JavaScript pro starší a pokročilé

Opáčko

- interpret v každém prohlížeči
- neexistuje koncept tříd
- **primitivní datové typy**
  - číslo, bool, řetězec, null, undefined
  - předávané hodnotou

```
1. var a = 3;
2. var b = 0.1 + 0.2;
3. var c = "ahoj" + 'ahoj';
4. var e = null;
5. var f = undefined;
```

- **komplexní datové typy**
  - objekt = neuspořádaná množina dvojic (klíč, hodnota)
  - podobjekty: array, function, date, regexp
  - předávané odkazem

```
1. var a = {};                  /* prázdný objekt */
2. var b = {c:3, "d":"hi"};
3. var e = [a, b];              /* pole o dvou položkách */
4. var f = function() {};
5. var g = /^.*/;               /* regulární výraz */
```

- **základní syntaktické prvky**
  - volitelný středník
  - if, for, while, switch
  - type coercion (konverze operandů různého typu – integer/boolean apod.)

```
1. var a = 1 + "dva";
2. var b = {} + {};
3. if (0 == "") { /* ... */ }
4. if (undefined == null)  { /* ... */ }
5. if (undefined === null) { /* ... */ }
```

Objekty, funkce a pole

- objekt = neuspořádaná množina dvojic
- klíč je řetězec
- hodnota je cokoliv
- pole je také objekt
- funkce je také objekt

```
1. var add = function(a, b) { return a+b; }
2. add.c = 123;
3. add(add.c, add["c"]);
```

```
1. var pole1 = [3, 2, 1];
2. var pole2 = [];
3.
4. pole1.length == 3;
5. pole1[1]     == 2;
6. pole2.length == 0;
7.
8. pole2.push(pole1);
```

Pilíř 1: Uzávěry

- Jaký je obor platnosti proměnných?
- funkce lze definovat uvnitř jiné funkce
- proměnná existuje v rámci své funkce a všech jejich podfunkcí

```
1. var outer = function(a, b) {
2.     var inner = function(x) { return 2*x; }
3.     return a + inner(b);
4. }
```

- Co když použijeme proměnnou ve vnitřní funkci?
- … vznikne uzávěra

```
1. var outer = function(a, b) {
2.     var inner = function() { return 2*b; }
3.     return a + inner();
4. }
5.
6. var outer = function(a, b) {
7.     var inner = function() { return 2*b; }
8.     return inner;
9. }
```

Pilíř 2: Klíčové slovo „this“

- zásadní rozdíl oproti jiným jazykům – hodnota je určena při volání

```
1. var fun = function() { alert(this); }
2. fun(); /* ? */
3.
4. var obj1 = { fun:fun };
5. var obj2 = { fun:fun };
6.
7. obj1.fun(); /* this == obj1 */
8. obj2.fun(); /* this == obj2 */
9.
10. fun == obj1.fun == obj2.fun
```

- pokud je při volání před názvem funkce tečka, hodnota „this“ je objekt vlevo
- jinak je this globální jmenný prostor (změna v ES6)
- this lze explicitně určit

```
var fun = function() { alert(this); }

var obj1 = {};
var obj2 = {};

fun.call(obj1, arg1, ...);    /* this == obj1 */
fun.apply(obj2, [arg1, ...]); /* this == obj2 */
```

Pilíř 3: Prototype chain

- dva objekty je možno provázat

```
var obj1 = { klic:"hodnota" }; var obj2 = {};
obj2.vazba = obj1;
alert(obj2.vazba.klic); /* "hodnota" */
```

- existuje speciální neviditelná vazba, tzv. „**prototype link**“
    - používá se při přístupu k neexistujícím vlastnostem

```
var obj1 = { klic:"hodnota" };
var obj2 = Object.create(obj1);
alert(obj2.klic); /* "hodnota" */

obj2.__proto__ == obj1;
```

- říkáme, že jeden objekt je prototypem druhého
- druhý nabízí vše, co první (+ možná něco navíc)
- více prototype linků = **prototype chain**

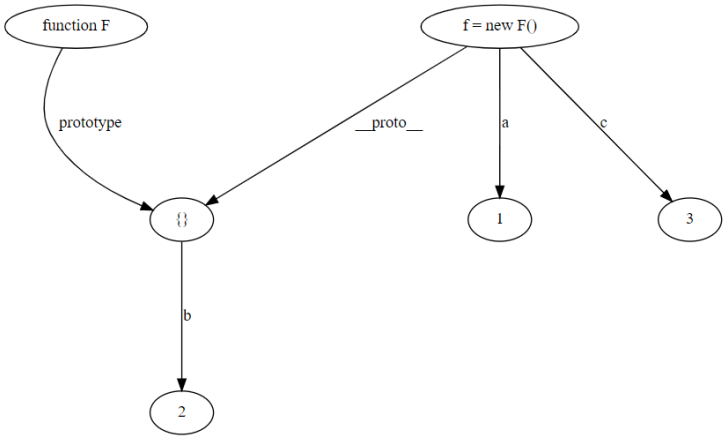
```
var obj1 = { klic:"hodnota" };
var obj2 = Object.create(obj1);
var obj3 = Object.create(obj2);

obj3.klic = "jina hodnota";
obj1.jinyKlic = "jeste jina hodnota";

alert(obj2.klic);     /* "hodnota" */
alert(obj3.klic);     /* "jina hodnota" */
alert(obj3.jinyKlic); /* "jeste jina hodnota" */
```

Operátor „new“

- neexistuje koncept tříd
- OOP lze realizovat pomocí prototypů
- vzor (třída) = rodičovský objekt
- objekt (instance) = objekty s prototypovým odkazem na rodiče

- prototype link nelze vytvářet přímo (x.__proto__)
- varianta 1: Object.create
- varianta 2: new
- každá funkce má (téměř prázdný) objekt **.prototype**
- zapás **new f** vytvoří nový objekt, nastaví mu **__proto__** na **f.prototype** a vykoná nad ním **f()**

```
1. var F = function() {
2.     this.a = 1;
3. }
4. F.prototype.b = 2;
5.
6. var f = new F();
7. f.c = 3;
```



- terminologický guláš – vlastnost **.prototype** vs. **__proto__**
- objekt („instance“) nemá příliš souvislost se svou vytvořující funkcí
- do prototypu funkce zpravidla vkládáme metody (neboť mají být sdíleny)

Dědičnost pomocí prototypů

```
1. var Parent = function() {}
2. Parent.prototype.hi = function() { return "hello"; }
3.
4. var Child = function() {}
5. Child.prototype = Object.create(Parent.prototype);
6. /* alternativně: Child.prototype = new Parent(); */
7.
8. var ch = new Child();
9. ch.hi(); /* "hello" */
```

+ diagram na konci souboru

Obohacování prototypů

```
1. String.prototype.lpad = function(what, length) {
2.   var count = length - this.length;
3.   var padding = "";
4.   for (var i=0; i<count; i++) { padding += what; }
5.   return padding + this;
6. }
```

Bind

```
1. var f1 = function() { alert(this.name); }
2. var foo = { name:"foo" }
3. var bar = { name:"bar" }
4. f1.call(foo); /* "foo" */
5.
6. var f2 = f1.bind(bar);
7. f2() /* "bar" */
```

- funkce/metoda **bind** vrací novou funkci
- v nové funkci je **this** napevno definováno ve chvíli volání **bind**

```
1. Function.prototype.bind = function(newThis) {
2.     var func = this;
3.     return function() {
4.         return func.apply(newThis, arguments);
5.     }
6. }
```

Využití

```
1. var Obj = function() {}
2. Obj.prototype.foo = function() {}
3.
4. var bar = new Obj();
5. setTimeout(bar.foo, 100);
6.
7. setTimeout(bar.foo.bind(bar), 100);
8.
9. /* dtto addEventListener atp. */
```