

O knihe

Predslov

Učebnica Javascript / jQuery je určená dizajnérom, ktorí už nechcú otravovať programátora kvôli každému malému efektu na stránke. Je určená HTML rezačom a CSS štýlovačom, ktorí sa chcú naučiť svoju stránku rozhýbať. Je pre každého, kto robí stránky alebo sa ich robiť učí a Javascriptu sa doteraz vyhýbal.

Učebnica je určená začiatočníkom a prevažne sa venuje knižnici jQuery. jQuery je nadstavba jazyka Javascript a každý, kto píše jQuery, píše Javascript. Niektoré koncepty jazyka Javascript prejdú čitateľovi do krvi skrz samotné používanie jQuery, iné budú vysvetlené podrobnejšie.

Kniha predpokladá znalosť technológií HTML a CSS.

Autor

Autorom učebnice je *Roman Hraška*, ktorý sa tvorbe webov venuje dlhé roky. Je autorom niekoľkých videokurzov, ako napríklad *Nauč sa jQuery* alebo *Robíme Wordpress*. O webe píše na stránke brm.sk a patril medzi zakladajúcich autorov portálu zajtra.sk.

Pripomienky

Pripomienky, návrhy alebo otázky smerujte na adresu yablko@learn2code.sk.

Prehľad technológií

Javascript je programovací jazyk vstavaný do prakticky každého internetového prehliadača. Najčastejšie sa používa tzv. „na strane klienta“. To znamená, že po načítaní stránky s ňou prehliadač dokáže manipulovať, často na popud používateľa.

Ak sa obsah stránky zmení po jej načítaní, má v tom prsty Javascript. Ak kliknete „palec hore“ pod youtube videom a toto opalcovanie sa uloží do databázy bez toho, aby sa stránka refreshla, má v tom prsty Javascript. Ak sa niečo zasunie alebo vysunie alebo vyroluje alebo ak obrázok vyskočí v peknom okne v strede stránky, bude to Javascript.

Javascript je všestraný jazyk, dokáže mnohé, ale najčastejšie sa používa pri tvorbe internetových stránok. Preto sa oplatí dať si ho do kontextu s ostatnými technológiami, ktoré už ovládate.

HTML

HTML je zodpovedný za **obsah** stránky. Je to základná kostra každej stránky. Obsahuje všetky odstavce a zoznamy a obrázky a odkazy a ostatné dôvody, prečo návštevník zavítal na vašu stránku.

CSS

Návštevník síce dostane všetko čo potrebuje vďaka HTML, ale návštevník je rozmaznaný. On nechce kus textu. On chce kus pekného textu! A na to je tu CSS. Ak HTML je obsah stránky, CSS je **forma**. To, ako sa stránka vyzerá.

Javascript

HTML je obsah stránky, CSS je to, ako stránka vyzerá a Javascript je to, ako sa stránka správa. Javascript je zodpovedný za **správanie** stránky a jej **interaktivitu**.

Javascript je dnes podopiera prakticky na každú stránku, ktorú navštevujete. Existuje však isté (mizivé) percento ľudí, ktorí majú v prehliadači Javascript zakázaný. Každá stránka by mala fungovať aj pre nich. Javascript, teoreticky, nemá vytvárať novú funkcionálnosť, má len spríjemňovať existujúcu.

Ak v prehliadači kliknem na odkaz smerujúci na obrázok, otvorí sa tento obrázok v nom tabe prehliadača. Vďaka Javascriptu sa môže otvoriť pekne v strede v stránky, ešte aj s chrumkavou animáciou. Toto je ideálny príklad, kedy Javascript spríjemní pobyt na stránke, ale prežili by sme to, aj keby neexistoval. Boli by sme síce vytrhnutí zo stránky a museli by sme liezť do nového tabu a ten potom zatvárať a bolo by to celé škaredé a otravné, ale fungovalo by to.

A tak to má byť.

Javascript mimochodom nemá nič spoločné s programovacím jazykom Java. Okrem prvých štyroch písmen. Tie má veľmi spoločné. Inak je to ale niečo úplne iné.

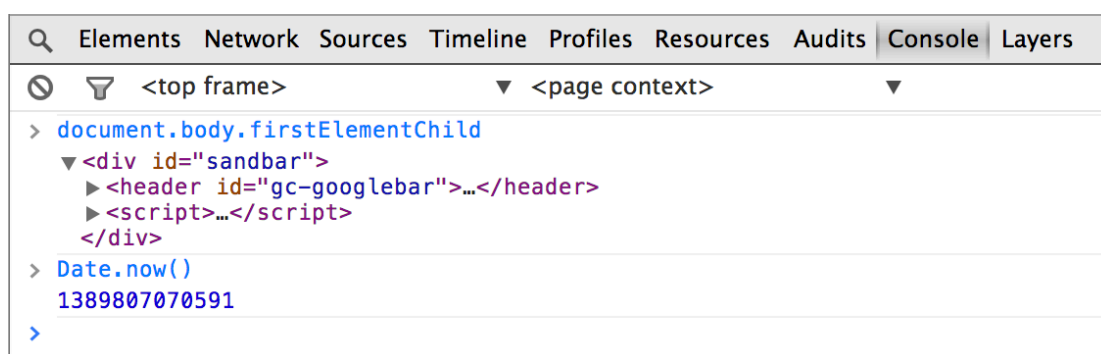
Rýchlokurz programovania

Táto sekcia je určená tým, ktorí sa s programovaním nikdy nestretli. Prípadne sa utekali schovať vždy, keď bolo spomenuté. O programovaní prevláda názor, že je náročné a komplikované a pochopiteľné iba pre špeciálne vyšľachtený typ superčloveka. Ale to platí iba keď programujete *World of Warcraft* alebo počítate statiku domu. Inak je to pohodička.

Jediný rozdiel medzi programátorom a neprogramátorom je, že ten prvý do toho investoval viac času.

Chrome Developer Tools

Každý moderný prehliadač vie spúšťať Javascript a každý má v sebe nástroje, ktoré nám pomáhajú s jeho písaním. Otázka výberu zostáva na každom čitateľovi, táto učebnica však budete odkazovať na **Chrome Developer Tools**, ktoré sú súčasťou prehliadača **Google Chrome**.



K tzv. **DevTools** sa dostaneme cez *Chrome menu > Tools > Developer Tools* alebo pomocou klávesovej skratky:

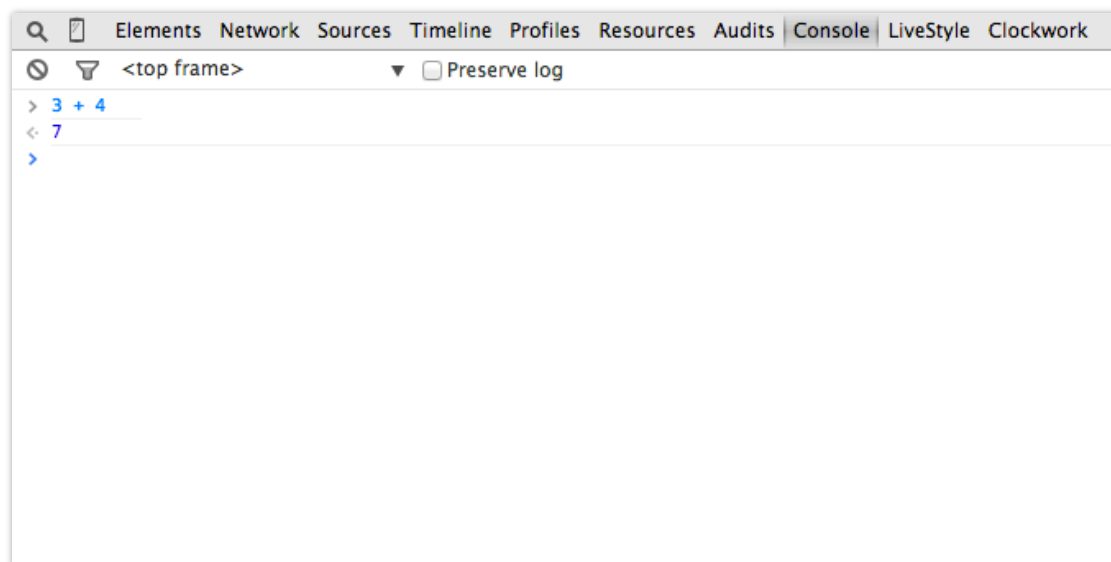
	Windows / Linux	Mac
Open Developer Tools	F12, Ctrl + Shift + I	Cmd + Opt + I
Open / switch from inspect element mode and browser window	Ctrl + Shift + C	Cmd + Shift + C
Open Developer Tools and bring focus to the console	Ctrl + Shift + J	Cmd + Opt + J

Inspect the Inspector (*undock first one and press*)

Ctrl + Shift + J

Cmd + Opt + J

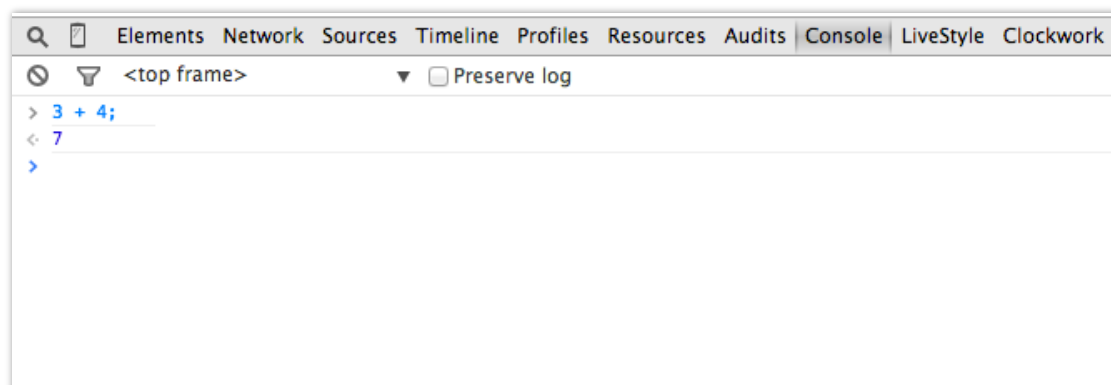
Otvorte DevTools, prejdite do tabu **Console** a napíšte `3 + 4`:



Gratulujem, práve ste programovali!

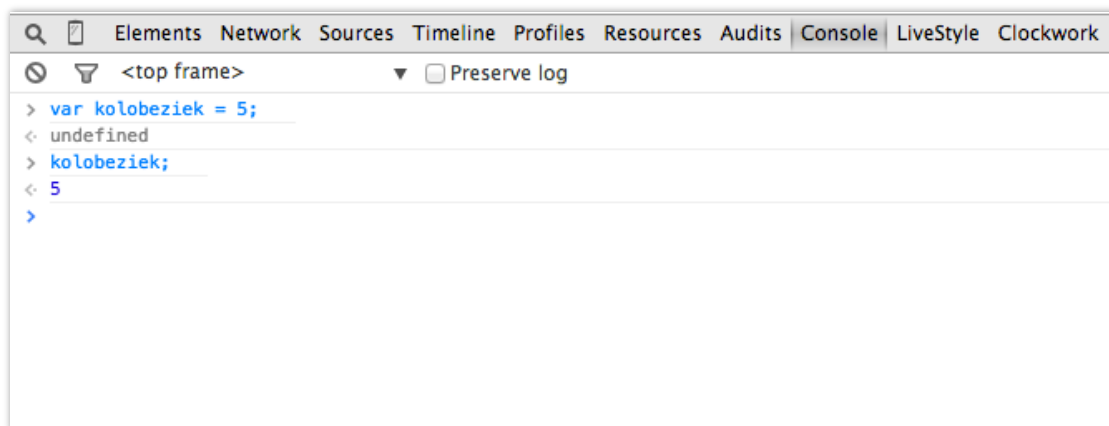
Pomocou operátora `+` ste zavolali operáciu sčítania na čísla `3` a `4`. To je váš kód, zobrazený v prvom riadku. V druhom riadku je číslo `7`, výsledok tejto operácie.

Zároveň ste ale spravili jednu drobnú chybu. Každý príkaz v Javascripte by mal končiť **bodkočiarkou**. Chrome to síce Schrúmal aj bez nej, lebo je šikovný, ale ak si chcete byť na istom, vždy ukončujte príkazy bodkočiarkou.



Premenné (variable)

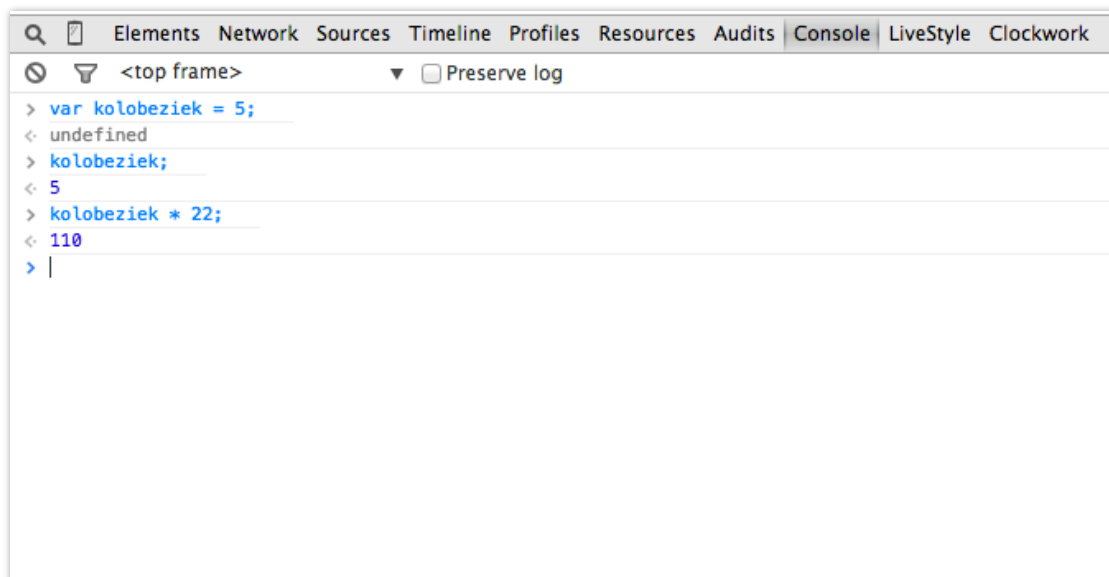
3 a 4 sú fajn čísla, ale niekedy nevieme, ktoré čísla budeme potrebovať. Programátor vášho internetového obchodu nevie, či si návštevník príde kúpi 5 alebo 13 kolobežiek. V tomto prípade môžeme to číslo vyhľadať na stránke a uložiť si ho do **premennej**.



```
Elements Network Sources Timeline Profiles Resources Audits Console LiveStyle Clockwork
<top frame>
> var kolobeziek = 5;
< undefined
> kolobeziek;
< 5
>
```

Začneme kľúčovým slovom **var**, ktoré dáva najavo, že nasledovať bude názov premennej. V našom prípade **kolobeziek**. Znak **=** slúži na priradenie hodnoty k premennej.

Program si teraz pamätá, že k slovu „kolobeziek“ je priradená hodnota 5 a tento fakt môžeme následne používať.

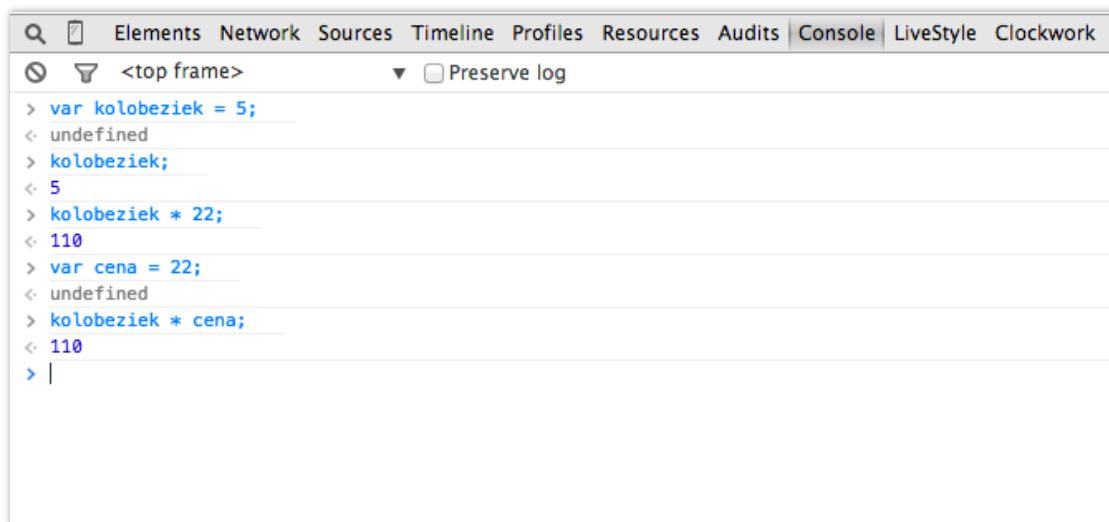


```
Elements Network Sources Timeline Profiles Resources Audits Console LiveStyle Clockwork
<top frame>
> var kolobeziek = 5;
< undefined
> kolobeziek;
< 5
> kolobeziek * 22;
< 110
> |
```

Za predpokladu, že jedna kolobežka stojí 22€, sme práve vypočítali, koľko bude návštevníka stáť celý nákup. Symbol ***** je pri číslach zodpovedný za ich násobenie, naopak / sprostredkuje delenie. Symbol **-** **slúži**, ako by sa dalo čakať, na odčítanie.

Lepší nápad by však bol, keby si cenu jednej kolobežky taktiež uložíme do premennej. V programe ju pravdepodobne budeme potrebovať na viacerých

miestach – bude zobrazená pri produkte, bude použitá pri sčítavaní po pridaní do košíka, atď. Ak by sme potrebovali v budúcnosti cenu zmeniť, museli by sme ju prepisovať na mnohých miestach.



```
Q [🔍] Elements Network Sources Timeline Profiles Resources Audits Console LiveStyle Clockwork
<top frame> [▼] [ ] Preserve log
> var kolobeziak = 5;
< undefined
> kolobeziak;
< 5
> kolobeziak * 22;
< 110
> var cena = 22;
< undefined
> kolobeziak * cena;
< 110
> |
```

V tomto prípade, ak ju niekedy budeme musieť zmeniť, stačí cenu prepísať iba na jednom mieste.

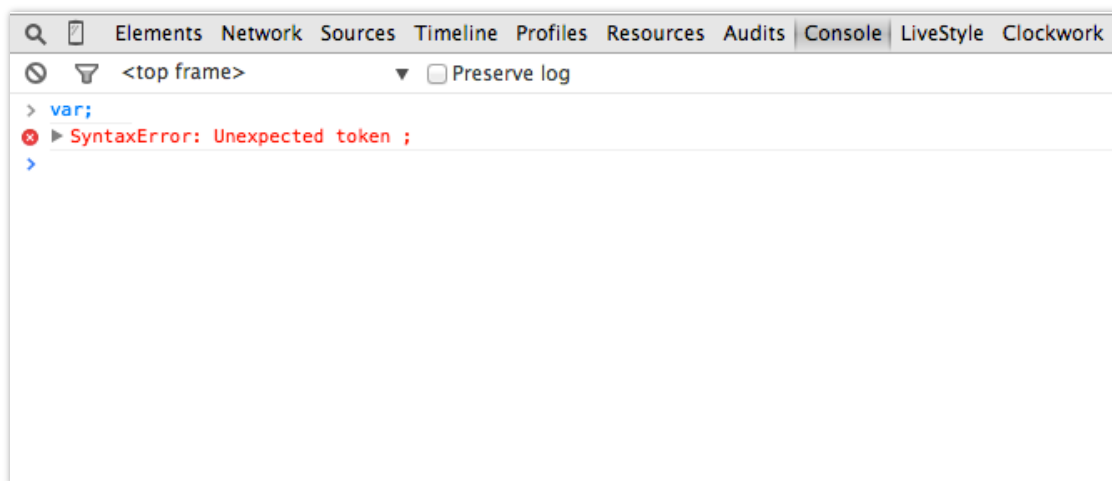
*Pozn.: slovíčko **undefined** v obrázkoch si teraz netreba všímať. Konzola sa snaží vyhodnotiť výsledok operácie a vypísať ho na obrazovku. Avšak operácia priradenia do premennej nemá žiadny zobraziteľný výsledok, výsledok je „nedefinovateľný“ resp. „nie je zadefinovaný“ a preto vidíme „undefined“.*

Reťazce (string)

String znamená reťazec a reťazce v tomto prípade znamenajú reťazce znakov. Reťazec môže byť slovo, veta, odstavec... A to sa oplatí, pretože v programovaní a na stránkach a v živote často potrebujeme aj slová, nie len čísla. Pozorný čitateľ si všimne, že autor používa slová aj priamo v tejto učebnici!

V príkladoch sme videli slovíčko **var**. Čo ak by sme chceli vypísať slovo „var“? Možno ako súčasť škaredého príkazu: „Hej, kuchár, var!“

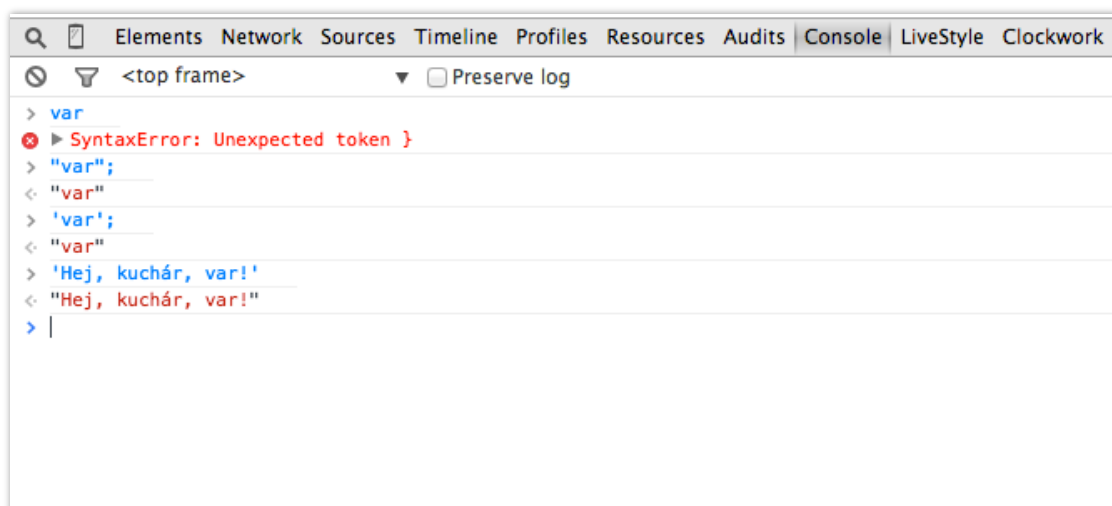
Asi nepochodíme. Pretože **var** je jedno zo slovíčok, ktoré má Javascript vyhradené pre svoje vlastné machinácie.



```
Elements Network Sources Timeline Profiles Resources Audits Console LiveStyle Clockwork
<top frame>
> var;
SyntaxError: Unexpected token ;
>
```

Program vyhodí chybu – **neočakávaná bodkočiarka!** Pretože slovíčko var znamená, že ideme zdefinovať premennú a za ním by mal nasledovať jej názov a teda rozhodne nie bodkočiarka.

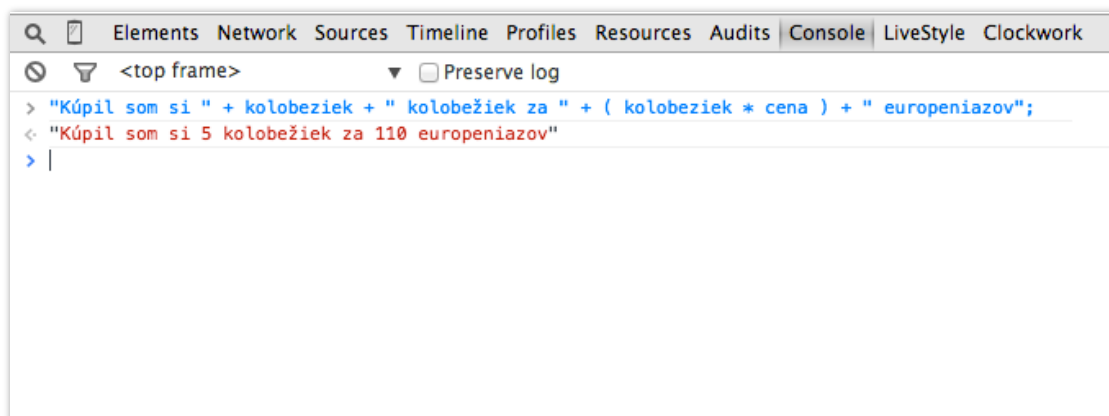
Ak však použijeme úvodzovky (buď obyčajné alebo dvojité), sme doma.



```
Elements Network Sources Timeline Profiles Resources Audits Console LiveStyle Clockwork
<top frame>
> var
SyntaxError: Unexpected token }
> "var";
< "var"
> 'var';
< "var"
> 'Hej, kuchár, var!'
< "Hej, kuchár, var!"
> |
```

Voľba úvodzoviek je na vás, ale nemixovať! Jedno alebo druhé.

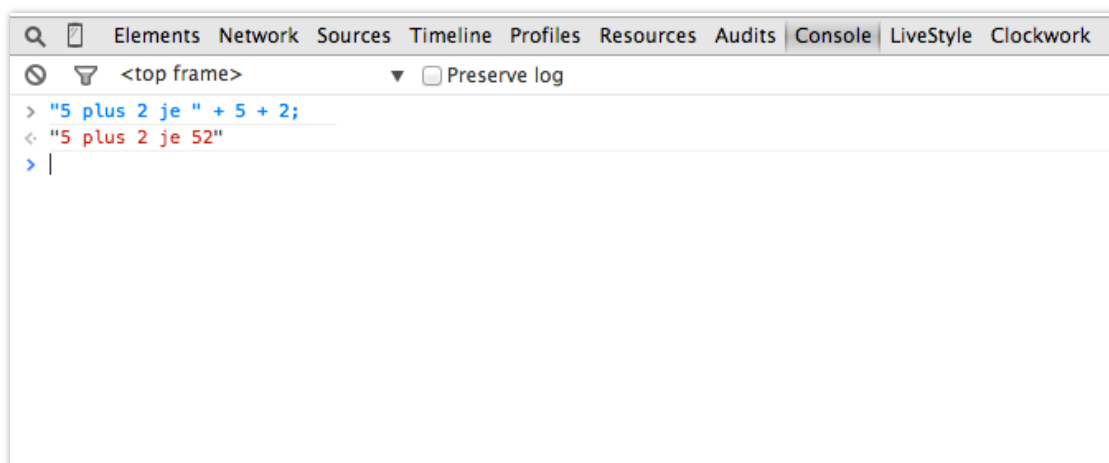
Teraz môžeme použiť naše hodnoty uložené v premenných a vytvoriť inteligentne znejúcu vetu:



```
Elements Network Sources Timeline Profiles Resources Audits Console LiveStyle Clockwork
<top frame>
> "Kúpil som si " + kolobežiek + " kolobežiek za " + ( kolobežiek * cena ) + " europeniazov";
< "Kúpil som si 5 kolobežiek za 110 europeniazov"
> |
```

Vieme, že v premenných **kolobežiek** a **cena** máme uložené čísla. Tu sa ich však snažíme nalepiť na string. Vidíme, že sme znova použili symbol **+**. Ak tento symbol použijeme medzi dvoma číslami, ščítame ich. Ak ho použijeme medzi dvoma stringami, spojíme ich do jedného. Ak medzi stringom a číslom, prekladač jazyka Javascript zhodnotí, že najlepšie bude zmeniť číslo na string a spojiť to celé do jedného megastringu.

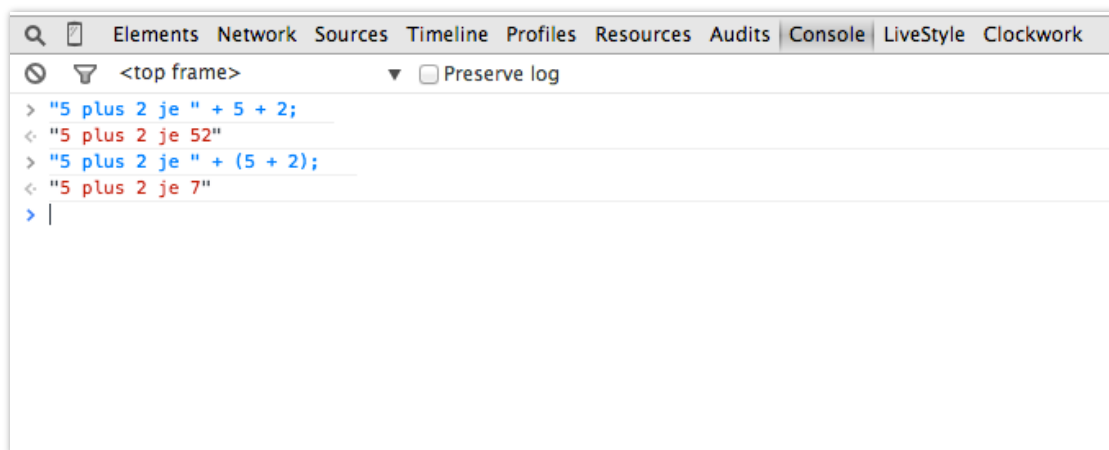
To je veľmi šikovné, ale môže to robiť problémy v nasledovnej situácii:



```
Elements Network Sources Timeline Profiles Resources Audits Console LiveStyle Clockwork
<top frame>
> "5 plus 2 je " + 5 + 2;
< "5 plus 2 je 52"
> |
```

Kto niekedy skúšal položiť 2 eurovú mincu na 5 eurovú bankovku vie, že z toho 52 eur nevznikne. Tak kde sa stala chyba? V podstate nikde, Javascript len videl, že cheme ku stringu pridať číslo 5, tak ho zmenil na string "5" a pridal na koniec. Potom videl, že k tomuto novému stringu chceme pridať číslo 2, tak ho zmenil na string "2" a pridal na koniec.

Ak však operáciu sčítania uzavrieme do zátvorky, stane sa to, čo by každý mladý vlastník siedmich eur očakával:



Operácia v zátvorkách, podobne ako v matematike, má prioritu a najprv sa vyhodní ona. Až jej výsledok sa potom napojí na string.

Polia (array)

Pole je spôsob ako uložiť viac hodnôt na jedno miesto. S tým, že každá má svoju pozíciu (svoj index) na základe ktorého k hodnotám pristupujeme. Podobne, ako keď 5 ľudí stojí v rade na banány, Štefan je prvý, Timotej je druhý, atď. Až na to, že v programovaní vždy začíname ako Rytmus, od nuly.

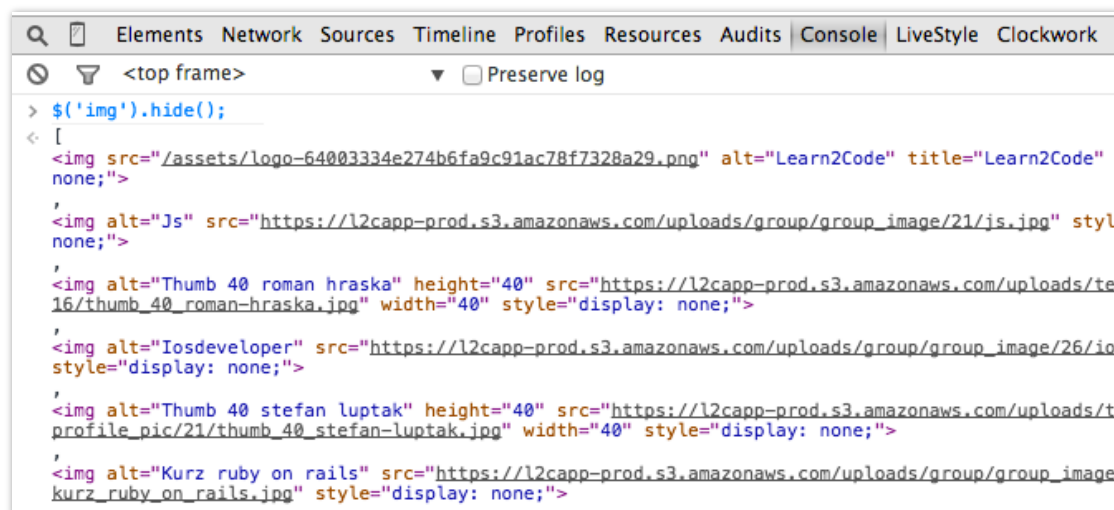


Pole definujeme pomocou hranatých zátvoriek, každý prvok poľa od ďalšieho oddelíme čiarkou.

K indexu prístupujeme pomocou hranatých zátvoriek. Zápis **rad[2]** znamená „vráť mi z poľa **rad** hodnotu prvku s indexom **2**“ a keďže indexy začíname počítať od nuly, znamená to „Vráť mi z poľa prvok na **treťom** mieste.“ V príklade vidíme, že v poli sa môžu nachádzať rôzne typy hodnôt. Tu kombinujeme čísla s reťazcami.

V praxi budeme polia pri práci s jQuery používať neustále, aj keď to nebude na prvý pohľad zrejmé. jQuery totiž pracuje s kolekciami elementov.

Tento jQuery príkaz znamená „vyber všetky `` elementy na stránke a skry ich.“



```
> $('img').hide();
< [
  ,
  ,
  ,
  ,
  ,
  
]
```

A ako vidíme, tento príkaz nám vrátil pole (kolekciu) všetkých obrázkov na stránke hneď po tom, ako ich skryl. Vidíme ľavú hranatú zátvorku, vidíme `img` elementy, každý z nich je oddelený čiarkou reprezentujúce prvky poľa, a keby dovidíme až na koniec, bola by tam pravá hranatá zátvorka.

Objekty (object)

Polia definujeme hranatými zátvorkami, objekty definujeme zloženými zátvorkami. Pole je kolekcia elementov, objekty je kolekcia párov **klúč -> hodnota**. Ktorýkoľvek objekt z reálneho sveta by sme dokázali popísať jeho vlastnosťami a schopnosťami.

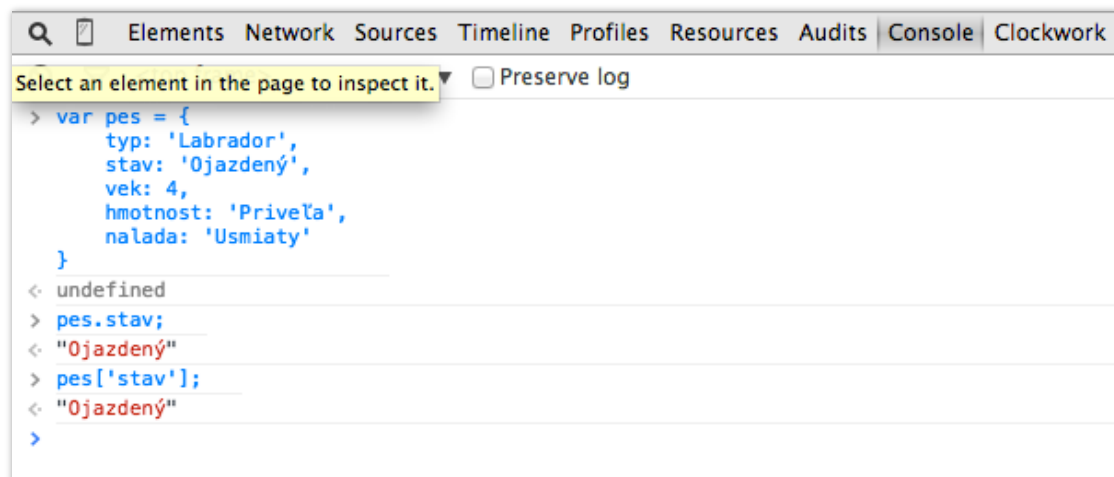
PES	
Kľúč	Hodnota
Typ	Labrador
Stav	Ojazdený
Vek	4
Hmotnosť	Priveľa
Nálada	Usmiaty

V JavaScripte by sme toho havina zapísali nasledovne:

```
var pes = {  
  typ: 'Labrador',  
  stav: 'Ojazdený',  
  vek: 4,  
  hmotnost: 'Priveľa',  
  nalada: 'Usmiatey'  
}
```

Object sme definovali zloženými zátvorkami. Hodnota kľúča **vek** je typu číslo, hodnota kľúča **stav** je typu reťazec. Hodnoty môžu byť aj typu pole alebo pokojne aj ďalší objekt. Kľúče zapisujeme bez interpunkcie, každý pár je od seba oddelený čiarkou, za posledným párom **nesmie** byť čiarka.

K hodnotám objektu pristupujeme podobne ako k hodnotám poľa, alebo pomocou „bodkovej notácie“:



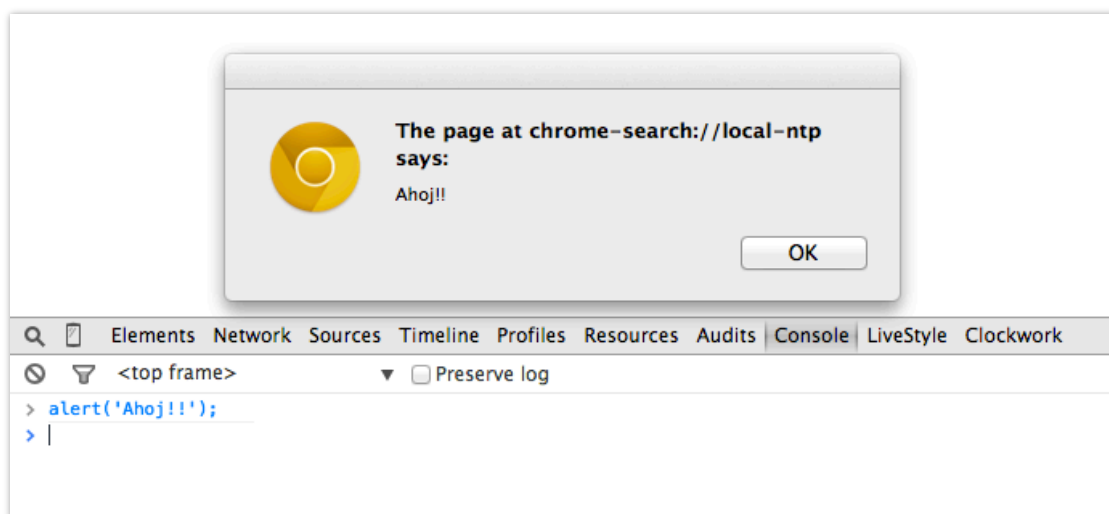
Každá vec, každá burgyňa, čo rastie v záhrade, iPhone, do ktorého by ste sa radšej pozerali namiesto čítania tohoto, miešačka, cymbal, Tomáš Mašťalír, ...každá vec má svoje vlastnosti.

A každá má tiež svoj schopnosti. K tým sa dostaneme.

Funkcie (funkcie)

Z matematiky si možno funkcie pamätáte ako kadejaké divné grafy, ktorým ani Svätý Peter nerozumie. Funkcie v programovaní sú však len zgrupené príkazy, ktoré sa snažia vyriešiť nejaký problém.

Javascript má kopu vstavaných funkcií, ktoré môžeme používať.

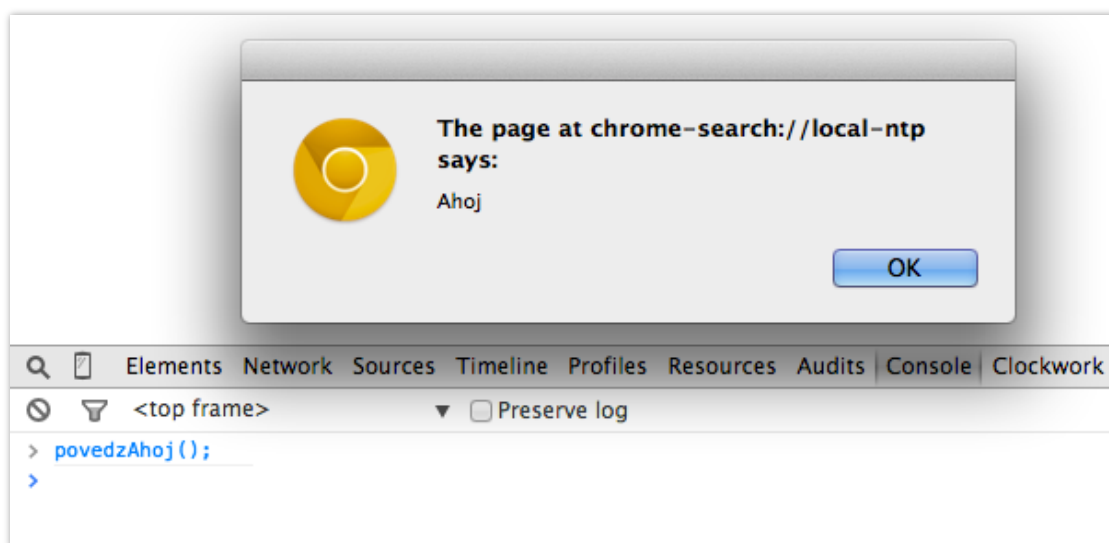


Napríklad **alert()** rieši problém, že na stránke nemáme dostatočný počet otravných, vyskakujúcich okien.

Môžeme však definovať aj funkcie vlastné:

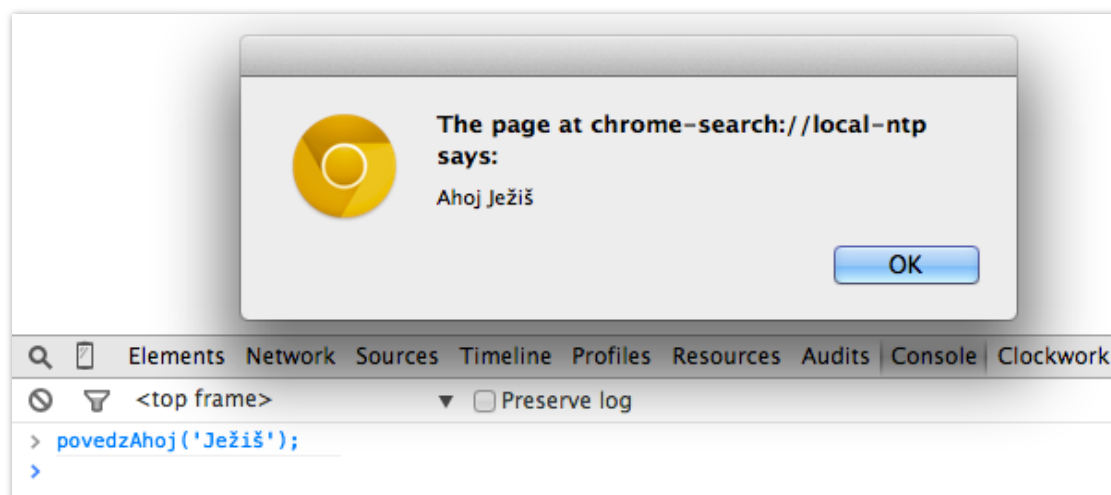
```
function povedzAhoj() {  
    alert('Ahoj');  
}
```

Začneme kľúčovým slovíčkom **function**, pokračujeme názvom a zátvorkami. Tie sú de facto symbol funkcií. Samotné telo funkcie (kód funkcie) je ohraničené zloženými zátvorkami. Funkciu zavoláme (spustíme) napísaním jej mena so zátvorkami.



Často potrebujeme do funkcie poslať informácie. Chceme funkciu „zavolať s argumentom“. Na to ju však musíme zadať tak, aby argumenty vedela prijímať:

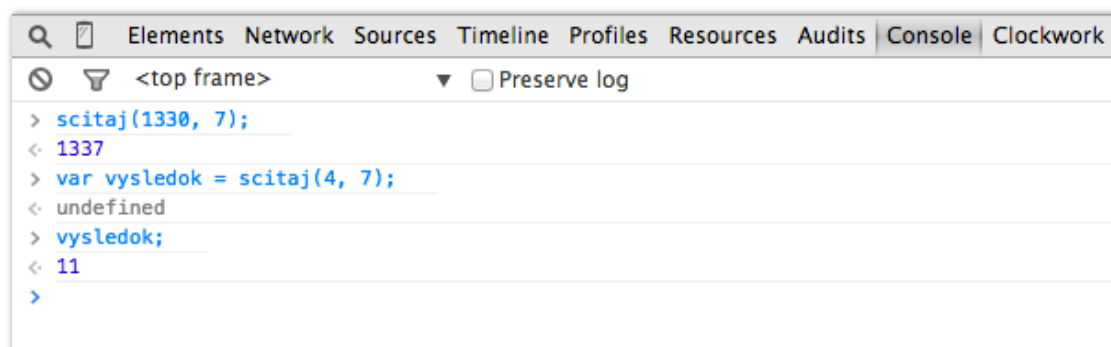
```
function povedzAhoj( meno ) {  
    alert('Ahoj' + meno);  
}
```



Často budeme chcieť, aby funkcia spracovala údaje a **vrátila** nám výslednú hodnotu. Vtedy použijeme kľúčové slovo **return**.

```
function scitaj( a, b ) {  
    return a + b;  
}
```

Výhoda je, že teraz si výsledok výpočtu vieme odchytiť do premennej a ďalej s ním pracovať v kóde:



V skutočnosti naše funkcie budú o niečo zložitejšie, ako náš príklad so sčítaním. Ak potrebujeme nejaký výpočet realizovať v programe viac krát, vyčleníme si kód za to

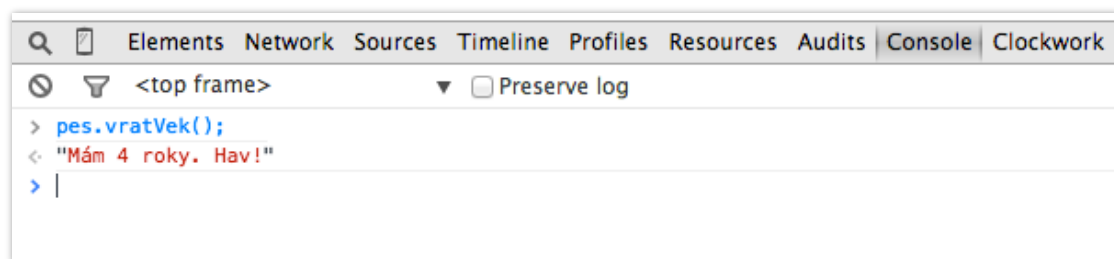
zodpovedný do funkcie a budeme sa vo zvyšku programu odvolávať na ňu. To zaručí, že ak budeme musieť v budúcnosti výpočetný vzorec zmeniť, bude to stačiť spraviť na jednom mieste – v tejto funkcii.

Metódy

Objekty môžu mať schopnosti. Tak, ako môžeme kľúč priradiť konkrétnu hodnotu, môžeme priradiť aj funkciu. Ak funkcia patrí objektu, hovoríme o **metóde**.

```
var pes = {  
  typ: 'Labrador',  
  vek: 4,  
  vratVek: function() {  
    return 'Mám ' + pes.vek + ' roky. Hav!';  
  }  
}
```

Ak metódu zavoláme:



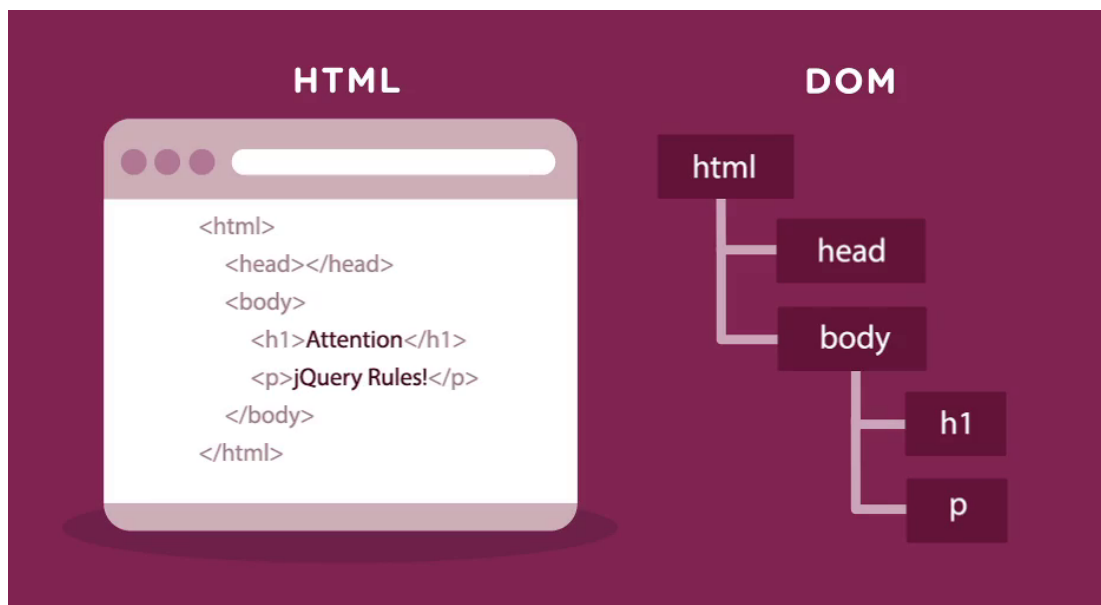
V jQuery budeme pracovať s objektami (alebo kolekciami objektov) a budeme na ne volať metódy.

```
<p>Toto je môj text.</p>  
  
<script>  
$('p').text();  
</script>
```

V tomto príklade si vyberieme element `p`, z ktorého jQuery pomocou volania `$('#p')` vytvorí jQuery objekt, ktorému pridá rôzne metódy. Napríklad metódu `.text()`, pomocou ktorej vieme získať text nachádzajúci sa medzi `<p></p>` tagmi.

Document Object Model (DOM)

DOM je reprezentácia HTML v stromovej štruktúre. Je to spôsob, akým si prehliadač interne interpretuje HTML súbory.



zdroj: teamtreehouse.com

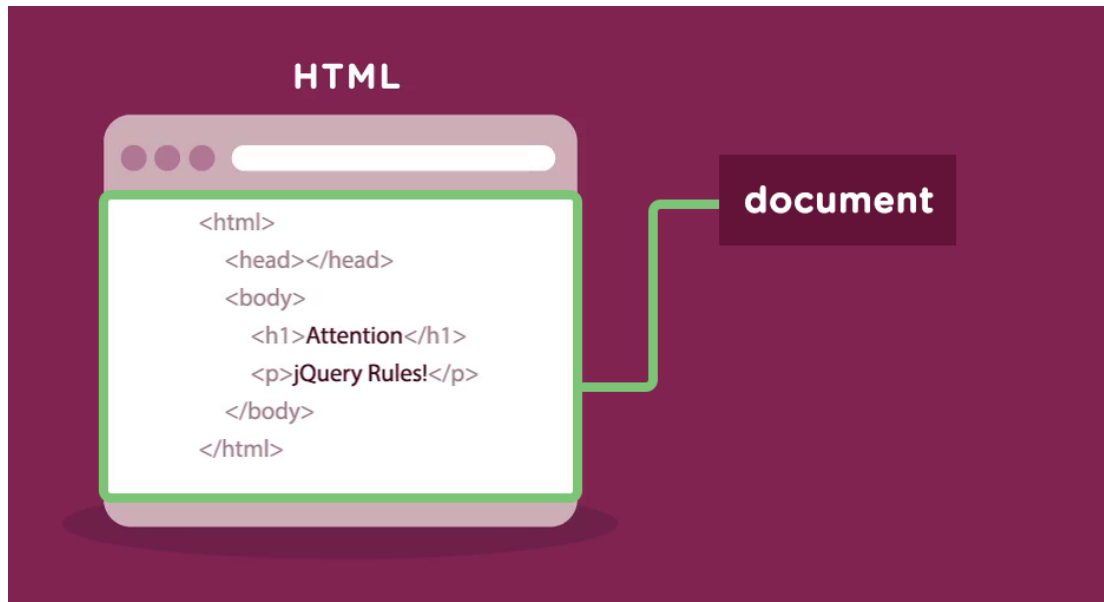
Document Object Model definuje elementy, ich atribúty a taktiež vzťahy medzi nimi. Napríklad:

- element **html** je *rodič* elementu **body**
- elementy **h1** a **p** sú *súrodenci*
- elementy **h1** a **p** sú *potomkovia* elementu **body**

Poskytuje tiež rozhranie, cez ktoré vieme k elementom a ich rôznym atribútom programaticky pristupovať. Pri práci s JavaScriptom (a teda aj jQuery) budeme manipulovať s elementami v DOM a taktiež bude vytvárať nové elementy a do DOM ich vkladať.

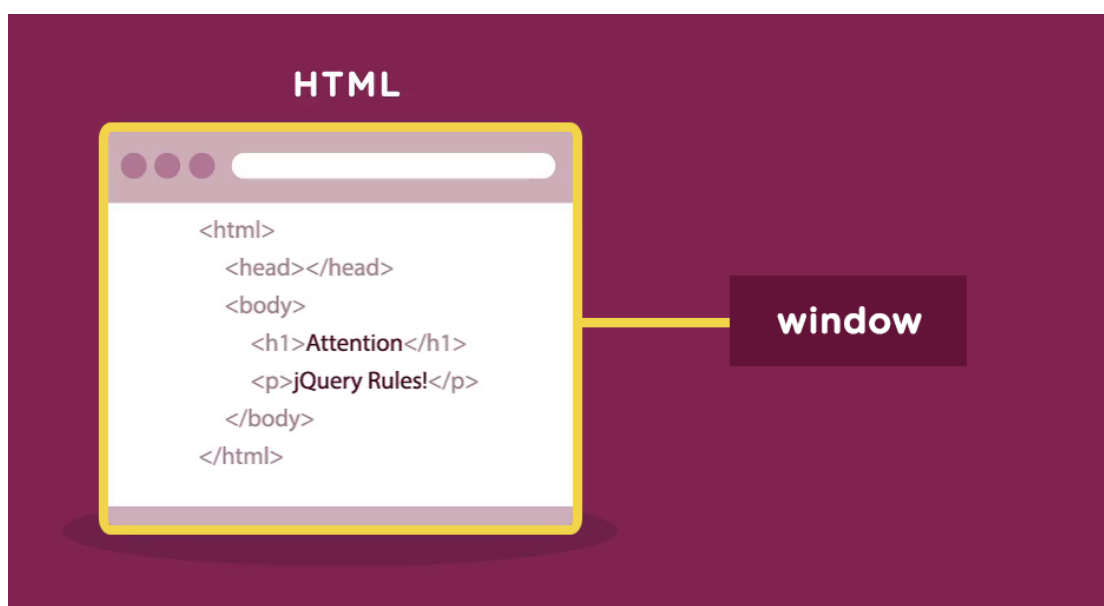
Document

Každá stránka má v prehliadači svoj vlastný object s menom **document**. Tento object nám dáva prístup k metódam, vďaka ktorým vieme manipulovať s DOM.



Window

Object **document** je vlastne atribút nadradeného objectu **window**. Ten odkazuje na každé okno alebo tab, v ktorom je otvorená stránka a dáva prístup k mnohým metódam a taktiež hodnotám, ako je napríklad URL adresa samotného okna.



jQuery

"jQuery je rýchla, malá a na funkcie bohatá knižnica jazyka JavaScript. Slúži na traverzovanie HTML dokumentom, manipuláciu s elementami, na spracovanie udalostí, na animácie a zjednodušuje použitie Ajaxu cez API rozhranie, ktoré funguje v celej rade prehliadačov. Vďaka kombinácii všestrannosti a rozšíriteľnosti, jQuery zmenil spôsob, akým milióny ľudí píšú JavaScript."

- jquery.com

jQuery je nadstavba jazyka JavaScript. Všetko, čo sa dá spraviť v jQuery sa dá spraviť v JavaScripte, pretože jQuery „je iba“ JavaScript. V jQuery to však ide jednoduchšie, pretože ponúka množstvo predpripravených funkcií a efektov a na rozdiel od JavaScriptu sa vo všetkých prehliadačoch správa rovnako.

Stiahnutie jQuery a vloženie do stránky

Pred používaním je potrebné súbor s jQuery skriptom stiahnuť a vložiť do stránky, pomocou HTML `<script>` tagu. Buď do `<head>` tagu:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>jQuery mi šmakuje</title>
  <meta name="viewport" content="width=device-width">
  <link rel="stylesheet" href="css/home.css">
  <script src="js/jquery.js"></script>
</head>
```

Alebo na koniec stránky, pred ukončovacím `</body>` tagom:

```
...
  <script src="js/jquery.js"></script>
</body>
</html>
```

V dobe písania existujú 2 rôzne verzie jQuery – jQuery **1.x** a jQuery **2.x**. Obe robia to isté, ale **1.x** podporuje Internet Explorer 6, 7 a 8. Verzia **2.x** podporuje už iba tzv. „moderné prehliadače“.

Stiahnuť sa dá **compressed** a **uncompressed** súbor. **Compressed** zaberá menej a mala by sa používať pri spustení stránky do produkcie.

Content Delivery Network (CDN)

Druhá možnosť, okrem stiahnutia samotného súboru, je použitie tzv. **Content Delivery Networks (CDN)**.

Do **<script>** tagu vložíme URL adresu smerujúcu na jQuery súbor hostovaný jedným z CDN poskytovateľov:

CDN Hosting	URL
jQuery	http://code.jquery.com
Google	https://developers.google.com/speed/libraries/devguide?csw=1#jquery
Microsoft	http://asp.net/ajaxlibrary/cdn.ashx#jQuery_Releases_on_the_CDN_0
CDNJS	http://cdnjs.com/libraries/jquery

Výhoda využitia CDN je, že ak návštevník už bol na stránke s CDN hostovaným jQuery, bude tento skript uložený v cache prehliadača a nebude ho nutné sťahovať znova. Tým pádom sa stránka zobrazí rýchlejšie.

Nevýhoda je, že ak v dobe vyvíjania stránky pridáte o prístup k Internetu, prestane jQuery fungovať.

V jednoduchosti sa dá povedať, že pri tvorbe stránky sa oplatí mať jQuery súbor stiahnutý a vložený lokálne, pri spustení stránky sa oplatí využiť služby CDN.

Po vložení jQuery do stránky, ho môžeme začať používať:

```
<script src="js/jquery.js"></script>
<script>
    jQuery('.social-icon').hide();
</script>
```

Toto volanie vyberie všetky HTML elementy s **class="social-icon"** a skryje ich. Podobne, ako keby sme v CSS napísali:

```
.social-icon { display: none; }
```

Document ready

Skripty sa odporúča pridávať na koniec **<body>** elementu. Dôvod je ten, že hlavne v starších prehliadačoch, skripty blokujú sťahovanie. Ak prehliadač pri sťahovaní obsahu stránky narazí na **<script>** tag, musí zastaviť všetko ostatné sťahovanie a venovať celú svoju pozornosť JavaScriptu. Kým iné elementy, napríklad obrázky, dokáže spracovať asynchrónne a súbežne, u JavaScriptu to nie vždy platí. JavaScript je ako Hummer, ktorého všetky Smarty a Priusy nemôžu predbehnúť. Ak ho však dáme na koniec stránky, neblokuje cestu slušným ľuďom.

V prípade, že náš skript **nie je** umiestnený na konci stránky, musíme počkať, kým je DOM pripravený na manipuláciu. Povedzme, že chceme skryť všetky **** elementy na stránke.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>jQuery mi šmakuje</title>
  <meta name="viewport" content="width=device-width">
  <link rel="stylesheet" href="css/home.css">
  <script src="js/jquery.js"></script>
  <script>
    jQuery('img').hide();
  </script>
</head>
<body>
```

Tento kód by nefungoval, pretože v momente, keď prehliadač pri sťahovaní stránky narazí na náš kód, vypýta si všetky **** na stránke. Žiadne však na stránke zatiaľ nie sú. Tie sa nachádzajú až v tele **<body>** elementu a to ešte prehliadač nestiahol. Potrebujeme spôsob, akým zistíme, či je document pripravený na manipuláciu.

```
<script>
jQuery(document).ready(function() {
  jQuery('img').hide();
});
</script>
```

Tento zápis sa doslovene dá preložiť nasledovne: „ak je document pripravený, spusti túto funkciu.“ V tele samotnej funkcie je potom náš kód, ktorým skryjeme obrázky.

Ak máme na stránke iba jQuery, ak k jQuery nepridáme žiadnu „konkurenčnú“ knižnicu, môžeme namiesto zápisu **jQuery()** používať skratku **\$()**.

```
$(document).ready(function() {
  $('img').hide();
});
```

Často sa tiež môžeme stretnúť so skráteným zápisom „document ready“ volania:

```
<script>
$(function() {
    $('img').hide();
})
</script>
```

V prípade, že by sme však kód vložili na koniec stránky – pred uzatvárací **</body>** tag, úplne by stačil nasledovný zápis:

```
    <script>
        $('img').hide();
    </script>

</body>
</html>
```

Odporúča sa však obaliť všetok náš kód do samostatnej funkcie, napríklad takto:

```
    <script>
    (function($) {
        $('img').hide();
    })(jQuery);
    </script>

</body>
</html>
```

Týmto zabezpečíme, že všetky premenné, ktoré v kóde vytvoríme, budú izolované od takzvaného „globálneho prostredia“. V rámci jedného projektu budeme často využívať množstvo rôznych skriptov od rôznych programátorov a z rôznych prostredí. Ak by títo programátori neizolovali svoje premenné a názvy týchto premenných by sa zhodovali s našimi názvami, mohli byť nastať problémy. Hodnoty by sa mohli poprepisovať, skripty by mohli prestať fungovať. Týmto zápisom vytvoríme bezpečný obal pre všetok náš kód.

Pre začiatok si stačí zapamätať, že náš jQuery kód bude obalený buď anonymnou funkciou (ak je na konci stránky) alebo document ready funkciou (ak je na jej začiatku).

Volanie jQuery metód s argumentami a reťazenie

```
<script>
(function($) {
    $('img').fadeOut(2000);
})(jQuery);
</script>

</body>
</html>
```

Tento kód by všetky obrázky na stránke nechal postupne zmiznúť. Celá animácia by trvala 2 sekundy. Použili sme na to metódu **fadeOut()** s argumentom s hodnotou **2000**. jQuery funkcie môžu brať argumenty, pri tých animačných jeden z argumentov býva **trvanie** – doba, po ktorú má animácia prebiehať, ktorá sa zadáva v milisekundách. **1000 milisekúnd** sa rovná **1 sekunda**. Naše obrázky budú miznúť dve sekundy.

jQuery používa CSS selektory.

```
<script>
(function($) {
    $('img').fadeOut(2000); // nechaj obrázky zmiznúť za 2 sekundy
    $('.icon').hide(); // skry elementy s class="icon"
    $('#content').show(); // zobraz elementy s id="content"
})(jQuery);
</script>
```

Tieto, a taktiež všetky ostatné zápisy, ktoré poznáte z CSS, fungujú aj v jQuery.

Volania jQuery metód môžeme **reťaziť**. Namiesto ukončovacej bodkočiarky stačí na kolekciu vybraných elementov okamžite zavolať ďalšiu metódu.

```
<script>
(function($) {
    $('img').hide().fadeIn(2000);
})(jQuery);
</script>
```

Tento kód najskôr skryje všetky obrázky, potom ich nechá postupne zobrazíť po dobu 2 sekúnd. Zreťazené metódy nemusia byť v jednom riadku:

```
(function($) {
    $('img')
        .hide()
        .fadeIn(2000);
})(jQuery);
```

Reťazenie a odsadenie do viacerých riadkov môže pomôcť prehľadnosti. Reťaziť môžeme toľko volaní, koľko nám morálka dovolí. Reťazenie ukončíme bodkočiarkou.

```
<script>
(function($) {

    $('.icon').css({
        background: 'yellow'
    });

})(jQuery);
</script>
```

V tomto prípade vyberáme všetky HTML elementy s **class="icon"** a pomocou metódy **.css()** meníme ich pozadie na žltú farbu. Tento zápis znova ukazuje podobnosť jQuery a CSS volaní.

Ako argument sme do metódy vložili nie číslo, ako v prípade animácií, ale **object**. Pre zreteľnosť by sme mohli tento kód zapísať nasledovne:

```
<script>
(function($) {

    var options = {
        background: 'yellow'
    };

    $('.icon').css( options );

})(jQuery);
</script>
```

Pozn.: Všetky metódy a argumenty, ktoré im môžeme poskytnúť, nájdeme na adrese <http://api.jquery.com/>.

Ak do metódy **.css()** pošleme **string** s názvom CSS atribútu, dostaneme späť hodnotu daného atribútu. Tú si môžeme uložiť do premennej alebo napríklad pomocou **console.log()** metódy nechať vypísať do DevTools konzole:

```
<script>
(function($) {

    console.log( $('.icon').css('background') );

})(jQuery);
</script>
```

Do konzole bude vypísaná aktuálna hodnota CSS background atribútu. Treba si dať pozor na rozdielnú notáciu v jazykoch JavaScript a CSS. CSS oddeľuje slová pomlčkou, JavaScript používa CamelCase notáciu:

```
<script>
(function($) {

    // CSS:  background-color
    // JS:   backgroundColor

    $('.social-icon').css({
        backgroundColor: '#bada55'
    });

})(jQuery);
</script>
```

Mnohé metódy v jQuery buď nastavujú novú hodnotu alebo vracajú aktuálnu hodnotu podľa toho, aký argument im poskytnete. Náš prvý príklad volania metódy **.css()** fungoval ako **setter**, kde sme **nastavili** pozadie na žltú farbu. Druhý príklad fungoval ako **getter**, ktorým sme **získali** aktuálnu hodnotu pozadia pre vybraný element.

Vyčlenenie kódu do vlastného súboru

V príkladoch sme písali náš jQuery kód do **<script></script>** tagov. Kód by sme však mohli (a vo väčšine prípadov aj **mali**) vyčleniť do samostatného súboru. Podobne, ako do stránky vkladáme jQuery, môžeme do nej vložiť aj nový súbor s názvom napríklad **script.js**. Tento vložíme do stránky **po** vložení jQuery. Na poradí skriptov na stránke záleží.

```
<script src="js/jquery.js"></script>
<script src="js/script.js"></script>

</body>
</html>
```

Doň vložíme náš kód, už samozrejme bez **<script></script>** tagov.

```
(function($) {

    $('.social-icon').css({
        backgroundColor: '#bada55'
    });

})(jQuery);
```


Jednoduché animácie v jQuery

Animácie v jQuery často znamenajú postupnú zmenu hodnoty CSS atribútu alebo atribútov. Postupom času meníme CSS hodnotu.

Medzi jednoduché animačné metódy patria spomínané **.fadeIn()** a **.fadeOut()** ktoré animujú CSS atribút **opacity** po istú dobu, to uplnutí ktorej – teda po úplnom odhalení alebo úplnom zahalení elementu – zmenia hodnotu CSS atribútu **display**.

V jQuery teda vieme použiť už predpripravené animácie. Vieme si však vytvoriť aj animácie vlastné.

Metóda .animate()

```
$('#cover').animate({ width: 600 }, 1000);
```

Ak máme element s **id="cover"** a tento element má **width: 400px**; naša **.animate()** metóda natiahne element na 600px po dobu jednej sekundy.

Ak animačným metódam neposkytneme trvanie, jQuery použije predvolenú hodnotu 400ms.

Práca s jQuery často vyžaduje balansovanie HTML a CSS kódu, je to súhra technológií. Ak v jQuery chceme animovať pozíciu elementu, vieme, že element musí mať v CSS nastavený **position** iný ako **static**. Tiež budeme chcieť, aby sa tento element zobrazoval nad ostatnými, aby ho tieto neprekryli, preto nastavíme aj hodnotu **z-index**.

```
.icon {  
    position: relative;  
    top: 0;  
    z-index: 2;  
}
```

Vieme, že tieto hodnoty môžeme nastaviť pomocou jQuery metódy **.css()**. Ak však máme prístup k samotným CSS súborom, je lepšie upraviť priamo tie. Každá animácia s DOM stojí čas a námahu. Čím menej dotykov, tým lepšie. V tomto prípade. Nie v každej situácii v skutočnom svete to platí. Nie je to univerzálne pravidlo.

Z tohoto pramení ďalšia výhoda reťazenia volaní, ktoré sme spomínali vyššie. Pri reťazení všetky zreťazené metódy pracujú s na začiatku vybranou kolekciou elementov. Túto kolekciu sme teda vybrali len raz, čo znova šetrí výpočtový čas a zrýchľuje chod nášho programu.

Ak máme hodnoty nastavené, môžeme animovať:

```
$('.icon').animate({ top: 1000 }, 2000);
```

Hodnota **top** sa pre vybrané elementy mení zo začiatočných **0px** na nových **1000px** po dobu **2 sekúnd**.

Pozn.: Nie všetky naše príklady budú mať praktické využitie na stránkach. Pomocou niektorých si ukážeme koncepty a metódy, ktoré budeme pri programovaní v jQuery používať.

Povedzme, že chceme, aby elementy s triedou **icon** “vypadli von” zo stránky. Chceme animovať hodnotu **top** na číslo vyššie, ako je výška samotnej stránky. Číslo 1000 z nášeho príkladu nestačí, pretože stránka môže byť dlhšia ako 1000px a nechceme, aby elementy zostali zaseknuté uprostred.

```
(function($) {  
  
    // do premennej icons si uložíme elementy s class="icon"  
    var icons = $('.icon');  
  
    // do siteHeight metódou .height() zistíme výšku stránky  
    var siteHeight = $('body').height();  
  
    // teraz môžeme posunúť elementy o výšku celej stránky  
    icons.animate({ top: siteHeight }, 2000);  
  
})(jQuery);
```

Callback funkcie

Metódy, ktoré sa vykonávajú postupne po istú dobu, často akceptujú argument vo forme **callback funkcie**. Toto je funkcia, ktorá sa spustí okamžite po dokončení akcie.

```
var icons = $('.icon'),  
    siteHeight = $('body').height();  
  
icons.animate({ top: siteHeight }, 2000, function() {  
    $(this).fadeOut();  
});
```

Povedzme, že mám na stránke 4 elementy s triedou **icon**. Po skončení animovania top pozície sa **pre každý z nich** spustí callback funkcia, ktorá daný element nechá postupne zmiznúť.

Keď skončí animate, spustí sa callback funkcia, ktorý spraví fadeOut.

Kľúčové slovíčko **this**

Slovíčko **this** má v Javascripte v rôznych kontextoch rôzne významy. My sa najčastejšie budeme s **this** stretávať v kontexte callback funkcie nejakej jQuery metódy.

Ak je funkcia zavolaná ako metóda objektu, **this** bude odkazovať na konkrétny objekt, na ktorý bola metóda zavolaná.

Vo hore uvedenom príklade bude **this** odkazovať na ten element, ktorý práve teraz dokončil svoju animáciu. V príklade vidíme, že **this** je obalené do jQuery volania **\$(this)**. To preto, aby sme na daný element vedeli volať jQuery metódy, ako v tomto prípade **.fadeOut()**.

Nasledujúca kapitola bližšie objasní, akým spôsobom budeme používať **this** v našom jQuery kóde.

Udalosti (eventy)

V predošlom príklade sa elementy s triedou **icon** začnú animovať okamžite po tom, čo sa stránka zobrazí. Ak by sme chceli aktivovať animáciu na daný element po tom, čo naň používateľ klikne myšou, využili by sme tzv. „udalosti“ – **eventy**.

```
<script>
$('.icon').on('click', function() {
    $(this).animate({ top: siteHeight }, 2000);
});
</script>
```

Metóda **.on()** naviaže na elementy s triedou **icon** funkciu, ktorá sa spustí, keď nastane **click event** – udalosť kliknutia na element.

Ak by sme chceli spustiť funkciu po nastaní inej udalosti, napríklad ak používateľ prejde na element myšou, stačí zmeniť prvý argument metódy **.on()**.

```
<script>
$('.icon').on('mouseenter', function() {
    $(this).animate({ top: siteHeight }, 2000);
});
</script>
```

Pozn.: zoznam všetkých eventov sa dá nájsť tu: <https://developer.mozilla.org/en-US/docs/Web/Events>. Výber eventov, s ktorými často pracujeme v jQuery, nájdeme napríklad tu: <http://www.sitepoint.com/jquery-list-events-bind/>.

Event Listener

Event Listener je „ploštica“, ktorú nalepíme na element a ktorá počúva, či na danom elemente nenastala udalosť. V našom príklade sa **na každý** element, ktorý v dobe volania metódy **.on()** **existuje na stránke**, pripne event listener, ktorý počúva na kliknutie ľavého tlačítka na myši (click event). Ak nastane tento event, spustí sa Event Handler.

Event Handler

Event Handler je funkcia, ktorá sa má spustiť, keď nastane daná udalosť. V našom príklade je handler funkcia, v ktorej tele sa odohráva samotná animácia:

```
$('.icon').on('click', function() {
    // táto funkcia je event handler
    $(this).animate({ top: siteHeight }, 2000);
});
```

Event Object

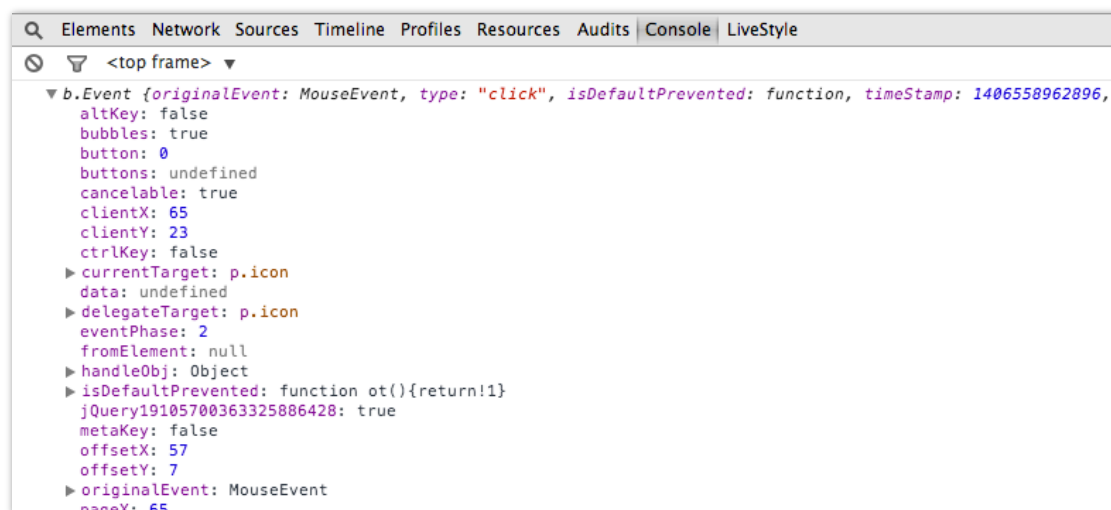
Event object je objekt, ktorý obsahuje dodatočné informácie o danom evente. Ak sa jedná napríklad o **keyDown** event (nastane, keď používateľ zatlačí klávesu na klávesnici) vieme cez event object zistiť, ktorá klávesa bola zatlačená.

Ak sa jedná o kliknutie myšou, vieme zistiť X a Y koordináty myši, kde sa nachádzala v dobe kliknutia. To sú len niektoré z príkladov.

Event object je do každého event handlera preposlaný ako argument. Ak rozšírime náš handler tak, aby dokázal prijať argument, dostaneme prístup k event objectu.

```
$('.icon').on('click', function( event ) {  
    console.log( event ); // nechajme si do konzole vypísať obsah event objectu  
    $(this).animate({ top: siteHeight }, 2000);  
});
```

Ak si pomocou `console.log(event)` necháme do konzole vypísať event object, dostaneme sa ku všetkým dodatočným informáciám, ktoré môžeme potrebovať.



Event object ponúka aj prístup k niekoľkým metódam, najčastejšie budeme používať jeho metódu **.preventDefault()** a to najmä pri kliknutí na odkaz – `<a>` element.

```
<script>  
$('.a').on('click', function( event ) {  
    event.preventDefault();  
});  
</script>
```

Táto metóda zabráni predvolenej akcii po nastaní udalosti. Po kliknutí na odkaz, predvolená akcie je otvoriť stránku, kam odkaz smeruje, v prehliadači. My často

budeme chcieť tejto udalosti zamedziť a namiesto toho spracovať túto adresu pomocou jQuery.

Ako príklad môžeme uviesť **image lightbox** – po kliknutí na link smerujúci na obrázok, sa tento otvorí v okne v strede našej stránky. Predvolená akcia by bola poslať prehliadač na adresu s obrázkom, my však vďaka **event.preventDefault()** zostaneme na našej stránke a pomocou jQuery zobrazíme obrázok v strede stránky.

Ak by sme **event.preventDefault()** zabudli, obrázok by sa síce v strede stránky zobrazil, ale zrejme by sme to ani nevideli, pretože prehliadač by okamžite skočil na adresu obrázku.

Traverzovanie DOMu

Traverzovaním myslíme prechádzanie sa po stromovej štruktúre Document Object Modelu, za účelom nájdenia elementov, ktoré spĺňajú určité podmienky. Často potrebujeme nájsť element a následne všetky jeho dcérske elementy. Môžeme tiež chcieť nájsť element, potom nájsť element, ktorý nasleduje priamo po ňom a ten skryť.

Pre príklad budeme pracovať s nasledovným **description listom**.

```
<dl class="jokes">
  <dt>I've spent the past two years looking for my ex-girlfriend's killer...</dt>
  <dd>But no one will do it.<small class="author">- Anthony Jeselnik</small></dd>

  <dt>Yesterday I accidentally hit a little kid with my car.</dt>
  <dd>It wasn't serious - nobody saw me. <small class="author">- Anthony Jeselnik</small>
</dd>

  <dt>I once went on a date with a girl where we went hiking...</dt>
  <dd>...and she gets bit by a snake in between her toes, and I had to suck out the poison...
  So she's dead.<small class="author">- Anthony Jeselnik</small>
</dd>

  <dt>When I finished high school I wanted to buy myself a motorcycle.</dt>
  <dd>Buy my mom said no. See, she had a brother who died in a horrible motorcycle
  accident when he was 18. And I could have his motorcycle. <small class="author">
  - Anthony Jeselnik</small>
</dd>

  <dt>Whenever I meet a pretty girl, the first thing I look for is intelligence</dt>
  <dd>Because if she doesn't have that, then she's mine. <small class="author">- Anthony
  Jeselnik</small>
</dd>
</dl>
```

Pre traverzovanie využívame fakt, že DOM je stromová štruktúra elementov, v ktorej elementy medzi sebou existujú v určitom vzťahu. Napríklad:

- element **dl** je rodič elementov **dt** a **dd**.
- naopak elementy **dt** a **dd** sú potomkovia elementu **dl**.
- **dt** a **dd** sú v štruktúre vedľa seba, sú to súrodenci.

jQuery ponúka metódy, ktoré vedia z tejto štruktúry ťažiť. Povedzme, že chceme skryť **dd** elementy **spadajúce pod zoznam s triedou jokes** a zobrazíť ich po kliknutí na predošlého **dt** súrodenca.

```
<script>
// ulozime si zoznam vtipov do premennej
var list = $('.jokes');
```

```
// najdeme všetky dd elementy spadajúce pod zoznam
list.find('dd').hide();

// zobrazíme dd po kliknutí na jeho predchádzajúci dt element
list.find('dt').on('click', function() {
    $(this).next().slideDown();
});
</script>
```

Pomocou metódy **.find()** nájdeme iba tie dd elementy, ktoré spadajú pod náš uložený zoznam s triedou **jokes**. Nezabudnime, že naša stránka môže obsahovať nespočet **dl** zoznamov. My nechceme skryť všetky **dd** elementy, ktoré sa nachádzajú na stránke. Chceme skryť špecificky iba tie, ktoré sa nachádzajú v zozname **<dl class="jokes"></dl>**.

Po kliknutí na **dt** element spadajúci pod náš zoznam máme v **\$(this)** odkaz na konkrétny dt element, na ktorý sme klikli. Pomocou metódy **.next()** nájdeme nasledovný element (náš dd) a zobrazíme ho pomocou animačnej metódy **.slideDown()**. Správa sa podobne ako nám známa **.fadeIn()**, len element sa vyroluje namiesto postupného zjavenia sa.

Metódy v jQuery sú často dopĺňané ich opakmi – ak existuje **.next()**, existuje aj **.prev()**, ktorá nájde predošlý element. Ak existuje **.slideDown()**, existuje aj **.slideUp()**, ktorá element zroluje. Existujú aj tzv. prepínače stavov – toggle metódy. Napríklad **.slideToggle()** element zroluje ak je vyrolovaný a naopak.

jQuery ponúka mnoho metód, ktoré uľahčujú traverzovanie:

```
<script>
var list = $('.jokes');

// najde všetkých potomkov zoznamu
list.children();

// najde iba potomkov typu dt
list.children('dt');

// iba potomkov typu dd
list.children('dd');

// najde všetkých potomkov a z nich vyberie prveho
list.children().first();

// css pseudo selektor, taktiez najde prveho potomka
list.children(':first-child');

// najde všetkých surodencov prveho potomka
list.children(':first-child').siblings();

// to iste, avsak vyberie iba surodencov typu dt
list.children(':first-child').siblings('dt');
```



```

// najde rodica prveho potomka
list.children(':first-child').parent();

// najde vsetkych rodicov prveho potomka
list.children(':first-child').parents();
// najde iba rodicov type section prveho potomka
list.children(':first-child').parents('section');

// najde tretieho potomka
list.children(':nth-child(3)');

// najde vsetky dt, ktore nasleduju po tretom potomkovi
list.children(':nth-child(3)').nextAll('dt');

// najde element, ktory nasleduje po tretom
list.children(':nth-child(3)').next();

// najde element, ktory predchadza tretiemu
list.children(':nth-child(3)').prev();

// najde potomka s idexom 2 (tretieho)
list.children().eq(2);
</script>

```

Mnohé ďalšie nájdeme na adrese: <http://api.jquery.com/category/traversing/>

Na základe vyššie uvedeného, vieme spraviť jednoduchý „akordeónový“ efekt. Ak klikneme a **dt** element, nasledovný **dd** sa zobrazí a všetky ostatné **dd** sa skryjú. Vždy bude zobrazený iba jeden.

```

(function($) {

    var list = $('.jokes');

    // zrolujeme vsetky dd elementy spadajuce pod .jokes
    list.find('dd').slideUp();

    // po kliknutí na dt..
    list.find('dt').on('click', function(event)
    {
        // najdeme nasledovny dd element, ktory vyrolujeme / zrolujeme
        // potom najdeme vsetkych ostatnych dd surodencov, a tych skryjeme
        // vzdy teda zostane vyrolovany iba jeden dd, ostatne sa zroluju
        $(this).next().slideToggle().siblings('dd').slideUp();

        // chceme zabranit moznej predvolenej akcii
        event.preventDefault();
    });
})(jQuery);

```

Manipulácia DOMu

Doteraz sme pracovali s elementami, ktoré existovali na našej stránke, ale výrazne sme do nich nezasahovali. Teraz si ukážeme, ako môžeme elementy na stránke meniť, upravovať a mazať a ako vieme pridávať nové elementy.

```
<p>Toto je môj text.</p>

<script>
// vyberieme do premennej textový obsah elementu
var text = $('p').text();

// vyberieme do premennej html obsah elementu
// na rozdiel od .text() zachová aj html elementy v jeho vnútri
var html = $('p').html();

// zmení text zoznamu
$('p').text('Som sa zmenil');

// nahradí text v elemente odkazom
$('p').html('<a href="http://google.com">hľadám</a>');
</script>
```

jQuery ponúka aj mnoho metód na pridanie nových elementov.

```
<p>Toto je môj text.</p>

<script>
// vytvoríme nový <a> element
var novy = $('<a href="http://google.com">hľadám</a>');

// pridáme ho na koniec p elementu
$('p').append( novy );

// vložíme ho na začiatok elementu
$('p').prepend( novy );

/* môžeme na to ísť aj s opačnej strany */
novy.appendTo('p');
novy.prependTo('p');

// pridáme nový element PRED element p
$('p').before( novy );

// pridáme nový element ZA element p
$('p').after( novy );

/* môžeme na to ísť aj s opačnej strany */
novy.insertBefore('p');
novy.insertAfter('p');
</script>
```

Lepší spôsob, ako vytvoriť nový element, je poslať object s nastaveniami nového elementu ako druhý argument:

```
<script>
var novy = $('<a/>', {
    href: 'http://google.com',
    text: 'hladam'
});
</script>
```

Elementy vieme aj naklonovať:

```
<script>
// naklonujeme element p a pridame na koniec stranky
$('p').clone().appendTo('body');

// teraz zachovame aj event handlers
$('p').clone(true).appendTo('body');
</script>
```

Ak by sme mali na element naviazané event handleri, v prvom prípade by tieto neboli naklonované. Ak metódu **.clone()** zavoláme s argumentom **true**, skopírujú sa aj tie na nový element.

Vytvoríme si zoznam **ul** a na každý **li** element zoznamu naviažme funkciu, ktorá po kliknutí vytvorí ďalší li element a pridá ho do zoznamu.

```
<ul class="list">
    <li>ja som malý zoznamček</li>
    <li>zoznámim sa rád</li>
    <li>ja som malý zoznamček</li>
    <li>bud' môj kamarát</li>
</ul>

<script>
var ul = $('ul.list');

// po kliknutí na každý li element, pridame do zoznamu ďalší li element
ul.find('li').on('click', function() {
    ul.append('<li>JA SOM TU NOVY</li>');
});
</script>
```

Každé kliknutie na **li** element vytvorí a do zoznamu vloží nový element. Všimneme si však, že po kliknutí na novo vytvorený li element, sa nič nestane. Event handleri fungujú iba na elementy, ktoré existovali pri zobrazení stránky. Na dynamicky, javascriptom pridané elementy, naša funkcia nefunguje.

V ďalšej kapitole si povieme prečo to tak je a ako to vyriešiť.

Bublanie a delegácia

Povedali sme si, že listeneri fungujú ako také ploštice, ktoré načúvajú, či nenastal event. Tieto ploštice sú rozdávané po načítaní stránky. V príklade z minulej kapitoly, sa ploštice nalepia na každý li element, **ktorý existoval** v dobe načítania stránky. Dynamicky pridávané elementy vtedy neexistovali a tak na ne nie sú naviazané žiadne listeneri.

Riešením je naviazať listenera na rodičovský element a dať metóde **.on()** najavo, ktorý element chceme sledovať na odpálenie eventu.

```
<script>
var ul = $('.list');

// po kliknutí na každý li element, pridáme do zoznamu ďalší li element
ul.on('click', 'li', function() {
    ul.append('<li>JA SOM TU NOVY</li>');
});
</script>
```

V predošlom príklade mala **.on()** metódy dva argumenty, teraz zadávame argumenty tri. Metódu voláme na rodičovský element a ako druhý argument definujeme element, ktorý chceme sledovať – li.

Zdelegovali sme robotu na rodiča. Toto funguje vďaka tzv. „**bublaniu**“ eventov.

Ak klikneme na li element, tento element spadá pod ul element, takže v praxi sme klikli aj naň. Tento ul element zasa spadá pod body element, takže sme klikli aj naň. A takto to pokračuje (bublá) až k samotnému **document**-u. Listenera by sme mohli naviazať aj naň:

```
<script>
$(document).on('click', 'ul li', function() {
    ul.append('<li>JA SOM TU NOVY</li>');
});
</script>
```

Vo väčšine prípadov však bude stačiť naviazanie na jeden z rodičovských elementov.

Toto je dôležitá téma, keď príde na prácu s AJAXom. Ak máme galériu obrázkov, ktoré chceme zobrazovať v lightboxe, naviažeme listenera na každý element galérie. Ak je však táto galéria rozdelená na 4 podstránky, ktoré mi chceme AJAXom na pozadí načítať a vložiť do stránky, prestane lightbox na novo vložené elementy fungovať.

Problém vyriešime, ak listenera pripneme na rodičovský element, ktorý drží obrázky galérie namiesto samotných obrázkov.

Občas potrebujeme zastaviť bublanie. Chceme vykonať našu akciu po udalosti, ale nechceme, aby element o tejto udalosti „povedal rodičovi“.

```
<script>
$('a').on('click', function(event) {
    event.stopPropagation();
});
</script>
```

Občas tiež chceme vykonať našu akciu, ale chceme okamžite zastaviť vykonávanie ďalších akcií naviazaných na udalosť pre tento istý element.

```
<script>
$('a').on('click', function(event) {
    event.stopImmediatePropagation();
});
</script>
```

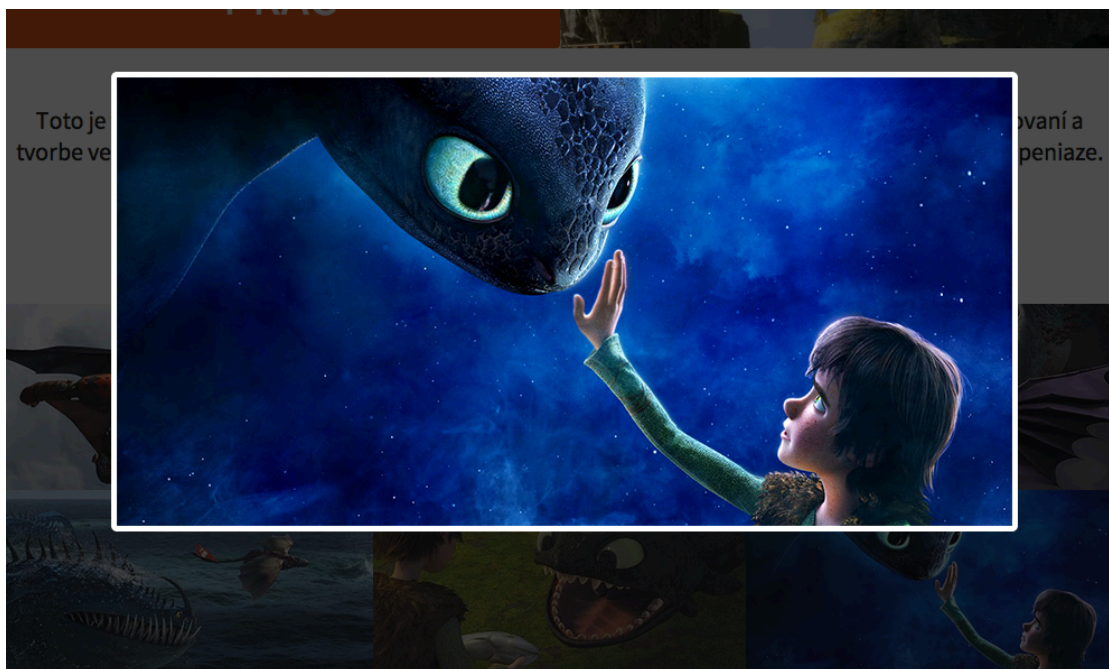
V tomto prípade element **a** nič nepovie rodičovi, ale taktiež všetky ostatné handleri naviazané na element **a** a počúvajúce na klik, budú odignorované.

Tvorba jednoduchého image lightboxu

Majme galériu, v ktorej odkazy obsahujúce malý obrázok, smerujú na veľkú verziu toho istého obrázka.

```
<div class="gallery">
  <a href="img/image-1.jpg">
    
  </a>
  <a href="img/image-2.jpg">
    
  </a>
  <a href="img/image-3.jpg">
    
  </a>
  <a href="img/image-4.jpg">
    
  </a>
  <a href="img/image-5.jpg">
    
  </a>
  <a href="img/image-6.jpg">
    
  </a>
</div>
```

Po kliknutí nechceme byť presmerovaní na adresu v **href** atribúte odkazu. Namiesto toho chceme vytvoriť nový obrázok, ktorého **src** atribút bude nastavený na túto adresu a ktorý sa bude nachádzať v strede obrazovky, **nad** ostatným obsahom stránky. Tento nový obrázok umiestnime do element **<div id="overlay"></div>**, ktorý bude polopriehľadný a bude prekryvať obsah stránky.



Chceme:

- po kliknutí na obrázok v galérii vytvoriť overlay element.
- do overlay elementu umiestniť obrázok v plnej verzii, na stred.
- po kliknutí na overlay nechať overlay zmiznúť.

```
// vytvoríme si overlay, pridáme ho na koniec stránky a necháme ho skryt
var overlay = $('<div>', { id: 'overlay' });
overlay.appendTo('body').hide();

// po kliknutí na overlay ho skryjeme
overlay.on('click', function() {
    $(this).fadeOut('fast');
});
```

V CSS pridáme overlay-u štýl:

```
#overlay {
    background: rgba(0,0,0,.7);
    width: 100%;
    height: 100%;
    position: fixed;
    top: 0;
    left: 0;
    z-index: 20;
    text-align: center;
}

#overlay img {
    margin-top: 10%;
    border: 5px solid white;
    border-radius: 5px;
}
```

Overlay je pridaný na stránke. Teraz potrebujeme zabezpečiť, aby po kliknutí na odkaz bol vytvorený **img** element, ktorého **src** hodnota sa nastaví na **href** hodnotu kliknutého linku. Následne vložíme nový **img** do overlay a ten zobrazíme.

```
// po kliknutí zobrazíme lightbox
$('.gallery').on('click', 'a', function(event) {
    // vytiahneme href hodnotu, vytvoríme nový img a nastavíme src hodnotu
    var href = $(this).attr('href'),
        image = $('<img>', { src: href });

    // pridáme obrázok do overlaya a overlay zobrazíme
    overlay.html( image ).show();

    // zabránime predvolenej akcii, pretože nechceme opustiť našu stránku
    event.preventDefault();
});
```

Event handler sme naviazali na rodičovský element, pre prípad, že by sme do stránky chceli dynamicky vkladať nový a elementy. V takom prípade by sme totiž chceli zachovať lightbox funkcionality.

Lightbox by sme mohli vylepšiť napríklad tak, že pri kliknutí na ESC sa overlay zavrie. Výsledný kód by bol nasledovný:

```
(function($) {  
    // vytvorime si overlay, pridame ho na koniec stranky a nechame ho skryt  
    var overlay = $('<div>', { id: 'overlay' });  
    overlay.appendTo('body').hide();  
  
    // po kliknutí na overlay ho skryjeme  
    overlay.on('click', function() {  
        $(this).fadeOut('fast');  
    });  
  
    // skryjeme overlay na ESC  
    $(document).on('keyup', function(event) {  
        if ( event.which === 27 ) overlay.fadeOut('fast');  
    });  
  
    // po kliknutí zobrazíme lightbox  
    $('.gallery').on('click', 'a', function(event)  
    {  
        // vytiahneme href hodnotu, vytvoríme nový img a nastavíme src hodnotu  
        var href = $(this).attr('href'),  
            image = $('<img>', { src: href });  
  
        // pridáme obrazok do overlaya a overlay zobrazíme  
        overlay.html( image ).show();  
  
        // zabránime predvolenej akcii, pretože nechceme opustiť našu stránku  
        event.preventDefault();  
    });  
})(jQuery);
```

Event **keyup** nastane, keď používateľ prestane držať klávesu. Event object nesie dodatočné informácie o evente, jeden z nich je dostupný cez atribút **event.which**. Je to numerická reprezentácia zatlačenej klávesy. ESC je reprezentovaná číslom 27.

Efekt plynulého zoscrollovania k elementu

Na stránke budeme mať **ul** zoznam slúžiaci ako menu s odkazmi. Nižšie na stránky budú elementy, ktorých **id** atribúty budú korešpondovať s odkazmi v menu. Po kliknutí na príslušný odkaz sa chceme plynule posunúť na daný element.

```
<ul class="menu">
  <li><a href="#web">Web</a></li>
  <li><a href="#branding">Branding</a></li>
  <li><a href="#fotografia">Fotografia</a></li>
  <li><a href="#video">Video</a></li>
</ul>

...

<div id="web">...</div>

...

<div id="branding">...</div>

...

<div id="fotografia">...</div>

...

<div id="video">...</div>
```

Ak v **href** atribúte odkazu máme iba tzv. „hash“, po kliknutí na odkaz prehliadač zoskočí na daný element. To je základná funkcionálna prehliadačov, žiaden Javascript tu nie je potrebný. Javascript však môže túto interakciu spríjemniť – môžeme vďaka nemu pohyb rozanimovať.

Začneme klasicky – odchyťme kliknutie a zamedzíme predvolenému správaniu prehliadača:

```
<script>
var    menu = $('.menu'),
      menuLinks = menu.find('a');

menuLinks.on('click', function(event)
{
    event.preventDefault();
});
</script>
```

Ďalej potrebujeme získať hash daného linku. Vieme, že **this** korešponduje s elementom, na ktorý bolo kliknuté. V prípade prvého odkazu bude atribút

this.hash obsahovať hodnotu **#web**, čo môžeme v jQuery jehoducho použiť na získanie elementu s **id="web"**.

```
var el = $(this.hash);
```

Ďalej potrebujeme zistiť pozíciu tohoto elementu od vrchu stránky. K tomu slúži v jQuery metóda **.offset()** ktorá vráti object s atribútmi **top** a **left** pre odsadenie v pixeloch od vrchu a od ľavej strany stránky. My potrebuje hodnotu od vrchu.

```
el.offset().top
```

Pomocou metódy **.scrollTop()** vieme získať vertikálnu pozíciu scrollbaru daného elementu. Niektoré prehliadače viažu túto hodnotu na **html** element, iné na **body** element. My sa môžeme poistiť tým, že animovať hodnotu **scrollTop** budeme pre oba elementy.

```
<script>
var    menu = $('.menu'),
      menuLinks = menu.find('a');

menuLinks.on('click', function(event)
{
    // podľa hashu vyberieme element, na ktorý chceme scrollovať
    var el = $(this.hash);

    // animujeme hodnotu scrollTop k vrchnému odsaniu nášho elementu
    $('html,body').animate({ scrollTop: el.offset().top });

    // zabránime predvolenej akcii (priamy skok na element)
    event.preventDefault();
});
</script>
```

Ak by sme potrebovali scrollovať na vrch stránky, stačí zmeniť hodnotu **scrollTop** na nula.

```
// animujeme hodnotu scrollTop na vrch stránky
$('html,body').animate({ scrollTop: 0 });
```

Veľmi jednoducho sme dosiahli danú funkcionálnosť a to kompletne bez použitia pluginu. Na samotnú animáciu stačil jeden riadok kódu. Menej pluginov znamená menej súborov, ktoré je potrebné sťahovať a menej súborov znamená rýchlejšiu stránku. To je niečo, k čomu by sme mali smerovať.

Pokročilé jQuery animácie

Majme element **h1** s **position: relative** a **left: 0px** a zavolajme naň nasledovný jQuery kód:

```
$('#h1')
  .animate({ left: 400 }, 1000)
  .animate({ top: -400 }, 500);
```

Zbadáme, že najprv sa spustí prvá animácia a keď skončí, spustí sa druhá animácia. V jQuery sa totiž animácie stavajú do radu. Pridávajú sa do tzv. **queue**. Ak chceme spustiť animácie súčasne, môžeme prvej z nich zakázať, aby sa pridala do queue.

```
$('#h1')
  .animate({ left: 500 }, {
    duration: 1000,
    queue: false
  })
  .animate({ top: -400 }, {
    duration: 500
  });
```

Použili sme druhý spôsob zápisu metódy **.animate()**, kde ako druhý argument sme do metódy namiesto čísla poslali object. To nám dáva ďalšie možnosti, ako napríklad zrušenie pridania do queue pomocou **queue: false** u prvého volania.

Ak by sme naopak chceli, aby nielen že sa druhá animácia spustila po prvej, ale aby medzi nimi bola pauza, môžeme použiť metódu **.delay()**, ktorá ako argument berie počet milisekúnd.

```
$('#h1')
  .animate({ left: 500 }, {
    duration: 1000
  })
  .delay( 2000 )
  .animate({ top: -400 }, {
    duration: 500
  });
```

Teraz sa druhá animácia spustí po 2 sekundovej pauze.

Do konfiguračného objectu môžeme uviesť aj meno **easing** funkcie, podľa ktorej sa animácia zrýchľuje alebo spomaľuje.

```
$('#h1').animate({ top: -400 }, {
  duration: 500,
  easing: 'linear'
});
```

jQuery od základu podporuje easing funkcie **swing** a **linear**. Po pridaní pluginu sa ich vie naučiť oveľa viac. Prehľad rôznych funkcií aj s odkazom na plugin nájdeme na <http://easings.net/>.

Doteraz sme pri animovaní pozície posúvali element **na** určitý počet pixelov. Občas však potrebujeme posunúť element **o** určitý počet pixelov. Zobrať jeho aktuálnu pozíciu a zvýšiť ju dajme tomu o **50px**.

```
// po kazdom kliknutí o 50px doprava
$('h1').on('click', function() {
    $(this).animate({ left: '+=50' });
});

// po kazdom kliknutí o 50px dolava
$('h1').on('click', function() {
    $(this).animate({ left: '-=50' });
});
```

Moderné prehliadače majú solídnu podporu CSS3 animácií. jQuery sa často používa nie na realizáciu samotnej animácie, ale ako štartovač CSS3 animácií. Vytvorme si v CSS animáciu točenia sa a spustíme ju cez jQuery:

```
.round-round {
    -webkit-animation: spin 1s infinite linear;
}

@-webkit-keyframes spin {
    0% { -webkit-transform: rotate(0deg) }
    100% { -webkit-transform: rotate(360deg) }
}
```

Každý element s triedou **round-round** sa bude točiť dokola. V jQuery stačí pridať elementu túto triedu. Napríklad po kliknutí:

```
$('h1').on('click', function() {
    $(this).addClass('round-round');
});
```

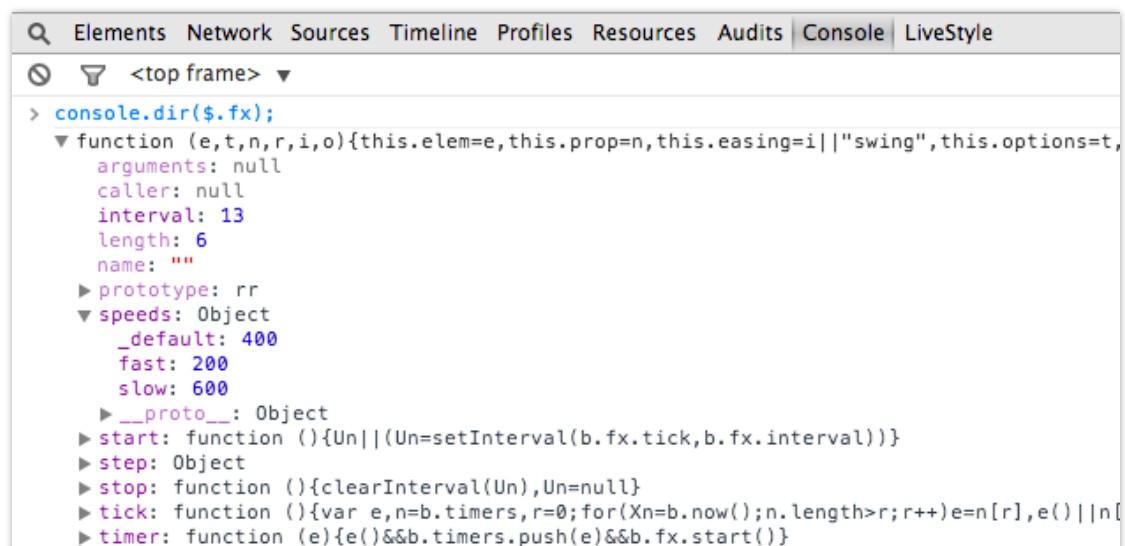
Použili sme metódu **.addClass()**. Podobne ako pri animáciách máme aj opak – metódu **.removeClass()** a prepínač – metódu **.toggleClass()**. Teraz vieme po kliknutí zapínať a vypínať točenie sa:

```
$('h1').on('click', function() {
    $(this).toggleClass('round-round');
});
```

Vieme, že animácie sa v jQuery pridávajú do radu. Môže sa stať, často po odpalovaní animácií na **mouseenter** alebo **mouseover** event, že sa tá istá animácia pridá do radu mnohokrát a elementy sa nám budú točiť a skákať po stránke, aj keď sme chceli, aby sa otočili alebo poskočili iba raz. Vtedy sa oplatí kontrolovať, či už element nie je animovaný a animáciu začať iba v prípade, že nie je.

```
// animujeme iba ak sa prave neanimuje
if (!$(this).is(':animated')) {
    // kod samotnej animacie
}
```

jQuery je možné rozšíriť o vlastné nastavenia alebo funkcionality. Na tomto stavajú jQuery pluginy. Tie často pridávajú novú funkcionality. Čo sa nastavení týka, jQuery ponúka niekoľko základných nastavení, ku ktorým máme prístup cez **\$.fx**.



Vidíme atribút **\$.fx.speeds**, kde sú uložené hodnoty pre rýchlosti animácií. Hodnota **_default** je rýchlosť, ktorou prebiehajú animácie, ak nezadáme žiadnu hodnotu špecificky. Vidíme tiež možnosti **fast** a **slow**. Namiesto zadania priamo počtu milisekúnd, môžeme zadať hodnoty **'fast'** alebo **'slow'** a vidíme, že fast sa zmení na zooms a slow na 6ooms. Tieto hodnoty môžeme jednoducho prepísať alebo si pridať hodnoty vlastné.

```
// prepiseme rychlost fast na 10ms
$.fx.speeds.fast = 10;

// zadame novu rychlost superSlow na 2000ms
$.fx.speeds.superSlow = 2000;

// použijeme nasu novo vytvorenu rychlost animacie
$('h1').animate({ left: 500 }, 'superSlow');
```

jQuery je šikovné, ale sú veci, ktoré samé od seba nedokáže. Napríklad animovať farby pozadí alebo textu písma. Našťastie je jQuery však veľmi ľahko upraviteľné a existuje množstvo pluginov, ktoré pomáhajú riešiť každý problém, na ktorý by ste mohli naraziť.

Ak potrebujeme animovať farby, stačí na stránku pridať **jQuery color plugin**:
<https://github.com/jquery/jquery-color/>.

AJAX

Keď kliknete na like a on sa uloží do databázy, keď máte otvorený twitter a bez toho, aby sa refreshla stránka, sa vám načítajú nové tweety, keď máte ľubovoľnú galériu obrázkov rozdelenú na niekoľko podstránok a kliknete na link a na chvíľu sa na stránke zobrazí točiac sa kolečko a potom zmizne a potom sa vám zobrazí nová strana obrázkov, má v tom prsty AJAX.

AJAX je "asynchrony JavaScript a XML" a slúži na realizáciu asynchrónnych requestov. Ak napíšete do prehliadača adresu stránky, prehliadač spraví **request** (vyšle žiadosť) na túto stránku, následne obdrží **response** (dostane odpoveď) na základe ktorej stránku vie zobrazíť. Prípadne nevie, keď padne server alebo nastane podobný problém.

Ak používate AJAX, rovnako zbežne **request – response** mechanizmus, ibaže sa tak stane na pozadí. Prehliadač sa potajomky pozrie na druhú stranu galérie a vráti dáta, ktoré vy, ako programátor, viete Javascriptom odchytiť a vložiť do stránky na miesto predošlých údajov.

Pozn.: Na testovanie AJAX requestov potrebné, aby na počítači bežal server. Ak iba otvoríte html súbor v prehliadači, nebude AJAX fungovať.

Majme nasledovný HTML kód:

```
<ul class="menu">
  <li><a href="index.html">Všetko</a></li>
  <li><a href="web.html">Web</a></li>
  <li><a href="branding.html">Branding</a></li>
  <li><a href="fotografia.html">Fotografia</a></li>
  <li><a href="video.html">Video</a></li>
</ul>

<div class="gallery">
  <div class="gallery-set" id="vsetko">
    <a href="img/dragon/image-1.jpg"></a>
    <a href="img/dragon/image-2.jpg"></a>
    <a href="img/dragon/image-3.jpg"></a>
    <a href="img/dragon/image-4.jpg"></a>
    <a href="img/dragon/image-5.jpg"></a>
    <a href="img/dragon/image-6.jpg"></a>
  </div>
</div>
```

V menu vidíme odkazy na podstránky. Ich kód je identický, s výnimkou obsahu elementu **<div class="gallery-set">**. Ten obsahuje pre každú podstránku iný set obrázkov.

Po kliknutí na odkaz v menu nechceme odísť na podstránku, chceme náš **.gallery-set** nahradiť elementom z podstránky, na ktorej odkaz sme klikli.

```
// vytiahneme si gallery element
var gallery = $('.gallery');

// po kliknutí na link v menu
$('.menu a').on('click', function(event) {

    // vytiahneme url adresu, ktoru chceme ist prezrieť ajaxom
    var a = $(this),
        href = a.attr('href');

    // ajax request, vytiahneme iba .gallery-set
    gallery.load(href + '.gallery-set');

    // nechceme byť presmerovaní na stránku
    event.preventDefault();

});
```

Najjednoduchšie to vieme dosiahnuť pomocou **.load()** metódy, ktorá akceptuje adresu stránky, za ktorú môžeme umiestniť selektor elementov, ktoré chceme z tejto stránky vytiahnuť. Ak by sme neposkytli tento selektor, do elementu **.gallery** by bol vložený kompletný kód podstránky, na ktorú sme klikli.

Metódy na spracovanie AJAX requestov

Poznáme rôzne typy requestov, najčastejšie pracujeme s requestami typu **GET** a **POST**. jQuery nám ponúka metódy usporiadané na realizáciu týchto requestov. Tieto metódy sa volajú **.get()** a **.post()**, pretože jQuery má vo zvyku dávať zmysel. Všetky AJAXové metódy, vrátane vymenovaných **.load()**, **.get()** a **.post()**, na pozadí využívajú metódu **.ajax()**.

```
$.ajax({
    url: href, // adresa, na ktoru robíme request
    type: 'GET', // typ requestu
    dataType: 'html', // typ obsahu, ktorý očakávame späť
    success: function(data) { // vykona sa po úspešnom requeste
        console.log(data);
    }
});
```

V tomto prípade sa nám v **data** vráti celý obsah stránky. Z neho si potrebujeme vytiahnuť **.gallery-set** a vložiť ho do našej galérie.


```

// nechame zmiznut aktualnu galeriu
gallery.find('.gallery-set').fadeOut();

// ajax request
$.ajax({
    url: href,
    type: 'GET',
    dataType: 'html',
    success: function(data) {

        // najdeme novy gallery set a skryjeme ho
        var newGallery = $(data).find('.gallery-set').hide();

        // nahradime aktualne set novym setom
        gallery.html( newGallery );

        // nechame ho zobrazit
        newGallery.fadeIn('fast');

    }
});

```

Nechat gallery set na začiatku zmiznúť je dobrý nápad z toho titulu, že používateľ dostane nejaký vizuálny feedback o tom, že sa niečo deje. Kým mu galéria pomaly mizne pred očami, na pozadí už beží AJAX, ktorý ťahá nové obrázky a zrazu šup – sú na stránke.

JSON

V dnešnej dobe nie je bežné, aby stránka, podobne ako my, načítala celý obsah a z neho vybrala element. Častejšie sa pošle request na server, ktorý vráti iba to, čo je naozaj treba, vo forme JSONu. **JSON** je „javascript object notation“ a je to zápis, ktorý je takmer identický so zápisom, ktorý používame pri definovaní objektu.

Napríklad keď si otvoríme v twitter a zoscrollujeme dole, nech sa načítajú ďalšie tweety, vráti sa nám json object, ktorý obsahuje iba HTML kód tých tweetov, ktoré sa priložia do stránky. Nie teda celá stránka, ako v našom prípade.

Ak chceme nasimulovať prácu s JSONom, môžeme poslať zopár GET requestov na graph.facebook.com.

```

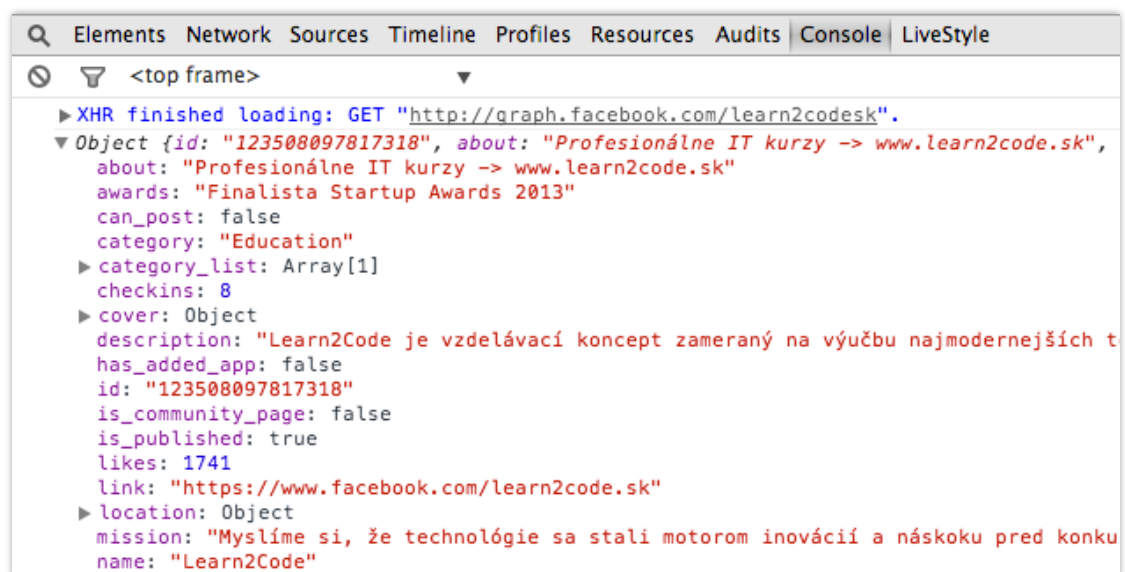
// pole adres, na ktore budeme robit get requesty
var fb = [
    'http://graph.facebook.com/learn2codesk',
    'http://graph.facebook.com/spaceunicorn',
    'http://graph.facebook.com/bedroomssk'
];

// prejdeme cez kazdy link v poli fb

```

```
$.each(fb, function(index, value) {
    $.ajax({
        url: value,
        type: 'GET',
        dataType: 'json', // očkávame, že sa vrátia data typu json
        success: function(data) {
            console.log( data );
        }
    });
});
</script>
```

Výsledok sa vráti vo forme JSONu, ktorý jQuery spracuje do klasického objektu:



Asynchrónne requesty

Máme 3 rôzne adresy, na ktoré posielame request. Ak by sme toto volanie niekoľkokrát zopakovali, videli by sme, že výsledky sa dostavia nie vždy v tom istom poradí. Poradie odkazov v poli sa nemení, ale poradie výsledkov sa mení. Toto je to, čo myslíme, keď hovoríme o „asynchrónnom“ requeste.

jQuery odpáli requesty a potom čaká, kým sa vráti odpoveď. Dopredu nevie, kedy odpoveď príde a ktorá bude prvá.

My nikdy nevieme, koľko bude takýto request trvať, ale často potrebujeme pracovať s jeho hodnotou. A teraz nemôžeme vyjsť von z ajaxovej success funkcie a písať ďalej náš skvelý kód, v ktorom použijeme hodnotu z requestu. Pretože je ešte nemáme. A preto je náš kód v tele success callback funkcie. To ale môže byť problém.

Čo ak potrebujeme napísať mnoho stoviek riadkov kódu? A čo ak niektoré z tých riadkov budú ďalšie AJAX requesty? Budeme naveky na seba vrstviť success callback funkcie? Neexistuje lepší spôsob?

Promises

jQuery pozná systém tzv. „sľubov“.

Zjednodušíme si kód na jedno volanie a odstránime success callback:

```
$.ajax({
  url: 'http://graph.facebook.com/learn2codesk',
  type: 'GET',
  dataType: 'json'
});
```

Uložme si toto ajax volanie do premennej a nechajme si vrátiť jeho sľub:

```
var req = $.ajax({
  url: 'http://graph.facebook.com/learn2codesk',
  type: 'GET',
  dataType: 'json'
}).promise();
```

Teraz môžeme použiť metódu **.then()**, ktorá odpáli, až keď sa sľub naplní. A už nie som obmedzený na success callback funkcie, môžem mať kopu volaní **.then()** metódy a medzitým kopu iných riadkov kódu.

```
// ... milion riadkov kodu ...

req.then(function(d,ta) {
  $('<p>', {
    text: d.ta.description
  }).insertAfter( gallery );
});

// ...milion riadkov kodu...

req.then(function() {
  alert('whee');
});
```

Samozrejme, nie vždy sa ajax volanie skončí úspešne. Vtedy chceme dať používateľovi najavo, že sa niečo pokazilo. Namiesto metódy **.then()** môžeme použiť metódu **.fail()** a v jej tele sa vysporiadať so zlyhaním.

Metóda **.always()** sa spustí vždy po ukončení requestu, bez ohľadu na jeho úspech. Pomocou metódy **.when()** vieme odpáliť funkciu, až keď sa dokončia všetky 3 requesty z nášho predošlého príkladu.

```
var fb = [  
    'http://graph.facebook.com/learn2codesk',  
    'http://graph.facebook.com/spaceunicorn',  
    'http://graph.facebook.com/bedroomssk'  
]  
  
// funkcia, ktora odpali ajax request  
function getData( index ) {  
    return $.ajax({  
        url: fb[index],  
        type: 'GET',  
        dataType: 'json'  
    }).promise();  
}  
  
// a vypiseme 'all finished' az ked skoncia vsetky 3 requesty  
$.when( getData(0), getData(1), getData(2) ).then(function() {  
    console.log( 'all finished!' );  
});
```

jQuery UI

jQuery UI je oficiálna User Interface nadstavba jQuery. Pridáva možnosť používať dialógové okná a progress bary a slidery a akordeóny a podobné vymoženosti. Vďaka jQuery UI vieme do našej stránky jednoducho zapracovať drag and drop a vo všeobecnosti nám pomáha tvoriť interaktívne webové aplikácie.

V podstate sa jedná o veľký jQuery plugin a podobne ako to pri nich býva zvykom, stačí pridať súbory pluginu do stránky a môžeme využívať jeho schopnosti. Na stránke <http://jqueryui.com/> nájdeme popis všetkej funkcionality, ktorú nám jQuery UI ponúka a samozrejme nájdeme tu aj odkaz na stiahnutie knižnice cez Download Builder (<http://jqueryui.com/download/>) kde si môžeme navoliť len tú funkcionality, ktorú potrebujeme a udržať tak veľkosť súboru čo najnižšie.

Vložíme jQuery UI na stránku:

```
<link rel="stylesheet" href="assets/ui/jquery-ui.min.css">
<!-- ostatne stylesheety a skripty -->
<!-- jquery ui skript vlozime ZA jquery skript -->
<script src="assets/ui/jquery-ui.min.js"></script>
```

Budeme pracovať s nasledovným HTML kódom:

```
<ul class="features">
  <li id="feature_1" class="badge">
    <div class="badge-left"><i class="fa fa-laptop fa-3x"></i></div>
    <div class="badge-right"><h5>Responzívny design</h5></div>
  </li>
  <li id="feature_2" class="badge">
    <div class="badge-left"><i class="fa fa-lightbulb-o fa-3x"></i></div>
    <div class="badge-right"><h5>Unikátne prvky webu</h5></div>
  </li>
  <li id="feature_3" class="badge">
    <div class="badge-left"><i class="fa fa-globe fa-3x"></i></div>
    <div class="badge-right"><h5>Široká paleta farieb</h5></div>
  </li>
  <li id="feature_4" class="badge">
    <div class="badge-left"><i class="fa fa-laptop fa-3x"></i></div>
    <div class="badge-right"><h5>Dynamické správanie</h5></div>
  </li>
  <li id="feature_5" class="badge">
    <div class="badge-left"><i class="fa fa-lightbulb-o fa-3x"></i></div>
    <div class="badge-right"><h5>Rýchly prvý návrh</h5></div>
  </li>
  <li id="feature_6" class="badge">
    <div class="badge-left"><i class="fa fa-globe fa-3x"></i></div>
    <div class="badge-right"><h5>Dôraz na detaily</h5></div>
  </li>
</ul>
```

Jedná sa o zoznam **li** elementov, ktoré chceme zmeniť na usporiadateľné.

Vďaka jQuery UI dostaneme prístup k novým metódam, napríklad **.sortable()**.

```
$('.features').sortable();
```

Každá UI metóda má svoje predvolené nastavenia a pre základnú funkcionálnosť stačí zavolať metódu bez akýchkoľvek argumentov. Každá metóda však ponúka množstvo nastavení a eventov, ktoré sú podrobne rozpísané v dokumentácii: <http://api.jqueryui.com/>.

```
$('.features').sortable({
  items: 'li:not(.pinned)', // usporiadať môžeme iba li elementy, ktoré nemajú class="pinned"
  axis: 'y', // hýbať elementami môžeme iba po osi y
  cursor: 'move', // pri ťahaní sa kurzor zmení na move
  handle: '.badge-left', // uchopiť a ťahať môžeme iba za element .badge-left
  update: function(event, ui) {
    // funkcia sa spustí, keď používateľ prestane ťahať element
  }
});
```

Chceme dať používateľovi možnosť, aby si tento zoznam usporiadal tak, ako ho potrebuje a aby sa táto pozícia mu zobrazila nabudúce, keď príde na stránku. Zameriame sa iba na jQuery stranu tohoto procesu, pretože o uloženie nastavení do databázy sa stará jazyk na strane servera.

Keď používateľ prestane hýbať so zoznamom, spravíme AJAX request na skript, do ktorého pošleme nové usporiadanie tohoto zoznamu. V dokumentácii sa môžeme dočítať o metóde **.serialize()**, ktoré vie uložiť pozíciu elementov vo forme, ktorá sa dá poslať AJAX requestom.

Aby metóda fungovala, musia mať usporiadateľné elementy atribút **id** s hodnotou vo forme **nazov_číslo**. Naš zoznam to spĺňa:

```
<li id="feature_1" class="badge">
<li id="feature_2" class="badge">
```

jQuery UI **.sortable()** ponúka metódu **update()**, ktoré sa spustí, keď používateľ prestane manipulovať so zoznamom. Do nej môžeme vložiť náš request:

```

$('.features').sortable({
  items: 'li',
  axis: 'y',
  cursor: 'move',
  handle: '.badge-left',
  update: function(event, ui) {

    var item = ui.item, // uložíme si element, ktorý sme presunuli
        order = features.sortable('serialize'); // a nové poradie elementov

    // vykonáme ajax request
    $.ajax({
      url: 'sort', // adresa serverového skriptu, ktorý uloží nové poradie do DB
      type: 'GET',
      data: order, // odosláme naše nové poradie
    }).then(function(data) {
      // po úspešnom preusporiadaní zvýrazníme presunutý element
      ui.item.find('.badge-right').effect('highlight');
    });

  }
});

```

jQuery UI tiež ponúka metódu **.effect()**, ktorou môžeme na element aplikovať rôzne animácie. Animácia **.effect('highlight')** na zlomok sekundy zvýrazní element žltou farbou, čo v tomto prípade symbolizuje úspešné uloženie novej pozície elementu do databázy.

Tvorba vlastného pluginu

V kapitole Pokročilé jQuery animácie sme rozširovali jQuery funkcionality o nové nastavenia. Zmenili sme prednastavenú rýchlosť animácie a pridali novú, vlastnú rýchlosť:

```
// prepiseme rychlost fast na 10ms
$.fx.speeds.fast = 10;

// zadame novu rychlost superSlow na 2000ms
$.fx.speeds.superSlow = 2000;
```

Na tomto istom princípe fungujú pluginy. Namiesto novej hodnoty však pridáme novú metódu. Rozšírime jQuery object o novú funkcionality. Pri nastaveniach animácií rozširujeme \$.fx object, pri nových metódach rozširujeme \$.fn object:

```
// .fx je skratka pre "effects"
// .fn je skratka pre "functions"
$.fn.lightbox = function() {

}
```

Pridali sme do \$.fn novú metódu – **lightbox**. Chceme náš príklad z kapitoly **Tvorba jednoduchého image lightboxu** prepísať na jQuery plugin, ktorý sa bude v kóde používať nasledovne:

```
// plugin mozeme iba jednoducho zavolať na elementy galerie
$('.gallery a').lightbox();

// prípadne mozeme jeho funkcionality upraviť nastaveniami
$('.gallery a').lightbox({
  speed: 300,
  showEffect: 'fadeIn',
  hideEffect: 'slideUp',
  complete: function() {
    // tato funkcia sa vykona po skryti lightboxu
  }
});
```

Vieme, že náš kód by vždy mal byť izolovaný od ostatného kódu, ktorý môžeme mať na stránke. To sme si popisovali v kapitole **Document ready**. Náš plugin obalíme do robustnejšej verzie anonymnej funkcie z tejto kapitoly.


```
;(function($, window, document, undefined) {

    $.fn.lightbox = function() {
        return this.each(function() {
            // telo samotneho pluginu
        });
    }

})(jQuery, window, document);
```

Telo pluginu sa bude nachádzať v **.each()** metóde, pretože ak zavoláme plugin na kolekciu elementov (viac elementov ako jeden), chceme kód pluginu naviazať na každých z týchto elementov.

Naša **lightbox()** metóda vracia jQuery object pomocou **return**. To je potrebné, aby sa zachovala schopnosť reťazenia jQuery metód.

Vieme, že objekty v JavaScripte môže niesť atribúty a metódy. Tento fakt môžeme využiť na prepísanie funkcionality nášho existujúceho lightboxu do tzv. Object literal patternu.

```
var Lightbox = {
    image: null,
    overlay: null,
    element: null,
    settings: null,

    init: function( element, settings ) {
        this.element = element;
        this.settings = settings;
        this.overlay = $('#overlay');

        if ( this.overlay.length )
            this.image = this.overlay.find('img');
        else
            this.createOverlay();

        this.attachHandlers();
    },

    // vytvor overlay, vlož do neho img element
    createOverlay: function() {
        this.overlay = $('<div/>', { id: 'overlay' }).hide();
        this.image = $('<img/>', { src: "", alt: "" }).appendTo( this.overlay );
        this.overlay.appendTo('body');
    },

    // prilep plošnice
    attachHandlers: function() {
        // skryjeme lightbox na click na overlay
        this.overlay.on('click', function() {
```

```

        Lightbox.hide();
    });
    // skryjeme na ESC
    $(document).on('keyup', function(event) {
        if ( event.which === 27 ) Lightbox.hide();
    });
    // zobrazime lightbox po kliknutí na element
    this.element.on('click', function(event) {
        event.preventDefault();
        Lightbox.setImage( $(this).attr('href') );
        Lightbox.show();
    });
},

// nastavime adresu obrazka
setImage: function( url ) {
    this.image.attr('src', url);
},

// zobraz lightbox
show: function() {
    this.overlay[ this.settings.showEffect ]( this.settings.speed );
},

// skry lightbox
hide: function() {
    var config = this.settings;
    this.overlay[ config.hideEffect ]( config.speed, function() {
        if ( $.isFunction( config.complete ) ) {
            config.complete.call( this );
        }
    });
}

}
}

```

Lightbox funguje rovnako, teraz je však všetka jeho funkcionálna oddelená v samostatnej stojacom objecte. Tento môžeme v plugine pomocou **Lightbox.init()** naviazť funkcionálnu na každý element z kolekcie:

```

$.fn.lightbox = function( options ) {
    // základné nastavenia, ktoré sa spoja s options, ak user nejake zada
    var settings = $.extend({
        speed      : 250,
        showEffect : 'fadeIn',
        hideEffect : 'slideUp',
        complete   : null
    }, options);

    // naviazeme lightbox funkcionálnu na každý link z kolekcie a vrátime object

    return this.each( function() {
        Lightbox.init( $(this), settings );
    });
}

```

Pridali sme tiež možnosť zmeniť rýchlosť animácie alebo animačné efekty pri skrytí a zobrazení lightboxu. Výsledný kód pluginu je tu:

```
;(function($, window, document, undefined) {

    $.fn.lightbox = function( options ) {

        var Lightbox = {

            image: null,
            overlay: null,
            element: null,
            settings: null,

            init: function( element, settings ) {
                this.element = element;
                this.settings = settings;
                this.overlay = $('#overlay');

                if ( this.overlay.length )
                    this.image = this.overlay.find('img');
                else
                    this.createOverlay();

                this.attachHandlers();
            },

            // vytvor overlay, vlož do neho img element
            createOverlay: function() {
                this.overlay = $('<div/>', { id: 'overlay' }).hide();
                this.image = $('<img/>', { src: "", alt: "" }).appendTo( this.overlay );
                this.overlay.appendTo('body');
            },

            // prilep plošnice
            attachHandlers: function() {
                // skryjeme lightbox na click na overlay
                this.overlay.on('click', function() {
                    Lightbox.hide();
                });
                // skryjeme na ESC
                $(document).on('keyup', function(event) {
                    if ( event.which === 27 ) Lightbox.hide();
                });
                // zobrazíme lightbox po kliknutí na element
                this.element.on('click', function(event) {
                    event.preventDefault();
                    Lightbox.setImage( $(this).attr('href') );

                    Lightbox.show();
                });
            },

            // nastavíme adresu obrázka
            setImage: function( url ) {
```

```
        this.image.attr('src', url);  
    },
```

```
    // zobraz lightbox  
    show: function() {  
        this.overlay[ this.settings.showEffect ]( this.settings.speed );  
    },
```

```
    // skry lightbox  
    hide: function() {  
        var config = this.settings;  
        this.overlay[ config.hideEffect ]( config.speed, function() {  
            if ( $.isFunction( config.complete ) ) {  
                config.complete.call( this );  
            }  
        });  
    };  
}  
}
```

```
    // zakladne nastavenia, ktore sa spoja s options, ak user nejake zada  
    var settings = $.extend({  
        speed      : 250,  
        showEffect : 'fadeIn',  
        hideEffect : 'slideUp',  
        complete   : null  
    }, options);
```

```
    // naviazeme lightbox funkcionalitu na kazdy link z kolekcie a vratime object  
    return this.each( function() {  
        Lightbox.init( $(this), settings );  
    });  
}
```

```
})(jQuery, window, document);
```