Telerik Academy

Functions and Function Expressions

Closures, Function Scope, Nested Functions

Telerik Software Academy

Learning & Development Team

http://academy.telerik.com

Telerik Academy

Table of Contents

- Functions in JavaScript
- Function object
- Defining Functions
  - Function declarations
  - Function expressions
  - Function constructor
  - Expression vs. declaration
- Function properties
- Function methods

Telerik Academy

Table of Contents (2)

- Recursion
  - Factorial example
  - Traversing the DOM
  - Recursion with expressions
- Scope
  - Global and function
- Nested functions
- Immediately-invoked function expressions
- Closures

Telerik Academy

Functions in JavaScript

Telerik Academy

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 Academy

Functions in JavaScript (2)

- Different function usages:

```
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 Academy

Functions in JavaScript

Live Demo

Telerik Academy

Function Object

Function Object

- Functions are one of the most powerful features in JavaScript
  - And one of the most important
- In JavaScript functions are first-class objects
  - They can be assigned to variables or properties, passed as arguments and returned by other functions
  - They have properties of their own
    - `length`, `caller`, `name`, `apply`, `call`

```
function max(arr){ … }
console.log(max.length); //returns 1
console.log(max.name); //returns "max"
console.log((function(){}).name));
```

Function Object

- Functions are objects and they can be used as objects
  - Can be passed as arguments to functions
  - Can be stored in an array
  - Can be assigned to variable
  - Can be returned by another function

```
var arr = [3, 2, 1, 3, 4, 5, 1, 2, 3, 4, 5, 7, 9];
function orderBy(x, y) { return x - y; }

arr.sort(orderBy);
//better to be done using anonymous function
//arr.sort(function(x, y){return x - y;});
```

Telerik Academy

Function Object

Live Demo

Telerik Academy

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 Academy

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 Academy

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 Academy

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 Academy

Function Expressions

Live Demo

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

Telerik Academy

Function Constructor

- Function constructor is similar to expressions
  - A constructor initializes a function object
  - It is loaded when the parser reaches it
- The function constructor form is:

```
new Function([optional arguments],body);
```

  - Example

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

- Not a good practice
  - Bad performance and messy code

Telerik Academy

Function Constructor

Live Demo

Telerik Academy

Function Expression vs. Function Declaration

- Function declarations are loaded first, while function expressions are loaded when reached
  - i.e. function declarations can be used before they are declared, while expressions cannot
- Function declarations can be overridden
  - Imagine two function declarations in the same scope have the same name
  - Which will be the one to execute?

Telerik Academy

Function Expression vs. Function Declaration (2)

- Function declarations can be overridden
  - Results with unexpected behavior

```
if(true){
  function printMsg(msg) {
    console.log("Message (from if): " + msg);
  }
}
else{
  function printMsg(msg) {
    console.log("Message (from else): " + msg);
  }
}
printMsg("message");
```

Telerik Academy

Function Expression vs. Function Declaration (2)

- Function declarations can be overridden
  - Results with unexpected behavior

```
if(true){
  function printMsg(msg) {
    console.log("Message (from if): " + msg);
  }
}
else{
  function printMsg(msg) {
    console.log("Message (from else): " + msg);
  }
}
printMsg("message");
```

logs (from else) in Opera, IE and Chrome

Telerik Academy

Function Expression vs. Function Declaration (2)

- Function declarations can be overridden
  - Results with unexpected behavior

```
if(true){
  function printMsg(msg) {
    console.log("Message (from if): " + msg);
  }
}
else{
  function printMsg(msg) {
    console.log("Message (from else): " + msg);
  }
}
printMsg("message");
```

logs (from if) only in Firefox

logs (from else) in Opera, IE and Chrome

Telerik Academy

Function Expression vs. Function Declaration (3)

- Easy to fix with function expression:
  - Works well on all browsers

```
if(true){
  var printMsg = function (msg) {
    console.log("--from if");
    console.log(msg);
  }
}
else{
  var printMsg = function (msg) {
    console.log("--from else");
    console.log(msg);
  }
}
printMsg("message");
```

Telerik Academy

Function Expression vs. Function Declaration (3)

- Easy to fix with function expression:
  - Works well on all browsers

```
if(true){
  var printMsg = function (msg) {
    console.log("--from if");
    console.log(msg);
  }
}
else{
  var printMsg = function (msg) {
    console.log("--from else");
    console.log(msg);
  }
}
printMsg("message");
```

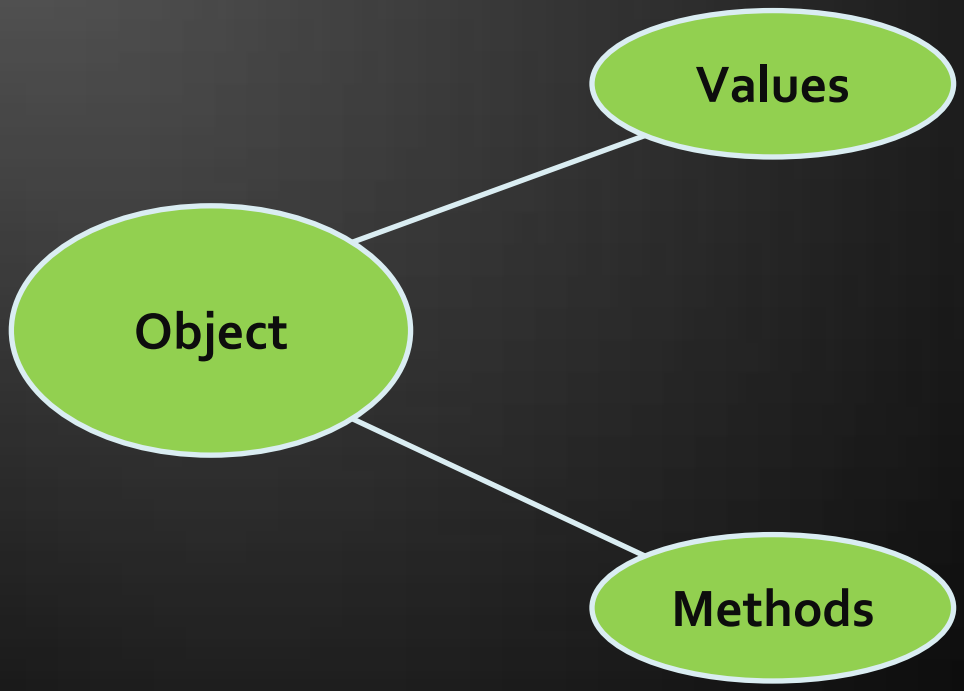
Logs ("from if") everywhere

Telerik Academy

Function Declaration vs. Function Expression

Live Demo

Function Properties



Function Properties

- Each function is an object
  - Created either with declaration, expression or constructor
- Functions have properties:
  - `function.length`
    - The count of parameters the function expects
    - The arguments object is not counted
  - `function.name`
    - Identifier of the function
    - Returns an empty string if anonymous

Telerik Academy

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

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

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
```

Telerik Academy

Function Methods

Live Demo

Telerik Academy

Recursion

Calling functions from themselves

Recursion

- Functions can refer to themselves as call to themselves
  - This is called recursion
- Example:

```
function factorial(n){
  if(n==0){
    return 1;
  }
  return factorial(n-1) * n;
}
```

Recursion

- Functions can refer to themselves as call to themselves
  - This is called recursion
- Example:

```
function factorial(n){
  if(n === 0){
    return 1;
  }
  return factorial(n-1) * n;
}
```

Bottom of the recursion.
A recursion must
always have a bottom!

Recursion (2)

- Recursion works quite well when:
  - Traversing data structures
    - Trees, matrices, graphs, DOM nodes
  - Generating combinations
  - Generating sequences
    - Fibonacci, factorial
- Every recursion can be replaced by enough loops, and form the so called iterative solution
  - Yet, in some cases using recursion is much simpler than using loops

Telerik Academy

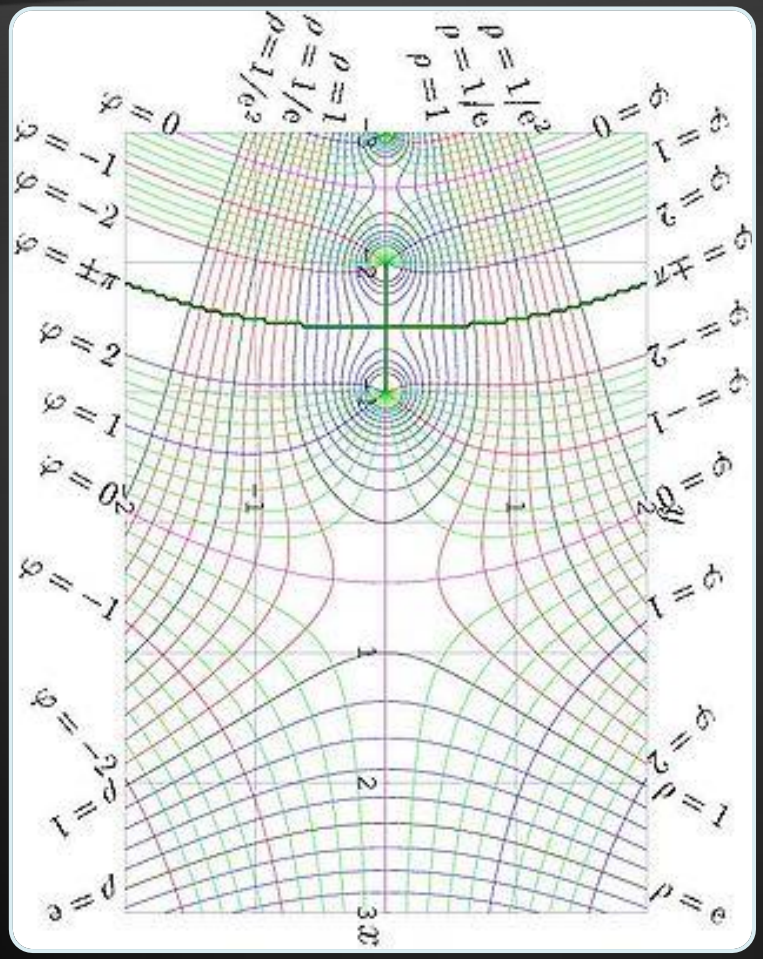
Recursion: Factorial

- Using recursion to calculate factorial numbers
  - Using the formula `F(N) = F(N-1) * N`

```
function factorial(n){
  if(n === 0){
    return 1;
  }
  return factoriel(n-1) * n;
}

console.log(factorial(5));  //120
console.log(factorial(12)); //479001600
```

Telerik Academy



Factorial

Live Demo

Traversing the DOM

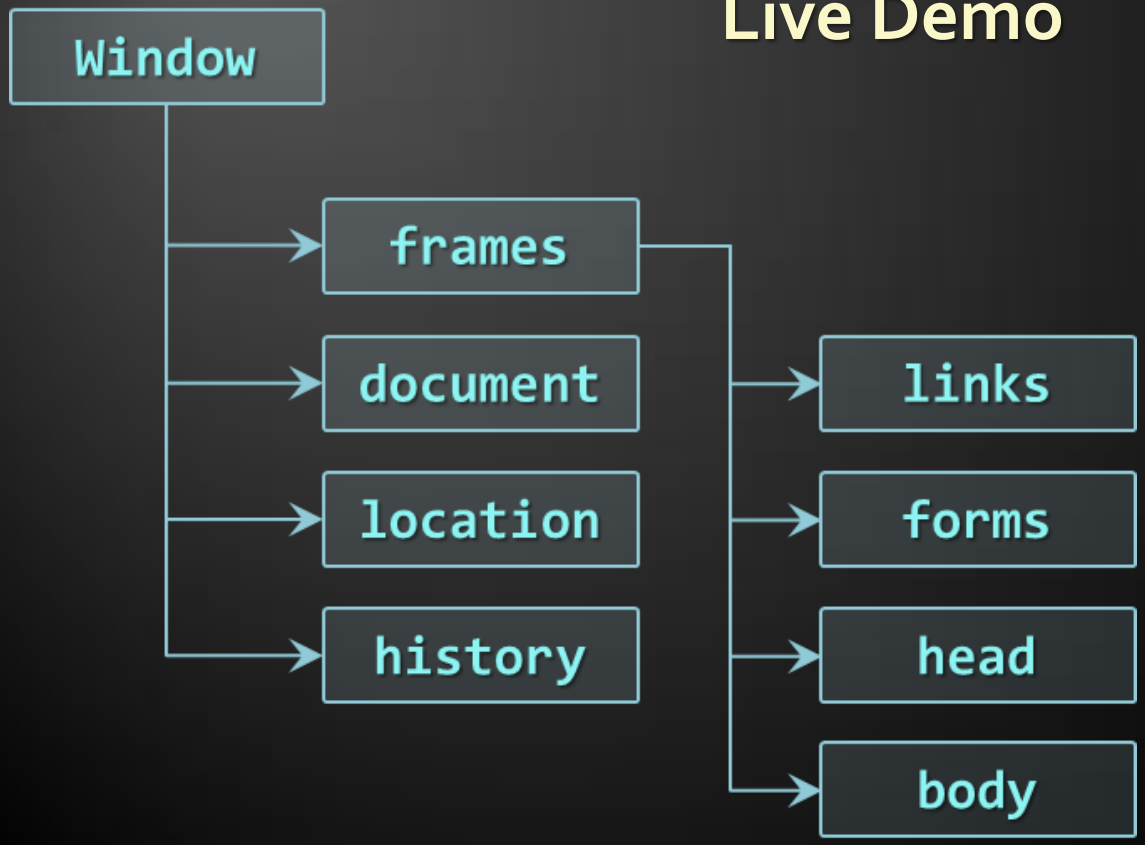
- Passing a root element
  - Each element print its tag name and invokes the same function for each of its children

```
function traverse(element) {
  function traverseElement(element, spacing) {
    spacing = spacing || "  ";
    console.log(spacing + element.nodeName);
    var len = element.childNodes.length;
    for (var i = 0; i < len; i += 1) {
      var child = element.childNodes[i];
      if (child.nodeType === 1) {
        traverseElement(child, spacing + "--");
      }
    }
    console.log(spacing + "/" + element.nodeName);
  }
  traverseElement(element, "");
}
```

Telerik Academy

DOM Traversal

Live Demo



Telerik Academy

Recursion with Function Expression

- Recursion is simple enough with function declarations
  - But not so easy with function expressions

```
var fact = function (n) {
  if (n === 0) {
    return 1;
  }
  return n * fact (n - 1);
};
```

Telerik Academy

Recursion with Function Expression

- Recursion is simple enough with function declarations
  - But not so easy with function expressions

```
var fact = function (n) {
  if (n === 0) {
    return 1;
  }
  return n * fact (n - 1);
};
```

```
fact(5);
```

  - Logs 120

Telerik Academy

Recursion with Function Expression

- Recursion is simple enough with function declarations
  - But not so easy with function expressions

```
var fact = function (n) {
  if (n === 0) {
    return 1;
  }
  return n * fact (n - 1);
};
```

```
fact(5);
```

```
var f = fact;
f(5);
```

  - Assign the function to a variable
  - Still logs 120

Telerik Academy

Recursion with Function Expression

- Recursion is simple enough with function declarations
  - But not so easy with function expressions

```
var fact = function (n) {
  if (n === 0) {
    return 1;
  }
  return n * fact (n - 1);
};
```

```
fact(5);
```

```
var f = fact;
f(5);
```

```
fact = 5;
f(5);
```

  - Assign a number value to the original function
  - Throws TypeError (Number is not a function)

Telerik Academy

Buggy Recursion with Function Expressions

Live Demo

Telerik Academy

Recursion with Function Expression (2)

- The previous example can be solved by giving an identifier to the function expression
  - Only the function itself can use this identifier

```
var factorial = function factorial(n) {
  if (n == 1) {
    return 1;
  }
  return n * factorial (n - 1);
  //or use arguments.callee
};
var factorial2 = factorial;
factorial = 5;
console.log(factorial2(5)); //logs 120 - correct
```

Telerik Academy

Working Recursion With Function Expressions

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);
```

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);
```

The scope of the if is the global scope.
sum is accessible from everywhere

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;
}
```

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;
}
```

arr, separator and i belong to the scope of printArr

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;
}
```

arr, separator and i belong to the scope of printArr

arrString and arrJoin belong to the global scope

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 Academy

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
```

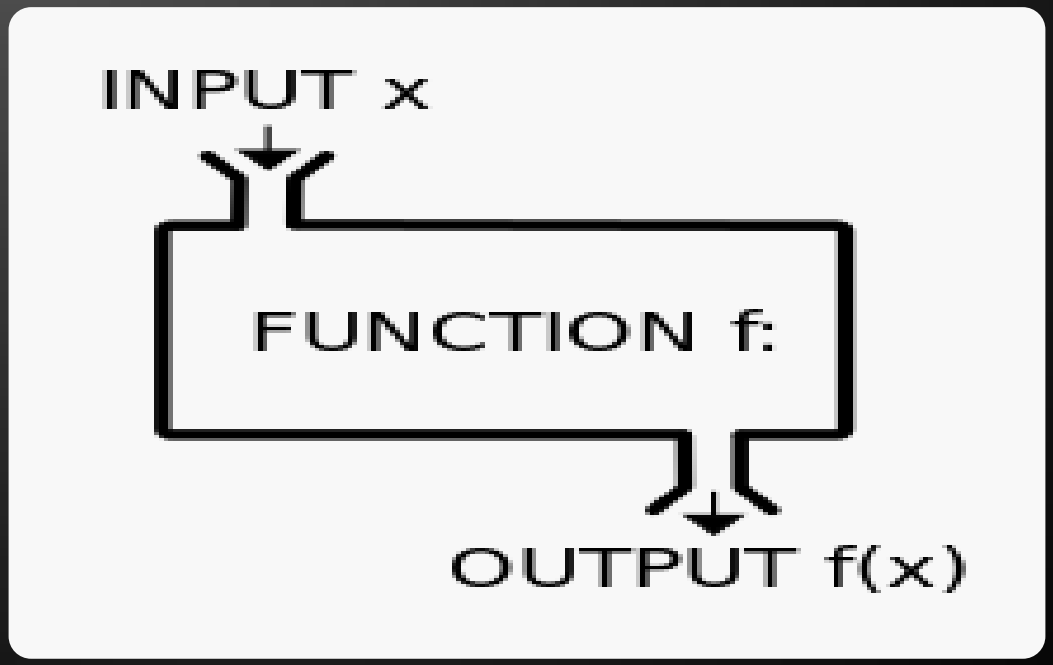
```
if(true) (function(){ var result = 5;})();
console.log(result);//ReferenceError
```

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

Telerik Academy

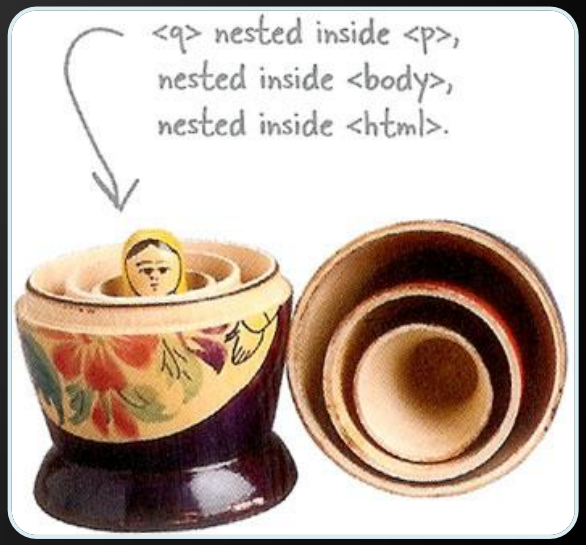
Function Scope

Live Demo



Telerik Academy

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) { … }
}
```

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 Academy

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 through it the global scope and etc…

```
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
```

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 Academy

Nested Functions

Live Demo

Telerik Academy

Immediately Invoked Function Expressions

Functions invoked immediately after they are created

Telerik Academy

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

Valid IIFE

- Valid IIFEs

```
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!"); }();
```

  - 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

Telerik Academy

Immediately Invoked Function Expressions

Live Demo

Telerik Academy

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;
}
```

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;
}
```

inner() forms a closure.
It holds a reference to x

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;
}
```

inner() forms a closure.
It holds a reference to x

In the context of f1,
x has value 5

```
var f1 = outer(5);
console.log(f1(7)); //outputs 5 7
```

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;
}
```

inner() forms a closure.
It holds a reference to x

In the context of f1,
x has value 5

```
var f1 = outer(5);
console.log(f1(7)); //outputs 5 7
```

In the context of f2,
x has value "Peter"

```
var f2 = outer("Peter");
console.log(f2("Petrov"));  //outputs Peter Petrov
```

Telerik Academy

Simple Closures

Live Demo

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 Usage

- Closures can be used for data hiding
  - Make objects invisible to the outside
    - 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
    };
})();
```

This is actually called a Module

Telerik Academy

Closures

Live Demo

Telerik Academy

Advanced Function

Questions?

http://academy.telerik.com

Homework

1. Create a module for working with DOM. The module should have the following functionality
   - Add DOM element to parent element given by selector
   - Remove element from the DOM by given selector
   - Attach event to given selector by given event type and event hander
   - Add elements to buffer, which appends them to the DOM when their count for some selector becomes 100
     - The buffer contains elements for each selector it gets
   - Get elements by CSS selector
   - The module should reveal only the methods

Homework (2)

1. (cont.) Create a module for working with DOM. The module should have the following functionality

```
var domModule = …
var div = document.createElement("div");

//appends div to #wrapper
domModule.appendChild(div,"#wrapper");

//removes li:first-child from ul
domModule.removeChild("ul","li:first-child");

//add handler to each a element with class button
domModule.addHandler("a.button", 'click',
                     function(){alert("Clicked")});

domModule.appendToBuffer("container", div.cloneNode(true));
domModule.appendToBuffer("#main-nav ul", navItem);
```

Homework (3)

2. Create a module that works with moving div nodes. Implement functionality for:
   - Add new moving div element to the DOM
     - The module should generate random background, font and border colors for the div element
     - All the div elements are with the same width and height
   - The movements of the div nodes can be either circular or rectangular
   - The elements should be moving all the time

```
var movingShapes = …
//add element with rectangular movement
movingShapes.add("rect");
//add element with ellipse movement
movingShapes.add("ellipse");
```

Homework (4)

3. Create a module to work with the console object. Implement functionality for:
   - Writing a line to the console
   - Writing a line to the console using a format
   - Writing to the console should call `toString()` to each element
   - Writing errors and warnings to the console with and without format

```
var specialConsole = …
specialConsole.writeLine("Message: hello");
//logs to the console "Message: hello"
specialConsole.writeLine("Message: {0}", "hello");
//logs to the console "Message: hello"
specialConsole.writeError("Error: {0}", "Something happened");
specialConsole.writeWarning("Warning: {0}", "A warning");
```