SUNSTONE

JavaScript

LECTURE 9 NOTES

Functions

- **What are Functions?**
  - In JavaScript, a function is a block of reusable code that performs a specific task. Functions allow you to divide your code into smaller, more manageable pieces, making it easier to understand, debug, and maintain your codebase.
  - Here's the basic syntax of a function in JavaScript:

```
function functionName(parameter1, parameter2, ...) {
  // Code to be executed
  // Return statement (optional)
}
```

  - Let's break down the parts of a function:
    - **function:** The keyword used to define a function.
    - **functionName:** The name you give to your function. It should be a valid identifier and follow naming conventions.
    - **parameters:** Optional placeholders that you can define within parentheses after the function name. They act as variables and hold values that are passed into the function when it is called. Parameters allow you to create flexible and reusable functions.
    - **code to be executed:** The statements or blocks of code that perform a specific task. This is the functionality you want your function to carry out. It can include any valid JavaScript code, such as variable declarations, control structures (if statements, loops), and other function calls.
    - **return statement:** An optional statement used to specify the value to be returned by the function. When the function encounters a return statement, it immediately exits, and the value specified is sent back to the caller. If there is no return statement or if it is omitted, the function returns undefined by default.
  - Here's an example of a simple function that calculates the sum of two numbers:

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

- In the example above, the addNumbers function takes two parameters a and b. It calculates the sum of a and b and assigns it to the sum variable. Then, it returns the value of the sum using the return statement.
- You can call the function and store its result in a variable like this:

```
var result = addNumbers(3, 5);console.log(result); // Output: 8
```

- In this case, the result will contain the value 8, which is the sum of 3 and 5 returned by the addNumbers function.
- Functions in JavaScript are quite versatile and can be assigned to variables, passed as arguments to other functions, and even defined within other functions (known as nested functions or closures). They are a fundamental building block of JavaScript and play a crucial role in creating reusable and modular code.
- JavaScript functions are used to perform operations. We can call the JavaScript function many times to reuse the code.

➢ Advantages of JavaScript Function

There are mainly two advantages of JavaScript functions:

1. **Code reusability:** We can call a function several times so it saves coding.
2. **Less coding:** It makes our program compact. We don't need to write many lines of code each time to perform a common task.

❖ Declaring a Function

- 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 a function is written within `{}`.

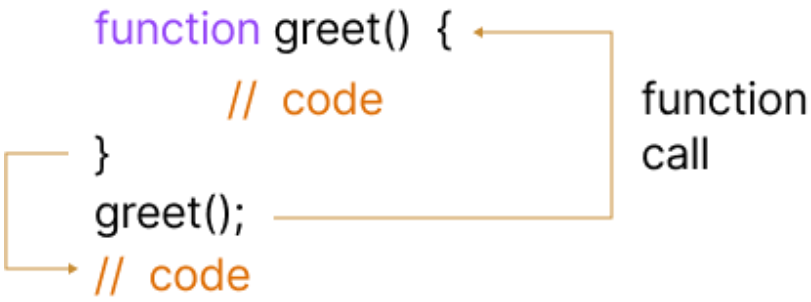
  For example,

```
// 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();
```



Working of a Function in JavaScript

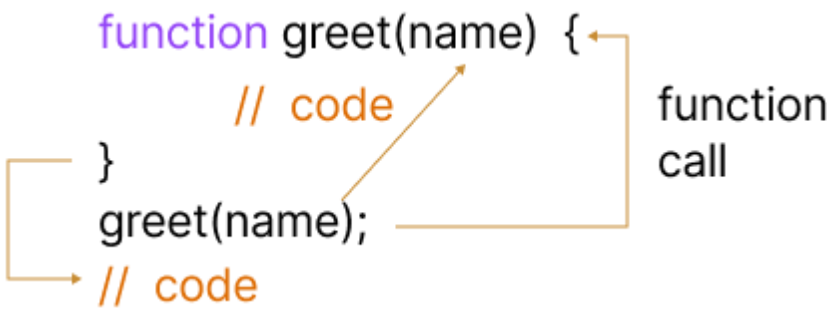
❖ Display a Text

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

Output: `Hello there!`

❖ Function Parameters

- A function can also be declared with parameters. A parameter is a value that is passed when declaring a function.



- Working of JavaScript Function with parameter

❖ Function with Parameters

```
// program to print the text// declaring a function
function greet(name) {
    console.log("Hello " + name + ":)");
}
// variable name can be different
var name = "Sunstone");
// calling function
greet(name);
```

Output:

```
Hello Sunstone :)
```

- In the above program, the greet function is declared with a name parameter. Then when the function is called, an argument is passed into the function.
- **Note:** When a value is passed when declaring a function, it is called parameter. And when the function is called, the value passed is called argument.

❖ 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.

❖ Return Statement

- There are some situations when we want to return some values from a function after performing some operations. In such cases, we can make use of the return statement in JavaScript. This is an optional statement and most of the time the last statement in a JavaScript function.

  SYNTAX: The most basic syntax for using the return statement is:

  ```
  return value;
  ```

  The return statement begins with the keyword return separated by the value which we want to return from it. We can use an expression also instead of directly returning the value.

• Function Scope and Closures

❖ Function Scope

- Scope refers to the part of a program where we can access a variable. Function scope refers to the scope or visibility of variables and functions within a particular function.
- Variables declared inside a function are only accessible within that function or any nested functions inside it but JavaScript allows us to nest scopes, and variables declared in outer scopes are accessible from all inner ones. Variables can be global-, module-, or block-scoped.
- For example,

```
function myFunction() {
  var x = 10; // Variable x is only accessible within myFunction
  console.log(x);
}
```

```
myFunction(); // Output: 10
console.log(x);  //  Error:  x  is  not  defined  (outside  the
function)
```

- In the example above, the variable x is declared inside the myFunction() function using the var keyword. It is only accessible within the function, and if you try to access it outside the function, you'll get an error.

❖ Closures

- A closure is a function enclosed with references to the variables in its outer scope. Closures allow functions to maintain connections with outer variables, even outside the scope of the variables.
- Closures allows a function to access and remember its lexical scope, even when it is executed outside its original scope. In other words, a closure allows a function to retain access to variables and parameters from the outer function, even after the outer function has finished executing.
- For example,

```
function outerFunction() {
  var outerVariable = 'I am from the outer function';
  function innerFunction() {
     console.log(outerVariable); // Accessing outerVariable from
the outer function
  }
  return innerFunction;
}
var closure = outerFunction();
closure(); // Output: I am from the outer function
```

- In the example above, the innerFunction is defined within the outerFunction. It has access to the outerVariable even after the outerFunction has finished executing. When the outerFunction is called and the innerFunction is returned, it forms a closure, preserving the scope chain of the outerFunction. Calling closure() executes the innerFunction, which still has access to the outerVariable.

- Closures are often used to create private variables or to encapsulate data and behavior within a function. They provide a way to maintain state across multiple function calls and enable more advanced JavaScript programming techniques.

- **Types of Functions**

1. **Named Functions**

   These are functions with a name specified using the function keyword. They can be defined using function declarations or function expressions.

```
// Function Declaration
function square(x) {
  return x * x;
}

// Function Expression
var multiply = function(a, b) {
  return a * b;
};
```

2. **Anonymous Functions**

   These are functions without a specified name. They are often used as callbacks or assigned to variables without a function name.

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

setTimeout(function() {
  console.log('Delayed message');
}, 1000);
```

3. **Arrow Functions**

   Introduced in ECMAScript 6 (ES6), arrow functions provide a concise syntax and

lexical ‘this’ binding. They are often used in functional programming and for writing shorter function expressions.

```
var multiply = (a, b) => a * b;

var numbers = [1, 2, 3, 4];
var squaredNumbers = numbers.map((num) => num * num);
```

4. **Generator Functions**

Generator functions allow you to define functions that can be paused and resumed, returning an iterator object. They use the function* syntax and the yield keyword.

```
function* countDown(from) {
  while (from > 0) {
    yield from;
    from--;
  }
}

var iterator = countDown(5);
console.log(iterator.next().value); // Output: 5
console.log(iterator.next().value); // Output: 4
```

5. **Higher-Order Functions**

These are functions that either take one or more functions as arguments or return a function. Higher-order functions are powerful and enable functional programming techniques.

```
function applyOperation(x, y, operation) {
  return operation(x, y);
}

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

var result = applyOperation(3, 4, add); // Passing the add function as an argument
```

```
console.log(result); // Output: 7
```