# JAVASCRIPT

**Softwarové inžinierstvo**

MARTIN TIMOTHY TIMKO

7.9. 2017 – 11.9. 2017

# 1 ÚVOD

Kedysi dávnejšie bol jazyk JavaScript v pozadí hlavného programátorského prúdu, ale dnes je to jazyk ohľadne webového vývoja na prvom mieste. Jeho hlavnou výhodou bolo a stále je, že dokáže zefektívniť prácu na strane klienta a znížiť zaťaženie servera pri opakovanom načitávaní web stránky a tým poskytuje plynulejšie fungovanie podobné napr. z klasických desktopových aplikácií.

Jazyk JavaScript bol predstavený v roku 1995 ako jazyk pomocou ktorého bolo možné pridávať programy do webových stránok v internetovom prehliadači *Netscape Navigator*. Rozšíril sa do všetkých internetových prehliadačov a stal sa určitým štandardom pre moderný web. Často sa zamieňa programovací jazyk Java s názvom programovacieho jazyka JavaScript, avšak jazyk Java je samostatný programovancí jazyk, ktorý s jazykom JavaScript nemá vôbec nič spoločné. V čase, keď vznikal jazyk JavaScript bol jazyk Java už veľmi dobre etablovaný a získaval na popularite. Pravdepodobne sa jednalo o marketingový ťah aj keď v tomto čase jazyk Java slúžil skôr pre účely v použití spotrebnej elektroniky než webových stránok.

Aj napriek tomu, že je dnes jazyk JavaScript rozšírený, z určitých dôvodov nepatrí medzi veľmi obľúbený jazyk. Klasické kompilovateľné programovacie jazyky ako napr. C/C++, C#, Java, Delphi, alebo Pascal majú striktné pravidlá, sú silne typové jazyky a staticky typované jazyky. To znamená, že tieto jazyky vyžadujú, aby dátové typy boli pred svojím uvedením definitívne určené a zároveň tieto jazyky vykonávajú vstupnú analýzu kódu, pomocou ktorej je možné zistiť dátový typ každej premennej. Jazyk JavaScript je tzv. *interpretovateľný jazyk* alebo tiež veľmi podobný *funkcionálnym programovacím jazykom* a z toho dôvodu je naopak slabo typový jazyk a dynamicky typovaný jazyk. Teda, typ premennej je zisťovaný až počas behu programu a ak danú hodnotu nie je možné priradiť premennej s nejakým odlišným typom, tak sa vykoná automatické pretypovanie.

A presne v tomto spočíva hlavný problém JavaScriptu. Konzervatívne programovanie používajúce kompilovateľné programovacie jazyky poskytujú pomerne malý priestor na nejaké nejasné, nepredvídateľné konštrukcie a preto je jazyk JavaScript (ako aj všetky funkcionálne jazyky) považovaný z tohto pohľadu za jazyk, ktorý nemá pevné základy a je dosť liberálny. Tento jazyk, ako už bolo spomínané, nadobudol svoje plné praktické uplatnenie len nedávno a preto sa dostal do povedomia takpovediac ako outsider. Práve z toho dôvodu je potrebné pri tomto jazyku zmeniť konzervatívne myslenie na liberálne alebo presnejšie povedané používať obidve typy myslení, pretože aj keď je dnes už jazyk JavaScript použiteľný aj ako samostaný jazyk, vo väčšine prípadoch sa používa v súčinnosti s webovými technológiami a teda primárne s nejakým ďalším konvervatívnym jazykom, C#, Java alebo PHP.

## 2 SYNTAX

Písať programy v JavaScripte je možné pomocou HTML stránky v elemente `script` oddelene v externom súbore s príponou `.js`, podobne ako pri CSS štýle. Okrem toho je možné použiť niektoré online prekladače, napr. *JSBin*, *<JSFiddle* alebo *Plunker*. V každom prípade výsledok bude všade rovnaký. Na editovanie JavaScriptových súborov môžme použiť napr. *NotePad++*.

V starších verziách HTML sme museli písať element `script` v podobe:

```
<script type="text/javascript">

    //write JavaScript here..

</script>
```

Verzia HTML5 umožňuje použitie v skrátenom tvare:

```
<script>

    //write JavaScript here..

</script>
```

Z hľadiska umiestnenia v HTML je možné použitie v elemente `head`: (aj s externým súborom `.js`)

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>JavaScript Demo</title>
    <script>
        //write JavaScript code here.

    </script>
    <script src="/PathToScriptFile.js"></<script> @* External JavaScript file *@
</head>
<body>
    <h1> JavaScript Tutorials</h1>

    <p>This is JavaScript sample.</p>

</body>
</html>
```

Rovnako je možné písať aj v medziach elementu `body`:

```
<!DOCTYPE html>
```

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>JavaScript Demo</title>

</head>
<body>
    <h1> JavaScript Tutorials</h1>
    <p>This is JavaScript sample.</p>

        <!--Some HTML here.. -->

    <script>
        //write JavaScript code here..

    </script>
    <script src="/PathToScriptFile.js"></script> @* External JavaScript file *@
</body>
</html>
```

Jazyk JavaScript je tzv. **case sensitive**, to znamená, že **záleží** na veľkosti písmen v premenných, funkciách, kľúčových slovách a podobne. Reťazce je možné zapisovať v úvodzovkách alebo apostrofoch:

```
<script>

    "Hello World" //JavaScript string in double quotes

    'Hello World' //JavaScript string in single quotes

</script>
```

Čísla je možné používať prakticky všetky dostupné, najpoužívanejšie celé čísla, reálne čísla, hexadecimálne a podobne. Jedinou podmienkou je, aby neboli uzavreté do úvodzoviek alebo apostrofov ako reťazce. Za číslo ako také sa považujú aj boolovské hodnoty, true a false.

V klasických kompilovateľných jazykoch ako napr. C#, Java je nutné písať po každej operácii, príkaze bodkočiarku. JavaScript je v tomto smere benevolentný, a teda bodkočiarka sa nemusí písať, ale doporučuje sa. Prekladač ignoruje biele znaky.

```
var one =1;
var one   =   1;
var one        =         1;
```

Komentáre sú rovnaké ako v jazyku C:

```
var one =1; // this is a single line comment
```

| Kľúčové slová | | |
|---|---|---|
| var | function | if |
| else | do | while |
| for | switch | break |
| continue | return | try |
| catch | finally | debugger |
| case | class | this |
| default | false | true |
| in | instanceOf | typeOf |
| new | null | throw |
| void | width | delete |

```
/* this
is multi line
comment*/

var two = 2;
var three = 3;
```

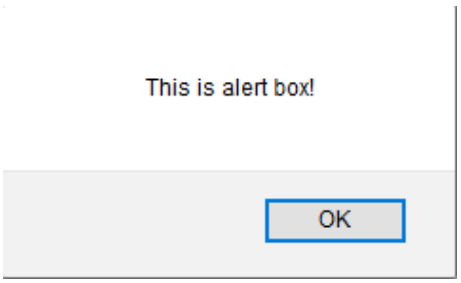
Tak ako každý programovací jazyk aj JavaScript používa niektoré kľúčové slová, ktoré sú rezervované prekladačom a nie je možné ich použiť v rámci premenných:

## 2.1 Dialógové okná

Dialógové okná sa podobajú oknám z čias prvotných výplodov *Win32 API*, ale našťasie sa v JavaScripte nepoužívajú príliš často, pretože z používateľského hľadiska pri prezeraní webovej stránky je akékoľvek vyskakovacie okno obťažujúce pre prehliadajúceho, takže je vždy lepšie mať všetky prvky prístupné na jednej stránke s nejakým sprístupnením. Dialógové okná sa rozdeľujú podľa charakteru informácie, ktoré nám poskytujú.

Takže jednoduché výstražné okno je možné napísať pomocou funkcie `alert()`:

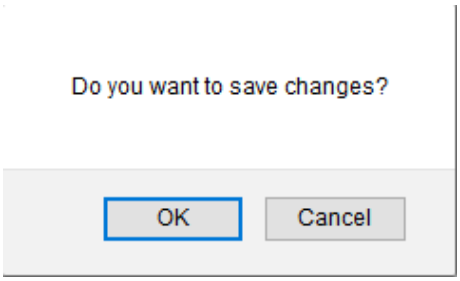
```
alert("This is alert box!");  // display string message

alert(100); // display number

alert(true); // display boolean
```



Ostatné okná sú analogické.

Ďalšie okná sú potrvdzovacie okná. Ako je možné vidieť, typ dialógového okna nám automaticky sprístupní jednotlivé prislúchajúce tlačidlá, ktoré korešpondujú zo stavom dialógového okna.

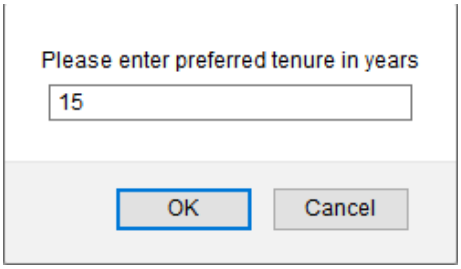
```
var userPreference;

if (confirm("Do you want to save changes?") == true) {

    userPreference = "Data saved successfully!";

} else {

    userPreference = "Save Cancelled!";

}
```




V tomto prípade je možné použiť funkciu `confirm()` ako podmienku napr. pre zapísanie stavu.

Posledným typom je dialógové okno poskytujúce nejaký vstup od používateľa.

```
var tenure = prompt("Please enter preferred tenure in years", "15");

if (tenure != null) {
    alert("You have entered " + tenure + " years" );
}
```



## 2.2 Premenné a dátové typy

JavaScript rozlišuje jednoduché dátové typy:

- String
- Number
- Boolean
- Null
- Undefined

A zložené dátové typy:

- Object
- Date
- Array

Ako už bolo spomenuté, nie je možné definovať dopredu daný dátový typ premennej, pretože JavaScript je slabo typový jazyk a dynamicky typovaný jazyk.

V poslednom čase okrem iného aj jazyk C# umožňuje písať konštrukcie s kľúčovým slovom `var`, čo predstavuje nejakú bližšie neurčenú premennú.

**Syntax:**

```
var <variable-name>;
var <variable-name> = <value>;
```

V JavaScripte existuje to isté a teda, ak chceme vytvoriť premennú, tak ju vytvoríme nasledujúcim spôsobom:

```
var one = 1; // variable stores numeric value
var two = 'two';  // variable stores string value
var three;  // declared a variable without assigning a value
```

Problém je ten, že ono **všetky** typy premenných, dátové typy sa definujú rovnakým spôsobom a teda môže veľmi ľahko dôjsť k chybe:

```
var one = 1;  // numeric value
one = 'one'; // string value
one = 1.1; // decimal value
one = true; // Boolean value
one = null; // null value
```

Vyhodnotenie na ľavej strane výrazu je podmienené typom premennej na strane pravej. Premenná `one` nie je jedinečná a je možné jej priradiť **akýkoľvek** typ. Avšak daná premenná sa v priebehu programu môže viac-krát meniť a prekladač nám to **neoznámi**, takže je všetko plne v kompetencii programátora.

Konkrtétne dátové typy je možné nájsť s jednoduchým popisom napr. na *JavaScript Data Types*.

| Operátor | Popis |
|---|---|
| == | Compares the equality of two operands without considering type. |
| === | Compares equality of two operands with type. |
| != | Compares inequality of two operands. |
| > | Checks whether left side value is greater than right side value. |
| | If yes then returns true otherwise false. |
| < | Checks whether left operand is less than right operand. |
| | If yes then returns true otherwise false. |
| >= | Checks whether left operand is greater than or equal to right operand. |
| | If yes then returns true otherwise false. |
| <= | Checks whether left operand is less than or equal to right operand. |
| | If yes then returns true otherwise false. |

## 2.3 Operátory

Situácia ohľadne operátorov je v jazyku JavaScript obdobná ako v iných programovacích jazykoch. JavaScript ponúka nasledovné kategórie operátorov:

- Aritmetické operátory
- Porovnávacie operátory
- Logické operátory
- Prideľovacie operátory
- Podmienečné operátory

Jedinú výnimku tvoria porovnávacie operátory.
Príklady jednotlivých operátorov:

```
var a = 5, b = 10, c = "5";
var x = a;

a == c; // returns true
a === c; // returns false
a == x; // returns true
a != b; // returns true
a > b; // returns false
a < b; // returns true
a >= b; // returns true
a <= b; // returns true
a >= c; // returns true
a <= c; // returns true
```

Teda jediný rozdiel spomedzi všetkých typov operátorov všeobecne je v druhom prípade pri operátore ===, ktorý slúži na porovnanie dvoch operandov s typom daného operandu. Zaujímavá situácia je v prvom type operátora, v klasickom porovnávacom operátorovi ==, kde je výsledok pravdivý aj napriek tomu, že sa jedná o dva operandy rôzneho typu (číslo a reťazec). Aj tu je možné vidieť určitú benevolenciu jazyka JavaScript.

## 2.4 Funkcie

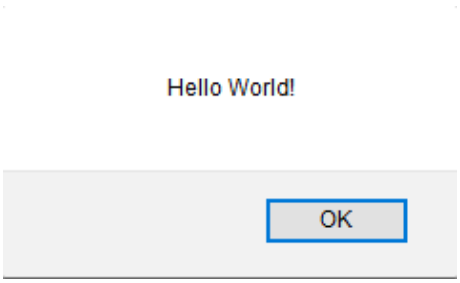
Funkcie sú veľmi podobné ostatným programovacím jazykom, avšak JavaScript vyžaduje pri tvorbe funkcie použiť kľúčové slovo `function`, čo je viac príznačné pre funkcionálne jazyky než pre moderné objektové jazyky.

**Syntax:**

```
//defining a function
function <function-name>()
{
    // code to be executed
};
//calling a function
<function-name>();
```

Jednoduchý príklad:

```
function ShowMessage() {
    alert("Hello World!");
}

ShowMessage();
```




Funkcia s parametrami bude vyzerať nasledovne:

```
function ShowMessage(firstName, lastName) {
    alert("Hello " + firstName + " " + lastName);
}

ShowMessage("Steve", "Jobs");
ShowMessage("Bill", "Gates");
ShowMessage(100, 200);
```

Tento príklad funkcie s parametrami je možné napísať aj použitím argumentov objektu. Argumenty objektu poskytujú hodnoty každého parametra. Argumenty objektu fungujú ako klasické polia, ktoré sú prístupne pomocou indexu. Rozdiel je v tom, že tieto argumenty objektu nefungujú pri samotných klasických poliach.

```
function ShowMessage(firstName, lastName) {
    alert("Hello " + arguments[0] + " " + arguments[1]);
}

ShowMessage("Steve", "Jobs");
ShowMessage("Bill", "Gates");
ShowMessage(100, 200);
```

Výsledok bude rovnaký ako v predchádzajúcich príklade.

**Zaujímavý príklad** je, že funkcia môže mať ako návratovú hodnotu samotnú funkciu. Takéto vnorené funkcie je možné mať niekoľko.

```
function multiple(x) {

    function fn(y)
    {
        return x * y;
    }

    return fn;
}

var triple = multiple(3);
triple(2); // returns 6
triple(3); // returns 9
```

JavaScript umožňuje priradiť funkciu nejakej premennej a následne použiť túto **premennú** ako funkciu.

```
var add = function sum(val1, val2) {
    return val1 + val2;
};

var result1 = add(10,20);
var result2 = sum(10,20); // not valid
```

Známejšie funkcie, tzv. anonymné funkcie poskytuje aj jazyk JavaScript.

```
var showMessage = function (){
    alert("Hello World!");
};

showMessage();

var sayHello = function (firstName) {
    alert("Hello " + firstName);
};

showMessage();
sayHello("Bill");
```

Jedná sa o funkcie bez uvedenia názvu funkcie. Takéto funkcie však musia byť priradené premennou.

## 2.5 Príkazy vetvenia

JavaScript ponúka veľmi podobné príkazy vetvenia aké majú ostatné programovacie jazyky, teda konkrétne sa jedná o dané typy podmieneného vetvenia:

- podmienka if
- podmienka if-else
- podmienka else-if

**Syntax podmienky if:**

```
if(condition expression)
{
    // code to be executed if condition is true
}
```

Teda klasický príklad:

```
if( 1 > 0)
{
    alert("1 is greater than 0");
}

if( 1 < 0)
{
    alert("1 is less than 0");
}
```

Určitou zaujímavosťou, ako už bolo spomínané, je napr. prípad, keď porovnávame dva operandy:

```
if(1=="1")
{
    alert("== operator does not consider types of operands");
}

if(1==="1")
{
    alert("=== operator considers types of operands");
}
```

V prvom prípade sa jedná o porovnanie bez ohľadu na typ operandov a v druhom prípade záleží na type jednotlivých operandov.

**Syntax podmienky if-else:**

```
if(condition expression)
{
    //Execute this code..
}
else{
    //Execute this code..
}
```

Jednoduchý príklad:

```
var mySal = 500;
var yourSal = 1000;

if( mySal > yourSal)
{
    alert("My Salary is greater than your salary");
}
else
{
    alert("My Salary is less than or equal to your salary");
}
```

**Syntax podmienky else-if:**

```
if(condition expression)
{
    //Execute this code block
}
else if(condition expression){
    //Execute this code block
}
```

Jednoduchý príklad:

```
var mySal = 500;
var yourSal = 1000;

if( mySal > yourSal)
{
    alert("My Salary is greater than your salary");
}
else if(mySal < yourSal)
{
    alert("My Salary is less than your salary");
}
```

Samozrejme, tak ako v iných jazykoch aj v tomto je možné použiť väčšie množstvo `else-if` podmienok:

```
var mySal = 500;
var yourSal = 1000;

if( mySal > yourSal)
{
    alert("My Salary is greater than your salary");
}
else if(mySal < yourSal)
{
    alert("My Salary is less than your salary");
}
else if(mySal == yourSal)
{
    alert("My Salary is equal to your salary");
}
```

Ďalší typ príkazov vetvenia je príkaz `switch`. Tento príkaz nevykazuje žiadne abnormality oproti všetkým typom programovacích jazykov.

**Syntax:**

```
switch(expression or literal value){
    case 1:
        //code to be executed
      break;
    case 2:
        //code to be executed
        break;
    case n:
        //code to be executed
        break;
    default:
        //default code to be executed
        //if none of the above case executed
}
```

Jednoduchý príklad:

```
var a = 3;

switch (a) {
    case 1:
        alert('case 1 executed');
        break;
    case 2:
        alert("case 2 executed");
        break;
   case 3:
        alert("case 3 executed");
```

```
        break;
    case 4:
        alert("case 4 executed");
        break;
    default:
        alert("default case executed");
}
```

## 2.6 Príkazy cyklu

Najpoužívanejší príkaz pre cyklus je príkaz `for`.

Syntax:

```
for(initializer; condition; iteration)
{
    // Code to be executed
}
```

Jednoduchý príkad:

```
for (var i = 0; i < 5; i++)
{
    console.log(i);
}
```

Zložitejší príkad:

```
var arr = [10, 11, 12, 13, 14];

for (var i = 0; i < 5; i++)
{
    console.log(arr[i]);
}
```

Tento príkaz je prakticky úplne ekvivalentný s príkazmi ostatných programovacích jazykov. Verzia cyklu for_each, ktorá existuje v plne objektových jazykoch nemá v JavaScripte svoje miesto, ale v zásade to ani nie potrebné, pretože samotné priradenie typu premenej a kontrola typov je v JavaScripte pomerne liberálna a z toho dôvodu to je prakticky možné písať priamo.

Ďalším typom cyklu je príkaz while.

**Syntax:**

```
while(condition expression)
{
    /* code to be executed
    till the specified condition is true */
}
```

Takisto sa jedná o obvyklý príkaz ako v predchádzajúcom prípade.

```
var i = 0;

while(i < 5)
{
    console.log(i);
    i++;
}
```

Druhá verzia príkazu `while` je `do-while`.
**Syntax:**

```
do{

    //code to be executed

}while(condition expression)
```

Teda, príkaz sa vykoná za každých podmienok vždy prvý-krát a ostatné cykly sú podmienené podmienkou pre tento cyklus.

```
var i = 0;

do {
     alert(i);
     i++;

} while(i < 5)
```

## 2.7 Výnimky

Napriek tomu, že sa jazyk JavaScript podobá jazyku C, výnimkou sú výnimky, ktoré sa podobajú do veľkej miery moderným objektovým jazkom.
**Syntax:**

```
try
{
    // code that may throw an error
}
catch(ex)
{
    // code to be executed if an error occurs
}
finally {
    // code to be executed regardless of an error occurs or not
}
```

Klasicky je `try` blok testovací, ochranný blok, blok `catch` zachytí túto výnimku a následne je možné ju nejakým spôsobom adekvátne popísať a nepovinný block `finally` sa vykoná za každých okolností, teda bez ohľadu na stav výnimky.

```
try
{
    var result = Sum(10, 20); // Sum is not defined yet
}
catch(ex)
{
    document.getElementById("errorMessage").innerHTML = ex;
}
finally {
    document.getElementById("message").innerHTML = "finally block executed";
}
```

## 2.8 Objektový model

JavaScript ponúka určitý objektový model podobný z moderných objektových jazykov, avšak nevykazuje znaky podobnosti s týmito jazykmi. Jedná sa skôr o akýsi umelý konštrukt. V zásade je možné za objekt považovať všetky premenné v JavaScripte okrem primitívnych typov. Nasledujúcim spôsobom je možné vidieť ako vytvoriť triedu.

```
function Person() {
    this.firstName = "unknown";
    this.lastName = "unknown";
}

var person1 = new Person();
person1.firstName = "Steve";
person1.lastName = "Jobs";

alert(person1.firstName + " " + person1.lastName);

var person2 = new Person();
person2.firstName = "Bill";
person2.lastName = "Gates";

alert(person2.firstName + " " + person2.lastName );
```

Funkcia `Person()` je považovaná za triedu. Nemá žiadny návratový typ a jednotlivé položky sú považované ako property položky. Pristupujeme k nej pomocou inštancie, teda funkcia `Person()` je len na základe inštancie prehlasovaná za triedu.

Nasledujúci príklad popisuje triedu s nejakou metódou, v podstate anonymnou metódou.

```
function Person() {
            this.firstName = "unknown";
            this.lastName = "unknown";
            this.getFullName = function() {
                return this.firstName + " " + this.lastName;
            }
```

```
        };

var person1 = new Person();
person1.firstName = "Steve";
person1.lastName = "Jobs";

alert(person1.getFullName());

var person2 = new Person();
person2.firstName = "Bill";
person2.lastName = "Gates";

alert(person2.getFullName());
```

Takýmto spôsobom je možné písať metódu v rámci triedy. Konštruktor je možné písať s parametrami nasledujúcim spôsobom.

```
function Person(FirstName, LastName, Age) {
        this.firstName = FirstName || "unknown";
        this.lastName = LastName || "unknown";
        this.age = Age || 25;
        this.getFullName = function () {
            return this.firstName + " " + this.lastName;
        }
};

var person1 = new Person("James","Bond",50);
alert(person1.getFullName());

var person2 = new Person("Tom","Paul");
alert(person2.getFullName());
```

Gettery a settery obdobné z jazykov C# a Java je možné pomocou funkcie písať v JavaScripte nasledovne:

```
function Person() {
    var _firstName = "unknown";

    Object.defineProperties(this, {
        "FirstName": {
            get: function () {
                return _firstName;
            },
            set: function (value) {
                _firstName = value;
            }
        }
    });
```

```
};

var person1 = new Person();
person1.FirstName = "Steve";
alert(person1.FirstName );

var person2 = new Person();
person2.FirstName = "Bill";
alert(person2.FirstName );
```

Takže takýmto spôsobom je možné programovať objektovo, aj keď za objektové programovanie sa považuje tzv. *prototypovanie*.

# 3 FORMULÁRE A FORMULÁROVÉ POLIA

Formulárové ovládacie prvky patria k štandardnej sade všetkých moderných programovacích jazykov. JavaScript ako programovací jazyk neobsahuje ovládacie prvky, ale rozširuje funkcionalitu HTML ovládacích prvkov. Existujú niektoré ďalšie HTML/CSS/JS knižnice, ktoré poskytujú esteticky prívetivejší vzhľad ovládacích prvkov ako aj samotnú funkcionalitu týchto prvkov, napr. *Bootstrap*, *jQuery*.

Tieto knižnice, frameworky sú dôležité okrem iného aj z toto dôvodu, že poskytujú množstvo zautomatizovaných funkcií ako napr. validácia formulárov alebo dnes už veľmi rozšírený *responzívny dizajn* a rôzne optimalizácie pre množstvo internetových prehliadačov. Práve z toho dôvodu sa stali už súčasťou moderného webového vývoja.

Takže, ak chceme vytvoriť jednoduchý formulár, tak môžme postupovať takto:

```
<!DOCTYPE html>
<html lang="en">
    <head>
            <meta charset=UTF-8" />
            <title>Form</title>
    </head>
    <body>

<form name="mojFormular" action="index.html" method="get">
             Meno: <input type="text" name="login"><br />
     Heslo: <input type="password" name="heslo"><br />
             <input type="submit" value="Prihlasit">
        </form>

    </body>
</html>
```


Ono, vytvoriť dynamický formulár, ktorý by nejakým spôsobom komunikoval zo serverom v JavaScripte nie je možné, pretože sa jedná len o klientský programovací jazyk. Teda, možme pripraviť daný formulár pre komunikáciu zo serverom. V uvedenom príklade vytvoríme formulár prostredníctvom elementu `form`. Pri formulároch sa jedná v podstate len o to, akým spôsobom budú jednotlivé ovládacie prvky obsiahnuté v danom formulári posielané na ďalšie spracovanie najčastejšie pre server. Tento problém rieši typ odosielania dát, ktorý pozostáva z dvoch metód:

- Metóda GET
- Metóda POST

Odovzdávanie dát z klienta na server pomocou metódy *GET* sa realizuje pridaním dvojíc `name=value` do URL adresy za symbol `?` a oddelením pomocou znaku `&`, alebo len pridaním nejakého názvu `name`, kde `name` je názov formulárového ovládacieho prvku a `value` je jedno hodnota.

Takže, ak by sme daný formulár s metódou `GET` poslali, vo vyhľadávacom paneli by sme získali nasledovný URL reťazec:

```
/index.html?login=admin&heslo=12345
```

Problém v tejto metóde je ten, že dané prihlasovacie heslo je viditeľné a uchováva sa v histórii internetového prehliadača a ani prípadné použitie šifrovania hesiel nezabráni úniku hesla. Heslá ako také sa zvyknú archovovať v rôznych logoch na napr. aplikačnom serveri. Práve z toho dôvodu sa používa metóda *POST*, ktorá neodovzdáva dáta z klienta na sever pomocou prepisovania URL adresy, ale odovzdáva ich prostredníctvom ovládacích prvkov na formulári.

```
<!DOCTYPE html>
<html lang="en">
    <head>
            <meta charset=UTF-8" />
            <title>Form</title>
    </head>
    <body>

<form name="mojFormular" action="http://google.sk" method="post">
             Meno: <input type="text" name="login"><br />
     Heslo: <input type="password" name="heslo"><br />
             <input type="submit" value="Prihlasit">
        </form>

    </body>
</html>
```

V tomto prípade formulár odošle dáta z formulárových prvkov s atribútom `name`, takže v našom prípade to budú informácie mena a hesla, na URL adresu uvedenú atribútom `action` elementu `form` metódou *HTTP POST*.

HTTP požiadavky nie sú primárne zobraziteľné pre používateľa, ale je možné ich zobraziť napr. v internetovom prehliadači *Mozilla Firefox* tzv. *Firefox Network Monitor* klávesovou skratkou `control` + `shift` + `E`.

Na strane servera nejaký kód alebo skript príjme z URL adresy, v našom prípade `http://google.sk`, dáta vo forme zoznamu dvoch položiek slovníkového typu kľúč/hodnota, ktorý je zahrnutý v HTTP požiadavke. Spôsob akým spracujeme tieto hodnoty záleží na serverových programovacích jazykov (napr. PHP, Python, Java, C# a podobne).

Takýmto spôsobom funguje odosielanie požiadaviek na server a spracovávanie formulárových prvkov. Ako už bolo spomínané v úvode, existujú rôzne knižnice, frameworky, ktoré rozširujú funkcionalitu pôvodných HTML formulárových prvkov a samotný jazyk JavaScript posyktuje API, ktorý tento zámer umožňuje realizovať. Niektoré príklady použitia formulárových prvkov je možné vidieť napr. na knižnici *Bootstrap*.

# 4 UDALOSTI

Ak chceme vytvoriť nejakú reakciu na stlačenie tlačidla, musí existovať nejaký mechanizmus, ktorý dané kliknutie na tlačidlo klávesnice alebo myši zaregistruje. Podobne ako mozog dokáže registrovať rôzne podnety z nášho okolia. Od čias prvopočiatkov programovacích jazykov ubehol nejaký čas a v dnešnej dobe už existuje nespočetné množstvo obslúžnych knižníc, ktoré umožňujú zachytávanie udalostí výrazne zjednodušiť.

Udalosť v rámci programovania znamená, že každý komponent obsahuje zoznam udalostí, ktoré poskytuje. Tieto udalosti sú definované v podobe funkcií s osobitným názvom pre konkrétny typ očakávanej udalosti. Samotná udalosť je reakcia alebo správa na určitý stav komponenty.

Každý operačný systém musí obsahovať okrem iného aj systém na manipuláciu správ. V každom časovom okamihu operačný systém vysiela množstvo rôznych správ a zapisuje ich do nejakej fronty správ. Je to všeobecná fronta, teda zoznam, v ktorom sú registrované aktuálne prebiehajúce stavy systému, niečo podobné ako u vyšších živočíchov *centrálny nervový systém*, ktorý je prakticky neustále v pohotovostnom kontakte. Daný program, ktorý vytvárame obsahuje nejaký cyklus, nekonečný cyklus (neustály stav pohotovosti), ktorý selektuje správy podľa príslušnej nami zamýšľanej udalosti a predáva túto informáciu na ďalšiu obsluhu.

Takže, ak by sme chceli naprogramovať vo *Win32 API* v jazyku C/C++, teda použiť nízkoúrovňový mechanizmus napr. jednoduché okno s tlačidlom pre kliknutie s nejakým dialógovým oknom, tak by sme potrebovali pomerne veľké množstvo kódu.

```
#include <windows.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
#include <stdio.h>

#define BUTTON_IDENTIFIER 1

HINSTANCE ghInstance = NULL;
ATOM RegisterWindowClass(HINSTANCE hInstance, char *pszWindowClassName);
BOOL InitInstance(HINSTANCE, int, char *pszWindowClassName);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPTSTR    lpCmdLine,
int       nCmdShow)
{
//Step 1: Register a Window Class..
char *pszWindowClassName = "Button Event Class";
if (RegisterWindowClass(hInstance, pszWindowClassName) == 0)
{
char szMessage[1024];
sprintf(szMessage, "Last Error = %d", GetLastError());
OutputDebugString((LPCWSTR)szMessage);
```

```
return -1;
}

ghInstance = hInstance;

//Step 2: Create a Main Window..
if (!InitInstance(hInstance, nCmdShow, pszWindowClassName))
{
return -1;
}

//Step 3: Loop through the message
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
{
TranslateMessage(&msg);
DispatchMessage(&msg);
}
}

ATOM RegisterWindowClass(HINSTANCE hInstance, char *pszWindowClassName)
{
WNDCLASSEX objWndClassEx;

memset(&objWndClassEx, 0, sizeof(objWndClassEx));

objWndClassEx.cbSize = sizeof(WNDCLASSEX);
objWndClassEx.style = CS_HREDRAW | CS_VREDRAW;
objWndClassEx.lpfnWndProc = WndProc;
objWndClassEx.hInstance = hInstance;
objWndClassEx.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
objWndClassEx.hCursor = LoadCursor(NULL, IDC_ARROW);
objWndClassEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
objWndClassEx.lpszClassName = (LPCWSTR)pszWindowClassName;

return RegisterClassEx(&objWndClassEx);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow, char *pszWindowClassName)
{
int nWidth = 500;
int nHeight = 250;

int nScreenX = GetSystemMetrics(SM_CXSCREEN);
int nScreenY = GetSystemMetrics(SM_CYSCREEN);

int x = (nScreenX / 2) - (nWidth / 2);
```

```
int y = (nScreenY / 2) - (nHeight / 2);

//Create main Window
HWND hWnd = CreateWindowEx(NULL, (LPCWSTR)pszWindowClassName, L"Príklad udalosti pre tlačidlo", WS_O

if (NULL == hWnd)
{
char szMessage[1024];
sprintf(szMessage, "Last Error = %d", GetLastError());
OutputDebugString((LPCWSTR)szMessage);
return FALSE;
}

//Display window that we just created.
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
switch (message)
{
case WM_CREATE:
{
//Step 4: Create Button control
int nWidth = 120;
int nHeight = 30;
RECT rect;
GetClientRect(hWnd, &rect);
HWND hButtonWnd = CreateWindowEx(NULL, L"BUTTON", L"Klikni na mna!", WS_TABSTOP | WS_VISIBLE | WS_CH
(rect.right - rect.left - nWidth) / 2,
(rect.bottom - rect.top - nHeight) / 2,
nWidth, nHeight, hWnd, (HMENU)BUTTON_IDENTIFIER, ghInstance, NULL);
}
break;

case WM_COMMAND:
{
switch (LOWORD(wParam))
{
case BUTTON_IDENTIFIER:
{
//Step 5: User click on the button
if (HIWORD(wParam) == BN_CLICKED)
{
```

```
UINT nButton = (UINT) LOWORD(wParam) ;
HWND hButtonWnd = (HWND) lParam ;

char szMessage[1024] ;
sprintf(szMessage, "Ahoj svet z klikni na mna. Tlačidlo ID je = %d", nButton);
MessageBox(hWnd, (LPCWSTR)szMessage, L"Klikni na mna", MB_OK);
}
}
break;
}
}
break;

case WM_CLOSE:
DestroyWindow(hWnd);
break;

case WM_DESTROY:
PostQuitMessage(0);
break;

default:
return DefWindowProc(hWnd, message, wParam, lParam);
}

return 0;
}
```

Kód obsahuje inicializáciu, vytvorenie samotného okna, vytvorenie tlačidla a konkrétne adresy pre zachytenie jednotlivých udalostí pre okno v rámci jeho zatvorenia ako aj udalosti pre kliknutie na tlačidlo. V takomto nízkoúrovňovom programovaní sú takéto chyby úplne bežné. Konkrétne sa jedná o nesprávne pretypovanie typu `char*` na typ *LPCWSTR*, takže z toho dôvodu je správa chybná.

Pre všeobecné pohoršenie radšej prepíšeme odchytenie tlačidla len vyvolaním jednoduchého dialógového okna:

```
if (HIWORD(wParam) == BN_CLICKED)
{
MessageBox(NULL, L"Ahoj Win32 API !", L"Klikni na mňa", MB_OK);
}
```

Samozrejme, ak vynecháme kód pre vytvorenie a obsluhu okna a tlačidla, tak vo Win32 API je pomerne jednoduché napísať len dialógové okno bez zachytenia nejakých udalostí:

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
```

```
MessageBox(NULL, L"Ahoj svet :)", L"Môj prvý program", MB_OK);
return 0;
}
```

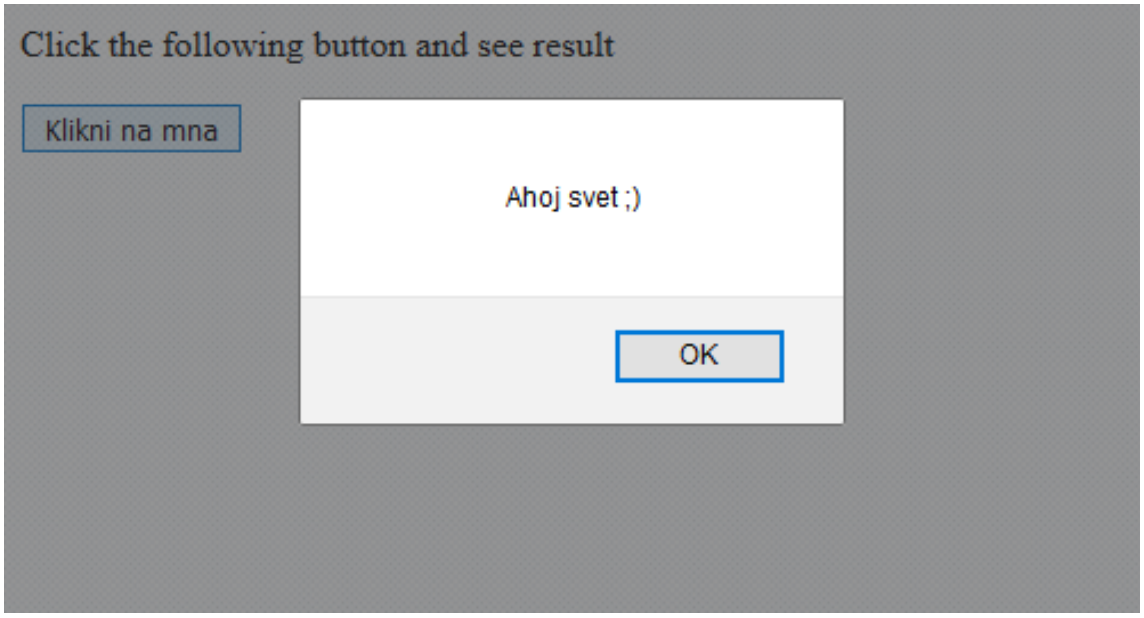
Takže takýmto pomerne komplikovaným spôsobom funguje odchytávanie udalostí v prostredí operačného systému Windows. V rámci Win32 API existuje určitá nadstavba, knižnica *MFC* (Microsoft Foundation Class), ktorá priamo vychádza z Win32 API a teda zjednodušuje programovanie, pretože dané konštrukcie sú prepísané do jednotlivých funkcií.

Podobne to funguje aj s JavaScriptom pre ktorý v dnešnej dobe už existujú rôzne nadstavby vo forme knižníc, ktoré zjednodušujú a zefektívňujú programovanie, napr. *Bootstrap*, *jQuery* alebo *AngularJS*. Aj keď samotné odchytávanie udalostí je výrazne jednoduchšie a to platí aj pre všetky ostatné jazyky.

Všeobecne povedané, udalosti nám poskytujú interakciu používateľa s daným aplikačným rozhraním, v našom prípade internetovým prehliadačom. Každý programovací jazyk obsahuje prezentačnú časť, ktorá je nejakým spôsobom prístupná v interakcii s používateľom. Tak ako v ostatných moderných programovacích jazykoch, tak aj v JavaScripte sa interakcia vykonáva prostredníctvom tzv. **udalosti**.

Udalosť sa najčastejšie pri vývoji webových stránok vykonáva nejakým externým zariadením, najčastejšie myšou a teda kliknutím na daný objekt v internetovom prehliadači.

```
<html>
   <head>
      <script>
         <!--
            function sayHello() {
               alert("Ahoj svet ;)")
            }
         //-->
      </script>
   </head>

   <body>
      <p>Click the following button and see result</p>

      <form>
         <input type="button" onclick="sayHello()" value="Klikni na mna" />
      </form>

   </body>
</html>
```




Podobne je možné napísať udalosti aj pre zachytenie pohybu myšou.

```
<html>
   <head>
      <script>
         <!--
            function over() {
               document.write ("Mouse Over");
            }

            function out() {
               document.write ("Mouse Out");
            }

         //-->
      </script>
   </head>

   <body>
      <p>Pohybom kurzora vo vymedzenom priestore sa zaznamená vzniknutá udalost’:</p>

      <div onmouseover="over()" onmouseout="out()">
 
 
 
      </div>
```

```
    </body>

</html>
```

Takýchto udalostí existuje pomerne veľké množstvo. Niektoré kompatibilné udalosti s verziou HTML5 sú napr. `onclick`, `onchange`, `ondblclick`, `onload` alebo `onmessage`.

# 5 DOCUMENT OBJECT MODEL (DOM)

Predchádzajúce časti pozostávajú zo základných princípov jazyka JavaScript, pomocou ktorých sme schopní riešiť základné programátorské úkony. Avšak problém spočíva v tom, že najskôr potrebujeme s danými objektmi v internetovom prehliadači a teda našej HTML stránke nejakým spôsobom manipulovať. A práve na tento účel slúži *DOM API* (Document Object Model API).

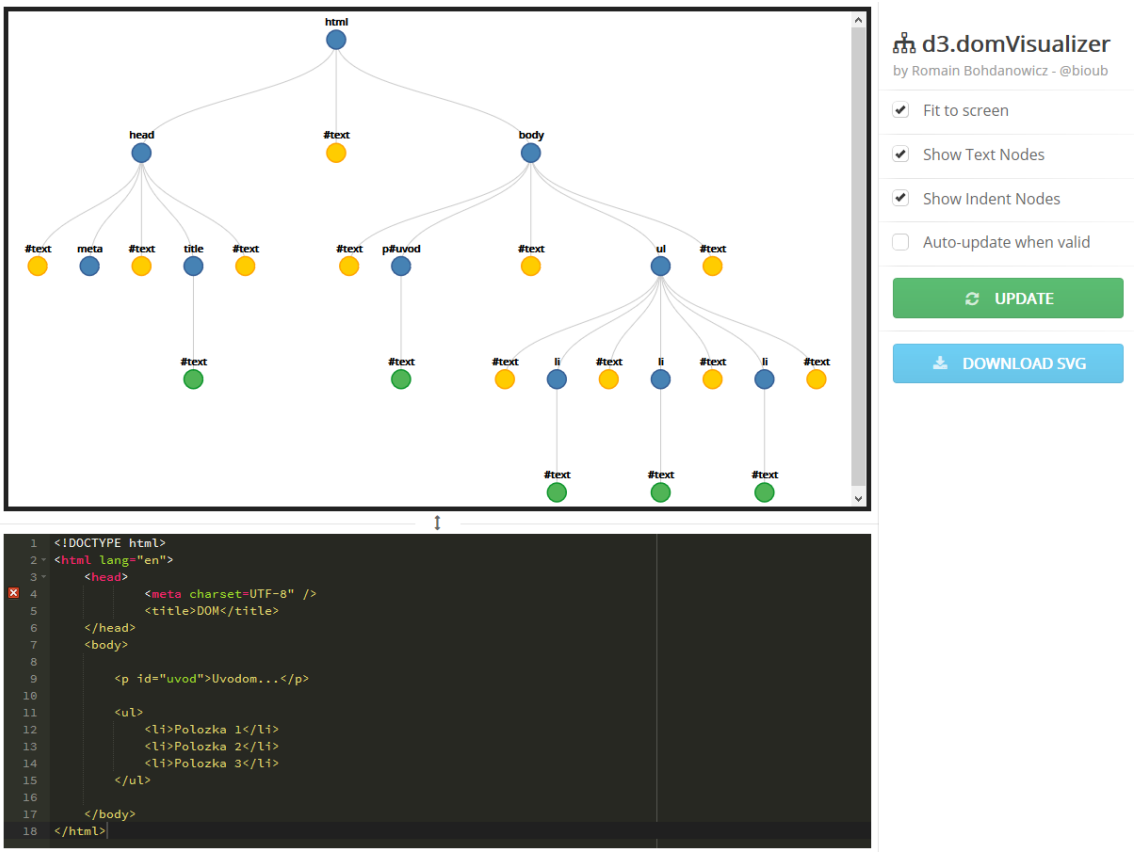
Jedná sa o knižnicu, ktorá poskytuje funkcionalitu pre interakciu s obsahom webovej stránky. Knižnica ponúka funkcie na prístup k webovým objektom, prehliadanie jednotlivých elementov a manipuláciu s objektmi HTML a XML. V zásade by bolo možné povedať, že akákoľvek zmena stavov elementov na webovej stránke podlieha použitiu tejto knižnice. HTML stránka, ako už bolo spomínané, pozostáva zo sémantickej časti a CSS štýly z prezentačnej časti. Význam HTML stránky definujeme jednotlivými elementmi, ktoré sú nejakým spôsobom usporiadané v hierarchii. Táto hierarchia tvorí určitú stromovú štruktúru.

Pomocou jednotlivých uzlov sme schopní identifikovať objekty v hierarchii. Z matematickej *teórie grafov* sa to podobá na grafové štruktúry, graf, ktorý definuje vrcholy a hrany podobne ako je to uvedené na obrázku.

Teda, každá HTML stránka má nejakú hierarchickú (stromovú) štruktúru. Vytvoríme si jednoduchú HTML stránku:

```
<!DOCTYPE html>
<html lang="en">
    <head>
            <meta charset=UTF-8" />
            <title>DOM</title>
    </head>
    <body>

        <p id="uvod">Uvodom...</p>

        <ul>
            <li>Polozka 1</li>
            <li>Polozka 2</li>
            <li>Polozka 3</li>
        </ul>

    </body>
</html>
```

A pre túto stránku je možné vytvoriť pomocou aplikácie *DOM Visualizer* hierarchickú štruktúru.




DOM model ponúka niekoľko spôsobov pomocou ktorých môžme pristupovať k objektom na webovej stránke:

- Prístup k elementu pomocou ID názvu
- Prístup k elementu pomocou názvu triedy (class)
- Prístup k elementu pomocou názvu elementu
- Prístup k elementu pomocou CSS selektora

Za týmto účelom si vytvoríme JavaScript program, ktorý bude pristupovať k elementu `p` **podľa názvu ID**.

```
<script>
var el = document.getElementById('uvod');
el.innerHTML = 'Zaverom...';
</script>
```

Týmto kódom vyhľadáme element, ktorý obsahuje ID s hodnotou `"uvod"`, čo predstavuje element `p` a obsah tohto nájdeného elementu pomocou funkcie `innerHTML` prepíšeme nejakým odlišným textom. Vyhľadávanie podľa názvu ID je pomerne bezpečná metóda.

Ďalšou metódou je vyhľadávanie pomocou názvu elementu. Analogickým spôsobom ako v prechádzajúcom prípade je možné napr. spočítať počet odrážok a vypísať ich v cykle.

```
<script>
var list = document.getElementsByTagName('li');

document.write(list.length);
document.write("<br />");

for (var i=0; i<list.length; i++)
{
document.write("li cislo " + (i+1));
document.write("<br />");
}
</script>
```

Vyhľadávanie podľa názvu triedy a CSS selektora je analogické.

Existujú aj ďalšie metódy prístupu a spôsobu reprezentácie elemetov, ktoré je možné nájsť napr. na stránke *w3schools.com*.

Aplikovaním princípov DOM je možné vytvoriť funkciu na vytvorenie tabuľky: (od *developer.mozilla.org*)

```
<!DOCTYPE html>
<html lang="en">
    <head>
            <meta charset=UTF-8" />
            <title>DOM</title>
    </head>
    <body>

<input type="button" value="Vytvor tabulku" onclick="generate_table()">

<script>
function generate_table()
{
  // get the reference for the body
  var body = document.getElementsByTagName("body")[0];

  // creates a <table> element and a <tbody> element
  var tbl = document.createElement("table");
  var tblBody = document.createElement("tbody");

  // creating all cells
  for (var i = 0; i < 2; i++) {
    // creates a table row
    var row = document.createElement("tr");

    for (var j = 0; j < 2; j++) {
```

```
      // Create a <td> element and a text node, make the text
      // node the contents of the <td>, and put the <td> at
      // the end of the table row
      var cell = document.createElement("td");
      var cellText = document.createTextNode("cell in row "+i+", column "+j);
      cell.appendChild(cellText);
      row.appendChild(cell);
    }

    // add the row to the end of the table body
    tblBody.appendChild(row);
  }

  // put the <tbody> in the <table>
  tbl.appendChild(tblBody);
  // appends <table> into <body>
  body.appendChild(tbl);
  // sets the border attribute of tbl to 2;
  tbl.setAttribute("border", "2");
}
</script>

    </body>
</html>
```

Takže takýmto spôsobom je možné pristupovať k objektom na webovej stránke a vytvárať tak dynamický webový obsah.