

WEB222 - Web Programming Principles

Week 2: JavaScript Functions, Scope and Closure

Agenda

- Functions
 - User-defined functions
 - Built-in / Global functions
- Variable scope
- JavaScript Closure

JavaScript Function

- A function is a "subprogram" that can be called by code external (or internal in the case of recursion) to the function.
- Like the program itself, a function is composed of a sequence of statements called the function body.
- parameters are used to **pass values** to functions.
- A function can **return a value**.
- Function names must adhere to variable name rules.
- Every function in JavaScript is a Function object.
- A function **is not executed until it is called**.

JavaScript Function

- Where to use JavaScript Functions:
 - used for event handlers on the web pages, and can be called when some events occur on the web page
 - JavaScript functions are actions or behaviors that are associated with the events on web pages.
 - associated to an object to specify the behavior of the object.
 - a method or a **member function**.
- Two Types of Functions
 - User-defined functions / custom functions
 - Built-in functions/ global functions, which are the methods of the window object.

User-defined Functions

- There are several ways to define functions. e.g. function declaration and function expression:

1. Function declaration:

Syntax:

```
function functionName ( parameter1, parameter2, ...)  
{  
    functionBody  
}
```

Example:

```
function square(number) {  
    return number * number;  
}  
console.log( square(5) );
```

Note: A function is not executed until it is called.

User-defined Functions

2. Function expression:

Syntax:

```
var functionName = function(parameter1, parameter2, ...)  
{  
    functionBody  
};
```

Example:

```
var square = function (number) {  
    return number * number;  
}  
console.log( square(5) );
```

Note: This is actually assigning an "anonymous function" to a variable

Parameter and Return Value

- Parameters are used to pass values to functions
 - Parameters are also referred to as arguments
 - Multiple parameters can be used within each function
 - Passed by value vs passed by 'reference'
 - ▶ Primitive parameters (number, string and boolean) are passed to functions by value;
 - ▶ objects (i.e. a non-primitive value, such as Array or a user-defined object) are passed to functions by 'reference'
- Return value
 - Return data type is not necessary to be specified.
 - The return statement is optional.

Example

➤ Function without return

```
greetings("Justin"); // call the greetings function; works.
```

```
function greetings (name) { // using function declaration  
approach
```

```
    console.log("Hello " + name);  
}
```

```
//go(); // will give "Exception: TypeError: go is not a function"
```

```
var go = function () { // using function expression approach
```

```
    console.log( "GO LEAFS GO" );  
};  
go();
```

Example

➤ Functions with parameters

```
// using function declaration approach
function addTwoNumbers(a, b) {
    return a + b;
}

// using function expression approach
var add2numbers = function(a, b){
    return a + b;
};

console.log( addTwoNumbers(2, 3) ); // 5
console.log( add2numbers(2, 4) ); // 6
```

Example

- Function with multiple or without parameter(s)

```
function addNumbers() {  
    var sum = 0;  
    for (var i=0; i<arguments.length; i++) {  
        sum += arguments[i];  
    }  
    return sum;  
}  
  
console.log( addNumbers() ); // 0  
console.log( addNumbers(2, 6, 8) ); //16
```

- **Default parameter:** arguments - an array-like object

More about Function

- In JavaScript, functions are **first-class** values:
 - Functions are objects, **just like regular values**, and can be assigned, passed as parameters for another function and so on.

```
function sayHi(name) { console.log("Hi, "+name) }
```

```
var hi = sayHi // assign a function to another variable
```

```
hi("John") // call the function
```

More about Function

➤ Running at place

- It is possible to create and run a function created with Function Expression at once:

```
var f1 = (function() {  
    var a, b // local variables  
    // ...  
    // and the code  
})();
```

- Running at place is mostly used when we want to do the job involving local variables.

JavaScript Built-in / Global Functions

- They are built into the JavaScript language – methods of the **Window** object.
- They have already been defined and the logic behind them has already been coded for you to use.
 - `console.log()` or `window.console.log()`
 - `prompt()`
 - `parseInt()`, `parseFloat()`
 - `Number()`, `String()`
 - `isNaN()`, `inFinite()`, `eval()`,
 - etc...

The prompt() Function

➤ Example:

```
var a = prompt("Enter first number");    // enter 11  
var b = prompt("Enter second number"); // enter 12  
var result = a + b;  
console.log("The result is " + result);
```

➤ Default values

```
var school = prompt("What is your school?", "Seneca");  
console.log("The school you are attending :\n" + school);
```

parseFloat() Function

- The `parseFloat()` function parses a string (**from left to right**) and returns a floating point number.
- If a character cannot be converted to a number, the function returns the value up to that point.
- If the first character in the string cannot be converted to a number, the function returns "**NaN**".
- The function **trim** the string before parsing.
- Example:
 - `console.log(parseFloat("15.25")); // 15.25`
 - `console.log(parseFloat("0.000345")); // 0.000345`
 - `console.log(parseFloat("0.00159+E")); // 0.00159`
 - `console.log(parseFloat(" 1234")); // 1234`
 - `console.log(parseFloat("x 1234")); // NaN`
 - `console.log(parseFloat("1 2 3 4")); // 1`
 - `console.log(parseFloat("1234ABC")); // 1234`

parseInt() Function

- The `parseInt()` function parses its first argument (a string), and then tries to return an integer of the specified radix (or base). The default base is 10.
- If a number in the string is beyond the base, `parseInt()` ignores the rest of the characters and returns an integer value up to that point.
- Example

```
console.log( parseInt('15') );      // returns 15
console.log( parseInt("15.99") );   // returns 15
console.log( parseInt('15*3') );    // returns 15
console.log( parseInt('Hello') );   // returns NaN
```

Examples with radix (or base)

- base 10 (decimal) examples

```
console.log( parseInt('15', 10) );      // returns 15  
console.log( parseInt('15*3', 10) );    // returns 15
```

- base 16 (hex) examples

```
console.log( parseInt('F', 16) );      // returns 15  
console.log( parseInt('FXX123', 16) ); // returns 15
```

- base 8 (octal) example

```
console.log( parseInt('17', 8) );      // returns 15  
console.log( parseInt('18', 8) );      // returns 1
```

- base 2 (binary) example

```
console.log( parseInt('1111', 2) );    // returns 15  
console.log( parseInt('1211', 2) );    // returns 1
```

Number() and String() Functions

- Convert an object to a number or a string.

```
var x = "12.78";
var y = 10;
var z = Number(x) + y;
console.log(z);
console.log("sss = " + String(y));
```

- Note:

- Number() can convert both integer and float numbers.
- Number() convert the parameter as a whole - no partial conversion. e.g.

```
console.log( Number("1234ABC") ); // NaN
```

Converting Without Using Functions

```
var str1 = "1234";  
var num1 = str1 * 1;
```

```
console.log(num1 + "\n" + typeof num1);
```

```
var str2 = "1234.5678";  
var num2 = +str2; // The Unary + Operator
```

```
console.log(num2 + "\n" + typeof num2);
```

isNaN() Function

- The isNaN() function is used to determine if an argument is "NaN" (not a number).
- The function checks the whole parameter, not partially.
- It does "trim" and conversion before checking.
- Example

```
console.log( isNaN("123") );    // false  
console.log( isNaN(123) );     // false  
console.log( isNaN("123 456 ") ); // true  
console.log( isNaN("+123") );   // false  
console.log( isNaN("123+") );   // true  
console.log( isNaN(" 123 ") );  // false
```

isFinite() Function

- The global `isFinite()` function determines whether the passed value is a finite number.
 - The parameter is first converted to a number.
- Example

```
console.log( isFinite(Infinity) );    // false
console.log( isFinite(NaN) );         // false
console.log( isFinite(-Infinity) );   // false
console.log( isFinite(0) );           // true
console.log( isFinite(2e12) );        // true
```

eval() Function

- One argument: a string.
 - If the string is an **expression**, eval() evaluates/executes the expression.
 - If the string is made up of JavaScript **statements**, eval() executes the statements.
- Example:

```
var x = 2;  
var y = 3;  
  
console.log("x + y");           // x + y  
  
console.log( eval("x + y") ); // 5
```

encodeURI() Function

- The encodeURI() function encodes a Uniform Resource Identifier (URI) .
- This function encodes special characters, except: ; , / ? : @ & = + \$ - _ . ! ~ * ' () #
- Example:

```
var uri = "my test.php?name=Ålan&city=Toronto";  
  
console.log( encodeURI(uri) );  
// my%20test.php?name=%C3%85lan&city=Toronto
```

toFixed() Method

- The `toFixed()` method formats a number to a specific number of digits to the right of the decimal.

```
var amount = 165.25456;  
  
console.log( amount.toFixed() ); // 165  
console.log( amount.toFixed(6) ); // 165.254560  
console.log( amount.toFixed(2) ); // 165.25
```

Note: this is a function of Number object instead of a global function

Variable Scope

- In JavaScript, variable scope can be **local** or **global** – the ways of variables to be accessed. Scope is determined by where and how a variable is declared.

1. Global variable

A variable that is declared outside any **functions** is global.

A global variable can be accessed anywhere in the current file or other files.

- Declared **outside any functions**, with or without the **var** keyword.
- Undeclared variable – “Declared” **inside a function without using the var keyword**,
 - ▶ but the variable exists only after the function has been called.

Variable Scope

2. Local variable

A variable that is declared inside a function with the `var` keyword is **local**. A local variable can only be accessed inside the function where it is declared in.

- If you reference a local variable globally or in another function, JavaScript will trigger the "**is not defined**" error. (this is different error from the "**undefined**" that is for a variable that is not initialized.)

Example

```
var display = "";    // Global variable
ident_A = 5;        // Global variable - bad practice

function someFunction() { // Start of function

    var ident_B = 15;    // Local variable
    ident_C = 34;        // Global variable - bad practice
    var ident_A = 0;
    ident_C++;           // increment ident_C by 1
    ident_A = ident_B + ident_C;
    console.log(ident_A); // show the value of ident_A inside the function

} // End of function

someFunction(); // call the function. If remove this line, what result?
console.log(ident_A); // show the value of ident_A outside the function
console.log(ident_C); // show the value of ident_C
console.log(ident_B); // what happens here?
```

About Variable Scope

- It is recommended that you
 - Avoid using global variables.
 - Always use the `var` keyword when declaring variables.
 - For large web application, use Immediately-Invoked function expressions (IIFE) to wrap JavaScript files:

```
(function() {  
    // your code  
})();
```

- Notes
 - **Functions** are the only construct that can be used to limit scope of variables.
 - In JavaScript, code blocks {} do not determine variable scope.

Scope – C vs JavaScript

Local Block in C	Block scope in JavaScript
#include <stdio.h> int main() { int x = 10; { int x = 30; printf("%d ", x); } printf("%d", x); } Output: 30 10	var a = 10; { var a = 30; b= 20; } for (var i = 0; i < 5; i++) { var c = i; } console.log(a); console.log(b); console.log(c); Output?

"Closure" in JavaScript

- In JavaScript, a closure is created when a function is nested within another function. The nested function forms a closure.
- Closures are one of the most powerful features of JavaScript.

"Closure" in JavaScript

- The nested (inner) function is private to its containing (outer) function.
- The nested (inner) function is a closure.
 - This means that a nested function can access and 'remember' the outer function's context (variable and parameters).
- Meanwhile
 - The inner function can be accessed only from statements in the outer function.
 - The outer function cannot use the arguments and variables of the inner function.

Closure Example

```
function program(prog) {  
    var school = "ICT";  
    function student(name) {  
        return "Student name: " + name + ", Program: " + prog + ", School of " +  
school;  
    }  
    return student;  
}
```

```
var bsd_student = program("BSD"); // returns the inner function with an initial value  
var cpa_student = program("CPA");  
  
var john = bsd_student("John Smith");  
var dave = cpa_student("Dave Lee");  
var dave2 = program("CPD")("Jr. Dave Lee");  
  
console.log(john);  
console.log(dave);  
console.log(dave2);
```

Function Expression and Anonymous Function

- Using Function Expression in closure

```
function program(prog) {  
    var student = function (name) {  
        return "Student name: " + name + ", \nProgram: " + prog;  
    };  
    return student;  
}
```

- Using anonymous function in closure

```
function program(prog) {  
    return function (name) {  
        return "Student name: " + name + ", \nProgram: " + prog;  
    };  
}
```

Why closures?

➤ Analogy to OOP.

- A closure makes it possible to **associate** some **data** (the environment) with a function to operate on the data.
- This is analogous to Object Oriented Programming (OOP), where we can associate some data (properties) to the object with one or more methods
- The scoped variables in the inner function become private variables, which is the "**Encapsulation**" in Object Oriented Programming.

➤ Avoid global variables.

- Global variables are not reliable.
- They are not secure.
- They may conflict with other global variables in the same application
- which may cause your code failure and their code failure.
- And it is almost impossible to test it.

Why closures?

➤ Private methods.

- The mechanism of closures, (inner function can only be accessed/ invoked by its outer function),
- implements the same concept of private methods in other Object Oriented Programming (OOP) languages, such as Java.
- Private methods provide powerful ways to manage the global namespace to keep the non-essential methods from cluttering up the public interface.

Why closures?

- **Function factory.**
 - you can create more functions with the same function body definition and different environments

```
function makeAdder(x) {  
    return function(y) {  
        return x + y;  
    };  
  
var add5 = makeAdder(5);  
var add10 = makeAdder(10);  
  
console.log(add5(2)); // 7  
console.log(add10(2)); // 12
```

More Example of Closure

- Example: increments a counter (avoid using global variable)

```
var incrementer = function() { // outer function
    var count = 0;
    return function () { // inner function
        return ++count;
    };
}

var inc = incrementer();
var count = inc();
console.log(count); // 1
count = inc();
console.log(count); // 2
console.log(inc()); // 3
```

- The inner anonymous function has access to the outer function's 'count' variable (and parameters if existed).
 - But the 'count' variable is not accessible from outside the 'incrementer' function

Advanced - Improved Counter Using Closure

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
})();

console.log('Counter value ' + counter.value()); // returns 0
console.log(counter.increment()); // counter increased but the return result is 'undefined'(due to no return)

console.log('Counter value ' + counter.value()); // returns 1
counter.increment();
counter.increment();
console.log('Two increments ' + counter.value()); // returns 3
counter.decrement();
console.log('Decrement ' + counter.value()); // returns 2
```

Thank you!

Any Questions?