More on Functions

IT TALENTS
Training Camp

Functions

Mastering functions is an important skill when you learn JavaScript.

Much of the language's flexibility and expressiveness comes from them.

Where most programming languages have a special syntax for some object-oriented features, JavaScript just uses functions.

Parameters

If you pass more than the function expects, the extra ones will be silently ignored:

```
var result = sum(1, 2, 3, 4, 5, 6, 7, 8);
alert(result); // 3
```

You can create functions that are flexible about the number of parameters they accept. This is possible because of special array **arguments** that is created automatically inside each function.

```
function testArgs() { return arguments; }
var x = testArgs(1, 2, 5, 9, “Hi”); // [1,2,5, 9, “Hi”]
```

Parameters

By using arguments, you can make a function to accept any number of arguments :

```
function sumThemAll() {
	var i, res = 0, number_of_params = arguments.length;
	for (i = 0; i < number_of_params; i++) {
		res += arguments[i];
	}
	return res;
}
```

```
sumThemAll(1, 1, 1); → 3
sumThemAll(1, 2, 3, 4); → 10
sumThemAll(1); → 1
sumThemAll(); → 0
```

Training Camp

Built-in Functions

- parseInt()
- parseFloat()
- IsNaN()
- isFinite()
- encodeURI()
- decodeURI()
- encodeURIComponent()
- decodeURIComponent()
- eval()

Variable Hoisting

What values this code alerts ?

```
var a = 123;
function f() {
	alert(a);
	var a = 1;
	alert(a);
}

f();
```

Variable Hoisting

The first alert will show **undefined**. This is because inside the function the local scope is more important than the global scope.

So, a local variable overwrites any global variable with the same name.

At the time of the first alert(), the variable a was not yet defined (hence the value undefined), but it still existed in the local space due to the special behavior called **hoisting**.

IT TALENTS
Training Camp

Variable Hoisting

When program execution enters a new function, all the variables declared anywhere in the function are moved (or elevated, or hoisted) to the top of the function.

Only the declaration is hoisted, meaning only the presence of the variable is moved to the top.

It's as if the last function was written like this:

```
var a = 123;
function f() {
	var a; // same as: var a = undefined;
	alert(a); // undefined
	a = 1;
	alert(a); // 1
}
```

Functions as Data

Functions in JavaScript are actually **data**.

This means that you can create a function and assign it to a variable:

```
var func = function () {
	return 5;
};
```

Functions as Data

When you use the **typeof** operator on a variable that holds a function value, it returns the string "function":

```
function define() {
return 1;
}

var express = function () {
return 1;
};

typeof define; → “function”
typeof express; → “function”
```

Functions as Data

The way to execute a function is by adding parentheses after its name and this works regardless of how the function was defined :

```
var sum = function (a, b) {
       return a + b;
};
 var add = sum;
 typeof add; →“function”
add(1, 2); →  3
```

Callback Functions

Because a function is just like any other data assigned to a variable, it can be also passed as an argument to other functions.

```
function invokeThenSum(a, b) {
return a() + b();
}

function numberOfLetters(a) {
	return a.length;
}
function random() {
return Math.floor(Math.random()*10);
}

invokeThenSum(numberOfLetters, random);
```

Callback Functions

Another example of passing a function as a parameter is to use **anonymous functions** (function expressions).

```
InvokeThenSum(
      function() { return 5; } ,
      function() { return 6; }
); → 11
```

When you pass a function, A, to another function, B, and then B executes A, then A is called a **callback function**.

If A doesn't have a name, then it's an **anonymous callback function.**

IT TALENTS
Training Camp

Immediate Functions

Another application of an anonymous function is calling a function immediately after it's defined.

```
(
function () {
	var x = Math.floor(Math.random() * 5);
	alert('Random number : ' + x);
}
) ();
```

Immediate Functions

One application of immediate (self-invoking) functions is when you want to have some work done without creating extra global variables.

A drawback is that you cannot execute the same function twice.

This makes immediate functions best for initialization tasks.

IT TALENTS Training Camp

Immediate Functions

It's not uncommon to see code that looks like the following:

```
var result = ( function () {
// something complex with
// temporary local variables...
// ...
// return something;
}) () ;
```

Surrounding parentheses are optional but don't skip them.

Inner(Private) Functions

You can define a function inside another one :

```
function calc(param) {
	function double(theinput) {
		return theinput * 2;
	}
	return 'The result is ' + double(param);
}
```

Inner function – not accessible from outside.

IT TALENTS Training Camp

Benefits of Private Functions

- You keep the global name space clean (less likely to cause naming collisions)

- Privacy—you expose only the functions you decide to the "outside world", keeping to yourself functionality that is not meant to be consumed by the rest of the application.

IT TALENTS Training Camp

Functions That Return Functions

A function can return only one value, and this value can be another function:

```
function a() {
	alert('A!');
	return function () {
		alert('B!');
	};
}
```

Functions That Return Functions

Because a function can return a function, you can use the new function to replace the old one.

```
a = a();
```

The function can actually rewrite itself from the inside :

```
function a() {
	alert('A!');
	a = function () {
		alert('B!');
	};
}
```

Functions That Return Functions

What will this code alert when:

- It is initially loaded?
- You call a() afterwards ?

```
var a = (function () {
        function someSetup() {
                alert('setup done');
        }
        function actualWork() {
                alert('working...');
        }
        someSetup();
        return actualWork;
}());
```

Closures

```
var globalVar = "global variable";
var F = function () {
	var localVar = "local variable";
	var innerFunc = function () {
		var innerVar = "inner local";
		return localVar;
	};
	return innerFunc;
};
```

Returning a function that has access to local variable.

IT TALENTS
Training Camp

Closures

```
var inner; // placeholder for a closure
var F = function () {
	var localVar = "local variable";
	var innerFunc = function () {
		return localVar;
	};
	inner = innerFunc;
};
```

Placing a function from inside into the placeholder.

Closures

```
function F() {
	var arr = [], i;
	for (i = 0; i < 3; i++) {
		arr[i] = function () {
		return i;
		};
	}
	return arr;
}

 var arr = F();
```

What will be the result of the following ?

Closures - Applications

Implementing getters and setters to access hidden variable.

```
var getValue, setValue; //placeholders

(function () {
        var secretValue = 0;

        getValue = function () {
                return secretValue;
        };

        setValue = function (v) {
                if (typeof v === "number") {
                        secretValue = v;
                }
        };
}());
```

Access control to a hidden value.

Closures - Applications

Wrapping the iterating logic over some data structure in a function. That is called **iterator.**

Iterator function for hidden collection of data

```
function setup(x) {
        var i = 0;
        return function () {
                return x[i++];
        };
}
```