JavaScript očima programátora v2

Najít na internetu článek nebo dokonce seriál, který by se systematicky zabýval JavaScriptem, není nic jednoduchého - převažují články, které ukáží, jak deklarovat proměnné, v lepším případě funkce, poví něco o datových typech a tím to většinou končí. Navíc většina článků je mířena na použití JavaScriptu v rámci prohlížeče. Zřejmě i takovýto nedostatek informací vedl k tomu, že se z JavaScriptu stal jazyk, jenž je obestřen mnoha mýty, legendami a polopravdami, a mezi skutečnými programátory je značně neoblíben a/nebo nepochopen. Proto jsem se rozhodl sepsat tento článek, který představuje JavaScript z pohledu programátora, který je odkojen klasickými programovacími jazyky jako Pascal, C, C++, C# nebo Java. **Článek představuje JavaScript jako univerzální jazyk, bez jakéhokoliv zaměření na nějakou konkrétní oblast nasazení (např. prohlížeče).** Proč? Dnešní internet je plný rich internet aplikací, jejichž výkonnost je často limitována rychlostí JavaScriptu a proto bylo v posledních letech vynaloženo mnoho úsilí na to, aby JavaScript běhal co nejrychleji. Proto lze očekávat nasazení JavaScriptu i v non-browser úlohách, např. se nabízí použití JavaScriptu na server-side záležitosti - pak by mohla být celá RIA napsána v jednom jediném jazyce. Např. použití JavaScriptu v NoSql databázi CouchDB na psaní MapReduce funkcí je již realitou.

Úvod

Co je JavaScript?
JavaSript je jedním z dialektů standardizovaného jazyka ECMAScript. Další používané dialekty jsou ActionScript (bohatší syntaxe), JScript, InScript nebo QtScript. Ukázky v tomto článku jsou většinou otestovány v JavaScriptu verze 1.8, konkrétně implementaci SpiderMonkey, což je vůbec první implementace JavaScriptu a využívá ji např. Firefox (i když dnes v optimalizované verzi TraceMonkey). Ke SpiderMonkey je také vynikají dokumentace, kterou doporučuji strčit do záložek, pokud to myslíte s JavaScriptem vážně.
K dalšímu studiu jazyka doporučil přímo ECMAScript specifikaci, která je dostupná taky v praktické HTML verzi.
Článek popisuje i některé techniky, které se nehodí pro použití na webu, protože je ne všechny současné prohlížeče podporují. Před začátkem vývoje konzultujte minimální verze podporovaných prohlížečů s těmito pěknými tabulkami.

JavaScript je dynamický jazyk. Dynamičnost spočívá v tom, že nic nemá pevný typ (jde o slabě typovaný jazyk), objekty mohou během běhu programu měnit svoje atributy a je zde magická funkce *eval*, která spustí string.
JavaScript je funkcionální jazyk. Funkce lze do sebe libovolně vnořovat, přičemž vnější funkce tvoří uzávěr (closure) vnitřních funkcí.
JavaScript je objektově orientovaný, avšak beztřídní, jazyk - nemá třídy, jen objekty. Pro dědění se používají tzv. prototypy (JavaScript podporuje prototypovou dědičnost).
JavaScript neumožňuje explicitní uvolnění paměti - o řízení paměti se stará Garbagge Collector.

Datové typy (tak, jak je lze získat pomocí operátoru typeof):

- *number* - číslo celé (*117*, *015* (oktalově), *0x15* (hexa)) i desetinné (*3.14*, *3.14E-15*)
- *string* - textový řetězec, "*bla*" nebo '*bla*'
- *boolean* - *true* nebo *false*
- *undefined* - má jedinou hodnotu *undefined*
- Function - funkce
- *object* - objekt

Kromě funkcí a objektů jsou to všechno primitivní typy, ale umí se zaboxovat.
Při porovnání hodnot máme dvě možnosti:

- Buď můžeme použít klasické *!=* a ==, které v případě neshodnosti typů provede potřebné konverze. Takže např. *3* == "*3*" je *true*.
- Druhou možností je použití striktního porovnání *!==* a ===, které neprovádí žádné konverze, takže *3* === "*3*" je *false*.

Identifikátory v JavaScriptu jsou case sensitive (narozdíl třeba od ActionScriptu verze 1.0).

Funkce

Definice funkce může vypadat nějak takto:

```
1 function secti(a, b) {
```

```
    var c = a + b; // lokalni promenna
    a += c;
    return a + b + c;
}
```

Jedná se o slabě typovaný dynamický jazyk, takže typy parametrů ani návratové hodnoty se neuváději, vše se vyhodnocuje až za běhu. Pokud např. *a* a *b* budou stringy, bude se funkce chovat jinak než kdybychom použili čísla.
Při volání funkce můžeme zadat libovolný počet parametrů. Pokud nějaký definovaný parametr při volání neuvedeme, bude jeho hodnota *undefined*. Pokud jich bude více, můžeme se k nim dostat přes pole arguments, které obsahuje všechny předané parametry (má položku *length* a indexuje se klasicky od nuly hranatými závorkami). Všechny parametry se vždy předávají hodnotou, není zde žádná přímá podpora pro předání parametru odkazem.
Možnost neuvedení parametrů de facto znamená, že všechny parametry jsou volitelné. Když parametrům chceme přiřadit nějakou defaultní hodnotu, dělá se to často takto:

```
function secti(a, b) {
    a = a || 0; // pokud je a undefined, null nebo prazdny retezec, priradi se do nej nula
    b = b || 0;
    var c = a + b; // lokalni promenna
    a += c;
    return a + b + c;
}
```

Uvnitř funkce můžeme definovat lokální proměnné pomocí klíčového slůvka *var*, jak bylo ukázáno.

Výše ukázaný způsob zavedení funkce se nazývá function declaration (nebo také *function statement*) a můžeme ho poznat tak, že mezi *function* a otvírací závorkou je povinné jméno. Tento způsob zavedení funkce vytvoří funkci daného jména a automaticky ji přiřadí do proměnné stejného jména.
Jiným způsobem zavedení funkce je tzv. function expression (také *function operator*), který se pozná podle toho, že jméno funkce je v tomto případě nepovinné a zavedení funkce je součástí výrazu. To typicky znamená, že výsledek *function expression* přiřadíme do nějaké námi definované proměnné.

```
var secti = function(a, b) {
    return a + b;
};
```

Nyní vytváříme anonymní funkci, kterou přiřazujeme do proměnné *secti*. Jméno funkce můžeme uvést i v tomto případě, ale na toto jméno se můžeme odkazovat jen z těla této funkce - odnikud jinud není vidět (hodí se tak např. pro rekurzi).

```
var secti = function secti_blabla(a, b) { // jmeno funkce a promenne se nemusi shodovat
    return a + b; // jen tady je secti_blabla videt
};
```

Zjednodušeně řečeno můžeme říci, že *function statement* vytvoří funkci a automaticky i proměnnou stejného jména, naproti tomu *function expression* pouze vytvoří funkci a o uložení do nějaké proměnné se musíme postarat sami.

Každá funkce je zároveň objektem (konkrétně instancí třídy Function) a lze ji zavést (poslední, třetí, způsob) i takto: *new Function("a", "b", "return a + b;")*
Poslední parametr je string nesoucí tělo funkce, všechny předchozí parametry jsou názvy parametrů funkce.

Uzávěry

Funkce lze do sebe libovolně vnořovat (tj. v těle funkce můžeme definovat další funkci). Důležité je vědět, že uzávěr tvoří vždy celá funkce, ne blok příkazů uzavřený ve složených závorkách, jak je to běžné např. v C#. Tam bychom napsali kód v následujícím smyslu a vše by fungovalo.

```
for (var i = 0; i < poleObjektu.length; i++) {
  var item = poleObjektu[i];
  document.getElementById(item.id).onclick = function() {
    alert(item.name);
  }
}
```

V JavaScriptu toto ale fungovat nebude, protože uzávěr se tvoří vždy jen na úrovni celé funkce, takže to, že *item* definujeme uvnitř těla cyklu, nám nepomůže (v C# ano).
Od verze JavaScriptu 1.7 má náš problém snadné řešení - místo klíčového slova *var* použijeme nové klíčové slovo let, které vytváří uzávěr na nižší úrovni než funkce.
Pokud jsme odkázáni pracovat s nižší verzí JavaScriptu, nezbývá nám než zajistit vytvoření další úrovně uzávěru - voláním další funkce v těle cyklu:

```
function makeAlertFunction(message) {
    return function() { alert(message); } // vracime funkci s uzaverem na funkci makeAlertFunction
}
for (var i = 0; i < poleObjektu.length; i++) {
    var item = poleObjektu[i];
    document.getElementById(item.id).onclick = makeAlertFunction(item.name);
}
```

Objekty

Objekt je vlastně jen hash table, nese tedy dvojice *klíč* - *hodnota*. Hodnotou může být cokoliv, tedy i funkce. Položka se definuje prostým přiřazením: *obj.novaPolozka = 5;*. Pokud položka neexistuje, dostáváme při jejím čtení hodnotu *undefined*. K položkám se dá přistupovat dvěma způsoby:

```
obj.polozka = 5;
obj["polozka"] = 5;
```

Položku objektu lze i zrušit: *delete obj.polozka;*

Nyní bych rád udělal menší vsuvku a zmínil jak vypadá běhové prostředí (kontext, ve kterém náš program běží). Je to děsivě jednoduché - vše je objekt (i primitivní objekty se umí zaboxovat) a vždy se nacházíme v kontextu nějakého objektu (k němuž máme přístup pomocí všudypřítomného readonly *this*). Pokud jsme na top-level úrovni, ukazuje *this* na globální objekt (v prohlížečích je to window). Celý program v JavaScriptu je pak vlastně tělo globální funkce.

Zpět k objektům. Jak vytvořit nový objekt?

- Anonymní objekt snadno - do složených závorek uzavřeme dvojice *jmeno : hodnota* a oddělíme je čárkami. Hodnota může být cokoliv, klidně další objekt nebo funkce (už víme, že funkce je objekt, konkrétně instance třídy Function):

  ```
  var car = {
      name : "Honda",
      model : "Civic",
      owner : { name : "Jiri", surname : "Novak" },
      printMe : function() {
          return this.name + ' ' + this.model + ' owned by ' + this.owner.name + ' ' + this.owner.surname;
      }
  };
  ```

  Samozřejme můžeme vytvořit i prázdný objekt: *var empty = {};*
  Takovému způsobu zápisu objektu do složených závorek se říká object literal nebo object initializer a zápis je podobný jako v případě Jsonu. Ale rozdíly tam jsou, např. v Jsonu nemůžeme logicky použít funkce, protože slouží pouze k přenosu dat, v Jsonu jsou u stringů povoleny jen dvojité uvozovky atp...
  Pokud voláme funkci objektu (tedy metodu) *car.printMe();*, je před voláním metody do *this* přiřazeno *car*. Proto se metoda chová dle očekávání.
- Voláním funkce prefixované klíčovým slovem *new* - takové funkce je slušné pojmenovat s prvním písmenem velkým a říkat jim *konstrukční funkce* (*constructor functions*). Klíčovým slovem *new* řekneme, že chce vytvořit nový (prázdný) objekt a tento použít jako *this* uvnitř volání (konstrukční) funkce. Uvnitř funkce máme tedy k dispozici *this*, což je úplně prázdný objekt. Logickým cílem konstrukční funkce je tento objekt zinicializovat.

  ```
  function Car(carName, model) {
      this.name = carName;
      this.model = model;
      this.printMe = function() { // tohle neni nejefektivnejsi, stay tuned
          return this.name + ' ' + this.model;
      };
  ```

```
}
```

Na konstrukční funkce tudíž můžeme pohlížet podobně jako na třídy v jiných jazycích - je to totiž předpis, jak zkonstruovat konkrétní objekt. Pokud budu v dále textu mluvit o třídách, budu mluvit právě o konstrukčních funkcích.
Pokud zavoláme toto, vytvoří se nám objekt podle očekávání: *var c = new Car("Honda", "Civic");*. Co když ale na slovíčko *new* zapomeneme? Pak nedojde k vytvoření nového prázdného objektu a jeho přiřazení do *this*. Bez *new* se jedná o úplně normální volání funkce a tudíž uvnitř funkce dojde k přiřazením do aktuálního *this* objektu. Pokud se nám podaří takovou funkci zavolat bez *new* z top-level úrovně, nadefinujeme položky *name* a *model* a funkci *printMe* na globálním objektu.

Následuje komentovaný příklad, na kterém si ukážeme i něco nového, zajímavého a užitečného, takže nepřeskakovat ;-)

```
// vytvorime promennou Car, do ktere priradime funkci se tremi parametry
// velke pismenko na zacatku jmena funkce indikuje, ze se jedna o konstrukcni funkci a tudiz by mela byt volana pomoci new
var Car = function(name, model, manufactured) {
    // nadefinujeme dve polozky v aktualnim this objektu
    this.name = name || 'Honda'; // pokud nebylo jmeno specifikovano, pouzije se 'Honda'
    this.model = model || 'Civic';

    // jsme ve funkci, takze muzeme nadeklarovat lokalni promennou
    // ta bude zcela podle ocekavani viditelna jen z teto funkce a funkci vnorenych
    // <strong>o lokalnich promennych v konstrukcnich funkcich se casto mluvi jako o privatnich polozkach objektu</strong>
    // k takovym polozkam se pristupuje jen pres jejich jmeno, tedy "made", zadne "this.made"!
    var made = manufactured;

    // nadefinujeme polozku, do niz priradime funkci, tedy vytvorime metodu objektu this
    // uvnitr metody muzeme pouzit lokalni promennou made
    this.howOld = function(now) { return now - made; }

    // stejne tak ale vidime i parametry funkce
    // <strong>parametry funkce jsou taktez oznacovany za privatni polozky objektu</strong>
    // a ze jsme pouzili stejne jmeno metody jako predtim? neva, stara se prepise
    this.howOld = function(now) { return now - manufactured; }

    // <strong>dve vyse uvedene metody pristupovaly k privatnim polozkam</strong>
    // <strong>takove metody se nazyvaji privilegovane</strong>

    // lokalni promenna muze byt jakehokoliv typu, tedy klidne funkce
    // takze toto je privatni metoda
    // <strong>k privatnim polozkam pristupujeme primo jejich jmenem</strong>
    // <strong>neprivatni polozky objektu musime prefixovat "this."</strong>
    // polozky jsou vsechno, i funkce, takze volani metody musi byt vzdy necim prefixovano
    // (v pripade tehoz objektu prefixem "this.")
    var printInfo = function () { println(this.name + " from " + made); }

    // vyse uvedene se da zapsat zkracene takto (drobne rozdily v implementaci tam ale jsou)
    function printInfo2() { println(this.name + " from " + made); }

    // je dobrym zvykem u novych trid definovat metodu toString, ktera vraci popis objektu
    // toto je public metoda, neni privatni ani privilegovana
    // vsimneme si, ze do polozky toString neprirazuji nejakou anonymni funkci jako v predchozich ukazkach,
    // ale pojmenovanou funkci - to muze pomoci pri debugu pri prochazeni stacku
    this.toString = function toString () { return this.name };
}

// vytvorime Hondu Civic s neznamym datem vyroby
var hc = new Car();
```

```
hc.howOld(); // vrati Nan (takovy ciselny undefined)
hc.toString(); // vrati "Honda"
//hc.printInfo(); // skoncilo by chybou - v konstrukcni funkci nikde nevidim prirazeni do this.printInfo
```

Vestavěné objekty JavaScriptu

JavaScript obsahuje pár vestavěných tříd. Jednak to jsou třídy, které zjednodušeně řečeno reprezentují zaboxované hodnotové typy (Boolean, Number, String); o třídě Function už řeč byla; a dále tu máme třídy, které nám usnadní řešení některých specifických úkolů: Date, RegExp (pro jeho vytvoření je možné použít i speciální literály uzavřené do slešů, např. *var re = /ab+c/;* je to samé jako *var re = new RegExp("ab+c");*), Math (obsahuje různé užitečné matematické funkce).

Asi tou nejzajímavější třídou je ale Array, tedy pole, konkrétně "dynamické" pole. Má položku *length*, která vrací hodnotu nejvyššího indexu v poli + 1. Pokud přiřazujeme do objektu pole položky s celočíselnými klíči, strkají se do pole. Protože je pole ale zároveň objekt, můžeme do něj přiřazovat i položky s nečíselnými klíči, např. *pole.bla = "nazdar";*

```
var pole = []; // to same jako new Array();
var inicializovanePole = [ 1, "bla", { name: "ja" }, 8, 5.8 ];
pole[2] = "dva"; // pole.length == 3, pole[0] == undefined, pole[1] == undefined
pole["2"] = "dva"; // stejne jako predchozi
pole.push("dalsi"); // dana hodnota se umisti na konec pole
pole[pole.length] = "dalsi"; // lamerska verze predchoziho
var dalsi = pole.pop(); // vrati posledni prvek pole a prvek z pole vynda
pole.length = 1; // nastaveni velikosti pole
```

JavaScript obsahuje vedle vestavěných tříd také pár užitečných funkcí. To nejzajímavější je dozajisté eval, která umí spustit JavaScriptový kód uložený ve stringu.

this

Klíčové slovíčko this je v JavaScriptu obestřeno mnoha mýty, přitom jeho fungování je docela jednoduché. Na začátku programu je nastavené na globální objekt (v případě prohlížeče objekt window) a změnit se dá čtyřmi způsoby:

- Zavoláním funkce s klíčovým slovem *new*. Jak už bylo řečeno, v tomto případě dojde k vytvoření nového prázdného objektu a jeho přiřazení do *this*, tedy něco jako *this = {};*. Po návratu z (konstrukční) funkce je *this* vráceno na původní hodnotu.
- Volání metody. Viz příklad výše - *hc.howOld();*. Dojde k nastavení *this*u na *hc*, zavolání funkce *howOld* a obnovení původní hodnoty *this*u.
- Volání metody pomocí call objektu Function. Metodu můžeme zavolat takto: *hc.howOld.call(someOtherObject, par1, par2);*. Zde se stane něco podobného jako v předchozím případě, pouze *this* je nastaveno na námi zadanou hodnotu *someOtherObject*. Můžeme tedy volat metodu nad objektem, který danou metodu vůbec neobsahuje.
- Obdobně jako předchozí případ funguje apply. Rozdíl je pouze v tom, že nyní nepředáváme parametry jednotlivě, ale *apply* má dva parametry - první parametr je nová hodnota *this*u, druhý parametr je pole parametrů. Metoda *apply* se využívá např. tehdy, když chceme z funkce zavolat funkci se stejnými parametry jako má aktuální funkce - pak zavoláme *funkce.apply(this, arguments);* Jak bylo řečeno, arguments obsahuje pole aktuálních parametrů, takže tímto ho pouze přepošleme do jiné funkce.

Prototypy

Vše je objekt, tedy i funkce je objektem. To je vidět v předcházejícím odstavci - funkce (reprezentovaná objektem třídy Function) má metody *call* a *apply*. Dále má ještě metodu toString(), která vrací zdrojový kód funkce (pokud je k dispozici). Objekt Function má dále položku length, která udává počet deklarovaných parametrů, a hlavně má položku prototype. *prototype* obsahuje položky, které jsou společné pro všechny objekty vytvořené pomocí této (konstrukční) funkce.
V praxi to funguje takto: napíšeme třeba *var spz = hc.spz;*. JavaScript se nejprve podívá, zda má objekt *hc* nějakou položku s názvem *spz*. Pokud ano, prostě ji vrátí. Pokud ne, podívá se do objektu ***Car***.*prototype*, zda ten nemá položku položku *spz*. Pokud ji ani ten nemá, vrátí se *undefined*. Do objektu ***Car***.*prototype* se dívá proto, že *hc* bylo vytvořeno pomocí konstrukční funkce *Car*.
Důležité je si uvědomit, že *prototype* je pořád jen jeden, bez ohledu na to, kolik objektů jsme pomocí dané konstrukční funkce vytvořili - všechny používají ten samý *prototype*. *prototype* si tedy můžeme představit jako kontejner pro statické položky třídy.

Jak je to ale s přiřazením do takové položky? To probíhá jinak, tak bacha na to. Když uděláme *hc.spz* = '*neco*';, tak se JavaScript koukne, jestli existuje v objektu *hc* položka s názvem *spz*. Pokud ano, tak je její hodnota přepsána novou hodnotou. Pokud položka není nalezena, je v objektu vytvořena. **Při přiřazování se *prototype* neuplatní!**

```
01 var hc = new Car();
02 var sf = new Car("Skoda", "Fabia");
03
04 // zajistime, aby vsechny objekty vytvorene pomoci Car mely polozku spz
05 Car.prototype.spz = 'prvni';
06 println(hc.spz); // 'prvni'
07 println(sf.spz); // 'prvni'
08
09 // pri prirazeni se prototype neuplatnuje
10 hc.spz = 'druha';
11 println(Car.prototype.spz); // 'prvni'
12 println(hc.spz); // 'druha'
13 println(sf.spz); // 'prvni'
14
15 // samozrejme kdyz priradime do prototype...;)
16 Car.prototype.spz = 'treti';
17 println(Car.prototype.spz); // 'treti'
18 println(hc.spz); // 'druha'
19 println(sf.spz); // 'treti'
20
21 // oba objekty byly zkonstruovany stejnou constructor function
22 println(hc.constructor == sf.constructor); // true
23 // a byla to constructor function Car
24 println(hc.constructor == Car); // true
25
26 // zmena prototypu bez explicitniho vypsani jmena constructor function
27 hc.constructor.prototype.spz = 'ctvrta'; // stejne jako Car.prototype.spz =
   'ctvrta';
28 println(Car.prototype.spz); // 'ctvrta'
29 println(hc.spz); // 'druha'
30 println(sf.spz); // 'ctvrta'
31
32 // vyse uvedene neplati jen na stringy, ale na vse, tedy i na funkce/metody
```

Co je vhodné umístit do *prototype*? Jsou to položky, které se nemění, tzn. především metody. Když totiž umístíme metodu do *prototype*, tak se už při každém volání konstrukční funkce (tedy vytváření nové instance) nebude metoda znovu vytvářet a přiřazovat do *this*, ale místo toho bude existovat jen jednou v *prototype* - tudíž šetříme pamětí (viz poznámka o neefektivnosti v jednom z předchozích příkladů).
Ale pozor, do *prototype* můžeme dát jen ty metody, které nejsou privilegované, tedy ty, které si nesahají na nějakou lokální proměnnou nebo parametr konstrukční funkce. Ono to totiž ani nedává moc smysl.

```
1 var Car = function(name) {
2     // blbost!
3     this.constructor.prototype.testMethod = function() { return name; }
4 }
```

Abychom měli přístup k privátním položkám, musí být přiřazení do *prototype* umístěno v konstrukční funkci. To ale znamená, že vytvoření oné metody a přiřazení do *prototype* probíhá při každém vytvoření instance třídy *Car*, přičemž se do *prototype* přiřadí funkce, která má uzávěr posledního volání funkce *Car*! Tedy metoda *testMethod* by vždy vracela jméno poslední vytvořené instance.
Stačí zapamatovat si jednoduché pravidlo - nepřiřazovat do *prototype* v konstrukční funkci, ale až za ní.

Kdyby někdo ale výše popsané chování vyžadoval (což se může stát, člověk nikdy neví), tak bych zde upozornil na jednu věc (vyžaduje ale znalosti z další části článku, takže tuto poznámku zatím klidně přeskočte). Privilegované položky v *prototype* totiž nejsou enumerabilní, tj. pokud budeme projíždět všechny položky objektu cyklem *for-in*, nedostaneme se k privilegované metodě. A to může způsobit problémy při implementaci vícenásobné dědičnosti. Abychom se tomuto problému vyhli, nepřiřazujeme v ukázce do *Car.prototype*, ale do *this.constructor.prototype*, což bude např. v případě volání konstruktoru z poděděné třídy *BestCar* znamenat totéž co *BestCar.prototype*. Použitím konstrukce *this.constructor.prototype* tedy umožníme přiřazení do *prototype* aktuálního objektu a vyhneme se nutnosti vyenumerovat tuto položku při kopírování z *prototype* do *prototype*.

Kromě metod můžeme do prototypů umístit i konstanty, pokud chceme k těmto konstantám přistupovat přes instance. V opačném případě je vhodnější umístit konstanty přímo do objektu konstrukční funkce,

tedy např. *Car.MAX_SEATS = 10;*. V takovém případě musíme ale počítat s tím, že (pokud nepoužijeme nějaký trik) se k této konstantě nedostaneme přes jméno odvozené třídy, takže např. *BestCar.MAX_SEATS* bude *undefined*.

Prototypy podrobněji

Výše uvedený popis prototypů v JavaScriptu je trošku zjednodušený, avšak pro běžnou praxi dostatečný. Pro zájemce se mrkneme na prototypy podrobněji.
Specifikace ECMAScriptu mluví o vnitřní property každého objektu s názvem [[Prototype]], ke které se nedá nijak dostat (např. SpiderMonkey to ale umožňuje přes __proto__). A právě toto je ta property, přes kterou se skutečně provádí hledání položky objektu! Je užitečné znát tyto skutečnosti:

- Prototypové fallbackování při hledání properties se neděje přes property *prototype* (instance ani žádnou takovou property nemá), ale přes nepřístupnou property *[[Prototype]]* (kterou má každý objekt).
- Každá funkce (instance "třídy" Function) má property *prototype*, což je iniciálně prázdný objekt s položkou *constructor*, která vede zpět na onu funkci. Toto je jediná "standardní" property s názvem *constructor* v JavaScriptu.
- **Když vytvoříme instanci objektu podle nějaké konstrukční funkce (klasické *new*), tak se do *[[Prototype]]* nové instance zkopíruje property *prototype* použité konstrukční funkce.** Nová instance nemá žádnou property jménem *prototype*.
- *[[Prototype]]* sice přímo měnit nemůžeme (protože k ní nemáme většinou žádný přístup), ale můžeme využít skutečnosti z předchozího bodu a měnit její položky přes property *prototype* použité konstrukční funkce, např. přiřazovat do *Car.prototype.spz*. Pokud to chceme dělat dynamičtěji, může použít konstrukci *hc.constructor.prototype* (která ale při změně prototypu za běhu nemusí fungovat podle očekávání) nebo nejlépe použít funkci getPrototypeOf, která byla zavedena v ECMAScriptu 3.1 (JavaScriptu 1.8.1).
  Pokud tyto novější verze použít nemůžeme, ale v budoucnu se to pravděpodobně změní, můžeme použít ne-ideální vlastní implementaci této funkce, jak je popsána mj. v tomto článku (sekce "*Cross-Browser Implementation*").
- Když vytváříme funkci, tak tím vlastně vytváříme instanci třídy Function. Z předchozích bodů tedy vyplývá, že třeba *Car* má jako *[[Prototype]]* nastavenu hodnotu *Function.prototype*.
- Operátor instanceof vůbec nesahá na property *constructor*. Místo toho vezme property *prototype* konstrukční funkce (zadána jako pravý operand). Dále pak vezme *[[Prototype]]* levého operandu a dokud není *null*, tak při shodě vrací *true*, jinak postupuje přes *[[Prototype]]* dále chainem (který končí u prapředka Object, který má *[[Prototype]]* nastaven na *null*).
- Pokud tedy přiřadíme do property *prototype* konstrukční funkce **po** vytvoření instance, tak nám *instanceof* nebude fungovat dle očekávání.

Properties

Jak jsme si již řekli, položky objektu (properties) lze definovat prostým přiřazením, tedy *obj.polozka = hodnota;* či *obj["polozka"] = hodnota;*. V moderních JavaScriptových enginech (implementující tuto vlastnost ECMAScriptu5) je tu ale ještě další možnost - property můžeme definovat pomocí speciální funkce defineProperty, kterou si hned ukážeme na příkladu:

```
// prvni parametr je objekt, pro ktery property definujeme
// druhy parametr je jmeno property
// treti parametr je tzv. property descriptor, coz je anonymni objekt s predepsanymi properties
Object.defineProperty(Car.property, "someProperty", {
    value: 36, // hodnota
    writable: true, // zda je mozne do property priradit
    enumerable: true, // zda bude property videt pri enumerovani pomoci for cyklu
    configurable: false // zda se da property smazat pomoci delete
});
```

Je to tedy velmi podobné, jako kdybychom udělali *Car.property.someProperty = 36;*, pouze máme větší kontrolu nad vlastnostmi definované property. Takovémuto *property descriptoru* se říká *data descriptor*. Zajímavější jsou *accessor descriptory*:

```
Object.defineProperty(Car.property, "someProperty", {
    get: function() { return this.someValue; }, // getter (volitelny)
    set: function(value) { this.someValue = value; }, // setter (volitelny)
    //writable: true, // writable nema u accessor descriptoru smysl
    enumerable: true, // zda bude property videt pri enumerovani pomoci for cyklu
    configurable: false // zda se da property smazat pomoci delete
```

```
});
```

Buď *getter* nebo *setter* lze vynechat a tím docílíme read-only, resp. write-only property.
Pokud chceme získat *property descriptor* nějaké existující property, máme k dispozici funkci getOwnPropertyDescriptor.

Jak jsem psal, výše popsané je záležitost ECMAScriptu 5 a třeba ve webových prohlížečích je v současnosti možné toto použít pouze na DOM objekty v IE8+. Pokud potřebujeme nějaké takové chování už nyní, můžeme v prohlížečích Safari, Opeře, Firefoxu a Chrome, příp. JS enginech SpiderMonkey a TraceMonkey, použít následující konstrukce:

```
// definice na existujicim objektu
Car.prototype.__defineGetter__("someProperty", function() { return this.propValue; } );
Car.prototype.__defineSetter__("someProperty", function(value) { this.propValue = value; } );

// definice v ramci object initializeru
var obj2 = {
    get someProperty() { return this.propValue; },
    set someProperty(value) { this.propValue = value; }
};
```

Místo funkce *getOwnPropertyDescriptor* zde můžeme použít funkce __lookupGetter__ a __lookupSetter__.
A když jsme už u těch podtržítkových záležitostí, tak přihodím jednu třešničku, kterou umožňuje *SpiderMonkey*. Do *obj.__noSuchMethod__* můžeme přiřadit funkci, která očekává dva parametry. První je jméno funkce a druhý její parametry. Tato přiřazená funkce bude zavolána vždy, když někdo na objektu *obj* zavolá metodu, která není definována.

Tato stránka docela pěkně ukazuje, kde si člověk může dovolit použít jaké způsoby vytváření properties. K properties s gettery a/nebo settery bych ještě dodal, že jsou to de facto metody, takže pokud přistupují jen k public položkám (tj. přes *this*), patří do *prototype*.

Statements

Teď si dáme trošku oddech a mrkneme na statements. Ty jsou téměř stejné jako v C, takže je netřeba moc rozebírat - máme tu for, while, do-while, break, continue, if-else, switch...
Za zmínku snad stojí jen for-in. Ten nám totiž umožní iterovat přes všechna jména položek objektu:

```
for(var key in someObj) {
    println(key + ': ' + someObj[key]); // klic: hodnota
}
```

Pokud chceme iterovat přímo přes hodnoty položek objektu, můžeme přes for each-in:

```
for each (var v in someObj) {
    println(v);
}
```

Dědičnost

A to nejlepší nakonec - klasická třídní dědičnost není JavaScriptem přímo podporována, JavaScript umí prototypovou dědičnost (vzpomeňte na fallbackování při hledání properties popsané výše). JavaScript je ale natolik ohebný jazyk, že je možné klasickou třídní dědičnost různými způsoby simulovat. Zde bych rád ukázal některé používané přístupy a přidám i nějaké svoje nápady. Nejprve předvedu řešení, která jsou k vidění na internetu a nejsou správná, a popíšu, v čem je u nich problém - i některá špatná řešení může být důležité znát.
V následujících příkladech budu předpokládat, že *T1* je bázová třída a *T2* je třída odvozená od *T1*.

Často je k vidění následující konstrukce: *T2.prototype = new T1;*
Toto velmi jednoduché řešení jakž-takž funguje, ale má řadu nevýhod:

- Při této deklaraci dojde k vytvoření instance třídy *T1*, což nemusí být úplně ok (kdyby třeba *T1* alokovala nějaké zdroje apod.).
- *T2.prototype.constructor* bude nelogicky roven *T1*. Lze řešit přiřazením *T2.prototype.constructor = T2;*
- *T2* nezdědí z *T1* privátní položky, takže volání privilegovaných metod selže.

- Při vytváření instance *T2* se nevolá konstrukční funkce *T1*.
- V této podobě neumožňuje vícenásobnou dědičnost.

Výhodou tohoto řešení je jednoduchost zápisu.

Následující řešení je zajímavější (avšak stále ne moc dobré):

```
01 var T2 = function(cpar1, cpar2, cpar3) {
02     // vytvorime public polozku s nazvem parentClass, ktera ukazuje na constructor function predka
03     // tim dame do T2 metodu, ktera umi zkonstruovat objekt typu T1
04     this.parentClass = T1;
05     // zde udelame to, pred cim jsem varoval - zavolame constructor function bez "new"
06     // protoze se ale jedna o volani metody objektu T2, preda se do ni aktualni this
07     // takze tato trida bude zinicializovana jako T1
08     this.parentClass(cpar1, cpar2 + cpar3);
09     // nyni nasleduji T2 specific zalezitosti jako obvykle
10 }
```

Výhody:

- Informace o dědění je obsažena přímo v definici třídy.
- Můžeme volat konstruktor předka s libovolnými parametry.
- Privátní položky *T1* jsou správně zinicializovány a neperou se s položkami *T2*, neboť každé žijí v jiné *closure*.
- Nejsme omezeni na jednoho předka - můžeme volat více *constructor functions*.

Nevýhody:

- Nedědí se *prototype*.
- Zápis dědičnosti jsou dva příkazy.
- Zavádí se nová public položka objektu (v ukázce pojmenovaná *parentClass*).

U posledních dvou nevýhod jsem sám vymyslel (heč! :)) jak se jich zbavit. Jednoduše využijeme jedné z možností, jak protlačit vlastní *this* do funkce.

```
1 var T2 = function(cpar1, cpar2, cpar3) {
2     // zavolame constructor function T1, ale podstrcime mu aktualni this
3     T1.call(this, cpar1, cpar2 + cpar3);
4     // pokud maji T1 a T2 stejne parametry, lze zapis zkratit nasledovne
5     //T1.apply(this, arguments);
6     // nyni nasleduji T2 specific zalezitosti jako obvykle
7 }
```

Toto řešení nám krásně pořeší dědění instančních položek, ale stále nám zde zůstává problém s neděděním prototypu. Zde ukážu několik přístupů k dědění prototypu, resp. vytvoření správného prototype chainu - prostě aby se při hledání property v prototypech správně propadávalo do prototypů nadtříd.

- Zdánlivým cílem je, aby *T2.prototype* obsahovalo to samé, co *T1.prototype*, plus něco navíc. Řešení je tudíž naprosto přímočaré:

  ```
  1 T2.prototype = T1.prototype;
  2 // obnoveni konstruktoru
  3 T2.prototype.constructor = T2;
  4 // T2 specific prototype polozky
  ```

  Nevýhodou tohoto přístupu je to, že neumožňuje vícenásobnou dědičnost. Té se ale v praxi já osobně snažím vyhnout, protože to přináší další zlo, na které je třeba pamatovat - možné překrývání jmen položek jednotlivých bázových tříd.
  Další (poměrně zásadní) nevýhodou tohoto řešení je to, že když něco přidáme do *T2.prototype*, tak se nám to objeví i v *T1.prototype*.
- Pokud chceme přesto umožnit vícenásobnou dědičnost, postupoval bych následovně. Do všech funkcí si přidáme metodu *extend*, která nám překopíruje všechny (resp. všechny enumerabilní) položky daného prototypu do prototypu aktuálního objektu.

  ```
  01 // metoda na kopirovani z prototypu do prototypu
  02 // pokud nechceme vicenasobnou dedicnost, tak ji nepotrebujeme
  ```

```
Function.prototype.extend = function extend(b) {
    b = b.prototype;
    for (var i in b) {
        // SpiderMonkey/TraceMonkey specific
        var g = b.__lookupGetter__(i), s = b.__lookupSetter__(i);
        if (g || s) {
            if (g) this.prototype.__defineGetter__(i, g);
            if (s) this.prototype.__defineSetter__(i, s);
        } else {
            this.prototype[i] = b[i];
        }
    }
};
```

Místo přiřazení do prototypu pak stačí zavolat *T2.extend(T1);*.
Nevýhodou tohoto řešení je, že pokud přidáme něco dodatečně do *T1.prototype*, tak se nám změna neprojeví v *T2.prototype*. Pokud máme striktně oddělenu "implementaci" a "použití", tak to není problém, ale trošku se tím omezujeme v jinak krásně dynamickém prostředí JavaScriptu.
Na internetu je občas toto řešení s kopírováním properties použito i na implementaci jednoduché dědičnosti - to je ale zbytečné a je lepší použít některé řešení popsané níže.

- Abychom využili prototypovou dědičnost pro implementaci třídní dědičnosti naplno, potřebujeme dosáhnout toho, aby při **ne**nalezení property v *T2.prototype* došlo k hledání v *T1.prototype* (potřebujeme zřetězit prototypy). Takže cílem je, aby skrytá property *[[Prototype]]* objektu *T2.prototype* (tedy "*T2.prototype.[[Prototype]]*") byla rovna *T1.prototype*.

```
// metoda na zretezeni prototypu
Function.prototype.extend = function extend(child) {
    // pokud mame primy pristup k [[Property]] pres __proto__ (SpiderMonkey/TraceMonkey), je to jednoduche
    if (child.prototype.__proto__) {
        child.prototype.__proto__ = this.prototype;
    } else {
        // vytvorime si pomocnou konstrukcni funkci...
        var F = function() { };
        // ...ktera ma prototype nastaveno na "T1.prototype"
        F.prototype = this.prototype;
        // "T2.prototype.[[Prototype]]" bude odkazovat na F.prototype (=== T1.prototype)
        child.prototype = new F();
        // nastavime spravny constructor
        child.prototype.constructor = child;
    }
};

T1.extend(T2); // pouziti
```

Když můžeme *[[Prototype]]* přímo změnit, je implementace jednoduchoučká. Pokud ne, musíme si trošku pomoci.
Abychom zajistili, že "*T2.prototype.[[Prototype]]*" bude nastaveno na *T1.prototype*, musíme *T2.prototype* vytvořit nějakou funkcí, která má *prototype* rovno *T1.prototype*. A není nic jednoduššího, než si takovou pomocnou funkci vyrobit (*F*).

- Výše uvedené řešení je nejlepší, jaké v době psaní článku znám. Protože se ale může tato "best-practice" změnit, může být do budoucna výhodné abstrahovat způsob vytváření tříd, resp. hierarchie tříd, jak to ukazuje např. článek Daniela Steigerwalda.

Finální komentovaná ukázka dědičnosti

```
var Trida1 = function(cpar1, cpar2) {
    // instancni polozky
    this.field1 = cpar1 + cpar2;
    this.field2 = "hola";
    // privatni polozky
    var pfield1 = cpar2;
    var pfield2 = "hola hej";
    // privatni metoda
    var pmethod1 = function(par1) { println("Trida1.pmethod1 called"); };
    // metoda pristupujici k privatnim polozkam, tedy privilegovana
    this.method1 = function(par1) { println("Trida1.method1 called");
this.method2(); println(pfield1); };
```

```
    // property pristupujici k privatnim polozkam (SpiderMonkey/TraceMonkey specific syntax)
    this.__defineGetter__("prop1", function() { return pfield1; } );
    println("Trida1 contructor called");
};

// metoda (a prip. property) pracujici pouze nad public polozkami
Trida1.prototype.method2 = function(par1) { println("Trida1.method2 called"); };
// konstanty dostupne pres instance
Trida1.prototype.CONST1 = 1;
// konstanty dostupne pres jmeno tridy
Trida1.CONST2 = 2;

// podedime tridu
var Trida2 = function(cpar1, cpar2, cpar3) {
    // volame konstruktor predka se dvema parametry
    Trida1.call(this, cpar1, cpar2 + cpar3);
    // instancni polozka
    this.newfield1 = cpar1;
    // privatni polozka stejneho jmena jako privatni polozka v predkovi
    // diky ruznym closures zadny problem
    var pfield1 = cpar2;
    // property, ktera nam zpristupnuje privatni polozky pfield1 teto tridy (SpiderMonkey/TraceMonkey specific syntax)
    this.__defineGetter__("prop2", function() { return pfield1; } );
    println("Trida2 contructor called");
};

Trida1.extend(Trida2); // zretezime prototypy


// tridy nadefinovany, nyni je jdeme pouzivat
var t1  = new Trida1(58, 12);
var t2  = new Trida2(-28, -98, -5);
var t22 = new Trida2(59, 65, 158);
t1.method1();
t2.method1();
// cteni privatnich polozek stejneho jmena
println(t22.prop1);
println(t22.prop2);
```

Singleton

Jako bonus na závěr bych ukázal dvě možnosti, jak implementovat návrhový vzor singleton, který se často v JavaScriptu označuje jako module pattern.

```
// nadeklarujeme funkci, kterou hned zavolame
var Singleton1 = function() {
    // datove polozky musi byt jako lokalni promenne
    // prirazeni do this by znamenalo prirazeni do aktualniho objektu, coz muze byt ledasco
    var val1 = 1;
    var val2 = 2;
    var val3 = 3;

    // z funkce vracime anonymni objekt, ktery tvori verejny interface pro nas singleton
    return {
        prop1 : 2,
        funkce : function(par1, par2) {
            return par1 + par2;
        },
        get value1() { return val1; }, // SpiderMonkey/TraceMonkey specific syntax
    }
}(); // vsimnete si () - funkci ihned volame
```

Udělali jsme to, že jsme nadeklarovali funkci, ale nijak jsme ji nepojmenovali ani do ničeho nepřiřadili, ale hned po definici jsme ji zavolali. Funkce vrací anonymní objekt, který tvoří veřejný interface pro náš singleton a pouze přes něj můžeme přistupovat k privátním položkám singletonu. Ty jsou implementovány jako lokální proměnné oné nepojmenované funkce.

```
01 // nadeklarujeme konstrukcni funkci, kterou hned zavolame pomoci new
02 var Singleton2 = new function() {
03     // datove polozky mohou byt privatni i public
04     this.val1 = 1;
05     var val2 = 2;
06     var val3 = 3;
07
08     this.prop1 = 2;
09     this.funkce = function(par1, par2) {
10         return par1 + par2;
11     }
12     this.__defineGetter__("value1", function() { return val1; });
13 };
```

Toto řešení funguje tak, že nadeklarujeme konstrukční funkci a hned ji zavoláme pomocí *new*. Zde je úroveň "zatajení implementace" o něco nižší, neboť pomocí *Singleton2.constructor* se dostaneme k oné nepojmenované konstrukční funkci.
Rozdíl mezi těmito dvěma řešeními je spíše kosmetický a není problém vymyslet další způsoby implementace singletonu.

Shrnutí

- JavaScript je dynamický objektově orientovaný skriptovací jazyk s prototypovou dědičností
- všechno je objekt, včetně funkcí
- objekt je hash mapa (jméno -> hodnota)
- objekt můžeme vytvořit dvěma způsoby - zavoláním *new Trida* nebo pomocí *object literalu*: *{ name: "Ja", age : 27, }* (podobné jako Json)
- nějaký objekt je vždy aktuální - *this* (na top-level úrovni je to globální objekt, v prohlížečích window)
- aktuální objekt (*this*) můžeme změnit čtyřmi způsoby - zavoláním funkce na objektu (tedy metody), voláním *new Trida* a dále pomocí speciálních metod třídy Function: call a apply
- Každá funkce má přiřazen *prototype*, na který vede odkaz (přes "skrytou" property *[[Prototype]]*) ze všech objektů, které byly vytvořeny touto (konstrukční) funkcí.
- *prototype* se používá především na definování metod
- co bylo definované jako poslední (stejného jména), to platí
- tento článek nepostihuje všechny možnosti JavaScriptu, ty jsou daleko širší