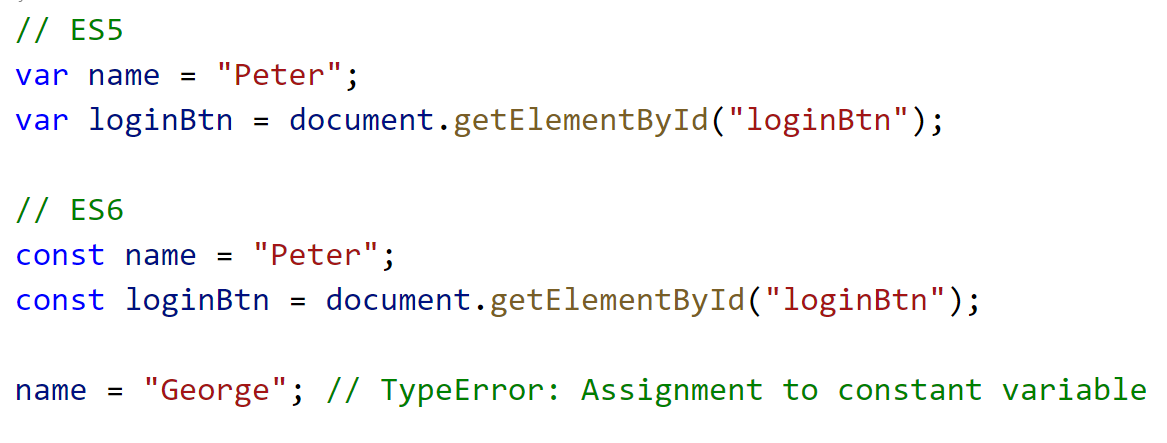
# EcmaScript 6 syntax

След излизането на **ES6** (**EcmaScript 6**) синтаксиса, **JS** разработчиците се радват на **нов** и наистина **динамичен** код, който може вече да се нарече модерен и по-четим (колкото и това да е възможно в еднo JS приложение.. :D). **ЕS6** синтаксиса ни предоставя писането на **по-малко код**, който допринася **повече** като **функционалност**. Нека всъщност да видим всеки един от тези нови features.

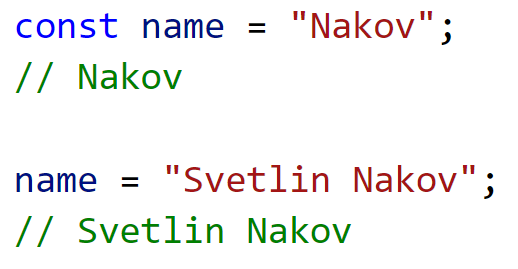
## const and let

**const** е нова ключова дума в **ES6** за деклариране на променливи. Тя е много по надеждна от **var**. Веднъж след като променлива бъде инициализиране с **const** тя не може да бъде презаписана. С други думи тя е **immutable** променлива, с изключение когато се използва за обекти.

Може да бъде изключително полезна в случаи, когато селектираме елементи, които знаем и не очакваме по никакъв начин да бъдат променени. Ако това се случи и се пробваме да променим **const** променлива, ще получим грешка.

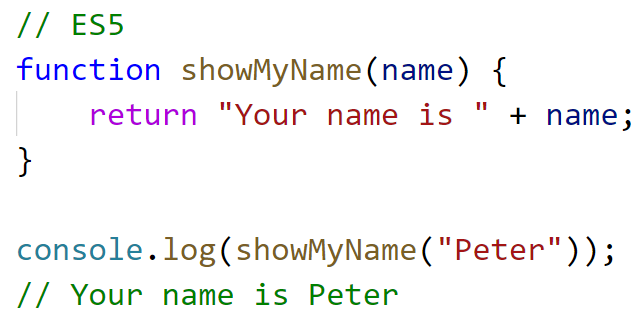


**let** може да бъде презаписана и ще приеме новата стойност, която му бъде зададена (Създава mutable променлива). **let** е както и **const** са blocked-scope променливи (променливата е достъпна само в текущия **scope**). Това накратко казано, значи ще "живеят" от последната отваряща къдрава скоба ("**{**"} до първата затваряща такава ("**}**").

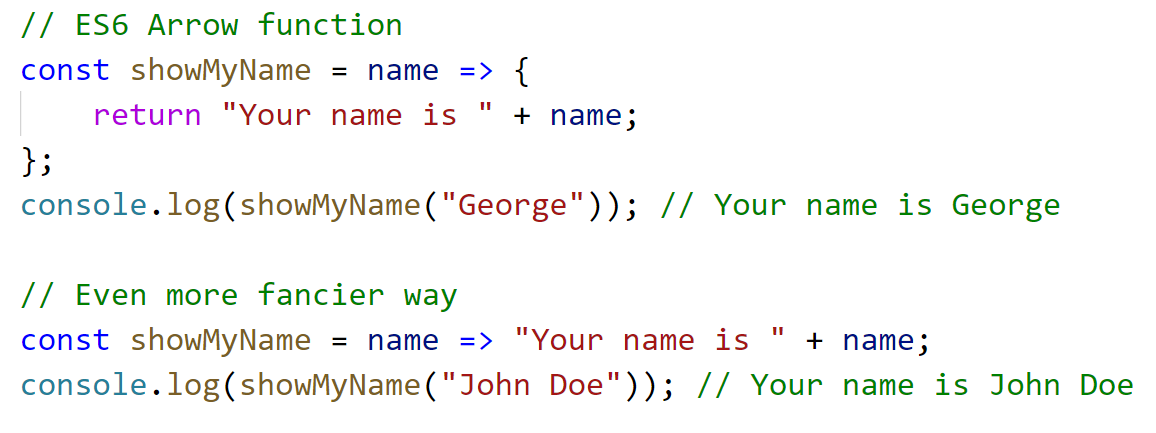


## Arrow functions

Arrow функциите са един от наистина готини features, които ни предоставя ES6. Правят кода ни **по-четим**, **по-структуриран** и изглежда доста **fancy**. Вместо да използваме:

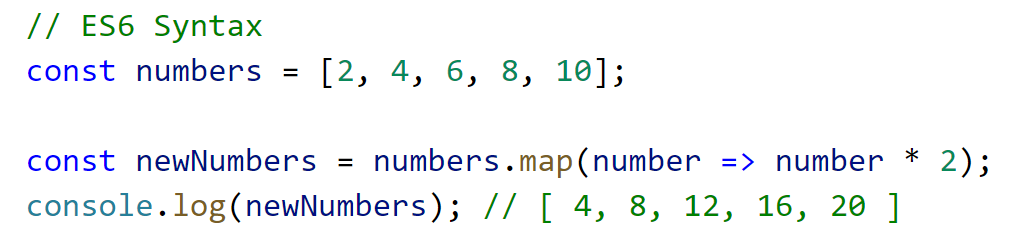


Вече може да използваме:



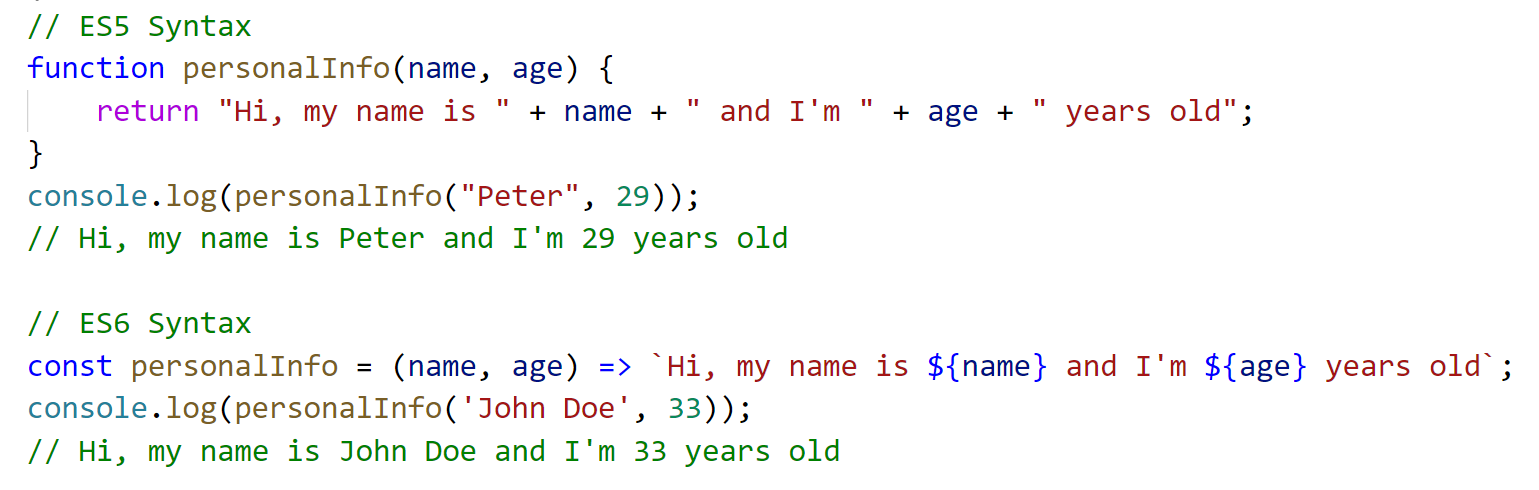
Както виждате, **arrow** **функциите** изглеждат визуално много **по-чисти**. Няма нужда вече да използваме стария синтаксис, освен ако не ни се налага разбира се.

Също така, може да бъдат използвани с **map**, **filter** and **reduce** вградените функции.



**Map** функцията разписана с arrow функция в случая изглежда по-четима и чиста. ES6 синтаксиса ни предоставя по-кратък, но същевремено по-елегантен код. Може да използвате същата логика с **filter** и **reduce** функциите.

## Template Literals

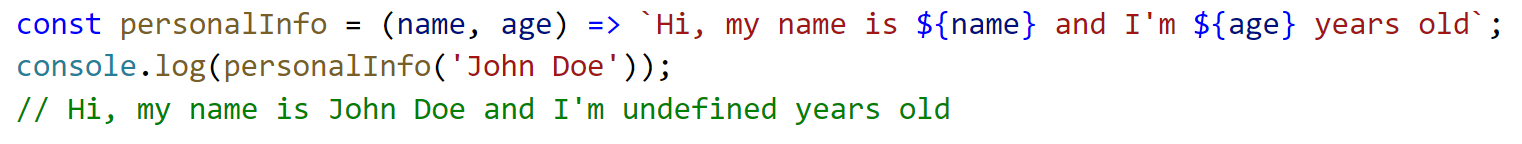
С **темплейтния литерал** или **темплейтен стринг** вече няма нужда да използваме плюс (**+**) оператора за да конкратенираме стрингове или когато искаме да използваме променлива в стринг. 

Изглежда толкова лесно нали? **Разликата** между стария и новия синтаксис е наистина **голяма**. Когато работим със стрингове, **темплейтния литерал** изглежда доста по-организиран и добре структуриран от стария ES5. А и както сами виждате, успяхме да съберем цялата функция на 1 ред 😊

## Default parameters

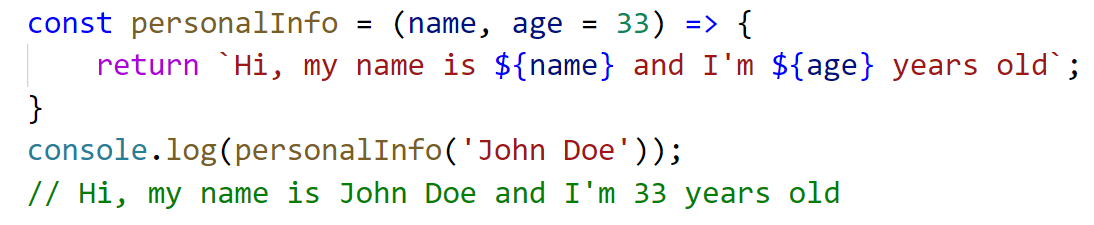
Параметрите по подразбиране (**default parameters**) ни позвоялват да дефинираме параметри в аванс така, че когато забравим да добавим параметър във функцията, която използваме, да не получим **undefined** или грешка. В ситуациите, когато извикаме функция с липсващ параметър, ще бъде взета стойността от параметъра, който сме създали в аванс.

Ето направо един пример:

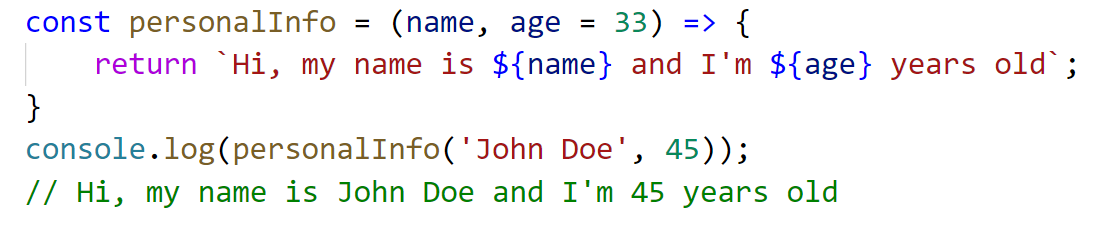


В текущата ситуация, функцията връща **undefined,** защото не сме подали вторния параметър за годините.

Ако имахме **параметър по подразбиране**, нямаше да получим **undefined**, а именно стойността която сме задали.



В случаите, когато зададем параметри в аванс, но при извикването на функцията подадем всички очаквани параметри, които функцията очаква нашите параметри в аванс ще бъдат **презанисани** с тези които функцията получи.

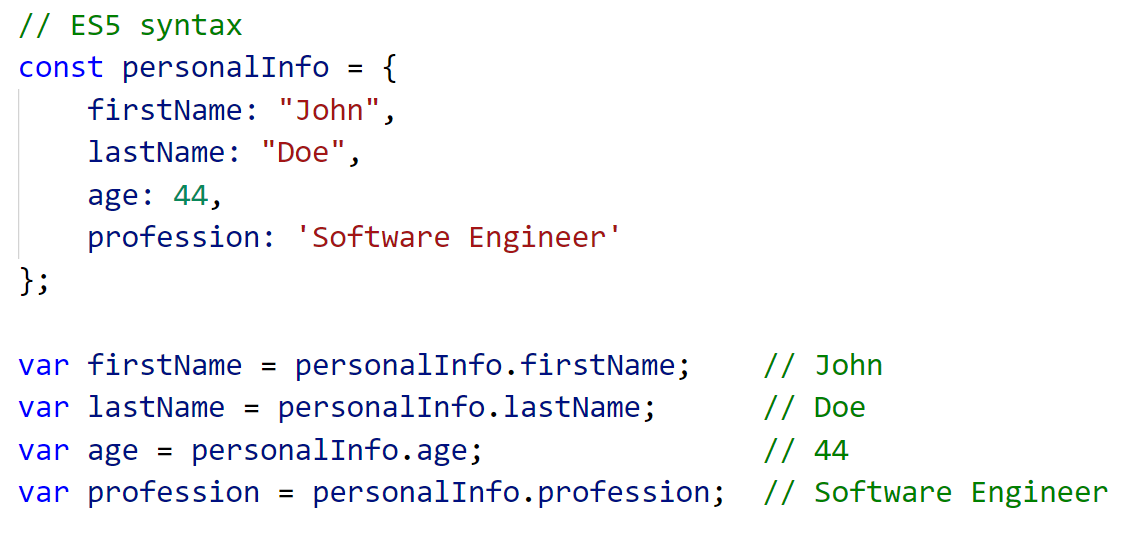


Както виждате, функцията връща стойност, въпреки че пропуснахме втория параметър. Сега с параметъра по подразбиране можем да се справим с грешката предварително.

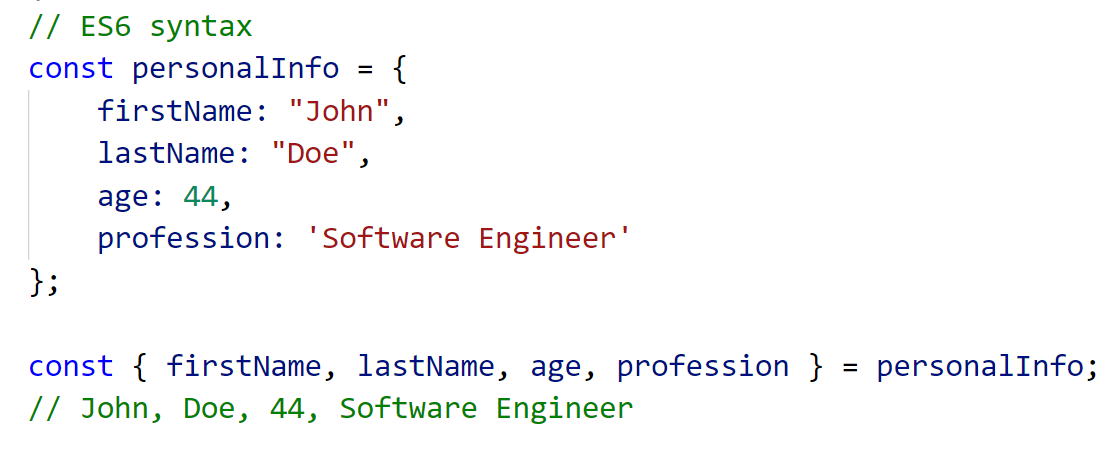
## Array and object destructing

Деструктурирането прави задаването на стойност на масив или обект много по-лесно.

По стария начин, нещата биха изглеждали подобно на:



С ES6 синтаксиса обаче:



С ES5 синтаксиса, трябва да зададем стойност на всяка една променлива по отделно. С ES6 просто задаваме нашите стойностти в къдрави скоби "**{}**", за да вземем всяко едно от пропъртитата на обекта.

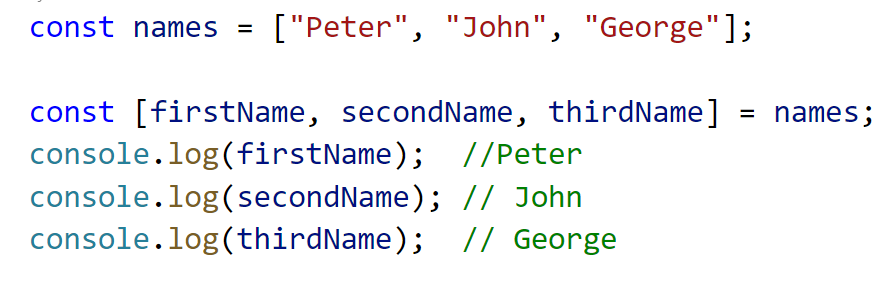
Това с което трябва да се внимава, е че ако инициализираме променлива с неидентично име на прoпърти на самия обект, ще получим undefined. Например ако имаме пропърти с име "**name**" и то бъде записано в променлива "**username**" посредством деструктуриране ще получим **undefined**.



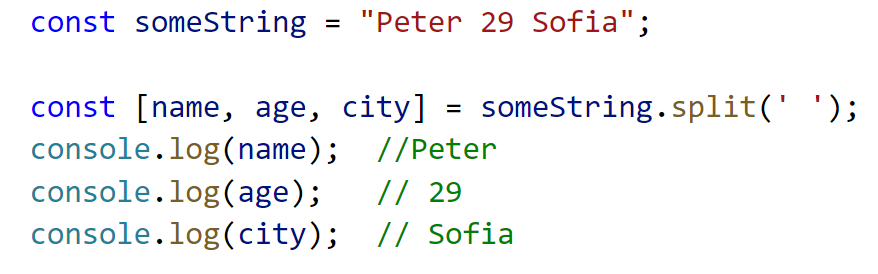
Винаги името на променливата трябва да бъде идентично на името на самото пропърти. В случаите когато искаме да променим името на променливата използвайки деструктуриране, може да използваме "**:**"



В ситуациите когато използваме масиви, може да използваме същия синтаксис като този с обектите. Просто трябва да заместим къдравите скоби с квадратни "**[]**".



Деструктурирането на масиви, може да бъде използвано и в ситуации на **split**.

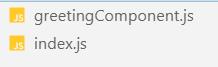


## Import and export

Използвайки **import** и **export** в JavaScript приложение, то става наистина доста "гъвкаво". Те позволяват създаването и преизползването на отделни компоненти (JS логика) през различни файлове.

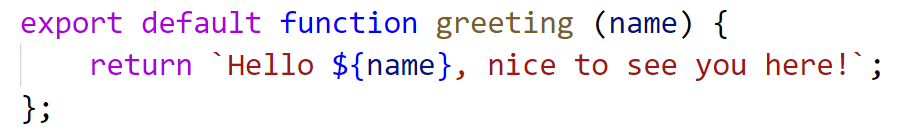
**Еxport** ключовата дума позволява "изнасянето" на модул за да бъде използван в друг JavaScript файл. **Import** от своя страна, "вмъква" вече "изнесен" модул и позволява използването на разписаната в него функционалност.

Да кажем, че имаме 2 отделни файла, "**greetingComponent.js**" и стартова точка "**index.js**"

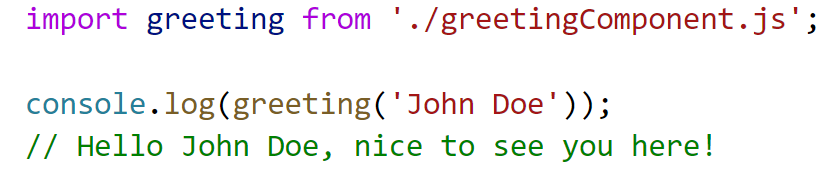


Ако искаме да използваме функционалността на **greetingComponent** в **index** файла, това може да се случи по следния начин:

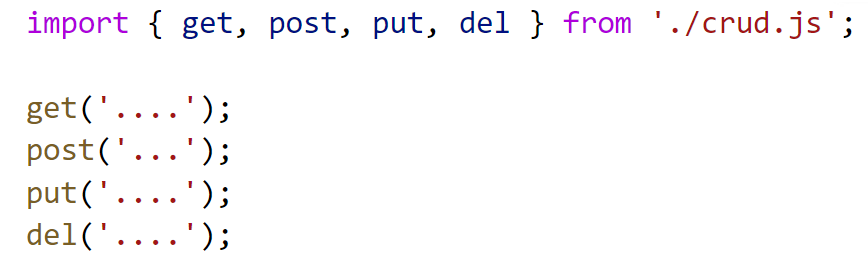
**greetingComponent.js** ще изглежда по подобен начин:



И аналогично, за да използваме вече **export**-натия модул, той трябвада бъде **import**-нат във файла където искаме да бъде използвам.



Ако искаме да изнесем повече функционалност от един модул (файл) това, може да стане като използваме деструктуриране на обект.

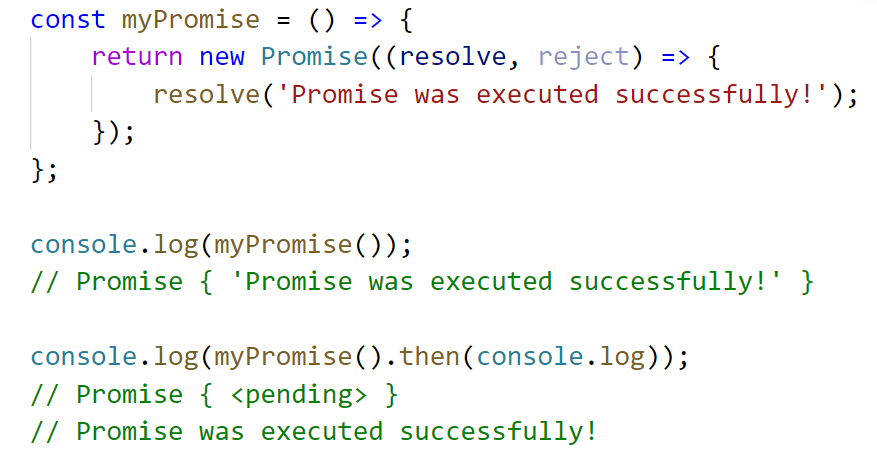


Доста яко, нали? 😊

## Promises

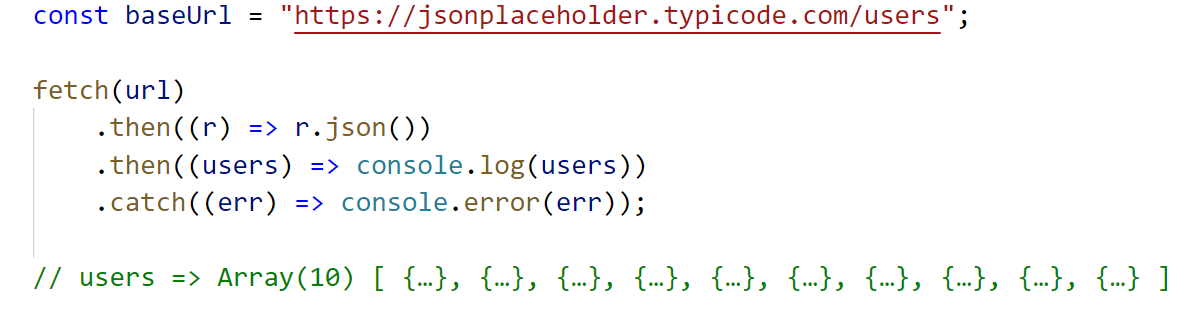
Промисите предоставят възможността за асинхронност в JavaScript. Може да бъдат използвани когато искаме да вземем (достъпим) информация от API (Application Programming Interface) или имаме функция която отнема известно време за да бъде изпълнена. Промисите правят решаването на дадени проблеми много по лесно.

Ето и пример за promise:

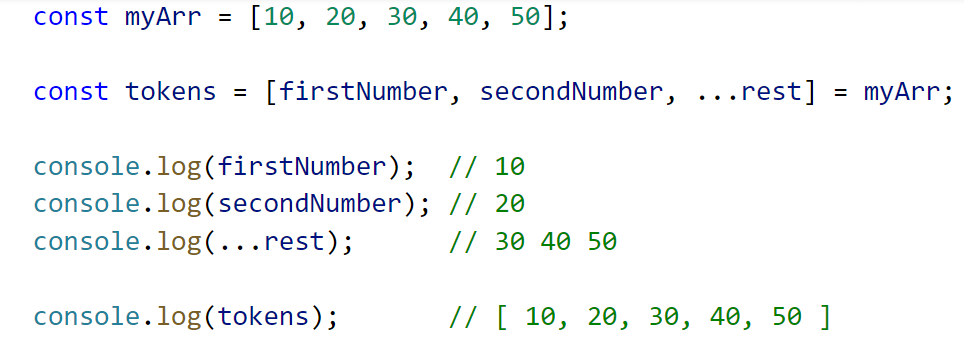


Ако принтираме нашaта функция ще видим, че връща **Promise**. В този случай ако искаме да изпълним функцията след като сме получили информацията от която имаме нужда, трябва да използваме **Promise**. Promise приема два параметъра **resolve** и **reject** за да обработим **неочаквани грешки**.

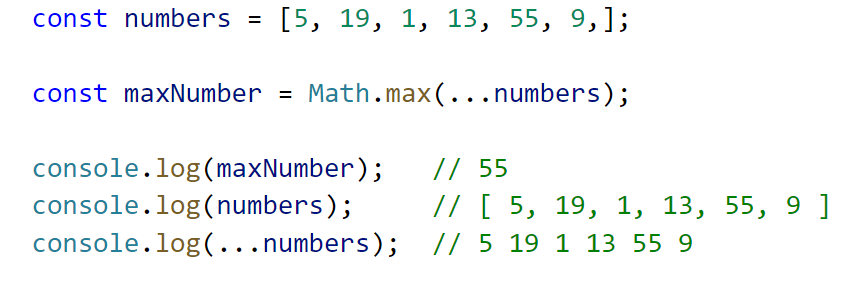
**fetch** функцията сама по себе си връща **Promise** така, че трябва да обработим 2 промиса:



## Rest parameter and Spread operator

**Rest параметъра** се използват за получаване на аргумента на масив и връща нов масив.

**Spread** **оператора** има същия синтаксис като **rest параметъра**, но той приема масив, а не аргументи от такъв. Може да използваме **spread оператора** за да вземем стойностите от масив вместо да използваме цикъл или друг метод.

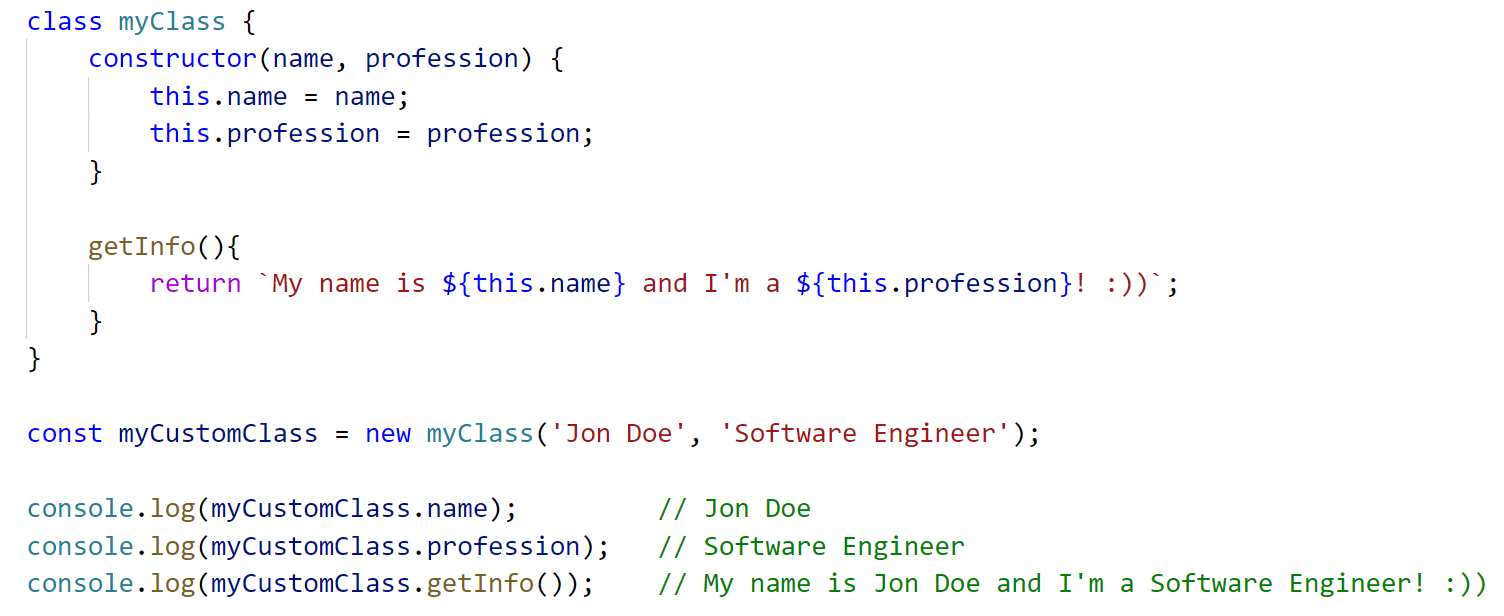


## Classes

**Класовете** са **ядрото** на обектно ориентираното програмиране (**ООП**). Те правят кода по-структуриран както и капсулиран (защитен). Използването на класове придава на нашия код добра структура и лесно може да се ориентираме в него.

За да създадем **клас** трябва да използваме ключовата дума "**class**"следвана от **име** на класа и **къдрави** **скоби**.

След като разпишем някаква логика в нашия клас, можем да получим достъп до нея, като използваме ключова дума "**new"**:



## Conclusion

***Надяваме се статията да е била полезна и да сте научили нещо ново, което ще ви бъде полезно за вбъдеще. Поздрави!***

☺