

# Function

## JavaScript



# Table of Content

- Invocation patterns
- Recursion
- Generator function
- Arrow function
- Variadic functions



# Invocation patterns

(JavaScript)

# Function Invocation Patterns

- The code in a function is not executed when the function is defined
- Function is executed when it is invoked
- Functions can be invoked in 4 different ways
  - Invoking a function as a “function”
  - Invoking a function as a “method”
  - Invoking a function with a “Constructor function”
  - Invoking a function with a “apply and call”

# Invocation pattern

(invoking as function)

```
<script>  
  var obj;  
  function square(a) {  
    return a * a;  
  }  
  document.write("Square of 3 is " + square(3));  
</script>
```

# Invocation pattern

## (invoking as method)

- When a function is part of an object, it is called a method
- Method invocation is the pattern of invoking a function that is part of an object
- JavaScript will set the **this** parameter to the object where the method was invoked on
- JavaScript binds this at execution (also known as late binding)

# Invocation pattern

(invoking as method)

```
<script>
    var obj = { firstName: "Smith", lastName: "Doe",
        fullName: function() {
            return this.firstName + " " + this.lastName;
        }
    };
    // In the example above, this would be set to obj
    document.write("Full name is " + obj.fullName());
</script>
```

# Invocation pattern

## (Constructor invocation)

- The constructor invocation pattern involves putting the new operator just before the function is invoked
- If the function returns a primitive type (number, string, boolean, null or undefined)
  - The return will be ignored and instead **this** will be returned (which is set to the new object)
- If the function returns an instance of Object
  - The object will be returned instead of returning this



# Invocation pattern

## (Constructor invocation - 1)

```
<script>
    var Fullname = function(firstname, lastname) {
        // create a property fullname
        return this.fullname = firstname + ' ' + lastname;
    };
    var obj = new Fullname("Tenali", "Raman");
    document.write("Full name is " + obj.fullname);
</script>
```

# Invocation pattern

## (Constructor invocation - 2)

```
<script>
    var Fullname = function(firstname, lastname) {
        // return object
        return { fullname : firstname + ' ' + lastname }
    };
    var obj = new Fullname("Tenali", "Raman");
    document.write("Full name is " + obj.fullname);
</script>
```

# Invocation pattern

## (with `apply()` and `call()` methods)

- JavaScript functions are objects and have properties and methods
- The `call()` and `apply()` are predefined method, invoke the function indirectly
- The `call()` method uses its own argument list as arguments to the function
- The `apply()` method expects an array of values to be used as arguments

# Invocation pattern (by call() method)

```
<script>  
    var obj;  
    function square(a) {  
        return a * a;  
    }  
    document.write("Square of 3 is " + square.call(obj, 3));  
</script>
```

# Invocation pattern (by apply() method)

```
<script>
    var obj, arrSum;
    function sum(x, y) {
        return x + y;
    }
    arrSum = [5, 4];
    document.write("Sum = " + sum.apply(obj, arrSum));
</script>
```

# Function Expression

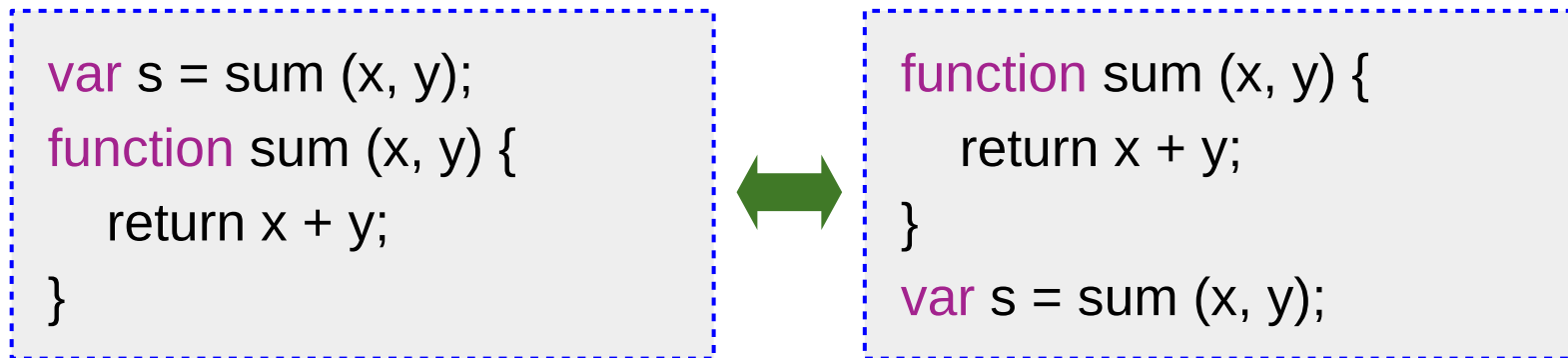
## Syntax:

```
var func = function (param-1, param-2, . . . , param-n) {  
    statement(s);  
}
```

- Variable can be used to invoke the function
- Above function is an anonymous function (a function without a name)

# Function Hoisting

- JavaScript moves variable and function declarations to top of the current scope; this is called hoisting
- Due to hoisting JavaScript functions can be called before they are declared



# Function Hoisting

- Function expressions are not hoisted onto the beginning of the scope, therefore they cannot be used before they appear in the code



# Self Invoking Function

- Function expressions can be used to self-invoke function (start automatically without being called)
- This is done by using parenthesis () -- also known as function invocation operator

## Syntax:

```
( function_expression ) ( );
```

# Self Invoking Function

- Such expressions also known as IIFE (Immediately Invokable Function Expression)

## Example:

```
( function () {  
    document.write("I am self invoking function");  
} ) ( );
```

# JavaScript Scopes

## (what?)

- Scope determines the accessibility (or visibility) of variables, objects, and functions from different parts of the code at runtime
- JavaScript has two types of scope
  - Local scope
  - Global scope
- Please note that, in JavaScript, objects and functions are also variables

# JavaScript Scopes

## (why?)

- Scope provides security to data that, in principle, shall be accessed only by intended part of the code
- If data is exposed to all parts of program then it can be modified anytime without your notice which will lead to unexpected behaviour and results
- Scope also allow use to use same names in different functions

# JavaScript Scopes

## (Local Scope)

- Variables defined within a function are local to the function
- Such variables have **local scope** and can't be accessed outside the function
- Since local variables are only recognized inside their functions, variables with the same name can be used in different functions
- Local variables are created when a function is invoked (started), and deleted when the function exits (ended)

# JavaScript Scopes

## (Local Scope)

```
<script>
function square(num) {
    var result = num * num; // variable with local scope
}
Square(3); // invoke the function
document.write("Square of number = " + result); // Exception
</script>
```

# JavaScript Scopes

## (Global Scope)

- Variables defined outside a function have **global** scope
- Such variables are visible to all the functions within a document, hence, can be shared across functions

# JavaScript Scopes

## (Global Scope)

```
<script>
var result; // variable with global scope
function square(num) {
    result = num * num;
}
square(3); // invoke the function
document.write("Square of number = " + result);
</script>
```



# JavaScript Scopes

## (Function and block scopes)

- Further, in JavaScript, there are two kinds of local scope
  - Function scope
  - Block scope
- A block of code is created with curly braces { }
- Conditional statements (if, switch) and loops (for, while, do-while) do not create new scope

# JavaScript Scopes

## (How variables are created?)

- JavaScript processes all variable declarations before executing any code, whether the declaration is inside a conditional block or other construct
- JavaScript first looks for all variable declarations in given scope and creates the variables with an initial value of undefined
- If a variable is declared with a value, then it still initially has the value undefined and takes on the declared value only when the line that contains the declaration is executed

# JavaScript Scopes

## (How variables are created?)

- Once JavaScript has found all the variables, it executes the code
- If a variable is implicitly declared inside a function -
  - Variable has not been declared with keyword “**var**”
  - And, appears on the left side of an assignment expression

*is created as a global variable*

# JavaScript Scopes

## (Global Scope)

```
<script>
function square(num) {
    result = num * num; // Automatically global variable
}
document.write("Square of number = " + result);
</script>
```

# JavaScript Scopes

## (Global Scope)

- Global variables are not automatically created in "Strict Mode"

```
<script>
function square(num) {
    "use strict";
    result = num * num;
}
document.write("Square of number = " + result); // Exception
</script>
```

# JavaScript Scopes

## (Global Scope)

- Global variables belong to window object

```
<script>
function square(num) {
    result = num * num;
}
document.write("Square of number = " + window.result);
</script>
```

# JavaScript Scopes

## (Function scope variable)

- Variables declared using keyword “**var**” within a function has function scope

# JavaScript Scopes

## (Function scope variable)

```
<script>
  function func() {
    var x = 10; // function scope variable
    {
      var x = 20; // function scope variable
      document.write("<br>x = " + x); // shall print 20
    }
    document.write("<br>x = " + x); // shall print 20
  }
</script>
```



# JavaScript Scopes

## (Block scope variable)

- ECMA script 6 has introduced keywords “**let**” and “**const**”
- Variables declared using these keywords will have block level scope
- For these variables, the braces { . . . } define a new scope

# JavaScript Scopes

## (Block scope variable - let)

```
<script>
  function func() {
    let x = 10;
    {
      let x = 20;
      document.write("<br>x = " + x); // shall print 20
    }
    document.write("<br>x = " + x); // shall print 10
  }
</script>
```

# JavaScript Scopes

## (Block scope variable - const)

```
<script>
  function func() {
    const name = "Webstack Academy";
    {
      const name = "Hello world!";
      document.write("<br>name = " + name);
    }
    document.write("<br>name = " + name);
  }
</script>
```

# JavaScript Scopes

## (Life time of variables)

- Life time of a variable is the time duration between it's creation and deletion

Variable	Keyword	Scope	Life time
Local	Var	Function	<ul style="list-style-type: none"><li>Created when function is invoked</li><li>Deleted when function exits</li></ul>
	Let, const	Function or block	<ul style="list-style-type: none"><li>Created when function is invoked</li><li>Deleted when function exits</li></ul>
Global	var, let, const	Browser Window	<ul style="list-style-type: none"><li>Created when web page is loaded in the browser window (tab)</li><li>Deleted when browser window (tab) is closed</li></ul>
	var, let, const	Block	<ul style="list-style-type: none"><li>Created when block is entered</li><li>Deleted when block is exited</li></ul>

# JavaScript Scopes

## (important notes)

- Do not create global variables unless needed
- Your global variables or functions can overwrite window variables or functions
- Opposite is also possible; any function (including window object) can overwrite your global variables and functions

# Function Closures

- In JavaScript, an inner (nested) function stores references to the local variables that are present in the same scope as the function itself, even after the function returns
- The inner function has access to the outer function's variables; this behavior is called **lexical scoping**
- However, the outer function does not have access to the inner function's variables

# Function Closures

- A closure is an inner function that has access to the outer function's variables – scope chain
- The closure has three scope chains:
  - It has access to its own block scope (variables defined between its curly brackets)
  - It has access to the outer function's variables
  - It has access to the global variables

# Function Closures

## (Example)

```
<script>
function disp() { // Parent function
    var name = "Webstack Academy"; // name is a local variable
    displayName() { // displayName() is the inner function, a closure
        alert (name); // The inner function uses variable of parent function
    }
    displayName(); // child function call
}
disp(); // parent function call
</script>
```



# JavaScript Scopes

## (Lexical Scope)

# Recursion

(JavaScript)



# Recursion

- Recursion is the process in which a function is called by itself
- Recursion is a technique for iterating over an operation by having a function call itself repeatedly until it arrives at a result

# Recursion

## (Example)

```
<script>
var factorial = function(n) {
  if (n <= 0) {
    return 1;
  } else {
    return (n * factorial(n - 1));
  }
};
document.write("factorial value"+factorial(5));
</script>
```

# Exercise

- Write a JavaScript function to find sum of digits of a number
- Write a JavaScript program to compute x raise to the power y using recursion



## Web Stack Academy (P) Ltd

#83, Farah Towers,  
1st floor, MG Road,  
Bangalore - 560001

M: +91-80-4128 9576

T: +91-98862 69112

E: [info@www.webstackacademy.com](mailto:info@www.webstackacademy.com)

*Thank  
you*