KAPITOLA 3

Pole

V této kapitole:
- Tvorba pole
- Přidávání prvků do pole
- Načítání hodnot z pole
- Vnořená pole
- Co lze dělat s poli

V kapitole 2, „Proměnné", jsme si ukázali, jak ukládat data nejrůznějších typů do proměnných. Pokud ale máme vytvořit seznam úkolů, budeme muset pracovat s více prvky – budeme muset ukládat celou skupinu dat. S tím nám přijdou na pomoc pole.

Tvorba pole

Pole lze vytvořit dvěma způsoby:

```
var mePole = new Array();
```

nebo:

```
var mePole = [];
```

Zápis s dvojicí hranatých závorek `[]` nazýváme **literál pole** – výše uvedený pak představuje prázdné pole. Je kratší a bezpečnější než zápis `new Array()`, protože konstruktor `Array()` je možné přepsat, a tedy případně nahradit zákeřným kódem; například funkcí, která předstírá, že vytváří pole, ale ve skutečnosti odesílá data cizímu serveru na Internetu. Prostřednictvím literálu pole můžeme snadno vytvořit nové pole obsahující několik hodnot – kupříkladu následovně:

```
var mePole = [4, 8, 15, 16, 23, 42];
```

Pole se však neomezují pouze na čísla. Můžeme vytvářet rovněž pole s textovými řetězci:

```
var ovoce = ["jablko", "pomeranč", "hruška", "vinné hrozny"];
```

Navíc můžeme datové typy různě kombinovat:

```
var kramy = [1, "jablko", undefined, 42, "tanky", null, []];
```

Přidávání prvků do pole

Pole nemusíme naplnit všemi daty už ve chvíli, kdy ho vytváříme. Můžeme vytvořit prázdné pole a později do něj přidávat data, a to hned dvěma způsoby. Jedním z nich je použití indexového pořadí:

```
var mePole = [];

mePole[0] = "Ahoj";
mePole[1] = "světe";
```

Právě jsme vytvořili dvouprvkové pole `["Ahoj", "světe"]`. Pokud zapíšeme číslo `0` do hranatých závorek za názvem pole na levé straně přiřazovacího příkazu (vlevo od `=`), říkáme tím, že chceme uložit hodnotu vpravo od znaku `=` na pozici s indexovým pořadím `0`. V tomto případě ukládáme textový řetězec `"Ahoj"` na nultou pozici pole `mePole`. Obdobně vkládáme textový řetězec `"světe"` na pozici s indexovým pořadím `1`. Obrázek 3.1 ukazuje, jak toto pole vypadá.

0	"Ahoj"
1	"Světe"

Obrázek 3.1. Dvouprvkové pole

Číslování prvků polí začíná nulou. První prvek pole se tedy nachází na pozici `0`, druhý prvek na pozici `1`, třetí prvek na pozici `2` atd. Tento způsob číslování však může působit zmateně a výjimečně může vést ke vzniku chyb. Například se může stát, že se budete dotazovat na prvek s indexovým pořadím `1` v očekávání zisku prvního prvku pole, ale ve skutečnosti obdržíte druhý prvek.

Pole je však možné indexovat také textovými řetězci:

```
var mePole = [];

mePole["ovoce"] = "jablko";
mePole["vozidlo"] = "tank";
```

Tímto způsobem vytvoříme **asociativní pole**, do nějž ukládáme prvky pod jmény místo čísly. Asociativní pole ale není příliš dobré používat. Mnohem lepší je ukládat pojmenovaná data do objektů, o nichž si povíme více v příští kapitole.

Na začátku této části kapitoly jsme si řekli, že data můžeme přidávat do pole dvěma způsoby. Druhý způsob představuje metoda `push()`. Ta je užitečná v situacích, kdy chceme přidávat prvky na konec pole, ale nechceme počítat indexové pořadí posledního prvku v poli. V takovém případě jednoduše zavoláme tuto metodu:

```
mePole.push("ahoj");
```

Pomocí metody `push()` přidáváme nový prvek do pole. Pokud by se jednalo o prázdné pole, uložili bychom tím tato data na pozici `0`. Kdyby se jednalo o pole s 10 prvky, uložili bychom tato data na pozici `10` (protože indexové pořadí začíná nulou).

Načítání hodnot z pole

Načítání hodnot z pole je poměrně snadné. Jednoduše se odkážeme na prvek pole prostřednictvím jeho indexového pořadí a pole nám vrátí tento prvek:

```
var maHodnota,
    mePole = ["Ahoj", "světe", "já", "jsem", "pole"];

maHodnota = mePole[3]; // vrací "jsem"
```

Tentokrát jsme vytvořili pole s pěti prvky. Jelikož pole číslujeme od nuly, slovo `"Ahoj"` najdeme na pozici `0`, slovo `"světe"` na pozici `1`, slovo `"já"` na pozici `2` atd. Předáme-li poli `mePole` pořadí `3`, požadujeme čtvrtý prvek pole a tím je slovo `"jsem"`.

Vnořená pole

Občas se stane, že budete potřebovat uložit pole do pole. Někdy můžete potřebovat i více zanořování. Malé varování – ačkoliv je možné vnořovat do sebe více polí, neměli byste se nechat příliš unést, protože vyznat se ve více úrovních číselných pořadí může být poněkud obtížné.

```
var yusuf, snilci;

yusuf = [];
snilci = ["cobb", "arthur", "ariadne", "saito", "fischer"];
```

Prozatím načítáme hodnoty z pole `snilci` jednoduše – předáme mu pořadí, které zapíšeme mezi hranaté závorky za název tohoto pole. Kdybychom tedy chtěli načíst prvek `"cobb"`, použili bychom pořadí `0`:

```
var snilek = snilci[0]; // vrací "cobb"
```

Co kdybychom však pole `snilci` nepojmenovali, ale vložili ho do pole `yusuf`?

```
var yusuf;

yusuf = [["cobb", "arthur", "ariadne", "saito", "fischer"]];
```

Jak bychom načetli hodnotu `"cobb"` teď? Museli bychom napsat `yusuf[0][0]`. Uvnitř první dvojice hranatých závorek definujeme indexové pořadí pole, ze kterého budeme načítat textové hodnoty, a až uvnitř druhé dvojice závorek uvádíme indexové pořadí samotného textového řetězce. Prvek `"arthur"` bychom načetli zápisem `yusuf[0][1]`, prvek `"ariadne"` zápisem `yusuf[0][2]` atd.

Pojďme ještě o něco dále:

```
var skutecnost = ["yusuf", ["arthur", ["eames", ["cobb", "ariadne",
  "saito", "fischer"]]]];
```

V předchozím kódu jsme deklarovali pole s názvem `skutecnost`. To obsahuje textový řetězec `"yusuf"` a vnořené pole, které obsahuje textový řetězec `"arthur"` a další vnořené pole atd. Načítání hodnot z tohoto vícerozměrného pole by vypadalo následovně:

```
skutecnost[0]          // vrací "yusuf"
skutecnost[1][0]       // vrací "arthur"
skutecnost[1][1][0]    // vrací "eames"
skutecnost[1][1][1][0] // vrací "cobb"
skutecnost[1][1][1][1] // vrací "ariadne"
skutecnost[1][1][1][2] // vrací "saito"
skutecnost[1][1][1][3] // vrací "fischer"
```

Jak vidíte, neztratit se v takovém poli je opravdu těžké, a to zejména tehdy, když každé pole obsahuje několik prvků, z nichž některé jsou také poli.

Co lze dělat s poli

Protože jsme se seznámili se základními koncepcemi polí a dozvěděli jsme se, jak do nich ukládat data, ukažme si, co ještě můžeme s poli provádět. Existuje sedm modifikačních metod, s nimiž je možné upravovat obsah pole – `pop()`, `push()`, `reverse()`, `shift()`, `sort()`, `splice()` a `unshift()`. Jsou k dispozici rovněž čtyři přístupové metody, které nemění obsah pole, ale umožňují nám přistupovat k jeho obsahu – `concat()`, `join()`, `slice()` a `toString()`.

Poznámka: Podpora napříč webovými prohlížeči

Jelikož různé webové prohlížeče poskytují různé úrovně podpory jazyka JavaScript, musíte počítat s tím, že můžou nabízet jiné skupiny funkcí. Kupříkladu webové prohlížeče s implementací verze 1.6 (a vyšší) tohoto jazyka podporují také přístupové metody `indexOf()` a `lastIndexOf()`. Kromě toho budou podporovat rovněž následující iterační metody – `forEach()`, `every()`, `some()`, `filter()` a `map()`. Webové prohlížeče implementující jazyk JavaScript 1.8 (a vyšší) podporují navíc iterační metody `reduce()` a `reduceRight()`.

Dříve vývojáři webových prohlížečů přicházeli s novou verzí svého produktu jednou za několik měsíců, nebo dokonce let. V dnešní době však tento interval značně zkracují, a to zejména společnosti Google a Mozilla se svými prohlížeči Google Chrome a Firefox. Vlastní představu si jistě uděláte, pokud se dozvíte, že v době psaní této knihy byla aktuální verzí prohlížeče Internet Explorer verze 10, prohlížeče Safari verze 6, prohlížeče Firefox verze 21 a prohlížeče Chrome verze 28. Protože nové verze prohlížečů vycházejí tak rychle (a často probíhají aktualizace na pozadí při spuštění prohlížeče), je velmi těžké odhadnout, jaká verze podporuje jakou funkčnost. Pokud chcete zjistit, která verze webového prohlížeče podporuje konkrétní funkci, můžete použít například nástroj na adrese *http://caniuse.com/*. Do vyhledávacího pole napište název metody a obdržíte tabulku s informacemi o její podpoře napříč prohlížeči (a jejich verzemi).

Modifikační metody

Metoda pop()

Metodou `pop()` odstraňujeme poslední prvek z pole, přičemž tento prvek vracíme:

```
var ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

ukoly.pop(); // vrací "Jít na procházku se psem."
```

Metoda push()

Pomocí metody `push()` přidáváme prvek na konec pole a poté získáme novou délku tohoto pole (počet prvků pole):

```
var ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

ukoly.push("Nakrmit kočku."); // vrací 4
// Pole ukoly vypadá nyní takto:
// ["Zaplatit účet za telefon.",
//  "Napsat nejprodávanější román.",
//  "Jít na procházku se psem.",
//  "Nakrmit kočku."]
```

Metoda reverse()

Metoda `reverse()` převrací pořadí prvků v poli:

```
var ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

ukoly.reverse();
// Pole ukoly vypadá teď následovně:
// ["Jít na procházku se psem.",
//  "Napsat nejprodávanější román.",
//  "Zaplatit účet za telefon."]
```

Metoda shift()

Metoda `shift()` odstraňuje první prvek z pole a vrací ho:

```
var ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

ukoly.shift(); // vrací "Zaplatit účet za telefon."
// Pole ukoly vypadá nyní takto:
// ["Napsat nejprodávanější román."
//  "Jít na procházku se psem."]
```

Metoda sort()

Jak už název metody `sort()` napovídá, slouží k uspořádání pole ve vzestupném pořadí. Řadicí algoritmus je velmi jednoduchý. Bez ohledu na to, jestli seřazujeme textové řetězce nebo čísla, vše převádí na textové řetězce a poté porovnává. Když se pokusíme uspořádat pole `[3, 10, 1, 2]`, nezískáme uspořádání `[1, 2, 3, 10]`, ale uspořádání `[1, 10, 2, 3]`. Je tomu tak z toho důvodu, že v lexikálním pořadí (nebo také abecedním pořadí) předchází textový řetězec `"10"` textovému řetězci `"2"`, jelikož začíná znakem `"1"`.

Naštěstí můžeme metodě `sort()` předat vlastní srovnávací funkci:

```
pole.sort([srovnej]);
```

Tímto způsobem lze porovnávat prvky pole dle vlastních kritérií, aniž bychom je museli převádět na textové řetězce, pokud nechceme. Pomocí vlastní srovnávací funkce můžeme jednoduše obrátit pořadí prvků nebo volit jakkoliv komplikovaná pravidla řazení – například řadit podle předposledního písmena předposledního slova každého prvku, kdybychom chtěli.

Vlastní funkci `srovnej()` si popíšeme podrobně později v této knize, jakmile se seznámíme s funkcemi. Zatím se spokojíme s prostým řazením:

```
var ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

ukoly.sort(); // řadíme pole ve vzestupném pořadí
// Pole ukoly vypadá teď následovně:
// ["Jít na procházku se psem.",
//  "Napsat nejprodávanější román.",
//  "Zaplatit účet za telefon."]
```

Metoda splice()

```
pole.splice(indexovePoradi, kolik[, prvek1, ..., prvekN]);
```

Prostřednictvím metody `splice()` můžeme kompletně „rozpitvat“ pole, a to tak, že do něj můžeme současně přidávat a odebírat z něj prvky:

```
var ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

// Vrací "Napsat nejprodávanější román."
ukoly.splice(1, 1, "Ovládnout svět.");
// Pole ukoly vypadá nyní takto:
// ["Zaplatit účet za telefon.",
//  "Ovládnout svět.",
//  "Jít na procházku se psem."]
```

Ve výše uvedeném kódu jsme metodě `splice()` sdělili, že má začít indexovým pořadím `1`, na němž se nachází prvek `"Napsat nejprodávanější román."`, odstranit jeden prvek (tj. samotný prvek `"Napsat nejprodávanější román."`) a následně na jeho pozici vložit prvek `"Ovládnout svět."`. Podobně bychom mohli vložit více prvků zaráz:

```
var ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

// Vrací "Napsat nejprodávanější román."
ukol = ukoly.splice(1, 1, "Ovládnout svět.", "Přezout pneumatiky.",
  "Najmout údernou jednotku.");
// Pole ukoly vypadá teď následovně:
// ["Zaplatit účet za telefon.",
//  "Ovládnout svět.",
//  "Přezout pneumatiky.",
//  "Najmout údernou jednotku.",
//  "Jít na procházku se psem."]
```

V předchozím kódu jsou důležité dvě věci – prvek `"Napsat nejprodávanější román."` jsme nahradili třemi jinými prvky, přičemž tyto nové prvky posunuly prvek `"Jít na procházku se psem."`, aniž by ho přepsaly.

Při volání metody `splice()` nemusíme samozřejmě přidávat žádné nové prvky. Mohli bychom pouze odstranit prvky (a nechat si je vrátit):

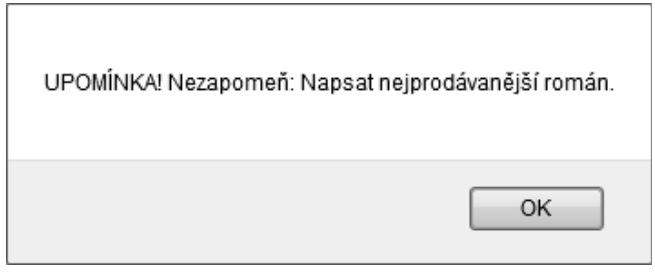
splice.html (úryvek)

```
var ukoly, ukol;

ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

ukol = ukoly.splice(1, 1); // Vrací "Napsat nejprodávanější román."
alert("UPOMÍNKA! Nezapomeň: " + ukol);
```

V tomto případě jen odstraňujeme prvek `"Napsat nejprodávanější román."`, jejž ukládáme do proměnné `ukol`. Posléze voláme funkci `alert()`, která vypíše zprávu `"UPOMÍNKA! Nezapomeň: Napsat nejprodávanější román."`, jak je patrné na obrázku 3.2.



Obrázek 3.2. Výstražná zpráva s upomínkou

Poznámka: Jak upoutat pozornost uživatele

Zobrazení výstražné zprávy funkcí **`alert()`** představuje základní způsob, jak upoutat pozornost uživatele. Stěží se jedná o elegantní řešení, ale je jednoduché a splňuje požadavky, a to zejména pro účely této knihy.

Uživatelé bohužel nemůžou pracovat se stránkou, zatímco se zobrazuje tato výstražná zpráva. Musí klepnout na tlačítko **OK**, aby tuto zprávu potvrdili.

Metoda unshift()

Metodou `unshift()` přidáváme jeden nebo více prvků na začátek pole a jako návratovou hodnotu obdržíme novou délku pole:

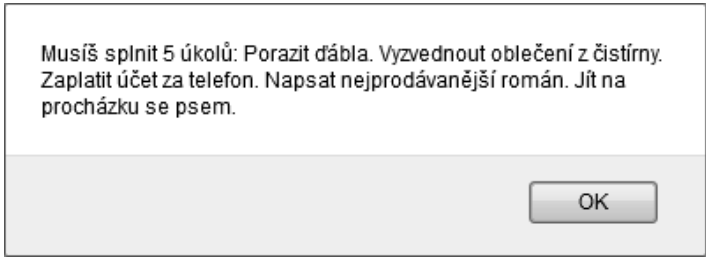
unshift.html (úryvek)

```
var ukoly, pocet;

ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];

pocet = ukoly.unshift("Porazit ďábla.",
  "Vyzvednout oblečení z čistírny.");
alert("Musíš splnit " + pocet + " úkolů: " + ukoly.join(" "));
```

Ve výše uvedeném zdrojovém kódu jsme přidali dva nové úkoly na začátek našeho seznamu, a to pomocí metody `unshift()`. Následně jsme si uložili nový počet prvků pole a sestavili zprávu pro uživatele. Výslednou zprávu lze vidět na obrázku 3.3.



Obrázek 3.3. Výstražná zpráva informující o úkolech

Pravděpodobně jste si všimli metody `join()`. O této metodě se více dozvíte za chvíli.

Přístupové metody

Metoda concat()

S metodou `concat()` je možné spojovat dvě nebo více polí do jednoho. Původní pole zůstanou netknutá. Tato metoda vrací nově sloučené pole se všemi hodnotami:

```
var pole1, pole2, pole3;

pole1 = ["Zaplatit účet za telefon."];
pole2 = ["Napsat nejprodávanější román."];
pole3 = ["Jít na procházku se psem."];
pole4 = pole1.concat(pole2, pole3);
// Pole pole4 obsahuje:
// ["Zaplatit účet za telefon.",
//  "Napsat nejprodávanější román.",
//  "Jít na procházku se psem."]
```

S pomocí metody `concat()` jsme do pole `pole4` uložili obsah prvních tří polí. Původní tři pole zůstanou beze změny – každé z nich stále obsahuje jediný prvek.

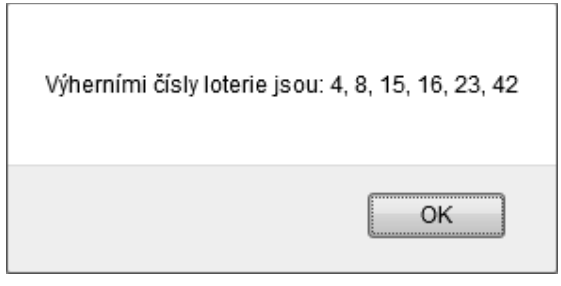
Metoda join()

Praktický příklad použití metody `join()` už jsme si ukázali, když jsme si popisovali metodu `unshift()`. Metoda `join()` spojuje prvky pole do textového řetězce. Můžeme jí předat volitelný argument, kterým určíme, jaké znaky (nebo znak) vloží mezi jednotlivé prvky. Pokud ho neuvedeme, tato metoda oddělí prvky čárkami. Na datových typech prvků pole nezáleží, protože metoda `join()` převádí všechny prvky na textové řetězce pomocí metody `toString()` (kterou si popíšeme za okamžik), jak lze vidět na obrázku 3.4:

join.html

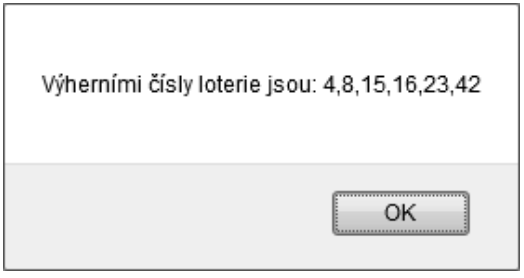
```
var cisla;

cisla = [4, 8, 15, 16, 23, 42];
alert("Výhernímí čísly loterie jsou: " + cisla.join(", "));
```



Obrázek 3.4. Výstražná zpráva s praktickou ukázkou metody join()

Možná se divíte, proč předáváte metodě `join()` čárku, když vkládá čárku automaticky. Povšimněte si však mezery za čárkou – kdybyste ji nepoužili, skončili byste se zprávou `"Výherními čísly loterie jsou: 4,8,15,16,23,42"`, jak je patrné na obrázku 3.5.



Obrázek 3.5. Výstražná zpráva předvádějící volání metody join() bez mezery

Metoda slice()

Metoda `slice()` kopíruje vybranou část pole a vrací ji. Neupravuje původní pole, ale vytváří jeho mělkou kopii. Této funkci říkáme, kde by měla začít, a případně ještě, kde by měla skončit. Volání `pole.slice(2)` tudíž vrací kopii pole `pole` počínaje indexovým pořadím `2` až do konce tohoto pole. Volání `pole.slice(-2)` naopak začíná od konce pole a vrací poslední dva prvky. Volání `pole.slice(2, 4)` kopíruje pole `pole` od indexového pořadí `2` po indexové pořadí `4`.

Poznámka: Mělká kopie

Všimli jste si, že metoda **`slice()`** vytváří **mělkou kopii**? Kdyby kupříkladu vaše pole obsahovalo další pole jako jeden ze svých prvků, metoda **`slice()`** by zkopírovala pouze odkaz na toto vnořené pole. Jinými slovy – veškeré změny provedené dodatečně na původním vnořeném poli by se projevily také v jeho kopii.

Vytvořme seznam úkolů s dceřiným seznamem úkolů týkajících se úklidu, poté ho zkopírujme a rozdělme na menší seznamy:

```
var ukoly, todo, uklid, ostatni;

ukoly = [
  "Pouštět papírového draka.",
  "Zachránit svět.",
  [
    "Uklidit koupelnu.",
    "Uklidit garáž.",
    "Vyčistit si hlavu."
  ]
];
todo = ukoly.slice(0); // kopíruje celé pole ukoly
uklid = ukoly.slice(-1); // kopíruje jen vnořené pole
ostatni = ukoly.slice(0, 2); // kopíruje jen první dva prvky
```

Prvním důležitým poznatkem z výše uvedeného kódu je, že třetím prvkem pole `ukoly` je pole. Již jsme si vysvětlili, že metoda `slice()` vytváří mělkou kopii takového pole. Tento příklad dokazuje, že se musíme mít na pozoru, když vytváříme kopii pole metodou `slice()`, jelikož ta kopíruje vnořená pole pouze odkazem. Jinými slovy – dceřiné pole se stále odkazuje na svůj originál, a pokud ten se změní, změní se také.

Metoda toString()

Metoda `toString()` vrací textovou reprezentaci prvků pole:

```
var pole = ["Tato", "slova", "jsou", "oddělená", "čárkami"];

pole.toString(); // vrací "Tato,slova,jsou,oddělená,čárkami"
```

Pokud pole obsahuje pouze samé textové řetězce, jako je tomu v předchozím příkladu, metoda `toString()` je jednoduše zřetězí do seznamu odděleného čárkami, který nakonec vrátí. Čísla před řetězením převádí na textové řetězce:

```
var pole = ["Těchto", 7, "slov", "a", "číslo", "jsou", "oddělené",
  "čárkami"];

pole.toString(); // vrací " Těchto,7,slov,a,číslo,jsou,oddělené,čárkami"
```

Mezi zobrazováním polí a objektů (o nichž se budeme bavit později) můžeme pozorovat rozdílné chování:

```
var pole = ["a", "b", "c", 100, 200, 300, [1, 2, 3], {"foo": "bar"}];

pole.toString(); // vrací "a,b,c,100,200,300,1,2,3,[object Object]"
```

Všimněte si, že metoda `toString()` zobrazuje vnořené pole `[1, 2, 3]` stejně jako jiné prvky, ale místo literálu objektu vypisuje pouze text `[object Object]`.

Metoda indexOf()

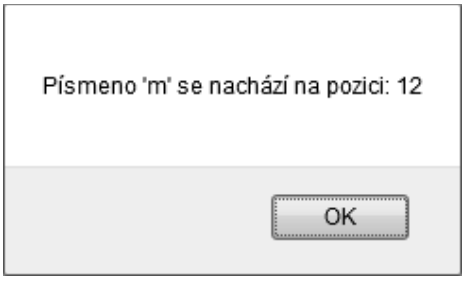
Metoda `indexOf()` hledá první výskyt daného prvku v poli a vrací jeho indexové pořadí. Hodnoty při hledání porovnává striktně, tedy pomocí operátoru `===`, a ne operátoru `==`. Zde je příklad:

```
pole.indexOf(hledanyPrvek[, pocatecniPoradi]);
```

V tomto případě hledáme v poli `pole` hodnotu `hledanyPrvek`. Když budeme vědět, že se tato hodnota nachází až za určitou pozicí, můžeme definovat také počáteční indexové pořadí, od kterého metoda `indexOf()` začne prohledávat dané pole. Výstup níže uvedeného kódu zobrazuje obrázek 3.6.

```
var abeceda;

abeceda = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
  "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"];
alert("Písmeno 'm' se nachází na pozici: " + abeceda.indexOf("m"));
```



Obrázek 3.6. Výstražná zpráva ukazující výsledek vyhledávání metodou indexOf()

Mohli bychom specifikovat, kde má metoda `indexOf()` začít hledat; například bychom zavolali `abeceda.indexOf("m", 10)`. V tomto případě by byl ale rozdíl zanedbatelný, protože pracujeme s malým počtem prvků. Ve velkých polích bychom však mohli vylepšit efektivitu vyhledávání, jelikož bychom prohledávali menší počet prvků.

Metoda lastIndexOf()

Metoda `lastIndexOf()` funguje podobně jako metoda `indexOf()`, ale pole prohledává od konce, a ne od začátku. Díky ní můžeme najít poslední výskyt hledaného prvku:

```
pole.lastIndexOf(hledanyPrvek[, pocatecniPoradi]);
```

Iterační metody

Metoda forEach() (JavaScript 1.6)

Kdybychom chtěli procházet všechny prvky pole, procházeli bychom běžně celé pole v cyklu. Přestože cykly se budeme zabývat v následující kapitole, měli bychom si předvést, jak se takové procházení dělá, aby bylo jasnější, proč je metoda `forEach()` užitečná.

V níže uvedeném programu použijeme tradiční způsob procházení pole v cyklu. Toto pole bude obsahovat několik čísel, přičemž v každém kroku cyklu přičteme aktuální číslo k celkovému součtu. Program ukončíme zobrazením celkového součtu, což bude v tomto případě číslo 108:

```
var pole, i, cislo, soucet;

pole = [4, 8, 15, 16, 23, 42];
soucet = 0;

for (i = 0; i < pole.length; i = i + 1) {
  cislo = pole[i];
  soucet = soucet + cislo;
}

alert("Součet činí: " + soucet);
```

Na obrázku 3.7 si můžete prohlédnout výsledek.

Obrázek 3.7. Výstražná zpráva informující o součtu čísel z pole

Ačkoliv existuje více způsobů, jak procházet pole v cyklu, cyklus `for` je z nich nejobvyklejší. Tento cyklus se skládá ze čtyř částí:

- inicializace (`i = 0`),
- podmínky (`i < pole.length`),
- konečného výrazu (`i = i + 1`),
- těla cyklu (`cislo = pole[i]...`).

Náš konkrétní cyklus zahajujeme přiřazením čísla `0` proměnné `i`. Pokud je hodnota proměnné `i` menší než délka pole, a to je, provede se jedenkrát tělo cyklu. Na konci těla cyklu se spustí konečný výraz `i = i + 1`, a tudíž se hodnota proměnné `i` navýší o `1`. Celý tento proces se opakuje, dokud je hodnota `i` menší než hodnota `pole.length` (tj. jakmile se tyto hodnoty začnou rovnat, náš cyklus skončí). Když jsme si popsali, jak funguje cyklus `for`, ukážeme si, jak můžeme zjednodušit náš program metodou `forEach()`:

forEach.html (úryvek)

```
var pole, soucet;

pole = [4, 8, 15, 16, 23, 42];
soucet = 0;

pole.forEach(function(cislo) {
  soucet = soucet + cislo;
});

alert("Součet činí: " + soucet);
```

Povšimněte si, že celý cyklus se nyní přeměnil na volání metody `foreEach()` pole `pole`. Této metodě předáváte anonymní funkci (o níž se dozvíte více později), která přijímá jako argument hodnotu aktuálního prvku a tu ukládá do proměnné `cislo`. Jinými slovy – všechna čísla, která

metoda `forEach()` zpracovává, ukládá do lokální proměnné `cislo`. Tělo této vnitřní funkce obsahuje pouze jednoduchý matematický výpočet a na konci celého kódu se nachází volání funkce `alert()`, stejně jako u předchozího příkladu.

Metoda map() (JavaScript 1.6)

Metoda `map()` se téměř shoduje s metodou `forEach()`. Jediný rozdíl spočívá v tom, že metoda `map()` vrací pole, jež obsahuje hodnoty, které vrátíme funkcí zpětného volání. V níže uvedeném příkladu počítáme pomocí metody `map()` druhé mocniny všech prvků pole `pole`. Výsledné pole ukládáme pod názvem `umocnene`:

map.html (úryvek)

```
var pole = [1, 2, 3, 4, 5];
var umocnene;

umocnene = pole.map(function(cislo) {
  return (cislo * cislo);
}); // pole umocnene obsahuje [1, 4, 9, 16, 25]
alert(umocnene);
```

Metoda every() (JavaScript 1.6)

Někdy se stává, že musíme ověřovat, zda data v poli vyhovují nějakým pravidlům. Mohli bychom, podobně jako ve výše uvedeném příkladu, provést tento test ručně pomocí tradičního cyklu `for` nebo metody `forEach()`. Můžeme ale také použít metodu `every()`, jež pro každý prvek z pole spouští jí předanou funkci zpětného volání. Metoda `every()` vrací hodnotu `true`, pokud jsou všechny hodnoty v poli platné, jinak vrací hodnotu `false`:

every.html (úryvek)

```
var pole, jePlatne;

pole = [1, 2, 3, 4, 5];
jePlatne = pole.every(function(cislo) {
  return (cislo < 10);
}); // jePlatne obsahuje true
```

V předchozím příkladu ověřujeme prostřednictvím metody `every()`, jestli jsou všechny prvky našeho pole menší než 10. Tato metoda prochází postupně celé pole, přičemž pro každý jeho prvek volá funkci, kterou zapíšeme mezi její kulaté závorky. Aktuální prvek pole najdeme v proměnné `cislo` a pomocí našeho příkazu (`return (cislo < 10);`) kontrolujeme, zda je toto číslo menší než 10, následně vracíme hodnotu `true` nebo `false` jako výsledek tohoto porovnání. Metoda `every()` sleduje odpovědi, které dostává z naší funkce, a pokud obdrží hodnotu `false`, okamžitě ukončuje celý cyklus a vrací hodnotu `false`. Návratovou hodnotu volání této metody ukládáme do proměnné `jePlatne`. V našem příkladu obsahuje proměn-

ná `jePlatne` hodnotu `true`. Kdybychom změnili porovnávání tak, že bychom ověřovali, jestli jsou čísla z pole vždy menší než číslo 3, proměnná `jePlatne` by obsahovala hodnotu `false`:

```
var pole, jePlatne;

pole = [1, 2, 3, 4, 5];
jePlatne = pole.every(function(cislo) {
  return (cislo < 3);
}); // jePlatne obsahuje false
```

Metoda some() (JavaScript 1.6)

Jestliže potřebujeme ověřit, zda alespoň jeden prvek pole, vyhovuje zvolenému testu, můžeme zvolit metodu `some()`. Metoda `some()` funguje na podobném principu jako metoda `every()`, ale vrací hodnotu `true`, když z testu na některý prvek pole obdrží hodnotu `true`:

some.html (úryvek)

```
var pole, jePlatne;

pole = [1, 2, 3, 4, 5];
jePlatne = pole.some(function(cislo) {
  return (cislo < 2);
}); // jePlatne obsahuje true
```

Přestože v tomto případě naše pole obsahuje jen jediné číslo menší než 2, proměnná `jePlatne` obsahuje hodnotu `true`.

Metoda filter() (JavaScript 1.6)

To, že můžeme ověřovat platnost podmínek na prvcích pole, je skvělé, ale co kdybychom chtěli vytvořit nové pole složené jen z platných prvků? K tomuto účelu slouží metoda `filter()`. Funguje podobně jako metody `every()` a `some()`, ale s tou výjimkou, že všechny prvky, které vyhoví kritériím, kopíruje do nového pole:

filter.html (úryvek)

```
var pole, filtrovane;

pole = [1, 2, 3, 4, 5, 6, 7, 8, 9];
filtrovane = pole.filter(function(cislo) {
  return (cislo < 5);
});
// pole filtrovane obsahuje [1, 2, 3, 4]
```

Metody reduce() a reduceRight() (JavaScript 1.8)

Občas potřebujeme provést matematický výpočet nad prvky pole a obdržet tak jedinou výslednou hodnotu; když kupříkladu počítáme součet hodnot pole. V takové situaci nám přijde vhod metoda `reduce()` (nebo metoda `reduceRight()`). Metoda `reduce()` prochází dané pole v cyklu a předává naší funkci zpětného volání jako argumenty předchozí a aktuální hodnotu (konkrétně – hodnotu spočítanou v předchozím kroku nebo hodnotu prvního prvku a hodnotu aktuálního prvku). Navíc předává funkci zpětného volání také aktuální indexové pořadí a odkaz na samotné pole, kdybychom tyto údaje náhodou potřebovali pro náš výpočet. V následujícím příkladu si ale vystačíme s argumenty `predchozi` a `aktualni`:

reduce.html (úryvek)

```
var pole, soucet;

pole = [1, 2, 3, 4, 5];
soucet = pole.reduce(function(predchozi, aktualni) {
  return predchozi + aktualni;
}); // soucet obsahuje 15
```

Metoda `reduce()` prochází pole podobně jako dřívější metody, ale s několika rozdíly. Protože tato metoda by jinak v prvních průchodu nemohla nic předat jako argument `predchozi`, předává rovnou první a druhý prvek jako argumenty `peedchozi` a `aktualni` (v tomto případě hodnoty `1` a `2`). V následujících krocích předává jako argument `predchozi` hodnotu, kterou jsme vrátili z naší funkce zpětného volání, a jako argument `aktualni` hodnotu dalšího prvku v poli. My jen sčítáme hodnoty `predchozi` a `aktuální`, čímž získáváme průběžný součet. Konečný výsledek je, že sečteme všechny hodnoty pole a obdržíme celkový součet `15`.

Metoda `reduceRight()` funguje stejně jako metoda `reduce()`, ale prochází pole v opačném pořadí. Jinými slovy – začíná na konci pole a postupuje směrem k jeho začátku.

Projekt

Naším cílem je vybudovat plně funkční program pro správu úkolů, ale jelikož jsme si doposud popisovali jen proměnné a pole, nemůžeme zatím psát mnoho kódu. V předchozí kapitole jsme uložili tři úkoly do tří samostatných proměnných. Uložme tudíž tyto tři úkoly do jediného pole s názvem `ukoly`:

projekt.js

```
var ukoly;

ukoly = [
  "Zaplatit účet za telefon.",
  "Napsat nejprodávanější román.",
  "Jít na procházku se psem."
];
```

Shrnutí

V této kapitole jste se seznámili s poli a mnoha metodami pro práci s nimi. V jazyce JavaScript jsou však pole ve skutečnosti objekty, takže abyste plně porozuměli polím, musíte porozumět také objektům. Nezoufejte – objektům se důkladně věnuje příští kapitola.