Advanced JavaScript for Web Sites and Web Applications

Scope, Hoisting and *this*

3 ways to create functions : 1

- Function declaration:

```
// Define function called "foo"
function foo(arg1, arg2) {
    // function body
}
```

- To execute (aka: call) the function:

```
var result = foo('a', 1);
```

3 ways to create functions : 2

- Anonymous Function expression:

```
// Store the function in the "bar" variable
var bar = function (arg1, arg2) {
    // function body
};
```

- To call the function:

```
var result = bar('a', 1);
```

3 ways to create functions : 3

- Named function expression:

```
// Store the function in "foobar"
var foobar = function foo(arg1, arg2) {
    // function body
};
```

- To call the function:

```
var result = foobar('a', 1);
```

JavaScript and Scope

- JavaScript does not have *block scope*
- Only *functions* introduce a new scope.

JavaScript and Scope (global scope)

- Variables created outside a function definition belong to the *global scope*.
  - They will be accessible throughout the script, from anywhere (even from inside functions).

JavaScript and Scope (function scope)

- Variables defined within a function definition with the `var` keyword will only be accessible *within that function*
- Variables defined within a function definition *without* the `var` keyword will belong to the *global scope*.

Scope example:

```
// "a" belongs to global scope
var a = "hello";

function newVar() {
    // "b" belongs to function's scope
    var b = 1;
}

console.log(a); // hello
console.log(b); // Generates error
```

Scope example:

```
var a = "hello";

function newVar() {
    // "b" belongs to global scope
    b = 1;
}

// Call function, so "b" gets created
newVar();
console.log(a); // hello
console.log(b); // 1
```

JavaScript and Scope

- Consider the following:

```
var a = "hello";

function test() {
    var a = 1;
}

console.log(a);
```

- What will be displayed in the console?

JavaScript and Scope

- The console will display *hello*
- The 2 instances of the variable a are in different scopes.
  - although they have the same *label* (a), they are not the same variable

JavaScript and Scope

- Now, consider this code:

```
var a = "hello";

if (true) {
    var a = 1;
}

console.log(a);
```

- What will be displayed in the console?

JavaScript and Scope

- The console will display *1*
- Because only functions create a new scope, in this case, we are simply changing the value of the already defined variable `a`
  - Note, in this scenario, the second `var` keyword is redundant

JavaScript and Scope

- Now, consider this code:

```
var a = "hello";
if (true) {
    (function () {
        var a = 1;
        console.log(a);
    }());
    console.log(a);
}
console.log(a);
```

- What will be displayed in the console?

JavaScript and Scope

- The console will display: *1*, *hello*, *hello*
- The *IIFE* creates a temporary scope where any variable declared inside the function will not effect the outside scope.

Exercises: Question 1

- Now do **Question 1** from the *Session 3 exercises* document that is available in Moodle

Disclaimer

- In the previous examples, we saw functions being defined inside a conditional block (if).
- This has been deprecated in recent versions of the language, so don’t do it!
- We are doing it on these slides for demonstration purposes only…

JavaScript and Scope

- Remember the `typeof` operator?
- Reminder:
  - `typeof` will return the *type* of a variable (*string*, *boolean*, *number* etc.), if that variable exists in the current scope.
  - If the variable does not exist in the current scope, it will return `undefined`.

Exercises: Question 2

- Now do **Question 2** from the *Session 3 exercises* document

Hoisting

- When running your script, the JavaScript parser moves any variable or function declarations to the top of the current scope.
- This process is called *hoisting*

Hoisting

- With variables, only the declaration is hoisted to the top of the scope, not the value assignment.
- With function declarations, the entire declaration is hoisted to the top of the scope.

Hoisting variables

```
// This code:
function foo() {
    bar(); // do something here
    var a = 0;
}

// Will be parsed as:
function foo() {
    var a;
    bar(); // do something here
    a = 0;
}
```

Hoisting functions

```
// This code:
function foo() {
    bar(); // do something here
    var a = 0;
    function myFunction() { /* Statements*/ }
}

// Will be parsed as:
function foo() {
    var a;
    function myFunction() { /* Statements*/ }
    bar(); // do something here
    a = 0;
}
```

Hoisting named function expressions

- For named function expressions, the *name of the variable* is hoisted, but not the *function body* or *function name*.

Hoisting named function expressions

```
// This code:
var test = "hello";
var foo = function myFunction() {
    // statements
}

// Will be parsed as:
var test, foo;
test = "hello";
foo = function myFunction() {
    // statements
}
```

Hoisting unused symbols

- Even if a variable or function declaration is never used in the script, it will still get hoisted within it’s scope.

Hoisting unused symbols

```
// We write this code:
function foo(){
    bar();// do something here
    if(false){
        // This never runs!
        var c ="abc";
        function func(){
            //do something
        }
}
// See next slide for parser interpretation...
```

Hoisting unused symbols

```
// Our code is parsed as:
function foo() {
    function func() {
        //do something
    }
    var c;
    bar(); // do something here
    if(false) {
        // This never runs!
        c = "abc";
    }
}
```

Hoisting - Why?

- How does this effect us?

Hoisting - Why?

- Coding with hoisting in mind will avoid a lot of issues that can arise with variable and function declarations.
- It also helps us to understand how the order in which we write our scripts effects the end result
- To negate the issues that can arise with hoisting, it is recommended to declare variables and functions at the top of each scope - they get moved there anyway!

Exercises: Question 3, 4 & 5

- Now do **Question 3, 4 & 5** from the *Session 3 exercises* document the *Session 3 exercises* document

`this`

- When you create a function, the inner scope automatically receives a **this** keyword
- The value of the keyword will vary, depending upon how the function is called.

`this` in the global scope

- In the global scope, outside of any function definitions, `this` refers to the *global object* (in the browser, that's the *window* object):

```
console.log(this);
```

`this` in functions

- Inside a function definition, `this` will also default to the *global object*.

```
function myFunction() {
    return this;
}
var result = myFunction();
```

`this` in object methods

- Inside an object method, the `this` keyword takes the value of the *object* the method belongs to.

```
var myObject = {
    data: "value",
    action: function () {
        return this;
    }
}
// The object "myObject" is returned by .action()
var result = myObject.action();
```

`this` in object methods

- Object methods can access other properties of the object via the `this` keyword:

```
var myObject = {
    data: "value",
    action: function () {
        // get "data" property of "this" object
        return this.data;
    }
}
myObject.action(); //returns "value"
```

`this` with constructors

- When a function is used as an object constructor, `this` refers to the object returned by the constructor.

```
function construct(number){
    this.a = number;
}
var myObject = new construct(37);
console.log(myObject.a); // 37
```

`this` - call, apply, bind

- Using a function's `call`, `apply` or `bind` methods, you can define the value of `this` that will be used by the function when it is executed.
- This allows us to use our functions in multiple contexts

`function.call()`

- With the `call` method, you pass an object as the first argument, followed by the target function arguments.
  - Target function arguments are comma separated
- The value of `this` when the function runs will be the object passed as the first argument

function.call() example

```
function add(c, d){
    return this.a + this.b + c + d;
}
// The object to use as "this" in function
var myObject = {
    a: 1,
    b: 3
};
// Pass object to "call",
// plus args for "add" (5, 7)
add.call(myObject, 5, 7);
```

`function.apply()`

- With the `apply` method, you pass an object as the first argument, followed by the target function arguments as an array.
- Again, the value of `this` will be the object passed as the first argument
- The only difference between `call` and `apply` is the way we pass the target function's arguments

function.apply() example

```
function add(c, d){
    return this.a + this.b + c + d;
}
var myObject = {
        a:1,
        b:3
    },
    args = [5, 7];

// Pass object to "apply",
// plus args for "add"  (the "args" array)
add.apply(myObject, args);
```

`function.bind()`

- With the `bind` method, you can create a new function where `this` will be a specified object
- Every time the new function is called, `this` will point to the object that was specified when the function was created
- This is different to `call` and `apply` which are *run once* solutions to the same issue

function.bind() example

```
var x = 9;
var module = {
    x: 81,
    getX: function () {
        console.log(this.x);
    }
};
module.getX(); // 81 ("this" is module object)
var newGetX = module.getX; // copy of function
newGetX();// 9 ("this" is window object)

var boundGetX = module.getX.bind(module);
boundGetX(); // result: 81
```

Exercises: Question 6

- Now do **Question 6a and 6b** from the *Session 3 exercises* document

Chained methods with `this`

- Using the `this` keyword, you can build *chainable* object methods!
- If a method does not `return` anything, it is a candidate for chaining
- All we have to do to enable chaining is return the `object` that the method belongs to (i.e. return `this`).

Chained methods example

```
var test = {
    message: '',
    method1: function () {
        this.message = 'Hello';
        return this;
    },
    method2: function () {
        console.log(this.message);
        return this;
    }
}
// method1 is executed first, then method2
test.method1().method2();
```

Exercises: Question 7

- Now do **Question 7** from the *Session 3 exercises* document