Basic Constructs: Intro to Functions and Scopes

Relevel
by Unacademy

Educator Introduction (10 mins)

Please recap the following concepts covered in the previous sessions.

- Variables
- Conditional statements
- Iterative statements

List of concepts involved (5 mins)

- What are functions?
- How to write a function?
- Invoking a function
- Advantages of using functions
- Function declaration vs Function expression
- Function Hoisting
- What is scope?
- Global vs Local scope

What are functions? (20 mins)

A function is a piece of code that performs a specific task and which can be reused.
Suppose you need to write a program to find if the sum of two numbers is greater than 10.
You could write the following functions for it:

- A function that adds two numbers.
- A function which checks if the number is greater than 10.

Functions make our code more readable and reusable

How to write a function? (20 mins)

A **function declaration** consists of the *function* keyword, followed by:

- The **name** of the function.
- A list of **parameters** to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, {...}.

```
1  // declaring a function named greet()
2  function greet() {
3      console.log("Hello there");
4  }
5
```

Function with parameter and return value

A function can also have one or more parameters and return values. For example, the function *square* defined below has a single parameter called *number* and returns the square of the number.

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

When we do not explicitly return any value from the function, *undefined* is returned by default.

Invoking a function (25 mins)

We can **invoke/call** the function *greet* using *greet()*

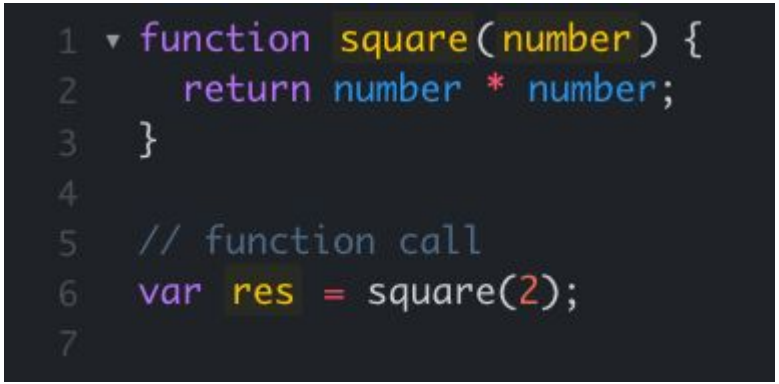
```
function greet() {
    console.log("Hello there");
}

// function call
greet();
```

Function Arguments vs Function Parameters

If the function has one or more parameters, we can pass values to these parameters using function arguments

- Function **parameters** are the variable names listed in the function definition.
- Function **arguments** are the real values passed to (and received by) the function.



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

// function call
var res = square(2);
```

For example, we can pass the argument *2* to the *square* function defined above using *square(2)*. This argument is received by the parameter named *number* in the function.

Advantages of using functions (10 mins)

- Functions make the code reusable. You can declare a function once and invoke it multiple times.
- Functions improve readability by abstracting the functionality of a piece of code.
- Functions allow us to divide a complex problem into smaller problems.

Function expression (15 mins)

The syntax that we used earlier for creating a function is called a **function declaration**.
There is another syntax for creating a function that is called a **function expression.**

```
1  // function expression
2  let sayHi = function() {
3    console.log( "Hi" );
4  };
5
6  sayHi();
7
```

Here, we can see a variable *sayHi* being assigned a value, the new function, created as *function() { console.log("Hi") }*. We then invoke the function *sayHi*.

Function hoisting (20 mins)

Hoisting allows functions to be safely used in code before they are declared. Consider the function *add* defined below:

```
1  add(2, 3);
2
3  function add(a, b) {
4    console.log(a + b);
5  }
6  /*
7  The result of the code above is: 5
8  */
```

Here, we might expect the invocation of the function *add* to throw an error since it is being invoked before being defined. However, function hoisting lets us use the function before it is declared in our code and hence the invocation of the *add* function does not throw any error and logs 5 to the console.

Function expressions are not hoisted

In the case of functions, only function **declarations** are hoisted–but **not** the function **expressions.** Hence, the below piece of code would throw an error.

```
1  add(2, 3);
2
3  var add = function (a, b) {
4    console.log(a + b);
5  }
6  /*
7  This results in the error:  "Uncaught TypeError: add is not
   a function"
8  */
9
```

What is scope? (30 mins)

Scope is the accessibility of variables, functions, and objects in some particular part of your code during runtime. In other words, scope determines the visibility of variables and other resources in areas of your code.

In the JavaScript language there are two types of scopes:

- Global Scope
- Local Scope

Global Scope

When you start writing JavaScript in a document, you are already in the Global scope. There is only one Global scope throughout a JavaScript document. A variable is in the Global scope if it's defined outside of a function.

```
var name = 'Hammad';

console.log(name); // logs 'Hammad'

function logName() {
    console.log(name); // 'name' is accessible here and everywhere else
}

logName(); // logs 'Hammad'

```

In the above code, the variable name is defined in the global scope and is accessible anywhere in the document.

Local scope

Variables defined inside a function are in the local scope. And they have a different scope for every call of that function. This means that variables having the same name can be used in different functions. This is because those variables are bound to their respective functions, each having different scopes, and are not accessible in other functions.

```
// Global Scope
function someFunction() {
    // Local Scope #1
    function someOtherFunction() {
        // Local Scope #2
    }
}

// Global Scope
function anotherFunction() {
    // Local Scope #3
}
// Global Scope
```

Upcoming class teaser

- Advanced problem solving
- Pattern problems

Thank You!