



НИЕ ВЯРВАМЕ ВЪВ ВАШЕТО БЪДЕЩЕ

Преглед на домашните

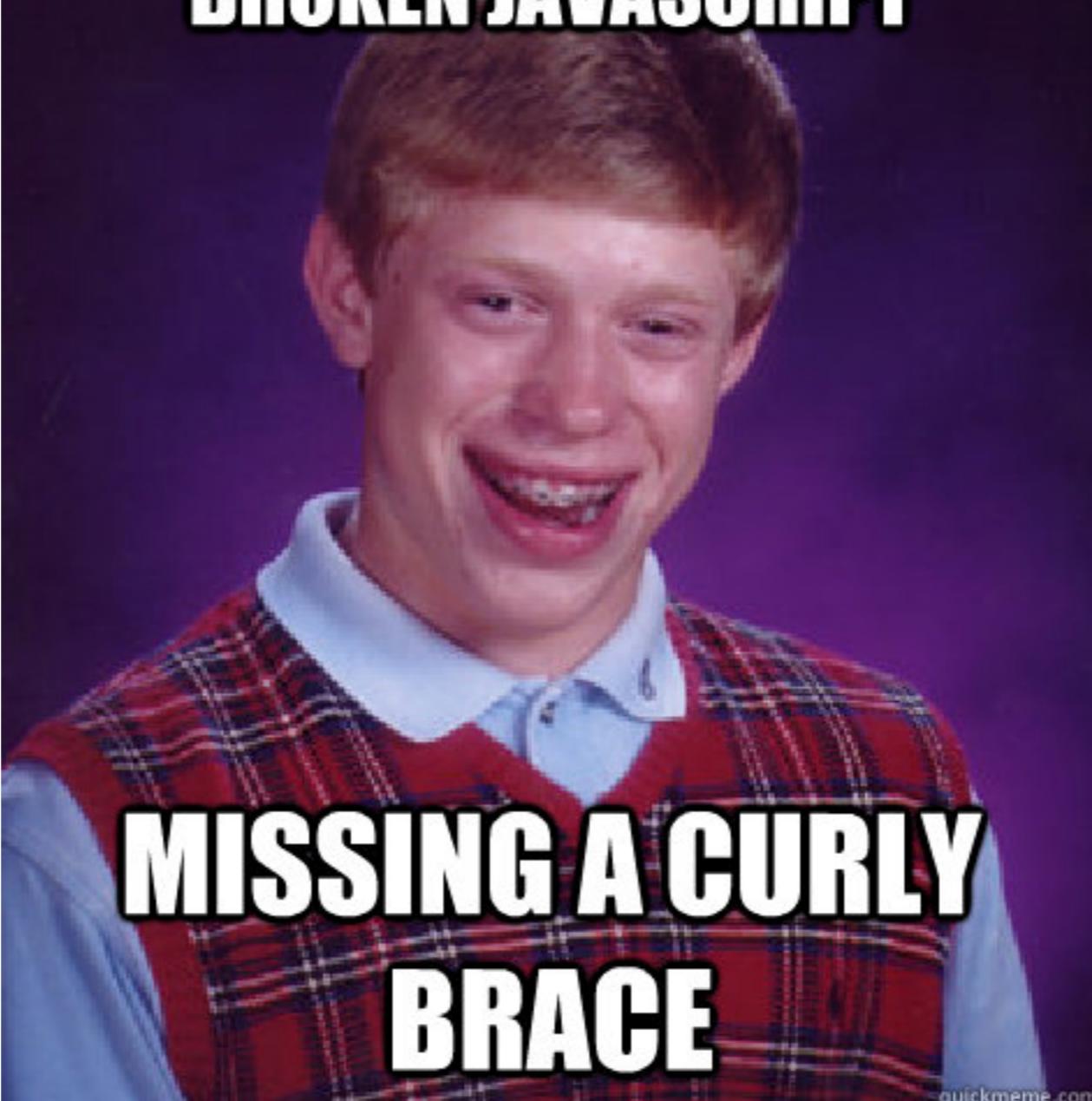
Често грешката е нещо много дребно.

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 точно става.

**WASTES 2 HOURS DEBUGGING
BROKEN JAVASCRIPT**



quickmeme.com

Scope



& Namespace

From the dictionary:

the extent of the area or subject matter that something deals with or to which it is relevant.

т.е.:

*Обхват или зона на действие на това за което се говори.
Извън тази зона на действие,
това за което се говори няма смисъл.*

Примери

- От реалния живот:

- Имаш ли цигари?
 - Да.

- Дай една!

извън контекста, това изречение няма смисъл

- Вчера се видях с Алекс.

дори и това (кой/коя по дяволите е Алекс?..)

- Какво ти каза?

това също

- От компютърните езици:

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

Определение

- Scope е зоната на действие на нашите функции и променливи. Т.е. тази част от кода, където те са дефинирани. Извън тази зона те нямат смисъл (не съществуват)
- Scope-а в JavaScript се определя от функциите
- 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 на вселената е самата вселена. В нея имаме следните вложени един в друг контексти:
 - галактика
 - Сълънчева система
 - атмосфера
 - земя
- 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 някаква функция (което е оградено от къдрави скоби: {})
- **Локалните функции и променливи не се виждат отвън!**
- Всяка локална функция или променлива, се вижда във всички вложени контексти (scopes) на текущия

```
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

Namespace

- Всеки път когато създадем нова функция или променлива, нейното име се записва в т.нар. неймспейс
- Това е каталог с имена подобен на списъците от ученици в даден клас и др. подобни, и е тясно свързан със scope-а
- Namespace е пространството, където се вписват всички имена на променли и функции от текущия scope
- Често се използва като синоним на scope и затова говорим за глобален namespace (всички имена на глобални променливи, функции и обекти) и текущ или локален namespace

Енкапсулация

- Истински добрите developers никога не създават глобални променливи и функции!
- За да го постигнат, разпределят колкото може от логиката на скрипта в класове (които не е проблем да са глобални, стига да не предефинират някой от вградените) и всичко останало се слага в една самоизвикваща се анонимна функция:

```
(function() {  
    ... // all your otherwise global-scope code comes here  
})();
```

- Всеки нов клас, който създаваме, дефинира отделен (собствен) неймспейс, така че всички негови функции и properties, са достъпни единствено чрез този клас и чрез неговите инстанции

Енкапсулация

- Т.е. чрез класовете изолираме много от нашите функции и променливи в локален неймспейс (namespace-а на класа)

Пример:

`Date.now()` може да се извика само през класа Date (т.е. тя е част от неговия namespace);

`getFullYear()` може да се извиква само през инстанции на Date класа

- Винаги трябва да се стремим да декларираме променливите и функциите ни така, че да са видими във възможно най-ограничен namespace
- Това нещо се нарича енкапсулация и е много важно, особено ако искате да покажете висока класа и добър стил на програмиране

return

когато направим return във функция, кодът от функцията 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 ("}")

Въпроси?

Упражнения (scope)

MAKE ALL EXAMPLES



[https://github.com/zzeni/swift-academy-homeworks/tree/
master/tasks/L14](https://github.com/zzeni/swift-academy-homeworks/tree/master/tasks/L14)

1. Напишете функция `greeting`, която приема параметър `name` и като резултат връща стринг, който поздравява.

Например:

```
greeting("Jeni"); // "Hello, Jeni!"
```

2. Напишете клас `Person`, който да задава собствено и фамилно име на конструираните обекти.

Пример:

```
new Person("Бойко", "Борисов"); // Person { firstName: "Бойко",  
lastName: "Борисов" }
```

3. Направете втора функция, която да поздравява, но я кръстете `personGreeting` и вместо име ѝ подавайте обект от тип `Person`.

Пример:

```
personGreeting(new Person("Бойко", "Борисов")); // "Hello, Бойко  
Борисов!"
```

4. Направете така, че последната функция да приема още един параметър `beFormal`, който може да има стойност `true` или `false`. Ако е `true`, нека поздрава да е официален (т.е. по двете имена на человека), а ако е `false` - да е само по първо име.

Пример:

```
var person = new Person("Бойко", "Борисов");
personGreeting(person, true); // "Hello, Бойко Борисов!"
personGreeting(person, false); // "Hi, Бойко!"
```

5. Направете функция (метод) `introduce` в класа `Person`, която да връща стринг, който представя текущия обект (т.е. человека, който е инстанция на класа)

Пример:

```
var person = new Person("Бойко", "Борисов");
person.introduce(); // "Здравейте, казвам се Бойко Борисов!"
```

6. Добавете още едно поле (property) в класа Person, което се казва isPolite

Нека при създаване на обект от този клас, това поле да се задава автоматично на true

Пример:

```
new Person("Бойко", "Борисов"); // Person { firstName: "Бойко",  
lastName: "Борисов", isPolite: true }
```

7. Направете функция (метод) bePolite() в класа Person. Тя трябва да може да променя isPolite пропъртито по подаден параметър, който може да е true или false

Пример:

```
var person = new Person("Бойко", "Борисов");  
person.isPolite; // true  
person.bePolite(false);  
person.isPolite; // false
```

8. Променете метода `introduce`, така че да връща учтиво представяне, когато `isPolite` полето е `true` и неформално представяне, когато `isPolite` е `false`

Пример:

```
var person = new Person("Бойко", "Борисов");
person.introduce(); // "Казвам се Бойко Борисов!"
person.bePolite(false);
person.introduce(); // "Аз съм Бойко!"
```

9. Направете функция (метод) `greet()` в класа `Person`, която получава като параметър друг човек (`greet(otherPerson)`) и която поздравява другия човек:

Пример:

```
var boiko = new Person("Бойко", "Борисов");
var lili = new Person("Лили", "Иванова");
boiko.introduce(); // "Казвам се Бойко Борисов!"
lili.bePolite(false);
lili.greet(boiko); // "Здрави, Бойко!"
lili.introduce(); // "Аз съм Лили."
```

10. Променете метода `introduce`, така че да може да получава параметър от тип `Person` (`introduce(otherPerson)`). Комбинирайте `greet` и `introduce` като използвате резултата от `greet(otherPerson)` в `introduce`.

Променете `greet`, така че да проверява дали параметъра `otherPerson` е подаден (`if (otherPerson != undefined) { .. }`). Ако да - тогава поздравете по име (използвайте съответно учтива и неучтива форма, както досега).

Ако не е подаден параметър `otherPerson`, тогава просто върнете само поздрав без име ("Здрави!" / "Здравейте!")

Пример:

```
var boiko = new Person("Бойко", "Борисов");
var lili = new Person("Лили", "Иванова");
boiko.introduce(); // "Здравейте! Казвам се Бойко Борисов!"
lili.bePolite(false);
lili.introduce(boiko); // "Здрави, Бойко! Аз съм Лили."
```

За домашно

Направете проста визуализация на функциите до момента чрез html.

Нека да имате 2 форми за попълване. Едната ще създава person1 а другата person2, които ще бъдат глобални променливи (такива, които се виждат навсякъде във кода)

Сложете action бутони, които да позволяват двамата човека да си говорят. Може да има и писане на свободен текст.

Визуализирайте всичко това в подходящ контейнер с подходящи стилове.

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

- “Хващайки” грешка, ако зададем някакво 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? - само тези, които са причинени от външни скриптове, ajax http рекуести или грешен user input

Примери

finally

- try-catch блока, има и още една инструкция, която се назова 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?

- Javascript е многооого толерантен към всякакви действия.
Например ако дадена стойност липсва или е лошо форматирана, интерпретатора ще я конвертира в стринг или число вместо да даде грешка
- Това обаче може да е проблем за нашите по-нататъшни изчисления и да предизвика грешка (или още по-лошото - извеждане на грешен резултат) на друго място
- Тогава е разумно да си валидирате "входните" параметри и да извеждате грешка ако те липсват или са лошо форматирани
- Често грешките се използват и за flow control

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;  
}
```

throw

```
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://zenifytheweb.com/courses/lessons/lesson14/example/index.html>

Домашно

<https://github.com/zzeni/swift-academy-homeworks/tree/master/tasks/L14>