two inputs, `width` and `height` .
- The parameters are specified between the parenthesis as `width` and `height` , and inside the function body, they act just like regular variables. `width` and `height` act as placeholders for values that will be multiplied together.
- When calling a function that has parameters, we specify the values in the parentheses that follow the function name. The values that are passed to the function when it is called are called *arguments*. Arguments can be passed to the function as values or variables.



```
calculateArea (10, 6) ;
```

In the function call above, the number 10 is passed as the widthand 6is passed as height. Notice that the order in which arguments are passed ar
assigned follows the order that the parameters are declared.



```
const rectWidth = 10 ;
const rectHeight = 6 ;

calculateArea (rectWidth, rectHeight) ;
```

The variables `rectWidth` and `rectHeight` are initialized with the values for the height and width of a rectangle before being used in the functic call.

By using parameters, `calculateArea()` can be reused to compute the area of any rectangle! Functions are a powerful tool in computer programmir so let's practice creating and calling functions with parameters.

Exercise :

1. The `sayThanks()` function works well, but let's add the customer's name in the message. Add a parameter called `name` to the function declaration for `sayThanks()` .

Code :

```
function sayThanks(name){
  console.log('Thanks for the purchase ') ;
}

sayThanks('name') ;
```

2. With `name` as a parameter, it can be used as a variable in the function body of `sayThanks()` . Using `name` and string concatenation, change the thank you message into the following:

```
  function sayThanks(name) {
console.log('Thank you for your purchase ' + name + '! We appreciate your business.');
}

sayThanks('John');
```

Default Parameters

One of the features added in ES6 is the ability to use *default parameters*. Default parameters allow parameters to have a predetermined value in cas there is no argument passed into the function or if the argument is `undefined` when called.

Take a look at the code snippet below that uses a default parameter:

```
function greeting (name = 'stranger') {
  console.log(`Hello, ${name}!`)
}
```

```
greeting('Nick') // Output: Hello, Nick!
greeting() // Output: Hello, stranger!
```

- In the example above, we used the `=` operator to assign the parameter `name` a default value of `'stranger'`. This is useful to have in case we ever want to include a non-personalized default greeting!
- When the code calls `greeting('Nick')` the value of the argument is passed in and, `'Nick'`, will override the default parameter of `'stranger'` to log `'Hello, Nick!'` to the console.
- When there isn't an argument passed into `greeting()`, the default value of `'stranger'` is used, and `'Hello, stranger!'` is logged to the console.

By using a default parameter, we account for situations when an argument isn't passed into a function that is expecting an argument.

Let's practice creating functions that use default parameters.

1. The function `makeShoppingList()` creates a shopping list based on the items that are passed to the function as arguments. Imagine that you always purchase milk, bread, and eggs every time you go shopping for groceries. To make creating a grocery list easier, let's assign default values to the parameters in `makeShoppingList()`. Change the parameters of `makeShoppingList()` into default parameters :

- Assign 'milk' as the default value of `item1`.
- Assign 'bread' as the default value of `item2`.
- Assign 'eggs' as the default value of `item3`.

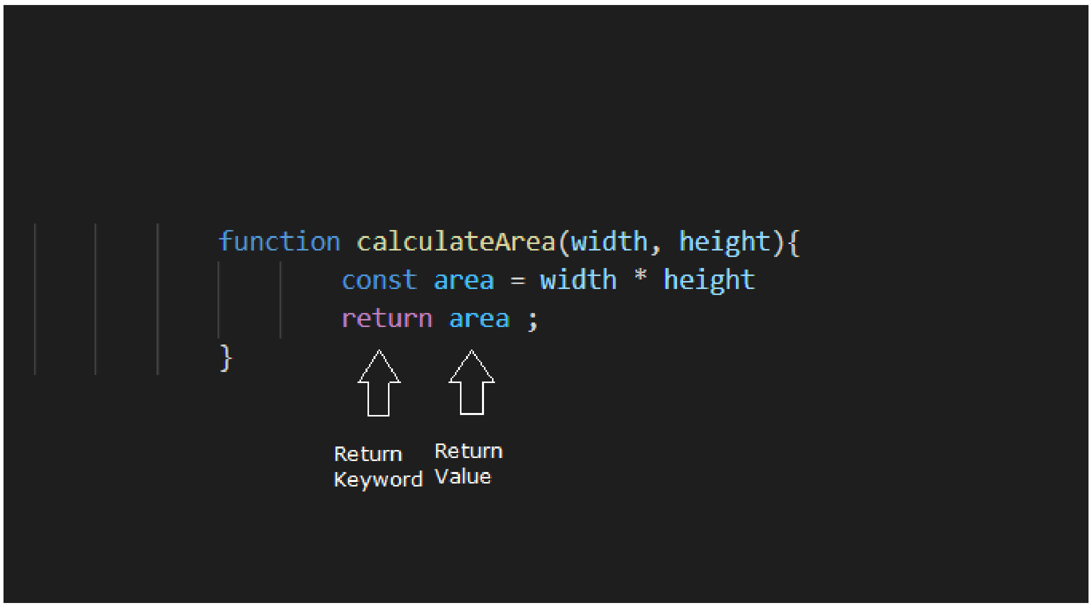
```
function makeShoppingList(item1 = 'milk', item2 = 'bread', item3 = 'eggs'){
  console.log(`Remember to buy ${item1}`);
  console.log(`Remember to buy ${item2}`);
  console.log(`Remember to buy ${item3}`);
}
```

Function Return

When a function is called, the computer will run through the function's code and evaluate the result of calling the function. By default that resulting valu is `undefined.`

```
function rectangleArea(width, height) {
  let area = width * height;
}
console.log(rectangleArea(5, 7)) // Prints undefined
```

In the code example, we defined our function to calculate the areaof a widthand height parameter. Then rectangleArea()is invoked with th arguments 5and 7. But when we went to print the results we got undefined.Did we write our function wrong? No! In fact, the function worked fine, and th computer did calculate the area as 35,but we didn't capture it. So how can we do that? With the keyword return



```
function calculateArea(width, height){
        const area = width * height
        return area ;
}
```

To pass back information from the function call, we use a return statement. To create a return statement, we use the `return` keyword followed by tl value that we wish to return. Like we saw above, if the value is omitted, `undefined` is returned instead.

When a `return` statement is used in a function body, the execution of the function is stopped and the code that follows it will not be executed. Look the example below:

```
function rectangleArea(width, height) {
  if (width < 0 || height < 0) {
    return 'You need positive integers to calculate area!';
  }
  return width * height;
}
```

If an argument for `width` or `height` is less than `0` , then `rectangleArea()` will return `'You need positive integers to calculat area!'` . The second return statement `width * height` will not run.

The `return` keyword is powerful because it allows functions to produce an output. We can then save the output to a variable for later use**.**

Exercise :

Imagine if we needed to order monitors for everyone in an office and this office is conveniently arranged in a grid shape. We could use a function to he us calculate the number of monitors needed!

1. Declare a function `monitorCount()` that has two parameters. The first parameter is `rows` and the second parameter is `columns` .

2. Let's compute the number of monitors by multiplying `rows` and `columns` and then returning the value. In the function body of the function you just wrote, use the `return` keyword to return `rows * columns` .

3. Now that the function is defined, we can compute the number of monitors needed. Let's say that the office has 5 rows and 4 columns. Declare a variable named `numOfMonitors` using the `const` keyword and assign `numOfMonitors` the value of invoking `monitorCount()` with the arguments `5` and `4` .

4. To check that the function worked properly, log `numOfMonitor` to the console.

Output :

```
function monitorCount(rows, columns) {
  return rows * columns;
}

const numOfMonitors = monitorCount(5, 4);

console.log(numOfMonitors);
```

Helper Functions

We can also use the return value of a function inside another function. These functions being called within another function are often referred to as *help functions*. Since each function is carrying out a specific task, it makes our code easier to read and debug if necessary.

If we wanted to define a function that converts the temperature from Celsius to Fahrenheit, we could write two functions like:

```
function multiplyByNineFifths(number) {
  return number * (9/5);
};

function getFahrenheit(celsius) {
  return multiplyByNineFifths(celsius) + 32;
};

getFahrenheit(15); // Returns 59
```

In the example above:

- `getFahrenheit()` is called and `15` is passed as an argument.
- The code block inside of `getFahrenheit()` calls `multiplyByNineFifths()` and passes `15` as an argument.
- `multiplyByNineFifths()` takes the argument of `15` for the `number` parameter.
- The code block inside of `multiplyByNineFifths()` function multiplies `15` by `(9/5)` , which evaluates to `27` .
- `27` is returned back to the function call in `getFahrenheit()` .
- `getFahrenheit()` continues to execute. It adds `32` to `27` , which evaluates to `59` .
- Finally, `59` is returned back to the function call `getFahrenheit(15)` .

We can use functions to section off small bits of logic or tasks, then use them when we need to. Writing helper functions can help take large and difficu tasks and break them into smaller and more manageable tasks.

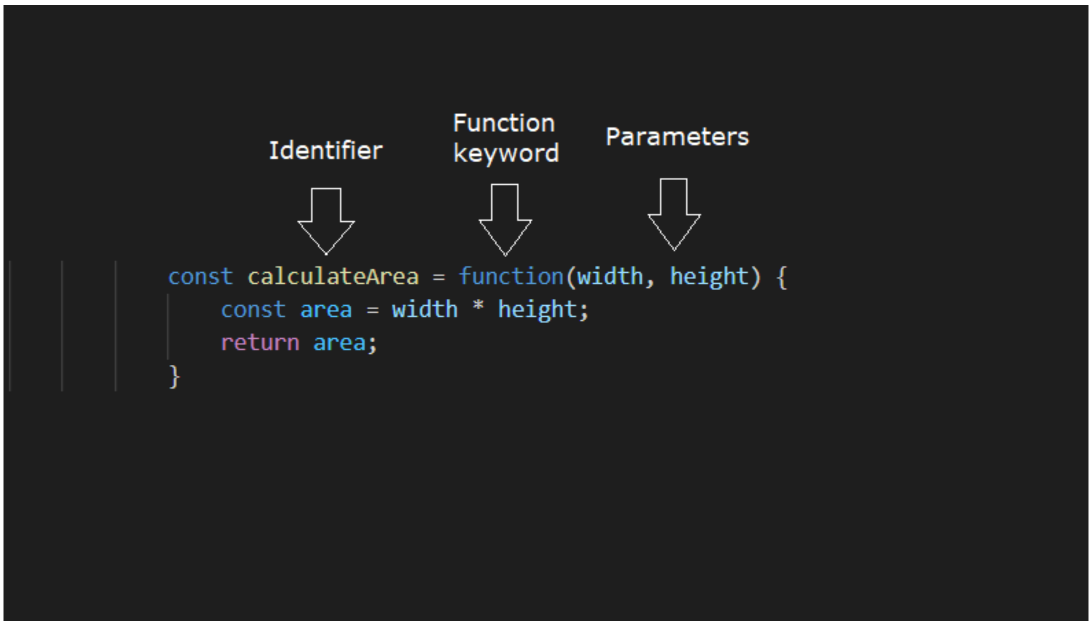
Benefits of Using a Function

- Function makes the code reusable. You can declare it once and use it multiple times.
- Function makes the program easier as each small task is divided into a function.
- Function increases readability.

Function Expressions

Another way to define a function is to use a *function expression*. To define a function inside an expression, we can use the `function` keyword. In function expression, the function name is usually omitted. A function with no name is called an *anonymous function*. A function expression is often store in a variable in order to refer to it.

Consider the following function expression:



```
const calculateArea = function(width, height) {
    const area = width * height;
    return area;
}
```

To declare a function expression:

1. Declare a variable to make the variable's name be the name, or identifier, of your function. Since the release of ES6, it is common practice to use `const` as the keyword to declare the variable.

2. Assign as that variable's value an anonymous function created by using the `function` keyword followed by a set of parentheses with possible parameters. Then a set of curly braces that contain the function body.

To invoke a function expression, write the name of the variable in which the function is stored followed by parentheses enclosing any arguments beir passed into the function.

```
variableName(argument1, argument2)
```

Unlike function declarations, function expressions are not hoisted so they cannot be called before they are defined.

Types of Functions in JavaScript

Arrow Function

ES6 introduced *arrow function syntax*, a shorter way to write functions by using the special "fat arrow" `() =>` notation.

Arrow functions remove the need to type out the keyword `function` every time you need to create a function. Instead, you first include the paramete inside the `( )` and then add an arrow `=>` that points to the function body surrounded in `{ }` like this:

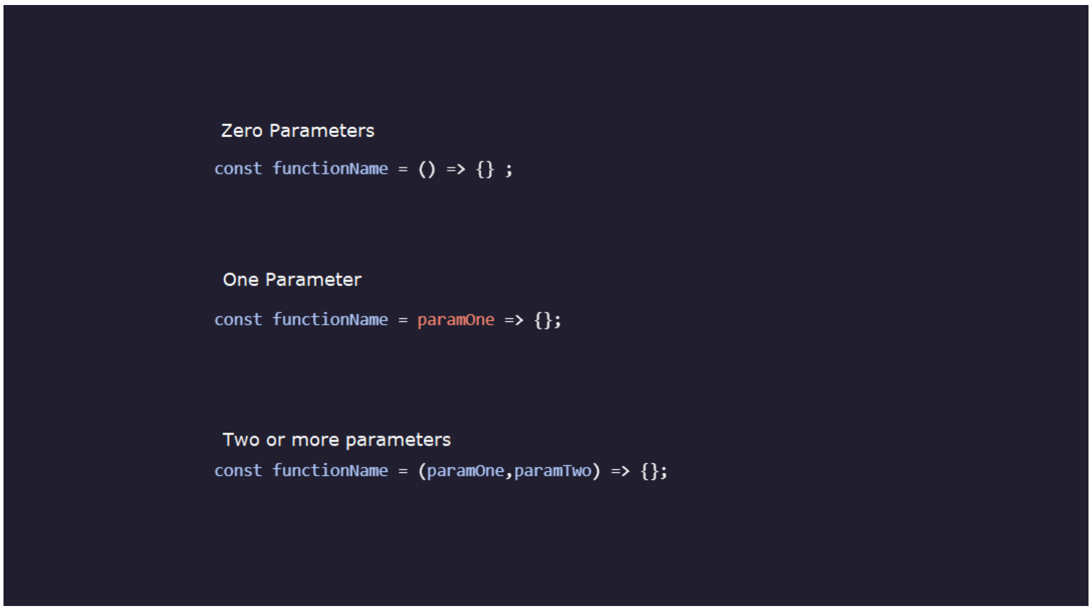
```
const rectangleArea = (width, height) => {
  let area = width * height;
  return area;
};
```

It's important to be familiar with the multiple ways of writing functions because you will come across each of these when reading other JavaScript code.

Concise Body Arrow Functions

JavaScript also provides several ways to refactor arrow function syntax. The most condensed form of the function is known as *concise body*. We explore a few of these techniques below:

1. Functions that take only a single parameter do not need that parameter to be enclosed in parentheses. However, if a function takes zero or multiple parameters, parentheses are required.


Zero Parameters

```
const functionName = () => {} ;
```

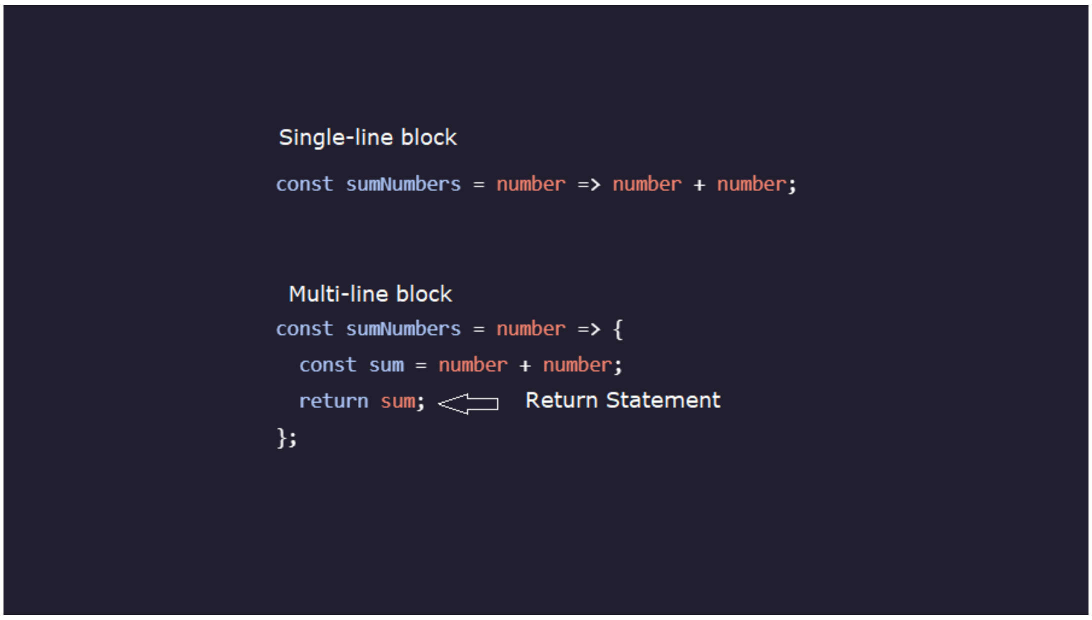
One Parameter

```
const functionName = paramOne => {};
```

Two or more parameters

```
const functionName = (paramOne,paramTwo) => {};
```

2. A function body composed of a single-line block does not need curly braces. Without the curly braces, whatever that line evaluates will be automatically returned. The contents of the block should immediately follow the arrow => and the return keyword can be removed. This is referred to as implicit return.


Single-line block

```
const sumNumbers = number => number + number;
```

Multi-line block

```
const sumNumbers = number => {
  const sum = number + number;
  return sum;
};
```

So if we have a function:

```
const squareNum = (num) => {
  return num * num;
};
```

We can refactor the function to:

```
const squareNum = num => num * num;
```

Anonymous function

Anonymous functions in JavaScript, are the functions that do not have any **name** or **identity**. Just like, you have a name by which everyone calls yo
or identifies you. But, the anonymous functions, do not have any name, so we cannot call them like any other function in JavaScript.

Syntax :

```
let variableName = function () {
    //your code here
}

variableName();     //Can call the anonymous function through this
```

Explanation:

We write the anonymous function in JavaScript by writing function() followed by the parentheses, where we may pass the parameters, and skipping th
function name. Then in a pair of curly braces {}, we write our desired code. Finally, we assign this function to a variable. We can later use the variab
whenever we need the value of the anonymous function.

Does Anonymous Functions Always Need to Store their Value Inside Variables? An anonymous function in JavaScript is not accessible after its initi
creation. Therefore, we need to assign it to a variable, so that we can use its value later. They are always invoked (called) using the variable name.

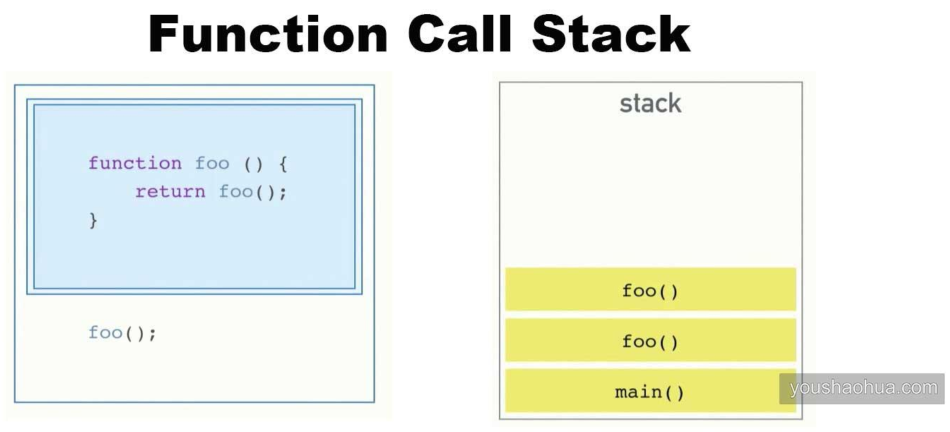
Also, we create anonymous functions in JavaScript, where we want to use functions as **values**. In other words, we can store the value returned by a
anonymous function in a variable. In the above example, we stored the value returned by the function in the variable variableName.

Key Takeaways:

- Anonymous functions do not have any name.
- We can write an anonymous function using the function keyword, followed by parentheses ().
- We can write the *function statements*, just like we do for any other javaScript function inside the curly parentheses {}.
- We store the results returned by an anonymous function in **variables**.

Function execution

- JavaScript is an interpreted language which means that it executes the code line by line on JavaScript runtime environment.
- The behavior is same of JavaScript on both the browser side and server-side. In case of executing synchronous code like functions, processing variables, conditional statements, loops, All of them are done at the call-stack level.
- Execution of synchronous code is done inside the call-stack. JavaScript execution environment pushes the code line-by-line in the call stack and one by one executes it.



Call Stack

- Call-stack internally uses the stack data structure.
- It's a LIFO data structure which stands for Last-In-First-Out.
- JavaScript execution environment reads the code line-by-line starting from first line and then starts pushing the code in the call-stack. Here one thing to note is all the asynchronous code is not pushed in call-stack. How it is handled by JavaScript is something we will going to discuss in the later slides.
- If a function is pushed in the call-stack, it will only be removed once we return or exit from the function then only it will be popped from the

call-stack.

- Like stack data-structure call-stack also supports PUSH,POP,PEEK,TOP operations.

JavaScript Hoisting

Hoisting in JavaScript is a behavior in which a function or a variable can be used before declaration. For example,

```
// using test before declaring
console.log(test);   // undefined
var test;
```

The above program works and the output will be undefined. The above program behaves as

```
// using test before declaring
var test;
console.log(test); // undefined
```

Since the variable test is only declared and has no value, `undefined` value is assigned to it.

Variable Hoisting

In terms of variables and constants, keyword `var` is hoisted and `let` and `const` does not allow hoisting.

For example,

```
// program to display value
a = 5;
console.log(a);
var a; // 5
```

In the above example, variable a is used before declaring it. And the program works and displays the output 5. The program behaves as:

```
// program to display value
var a;
a = 5;
console.log(a); // 5
```

However in JavaScript, initializations are not hoisted. For example,

```
// program to display value
console.log(a);
var a = 5;
```

Output:

```
undefined
```

The above program behaves as:

```
var a;
console.log(a);
a = 5;
```

Only the declaration is moved to the memory in the compile phase. Hence, the value of variable a is `undefined` because a is printed without initializir it.

Also, when the variable is used inside the function, the variable is hoisted only to the top of the function. For example,

```
// program to display value
var a = 4;

function greet() {
    b = 'hello';
    console.log(b); // hello
    var b;
}

greet(); // hello
console.log(b);
```

Output:

```
hello
Uncaught ReferenceError: b is not defined
```

In the above example, variable b is hoisted to the top of the function `greet` and becomes a local variable. Hence b is only accessible inside tl function. b does not become a global variable.

> **Note**: In hoisting, the variable declaration is only accessible to the immediate scope.

If a variable is used with the `let` keyword, that variable is not hoisted. For example,

```
// program to display value
a = 5;
console.log(a);
let a; // error
```

Output:

```
Uncaught ReferenceError: Cannot access 'a' before initialization
```

While using `let` , the variable must be declared first.

Function Hoisting

A function can be called before declaring it. For example,

```
// program to print the text
greet();

function greet() {
    console.log('Hi, there.');
}
```

Output :

```
Hi, there
```

In the above program, the function `greet` is called before declaring it and the program shows the output. This is due to hoisting.

However, when a function is used as an **expression**, an error occurs because only declarations are hoisted. For example;

```
// program to print the text
greet();

let greet = function() {
    console.log('Hi, there.');
}
```

Output :

```
Uncaught ReferenceError: greet is not defined
```

If varwas used in the above program, the error would be:

```
Uncaught TypeError: greet is not a function
```

> Note: Generally, hoisting is not performed in other programming languages like Python, C, C++, Java. Hoisting can cause undesirable outcomes in your program. And it is best to declare variables and functions first before using them and avoid hoisting. In the case of variables, it is better to use let than var.

Interview Questions

> Name and explain with example types of functions is JS

The types of function are:

- Named - These type of functions contains name at the time of definition. For Example:

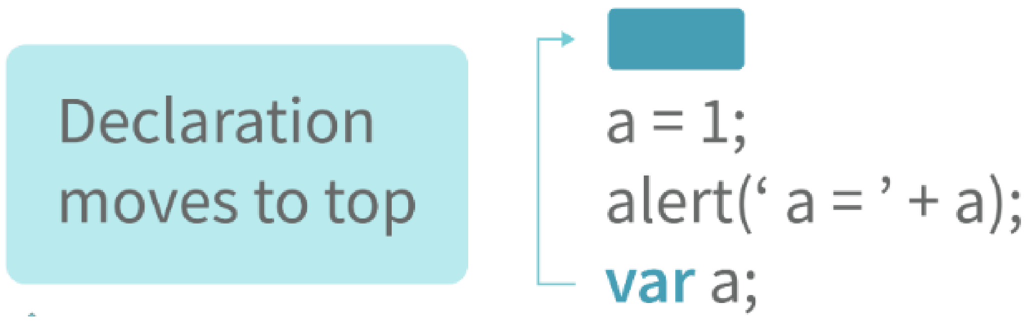
```
function display()
{
    document.writeln("Named Function");
}
display();
```

- Anonymous - These type of functions doesn't contain any name. They are declared dynamically at runtime.

```
var display=function()
{
    document.writeln("Anonymous Function");
}
display();
```

Explain Hoisting in javaScript

Hoisting is the default behavior of JavaScript where all the variable and function declarations are moved on top.



This means that irrespective of where the variables and functions are declared, they are moved on top of the scope. The scope can be both local ar global.

Example 1:

```
hoistedVariable = 3;
console.log(hoistedVariable); // outputs 3 even when the variable is declared after it is initialized
var hoistedVariable;
```

What do you mean by Self Invoking Functions?

Without being requested, a self-invoking expression is automatically invoked (initiated). If a function expression is followed by (), it will execu automatically. A function declaration cannot be invoked by itself.

Normally, we declare a function and call it, however, anonymous functions may be used to run a function automatically when it is described and will n be called again. And there is no name for these kinds of functions.

What is currying in JavaScript?

Currying is an advanced technique to transform a function of arguments n, to n functions of one or fewer arguments.

Example of a curried function:

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

add(3)(4)
```

Explain Closures in JavaScript

Closures are an ability of a function to remember the variables and functions that are declared in its outer scope.

```
var Person = function(pName){
  var name = pName;

  this.getName = function(){
    return name;
  }
}

var person = new Person("Neelesh");
console.log(person.getName());
```

Let’s understand closures by example:

```
function randomFunc(){
  var obj1 = {name:"Vivian", age:45};

  return function(){
    console.log(obj1.name + " is "+ "awesome"); // Has access to obj1 even when the randomFunc function is executed

  }
}

var initialiseClosure = randomFunc(); // Returns a function

initialiseClosure();
```

The function randomFunc() gets executed and returns a function when we assign it to a variable:

```
var initialiseClosure = randomFunc();
```

The returned function is then executed when we invoke initialiseClosure:

```
initialiseClosure();
```

The line of code above outputs “Vivian is awesome” and this is possible because of closure.

```
console.log(obj1.name + " is "+ "awesome");
```

When the function randomFunc() runs, it seems that the returning function is using the variable obj1 inside it:

Therefore randomFunc(), instead of destroying the value of obj1 after execution, **saves the value in the memory for further reference.** This is th reason why the returning function is able to use the variable declared in the outer scope even after the function is already executed. **This ability of function to store a variable for further reference even after it is executed is called Closure.**