- Millaiset algoritmien kirjoittamisen tyylit ovat luontevia JavaScriptille? Erityisesti: miten muuttujien, parametrien ja funktioiden tyypittömyys pitäisi ottaa huomioon? Pitäisikö tyypittömyyden hyväksikäyttöä välttää ja pyrkiä "javamaiseen" tyyliin? Vai tarjoaako tyypittömyys ehkä sittenkin oivallisia tekniikoita joustavien ja yleiskäyttöisten algoritmien ja funktioiden toteuttamiseen? Perustelut ovat tässä – kuten koko dokumentaatiossa – oleellisia!
- Funktionaalista vai imperatiivista? Kumpaa paradigmaa kaksiparadigmakielessämme kannattaa suosia? Vai onko yhdistelmä paras vaihtoehto: mitä funktionaalisesti, mitä imperatiivisesti? Selostettuja esimerkkejä!
- Sulkeuma on hyvin vahva ohjelmointitekniikka. Antakaa mallikkaita esimerkkejä luennoilla esitetyistä kahdesta erilaisesta tapauksesta: a) sulkeumaan suljetut vapaat muuttujat säilyvät sulkeuman suorituksen jälkeiseen aikaan, b) sulkeumaan suljetuttujen vapaiden muuttujien määritellyt funktio päättyy, mutta sulkeuma säilyy. Selitykset ja perustelut ovat tässäkin välttämättömiä
- Poikkeusten(kin) ohjelmointi on JavaScriptissä villiä ja vapaata. Kehitelkää johdonmukainen, selkeä ja luonteva tyyli heitellä ja sieppailla poikkeuksia. Perustelut!

Tyypittömyys

Javascriptin perusluonteeseen kuuluu tyypittömyys. Tyypittömissä kielissä muuttujat voivat olla siis mitä tahansa. Usein tyypitöntä kieltä ensimmäistä kertaa opeteltaessa on ainakin allekirjoittaneelle tullut mieleen kysymys: “Miten voisin korjata tämän vian?” Oikeasti kysymyksen pitäisi kuulua: “Miten voisin hyötyä tästä?”

Ohjelmoitaessa sovellusta kannattanee pyrkiä noudattamaan kielen tarjoamia ominaisuuksia. Jos haluaa pyrkiä “javamaiseen” tyyliin, niin miksei ohjelmoi suoraan Javalla. Tyyppiturvallisuus kannattaa tietyissä tilanteissa ottaa huomioon, mutta kielen joustavuus tarjoaa mukavia tekniikoita erilaisiin tilanteisiin.

Esimerkki: Hyvin yksinkertainen kahden luvun summa / merkkijonokonkatenaatio.

```
function plusCat( x, y ) {
        return x + y;
}
```

plusCat-funktio palauttaa parametriensa summan. Jos funktiolle antaa parametreina kaksi merkkijonoa “lol” ja “cat”, niin palautettava arvo on “lolcat”. Ei välttämättä maailman hyödyllisin funktio, mutta mahdollinen tyypittömyyden takia.

Tyypittömyyden ansiosta voidaan siis mahdollisesti säästää rivitilaa kun jokaista tyyppiä varten ei tarvitse tehdä samaa funktiota uudestaan.

Funktionaalisuus VS. imperatiivisuus

Imperatiivisessa ohjelmoinnissa keskitytään enemmän siihen, miten jokin asia tehdään. Funktionaalisessa taas siihen, mitä tehdään. Käytännössä funktionaalisessa ohjelmoinnissa muuttujan arvona voi olla funktio, joka voidaan antaa parametrina toiselle funktiolle jne. Funktionaalinen ohjelmointi voi siis antaa joustavuutta funktioiden toiminnalle.

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

Sulkeuma

- Sulkeuma on hyvin vahva ohjelmointitekniikka. Antakaa mallikkaita esimerkkejä luennoilla esitetyistä kahdesta erilaisesta tapauksesta: a) sulkeumaan suljetut vapaat muuttujat säilyvät sulkeuman suorituksen jälkeiseen aikaan, b) sulkeumaan suljettujen vapaiden muuttujien määritellyt funktio päättyy, mutta sulkeuma säilyy. Selitykset ja perustelut ovat tässäkin välttämättömiä

```
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 pokkeuksia. 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- rakenteleella. 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 matalemmalla 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 virheilmoitukse, 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.