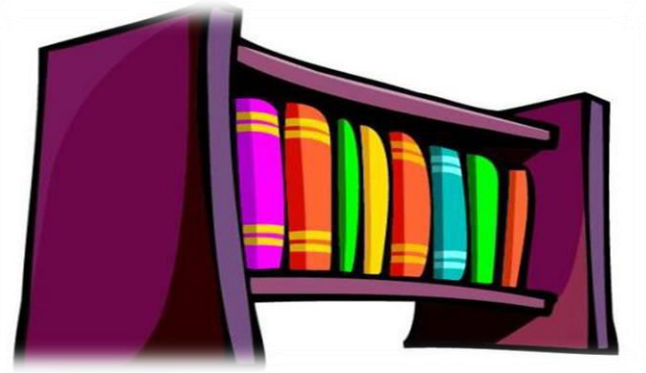
Telerik®
Develop experiences

Functions and Function Expressions

Closures, Function Scope, Nested Functions

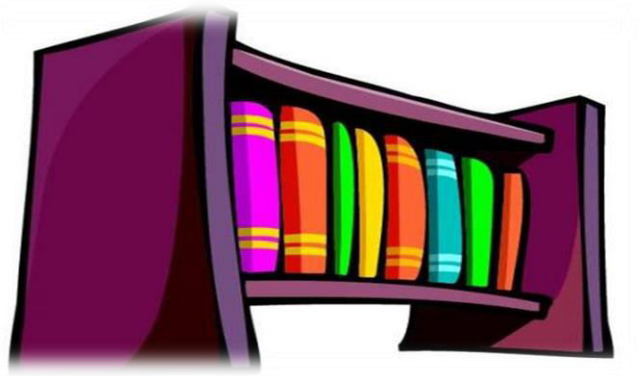
Telerik Software Academy
http://academy.telerik.com

Table of Contents



Telerik

Table of Contents



Telerik

Functions in JavaScript

- Functions are small named snippets of code
  - Can be invoked using their identifier (name)
- Functions can take parameters
  - Parameters can be of any type
- Each function gets two special objects
  - `arguments` contains all passed arguments
  - `this` contains information about the context
    - Different depending of the way the function is used
- Function can return a result of any type
  - `undefined` is returned if no return statement

Telerik

Functions usage

```
function max (arr) {
  var maxValue = arr[0];
  for     (var i = 1; i < arr.length; i++) {
    maxValue = Math.max(maxValue, arr[i]);
  }
  return maxValue;
}

function printMsg(msg){
  console.log(msg);
}
```

Telerik

Defining Functions

Creating Functions

- Many ways to create functions:
  - Function declaration:

```
function printMsg (msg) {console.log(msg);}
```

  - Function expression

```
var printMsg = function () {console.log(msg);}
```

  - With function constructor

```
var printMsg = new Function("msg",'console.log("msg");');
```

- Since functions are quite special in JavaScript, they are loaded as soon as possible

Telerik

Function Declaration

- Function declarations use the function operator to create a function object
- Function declarations are loaded first in their scope
  - No matter where they appear
  - This allows using a function before it is defined

```
printMsg("Hello");
function printMsg(msg){
    console.log("Message: " + msg);
}
```

Telerik

Function Declarations

Live Demo

Function Expression

- Function expressions are created using the function literal
  - They are loaded where they are defined
    - And cannot be used beforehand
  - Can be invoked immediately
- The name of function expressions is optional
  - If the name is missing the function is anonymous

```
var printMsg = function (msg){
    console.log("Message: " + msg);
}
printMsg("Hello");
```

Telerik

Function Expression (2)

- Function expressions do no need an identifier
  - It is optional
  - Still it is better to define it for easier debugging
    - Otherwise the debuggers show anonymous
- Types of function expressions

```
var printMsg = function (msg){
    console.log("Message: " + msg);
}
var printMsg = function printMsg(msg) {
 console.log("Message: " + msg);
}
(function(){…});
```

Telerik

Function Expressions

Live Demo

$$q(n) = \begin{vmatrix} 1 & & & & & & & & 1 \\ -1 & 1 & & & & & & & 0 \\ -1 & -1 & 1 & & & & & & -1 \\ 0 & -1 & -1 & 1 & & & & & 0 \\ 0 & 0 & -1 & -1 & 1 & & & & -1 \\ 1 & 0 & 0 & -1 & -1 & 1 & & & 0 \\ 0 & 1 & 0 & 0 & -1 & -1 & 1 & & 0 \\ 1 & 0 & 1 & 0 & 0 & -1 & -1 & & 0 \\ \vdots & & & & & & & \ddots & \vdots \end{vmatrix}_{(n+1)\times(n+1)},$$

Function Methods

Function Methods

- Functions have methods as well
  - `function.toString()`
    - Returns the code of the functions as a string
  - `function.call(obj, args)`
    - Calls the function over the obj with args
  - `function.apply(obj, arrayOfArgs)`
    - Applies the function over obj using the arrayOfArgs as arguments
- Basically call and apply to the same
  - One gets args, the other gets array of args

Telerik

Call and Apply

- function.apply(obj, arrayOfargs) applies the function over an specified object, with specified array of arguments
  - Each function has a special object this
- function.call(obj,arg1,arg2…) calls the function over an specified object, with specified arguments
  - The arguments are separated with commas
- apply() and call() do the same with difference in the way they receive arguments

Telerik

Call and Apply

- function.apply(obj, arrayOfargs) applies the function over an specified object, with specified array of arguments
  - Each function has a special object this
  - By invoking apply/call, obj is assigned to this

```
var numbers = […];
var max = Math.max.apply (null, numbers);
function printMsg(msg){
  console.log("Message: " + msg);
}
printMsg.apply(null, ["Important message"]);
//here this is null, since it is not used anywhere in //the function
//more about this in OOP
```

Function Methods

Live Demo

Scope

Scope

- Scope is a place where variables are defined and can be accessed
- JavaScript has only two types of scopes
  - Global scope and function scope
    - Global scope is the same for the whole web page
    - Function scope is different for every function
  - Everything outside of a function scope is inside of the global scope

```
if(true){
    var sum = 1+2;
}
console.log(sum);
```

Telerik

Global Scope

- The global scope is the scope of the web page
- Objects belong to the global scope if:
  - They are define outside of a function scope
  - They are defined without var
    - Fixable with 'use strict'

```
function arrJoin(arr, separator) {
  separator = separator || "";
  arr = arr || [];
  arrString = "";
  for (var i = 0; i < arr.length; i += 1) {
    arrString += arr[i];
    if (i < arr.length - 1) arrString += separator;
  }
  return arrString;
}
```

Telerik

Global Scope (2)

- The global scope is one of the very worst parts of JavaScript
  - Every object pollutes the global scope, making itself more visible
  - If two objects with the same identifier appear, the first one will be overridden

Telerik

Global Scope

Live Demo

Function Scope

- JavaScript does not have a block scope like other programming languages (C#, Java, C++)
  - { and } does not create a scope!
- Yet, JavaScript has a function scope
  - Function expressions create scope
  - Function declarations create scope

```
if(true)var result = 5;
console.log(result);//logs 5
```

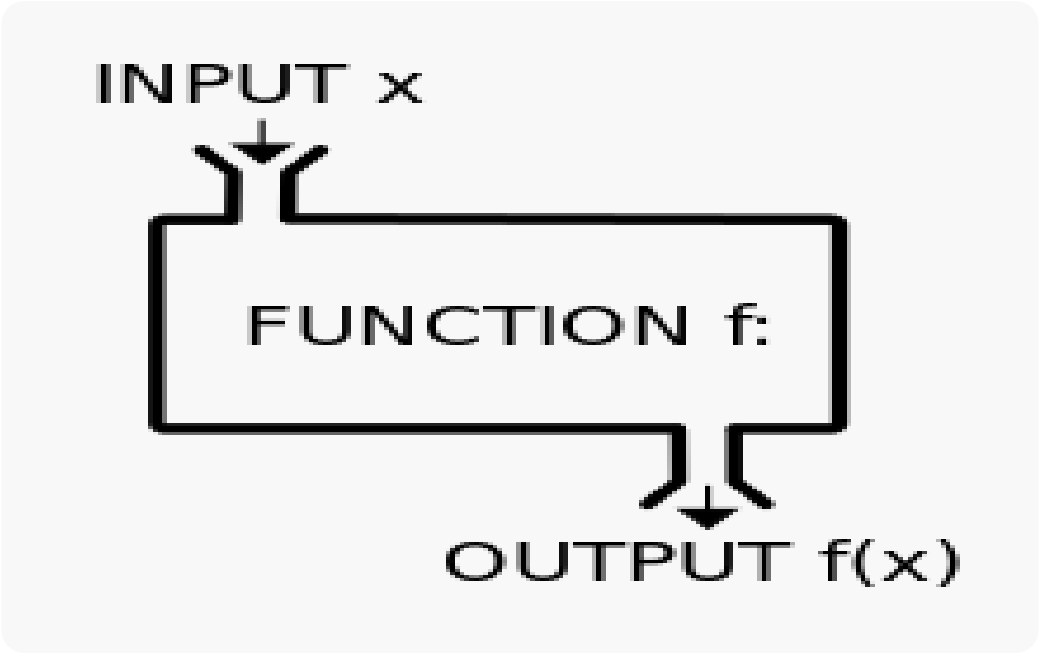
```
if(true) (function(){ var result = 5;})();
console.log(result);//ReferenceError
```

```
function logResult(){ var result = 5; }
if(true) logResult();
console.log(result); //ReferenceError
```

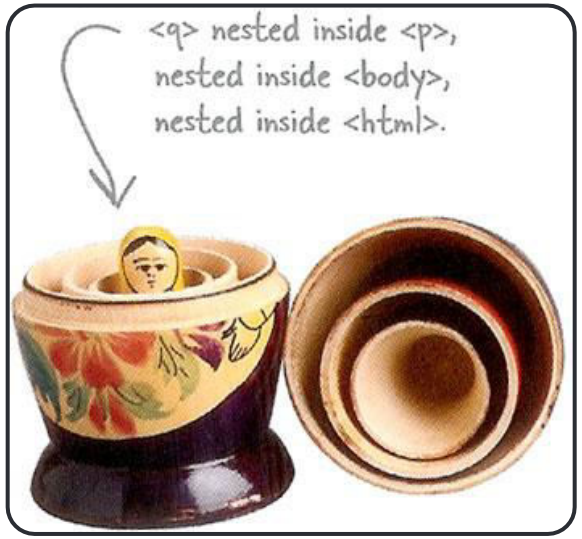
Telerik

Function Scope

Live Demo



Nested Functions



Nested Functions

- Functions in JavaScript can be nested
  - No limitation of the level of nesting

```
function compare(str1, str2, caseSensitive) {
  if(caseSensitive)
    return compareCaseSensitive(str1,str2)
  else
    return compareCaseInsesitive(str1,str2);

    function compareCaseSensitive(str1, str2) { … }

    function compareCaseInsesitive(str1, str2) { … }
}
```

Telerik

Nested Functions (2)

- Which function has access to which objects and arguments?
  - It's all about scope!
  - Objects can access the scope they are in
    - And objects in the scope they are in can access the scope where they are in, and so on…
    - Also called closure
  - The innermost scope can access everything above it

```
function compare(str1, str2, caseSensitive) {
  if(caseSensitive) return compareCaseSensitive(str1,str2)
  else return compareCaseInsesitive(str1,str2);
    function compareCaseSensitive(str1, str2) { … }
    function compareCaseInsesitive(str1, str2) { … }
}
```

Telerik

Nested Functions: Example

- Objects can access the scope they are in
  - outer() can access the global scope
  - inner1() can access the scope of outer() and the global scope

```
var str = “string”                  //global
function outer(o1, o2) {            //outer
  function inner1(i1, i2, i3) {  //inner1
    function innerMost(im1) {    //innerMost
      …
    }                           //end of innerMost
  }                             //end of inner1
  function inner2(i1, i2, i3) {  //inner2
    …
  }                             //end of inner2
}                               //end of outer
                                //global
```

Telerik

Nested Functions (3)

- What about objects with the same name?
  - If in the same scope – the bottommost object
  - If not in the same scope – the object in the innermost scope

```
function compare(str1, str2, caseSensitive) {
  if(caseSensitive) return compareCaseSensitive(str1,str2)
  else return compareCaseInsesitive(str1,str2);
    function compareCaseSensitive(str1, str2) {
      //here matter str1 and str2 in compareCaseSensitive
    }
    function compareCaseInsesitive(str1, str2) {
      //here matter str1 and str2 in compareCaseInsensitive
    }
}
```

Telerik

Nested Functions

Live Demo

Immediately Invoked Function Expressions

Immediately Invoked Function Expressions

- In JavaScript, functions expressions can be invoked immediately after they are defined
  - Can be anonymous
  - Create a function scope
  - Don't pollute the global scope
  - Handle objects with the same identifier
- IIFE must be always an expression
  - Otherwise the browsers don't know what to do with the declaration

Telerik

Valid IIFE

- Valid IIFEs
  - In all cases the browser must be explicitly told that the thing before () is an expression
- IIFEs are primary used to create function scope
  - And prevent naming collisions

```
var iife = function(){ console.log("invoked!"); }();
(function(){ console.log("invoked!"); }());
(function(){ console.log("invoked!"); })();
!function(){ console.log("invoked!"); }();
true && function(){console.log("invoked!"); }();
1 + function(){console.log("invoked!"); }();
```

Telerik

Immediately Invoked Function Expressions

Live Demo

Closures

Closures

- Closures are a special kind of structure
  - They combine a function and the context of this function

```
function outer(x){
  function inner(y){
    return x + " " + y;
  }
  return inner;
}
```

Telerik

Closures Usage

- Closures can be used for data hiding
  - Make objects invisible to their user
    - Make them private

```
var school = (function() {
    var students = [ ];
    var teachers = [ ];
    function addStudent(name, grade) {...}
    function addTeacher(name, speciality) {...}
    function getTeachers(speciality) {...}
    function getStudents(grade) {...}
    return {
        addStudent: addStudent,
        addTeacher: addTeacher,
        getTeachers: getTeachers,
        getStudents: getStudents
    };
})();
```

Closures

Live Demo

Functions and Expressions in JavaScript

http://academy.telerik.com