

## Лекция 2



JS data types

JS objects

Flow control



HTML / CSS

JS functions

JSVM

# Variables Scope



## JS scope

Терминът **scope** определя **областта на видимост** на променливите в JS.

Областта на видимост определя до кои променливи имаме достъп в дадена част от кода на програмата.

В JS има два основни вида области на видимост:

- **Глобална (global scope):** ако една променлива е декларирана извън функция или къдрави скоби {},  
то казваме, че тя е дефинирана като глобална (in the global scope). Глобалните променливи могат  
да бъдат достъпени от всяка част на кода.
- **Локална (local scope):** променливи, които са достъпни само в определени части на кода се наричат  
локални променливи.



# JS global scope

```
<script>
```

```
var globalVar = 'Hello';
```

←----- глобална  
променлива

```
function sayHello () {
```

```
var localVar = 'Hello from the inside';
```

←----- локална променлива

```
console.log( globalVar ); //OK
```

```
console.log( localVar ); //OK
```

```
}
```

```
sayHello();
```

```
console.log( globalVar ); //OK
```

```
console.log( localVar ); //Error, localVar is not defined
```

```
</script>
```



# JS local scope

В JS (ES6) има два вида локални области на  
ВИДИМОСТ:

- **Function scope** – определя се от къдравите скоби {}, които определят границите на функцията (цяло на функцията).
- **Block scope** – блокът е парче код, което е оградено между две къдрави скоби и не е цяло на функцията.

```
<script>
  function sayHello () {
    const localVar = 'Hello from the inside'

    if( true ) {
      const blockVar = 'Block';
      console.log( localVar ); //OK
    }

    console.log( localVar ); //OK
    console.log( blockVar ); //Error
  }

  console.log( blockVar ); //Error
  sayHello();
</script>
```

Diagram illustrating the scope of variables in the provided code:

- Function scope:** Indicated by a large dashed box on the left, it encompasses the entire function `sayHello` and its contents.
- Block scope:** Indicated by a smaller dashed box on the left, it encompasses the code block within the `if( true )` statement.

# var, let и const

или

тайният живот на променливите



## Обявяване на променлива с var

Преди ES6 всички променливи в JS се декларираха с ключовата дума **var**.

Променливите, декларирани с var могат да бъдат, както глобални, така и локални.

```
var globalVar = "Hello";
```

```
function greetMe() {  
  var localVar = 'Cheers';  
}
```

```
console.log( globalVar ); //OK  
console.log( localVar ); //Error
```



## Var scope

Променливите, декларирани с **var** са подвластни **само** на **global** и **function scope**. С други думи една var променлива, която е декларирана в блок игнорира скоупа му и е достъпна за цялата функция, където е блокът.

```
function greetMe() {  
    var localVar = 'Cheers';  
  
    if( true ) {  
        var blockVar = 'Greetings';  
    }  
  
    console.log( blockVar ); //OK  
}
```





## Var redefinition

Променливите, декларирани с **var** могат да бъдат декларирани повторно с едно и също име. Това води до промяна на стойността на променливата.

```
function greetMe() {  
  var localVar = 'Cheers';  
  var localVar = 'Hello'; //OK  
  
  console.log( localVar ); //Hello  
}
```

**ВНИМАНИЕ:** в JS това може и да е позволено, но ре-декларирането на променлива **не е добра практика** в никой език и програмистка реалност, защото...



## Var redefinition problem

**ВНИМАНИЕ:** в JS това може и да е позволено, но ре-декларирането на променлива **не е добра практика** в никой език и програмистка реалност, защото...

Така лесно можем да ре-декларираме променлива в блок без да обърнем внимание на това. Това води до синтактично вярна програма, която се изпълнява грешно, бели коси и ядосани колеги.

```
function greetMe() {  
    var localVar = 'Cheers';  
  
    if( true ) {  
        var localVar = 'Greetings';  
    }  
  
    console.log( localVar ); //Greetings  
}
```



# JavaScript hoisting

Hoisting е поведение на JS интерпретатора, при което декларацията на променливи и функции се премества в началото на техния scope преди кодът да се изпълни.

Това прави тези две парчета код еквивалентни.

```
console.log( localVar ); //undefined  
var localVar = 'Cheers';
```

```
var localVar;  
console.log( localVar ); //undefined  
localVar = 'Cheers';
```

При hoist-ването променливите се инициализират, като undefined.

let



## Обявяване на променлива с let

В ES6 се добавя още един начин за деклариране на променливи чрез ключовата дума `let`. Нейната цел е да отстрани недостатъците на `var`. За това ние ще използваме `let` вместо `var`.

Променливите обявени с `let` са подвластни на **block scope**.

```
function greetMe() {  
  
    let localVar = "Cheers";  
  
    if( true ) {  
        let blockVar = 'Greetings';  
        console.log( blockVar ); //Greetings  
    }  
  
    console.log( localVar ); //Cheers  
    console.log( blockVar ); //blockVar is not defined  
}
```



## Let re-declaration

Променливите обявени с **let** не могат да бъдат ре-декларирани (в същия скоуп).

```
let localVar = "Cheers";
```

```
let localVar = "Greetings"; //error: Identifier 'localVar' has already been declared
```

Но могат да бъдат обявявани, като **различни променливи** в различни скоупове.

Стойността на променливата ще се вземе от най-вътрешния скоуп.

```
function greetMe() {  
  let localVar = "Cheers";  
  
  if( true ) {  
    let localVar = 'Greetings';  
    console.log( localVar ); //Greetings  
  }  
  console.log( localVar ); //Cheers  
}
```



## Let hoisting

Правилото за hoisting важи и за променливите, обявени с `let`.

Но с разликата, че те не се инициализират автоматично, и ако се опитаме да ги използваме интерпретаторът ще хвърли грешка Reference Error.

```
console.log( localVar ); //Reference Error, 'localVar' is not defined  
let localVar = "Cheers";
```

**const**





## Ключова дума `const`

“Променливите”, обявени с `const` всъщност са... **константи**.

Веднъж инициализирана **тяхната стойност не може да бъде променяна**.

Константите обявени с `const` са подвластни на **block scope**, по подобие на променливите обявени с `let`.

Това означава, че тяхната стойност си остава една и съща в рамките на скоупа, в който са декларирани.

Константите също не могат да бъдат ре-декларирани.

```
const localConst = "Hi";  
localConst = "Greetings"; //error : Assignment to constant variable.
```

```
const localConst = "Hi"; //error : Identifier 'greeting' has already been declared
```

**ВАЖНО:** Всяка константа трябва да бъде инициализирана по време на създаването и.



# Константи и обекти

Обектите, които са декларирани като константи притежават една особеност. Както можем да предположим, ако един обект е деклариран като константен няма да можем да присвоим друг обект върху него.

```
const constObj = {  
  x: 4  
};  
constObj = {  
  x: 5  
}; //error : Assignment to constant variable.
```

Но можем да модифицираме стойностите на неговите член променливи.

```
const constObj = {  
  x: 4  
};  
constObj.x = 5;  
console.log( constObj.x ); //5
```



# Const hoisting

Правилото за hoisting важи и за константите.

Подобно на променливите, обявени с `let` те не се инициализират автоматично, и ако се опитаме да ги използваме интерпретаторът ще хвърли грешка `Reference Error`.

```
console.log( constObj ); //Reference Error, 'constObj' is not
defined
const constObj = {
  x: 4
};
```

# Data types



# Number

Типът `number` представя както цели числа, така и числа с плаваща запетая. Освен стандартните числови стойности типът може да приема и няколко специални такива. Това са `Infinity`, `-Infinity` и `NaN`.

**NaN** означава **Not a Number** или с други думи резултатът от операцията или стойността не е число.

**Infinity** се разглежда, като математическа безкрайност – стойност по-голяма (или малка) от всички други.

И двете могат да се използват в кода директно.

```
console.log( 1 / 0 ); //Infinity
```

```
let x = Infinity;
```

```
console.log( "string" / 0 ); //NaN
```

```
let y = NaN;
```

```
isNaN(x); //проверка дали стойността е  
NaN
```



# Boolean

Булевият тип има само две възможни стойности true и false.

```
let nameFieldChecked = true; // yes, name field is checked  
let ageFieldChecked = false; // no, age field is not checked
```

Булевият тип може да участва в сравнения, чиито резултат пак е от булев тип:

```
let isGreater = 4 > 1;  
console.log( isGreater ); // true (the comparison result is "yes")
```



## null

В JavaScript има специална стойност **null**, който представлява **празна стойност**.

За разлика от други езици, където null означава, че на мястото на променливата няма нищо и тя не съществува, тук смисълът на null е, че променливата съществува, но нейната стойност е празна, не е конкретизирана или незначителна към момента.

```
let age = null;
```



## undefined

Още една специална стойност е **undefined**.

Тя указва, че **променливата е създадена, но още не е била инициализирана** и към нея **няма присвоена никаква стойност**.

Разликата с null е, че при null все пак имаме нещо присвоено дори и то да е “незнайно”.

```
let x;  
console.log(x); //undefined
```

```
x = undefined;
```





# String

Стринговете в JS съхраняват текст и представляват съвкупност от символи.  
Стринговете се ограждат в кавички единични 'single', двойни "double" и обратен апостроф `backtick`.

Операции със стрингове:

```
let name = "John";
```

```
let fullName = name + ' ' + 'Connor';
```

```
let sentence = "My name is " + fullName + 'and my age is ' + 25;
```



## Template strings (template literals)

В ES6 кавичките обратен апостроф (backtick) дават възможност да се изпълняват изрази, които са вложени в стрингове.

За целта трябва да оградим целия стринг в обратни апострофи и да поставим израза в следната конструкция `${...}`.

```
let sentence = "My name is " + fullName + 'and my age is ' + 25;
```

```
let sentence2 = `My name is ${fullName} and my age is ${20 + 5}`;
```

Темплейтите могат да се влагат едни в

други:

```
const classes = `header ${ isLargeScreen() ? '' : `icon-${item.isCollapsed ? 'expander' : 'collapser'}` }`;
```

Можем да извикваме и функции, но ако те не връщат резултат в стринга ще се изпише `undefined`.



# Objects

Типовете, които разгледахме до тук се наричат “примитивни”, защото съдържат само една стойност.

Обектите са композитен тип, който **може да съдържа множество стойности от различни типове**.

Обект в JS се обявява с къдрави скоби {...} и в тях се поставят произволен брой двойки **име: стойност**.

Обектът всъщност е колекция от именувани променливи към които може да има някакви присвоени стойности:

```
let person = { firstName: "John", lastName: "Connor", age: 25, eyeColor: "blue" };
```

Обектите могат да съдържат и функции, наречени методи.

```
let person = {  
  firstName: "John",  
  lastName: "Connor",  
  age: 25,  
  eyeColor: "blue",  
  fullName: function() {  
    return this.firstName + " " + this.lastName;  
  }  
};
```



## Създаване на обекти

Можем да създадем обект директно, като го обявим с **object literal**:

```
let person = {firstName:"John", lastName:"Connor", age:25, eyeColor:"blue"};
```

Можем да създадем обект и чрез ключовата дума **new** и типа **Object**:

```
var person = new Object();  
person.firstName = "John";  
person.lastName = "Connor";  
person.age = 25;  
person.eyeColor = "blue";
```



## Ключовата дума this

Ключовата дума **this** връща референция към инстанцията на обекта, към когото принадлежи.

В JS тя връща различна стойност (се отнася до различен обект) в зависимост от това къде се използва.

Например:

- В метод **this** сочи към обекта, който притежава метода.
- Във функция **this** сочи към глобалния обект (window).
- В глобалния скоуп **this** сочи към глобалния обект (window).
- В event **this** сочи към обекта, който е породил събитието.
- Във функция при стриктен режим е undefined.

```
let person = {  
  firstName: "John",  
  lastName : "Connor",  
  age:25,  
  eyeColor:"blue",  
  fullName : function() {  
    return this.firstName + " " + this.lastName;  
  }  
};
```



# Array

Масивът е тип, който може да съдържа повече от една стойност в себе си. Масивът предоставя индексиран достъп до стойностите, които съхранява. Масивите в JS могат да съдържат стойности от различен тип. Обявяването на масив става чрез квадратни скоби [...].

```
let users = ['Peter', 'Stan', 'Ivan'];
```

Масив може да се създаде и като обект. Масивът всъщност е обект.

```
let users = new Array('Peter', 'Stan', 'Ivan');
```

Достъпът до елемент от масива става по неговия индекс. Индексите започват от 0.

```
console.log( users[1] ); //Stan
```



# Array

Масивите са специални видове обекти. Поради това, може да има променливи от различни видове в един и същ масив. Може да има обекти в масив. Дори може да има функции в масив. Може да има и вложени масиви в масив.

Тази особеност позволява на масива в JS да се държи, като асоциативен масив без, обаче, да бъде наистина такъв. Индексите могат да бъдат не само числа, но и стрингове.

```
var person = new Array()  
person["firstName"] = "John";  
person["lastName"] = "Connor";  
person["age"] = 25;
```

Достъпът до елемент от масива става чрез името на клетката, чиято стойност искаме да вчемем.

**ВНИМАНИЕ:** имената на клетките не се асоциират с индекси от масива.



# Array operations

```
var fruits = ['Apple', 'Banana'];  
fruits.length; //2  
var newLength = fruits.push('Orange'); // ["Apple", "Banana", "Orange"]  
var last = fruits.pop(); // remove Orange (from the end)  
var first = fruits.shift(); // remove Apple from the front  
var newLength = fruits.unshift('Strawberry') // add to the front  
fruits.push('Mango');// ["Strawberry", "Banana", "Mango"]  
var pos = fruits.indexOf('Banana');// 1  
var removedItem = fruits.splice(pos, 1); // this is how to remove an item  
var shallowCopy = fruits.slice(); // this is how to make a copy  
  
fruits.forEach(function(item, index, array) {  
    console.log(item, index);  
});  
// Apple 0  
// Banana 1
```





## Array map() method

Методът `map()` създава нов масив, като итерира през всички елементи на текущия и им прилага някаква функция.

```
var fruits = ['Apple', 'Banana'];  
var new_array = fruits.map(function callback(currentValue, index) {  
    return `new ${currentValue}`;  
}); //['new Apple', 'new Banana']
```



# Set

Set е колекция, в която обектите са с уникални стойности.

```
var set = new Set();
```

```
set.add(1); // Set [ 1 ]
```

```
set.add(5); // Set [ 1, 5 ]
```

```
set.add(5); // Set [ 1, 5 ]
```

```
set.add('some text'); // Set [ 1, 5, 'some text' ]
```

```
set.has(1); // true
```

```
set.has(3); // false
```

```
var o = {a: 1, b: 2};
```

```
set.add(o);
```

```
set.add({a: 1, b: 2}); // OK
```

# Operators



# Оператор typeof

Ако искаме да узнаем какъв **типът на дадено “нещо”** в JS ще използваме оператора **typeof**.

```
typeof undefined // "undefined"
```

```
typeof 0 // "number"
```

```
typeof true // "boolean"
```

```
typeof "foo" // "string"
```

```
typeof Symbol("id") // "symbol"
```

```
typeof Math // "object" (1)
```

```
typeof null // "object" (2)
```

```
typeof alert // "function" (3)
```



# Аритметични оператори

let y = 5;

Оператор	Описание	Пример	Резултат за y	Резултат за x
+	Събиране	$x = y + 2$	y = 5	x = 7
-	Изваждане	$x = y - 2$	y = 5	x = 3
*	Умножение	$x = y * 2$	y = 5	x = 10
/	Деление	$x = y / 2$	y = 5	x = 2.5
%	Модул / делене с остатък	$x = y \% 2$	y = 5	x = 1
++	Увеличаване с единица	$x = ++y$	y = 6	x = 6
		$x = y++$	y = 6	x = 5
--	Намаляване с единица	$x = --y$	y = 4	x = 4
		$x = y--$	y = 4	x = 5



## Оператори за присвояване

Оператор	Пример	Същото като
=	$x = y$	
+=	$x += y$	$x = x + y$
-=	$x -= y$	$x = x - y$
*=	$x *= y$	$x = x * y$
/=	$x /= y$	$x = x / y$
%=	$x \% = y$	$x = x \% y$



# Битови оператори

Оператор	Описание	Пример	Еквивалент	Резултат
&	AND	$x = 5 \& 1$	$0101 \& 0001$	0001
	OR	$x = 5   1$	$0101   0001$	0101
~	NOT	$x = \sim 5$	$\sim 0101$	0010
^	XOR	$x = 5 \& 1$	$0101 \wedge 0001$	0100
<<	Left shift	$x = 5 << 1$	$0101 << 1$	1011
>>	Right shift	$x = 5 >> 1$	$0101 >> 1$	1010



# Логически оператори

==	равно на	
===	равна стойност и тип	
!=	не е равно	
!==	не равни стойност и тип	
>	по-голяма от	
<	по-малко	
>=	по-голямо или равно	
<=	по-малко или равно	
?	тернарен оператор	Condition ? IfTrue: IfFalse;
&&	логическо И	
	логическо ИЛИ	
!	логическо НЕ	





## Оператор delete

Операторът delete се използва за изтриване на свойства от обекти. Този оператор е предназначен да се използва върху свойствата на даден обект - по тази причина не оказва влияние върху променливи или функции.

```
let person = {firstName:"John", lastName:"Connor", age:25, eyeColor:"blue"};  
delete person.eyeColor;
```