



НИЕ ВЯРВАМЕ ВЪВ ВАШЕТО БЪДЕЩЕ

# Как да пишем работащ JS код?

Работете итеративно (на малки, последователни стъпки).

След всяка нова стъпка, проверявайте как и дали скрипта ви все още работи.

Използвайте `console.log()`, за да си изведете текущите стойности на променливите и резултатите от функциите ви.

Използвайте дебъгера, за да виждате къде какво точно става.

Често грешката е нещо много дребно като лиспваща запетайка или скоба, но пък чупи цялата програма.

**WASTES 2 HOURS DEBUGGING  
BROKEN JAVASCRIPT**



quickmeme.com

# Scope



& Namespace

# Контекст

- Обхват (зона на действие) на това за което се говори. Извън тази зона на действие, това за което се говори няма смисъл.
- Пример от реалния живот:
  - Видя ли Лора?
  - Да.
  - **Как е тя!** извън контекста на разговора, това изречение няма смисъл
- От компютърните езици:

```
function calc(x) {  
    return x / 2; // ok, I know x  
}  
console.log(x); // who/what the hell is x?..
```

# Контекст в JavaScript

- *Scope-а в JavaScript се определя от функциите*
- Scope е зоната на действие на променливите и именуваните функции. Извън тази зона те нямат смисъл (не съществуват)
- Зоната на действие на една променлива или функция е тялото на функцията, в която те са дефинирани
- Т.е. тази зона започва там където е отварящата скоба на обграждащата функция ( “{” ) и свършва със затварящата скоба ( “}” ) или при първия `return`, който ще бъде изпълнен
- Понякога ще казваме “зона на видимост” или “текущ контекст”

# Глобален контекст

- Това е най-външния възможен контекст. Той съдържа всички останали.
- Глобалният контекст е закачен за глобалната променлива `window` (т.е. всичко което дефинираме директно в глобалният контекст, отива в `window`)
- Всички променливи и функции, които се декларират в глобалния контекст, са глобални и съответно видими отвсякъде
- Създаването на глобални функции и променливи се счита за лош стил и може да доведе до непредвидени грешки

# Локален контекст

- Всичко, което не е глобален контекст е локален контекст
- Един вид - всичко, което се намира в тялото на функция (което е оградено от `function() { .. }`)
- Локалните функции и променливи не се виждат отвън!
- Всяка локална функция или променлива, се вижда във всички вложени в нея контексти (scopes)
- Локалният контекст се нарича още *closure*

```
var x = "I'm global";  
  
function level1 () {  
  
    var y = "level 1 variable";  
  
    function level2 (param) {  
  
        var z = "level 2 variable";  
  
        console.log("I can see x and y and z");  
        console.log("I can also see param");  
  
        return x + y + z + param;  
    }  
  
    console.log("I can see x and y and level2");  
    return level2("surprise!");  
}  
}
```

Глобални:  
**x, level1**

Локални за level1:  
**y, level2**

Локални за level2:  
**z, param**

# return

когато направим return във функция, кодът който се намира след return-а не се изпълнява:

```
function example() {  
    console.log("This text will be printed");  
    return;  
    console.log("This text will NOT be printed");  
}
```

Счита се за лош стил ако има прекалено много returns в една функция. В идеалният случай има само един и той е най накрая във функцията, точно преди затварящата скоба ( "}" )

# Namespace

- Както и с контекста - има глобален и локален неймспейс
- Неймспейса е начин да си дефинираме глобални функции, които обаче да не цапат глобалният контекст
- За целта се ползват глобални обекти. За тях не считаме че цапат глобалният контекст, защото тяхните имена започват с главна буква и те могат да съдържат колкото си искат функции и променливи, които вече ще са в тяхния локален контекст, но ще са достъпни от вън през името на самият глобален обект
- пример: Array, Date, Math, Boolean

# Въпроси?

# Errors and Exception Handling



# JavaScript грешки

От време на време кода, който пишем води до появяването на грешки в JavaScript конзолата.

Това са така наречените: **Runtime Errors**.

Наричат се така, защото се случват по време на изпълнението на програмата (скрипта) и като цяло не са хубаво нещо..

# Пример за Runtime Error

WHAT?!..

GPS:

in 5 meters  
turn right.

# JavaScript грешки

- Наричаме ги още изключения (Exceptions) защото обикновено не са част от нормалното изпълнение на скрипта
- “Errors Will Happen!” (W3Schools)
- Всяка грешка спира по-нататъшното изпълнение на нашия скрипт!!!
- За да се справим с грешките има 2 подхода:
  - 1) да предвидим кога ще се случат и да избегнем този вариант
  - 2) да хванем грешката след като се случи и да я “обработим”

# Предвидими грешки

т.е. такива, които могат да се избегнат

```
if (gpsInstructions == nonsense) {  
    drive_without_navigation();  
}  
  
else {  
    drive_with_gps_navigation();  
}
```

# Непредвидими грешки

(такива, които не можем да избегнем)

Когато не сме сигурни какво ще причини  
грешката или тя не зависи от нас ..

(например: ползваме javascript API-а на  
Facebook)

В тези случаи използваме **try-catch** механизма

```
try {
    drive_with_gps_navigation();
}
catch(error) {
    if (error.name == “NavigationError”) {
        console.warn(“This GPS sucks!”);
        drive_without_navigation();
    }
    else { // any other error
        console.error(“Unexpected error: ” + error.message);
        console.log(error.stack);
    }
}
```

# Обработка на грешки (Exception handling)



# JavaScript грешки

- "Хващаме" грешките с catch, ако след това зададем продължение на изпълнението на програмата, казваме че грешката е "обработена"
- Всяка необработена грешка обезсмисля прихващането ѝ, защото само ще "замаже" проблема, без да го реши
- Кои грешки трябва да се прихващат?
  - тези, които са причинени от външни скриптове
  - "собствени" грешки - тези които сами сме направили, за да използваме за контрол на изпълнението на програмата, например при валидация на инпут форма

# Примери

# finally

try-catch блока, има и още една инструкция, която се казва `finally` и се използва за извършване на действия като разрушаване на обекти, затваряне на файлове и т.н. - неща които независимо дали е имало грешка или не, трябва да се изпълнят.

# Използване на собствени грешки

# Кога се налага?

- Често функциите ни очакват да получат параметър от определен тип, например обект или Array, но ако вместо това подадем число, може взичко да се обърка и накрая да получим грешен резултат
- Използваме собствените грешки за да гарантираме че параметрите, които сме получили отговарят на това, което очакваме
- Така се подсигуряваме че функцията ни или ще работи, или ще изведе грешка, но няма да върне грешен резултат
- Често грешките се използват и за **flow control**

# Пример: throw

```
function Person(firstName, lastName) {  
  
    if (firstName == undefined || lastName == undefined) {  
        throw new Error("you must pass first name and last name");  
    }  
  
    this.firstName = firstName;  
    this.lastName = lastName;  
}
```

# Validate the birthdate

```
function Person(name, burthdateString) {  
    this.birthdate = new Date(burthdateString);  
    if (this.birthdate == “Invalid Date”) {  
        throw new Error(“wrong birthdate format!”);  
    }  
}
```

# Error wrapping

- Често искаме да опаковаме една грешка в друга грешка - това се нарича error wrapping

```
try {  
    something();  
} catch (error) {  
    throw new Error("An unexpected error  
occurred: " + error.message);  
}
```

# Въпроси?

# Примери

<http://zenlabs.pro/courses/lessons/lesson20/examples.zip>

# Домашно

<https://github.com/zzeni/swift-academy-homeworks/blob/fe-03/tasks/L20>