Karhuryhmä
Antti Rantapelkonen
Kristian Hansson
Niklas Lillqvist
Henri Karhu

Javascript - Hyvät ohjelmointitekniikat

Johdanto

Javascript on yleisesti selainten ohjelmointiin käytetty kieli. Tässä dokumentissa ei kuitenkaan perehdytä selaimiin vaan itse javascriptiin ja sen tarjoamiin tapoihin ohjelmoida. Javascript tarjoaa meille helpon ohjelmointikielen kaikkine ominaisuuksineen, jotka ovat luotu helpottamaan ohjelmoijan arkea. Toisaalta kielen oikea ymmärtäminen ja standardimaisuuden puuttuminen saattavat luoda sekaannusta.

Tyyppiturvallisuus

Javascript on niin sanottu tyypitön kieli. Tämä tarkoittaa sitä, että muuttujaa ei alusteta tietyn tyyppiseksi varaamalla muistista aina tietyn verran tavuja. Muuttuja voi siis olla mitä tyyppiä tahansa. Tyyppeihin kuuluvat: Number, String, Boolean, Array, Object, Undefined ja Null. Tyypittömyys saattaa tuoda myös ongelmia. Esimerkiksi javascriptin muuttuja voi olla myös funktioliteraali, joten käyttäjä saattaa melko helposti aiheuttaa vahinkoa sovellukselle. Monissa ohjelmointikielissä on mahdollisuus tyypittää eri lukutyyppejä (Integer, Float, jne.) mutta Javascript tarjoaa ainoastaan tyypin Number.
Tulee kuitenkin muistaa, Javascriptin alkuperäiselle tarkoitukselle on tärkeää, että sovellus selviää useimmista virhetilanteista helposti. Ja tämä tyypittömyys luo hyvää turvaa tälle. Ohjelmoijan vastuulla on luoda tarvittavat tarkistukset, jottei sovellus kaadu turhaan ja virheilmoitukset ovat mielekkäitä käyttäjille.

Algoritmit, funktiot ja sulkeumat, poikkeusten käsittely

Funktionaalisuus vs. Imperatiivisuus

Javascript on ns. kaksiparadigmainen kieli, eli se tukee niin imperatiivista, kuin funktionaalista ohjelmointia. Imperatiivinen ohjelmointi keskittyy enemmän siihen miten jokin asia tehdään kun taas funktionaalinen lähestyy ongelmaa kysymyksellä “mitä tehdään?”
Puhtaasti funktionaalisessa ohjelmointikielessä ei ole käytössä tilamuuttujia, ainoastaan muuttujia jotka määritellään funktioiden sisällä. Funktioiden suoritusjärjestyksellä ei myöskään ole merkitystä, vaan ohjelma tulostaa aina saman syötteen. Javascript ei siis ole puhtaasti funktionaalinen ohjelmointikieli, ja itse asiassa hyvin harva funktionaaliseksi mielletty kieli on. Javascriptissä on kuitenkin funktionaalisen kielen tärkeimmät asiat, eli funktioiden välitys arvoina ja anonyymit funktiot.

```
var increment = function(x){
    return x + 1;
}
```
esimerkki anonyymista funktiosta

Javascript mahdollistaa sekä funktionaalisen että imperatiivisen ohjelmointityylin. Se, kumpaa käyttää riippuu ongelmasta ja kumpi tyyli on ohjelmoijalle tutumpi tai helpompi ymmärtää. Koska Javascript ei mitenkään pakota käyttämään kumpaakaan, niitä voi tietysti myös käyttää sekaisin. Tällä tavoin saa ainakin sen ensimmäisen ohjelman luonnoksen nopeammin valmiiksi, kun ei tarvitse miettiä miten imperatiivinen ratkaisu menisikään funktionaalisesti tai toisinpäin.

```
function fib(n){
    if (n < 2)
        return n;

    var f0 = 0, f1 = 1, f;

    for(var i = 1; i < n; i++){
        f  = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}
```
imperatiivinen fibonacci

```
function fib(n)
{
    if (n <= 1)
     {
        return n;
     }
    else
    {
      return fib(n-1) + fib(n-2);
    }
}
```
funktionaalinen fibonacci

Sulkeumat

Sulkeumat ovat hyvä työkalu antamaan tietynlaista turvaa ja selkeyttä sovellukselle, niillä voidaan luoda muuttujille erilaisia näkyvyyksiä. Sulkeumien avulla funktion sisältä näkee ulos, mutta funktion sisälle ei näe.

```
function superFun ( x ) {
	var b = x*x;			// Vapaa muuttuja ylemmän funktion sisällä
	function closureFun (y) {	// Sulkeuma, joka palauttaa arvon.
		return b / y;
	}
	return closureFun;		// Itse sulkeuma palautetaan. Tässä
vaiheessa x on kuitenkin jo määritelty.
}
```

```
function superFun ( x ,y ) {
	var b = x*x
	return b/y;
	// Tämä funktio tekee saman kuin edellinen.
}
```

Poikkeukset

JavaScript tarjoaa meille Error-objectin jota voimme käyttää hyväksemme käsitellessämme poikkeuksia. Tosin sellaisenaan käytettynä poikkeuksesta saatu viesti voi antaa meille liikaa tietoa ja todellisen virhekohdan tunnistaminen saattaisi olla hankalaa. Voimme käyttäjäystävällisyyden nimissä tällöin tehdä omia poikkeuksia ja käsitellä niitä kätevästi try/catch-rakenteella. Toisaalta voimme tyytyä vain perinteiseen if/else-rakenteeseen ja virhetilanteessa kertoa käyttäjälle mikä meni väärin. Olisi ehkä mielekästä käyttää try/catch-rakennetta käyttäjätasolla, ja Error-objectin viestejä matalammalla tasolla. If/else voi olla vain yksinkertaisille tarkistuksille.

Perinteinen if-else käsittely, tällä on helppo tarkistaa yksinkertaisia asioita.

```
var ika = prompt(“Kerro ikäsi”);
if (ika <= 10) {
   write(‘olet liika nuori pelaamaan vihaista lintua!’);
} else {
   write(‘hei, nyt lennetään!’);
}
```

Seuraavaksi esimerkit, jossa kutsutaan metodia, joka epäonnistuessaan antaa virheviestin, lisäksi toisena esimerkkinä käsittelyyn lisättynä myös finally. Meillä on funktion tarkistaIka(ika), jolla voimme tarkistaa onko annettu ikä sopiva (tulee olle välillä 10-35). Jos ei, heitämme poikkeuksen, jonka olemme itse määritelleet. Itse määritellyssä poikkeuksessa on hienoa se, että käyttäjä saa selkeän virheilmoituksen, olettaen että itse kirjoitetut virheviestit ovat selkeitä.

```
function tarkistaIka(ika) {
            if(ika >9 & ika <36) return;
               if (isNaN(ika)) {
                  throw {name: 'BadAge', message: 'Ikäsi ei ollut luku'}
               }
               if (ika < 20 || ika > 100) {
                  throw {name: 'BadAge', message: 'Ikasi ei ollut sallituissa rajoissa'}
               }
            }

            try {
               var ika = prompt("ikäsi");
               tarkistaIka(ika);
               write('käypä ikä');
            }
            catch (e) {
               alert("väärän ikänen oot:" + e.message);
            }
```

Voimme myös lisätä try/catch rakenteeseen finally, joka käydään läpi huolimatta edellisistä. Esimerkiksi edellinen poikkeus lisättynä finally. Tämä voi olla hyödyllinen lisä, jos haluamme kertoa käyttäjälle, että joka tapauksessa se mitä teimme on tehty. Finally ei ole pakollinen try/catch-rakenteessa, sen sijaan tulee muistaa käytettäessä try, tulee sitä seurata joko catch, finally tai molemmat.

```
      function tarkistaIka(ika) {
            if (ika > 9 & ika < 36) {
               return;
            }
            if (isNaN(ika)) {
               throw {name: 'BadAge', message: 'Ikäsi ei ollut luku'}
            }
            if (ika < 20 || ika > 100) {
               throw {name: 'BadAge', message: 'Ikasi ei ollut sallituissa rajoissa'}
            }

         }


         try {
            var ika = prompt("ikäsi");
            tarkistaIka(ika);
            write('käypä ikä');
```

```
}
catch (e) {
   alert("väärän ikänen oot:" + e.message);
}
finally {
   alert("kuin kivaa tämä on");
}
```

Poikkeusten käsittely on helppoa JavaScriptissä, joten miksi emme sitä myös toteuttaisi helpottamaan sovellusten käyttäjien elämää. On toki suotavaa että poikkeusten käsittely olisi järkevällä tasolla, eikä niiden takia sovelluksesta tulisi liika hidas.

Oliot ja periytyminen

Javascriptin olio-ohjelmointi on melko helppoa, tuntuu kuin olioihin voisi viitata ihan miten sattuu. Välillä tuntuukin, että rakenne on melko hauras ja sekava. Toisaalta koodia siistimällä ja järkeilemällä saa aikaan melko selkeää ja turvallisen oloista olio-ohjelmoinnille tyypillistä koodia.

Javascriptissä ei ole luokkia Javan tapaan. Javascriptin olio on **assosiaatiotaulukko**, eli joukko avain-arvo -pareja. Arvo voi olla tietokenttä, funktio, toinen olio, jne. Olioita luodaan toisten olioiden perusteella käyttämällä niitä prototyyppeinä. Täten muodostuu prototyyppiketju, jonka lopussa on Object-luokka.

```
var olio = {
    avain1: 1,
    avain2: 2
    avain3: function () {
        return this.avain1
    }
}
```

Olion luonti olioliteraalilla, josta assosiaatiotaulukko-luonne paljastuu

Olioihin voi dynaamisesti lisätä ja poistaa kenttiä, eli niitä ei tarvitse määritellä oliota tehdessä. Esim:

```
olio.avain3 = 3 // siellä on nyt avain3-avain
delete olio.avain1 // ja, tadaa, ei avain1:stä
```

Olioista:

```
function objekti(name) {
    this.nimi = name;
```

```
}

function scheisse(elain, koko) {
    this.elukka = elain;
    this.size = koko;
    this.mjono = "Tämän elukan laji on: " + this.elukka + " Ja sen koko   on:
" + this.size;
}

function maatila(nimi) {
    this.name = nimi;
    this.barn = new Array();
    this.amount = 0;

}

function lisaaElukka(tila, elain) {
    this.farm = tila
    tila.barn[tila.amount] = elain;
    document.writeln("Lisätty: " + elain.elukka);
    tila.amount++;
}

var b = new objekti("tyhjis");
var c = new scheisse("lehmä", "400kg");
var d = new scheisse("sika", "500kg");
var hevonen = new scheisse("hevonen", "1000kg");
var tila = new maatila("Piippolan vaarila");
lisaaElukka(tila, c);
lisaaElukka(tila, d);
lisaaElukka(tila, hevonen);
document.writeln(tila.name);
var arr = tila.barn;
document.writeln(arr.length);
document.writeln(arr[2].elukka);
```

Perintä

Javascriptin perintä voidaan toteuttaa monella eri tapaa, näistä mielekkäämmäksi ja turvallisimmaksi luokittelimme perinteisen tavan jossa aliluokan _prototype_ viite muutetaan osoittamaan yliluokkaan. Selkeä ja turvallinen tapa. Olioista siis löytyy prototyyppikenttä, jossa on viite edellisenä perintäketjussa olevaan olioon.

Olioiden luomisen helpottamiseksi voi käyttää esim. seuraavanlaitasta apumetodia:

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        var F = function () {};
        F.prototype = o;
        return new F();
    };
}
```

Metodi palauttaa uuden olion, ja asettaa tämän prototyypiksi argumenttina saadun olion. Jos luodusta oliosta poistetaan kenttä (deletellä) jota siinä ei ole, etsitään se prototyyppilistassa olevasta edellisestä oliosta.

```
function Yli() {
    this.name = "Väinö";
}

function Ali() {
    this.ika = 11;
    //this.name = nimi;
}
// Ali perii Ylin
Ali.prototype = new Yli();

var ali = new Ali();
document.writeln(ali.name);
document.writeln(ali.ika);

function Uiva() {
    this.osaauida = "Osaan uida";
}

function Koira() {
    this.kerroTaidot = "Osaanko uida? " + this.osaauida;
}
Koira.prototype = new Uiva();

function Lammas() {
    var villatpaalla = true;
    this.villatpaalla = true;
    this.kerinta = keritse;
    if (villatpaalla) {
        this.kerroTaidot = "MäÄä en osaa uida!";
```

```
    } else {
        this.kerroTaidot = this.osaauida;
    }
}

function keritse() {
    document.writeln("Keritään lammas");
    this.villatpaalla = false;
}

Lammas.prototype = new Uiva();

var Late = new Lammas();
var Gromit = new Koira();
document.writeln("Late Lammas: " + Late.kerroTaidot);
document.writeln("Gromit-koira: " + Gromit.kerroTaidot);
Late.kerinta();
document.writeln("Late Lammas: " + Late.kerroTaidot);
```

Yhteenveto

Javascript tarjoaa siis hyvin monipuolisen ja loppujen lopuksi syntaksiltaan melko selkeän tavan ohjelmoida. Pienet erot moniin muihin kieliin kuitenkin vaativat syvempää perehtymistä kielen rakenteeseen. Sama neuvo tosin pätee kaikkien ohjelmointikielten kohdalla. Kuitenkin C-kielen tapaiseen syntaksiin tottuneelle ohjelmoijalle hyppäys Javascriptiin saattaa olla hämmentävä, sillä Javascript sisältää niin samankaltaisuuksia kuin suuria eroja.