04. JS Functions

Justina Balsė

04. JS Functions

- Functions
- Arrow functions
- Callback functions
- Global vs. Local scope
- let vs. var
- Hoisting
- Closure
- IIFE
- Variable Shadowing
- Template Strings

Functions

- There are three important pieces to creating a function:
  - Arguments
  - Function code
  - Return value

Functions

```
let greetUser = function () {
    console.log("Welcome user!");
}
greetUser();
```

Functions

```
let square = function (num) {
    let result = num * num;
return result;
}

let value = square(3);
let otherValue = square(10);

console.log(value);
console.log(otherValue);
```

Functions | Multiple arguments

```
let add = function (a, b, c) {
    return a + b + c;
}
let result = add(10, 1, 5);
console.log(result);
```

Functions | Default arguments

```
let getScoreText = function (name = "Anonymous", score = 0) {
return "Name: " + name + " - Score: " + score;
}

let scoreText1 = getScoreText();
console.log(scoreText1);

let scoreText2 = getScoreText("Tom", 33);
console.log(scoreText2);

let scoreText3 = getScoreText(undefined, 99);
console.log(scoreText3);
```

Functions | Arguments object

- The **arguments object** is a local variable available within all functions.
- This object contains an entry **for each argument passed to the function**, the first entry's index starting at 0.

```
function add(){
    let sum = 0;
    for(let i = 0; i<arguments.length; i++){
        sum = sum + arguments[i];
    }
    return sum;
}

// "0LT-123456"
console.log(add("LT", "-", 123456));
```

Arrow function expresion

- An **arrow function expression** has a shorter syntax than a function expression.

Arrow function expresion

```
// ES5
var selected = allJobs.filter(function (job) {
    return job.isSelected();
});

// ES6
var selected = allJobs.filter(job => job.isSelected());
```

Arrow function expresion

- When you just need a simple function with one argument, the new arrow function syntax is simply *Identifier* => *Expression*. You get to skip typing function and return, as well as some parentheses, braces, and a semicolon.

Arrow function expresion

- Specifying parameters:
  - () => { ... } // no parameter
  - x => { ... } // one parameter
  - (x, y) => { ... } // several parameters
- Specifying a body:
  - x => { return x * x } // block
  - x => x * x // expression, equivalent to previous line

Arrow function expresion

```
let arr = [1, 2, 3];

let squares1 = arr.map(function (x) {
    return x * x;
});

let squares2 = arr.map(x => x * x);

// [1, 4, 9]
```

Callback functions

- **Callback functions** are derived from a programming paradigm known as **functional programming**.
- Functional programming specifies the use of **functions as arguments**.
- A callback function is essentially a pattern.

Callback functions

```
let friends = ["Mike", "Stacy", "Andy", "Rick"];

friends.forEach(function (eachName, index){
   // 1. Mike, 2. Stacy, 3. Andy, 4. Rick
   console.log(index + 1 + ". " + eachName);
});
```

Undefined for variable

```
let name;
name = "Jen";

if (name === undefined) {
    console.log("Please provide a name");
} else {
    console.log(name);
}
```

Undefined for function arguments

```
let square = function (num) {
    console.log(num);
}

square();
```

Undefined as function return default value

```
let square = function (num) {
    console.log(num);
}
let result = square();
console.log(result);
```

Global vs. Local scope

- **Global scope**. This is a scope that's visible to all other scopes. It contains variables defined outside of any code block.
- **Local scope**. This is scope created by code block. A Local scope can access values defined in itself or **in any parent/ancestor scope**. **It's unable to access variables in a child scope.**

Global vs. Local scope

```
let var1 = "var1";

if(true){
        let var2 = "var2";
        if (true){
                let var3 = "var3";
        }
}

if(true){
        let var4 = "var4";
}
```

let vs. var

- The scope of a variable defined with **var** is **function scope** or declared outside any function, global.
- The scope of a variable defined with **let** is block scope.

let vs. var

```
for (let i = 0; i < 10; i++) {
   //i is visible
   console.log(i);
}
//throws an error as "i is not defined" because i is not visible
console.log(i);
```

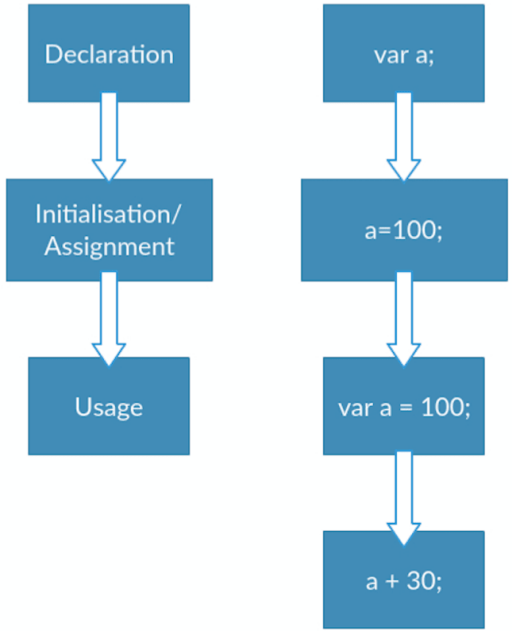
```
for (var i = 0; i < 10; i++) {
    //i is visible
    console.log(i);
}
//i is visible here too
console.log(i);
```

let vs. var

```
function aSampleFunction() {
    let letVariable = "Hey! What's up? I am let variable.";
    var varVariable = "Hey! How are you? I am var variable.";

    //Hey! What's up? I am let variable.
    console.log(letVariable);
    //Hey! How are you? I am var variable.
    console.log(varVariable);
}
// letVariable is not defined
console.log(letVariable);
// varVariable is not defined
console.log(varVariable);
```

Hoisting variables



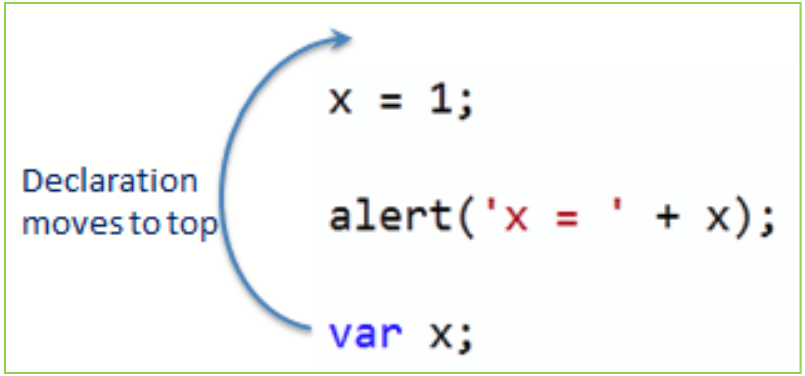
- Hoisting is a concept in JavaScript, not a feature.
- In other scripting or server side languages, variables or functions must **be declared before using it**.

Hoisting variables

```
x = 1;

// display x = 1
alert('x = ' + x);

var x;
```



Hoisting is only possible **with declaration** but not the initialization.

Function Hoisting

```
// 10
alert(Sum(5, 5));

function Sum(val1, val2){
    return val1 + val2;
}
```

- JavaScript compiler moves the **function definition** at the top in the same way as variable declaration.

Hoisting on function expression

```
// error
Add(5, 5);

var Add = function Sum(val1, val2){
   return val1 + val2;
}
```

- JavaScript compiler **does not move function expression.**

Points to Remember

- JavaScript compiler moves variables and function declaration to the top and this is called hoisting.
- Only variable declarations move to the top, not the initialization.
- Functions definition moves first before variables.

Closure

- A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain.
- The closure has three scope chains:
  - it has access to its own scope (variables defined between its curly brackets);
  - it has access to the outer function's variables;
  - it has access to the global variables.

Closure

```
function OuterFunction() {

    var outerVariable = 1;

    function InnerFunction() {
        alert(outerVariable);
    }

    InnerFunction();
}
```

Closure

```
function OuterFunction() {

    var outerVariable = 100;

    function InnerFunction() {
        alert(outerVariable);
     }

    return InnerFunction;
}
var innerFunc = OuterFunction();

// 100
innerFunc();
```

- return InnerFunction; returns InnerFunction from OuterFunction when you call OuterFunction();
- A variable innerFunc reference the InnerFunction() only, not the OuterFunction().
- when you call innerFunc(), it can still access outerVariable which is declared in OuterFunction().
- This is called Closure. ☺

IIFE

```
(function () {
    console.log("IIFE!");
})();
```

- **Immediately Invoked Function Expression** (IIFE) is one of the most popular design patterns in JavaScript.

Variable Shadowing

- If there's a variable in the global scope, and you'd like to create a variable with the same name in a function, that's not a problem in JavaScript.
- The variable in the inner scope will temporarily **shadow** the variable **in the outer scope.**

Variable Shadowing

```
let currencySymbol = "$";

function showMoney(amount) {
     let currencySymbol = "€";
     console.log(currencySymbol + amount);
}
showMoney("100");
```

Variable Shadowing

```
let name = "Tom"

if (true){
    let name = "Mark";
    if(true){
        name = "Bob";
    }
}
console.log(name);

if (true){
    name = "John";
}
console.log(name);
```

1. Bob, Tom
2. Bob, John
3. Tom, John
4. Mark, John

Variable Shadowing

```
//let name = "Tom"

if (true){
    //let name = "Mark";
    if(true){
        name = "Bob";
    }
}

if (true){
    console.log(name);
}

console.log(name);
```

1. Undefined, Undefined
2. Bob, Bob
3. Null, Null
4. Error

Template Strings

```
let petName = "Kitty";
let petAge = 3;

let bio = petName + " is " + petAge + " years old.";
console.log(bio);

let altBio = `${petName} is ${petAge} years old.`;
console.log(altBio);
```

Template Strings

```
// A 20% tip on $60 would be $12
let getTip = function (total, tipPercent = .2) {
   let percent = tipPercent * 100;
   let tip = total * tipPercent;
   return `A ${percent}% tip on $${total} would be $${tip}`;
}

let tip = getTip(60);
console.log(tip);
```

Template Strings

```
// A 20% tip on $60 would be $12
let getTip = function (total, tipPercent = .2) {
   return `A ${tipPercent * 100}% tip on $${total} would
be $${total * tipPercent}`;
}

let tip = getTip(60);
console.log(tip);
```

Praktika (1)

- Perrašyti funkciją su =>
- Išvedime panaudoti *Template Strings*

```
let friends = ["Mike", "Stacy", "Andy",
"Rick"];

friends.forEach(function (eachName, index){
// 1. Mike, 2. Stacy, 3. Andy, 4. Rick
console.log(index + 1 + ". " + eachName);
});
```