Tekijä: *Leo Leppänen*

Ohjelmointitekniikka JavaScript

Ryhmän *Leppänen* JavaScript-dokumentaatio
periodin II/2013 Ohjelmointitekniikka (JavaScript) -kurssia varten.

0. Johdanto

JavaScript on ohjelmointikielenä monitahoinen ja monimuotoinen. Se antaa ohjelmoijalle ilmaisuvoimiaisia ja yleiskäyttöisiä työkaluja, joilla tuottaa helposti ohjelmia, joiden tekeminen esimerkiksi Javalla olisi hankalampaa. Toisaalta tämä sama ilmaisuvoimaisuus ja anteeksiantavuus voi osoittautua kompastuskiveksi, mikäli työkalu on liiankin voimallinen ja koodi päätyy tekemään enemmän kuin mitä oli tarkoitus.

JavaScriptin monimuotoinen luonne onkin synnyttänyt moninaisia, suorastaan toistensa kanssa ristiriitaisia ohjelmointifilosofioita, joista on vaikea sanoa mikä on ehdottomasti kaikissa tilanteissa **paras**. Oletettavasti kukin näistä filosofioista on syntynyt jokin aiemman filosofian heikkouksien pohjalta.

Tämän dokumentin yleinen ohjenuora onkin **sisäinen yhtenäisyys**. Tällä tarkoitetaan ajatusta siitä, että mitä ohjelmointifilosofiaa projektissa ja/tai tiimissä päätetäänkin käyttää, mitä työkaluja halutaankaan hyödyntää ja miten kielen monimuotoisuuteen suhtaudutaankin, kaikkein tärkeintä on se, että samat päätökset pitävät läpi projektin. Muutoin hämmennyksen ja kaaoksen vaara kasvaa huomattavasti. Voidaan tehdä vertaus Picasson kubismiin ja Rembrandtin barokkityyliin, joista kumpaakaan ei voi sanoa objektiivisesti paremmaksi, mutta joita sekoittamalla ei saada - ainakaan helposti - mitään kovinkaan toimivaa aikaan.

Tästä huolimatta tämä dokumentti pyrkii antamaan esimerkin siitä, millainen tämä yhtäinen filosofia kielen käytön suhteen voisi olla. Yleinen teema on myös lähestymissuuntamme: tarkastelemme JavaScriptiä erityisesti Javan suhteen ja Javasta siirtyvän ohjelmoijan näkökulmasta.

Aloitamme JavaScript-matkamme tarkastelemalla kielen tyyppijärjestelmää, sen rajoituksia, sen rajoitusten kiertämistä ja sitä onko tällaiselle kiertämiselle tarvetta. Kielten syntaksin ollessa varsin samankaltaisia (molemmat kielet pohjaavat syntaksiltaa C-kieleen, sekä JavaScript edelleen Javaan), on tämä tyyppijärjestelmien ja tyypityksen eroavaisuus luultavasti ensimmäisiä asioita joita Javasta siirtyvä ohjelmoija huomaa. Typpityksen jälkeen tarkastelemme edelleen JavaScriptin ohjelmointiparadigmaa, sekä JavaScriptin oliota ja perintää, jotka kaikki eroavat huomattavasti Javasta.

1. Tyyppiturvallisuuden tavoittelua

Javan tiukkaan tyypitykseen verrattuna JavaScriptin dynaaminen tyypitys voi helposti tuntua "löysältä" tai "epämääräiseltä". Kaikki numeraalit ilmaistaan yhtenä tyyppinä, oli kyseessä sitten kokonaisluku tai liukuluku. Dynaaminen tyypitys kuitenkin mahdollistaa huomattavasti Javan funktioita monipuolisempien funktioiden kirjoittamisen, kun sama funktio voi käsitellä monimuotoista skaalaa syötteitä.

Javasta siirtyvää kehittäjää saattaa hyvinkin houkutella ajatus tyyppiturvallisuuden tavoittelusta kielen natiivia tasoa tarkemmalla tasolla. Yleisesti ottaen on kuitenkin pahasta tapella kielen paradigmaa vastaan ja pyrkiä tekemään "niin kuin muissakin kielissä". Tästä huolimatta on kuitenkin olemassa erityisiä käyttötapauksia, joissa kielen omaa tyypitystä tarkemmalle tyypitykselle on tarvetta, tai joissa siitä on vähintään hyötyä.

Monissa laskukaavoissa on voitava tehdä oletuksia syötteen tyypistä. Voidaan kuvitella esimerkiksi function `isPrime()`, joka kertoo onko syötteenä annettu luku alkuluku. Ajattelematon toteutus voi helposti testata onko syöte jaettavissa ilman jakojäännöstä millään syötettä pienemmällä (mutta yhtä suuremmalla) kokonaisluvulla. Syötteen ollessa murtoluku, palauttaa funktio tässä tapauksessa väistämättä `true`. Ohjelmoija voi joko vierittää vastuun käyttäjälle, tai vaihtoehtoisesti tarkastella syötettä ja murtoluvun tapauksessa esimerkiksi pyöristää sen lähimpään kokonaislukuun tai antaa virheilmoituksen. Näistä vastuun vierittäminen käyttäjälle on jossain määrin vastuutonta ja kaksi jälkimmäistä puolestaan vaativat funktioita syötteen tyypin tarkasteluun JavaScriptin natiivia tyypitystä tarkemmalla tasolla.

Saman kaltaisia tilanteita aiheuttavat muun muassa monet määrät: 1,33 purkkia maitoa; 2,5 lippua konserttiin ja 0,75 hotelliyötä ovat kaikki sinällään järkeviä asioita, eivätkä välttämättä sotke esimerkiksi laskun summan laskemisessa käytettävää matematiikkaa. Kyseiset määrät eivät kuitenkaan ole välttämättä **järkeviä** tai tosimaailmassa mahdollisia palvelun ylläpitäjän ja/tai käyttäjän kannalta.

Toisaalta tarkistuskirjaston tarpeellisuutta esimerkiksi laskennan kohdalla voidaan kyseenalaistaa myös sikäli, että JavaScript tarjoaa valmiin `Math`-kirjaston, josta löytyvät muun muassa funktiot `Math.ceil()` ja `Math.floor()`. Onko siis esimerkiksi hypoteettisen alkulukulaskurimme tapauksessa parempi käyttää koodia

```
if(!isInt(foo)) && isNumber(foo)) {
   foo = Math.ceil(foo);
   }
```

kun suurimmassa osasta tapauksista meille riittäisi varsin hyvin pelkkä `foo = Math.ceil(foo);` , sillä riippuen `if`-lausekkeen suorituksen kestosta jälkimmäinen versio saattaa olla jopa nopeampi.

Lisäksi JavaScriptin suhteellisesti kevyessä ja anteeksiantavassa tyypityksessä on hyvätkin puolensa. Ohjelmoijan ei tarvitse asettaa yhtä tarkkoja rajoitteita syötteelle, kuin monissa vahvemmin ja tarkemmin tyypitetyissä kielissä. Tämän ansiosta käyttäjä voi antaa ohjelmalle syötteitä, jotka ovat hänelle luonnollisempia. Esimerkiksi "tuotteen hinta" -kenttään voidaan syöttää "1.5", "1.5e" tai "1.5 euroa", ilman että `parseInt()`:n antama tulos muuttuu. Tämä myös vähentää käyttäjän syötteestä johtuvien virhetilanteiden määrää.

Yleisesti ottaen on kuitenkin hyvästä, että ohjelmoijalla on **mahdollisuus** tyypittää tarvittaessa kielen natiivitasoa tarkemmalla tasolla. Tarkastelkaamme siis seuraavaksi tapoja varmistaa syötteen tyyppi JavaScriptin natiivia tyypitystä tiukemmin.

1.1. Kokonaisluvut

Helpoin tapa tarkistaa numeron kokonaislukuisuus on tarkastella onko sillä desimaaliosaa, eli toisin sanoen onko sen jakojäännös nollasta eroava numerolla yksi jaettaessa.

Ennen tätä tarkastelua, tulee meidän kuitenkin varmistaa että syötteemme on numeerinen, sillä esimerkiksi merkkijonon `"123"` tapauksessa jakojäännös yhdellä jaettaessa on nolla. JavaScript tekee siis automaattisen tyyppimuunnoksen jakojäännösoperaattorin yhteydessä. Voisimme toteuttaa tämän tarkistuksella `typeof input === 'number'`. Tämä lähestymistapa kuitenkin törmää ongelmaan, mikäli syöte on luotu konstruktoria käyttäen:

```
> var a = new Number(1)
undefined
> a
Number {}
> a + a
2
> a - 1
0
> typeof a
"object"
> typeof a === 'number'
false
```

Tämä ilmiö koskee myös merkkijonoja.

Kiertääksemme tämän ilmiön, tulee meidän käyttää `typeof` operaattorin sijaan

funktiota `Object.prototype.toString.call(input)`, joka palauttaa sekä syötteillä `1` että`new Number(1)` merkkijonon `"[object Number]"`.

```
function isInteger(input) {
   return ((Object.prototype.toString.call(input) == '[object Number]') && (input % 1 ==
0));
   }
```

Huomattavaa on, toteutuksemme hyväksyy esim. syötteen `1e+1`, sillä se on "auki kirjoitettuna" `10`. Tämä on loogista, sillä luvun esitys ei vaikuta siihen kuuluuko luku kokonaislukujen joukkoon vai ei.

1.2. Luvut yleisesti

Tarkastelemme funktion `Object.prototype.toString.call()` tulostetta. Voimme suoraan hyväksyä kaiken, mille funktiokutsulle palauttaa `'[object Number]'`, sillä tällöin kyseessä on `Number` olion ilmentymä.

Kohdassa 1.1 funktiomme johti palautteeseen `false`, mikäli syöte oli `NaN` tai `Infinity`. Tämä on varsin loogista, sillä kumpikaan ko. arvoista ei ole kokonaisluku. Tällä kertaa kyseiset arvot ovat kuitenkin selvästi valideja numeerisia arvoja, sillä niillä voidaan suorittaa laskutoimituksia ja ne voivat olla muiden laskutoimitusten tuloksia. Tilanne, jossa kaksi itsessään numeerista arvoa voisi "laillisen" laskutoimituksen jälkeen palauttaa "laittoman" arvon ei ole kovin looginen.

```
function isNumeric(input) {
   return Object.prototype.toString.call(input) == '[object Number]';
   }
```

Esimerkkinä tilanteista joissa selvästi numeerisilla luvuilla tehdyt laskutoimitukset palauttavat `Infinity` tai `NaN` toimivat esim seuraavat:

```
> var a = 1e+308
undefined
> a
1e+308
> a+a
Infinity
> (a+a) / (a+a)
NaN
```

Toisaalta, koska voimme saada vastauksen `NaN` myös suorittamalla laittoman laskuoperaation, kuten `"foo" % 1`, voidaan perustellusti sanoa ettei NaN ole numeerinen arvo. Tällöin tulee yllä määriteltyyn funktioon liittää tarkistus `&& !isNaN(input)`. Samalla tavoin voidaan rajata `Infinity` pois hyväksytyistä syötteistä lisäämällä tarkistus `&& isFinite(input)`. Tällöin funktiomme olisi siis seuraava:

```
function isNumeric(input) {
   return ( Object.prototype.toString.call(input) == '[object Number]' &&
isFinite(input) && !isNaN(input));
   }
```

1.3. Merkkijonot

Merkkijonouden testaamisessa voimme edelleen hyödyntää 1.1:n ja 1.2:n `toString()`-pohjaista testausmenetelmää. Tarkistamme vain, että 1.2:n funktiokutsu palauttaa`'[object String]'`

```
function isString(input) {
   return (Object.prototype.toString.call(input) == '[object String]');
   }
```

1.4. Totuusarvot

Totuusarvojen tyypittäminen JavaScriptin natiivitasoa tiukemmin ei vaikuta kovin hyvältä idealta ottaen huomioon yhteistoiminnan kolmannen osapuolen kirjastojen kanssa. Voidaan esimerkiksi olettaa jokin kolmannen osapuolen funktio, joka palauttaa kahden luvun erotuksen itseisarvon lisättynä yhdellä. Ts. `| a - b | + 1`. Tällöin funktio palauttaa**käytännössä** totuusarvon `true`, mikäli `a` ja `b` ovat saman arvoiset, sekä `false` mikäli ne ovat eriarvoiset. Hyväksymällä koodissamme vain tiukat `boolean`-muotoiset syötteet, ym. funktion palaute ei kuitenkaan kelpaisi omille funktioillemme.

Onkin vaikea keksiä tapausta jossa JavaScriptin natiivitasoa tiukempi totuusarvojen määrittely olisi millään konkreettisella tasolla **hyödyllistä**. Lisäksi on helppo visioida tilanteita joissa tällainen ylimääräinen tarkistelu aiheuttaa haittaa.

Voidaan tehdä (hiukan yliampuva) vertaus ohjelmoijasta joka rajoittaa ohjelmointikielensä `Integer` muuttujan lukualueeseen -32,768 - 32,767 tai 0 – 255.

Jos tällaista tarkastelua kuitenkin välttämättä halutaan tehdä, lienee helpointa toteuttaa se aiempien kohtien tavoin `toString()` tulosteen tarkastelulla, seuraavasti:

```
function isBoolean(input) {
   return Object.prototype.toString.call(input);
   }
```

Mielenkiintoisempaa totuusarvojen osalta on JavaScriptin toiminnan eroavaisuus merkkijonon totuuden ja merkkijonosta rakennetun boolean-olion totuuden välillä:

```
> "true" == true
false
> "true" == false
false
>new Boolean("true") == true
true
>new Boolean("false") == false
true

> 1 == true
true
> 0 == true
false
> 15 == true
false
>new Boolean(1) == true
true
>new Boolean(0) == false
false
>new Boolean(15) == false
true
```

Vaikuttaa siltä, että literaalien vertailussa kaikki paitsi tiukasti tosiset arvot epätosia, kun taas luotaessa `Boolean`-oliota literaalin pohjalta, kaikki paitsi tiukasti epätodet arvot ovat tosia.

Toisien sanoen `new Boolean()` on epätosinen vain jos syötteenä on `0`, `-0`, `null`, `false`, `NaN`, `undefined`, tyhjä merkkijono `""` tai syötettä ei ole . Lisäksi `Boolean` käyttäytyy muutenkin hyvin erikoisesti verrattuna literaaleihin `true` ja `false`, katso esimerkiksi MDN: Boolean.

Tätä tosisuuden ja epätosisuuden määrittelyn eroavaisuutta voitanee hyödyntää joissain yhteyksissä, mikäli halutaan tarkemmin rajoittaa totisiksi arvoiksi hyväksyttävien literaalien joukkoa.

1.5. Kokonaislukutaulukot

Jos oletamme että käytössä on kohdassa 1.1 määritelty `isInteger()`-funktio, voimme käyttää JavaScriptin `Array.every()`-funktiota, joka palauttaa `true` vain jos taulukon kaikki alkiot läpäisevät funktiolle parametrina annetun testin.

Tätä ennen meidän tulee kuitenkin tarkistaa, että tarkasteltava kohde on taulukko. Tämä onnistuu helpoiten aiemmin käytetyllä `toString`-funktioon pohjaavalla metodilla.

```
function isIntegerArray(input) {
    if (Object.prototype.toString.call(input) != '[object Array]'){
        return false;
    }
    return input.every(isInteger);
}
```

1.6. Lukutaulukot

Tämä tarkistus on triviaali muutos kohdassa 1.5 esitettyyn tarkistukseen. Vaihdamme vain `isInteger`-funktion `isNumeric`-funktioon.

```
function isIntegerArray(input) {
    if (Object.prototype.toString.call(input) != '[object Array]'){
        return false;
    }
    return input.every(isNumeric);
}
```

1.7. Taulukot yleisesti

Ylempänä esitetyt `isIntegerArray()` ja `isNumberArray()` voidaan helposti yleistää funktioksi `isArrayOf(test, array)`, joka tarkistaa että annettu syöte on taulukko ja että kaikki se alkiot toteuttavat `test()`-funktion ehdon.

```
function isArrayOf(test, input) {
    if (Object.prototype.toString.call(input) != '[object Array]'){
        return false;
    }
    return input.every(test); }
```

Tällöin `isIntegerArray()` ja `isNumberArray()` voidaan toteuttaa helpommin seuraavasti:

```
function isIntegerArray(arr) {
    return isArrayOf(isInteger, arr);
}

function isNumberArray(arr) {
    return isArrayOf(isNumber, arr);
}
```

2. Javascriptin filosofiaa

Kuten yllä jo ohimennen mainittiin, JavaScript-ohjelmoinnissa - samoin kuin kaikessa muussakin ohjelmoinnissa - tulee ensisijaisesti käyttää hyväkseen kielen rakenteita ja mahdollisuuksia niitä vastaan tappelemisen sijaan. Taistelemalla jokaisessa vaiheessa kielen perusluonnetta vastaan aiheuttaa vain itselleen ja ennen kaikkea muille ohjelmoijille tuskaa ja kyyneleitä.

Tämän ajatusmallin oikeellisuus on varsin selvää puhuttaessa tyypeistä: ei ole mitään järkeä luoda omaa oliohin pohjaavaa ja esimerkiksi Javaa imitoiva tyyppijärjestelmää JavaScriptin luonnollisen tyypityksen rinnalle. Samoin on itsestäänselvää, että Haskellin ja Clojuren kaltaisten funktionaalisten ohjelmointikielten vääntäminen imperatiiviseen suuntaan on hölmöä. Tietysti kaikissa säännöissä on poikkeuksia, mutta yleisellä tasolla on varsin selvää, että lause "Älä taistele kielen paradigmaa vastaan" on hyvä ja järkevä ohje. Jos ohjelmoija löytää itsensä jatkuvasti "uimasta vastavirtaan", olisi hyvä hetkeksi pohtimaan olisiko sillä hetkellä käytössä joko väärä kieli tai väärä ohjelmoija.

Yllä esitetty ajatus tuskin aiheuttaa kummempia vastalauseita. Hankalammaksi tilanne menee, kun pohditaan kieltä jonka paradigma ei ole yhtä selvästi määritelty kuin esim. Haskellin tai FORTRANin kohdalla.

Mitä ohjelmointiparadigmaa tulisi noudattaa kielessä jossa on oliota mutta ei luokkia; kielessä joka ainakin periaatteellisella tasolla mahdollistaa funktionaalisen katsannon, mutta joka ei silti ole puhtaan funktionaalinen, ainakaan tiettyjen määritelmien mukaan?

2.1 Syleile tyypittömyyttä

JavaScriptin tyypittömyys on osa kieltä ja sen työkalupakkia. Ohjelmoijalla on käytännössä kolme mahdollista lähestymistapaa sen suhteen: Sitä vastaan taisteleminen, sen huomiotta jättäminen ja sen syleileminen.

Vaikka JavaScript olisikin mahdollista pakottaa tiukempaan tyypitykseen esimerkiksi itse tehtyjen olioiden avulla, voidaan kysyä "mitä hyötyä tästä on?". Ohjelmoija tuo vain kielen päälle lisää kompleksisuutta ja uuden abstraktiotason. Tästä seuraa vääjäämättä vain bugeja sekä huolia ohjelman tuleville kehittäjille, jotka joutuvat omaksumaan lisätyypityksen tekijän omat rakenteet ja käytänteet.

Toisaalta kielen tyypittömyyden voi jättää mahdollisuuksien mukaan huomiotta. Ohjelmoija voi ohjelmoida täysin samaan tapaan kuin esimerkiksi Javalla, luoden funktioita jotka palauttavat aina

numeerisia arvoja ja jotka saavat syötteekseen vain kokonaislukutaulukoita, jättäen funktion käyttäjien ongelmaksi huolehtia syötteen oikeellisuudesta. Tämän lähestymistavan ongelma on siinä, että vaikka kieltä ei pyritäkään aktiivisesti muuttamaan, on seurauksena ohjelmointityyli jossa kielen ominaisuuksia hyödynnetä: Mitä hyötyä siis tässä tapauksessa on juuri JavaScriptillä ohjelmoinnista?

Kolmas ja viimeinen lähestymistapa on syleillä tyypittömyyttä ja ottaa siitä kaikki mahdollinen irti. Miksi palauttaa `-1` epäonnistuneen operaation tuloksena, kun voidaan palauttaa myös `false`? Jos palautetaan `false`, on lopputuloksena intuitiivisesti selvempää koodia.

Samoin kuin kaikki muutkin työkalupakit, JavaScript tarjoaa mahdollisuuksia ja olisi hölmöä, jopa ylimielistä, jättää ne hyödyntämättä. Voidaan pohtia vertausta timpurista, joka hakkaa nauloja seinään viilalla, vaikka vieressä olisi naulapyssy, sillä "ei hänen edellisellä työmaallaan ollut kuin vasara, eikä sitä näy tämän työmaan työkalupakissa".

JavaScript-ohjelmoinnissa tulee siis ottaa kaikki irti kielen tyypittömyydestä. Koska funktioilla **voi** olla monenlaisia paluuarvoja ja syötteitä, tulee tätä ominaisuutta myös hyödyntää - sen ollessa järkevää - luomalla funktioita jotka toimivat monenlaisilla syötteillä ja palauttavat monenlaisia arvoja.

Jos merkkijonoja konkatenoiva funktio ei suostu toimimaan numeroliteraaleja syötteenään, ollaan todennäköisesti menty mönkään. Funktiota käytettäessä ohjelmoija joutuu muuntamaan numeroliteraalin ensin merkkijonoksi (luultavasti konkatenoimalla siihen tyhjän merkkijonon `+`-operaattorilla) ja vasta sitten tarjoamaan sitä funktiolle, kun todellisuudessa funktion sisäinen toteutus luultavasti toimisi aivan hyvin myös numeroliteraalilla. Ennen kaikkea ei siis pidä suotta vaikeuttaa asioita.

Toisaalta tämä ei kuitenkaan tarkoita, etteikö syötteitä tulisi tarpeen mukaan tarkistaa. Varsin loogista on esimerkiksi varmistaa ettei verkkokaupasta voi ostaa `-1.552` kovalevyä, tai että käyttäjän sukunimi ei ole `12 123 1234`. Tällainen syötteen loogisuuden varmistaminen eroaa kuitenkin kriittisesti yllä tarkoitetusta kielen tyyppijärjestelmän puukotuksesta.

Alla esimerkki siitä, kuinka tyypittömyyttä voidaan hyödyntää ohjelmoinnissa:

```
function haeJuttu(tunnus) {
   if (tunnus == 1121){
      var juttu = {
        nimi: "Jutun nimi",
        kuvaus: "Juttu vaan"
        }
```

```
        return juttu;
    } else {
        return false;
    }
}

function tulostaJutunTiedot(tunnus) {
  var juttu = haeJuttu(tunnus);
  if (juttu) {
    console.log(juttu.nimi);
    console.log(juttu.kuvaus);
  } else {
    console.log("Ei löytynyt");
  }
}

> tulostaJutunTiedot("a");
"Ei löytynyt"

> tulostaJutunTiedot("1121");
"Jutun nimi"
"Juttu vaan"
```

Koska meillä ei ollut mitään hyvää syytä erikseen kieltää haettavan "jutun" tunnuksen esittämistä merkkijonona numeraalin sijaan, voimme aivan yhtä hyvin tehdä yhtäläisyystarkistuksen löyhästi `==`-operaattorin avulla. Tämä toiminta tuskin koskaan herättää kysymystä "miksi ihmeessä?", kun taas vastakkainen toiminta, jossa syötteet `"1121"`ja `1121` tuottavat eri tuloksen on vain vaivoin perusteltavissa: tällaisen toiminnallisuuden luominen vaatii tietoista toimintaa, operaattorin vaihtamista `===`-operaattoriin.

2.2 Funktionaalinen vs. imperatiivinen JavaScript

Kuten luvun alussa jo mainittiin, JavaScript määritelmästä riippuen joko on tai ei ole funktionaalinen kieli. Määritelmistä riippumatta se kuitenkin tarjoaa ohjelmoijan käytettäväksi funktionaalisten kielten ominaisuuksia kuten sulkeumat, välitettävät funktiot sekä lambdat.

Samalla tavoin kuin tyypittömyyden kohdalla, imperatiivisesta kielestä siirtyvä ohjelmoija voi suhtautua näihin ominaisuuksiin kolmella tavalla. Tällä kertaa funktionaalisuuden täysi syleily ei kuitenkaan ole aina välttämättä paras toimintamalli. Tärkeämpää on projektin (ja yrityksen/tiimin) sisäisesti konsistentti ohjelmointitapa, joka hyödyntää kielen kaikki piirteitä parhaansa mukaan.

Toisin sanoen: syleillään JavaScriptin "sekoitustyyliä", eikä kumpaakaan ääripäätä skaalalla imperatiivinen - funktionaalinen. Oleellista on se, että tämä ohjelmointityyli on sellainen, jonka kanssa ohjelmoija on mahdollisimman sinut. Parasta olisi siis pyrkiä poimimaan "rusinat pullasta" pyrkien valitsemaan kunkin ohjelmointitehtävän kohdalla se työkalupakin työkalu, jolla ominaisuuden toteuttaminen on ennen kaikkea **helpointa** ja **luontevinta**. Ajatuksen taustalla on Brian Kernighan kuuluisa(hko?) lausahdus:

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
>
> - Brian Kernighan

Ei siis kannata kikkailla, vaan käyttää kulloinkin parasta työkalua, jolloin ongelman ratkaisu on mahdollisimman helppoa ja yksinkertaista. **Mennään sieltä mistä aita on matalin**, tosin sillä edellytyksellä että ohjelmallemme asetetut vaatimukset toteutuvat.

Tämä ajatusmalli toisaalta olettaa, että ohjemoija osaa käyttää JavaScriptin eri työkaluja ja osaa siten valita niistä parhaan. Tämä osaaminen vaatii kielen työkalupakin sisällön opettelua. Imperatiivisesta kielestä siirtyvän ohjelmoijan tulee siis tutustua myös niihin funktionaalisen tyylin työkaluihin joita JavaScript tarjoaa. Sama pätee luonnollisesti myös toisin päin. Vasta ohjelmoijan osatessa käyttää kaikkia työkaluja on hän valmis arvioimaan mikä niistä on todella paras kuhunkin ongelmaan.

Objektiivisesti (ainakin lähes), voidaan sanoa että funktionaalista tyyliä tulisi suosia erityisesti laskennassa, jossa funktionaalisen ohjelmoinnin (sivu)vaikutuksettomuus on hyödyllistä ja sen voimalliset ilmaisukeinot tuottavat lyhyen ytimekästä koodia. Samoin tulisi suosia imperatiivista ohjelmointia tapauksissa, joissa järjestelmän tilan hyödyntäminen helpottaa ohjelmointitehtävästä suoriutumista.

Esimerkiksi neliöjuuren ottaminen kaikista taulukon numeroista (esimerkki kopioitu MDN:stä) on on triviaalia käyttäen `map`-funktiota:

```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
/* roots is now [1, 2, 3], numbers is still [1, 4, 9] */
```

Vastaava koodi imperatiivisesti olisi esimerkiksi seuraavan näköinen:

```
var numbers = [1, 4, 9];
var roots = new Array();
for (var i = 0; i < numbers.length; i++){
   root[i] = (Math.sqrt(numbers[i]));
}
```

On toki täysin aiemmasta osaamisesta riippuvaista kokeeko funktionaalisen vai imperatiivisen tyylin helpommaksi ja/tai selkeämmäksi. Kiistatonta on kuitenkin se, että yllä olevassa esimerkissä funktionaalisessa versiossa on vähemmän merkkejä ja rivejä. Siten se on ainakin jollain tavoin määriteltynä yksinkertaisempi ja siten myös vähemmän altis bugeille ja "parempi".

Toisaalta JavaScriptin monet erikoisuudet vaativat myös funktionaalisesta kielestä siirtyvää ohjelmoijaa tutustumaan JavaScriptin funktionaalisiin työkaluihin ennen niiden käyttöä. Eräs "gotcha" on esimerkiksi `map()` ja `parseInt()` -funktioiden yhteistoiminta, mikä ei välttämättä ole aivan itsestäänselvä jostain toisesta kielestä saapuvalle ohjelmoijalle:

```
> ['10', '10', '10', '10', '10'].map(parseInt)
[ 10, NaN, 2, 3, 4 ]
```

[Kiitos linkistä esimerkistä @leadnose #tkt-javascript -kanavalta]

2.3 Sulkeumat

Java-ohjelmoijalle uusi työkalu saattaa olla erityisesti sulkeuma. Se on voimallinen ohjelmointitekniikka, jossa funktion sisällä luodaan sisempi funktio, jonka näkyvyysalueeseen kuuluvat myös ulomman funktion muuttujat. Tämä ei sinällään ole mitään kovin erityistä niin pitkään kun ulompi funktio on käytössä ja siihen on viittaus jostain. Sulkeuman parhaat puolet näkyvät vasta kun ulompaan funktioon ei enää ole viittausta - se ei käytännössä enää ole olemassa - mutta sisempään funktioon on olemassa viittaus. Tällöin ulomman funktion "katoamisesta" huolimatta sisempi funktio näkee edelleen ulomman funktion muuttujat: niiden ympärille on muodostettu sulkeuma, jossa sisempi funktio "elää". Jotta asiasta saataisiin vielä vähän monimutkaisempi, voi sama sulkeuma sisältää monta funktiota, jolloin ne kaikki näkevät ja käsittelevät samoja sulkeumassa olevia muuttujia.

Seuraavassa esimerkissä `tervehdi()` funktiossa määritelty `tervehdys` on funktion suorittamisen jälkeen tavoittamattomissa normaalein keinoin: siihen ei ole viitettä mistään ulkopuolelta, eikä itse `tervehdi()` funktioonkaan ole enää viitettä. `tervehdi()`-funktion palauttama funktio on kuitenkin tallennettu muuttujaan `terve`, mistä käsin sitä voidaa kutsua. Tässä tilanteessa

funktio `terve()` elää sulkeumassa, jossa sillä on pääsy muuttujaan `tervehdys`.

```
function tervehdi(nimi) {
    var tervehdys = "Terve, " + nimi;
    var tervehdiKonsoliin = function() { console.log(tervehdys); }
    return tervehdiKonsoliin;
}

> var terve = tervehdi('Arto')
> terve()
 "Terve, Arto"
```

Toisaalta meillä voi olla saman sulkeuman sisällä suurempi joukko funktiota. Seuraavassa esimerkissä kolme eri funktiota elää samassa sulkeumassa ja käsittelee samoja muuttujia.

```
function montaFunktiotaSamassaSulkeumassa() {
    var teksti = "Teksti";
    tulosta = function() { console.log(teksti); }
    resetoi = function() { teksti = "Teksti"; }
    aseta = function(uusiTeksti) { teksti = uusiTeksti; }
    return [tulosta, resetoi, aseta];
}
> var funktiot = montaFunktiotaSamassaSulkeumassa();
> funktiot[0]();
 "Teksti"
> funktiot[2]("Uusi");
> funktiot[0]();
 "Uusi"
> funktiot[1]();
> funktiot[0]();
 "Teksti"
> funktiot
[function () { console.log(teksti); }, function () { teksti = "Teksti"; }, function
(uusiTeksti) { teksti = uusiTeksti; }]
> teksti
 ReferenceError: teksti is not defined
> montaFunktiotaSamassaSulkeumassa.teksti
 undefined
> montaFunktiotaSamassaSulkeumassa().teksti
 undefined
```

Yllä olevaa sulkeumalla toteutettua versiota voidaan kontrastoida alla esitettyyn oliolla toteutettuun versioon, jossa ylläpidetään viitettä tuotettuun olioon, ja siten meillä on funktion suorittamisenkin

jälkeen suora pääsy sen sisällä määriteltyihin muuttujiin.

```
function MontaFunktiotaSamassaSulkeumassa() {
    this.teksti = "Teksti";
    this.tulosta = function() { console.log(this.teksti); }
    this.resetoi = function() { this.teksti = "Teksti"; }
    this.aseta = function(uusiTeksti) { this.teksti = uusiTeksti; }
}

> var funktiot = new MontaFunktiotaSamassaSulkeumassa();

> funktiot.tulosta();
 "Teksti"

> funktiot.aseta("Uusi");

> funktiot.tulosta();
 "Uusi"

> funktiot.resetoi();

> funktiot.tulosta();
 "Teksti"

> funktiot
 MontaFunktiotaSamassaSulkeumassa {teksti: "Teksti", tulosta: function, resetoi:
function, aseta: function}

> funktiot.teksti
 "Teksti"
```

Merkittävää sulkeumien ymmärtämisen osalta on myös se, että sulkeuma ei ole funktiokohtainen vaan **kutsukohtainen**. Jokaisella ulomman funktion kutsukerralla muodostuu uusi sulkeuma, jossa suuri sen kutsukerran sisäfunktio elää.

2.4 Poikkeukset

Virheiden heittäminen ja käsitteleminen on voimakas ohjelmointityökalu, jonka hyödyntämättä jättämiselle ei ole mitään hyvää syytä JavaScriptinkaan tapauksessa. Niiden käyttäminen on hyvää ohjelmointityyliä.

Jälleen, hyvä tyyli on lähes mikä tahansa **sisäisesti konsistentti** tyyli. Poikkeuksen tulisi kuitenkin sisältää perustiedot siitä **mitä** ja **missä** on tapahtunut. Tämä on käytännössä vaatimus poikkeuksien käytölle ohjelman bugien paikantamisessa jo korjaamisessa.

Ongelmallista tapauksessamme on se, että ECMAScript ei luonnostaan tarjoa stacktraceja, eikä

oikein muutakaan tietoa joka täyttäisi yllä esitetyn **missä** vaatimuksen. On siis perusteltua muokata virheitä ja/tai luoda uusia, jotta niihin saadaan sisällytettyä ohjelmoijalle tarpeellinen tieto. Ongelmaa ei helpota se, että JavaScriptin monet toteutukset ovat toisistaan poikkeavia: osa JavaScript toteutuksista sisällyttää poikkeuksiin stacktracet, osa ei. Jos haluamme saada saman virheilmoituksen eri ajoympäristöissä, tulee meidän luoda omat stacktracemme. Onneksi olemassa on valmiita työkaluja, joiden ansiosta emme joudu keksimään pyörää uudelleen:

> ## stacktrace.js
>
> A JavaScript tool that allows you to debug your JavaScript by giving you a stack trace of function calls leading to an error (or any condition you specify)
>
> - https://github.com/eriwen/javascript-stacktrace

Poikkeuksemme voisi siis olla esimerkiksi seuraavaanlainen:

```
function MyException(name, message, source, stack) {
 this.name = name;
 this.message = message;
 this.source = source;
 this.stack = stack;
 }

function teeJotainTaulukolle(taulukko){
 if (!isArray(taulukko) {
 /* Oletetaan että käytössä ym. stacktrace-kirjasto */
 throw new MyException("Invalid Input", "Syöte ei ollut taulukko",
"teeJotainTaulukolle()", printStackTrace());
 } else {

      /* Tehdään jotain */
   }}
```

Samaten poikkeuksien heittämiseen johtavien virheiden vakavuuden tulisi olla konsistenttiä läpi sovelluksen. Esimerkkifilosofia poikkeusten heittämiselle olisi esimerkiksi seuraava: Kaikki mikä voi mennä pieleen, tapahtuu `try-catch`:n sisällä. Mikäli tapahtuu virhe josta ei voida palautua, heitetään virheilmoitus ylös päin. Mikäli virheestä voidaan palautua, niin tehdään.

Taustalla on ajatus siitä, että heitettäessä virhettä ollaan tilanteessa jossa niin sanotusti "peli on menetetty": Suoritettavaa tehtävää ei yksinkertaisesti voida suorittaa loppuun millään järkevällä tavalla. Virhetilanteesta pyritään selviytymään kaikin mahdollisin keinoin ennen virheen heittämistä.

Tarkastellaan esimerkiksi toiminnallisuutta, jossa funktio noutaa toisen funktion avulla tietokannasta tietyn tuotteen tiedot ja esittää ne käyttäjälle: Mikäli haettavaa tuotetta ei ole, ei ole järkevää heittää virhettä ylöspäin, sillä on olemassa selvä käyttäytymismalli jolla virheen heittäminen voidaan välttää: Esitetään käytäjälle teksti "Tuotetta ei löytynyt". Toisin sanoen virhetilanne pyritään selvitetään paikallisella tasolla virheen ylemmäs heittämisen sijaan.

Toisaalta tilanteessa jossa tietokannasta tietoa hakeva funktio ei saa yhteyttä tietokantaan, on peli välittömästi menetetty. Emme funktiossa itsessään voi mitenkään palautua virhetilanteesta; ainoa (järkevä) vaihtoehto on heittää virheilmoitus ylemmäs. Tämä virheilmoitus puolestaan voidaan lopulta käsitellä vastaavasti kuin yllä.

Sama esimerkki JavaScriptinä:

```
function haeTuote(id) {
  try {
      var tuote = query("GET tuotetiedot FROM tuotteet WHERE tuote.id = tuote");
      /* ^ heittää esim. "NoDatabaseConnectionException" */

      return "<h2>"+tuote.nimi+"</h2><p>"+tuote.kuvaus+"</p>";
   } catch(e) {
      return "<p>Tuotetta ei löytynyt</p>";
   }
}

function query(sql) {
   if (!db.connected()){
      if(!db.connect()){
         throw new MyException("NoDBConnection", "Ei saatu yhteyttä tietokantaan",
"query()", printStackTrace()");
      }
   } else { /* Tehdään asioita */ }

}
```

3. Oliot ja periytyminen

JavaScriptin prototyypi-pohjainen oliomalli eroaa varsin huomattavasti esimerkiksi Javan luokka-pohjaisesta oliomallista. Nämä eroavaisuudet vaativat Javasta siirtyvältä kehittäjältä muutoksia ajattelu- ja toimintamalleihin, sekä kielten syvällisen erilaisuuden tunnustamista. Erityisen tärkeää on huomata, että prototyyppi-pohjainen oliomalli sallii luokka-pohjaisen oliomallin imitoinnin, mutta sama ei päde toiseen suuntaan. Toisin sanoen prototyyppi-pohjainen oliomalli mahdollistaa Javaa kirjavamman skaalan tapoja käsitellä oliota. Jälleen kerran, tärkeintä on projektin ja tiimin sisäinen koherenttius.

3.1 Konstruktorit

Konstruktorit ovat funktioita joiden tehtävä on yhdessä `new` operaattorin kanssa luoda uusia olioita. Ne - ja vain ne - tulisi aloittaa isolla alkukirjaimella, eikä niitä tule kutsua muussa tarkoituksessa kuin olioiden luomiseksi. Tarkastellaan konstruktorifunktiota yksinkertaisimmillaan:

```
function A(){
   this.nimi = "Olli Oletus";
   this.puhu = function() { console.log("Olen "+this.nimi); }
}

> var a = new A();
> a.puhu();
 "Olen Olli Oletus"
> a.nimi = "Camilla Custom";
> a.puhu();
 "Olen Camilla Custom";
```

Yllä konsturktorifunktio toimii tavallaan kuin naamiotu Javan luokka: se on itsenäinen komponentti joka sisältää kaiken olion alustukseen liittyvän. Emme kuitenkaan ota "kaikkea irti" perinnästä, sillä kentässä `puhu` oleva funktio on näin määriteltynä **olion** eikä sen prototyypin funktio.

Tarkastellaan toista tapaa rakentaa toiminnallisesti identtinen olio:

```
function B(){
 this.nimi = "Osku Oletus";
}
B.prototype.puhu = function() { console.log("Olen "+this.nimi); }
```

```
>var b = new B();
> b.puhu();
 "Olen Osku Oletus"
> b.nimi = "Oskari Olio";
> b.puhu();
 "Olen Osakari Olio";
```

Tällä kertaa kenttä `puhu` on luotavan olion prototyypin kenttä, eikä olion oma kenttä. Toisin sanoen, oliot `a` ja `b` näyttävät seuraavilta:

```
a = A{nimi "Olli oletus", puhu: function}
       nimi: "Olli oletus"
       puhu: function () { ... }
       __proto___: A
              constructor: function A()
              __proto__: Object

b = B{nimi "Osku oletus", puhu: function}
       nimi: "Olli oletus"
       __proto___: A
              constructor: function A()
              puhu: function () { ... }
              __proto__: Object
```

Näistä kahdesta tavasta luoda olioita jälkimmäinen on sikäli hienostuneempi, että se hyväksikäyttää perintää ja (käsittääkseni) vähentää kunkin olion muistijälkeä, kun kullekin oliolle ei tarvitse luoda omaa kopiota täysin identtisestä funktiosta. Toisaalta prototyyppiketjun läpikäynti itsessään on resursseja vievä operaatio, joten tavallaan vaihdamme muistia aikaan. Varsinkin web-ympäristössä, jossa viiveet ovat jo oletusarvoisesti tietokoneen kannalta valtavia, on tämä kuitenkin varsin todennäköisesti hyvä vaihtokauppa. Lisäksi koen henkilökohtaisesti jälkimmäisen tavan esteettisesti kauniimmaksi.

3.2 Perintäketjut

Yllä esitellyistä konstruktorimetodeista jälkimmäinen on parempi myös sikäli, että se noudattaa ja hyödyntää paremmin periytymistä: kullakin oliolla on vain se sisältö, mitä sen prototyypille ei voida asettaa. Tämä onkin yksi perinnän hyvistä puolista.

Eräs toinen perinnän hieno puoli on sen kyky mallintaa käsitehierarkioita. Tarkastellaan esimerkiksi seuraavaa perintähierarkiaa:

```
function Muoto(pintaala, ymparys){
 this.pintaala = pintaala;
 this.ymparys = ymparys;
}

Muoto.prototype.getPintaala = function() { return this.pintaala; }
Muoto.prototype.getYmparys = function() { return this.ymparys; }

function Ympyra(halkaisija){
 var pintaala = (Math.PI / 4 ) * halkaisija * halkaisija;
 var ymparys = Math.PI * halkaisija;
 Muoto.call(this, pintaala, ymparys);
   this.halkaisija = halkaisija;
}

Ympyra.prototype = new Muoto();
Ympyra.prototype.getHalkaisija = function() { return this.halkaisija; }

function Nelikulmio(leveys, korkeus){
   var pintaala = leveys * korkeus;
   var ymparys = (leveys + korkeus) * 2;
   Muoto.call(this, pintaala, ymparys);
   this.leveys = leveys;
   this.korkeus = korkeus;
}

Nelikulmio.prototype = new Muoto()
Nelikulmio.prototype.getKorkeus = function() { return this.korkeus; }
Nelikulmio.prototype.getLeveys = function() { return this.leveys; }

function Nelio(kantti){
   var pintaala = kantti * kantti;
   var ymparys = kantti * 4;
   Nelikulmio.call(this, kantti, kantti);
   this.kantti = kantti;
}

Nelio.prototype = new Nelikulmio();
Nelio.prototype.getKantti = function() { return this.kantti; }
```

Tämä on semanttinen hierarkia, missä kunkin olion prototyyppina on sen semanttinen yläkäsite. Toisin sanoen kullakin tasolla määritellään aina vain se, mikä **muuttuu** ylätason käsitteen suhteen. Täten pääsemme mahdollisimman vähällä koodilla ja oliohierarkiamme noudattaa meille luonnollisesta kielestä tuttua käsitehierarkiaa.

Järkevän perimyshierarkian luominen vaatiikin ymmärrystä siitä, että käsittelyssä on semanttinen hierarkia ylä- ja alakäsitteitä. Esimerkiksi ajatus "Svensson perii isänsä Svenin" ei ole kovin hyödyllinen konstruktio, sillä Svensson ja Sven ovat todellisuudessa käsitehierarkian rinnakkaisia käsitteitä. Enemminkin sekä Svenin että Svenssonin prototyyppinä on Mies, jonka protyyppi on puolestaan Ihminen ja niin edelleen.

Periytyminen käsitehierarkiassa sivuttaissuunnassa on yleisessä tapauksessa huono ajatus. Tarkastellaan edelleen ihmis-esimerkkiä. Oletetaan että Svensson periytyykin nyt isästään, Svenistä, esimerkiksi `Object.create()`-funktion avulla. Vuoden päästä osa ihmiskunnasta yhdistyy tietokoneisiin. Kuvaamme tätä lisäämällä Miehen prototyyppiin Ihminen kentän `elaaTietokoneessa`, joka on oletusarvoisesti `False`. Nyt Sven perii kentän Mieheltä ja Svensson Sveniltä. Oletusarvoisesti kaikkien kentät ovat `False`. Jos kuitenkin Sven siirtyy nyt elämään tietokoneeseen, käy hupsusti. Kysyttäessä Svenssonilta `elaaTietokoneessa` kentän sisältöä, löydämme lähimmän olemassaolevan sen nimisen kentän Sveniltä, jonka kenttä onkin eroava siitä mitä tarkoitimme asettaa oletusarvoksi.

Syy siihen miksi haluamme ylläpitää rajoitetta "ei sivuttaisperintää", on seuraava: Mikäli oliohierarkiamme ei sisällä sivuttaisperintää, voimme turvallisesti (ilman yllä esitetyn ongelmatilanteen pelkoa) turvallisesti propagoida oletusarvoja hierarkiassa alaspäin. Tarkastellaan esimerkkiä alasipäin propagoitumisesta:

```
/* Tyhjä yläkäsite, tästä voitaisiin periyttää myös nainen */
function Ihminen() {}
var ihminen = new Ihminen();

/* Tyhjä yläkäsite, tästä periytyvät yksilöt */
var mies = Object.create(ihminen);

var sven = Object.create(mies);
var svensson = Object.create(mies);

sven.nimi = "Sven";
svensson.nimi = "Svensson";

/*
Haluamme myöhemmin lisätä järjestelmään myös fiktiivisiä hahmoja.
```

```
Tätä varten lisäämme Ihminen-olioon seuraavan kentän.
Mahdollisille fiktiivisille hahmoille asetamme kenttään erikseen arvon "false".
*/

ihminen.onOlemassa = true;

/*
> sven.onOlemassa
true

> svensson.onOlemassa
true
*/

var matti = Object.create(ihminen);

/*
> matti.onOlemassa
true
*/

matti.onOlemassa = false;

/*
> sven.onOlemassa
true

> svensson.onOlemassa
true

> matti.onOlemassa
false
*/
```

Silti on aivan varmaa, että löytyy tilanteita joissa suoraan tietystä oliosta perityminen on järkevää ja hyödyllistä. Mielestäni sellaiset tilanteet tulisi kuitenkin mieltää erikoistilanteiksi, joissa voidaan rikkoa tätä perussääntöä.

Yllä olevassa esimerkissä käytimme `Object.create()`-metodia. Tämän metodin käyttö voi kuitenkin aiheuttaa ongelmia, joista selvimpänä esimerkkinä kaksijalkainen leijona:

```
var leg = {
 type: null
};

var Animal = {
 traits: {},
 leg: Object.create(leg)
};

var lion = Object.create(Animal);
lion.traits.legs = 4;
lion.leg.type = 'left';

var bird = Object.create(Animal);
bird.traits.legs = 2;
bird.leg.type = 'right';

alert(lion.traits.legs) // shows 2???
alert(lion.leg.type) // shows right???
```

[esimerkki luentokalvoista ja tältä sivulta.]

Tämä ongelma voidaan välttää periyttämällä vain funktioita ("metodeja"). Tällöin yllä esitetty tilanne ei pääse koskaan aiheutumaan. Pelkkien funktoiden periminen tarkoittaa käytännössä sitä, että prototyypeille ei joko aseteta mitään ei-funktioita sisältäviä kenttiä, tai että kaikki tällaiset kentät asetetaan myös objektin periviin olioihin. Tämä kuitenkin tarkoittaa sitä, että ohjelmoija jättää tarkoituksellisesti käyttämättä osan JavaScriptin potentiaalista turvallisuuden nimissä.

Kyseessä on kuitenkin sikäli perustavanlaatuinen ongelma (voimallisuus vs. turvallisuus), ettei siihen ole mielestäni olemassa yllä esitetyn kaltaista ehdotonta vastausta. Suurimmassa osassa tapauksista myös muiden arvojen periyttäminen alaspäin on erittäin hyödyllistä ja turvallista. Mielestäni parempi olisikin vain tiedostaa yllä esitetty kaksijalkaisen leijonan vaara ja kiertää sen kaltaiset monimutkaisempien tietorakenteiden perinnästä syntyvät ongelman tapauskohtaisesti, sen sijaan että kategorisesti kielletään kaikki paitsi funktioiden periminen.

4. Loppusanat

JavaScript on siitä harvinainen kieli, että harvan muun kielen kohdalla kieltä osaamaton ohjelmoija ryntää käyttämään sitä yhtä herkästi: "Tämähän on ihan kuin X". JavaScript on kuitenkin varsinainen outolintu monilla tavoilla, eikä se välttämättä taivu kovinkaan hyvin jonkin muun kielen paradigmojen mukaiseen käyttöön.

Verkko onkin täynnä erilaisia mielipiteitä siitä miten ja millaista JavaScriptiä tulisi kirjoittaa. Se on noussut web-sovellusten peruskielenä asemaan joka vastaa hyvinkin monella tasolla PHP:n asemaa: Se on kaikkialla, mutta sitä pidetään toisen luokan kielenä. Jonain, jota ei tarvitse erikseen opetella ja jolla ei saa yrittämälläkään mitään eleganttia aikaan. Yleisesti kuulee sanottavan jotain seuraavan tyyppistä: "Kyllä sillä asioita saa tehtyä, mutta 99% sillä tehdystä koodista on roskaa". Tosiasiassa kielen sisäisen yhtenäisyyden ja perustan vakauden näkökulmasta JavaScript ja PHP ovat kuin yö ja päivä. Myös yllä esitetty `map()` ja `parseInt()` -erikoisuus on täysin looginen, kun tarkastellaan JavaScriptin sisäistä toimintaa. Ongelmaksi se muodostuu vasta kun ohjelmoija **olettaa** JavaScriptin toimivan tavalla X, koska JavaScriptiin täysin liittymätön kieli Y toimii niin.

Väitän tämän "JavaScript on toisen luokan kieli" -ajatusmallin olevan objektiivisesti väärin. Kyse ei ole siitä että JavaScript olisi huono kieli, vaan siitä että sitä ulkopuolelta katsovat ohjelmoijat eivät ymmärrä JavaScriptin erilaisuutta. Kukin ohjelmoija näkee JavaScriptissä oman lempikielensä työkalut, sekä "jotain muuta roskaa". Näiden tuttujen työkalujen näkeminen saa edelleen aikaan ajatuksen siitä, että JavaScriptiä pitäisi kirjoittaa kuin tätä edellä mainittua omaa tuttua kieltä. Tämän ajatuksen synnyttyä, on helppo tuomita kaikki "outo" JavaScript koodi "huonona". Todellisuudessa varsin usein tätä koodia kirjoittanut henkilö on kuitenkin lähestynyt JavaScriptiä vain toisesta näkökulmasta: hänen mielestään hänen tapansa ja filosofiansa on "se oikea", ja kaikki muu on potaskaa. Toisin sanoen, kaikki ovat sitä mieltä että JavaScriptistä vain pieni osa on hyvää koodia, kaikki vain ajattelevat eri tavoin mikä se hyvä osa on.

Tämä "Tuttu hyvä osio ja muuta roskaa" -ajatusmalli on selvästi yhteydessä Paul Grahamin esittämään ns. "Blub paradoksiin" [Wikipedia, c2-wiki]. Tämän paradoksin tiedostaminen auttaa huomattavasti JavaScriptin hyviä ja huonoja puolia arvioitaessa. Toisaalta on pidettävä mielessä myös se, että kaikki "outo" ei välttämättä ole "hyvää".

Kaiken tämän voisi kiteyttää seuraavasti: JavaScript on varsin toimiva työkalu, joka näyttää hiukan liiankin tutulta omaksi parhaakseen. Sen käyttäjän tuliskin siis tunnustaa itselleen että se on uusi ohjelmointikieli siinä missä muutkin, ja vaatii saman opiskeluvaiheen kuin muutkin ohjelmointikielet.

Hyvä JavaScript on harvinaista. Ei siksi että JavaScriptissä kielenä olisi mitään vikaa, vaan siksi että monet tuntuvat kirjoittavan sitä kuin jotain toista kieltä.

C'est ça