Tallinna Ülikool

Informaatika Instituut

JavaScript programmeerimise tüüpvead ja parimad praktikad

Seminaritöö

Autor: Kristjan Tammekivi

Juhendaja: Andrus Rinde

Autor:.....,,2014

Juhendaja .....,,2014

Instituudi direktor .....,,2014

Tallinn 2014

Autorideklaratsioon

Deklareerin, et käesolev seminaritöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

....

(kuupäev) (autor)

Sisukord

Sissejuhatus

Enamikes veebilehitsemist võimaldavates seadmetes on olemas JavaScript'i interpretaator ja see on võimaldanud JavaScript'il saada üheks kõige laiemalt levinud programmeerimiskeeltest. Lisaks sellele, hinnanguliselt enam kui kolmandik maailma rahvastikust kasutab internetti ja see teeb JavaScript'ist väga tähtsa programmeerimiskeele. Seetõttu on väga tähtis, et seda kirjutatakse võimalikult väheste vigadega ning võimalikult arusaadavalt, et endal ja teistel oleks lihtsam täiendada seda koodi ning see toimiks võimalikult efektiivselt ning tõrkevabalt.

Käesoleva seminaritöö teema valimise põhjuseks on töö autori isiklikud kogemused JavaScript'iga. Autor on sellega viimased aasta aega aktiivselt tegelenud ning selle aja jooksul oma sellealaseid oskusi ja teadmisi oluliselt täiendanud. Muuhulgas toob autor välja ka osad vead, mille Tema või teised on teinud JavaScript'i kirjutades ning mida JavaScript'i programmeerides töö autori hinnangul tuleks osata vältida. Autor on tutvunud veebis leiduvate vabalt kättesaadavate JavaScript'i materjalidega, kuid neis ei käsitleta reeglina parimaid praktikaid. Reeglina peavad iseõppijad aega kulutama ja parimaid praktikaid omandama koodis vigu tehes ning lehtedelt nagu StackOverflow lahendusi otsides.

Käesoleva töö eesmärgiks on välja tuua ja kirjeldada tüüpilisi vigu, mis esinevad JavaScript programmeerimisel ning anda juhiseid, kuidas neid vältida. Samuti on eesmärgiks kirjeldada mõningaid JavaScript programmeerimise parimaid praktikaid, mis autori isikliku kogemuse põhjal võimaldavad kirjutada efektiivsemaid ja kvaliteetsemaid rakendusi.

Esimeses peatükis teeb autor kokkuvõtte JavaScript'ist ja selle ajaloost, kuidas JavaScript oma nime sai, selle spetsifikatsioonidest, kasutusaladest ning tulevikust. Teises peatükis , tuginedes foorumitele, temaatilistele artiklitele ja autori kogemusele, tuuakse välja enim esinevad vead, mis võivad esineda JavaScript'i kirjutamisel. Kolmas peatükk on pühendatud kindlatele, üldistele JavaScript'i parimatele praktikatele, mis autori arvates väärivad enam tähelepanu. Kuna üheks heaks praktikaks objekt-orienteeritud JavaScript'is on disainimustrite kasutamine, on terve neljas peatükk pühendatud just disainimustritele.

Antud seminaritööst on kindlasti kasu inimestel, kes oskavad JavaScript'i algtasemel ning soovivad seda rohkem osata. Ühtlasi võib sellest kasu olla ka nendele, kes on JavaScript'iga rohkem kogenenud. Inimesed, kes pole JavaScript'iga varem kokku puutunud, võivad käesoleva töö järgimisel aga raskustesse sattuda sest selles töös ei selgitata täpselt kogu süntaksit.

1 JavaScript'ist

JavaScript loodi 10 päevaga aastal 1995 Brendan Eich-i poolt, kes töötas sellel hetkel Netscape-is eesmärgiga muuta veebilehed dünaamiliseks (Young, 2010). Algne nimi selle jaoks oli Mocha. Septembrikuus aastal 1995 muudeti selle nimi LiveScriptiks ja üks kuu hiljem võeti kasutusele nimi JavaScript sest Java oli tol hetkel väga populaarne. (W3C, 2012)

1996-1997 lasti ECMA-l (*European Computer Manufacturers Association*) valmistada spetsifikatsioon, mida teised brauserid saaksid kasutada. See viis ECMAScriptini, mis on ametlik standard, ning JavaScript on selle kõike tuntum implementatsioon. (W3C, 2012)

Teine versioon ECMAScriptist avaldati aastal 1998 ja kolmas aastal 1999. Osapoolte vaidluste tõttu hüljati ECMAScript 4, mis põhjustas kolmanda versiooni edasiarendamise versiooniks 3.1, mis aastal 2009 nimetati ümber ECMAScript 5-ks. (W3C, 2012)

2005-ndal aastal tuli Jesse James Garret välja neologismiga Ajax, tähistamaks tehnoloogiaid, mida Google Maps ja Gmail kasutasid (Garrett, 2005). Sellega sai teha veebirakendusi, milles andmed laetakse taustal. Tänu sellele ei pea tervet lehte eraldi sisse laadima. (W3C, 2012)

Eelmise kümnendi keskpaigus hakkasid levima JavaScript'i teegid, kõige tuntumad neist jQuery, MooTools ja Prototype. Tänu nendele kasvas JavaScript populaarsuses. Tänapäeval on JavaScript üks populaarsemaid programmeerimiskeeli maailmas (LangPop, 2013), peaaegu igal veebilehitsemist võimaldaval seadmel on olemas JavaScript'i interpretaator. Hiljuti on ka loodud tarkvaraplatvorm nimega Node.js, mis võimaldab JavaScript'i abil luua hästi skaleeruvaid serveripoolseid rakendusi (Tilkov & Vinoski, 2010). Node'ile toetudes saab ka luua töölauarakendusi JavaScriptiga kasutades node-webkit ning AppJS projekte. (Wang, 2013) (AppJS, 2013)

Tulemas on uus versioon JavaSript'ist, ECMAScript 6 ehk *Harmony* (W3C, 2012), mis käesoleva töö autori arvates võiks sisaldada võimalust luua klasse ning lisada andmetüüpe (näiteks oleks vaja *decimal* andmetüüpi).

2 Kokkuvõte JavaScript'i programmeerimisel esinevatest vigadest

JavaScript'i on üsnagi paindlik keel, semikoolonite kasutamine pole ilmtingimata kohustuslik ja tüüpimine pole ei range ega staatiline. Sellegipoolest võib keele täpsemate omaduste mitteteadmine viia vigadeni rakenduse toimimises või loogikas. Näiteks võib mingitel juhtudel *if* lauses tõeväärtus tulla vale ning selle tõttu jääb mingite käskude, mida tegelikult oli vaja järgida, täitmine ära.

Järgnevalt kirjeldatakse sagedamini esinevaid vigu ja probleeme, mis esinevad JavaScript programmeerimisel, mille lahenduste teadmine võimaldab kirjutada efektiivsemaid ja kvaliteetsemaid rakendusi kiiremini. Vead on kogutud nii autori enda kogemustest kui ka JavaScript'iga rohkem kogenenud inimeste (nt Douglas Crockford, JavaScript'i vanemarhitekt PayPal'is) kommentaaridele.

2.1 Võrdsusoperaatoritega seotud vead

Üks suur osa programmeerimisest on mingite muutujate võrdlemine väärtustega ning selle põhjal otsuste tegemine. Seega on tähtis, et loogikatehted oleksid õigesti koostatud. Siin peatükis käsitletakse vigu, mis võivad tekkida kindlate väärtuste võrdlemisel.

2.1.1 Leebe ja range võrdlus

JavaScript'is on kaks võrdsusoperaatorit, kaks võrdusmärki (==) tähendab leebet võrdlemist, kus kaks poolt ei pruugi tõese väärtuse tagastamiseks sama tüüpi olla, ning kolme võrdusmärgiga operaator (===) tähendab ranget võrdlust, kus mõlemad osapooled peavad olema sama andmetüüpi, vastasel juhul ei saa võrdlusest tõest väärtust. (ECMA, 2011)

Kui võrdluse üks pool on number, ja teine pool string, siis teisendab interpretaator automaatselt stringi numbriks ja siis alles tehakse võrdlus. (ECMA, 2011) (vt Koodinäide 1)

```
if(1 == "1"){
  console.log("1 == \"1\"");// Siin toimub väljastamine
}
```

Koodinäide 1. Stringi ja numbri võrdlemine

2.1.2 Objektide võrdlemine

Objektide võrdlemisel tuleb arvestada sellega, et ei võrrelda mitte objektide atribuute ja meetodeid, vaid objektide mäluaadresse. Selle tõttu tuleb objekti võrdlemisel objektiga tõene väärtus ainult siis, kui mõlemad pooled viitavad samale objektile. (ECMA, 2011) (vt Koodinäide 2)

```
var a, b;
a = b = {}
if(a == b){
```

```
  console.log("a == b");// See osa logitakse välja
}

if({} != {}){
  console.log("{} != {}");// See osa ei logita välja
}
```

Koodinäide 2. Kahe objekti võrdlemine

2.1.3 Mittearvuliste väärtuste tuvastamine

NaN on JavaScript'is väärtus, mille väärtus tuleneb siis, kui tehakse matemaatiline tehe numbri ja mingi stringi vahel, mida JavaScript ei suuda automaatselt numbriks teisendada. Kahe (Not-a-Number) võrdsuse kontrollimisel tuleb alati väär vastus (vt koodinäide 3). Kui on vaja kontrollida, *NaN*kas mingi muutuja on *NaN*, tuleb kasutada funktsiooni *isNaN*() ning vaadata samaaegselt ka muutuja tüüpi (mis JavaScript'i loogika kohaselt on number). (ECMA, 2011) (vt Koodinäide 3)

```
var a = NaN
if(a != a){
  console.log("a != a");

}
if(isNaN("string")){
  console.log("isNaN(\"string\"");//Siin väljastatakse, sest tegu pole
                        //numbriga, ometi pole tegu päris NaN-iga
}
if(isNaN(a) && typeof NaN == "number"){
  console.log("isNaN(a) && typeof NaN == \"number\"");//Siin väljastatakse
}
```

Koodinäide 3. NaN arvu tuvastamine

2.1.4 Pakendobjektide kasutamisest tingitud vead

JavaScript'i andmetüübid võib jaotada primitiivideks ning viittüüpideks. Primitiivide puhul peetakse meeles väärtus ise, viittüüpide puhul mäluaadress. Primitiivid on *null*, *undefined*, *number*, *boolean* ja *string*. Viittüübiks on *objekt* (kuigi *typeof* (*function*(){}) tagastab väärtuse „*function*", on tegu objektiga). Siiski vea tõttu JavaScript'i spetsifikatsioonis on *typeof* päringu järgi *null*-i andmetüübiks *Object.* (ECMA, 2011)

JavaScript'i tüüpe (v.a. *null* ja *undefined*) on võimalik luua nii literaaliga, kuid on olemas ka konstruktorid, mis tagastavad pakendobjekti meetodiga valueOf, mis tagastab primitiivi (Crockford, JavaScript: The Good Parts, 2008). Pakendobjekte tuleks aga vältida, sest see põhjustab vigu võrdlemisel. Leebe võrdluse korral tehakse objekt kõigepealt primitiiviks, kuid ranges võrdluses seda ei toimu. (ECMA, 2011) (vt Koodinäide 4)

```
var stringLiteraal = "See on string";
var stringPakend = new String(stringLiteraal);
if(stringLiteraal !== stringPakend){
  console.log("Stringi literaal ei võrdu stringi pakendiga");
```

```
}
if(stringLiteraal === stringPakend.valueOf()){
  console.log("Stringi literaal on võrdne stringi pakendi väärtusega");
}
```

Koodinäide 4. Stringi literaal ja pakendobjekt

2.2 Muud võimalikud vead

Siin peatükis toob autor välja muud vead, mis võivad põhjustada programmi kokkujooksmise või ebakorrektse käitumise.

2.2.1 Murdarvudega seotud vead

JavaScript'is puudub kümnendarvu andmetüüp, ainus arvu väljendamiseks mõeldud andmetüüp on number, mis on 64 bitine ujukoma arv (ECMA, 2011). Selle tõttu esineb JavaScript'is ujukoma ümardamise vigu. Näiteks liites 0.1 ja 0.2 ei anna välja täpselt 0.3 (Goldberg, 1991). Sellistel juhtudel on võimalik arvutuses saadud arvu ümardada õige komakohani või kasutada mingit teeki, mis annab meetodit komakohtadega arvude liitmiseks, näiteks *sinful.js.* (guipn, 2013)(vt Koodinäide 5)

```
if(0.1 + 0.2 != 0.3){
  console.log("0.1 + 0.2 != 0.3");//tehtest saadud arv pole täpselt 0.3

}
if(Math.round((0.1 + 0.2) * 10)/10 == 0.3){
  console.log("true"); //Siin logitakse välja true
}
```

Koodinäide 5. Ujukoma tehete ebatäpsus

2.2.2 Arvusüsteemi spetsifitseerimine *parseInt* funktsiooni kasutades

JavaScript'is kasutuselolev *parseInt* funktsioon võtab stringi ja üritab seda teha arvuks (numbri tüüpi väärtuseks). *ParseInt* võtab sisse kaks argumenti, esimene on string, mida sõeluda, ning teine on number, mis tähistab mis arvusüsteemi peaks kasutama sõelumisel. Kui sõelutav string algab nulliga, siis vanemad JavaScript'i interpretaatorid, mis ei vasta ECMAScript 5 spetsifikatsioonile, arvavad, et tegu on kaheksandiksüsteemiga. Seetõttu, kuigi teise parameetri lisamine pole kohustuslik, on soovitatav siiski see sisestada. (Mozilla Developer Network, 2014)

```
var arvustring = "08";
var arv1 = parseInt(arvustring);//Võib tagastada osades interpretaatorites
                                //väärtuse 0
var arv2 = parseInt(arvustring, 10);//Tagastab 8 igas interpretaatoris
console.log("Arv 1: " + arv1 + ", arv 2: " + arv2);
```

Koodinäide 6. parseInt funktsiooni korrektne kasutamine

2.2.3 Muutujate deklareerimata jätmine

JavaScript'is on kõik deklareerimata muutujad globaalse objekti skoobis. Selleks, et funktsioonis tahtmata mingit globaalset muutujat ümber ei defineeriks, tuleb kasutada märksõna var (ECMA, 2011)(vt Koodinäide 7).

```
var x = 1;
function test1(){
  for(x = 0; x < 10; x++){}
  console.log(x)//Väljastatakse 10
}
test1();
console.log(x);//Väljastatakse 10
function test2(){
  for(var x = 0; x < 20; x++){}//Tänu var-ile luuakse uus muutuja selle
  //funktsiooni tasemel ja globaalset x-i ei muudeta
  console.log(x)//Väljastatakse 20
}
test2();
console.log(x);//Väljastatakse 10
```

Koodinäide 7. Märksõna var kasutamine

2.2.4 Alternatiivsete for tsüklite kasutamisest tingitud vead

JavaScript'is on võimalik tsükleid teha erinevat moodi. Üks neist järgnev (vt Koodinäide 8).

```
var arvud = [5, 2, 1, 3];

for(var i = 0, arv; arv = arvud[i++];){
  console.log(arv); //5, 2, 1, 3
}
```

Koodinäide 8. Alternatiivne for-tsükkel

Selles tsüklis hinnatakse massiivist arvud saadud elemendi tõeväärtust. Kui saadud element on undefined, siis see lõpetab tsükli. See võib töötada väga hästi näiteks massiivide jaoks, milles on objektid, kuid arvuliste väärtustega massiivis võib põhjustada vigu. Näiteks, kui massiivis on arv 0, siis 0 hinnatakse vääraks JavaScript'i interpretaatori poolt (ECMA, 2011)(vt Koodinäide 9).

```
var arvud = [5, 0, 1, 3];

for(var i = 0, arv; arv = arvud[i++];){
  console.log(arv); // väljastatakse ainult 5
}
```

Koodinäide 9. 0 peatab tsükli liiga vara

2.2.5 Blokkide mittekasutamine

JavaScript'is if, while, do ja for laused võivad võtta vastu nii bloki (kasutades loogelisi sulge), kui ka üksiku käsu. Pannes aga üksiku käsu, võib hiljem märkamata jätta loogeliste sulgude puudumine ning läbi selle võib tulla vigu (vt Koodinäide 10).

```
if(false)
    console.log("Seda ei väljastada");
console.log("Seda väljastatakse");
//Harilik if ilma blokita.

//Kui aga kommenteerida välja if-i sees olev väide, muutub if-i sisu
if(false)
    //console.log("Seda ei väljastata");
console.log("Seda peaks väljastama, aga ei väljastata");
//Ei väljastata midagi
```

Koodinäide 10. Blokita if lause

2.2.6 Tehete järjekord loogikaavaldistel

Nagu matemaatikas on kasutusel tehete järjekord, on see olemas ka JavaScript'is loogikaavaldistel ning selleks kasutatakse terminit *operator precedence*. Kui seda mitte teada, siis võivad esineda vead rakenduse loogikas, selle tulemusena ei tee rakendus seda, mida selle programmeerija soovib. (Mozilla Developer Network, 2013) (vt Koodinäide 11)

```
// === arvestatakse varem kui ||
console.log(true || false === false); //Väljastatakse true

//Sulgude lisamine parandab loogikavea
console.log((true || false) === false); //Väljastatakse false
```

Koodinäide 11. Tehete järjekord JavaScript'is

2.2.7 Massiivi kopeerimine

Eelnevalt tõin välja kõik andmetüübid ning selles nimekirjas puudus Array, sest tegu on objektiga, ning selle tõttu võib massiivide kopeerimisel vigu teha. Nimelt tuleb arvestada sellega, et muutujas hoitakse mäluaadressi, ning lihtsalt omistades ühe muutuja väärtuse teisele ei saavuta soovitud tulemust. Lisaks võimalusele kasutada tsüklit massiivi kopeerimiseks, on võimalik kasutada sisseehitatud meetodit slice, mis on kiirem. (runsun, 2013)(vt Koodinäide 12)

```
var a = [0, 1, 2, 3, 4];
var b = a;
var c = a.slice(0);
b[0] = 5;
console.log(a); // [5, 1, 2, 3, 4]
console.log(b); // [5, 1, 2, 3, 4]
console.log(c); // [0, 1, 2, 3, 4]
```

Koodinäide 12. Massiivi kopeerimine

Edasiseks lihtsustamiseks on võimalik lisada Array objektile meetod selle tegemiseks (Walsh, 2012)(vt Koodinäide 13).

```
Array.prototype.clone = function(){
  return this.slice(0);
};
var a = [0, 1, 2, 3, 4];
```

```
b = a.clone();
if(a != b){
  console.log("Massiivid ei võrdu üksteisega");
}
```

Koodinäide 13. Kloonimismeetodi lisamine massiivile

2.2.8 Massiivile sarnanevad objektid

Lisaks *Array* objektidele, on olemas ka *Array*-le sarnanevad objektid, kaks selle enimkasutatavat esindajat on *arguments* ja *nodelist*, kuna nende liikmetele saab ligi pääseda indeksitega, on lihtne neid segi ajada harilike *Array*-dega. Siiski puuduvad sellel enamus *Array*-le omasetest argumentidest ja meetoditest. (ECMA, 2011)

Nodelist on objekt, mis tagastatakse funktsioonide nagu *getElementsByTagName* ja *querySelectorAll* kasutamisel ning neid on kahte tüüpi, *querySelectorAll* tagastab staatilise nodelist objekti, ülejäänud *nodelist*-e tagastavad funktsioonid tagastavad dünaamilise *nodelist*-i, mis tähendab, et elementide lisandumisel või kustumisel redigeeritakse *nodelist*-i automaatselt. Siin võib tulla viga, kui elemente kustutatakse samal ajal, kui neid tsükliga itereeritakse. (ECMA, 2011) (vt)

```
var el, els;
//Loome 5 tühja elementi
for(var i = 0; i < 5; i++){
  el = document.createElement("div");
  el.className = "minuElement";
  document.body.appendChild(el);
}
//Üritame kustutada elemendid klassiga minuElement
els = document.getElementsByClassName("minuElement");
for(var i = 0; i < els.length; i++){
  els[i].parentNode.removeChild(els[i]);
  console.log(i); //Logitakse välja arvud 0 kuni 2, sest NodeListi
                  //pikkus muutub ning kõik elemendid ei kustutata ära
}
```

Koodinäide 14. Dünaamiline nodelist-i itereerimisel tulenev viga

Iga funktsiooni sees on olemas kohalik muutuja nimega *arguments*. Tegu on objektiga, mis sisaldab kõiki kaasaantud argumente. Kuigi argumentidele saab ligi kandilistes sulgudes oleva indeksiga, puuduvad sellel *Array* atribuudid (välja arvatud *length*) ning meetodid. Küllaga saab lihtsal moel teha *arguments* objektist *Array* objekt. (Mozilla Developer Network, 2013) (vt Koodinäide 15)

```
function proov(argument1){
  var argumendid = Array.prototype.slice.call(arguments, 0);
  /*Kasutame Array prototüübi alla kuuluvatslice meetodit ning
  käivitame selle arguments objekti peal */
  console.log(argumendid);//Väljastatakse [5, 6]
}
proov(5, 6);
```

Koodinäide 15. Arguments objektist massiivi tegemine

2.2.9 Dokumendi elementide kasutamine enne seda, kui need valmis on

JavaScript'i üheks eesmärgiks on DOM-i (*Document Object Model*) manipuleerimine. Üheks meetodiks selle juures on *getElementById*(), millega saab lehekülje struktuurist pärida elemendi, millel on vastav id. Kuna JavaScript käivitatakse sisselugemisel, peab olema kindel, et soovitav element on juba laetud enne selle pärimist. (ECMA, 2011) (vt Koodinäide 16)

```
<!DOCTYPE html>
<html>
<head>
  <script>
    var el = document.getElementById("tekst");
    console.log(el.innerHTML);//Siin annab välja veateate, sest
    //sellel hetkel, kui seda käsku täidetakse, pole veel elementi,
    //mille id on tekst, olemas
  </script>
</head>
<body>
  <p id="tekst"> Lorem ipsum dolor sit amet, consectetur adipiscing
elit.</p>
</body>
</html>
```

Koodinäide 16. DOM pole skripti käivitamise ajal veel täielikult laetud.

Selle vältimiseks on võimalik kas panna vastav JavaScript-i kood alles pärast selles nõutud elementi, või siis kasutada *onload* sündmuse kuularit. (ECMA, 2011) (vt Koodinäide 17)

```
<!DOCTYPE html>
  <html>
  <head>
    <script>
      window.onload = function(){
      var el = document.getElementById("tekst");
      console.log(el.innerHTML);//Logitakse välja paragrahvi sisu
    </script>
  </head>
  <body>
    <p id="tekst">Lorem ipsum</p>
    <script>
      var el = document.getElementById("tekst");
      console.log(el.innerHTML);//Logitakse välja paragrahvi sisu
    </script>
  </body>
</html>
```

Koodinäide 17. Elemendi kasutamine alles siis, kui see on laetud

2.2.10 Sündmuste kuularite kadumine

JavaScript'iga DOM elementide lisamiseks on mitu võimalust. Meetod innerHTML kasutamine teeb koodi lühemaks, kuid sellega kaasneb kõrvalnähtus: kuna kogu elemendi sisu kirjutatakse uuesti, kaovad vanad elemendid ja need luuakse uuesti, isegi kui liita selle sisule. Seejuures kaovad ka sündmuste kuularid (Stenström, 2007). Lahendusi sellele on mitmeid, üks neist on kasutada createElement ja appendChild meetodeid. (ECMA, 2011)

2.2.11 Funktsioonide ja muutujate deklaratsioonide tõstmine

JavaScript'is toimub käivitamisel funktsioonide ja muutujate deklaratsioonide tõstmine[1], kui sellega mitte arvestada, siis võib tekkida koodis vigu. (ECMA, 2011)(vt Koodinäide 18)

```
var a = "tekst1";

(function(){

  console.log(a);//Siin logitakse välja "undefined"
  var a = "tekst2";

})();
```

Koodinäide 18. Muutuja deklaratsiooni tõstmine

Kuna JavaScript'is tõstetakse muutujate deklaratsioonid käesoleva skoobi algusesse, on koodinäide 8 võrdväärne koodinäitega 9. (Way, Quick Tip: JavaScript Hoisting Explained, 2010) (vt Koodinäide 19).

```
var a;
a = "tekst1";

(function(){
  var a;
  console.log(a);
  a = "tekst2";
})();
```

Koodinäide 19. var tõstetakse käitusfaasis skoobi algusesse

Sarnane asi toimub ka funktsioonidega, funktsioone saab lisaks tavapärasele viisile ka teha pannes funktsiooni muutujasse. Sedasi tehes, tuleb vaadata, et funktsiooni ei käivitataks enne, kui see on defineeritud, vastasel juhul annab interpretaator veateate. (ECMA, 2011) (vt Koodinäide 20)

```
funk1();
funk2();//TypeError: undefined is not a function

function funk1(){}
var funk2 = function(){};
```

Koodinäide 20. funk1 tõstetakse üles, funk2-l tõstetakse ainult muutuja deklaratsioon

[1] Ingl. keeles *hoisting*, käesoleva töö autor ei leidnud eestikeelset vastet

2.2.12 Märksõna *this* konteksti muutumine

Vea tõttu JavaScript'i spetsifikatsioonis (Crockford, Private Members in JavaScript, 2011) kindlatel oludel viitab JavaScript sisemistes funktsioonides mitte funktsioonile, milles seda kasutatakse, vaid hoopis *window* objektile. Üheks selliseks juhuks on sündmuste kuularite puhul. Selle parandamiseks on konventsioon luua kohalik muutuja *that*, millele antakse *this* väärtus. (Crockford, Private Members in JavaScript, 2011) (vt Koodinäide 21)

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <p id="tekst"> Lorem ipsum</p>
      <script>
        document.getElementById("tekst").onclick = function(){
          var that = this;
          function funk1(){
            console.log(this);//Väljastatakse window objekt
            console.log(that);//Väljastatakse p element
          }
          funk1();
        }
    </script>
  </body>
</html>
```

Koodinäide 21. This konteksti säilitamine

2.2.13 Automaatne semikoolonite paigutamine

Kuigi JavaScript'i interpretaatorid sisestavad automaatselt semikoolonid vajalikesse kohtadesse, on soovitatav neid ikkagi ise sisestada, sest osades kohtades võib interpretaator semikoolonit mitte sisestada ning sellest võib tuleneda vigu. (Corey, 2013) (vt Koodinäide 22)

```
var test = function esimene(){
  // funktsiooni sisemus
} // Siit on semikoolon puudu

(function teine(){
  // Isekäivituv funktsioon
})();
```

Koodinäide 22. Semikooloni paigutamine on vabatahtlik

Kuna pole sisestatud semikoolonit tõlgendatakse see ühe lausena. (Corey, 2013) (vt Koodinäide 23)

```
var test = function esimene(){

}(function teine(){

})();
/*Kõigepealt käivitatakse esimene funktsioon automaatselt, andes
  parameetriks teine funktsioon. Seejäral väljastatakse TypeError,
  sest esimene funktsioon ei tagasta funktsiooni, mida viimased sulud
  käivitaksid */
```

Koodinäide 23. Semikooloni mitte kasutamine võib viia vigadeni rakenduse loogikas

2.2.14 Mitmerealise string-i koostamine

JavaScript'is ei saa lihtsalt teha mitmerealist string-i. Kui jaotada string mitmele reale niisama, väljastatakse pärast esimest rida veateade. Uusi ridu saab ignoreerida, kuid sellel juhul puudub väljastatavas stringis reavahetused. Järelikult on selle parandamiseks kaks võimalust. Esimene on *escape*-ida iga rea lõpus ning lisada *\n* reavahetuseks, teine on teha kõigepealt massiiv stringidest ja siis see ühendada kasutades *join* funktsiooni. (ECMA, 2011) (vt Koodinäide 24)

```
var string1 =
"Siin on\n\
mitmerealine \n\
string";
var string2 = [
"Siin on",
"mitmerealine ",
"string"
].join("\n");
```

Koodinäide 24. Mitmerealise stringi loomine

3 Üldised JavaScript'i parimad praktikad

Siin peatükis toob töö autor välja enda hinnangul kõige tähtsamad parimad praktikad, tuginedes nii enda kogemustele kui ka autoriteetide nagu Douglas Crockford (üks JavaScripti keele arendajatest) ja Jeffrey Way (Tuts+ veebiarenduskursuste juht) avaldustele.

JavaScript annab võrdlemisi vabad käed koodi kirjutamisel, pakkudes võimalusi mõndasid asju teha mitut viisi (näiteks elementide pärimine dokumendi objektilt), kuid on tekkinud kindlad tavad, mis aitavad rakendusel tegutseda kiiremini või muuta lihtsamaks endal ja teistel seda rakendust muuta ja täiendada.

3.1 Muutujate nimed

Nagu ka muudes programmeerimiskeeltes, pole mõistlik kasutada muutjate nimedeks üksikuid mitte midagi ütlevaid tähtede ja numbrite kombinatsioone. Muutuja nimest peaks olema lihtsasti arusaadav millega on tegu ja ideaalis on ka vihje sellele, mis andmetüübiga on tegu. Näiteks, nähes koodis *isMale()* on arusaadav, et tegu on funktsiooniga, mis tagastab *boolean* andmetüübi selle kohta, kas tegu on meessoo esindajaga. (W3C, 2013)

Ühtlasi on hea tava kasutada inglisekeelseid muutujanimesid, arvestades seda, et JavaScript ise on ka inglisekeelne ning on tõenäoline, et valdav enamus inimestest, kes selle koodiga peab tegelema, räägivad ka seda. (W3C, 2013)

3.2 Kasuta Javascript'i „strict mode'i"

Alates ECMAScript 5-st on JavaScript'is olnud võimalus kasutada *scrict mode*-i, mis ütleb interpretaatorile, et peaks käituma teistmoodi koodi käivitamisel. Ühe hea küljena kaotab see mõned „vaiksed" vead JavaScript'is andes välja veateate. Näiteks harilikult kui on tegu deklareerimata muutujaga, luuakse uus globaalne muutuja, *strict mode*-i kasutades aga väljastatakse koheselt *ReferenceError*. (ECMA, 2011) (vt Koodinäide 25)

```
(function(){
  "use strict";
  deklareerimataMuutuja = 15;
})();
```

Koodinäide 25. JavaScript-i strict mode-i kasutamine

3.3 Ühtse stiili kasutamine

Alati on mõistlik kasutada ühtset stiili terve koodi vältel, see parandab loetavust ning on eriti kasulik mitme programmeerija töö ühendamisel. Selles stiilis peaks kohe projekti alguses kokku

leppima, üks võimalus on kasutada teiste poolt loodud stiilijuhendeid, näiteks Google-i stiilijuhend. (http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml)

3.4 JSLint

JSLint on Douglas Crockfordi poolt loodud koodi kvaliteedi hindaja. See küll ei määra ära, et kas kood toimib või ei toimi, aga aitab sellegipoolest vältida kergestiesinevaid vigu, ning sellel põhjusel on soovitatav enne oma koodi ülespanemist kasutada selle peal JSLinti. (Way, 24 JavaScript Best Practices for Beginners, 2009) (Crockford, JSLint, 2013)

3.5 Skriptide paigutamine

Nagu eelnevas peatükis mainisin, võib esineda vigu, kui kasutada DOM-i enne seda, kui see on tervenisti laetud, ning üks võimalustest neid vigu vältida on see, et paigutada skriptid lehekülje allotsa. Sellel meetodil on veel üks eelis: lugeja näeb sel juhul juba lehekülge varem ja saab seda kasutama hakata. Kui skriptifail on suure mahuga ja lehekülje laadimiskiirus on aeglane, siis on kahtlemata parem, kui skript laetakse kõige lõpus. (Way, 24 JavaScript Best Practices for Beginners, 2009)

3.6 DOM päringute vähendamine

DOM päringud on tüüpiliselt märgatavalt aeglasemad, kui muud JavaScript'i käsud, ning seetõttu on eriti vajalik neid puhverdada, näiteks mingile elemendile sisu lisades tsükliga. (Way, 24 JavaScript Best Practices for Beginners, 2009) (vt Koodinäide 26)

```
var element1 = document.getElementById("tekst1");
for(var i = 0; i < 10; i++){
  element1.innerHTML += "<br>" + i;
}

//Alumine tsükkel on aeglasem kui ülemine
for(var i = 0; i < 10; i++){
  element2 = document.getElementById("tekst2");
  element2.innerHTML += "<br>" + i;
}
```

Koodinäide 26. DOM päringute vähendamine

3.7 Meetodite setInterval ja setTimeout kasutamine

setInterval meetod käivitab mingi funktsiooni iga etteantud aja tagant, ei loe, kas funktsioon on oma töö lõpetanud, või ei. Kui funktsioon kasutab näiteks asünkroonseid päringuid, võivad need kliendi poole kuhjuda. Lahendus sellele probleemile on kasutada setTimeout meetodit, millega saab teha samasuguse lahenduse. (Zetafleet, 2010) (Irish, 2010) (vt Koodinäide 27)

```
//setInterval
window.setInterval(function(){
  console.log("setInterval: " + (new Date()).toTimeString());
}, 1000);

//setTimeout IIFE-ga
(function logiAeg(){
  console.log("setTimeout: " + (new Date()).toTimeString());
  window.setTimeout(logiAeg, 1000);
})();
```

Koodinäide 27. setInterval-i emuleerimine kasutades setTimeout-i

3.8 Bloki alustamine

Paljud programmeerijad kasutavad Allman'i stiilis loogeliste sulgude paigutust blokkides, kus blokki alustavad ja lõpetavad sulud pannakse eraldi ridadele samale taandele, mis blokki alustav lause. (Sysque, 2013) (vt Koodinäide 28)

```
if(true)
{
  console.log(true);
}
```

Koodinäide 28. Allmani stiilis sulgude paigutamine

JavaScript'is aga võib selline notatsioon põhjustada vigu ning sellel põhjusel soovitatakse paigutada avanev sulg samale reale. Üks koht, kus loogelise sulu paigutamine järgmisele reale võib probleeme tekitada, on tagastades objekti literaali mingist funktsioonist. Kuna JavaScript'is toimub automaatne seminkoolonite sisestamine, arvab interpretaator et ei tagastata midagi ning selle tõttu tagastatakse *undefined.* (Corey, 2013) (vt Koodinäide 29)

```
console.log((function(){
  return {
    a: 3
  }
}()).a); //Väljastatakse 3

console.log((function(){
  return
  {
    a: 3
  }
}()).a); //Väljastatakse TypeError
```

Koodinäide 29. Sulgude paigutamine muudab programmi tööd

3.9 Lühendatud süntaksi kasutamine

JavaSciptis on võimalik kirjutada asju pikalt välja, aga on soovitatav kasutada lühendatud süntaksit loetavuse parandamiseks. Lisaks on seda võimalik kiiremini kirjutada ning vajadusel lihtsamini muuta (ECMA, 2011)(vt Koodinäide 30).

```
var massiiv1 = new Array();
massiiv1[0] = 5;
massiiv1[1] = 1;
massiiv1[2] = 9;
//Kuigi see töötab, on see liialt verboosne
//Mõistlikum on kasutada järgmist
var massiiv2 = [5, 1, 9];

function test1(muutuja){
  if(muutuja){
    muutuja = 5;
  }
  console.log(muutuja);
}

function test2(muutuja){
  muutuja = muutuja || 5;
  console.log(muutuja);
}

test1();//Logitakse välja 5, aga seda oleks saanud lühemalt kirjutada
test1(1);
test2();//Toimib täpselt samamoodi nagu test1
test2(1);

var testMuutuja = 15;
/*
Võib kasutada ternaarset lauset, kõigepealt vaatame, kas muutuja vastab
tingimustele, siis paneme väärtuse, mis tagastatakse siis, kui
loogikaavaldus on tõene ja seejärel selle, mis tagastatakse vääral juhul.
Teine võimalus on kasutada if ja else lauseid.
*/
testMuutuja = testMuutuja < 10 ? testMuutuja : 10;
console.log(testMuutuja);
```

Koodinäide 30. Näiteid lühendatud süntaksist

3.10 Funktsioonide querySelector ja querySelectorAll kasutamine

JavaScript'is on võimalik kasutada CSS stiilis elementide selekteerijaid, *querySelector*, mis tagastab esimese tingimustele vastava elemendi, ja *querySelectorAll*, mis tagastab staatilise *nodelist*-i. Käesoleva töö autor tegi JavaScript'i testimislehele jsperf testi, kus Ta võrdles querySelectorAll-i kiirust getElementById, getElementsByTagName ja getElementsByClassName ning leidis, et kuigi nende kasutamine on mugav, on need enamikes brauserites aeglasemad nii lihtsamate kui ka keerukamate päringute korral ning sellel põhjusel pole soovitatav neid kasutada. (Mozilla Developer Network, 2013) (Vladimir, 2013) (Tammekivi, 2013)

4 JavaScript'i disainimustrid

Disainimuster on taaskasutatav lahendus, mida saab kasutada tihtiesinevatele probleemidele tarkvaradisainis. Disainimustritel on kolm peamist eelist:

- mustrid on tõestatud lahendused;
- mustreid saab kergesti taaskasutada;
- mustrid on väljedusrikkad ning aitavad keerukaid lahendusi esitada lihtsalt.

Mustrid pole täpsed lahendused, vaid annavad üldise lahenduse skeem. Pole olemas ühte õiget disainimustrit, iga olukord on erinev ning disainimustrit tuleb ka valida vastavalt vajadusele. Kui on ainult ühte objekti vaja, siis pole mõtet teha selle jaoks eraldi konstruktorit, samamoodi kui on vaja 100 sarnast objekti, pole mõtet need kõik objekti literaaliga teha.

(Osmani, 2012)

Järgnevalt vaatleme autori hinnangul enimkasutatavaid ja olulisemaid disainimustreid.

4.1 Objektide loomise disainimustrid

Objekte saab luua paljudel eri viisidel vastavalt sellele, mida igas vastavas olukorras vaja on. Selles peatükis vaatan mõndasid paljudest objektide loomise disainimustritest, mida JavaScript'is on võimalik kasutada.

4.1.1 Disainimuster Creational

Creational pattern-is luuakse kõigepealt tühi objekt, seda on võimalik teha kolme eri moodi. (vt Koodinäide 31)

```
var objekt1 = {};//Objekti literaal
var objekt2 = Object.create(null);//Kasutusel alates ECMAScript 5.1-st
var objekt3 = new Object();//Objekti konstruktor
```

Koodinäide 31. Kolm moodust tühja objekti loomiseks

Pärast tühja objekti loomist saab sellele lisada atribuute ja meetodeid. Atribuute ja meetodeid saab lisada neljal moel, punkt-süntaksiga, kandiliste sulgude abil, kasutades Object.defineProperty-t või Object.defineProperties-i. *Creational pattern* on alus paljudele teistele disainimustritele ning selle eeliseks objekti literaaliga omaduste määramises ees on see, et saab määrata ära eraldi get ja set meetodid ning ühtlasi saab muuta seda, kas vastavat atribuuti saab muuta, itereerida või selle omadusi muuta. (Osmani, 2012)(vt Koodinäide 32)

```
objekt1.atribuut1 = "väärtus";
objekt1["meetod1"] = function(){};
Object.defineProperty(objekt1, "atribuut2", {
  value: "väärtus2",
  writable: true,
  enumerable: true,
  configurable: true
});
/*
writable, enumerable ja configurable ärajättes on need automaatselt väärad
writable – atribuudi väärtust saab muuta
enumerable – saab itereerda üle atribuutide kasutades
  for(var atr in objekt1){}
  tsüklit
configurable – saab kustutada atribuudi väärtust ja muuta writable,
    enumerable ja configurable väärtusi
*/
Object.defineProperties(objekt1, {
  "atribuut3": {
    value: "väärtus3"
  },
  "atribuut4": {
    get: function(){
      return this.atribuut3;
    },
    set: function(uus){
      this.atribuut3 = uus;
    }
  }
});
```

Koodinäide 32. Objektile atribuutide lisamine

4.1.2 Konstruktori muster

JavaScript'is puuduvad klassid, aga toetab erilisi konstruktori funktsioone. Kasutades märksõna *new* saab interpretaatorit panna käsitlema funktsiooni kui konstruktorit. Interpretaator loob seejärel uue objekti ning funktsiooni sees määratletakse selle omadused. Konstruktori sees viitab märksõna *this* äsjaloodud objektile. (Osmani, 2012)(vt Koodinäide 33)

```
function Inimene(nimi, vanus){
  this.nimi = nimi;
  this.vanus = vanus;
  this.tervita = function(){
    return console.log("Tere, minu nimi on "+this.nimi+", olen
"+this.vanus+" aastat vana");
  }
}

var juku = new Inimene("Juku", 22);
juku.tervita();
```

Koodinäide 33. Konstruktori kasutamine

Võimalik on unustada *new* märksõna ja see võib viia vigadeni. Õnneks on funktsiooni sees võimalik kontrollida, kas seda kutsuti koos märksõnaga *new* või ei, ja selleläbi kindlustada ennast vigade vastu. (vt Koodinäide 34)

```
function Inimene(nimi, vanus){
  if(!(this instanceof Inimene)){
    return new Inimene(nimi, vanus);
  }
  this.nimi = nimi;
  this.vanus = vanus;
  this.tervita = function(){
    return console.log("Tere, minu nimi on "+this.nimi+", olen
"+this.vanus+" aastat vana");
  }
}

var juku = new Inimene("Juku", 22);
juku.tervita();
var mari = Inimene("Mari", 20);
mari.tervita();
```

Koodinäide 34. Märksõna new kasutamise tuvastamine

Uue objekti loomisel aga defineeritakse kõik meetodid uuesti. See pole suurte objektide hulgal optimaalne ning parem oleks, kui meetodid jagaksid kõik ühest konstruktorist tulevad objektid. JavaScript-is saab läbi prototüübi päriluse staatilisi meetodeid teha. (Osmani, 2012)

Igal funktsioonil JavaScript-is on atribuut nimega *prototype*. Kui käivitada konstruktorit, antakse saadud objektile ligipääs kõikidele selle atribuutidele ja sellest tuleneb konstruktori kasutamise eelised: objektid võtavad vähem ruumi mälus ning kui on vaja meetodit muuta, saab seda teha korraga. (Osmani, 2012)(vt Koodinäide 35)

```
function Inimene(nimi, vanus){
  this.nimi = nimi;
  this.vanus = vanus;
}

Inimene.prototype.tervita = function(){
  return console.log("Tere, minu nimi on "+this.nimi+", olen
"+this.vanus+" aastat vana");
}

var juku = new Inimene("Juku", 22);
var mari = new Inimene("Mari", 21);
juku.tervita();
mari.tervita();
```

Koodinäide 35. Prototype pärilus

4.1.3 Muster „Singleton“

Harilikult singletoni kasutatakse selleks, et klassist ei tehtaks mitu eksemplari. Kui klassist juba on eksemplar, siis tagastatakse viide sellele objektile. Vastasel juhul luuakse uus eksemplar. JavaScript’is aga pakub singleton nimeruumi, et globaalses nimeruumis oleks võimalikult vähe „reostust“. Mida rohkem muutujaid on globaalses nimeruumis, seda suurem on tõenäosus, et mingi muutuja defineeritakse üle, kas siis enda või kolmanda osapoole koodis. Kõige lihtsamas vormis on singleton JavaScript’is kõigest objekti literaal. (Osmani, 2012)(vt Koodinäide 36)

```
var singleton = {
  atribuut1: "string",
  atribuut2: 25,
  meetod1: function(){
    console.log("meetod");
  }
};

singleton.meetod1();
```

Koodinäide 36. Singleton objekti literaaliga

Seda on võimalik laiendada privaatsete muutujate ja meetodite lisamisega kapseldades need funktsiooni sisse ning seejärel vajalikud asjad paljastades. Singleton on kasulik siis, kui on vaja objektist ainult üks eksemplar. (Osmani, 2012)(vt Koodinäide 37)

```
var singleton = function(){
  var privaatne = "See on privaatne muutuja";
  function paljasta(){
    console.log(privaatne);
  }
  return {
    avalikMeetod: function(){
      paljasta();
    },
    avalikAtribuut: "tekst"
  }
};
var yks = singleton();
yks.avalikMeetod();//Väljastatakse "See on privaatne muutuja"
```

Koodinäide 37. Singleton privaatsete muutujatega

4.1.4 Mooduli muster

Moodulimuster on võrdlemisi sarnane *singleton*-ile, kuid eksemplar loodakse läbi IIFE (*Immediately Invoked Functional Expression*) käivitamisel. Mooduli muster on hea viis koodi organiseerimiseks, lisaks annab see võimaluse kasutada nii privaatseid kui avalikke muutujaid ja meetodeid. Halvaks küljeks on see, et puudub ligipääs privaatsetele muutujatele, kui kasutada meetodit, mis on objektile hiljem lisatud. Lisaks puudub võimalus luua automaatseid *unit test*-e privaatsetele muutujatele. (Osmani, 2012)(vt koodinäide Koodinäide 38)

```
var moodul = (function(){
  var loendur = 0;
  return {
    suurenda: function(){
      return loendur++
    },
    nulli: function(){
      loendur = 0
    }
  }
})();
moodul.suurenda();
moodul.nulli();
```

Koodinäide 38. Mooduli muster

4.1.5 Prototüübi muster

Prototüübi mustris luuakse objekte kasutades olemasoleva objekti šablooni. Kui konstruktori *prototype* objektil on mingi atribuut, siis iga selle konstruktori poolt loodud objekt saab ka endale selle atribuudi. (Osmani, 2012)

Üks selle mustri eeliseid on see, et see kasutab ära prototüüpimise jõudlust, mis JavaScript'il on olemas, selle asemel, et imiteerida võimalusi, mis on teistes keeltes. Teiste mustritega see pole alati nii. Lisaks sellele, et selle mustriga saab lihtsasti kasutada pärilust, on see ka väga kiire: kui defineerida meetod objektile, siis sellele viidatakse, selle asemel, et luua igale lapsobjektile oma meetod. (Osmani, 2012)

ECMAScript 5 standardis on *Object.create*, mis võtab sisse üks või kaks parameetrit, objekti, mille põhjal luua uus objekt ning atribuudid, mille lisada. Tuleb tähele panna, et Object.create loob uue objekti uute atribuutide ja meetoditega, mitte ei kasuta viitasid. (ECMA, 2011)(vt Koodinäide 39)

```
var auto = {
  nimi: "Mazda 6",
  tuut: function(){
    console.log("Tuuuut, mu auto on "+this.nimi);
  }
};

var muAuto = Object.create(auto);
muAuto.tuut();
//Luuakse uus objekt mille prototype objektiks on auto
```

Koodinäide 39. Lihtne prototüübi muster

Kasutades *Object.create*-i saame ühtlasi lisada atribuute uuele objektile kasutades süntaksit, mis on sarnane *Object.defineProperties* meetodile (Osmani, 2012)(vt Koodinäide 40).

```
var soiduk = {
  tuut: function(){
    console.log("Tuuuut, mu auto on "+this.nimi);
  }
};
```

```
var muAuto = Object.create(sõiduk, {
  nimi: {
    value: "Mazda 6",
    enumerable: true
  }
});
muAuto.tuut();
```

Koodinäide 40. Atribuutide lisamine objekti luues

4.2 JavaScript'i nimeruumi disainimustrid

Nimeruumi kasutamine on moodus vältida kollisioone teiste muutujate ja funktsioonidega globaalses nimeruumis. Lisaks aitavad nimeruumid organiseerida koodi blokkidesse funktsionaalsuste kaupa. (Osmani, 2012)

4.2.1 Nimeruum objekti literaaliga

Tüüpiline nimeruumi kasutus võib välja näha selline: *application.utilities.drawing.canvas.2d*. JavaScript-is sellise nimeruumi defineerimine objekti literaaliga oleks selline. (Osmani, 2012) (vt Koodinäide 41)

```
var application = {
  utilities: {
    drawing: {
      canvas: {
        "2d": {
          //vastav kood siia
        }
      }
    }
  }
};
```

Koodinäide 41. Objekti literaaliga nimeruumi loomine

4.2.2 Nimeruumi loomine *string*-i sõelumisega

Eelnevas peatükis kasutatud notatsioon on väga raskesti loetav ning keerukus kasvab iga uue tasemega. Uute tasemete loomiseks on olemas lihtsam tehnika, luues funktsiooni selle tegemiseks. (Stefanov, 2010) (vt Koodinäide 42)

```
var objekt1 = objekt1 ||{}; //Kui selline objekt puudub, luuakse uus
objekt
function lisa(nim, nimString){
  var osad = nimString.split(".");
  for(var i = 0, ln = osad.length; i < ln; i++){
    if(typeof nim[osad[i]] == "undefined")
        nim[osad[i]] = {};
    nim = nim[osad[i]];
  }
  return nim;
}
var moodul = lisa(objekt1, "moodul1.moodul2.moodul3");
```

```
console.log(moodul == objekt1.moodul1.moodul2.moodul3);//Väljastatakse
true
console.log(objekt1);
```

Koodinäide 42. Lihtne moodus nimeruumile tasemete lisamiseks.

Sedasi nimeruumi osa puhverdades saab väiksel määral kiirendada käskude täitmist, sest see vähendab nimeruumist ülesotsimist. Praktikas võib see välja näha lihtsam. (Osmani, 2012) (vt Koodinäide 43)

```
//Oletame, et rakenduses on sellised meetodid
application.utilities.drawing.rect(95,105,100,200);
application.utilities.math.sin(30);
application.utilities.math.fibonacci(10);

//Kui meetodit ainult ühe korra kasutada, siis pole puhverdamist tarvis,
//aga kui neid meetodeid kasutatakse palju, on mõistlik teha kohalikud
//puhverdused
var utilities = application.utilities,
    maths = utilities.maths,
    drawing = utilities.drawing;

//Edasi saab funktsioone kutsuda sedasi:
drawing.rect(95,105,100,200);
math.sin(30);
maths.fibonacci(10);
```

Koodinäide 43. Nimeruumi osade puhverdamine.

4.2.3 Nimeruumi loomine kasutades IIFE-t

IIFE (*Immediately Invoked Functional Expression*) kasutamine on populaarne viis kapseldamiseks. (Osmani, 2012) (vt)

```
var nimeruum = (function(nimeruum, window, document, undefined){

  nimeruum.meetod1 = function(){
    console.log("meetod");
  };

  return nimeruum;

})(nimeruum || {}, window, document);
```

Näites võtame funktsiooni parameetritena vastu neli muutujat: nimeruum, *window*, *document*, *undefined*, argumente anname aga ainult 3. Parameeter nimeruum tuleb kas juba olemasolevast objektist nimeruum või selle puudumisel aga loome uue tühja objekti. Lisaks anname kaasa *window* ja *document* objekti kahel põhjusel. Esiteks aitab see koodi minimeerimisel, IIFE sees saab kasutada lühemaid muutujaid. (Irish, 2010)

Teine põhjus seisneb mikrooptimisatsioonis. Nimelt, kui koodis leidub muutuja, siis kõigepealt otsib interpretaator seda muutujat käesolevas skoobis, seejärel sellest ülemises skoobis jne. Tehes

window ja *document* objektidest kohalikud muutujad kiirendame vähesel määral nende poole pöördumist. (Irish, 2010)

Viimane parameeter funktsioonil on *undefined*, kuna koodinäites ei antud seda kaasa, saame kindel olla, et *undefined* on tõepoolest ilma mingi väärtuseta (juhuks, kui kuskil eelolevas koodis on märgitud *undefined* väärtuseks midagi muud) ja kondrollides, kas mingi muutuja on väärtusega või ei, tuleb korrektne vastus. (Irish, 2010)

Oletame aga, et see nimeruumis on meetodi nimi juba võetud. Sel juhul peame olema ettevaatlikud, et ei kirjuta seda meetodit üle, vaid laiendame seda. Sellist praktikat nimetatakse *duck punching*-uks, funktsiooni käivitades vaadatakse kõigepealt üle, millised parameetrid antakse ja seejärel käivitatakse vastavalt nendele õige meetodi. (Irish, 2010) (vt Koodinäide 44)

```
var nimeruum = {
  meetod1: function(a, b){
    return "Summa on: " + (a + b);
  }
};

var nimeruum = (function(nimeruum, window, document, undefined){

  var meetod1 = nimeruum.meetod1 || null;

  nimeruum.meetod1 = function(a, b, c){
    if(typeof c === "function"){
      return c(a, b);
    }else if(typeof meetod1 === "function"){
      return meetod1.apply(null, arguments);
    }
  };

  return nimeruum;

})(nimeruum || {}, window, document);

console.log(nimeruum.meetod1(1,3,function(a,b){
  return a < b ? "A on väiksem kui B" : "A on suuremvõrdne B-ga";
}));

console.log(nimeruum.meetod1(5,3));
```

Koodinäide 44. "Duck punching"

Kokkuvõte

Käesoleva töö eesmärgiks oli välja tuua tüüpvead, mis esinevad JavaScript programmeerimisel ja nende lahendused ning parimad praktikad, millest kinnipidamine aitab rakendusel kiiremini toimida ning ühtlasi muudab lähtekoodi lihtsamini arusaadavamaks ning kergemini muudetavamaks enda ja teiste poolt.

Absoluutselt kõikidest parimatest praktikatest kirjutamine oleks liiga mahukas, nii et töö autor pidas vajalikuks piirata nende hulka. Sellest hoolimata sai käesolevas töös kirjeldatud suur hulk praktikatest, mida võiks rakendada töötades JavaScript'iga. Autorile teadaolevalt pole kuskil mujal neid praktikaid koos käsitletud.

Töö on mõeldud inimestele, kes on vähemalt vähesel määral tutvunud JavaScript'iga ning saab selle süntaksist aru. Ühtlasi võivad ka JavaScript'iga kogenenumad inimesed leida sellest tööst midagi uut, mida varem ei teadnud.

Neile, kes soovivad JavaScript'i disainimustritest rohkem teada, soovitab autor Andy Osmani raamatut *Learning JavaScript Design Patterns*, mida on kasutatud ka käesoleva töö allikana. Selles on põhjalikumalt kirjutatud erinevatest disainimustritest. Abiks soovi korral enda JavaScript'i alaselt iga külje alt täiendada tuleb abiks David Flanagan'i raamat *JavaScript: The Definitive Guide* mis räägib JavaScript'ist väga põhjalikult.

Tööd võib edasi arendada sellisel suunal, et testida erinevate JavaScript-i tsüklite ja meetodite võimekust eri olukordades, lisaks võiks kirjutada polüfillidest[2], mida käesolevas töös ei käsitletud, ning tuua välja JavaScript'i API (Application programming interface) osad, mis on vananenud ning mida ei soovitata enam kasutada.

[2] Ingl. keeles polyfill, autor ei leidnud eestikeelset vastet. Polüfill on kood, mis tasandab brauserite iseärasusi

Kasutatud kirjandus

AppJS. (29. Oktoober 2013. a.). *appjs*. Allikas: GitHub: https://github.com/appjs/appjs

Corey, T. (30. Aprill 2013. a.). *JavaScript Best Practices*. Allikas: CodeProject: http://www.codeproject.com/Articles/580165/JavaScript-Best-Practices

Crockford, D. (2008). *JavaScript: The Good Parts.* O'Reilly Media / Yahoo Press.

Crockford, D. (11. Detsember 2011. a.). *Private Members in JavaScript*. Allikas: crockford.com: http://www.crockford.com/javascript/private.html

Crockford, D. (25. November 2013. a.). *JSLint*. Allikas: jslint.com: www.jslint.com/lint.html

ECMA. (Juuni 2011. a.). *ECMAScript® Language Specification*. Kasutamise kuupäev: 12. Detsember 2013. a., allikas ECMA International: http://www.ecma-international.org/ecma-262/5.1/

Garrett, J. J. (18. Veebruar 2005. a.). *Ajax: A New Approach to Web Applications*. Kasutamise kuupäev: 12. Detsember 2013. a., allikas Adaptive Path: https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 5-48.

guipn. (2013). *sinful.js*. Kasutamise kuupäev: 01. 01 2014. a., allikas GitHub: https://github.com/guipn/sinful.js/wiki/API#math

Irish, P. (Režissöör). (2010). *10 Things I learned from the jQuery Source* [Film].

LangPop. (25. Oktoober 2013. a.). *Programming Language Popularity*. Kasutamise kuupäev: 12. 12 2013. a., allikas LangPop: www.langpop.com

Mozilla Developer Network. (1. Juuli 2013. a.). *arguments*. Kasutamise kuupäev: 18. Detsember 2013. a., allikas Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/arguments

Mozilla Developer Network. (26. November 2013. a.). *Document.querySelectorAll*. Allikas: Mozilla Developere Network: https://developer.mozilla.org/en-US/docs/Web/API/Document.querySelectorAll

Mozilla Developer Network. (21. Detsember 2013. a.). *Operator Precedence*. Allikas: Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

Mozilla Developer Network. (1. Jaanuar 2014. a.). *parseInt().* Allikas: Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt

Osmani, A. (2012). *Learning JavaScript Design Patterns.* O'Reilly Media.

runsun. (16. Juuli 2013. a.). *Array clone slice vs for loop*. Kasutamise kuupäev: 18. Detsember 2013. a., allikas jsPerf — JavaScript performance playground: http://jsperf.com/array-clone-slice-vs-for-loop/3

Stefanov, S. (2010). *JavaScript Patterns.* Sebastopol: O'Reilly Media, Inc.

Stenström, E. (26. September 2007. a.). *Manipulating innerHTML removes events*. Allikas: Friendly Bit: http://friendlybit.com/js/manipulating-innerhtml-removes-events/

Sysque. (19. 06 2013. a.). *C Style: Standards and Guidelines*. Allikas: sysque.com: http://syque.com/cstyle/ch6.7.htm

Zetafleet. (22. Aprill 2010. a.). *Why I consider setInterval to be harmful*. Allikas: zetafleet.com: http://zetafleet.com/blog/why-i-consider-setinterval-harmful

Tammekivi, K. (31. Detsember 2013. a.). *querySelectorAll vs getElementById, getElementsByTagName & getElementsByClassName*. Allikas: jsperf.com: http://jsperf.com/queryselectorall-vs-getelementbyid-getelementsbytagname

Tilkov, S., & Vinoski, S. (01. November 2010. a.). Node.js: Using JavaScript to Build High-Performance Network Programs. *Internet Computing, IEEE, 14*(6), lk 80-83.

W3C. (27. Juuni 2012. a.). *A Short History of JavaScript*. Kasutamise kuupäev: 12. 12 2013. a., allikas w3.org: http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

W3C. (16. Veebruar 2013. a.). *JavaScript best practices*. Allikas: W3C: http://www.w3.org/wiki/JavaScript_best_practices

Walsh, D. (14. Juuli 2012. a.). *Clone Arrays with JavaScript*. Kasutamise kuupäev: 18. Detsember 2013. a., allikas The David Walsh Blog: http://davidwalsh.name/javascript-clone-array

Wang, R. (30. Detsember 2013. a.). *node-webkit*. Allikas: GitHub: https://github.com/rogerwang/node-webkit

Way, J. (16. Juuni 2009. a.). *24 JavaScript Best Practices for Beginners*. Allikas: Nettuts+: http://net.tutsplus.com/tutorials/javascript-ajax/24-javascript-best-practices-for-beginners/

Way, J. (7. Oktoober 2010. a.). *Quick Tip: JavaScript Hoisting Explained*. Allikas: Nettuts+: http://net.tutsplus.com/tutorials/javascript-ajax/quick-tip-javascript-hoisting-explained/

Vladimir. (15. August 2013. a.). *VT getElementById and querySelector*. Allikas: jsperf: http://jsperf.com/getelementsbyclassname-vs-queryselectorall/43

Young, A. (24. Mai 2010. a.). *History of JavaScript: Part 1*. Allikas: DailyJS: http://dailyjs.com/2010/05/24/history-of-javascript-1/