TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JESPER HAAPALINNNA

TYYPPIJÄRJESTELMÄT WEB-OHJELMOINNISSA

Kandidaatintyö

Tarkastaja: Maarit Harsu
Jätetty tarkastettavaksi 6.5.2018

TIIVISTELMÄ

JESPER HAAPALINNA: Tyyppijärjestelmät web-ohjelmoinnissa - Type systems in web programming
Tampereen teknillinen yliopisto
Kandidaatintyö, 20 sivua
Toukokuu 2018
Tieto- ja sähkötekniikan tekniikan kandidaatin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Maarit Harsu

Avainsanat: WWW, ohjelmointi, frontend, JavaScript, TypeScript

Internet-yhteydellisten laitteiden määrä on kasvanut nopeasti viime vuosikymmeninä. Tämä johtuu elektroniikkakomponenttien kehityksestä ja niiden valmistuskulujen halpenemisesta. Samaan aikaan myös tietoliikenneprotokollat ovat kehittyneet ja tiedonsiirtonopeudet kasvaneet. JavaScript luotiin alun perin skripti-ohjelmointikieleksi ja se sisällytettiin vuonna 1995 Netscape Navigator -selaimeen. Alkuperäinen tarkoitus oli mahdollistaa yksinkertaisten skriptien suorittamien ja animaatioiden näyttäminen selaimessa. Siitä lähtien JavaScriptin käyttötarkoitus on muuttunut yhä enemmän logiikkaa sisältävien ohjelmien ensisijaiseksi toteutuskieleksi. Tässä työssä tutkitaan syitä JavaScriptin kasvaneelle suosiolle ja ongelmille tyyppijärjestelmien näkökulmasta.

JavaScriptin ongelmien korjaamiseksi Microsoft on kehittänyt uuden staattisesti tyypitetyn ohjelmointikielen TypeScriptin. Kielen tarkoituksena on päästä yli JavaScript-kehityksen vaikeuksista. TypeScript on julkaissut useita ominaisuuksia, jotka ovat sisällytetty JavaScriptin noudattamaan ECMAScript-standardiin vasta paljon myöhemmin. Kun uusi ECMAScript-standardin mukainen ominaisuus toteutetaan JavaScriptillä, ei ominaisuutta välttämättä voi käyttää ohjelmassa, jossa halutaan mahdollistaan tuki vanhemmille selaimille. TypeScriptin avulla tällaiset ominaisuudet voidaan kääntää noudattamaan vanhempaa ECMAScript-standardia, jolle löytyy kattavampi selaintuki.

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

CSS	Cascading Style Sheet
ES	EcmaScript
HTML	Hypertext Markup Language
WYSIWYG	What You See Is What You Get
WWW	World Wide Web

1. JOHDANTO

Tänä päivänä on helppo huomata, kuinka yhä enemmän ohjelmistoja siirtyy perinteisestä työpöytäkäytöstä kohti verkkoselaimessa toimivaa ohjelmistoa. Web-ohjelmistojen hyödyt ovat varsin selkeät. Käyttäjän ei tarvitse asentaa tai päivittää ohjelmistoa manuaalisesti tietokoneelleen, sillä ohjelmisto suoritetaan selaimessa ja sen tarjoava palvelin voi päivittää ohjelmiston version huomaamattomasti. Web-ohjelmiston jakelu voidaan suorittaa välittömästi ympäri maailmaa, jolloin välikäsiä tai jakelijoita ei tarvita. [1]

Web-sovellusten ohjelmointi voidaan usein jakaa kahteen erilliseen osaan: selainpuolen ohjelmointi (frontend) ja palvelinpuolen (backend) ohjelmointi. Tässä työssä keskitytään vain selainpuolen ohjelmointiin. Selainpuolen ohjelmointiin ei edes ollut pitkään aikaan vaihtoehtona staattisesti tyypitettyä ohjelmointikieltä. Silti selainpuolen ohjelmointi on kasvattanut suosiotaan sovellusten vaatimuksien mukaisesti. Selainpuolen ohjelmoinnissa suosituimpana ohjelmointikielenä on jo pitkään käytetty JavaScriptiä. Ongelmaksi on kuitenkin noussut ohjelmien logiikan kasvu, jota on vaikea hallita JavaScriptin ja sen dynaamisen tyyppijärjestelmän avulla. Ongelmaa helpottamaan on kehitetty uusia ohjelmointikieliä kuten TypeScript, jotka tarjoavat perinteisten palvelinpuolen olio-ohjelmointikielten ominaisuuksia, esimerkiksi staattisen tyypityksen. Ohjelmakoodi käännetään kielen kääntäjän avulla puhtaaksi JavaScriptiksi, jolloin Internet-selain voi ajaa sitä avain kuin se olisi alun perinkin ohjelmoitu JavaScriptillä.

Ohjelmointikielten tyyppijärjestelmien paremmuudesta käydään väittelyä. Joidenkin mielestä staattinen tyyppijärjestelmä on ainoa oikea tapa ja dynaaminen tyypitys johtaa väistämättä virheisiin. Vastaväitteenä staattista tyypitystä pidetään ohjelmointia rajoittavana ja hidastavana tekijänä. Tässä työssä tutkitaan molempien tyyppijärjestelmien hyviä ja huonoja puolia. Lisäksi tutkitaan, miten dynaamisesti tyypitetyn JavaScriptin ongelmia voidaan ratkaista TypeScriptin tarjoaman staattisen tyyppijärjestelmän avulla.

Luvussa 2 tutkitaan ohjelmointikielten piirteitä keskittyen tyyppijärjestelmiin ja olio-ominaisuuksiin. Luvussa 3 käydään läpi webin historiaa ja mikä JavaScriptin merkitys on ollut siinä. Lopuksi luvussa 4 esitellään uusi ohjelmointikieli TypeScript ratkaisuksi selainpuolen web-ohjelmoinnin haasteisiin.

2. HYVÄN OHJELMOINTIKIELEN PIIRTEET

Ohjelmointikielen valinta voi olla ohjelmoijalle hankala tehtävä. Halutun ohjelman toteutuskieleksi saattaa löytyä useita kelpaavia vaihtoehtoja. Tässä luvussa tutkitaan, minkälaiset piirteet tekevät ohjelmointikielestä hyvän.

2.1 Tyyppijärjestelmä

Tyyppijärjestelmä on keskeisessä asemassa ohjelmointikielen rakennetta. Se määrittelee muun muassa ohjelmointikielen käyttämän tyyppitarkastelun. Tyyppitarkastelu on tyyppirajoitteiden tarkistamista, jonka avulla päätellään, onko esimerkiksi muuttujan sijoittaminen toiseen muuttujaan laillinen operaatio. Tyyppitarkastukset voidaan jakaa kahteen kategoriaan sen perusteella, milloin tyyppitarkastus tehdään. Staattisessa tyypityksessä tyyppitarkastus tehdään ohjelman käännösvaiheessa. Dynaamisessa tyypityksessä tyyppitarkastus tehdään vasta ajon aikana.

2.1.1 Staattinen tyypitys

Ohjelmointikieliä, joissa jokaisen lausekkeen tyyppi pystytään määrittelemään staattisella ohjelma-analyysillä, kutsutaan staattisesti tyypitetyiksi [2]. Käytännössä siis ohjelman käännösaikana kääntäjä suorittaa analyysin, jossa tarkastetaan operaatioiden laillisuus. On kuitenkin hyvä huomioida, ettei staattinen tyypitys vaadi tyyppien eksplisiittisesti ilmoittamista. Esimerkiksi lausekkeet

```
var i = 3;
var j = “42”;
```

ovat täysin laillisia staattisesti tyypitetyssä C#-ohjelmointikielessä. Näille lausekkeille osataan automaattisesti päätellä oikea tyyppi, joten sen täsmällinen ilmoittaminen ohjelmakoodissa on turhaa.

Staattisen tyypityksen selkeimpänä hyötynä on ohjelmointivirheiden havaitseminen aikaisessa vaiheessa. Tällaiset virheet pystytään korjaaman heti, sen sijaan että ne jäisivät ohjelmakoodin sekaan ja havaittaisiin vasta paljon myöhemmin. [3]

Myös ohjelmiston ylläpidettävyys hyötyy staattisesta tyypityksestä. S. Kleinschmager et. al. suorittivat empiirisen tutkimuksen, jossa verrattiin staattisen ja dynaamisen tyypityksen vaikutuksia ohjelmiston ylläpidettävyyteen. Tutkimuksessa koehenkilöille annettiin tehtäväksi joukko erilaisia ohjelmointitehtäviä, joiden kohteena oli saman ohjelmiston pohjalta luodut staattisesti ja dynaamisesti tyypitetyt versiot. Tutkimuksen lopputulos on tiivistetysti seuraavanlainen:

- Staattiset tyyppijärjestelmät helpottavat ihmisiä käsittelemään uusia luokkia.
- Staattiset tyyppijärjestelmät helpottavat ihmisiä korjaamaan tyyppivirheitä.
- Semanttisten virheiden korjaamiseen ei havaittu eroja ihmisen tekemään kehitykseen kuluvalla ajalla.

Tutkimuksessa kuitenkin huomautetaan, että dynaamisesti tyypitetyssä ohjelmistoversiossa esimerkiksi muuttujien nimistä oli poistettu tyyppiin viittaava informaatio. Tämä todennäköisesti aiheutti ylimääräistä hankaluutta. [4]

Staattisen tyypityksen haittana on sen joustavuuden puute. Tyyppien määritteleminen hidastaa ohjelmointityötä ja vähentää ilmaisuvoimaa. Ohjelmoijalle syntyy helposti tilanne, jossa ohjelman tyypit estävät yksinkertaisen toteutuksen johonkin ongelmaan ja tällöin ohjelmoijan on keksittävä keino kiertää tyyppien asettamat rajoitteet.

2.1.2 Dynaaminen tyypitys

Dynaamisesti tyypitetyissä ohjelmointikielissä tyyppiturvallisuutta tarkastellaan vasta ohjelman ajon aikana käännösajan sijaan. Tällaisissa kielissä ajon aikana käytetään tyyppimerkintöjä (type tags) erottamaan erilaiset rakenteet toisistaan [3]. Tämä johtaa myös siihen, että mahdolliset tyyppivirheet huomataan vasta ajon aikana. Yhteistä staattisesti tyypitetyille ohjelmointikielille kuitenkin on, ettei semanttisia virheitä voida havaita ennen suoritusta. Tällaiset virheet ovat sellaisia, jossa ohjelmakoodi itsessään on täysin oikeellinen, mutta sen lopputulos ei vastaa ohjelmoijan odotuksia. Yksinkertaisessa tapauksessa, jossa dynaamisesti tyypitetyssä kielessä suoritettaisiin operaatio

```
let i = 3;
let j = “42”;
let x = i + j;
```

saataisiin ajonaikana virhe. Staattisella analyysillä tämä voitaisiin helposti huomata etukäteen. Dynaamisessa tyyppitarkastelussa tärkeimpänä ajatuksena ei kuitenkaan ole tällaisten yksinkertaisen virheiden ennaltaehkäiseminen. Tärkeää on lykätä paljon monimutkaisempien lausekkeiden evaluointi ajon aikana tapahtuvaksi, mikä ei edes välttämättä olisi mahdollista käännöksen aikana.

Staattiseen tyypitykseen verrattuna dynaaminen tyypitys häviää suorituskyvyssä. Samalla tavalla toteutettu ohjelma dynaamisesti tyypitetyllä ohjelmointikielellä on hitaampi, kuin vastaava ohjelma staattisesti tyypitetyllä ohjelmointikielellä. Ehdotonta ajallista suorituskykyä tärkeämpänä voidaan pitää suorituskykyä suhteessa vaatimuksiin: ”suoriutuuko ohjelma tarpeeksi nopeasti?”. Tämä kysymys on kuitenkin usein tarpeeton johtuen tietokoneiden kehittyneistä suorituskyvyistä. Kuitenkin hyvin matalan tason tehtävissä ja alhaisen tehon tietokoneissa, kuten sulautetutuissa järjestelmissä, staattisen tyypityksen tuoma suorituskyky on tärkeä etu. [5] Esimerkiksi web-ohjelmiston palvelinpuolen toteutuksessa dynaamisen tyypityksen heikompaa suorituskykyä tuskin havaitaan. Pelkästään

HTTP-pyynnöt käyttöliittymältä palvelimelle vievät enemmän aikaa, kuin mitä staattinen tyypitys nopeuttaisi palvelimen tehtäviä.

Dynaamisesti tyypitetyille ohjelmointikielille yhteistä on niiden käsitteellinen yksinkertaisuus. Nämä kielet ovat pohjimmiltaan valmiita suorittamaan vaihtokaupan suorituskyvyn ohjelmoijan tuottavuuden välillä. Yleisesti ottaen tämä yksinkertaisuus tekee dynaamisesti tyypitetyistä ohjelmointikielistä yksinkertaisempia oppia ja käyttää verrattuna staattisesti tyypitettyihin ohjelmointikieliin. [5] Tästä johtuen esimerkiksi ohjelmointia vasta opettelemaan alkaneille voidaan suositella ensimmäiseksi ohjelmointikieleksi dynaamisesti tyypitettyä kieltä. Kokeneille ohjelmoijille eduksi nousee esiin parempi tuottavuus. Kun ohjelmointikieli asettaa vähemmän rajoitteita, on sillä helpompi ja täten nopeampi ratkaista ongelmia.

S. Kleinschmager, S. Hanenberg, R. Robbes, È Tanter, A. Stefik tutkivat jo pitkään jatkunutta väittelyä mahdollisista eduista ja haitoista staattisissa ja dynaamisissa tyyppijärjestelmissä. Tutkimuksen pääkysymyksenä on, onko staattinen tyyppijärjestelmä hyödyllinen ihmiselle (verrattuna tietokoneeseen) seuraavista näkökulmista tarkasteltuna:

1. luokkiin liittyvät käyttötapaukset
2. tehtävät jotka sisältävät virheiden korjaamista ohjelmassa.

Tutkimuksen tarkoituksena on tunnistaa tilanteet, joissa tyyppijärjestelmällä on merkitys kehitykseen kuluvaan aikaan. Kokeen lopputuloksena havaittiin staattisen tyyppijärjestelmän positiivinen vaikutus kuudessa yhdeksästä tehtävässä. Tämä tapahtui kaikissa tehtävissä, joissa tarkoituksena oli korjata tyyppivirheitä ja suurimmassa osassa luokkia käyttävissä tehtävissä. Semanttisten virheiden korjaamisen tyyppijärjestelmällä ei havaittu olevan vaikutusta. [4]

2.2 Olio-ominaisuudet

Olio-ohjelmointia hyödyntävät ohjelmointikielet vastaavat ohjelmistosuunnittelussa kolmeen tärkeään tarpeeseen: tarve uudelleenkäyttää ohjelmakomponentteja mahdollisimman paljon, tarve muuttaa ohjelman käyttäytymistä mahdollisimman vähäisillä muutoksilla olemassa olevaan ohjelmakoodiin ja tarve säilyttää eri komponenttien itsenäisyys [6]. Ohjelmakomponenttien uudelleenkäytöllä voidaan vähentää tarvittavan ohjelmakoodin määrää. Esimerkiksi funktioita ei tarvitse kirjoittaa uudelleen useaan eri paikkaan, vaan se voidaan sisällyttää luokkaan, josta luodaan olio tarvittaessa. Toinen selkeä hyöty saavutetaan muutoksia tehdessä – muutos tarvitsee tehdä vain yhteen paikkaan.

Uudelleenkäyttö voidaan käytännössä toteuttaa usealla eri tavalla. Luokkien periyttämisen avulla voidaan käyttää kantaluokan toteutusta metodille tai korvata se uudella. Kantaluokan toteutusta ei siis tarvitse kirjoittaa uudelleen aliluokissa. Tyypillisesti luokka perii toisen luokan, jonka palveluita se tarvitsee, ja toteuttaa vain tarvitsemansa uudet

palvelut. Periyttämisellä myös mahdollistetaan olemassa olevien toimintojen laajentaminen, ilman että alkuperäistä toteutusta tarvitsee muuttaa. Periytymisen huolellinen käyttö muodostaa lopulta luokkahierarkioita, jossa kantaluokan aliluokat liittyvät toisiinsa läheisesti.

Tämä myös kannustaa yhtenevien protokollien kehittämiseen. Koska aliluokat perivät jonkin luokan operaatiot, ne myös jakavat sen protokollan. Tällöin periytymistä käyttäen saadaan automaattisesti aikaiseksi yhtenevää protokollaa käyttävä perhe. Periytyminen ei siis pelkästään lisää uudelleenkäyttöä vaan auttaa myös standardien protokollien kehittämiseen. [7]

Monimuotoisuus (polymorphism) on tärkeässä roolissa olio-ohjelmoinnissa. Monimuotoisuudella tarkoitetaan, että jotkin arvot ja muuttujat voivat olla useampaa eri tyyppiä. Olio-ohjelmoinnin tapauksessa, kantaluokassa voidaan määritellä metodi, jota periytetyt luokat käyttävät. Esimerkiksi kutsussa `Animal.move()`, `Animal`-luokan tilalla voi olla mikä tahansa siitä periytetty luokka. Monimuotoiset funktiot ovat funktioita, joiden operandeilla, eli varsinaisilla parametreilla, voi olla enemmän kuin yksi tyyppi. Monimuotoisuus voidaan jakaa kahteen pääkategoriaan: yleispätevä monimuotoisuus (universal polymorphism) ja ad-hoc monimuotoisuus (ad-hoc polymorphism). Yleispätevästi monimuotoiset funktiot toimivat tavallisesti äärettömällä määrällä eri tyyppejä, joilla on kuitenkin tietty yhteinen rakenne. Ad-hoc monimuotoiset funktiot puolestaan toimivat vain rajallisella määrällä tyyppejä, jotka eivät välttämättä liity toisiinsa. Käytännön ero näille monimuotoisuuden kategorioille huomataan niiden toteutuksista. Universaalisti monimuotoiset funktiot suorittavat saman ohjelmakoodin riippumatta annetuista sallittujen tyyppien argumenteista. Ad-hoc monimuotoinen funktio voi suorittaa eri koodin jokaiselle argumentin eri tyypille. [2]

Monimuotoisuuden avulla saavutetaan siis komponenttien uudelleenkäyttöä. Valmis monimuotoisesti toteutettu funktio on helppo ottaa käyttöön, sen sijaan että kirjoitettaisiin täysin uusi funktio. Tämä tosin saattaa vaatia ohjelmoijalta lisätyötä komponentteja toteuttaessa. On myös olemassa ohjelmointikieliä, joissa käytetään monimuotoisuuden tyyppiä nimeltä pakotettu tyyppimuuntaminen (coercion). Pakotettu tyyppimuunnos on semanttinen operaatio, jossa argumentin tyyppi muunnetaan sellaiseksi, jota funktio odottaa parametrina tyyppivirheen välttämiseksi [2]. Monimuotoisuuden toteuttaminen ei siis aina ole ohjelmoijan vastuulla, vaan ohjelmointikieli saattaa tarjota sitä automaattisesti, joskin vain pienenä osajoukkona koko polymorfian käsitteen laajuudesta.

3. WEB-OHJELMOINTI

Tässä luvussa kerrotaan, mitkä asiat ovat vaikuttaneet ohjelmistojen siirtymisen kohti webiä. Muun muassa dynaamisesti tyypitetty JavaScript-ohjelmointikieli on ollut tässä tärkeässä roolissa. Edellisessä luvussa tarkasteltiin dynaamisen tyyppijärjestelmän hyviä ja huonoja puolia. Nämä puolet ovat selvästi havaittavissa tässä luvussa tarkemmin käsiteltävässä JavaScriptissa. Luvussa kerrotaan myös, miten JavaScript on kehittynyt ajan myötä ja mikä sen vaikutus on ollut ja tulee olemaan web-ohjelmoinnissa.

3.1 Web-ohjelmoinnin kehittyminen

Web-ohjelmoinnin suosion kasvua voidaan rinnastaa Internetin kasvuun. Internet-yhteyden omaavien pääsylaitteiden lukumäärä on kasvanut eksponentiaalisesti viime vuosikymmeninä [8], jota puolestaan voidaan selittää muun muassa komponenttien kehittymisellä ja hinnan alenemisella. Samalla myös tiedonsiirtoteknologiat ovat kehittyneet ja tiedonsiirtonopeudet kasvaneet.

Vuosi 1993 oli Internetin historian käännekohta, jolloin juuri julkaistun WWW:n (World Wide Web) suosio lähti suureen nousuun Mosaic-selaimen myötä. Muutokset Internetin keskeisimpiin protokolliin ovat kuitenkin pysyneet pieninä. WWW:n kaltaisiin kaupallisiin verkkoihin muutoksia otetaan käyttöön, kun ne ratkaisevat jonkin välittömän ongelman tai niiden avulla voidaan tuottaa enemmän rahaa. [8] Mosaic oli laatuaan ensimmäinen graafinen selain, sillä se pystyi näyttämään muutakin kuin pelkkää ACII-tekstiä, muun muassa HTML-sivuja. Vuotta myöhemmin julkaistiin Netscape Navigator -selain, joka saavutti parhaimmillaan yli 80 % osuuden markkinoista. Suosio johtui Netscapen uusista innovatiivista ominaisuuksista, kuten verkkosivun sisällön esittämisestä samalla kun se vielä latautuu ja tuesta JavaScript-ohjelmoinnille. [9].

HTML (Hypertext Markup Language) on merkkauskieli, joka kuvailee verkkosivua ja jonka selain esittää. HTML-dokumentti koostuu elementeistä, jotka puolestaan koostuvat aloitus- ja lopetus-tageista ja niiden välisestä sisällöstä. HTML elementti on esimerkiksi

```
<h1>Web-ohjelmointi on helppoa!</h1>,
```

jossa <h1> on ennalta määrätty tagi, joka kuvailee otsikkoa ja </h1> on sen lopetustagi. HTML:stä on tullut erittäin suosittu työkalu informaation lisäämiseen webiin, ilman tarvetta ohjelmointikielten ymmärtämiselle [9]. Tätä suosiota voidaan selittää HTML:än suurella virheensietokyvyllä, jossa virheet yksinkertaisesti sivuutetaan ja laajalla määrällä erilaisia työkaluja, kuten WYSIWYG (What You See Is What You Get) –editorit [9].

Tällaiset työkalut ovat mahdollistaneet verkkosivujen luonnin myös henkilöille, jotka eivät osaa lainkaan tai hyvin vähän HTML:ää. Graafista editoria käyttämällä, käyttäjälle generoidaan valmis HTML-dokumentti eikä itse merkkaukseen tarvitse koskea lainkaan.

Pelkällä HTML-merkkauksella ei kuitenkaan ole mahdollista luoda kovinkaan monimutkaisia käyttöliittymiä verkkosivuille. Sivujen esitystavan eriyttämiseksi sisällöstä ja paremman tuen mahdollistamiseksi eri päätelaitteille kehitettiin CSS (Cascading Style Sheet), jonka ensimmäinen versio julkaistiin vuonna 1997 W3C:n (World Wide Web Consortium) toimesta. HTML-dokumentti voi sisältää yhden tai useamman linkin tyylitiedostoihin, jotka määrittelevät sivun ulkoasun. Tällä tavalla saavutetaan selkeä rakenteen ja ulkoasun erottelu säilyttäen silti suunnittelijan ohjakset sivun lopulliseen ulkoasuun. [10] HTML-elementeille voidaan määritellä tyyli joko `style` tai `class`-attribuutin avulla. Näistä ensimmäisellä elementille voidaan suoraan kirjoittaa tyyli, esimerkiksi

```
<p style=”color: red”>
```

Class-attribuutilla puolestaan kerrotaan, minkä CSS-luokan määritelmää elementti noudattaa. Nämä määritelmät ovat yleensä erillisissä tyylitiedostoissa, jotta aiemmin mainittu rakenteen ja ulkoasun erottelu toteutuisi kokonaisuudessaan. Toinen hyöty erillisistä tyylitiedostoista on mahdollisuus tarjota sama HTML-dokumentti eri päätelaitteille, mutta eri tyylitiedostoilla varustettuna. Ei ole myöskään kustannustehokasta tehdä omat sivut jokaiselle eri päätelaitteelle ja resoluutiolle. Tyylitiedostot eivät ole tärkeitä pelkästään esteettisen ulkonäön vuoksi; saavutettavuus on yhä tärkeämpi asia palvelujen siirtyessä webiin. Käyttäjälle pitää pystyä tarjoamaan ulkoasu, jota voi käyttää kyvyistään ja päätelaitteestaan riippumatta [11].

Web-ohjelmointi voidaan jakaa kahteen kategoriaan, sen mukaan suoritetaanko ohjelmaa selaimessa vai palvelimella. Web-sivuista tarvittiin dynaamisia ja interaktiivisia, joten HTML-dokumentteihin sallittiin sulauttaa yleensä JavaScriptillä toteutettuja skriptejä. Hyvin yleinen käyttötapaus selainpuolen ohjelmoinnille on syötekentän oikeellisuuden tarkastaminen, ennen kuin lomake lähetetään palvelimelle. [9] Tällä tavalla voidaan ennaltaehkäistä virheitä, joiden tarkastelu muuten tapahtuisi vasta palvelimella, ja tieto virheestä pitäisi palauttaa takaisin käyttöliittymälle. Jo selaimessa havaittu virhe nopeuttaa virheen korjaamista käyttäjälle, kun virheilmoitus voidaan näyttää välittömästi, eikä vasta lomakkeen lähetyksen jälkeen.

Kaikkea ei kuitenkaan voida tehdä selainpuolen ohjelmassa. Esimerkiksi tiedon hakeminen tietokannasta on pakko tehdä palvelinpuolen ohjelmassa [9]. Vaikka virheitä voidaan tarkastella selaimessa, pitää samat validoinnit tehdä myös palvelimella. Käyttäjä pystyy nimittäin ohittamaan selaimessa tapahtuvan syötteentarkastelun poistamalla JavaScriptin käytöstä selaimessa. Nykyään yleistyneisiin REST (Representational state transfer)-rajapintoihin voidaan lähettää pyyntöjä muualtakin kuin selaimesta, jolloin myöskään selaimessa tapahtuvaa validointia ei suoriteta.

3.2 Ajax

Perinteisesti kommunikaatio käyttöliittymän ja palvelimen välillä suoritettiin asiakas–palvelin-mallin mukaisesti lähettämällä dataa sisältävä lomake palvelimelle, joka palautti vastauksena uuden sivun. Tämä oli käyttäjäkokemuksen kannalta kankea tapa käyttää ohjelmaa, etenkin verrattuna työpöytäohjelmistoihin. Tähän ongelmaan kehittyi ratkaisuna vuonna 2005 Ajax (Asynchronous JavaScript + XML), jonka suurimpina esittelijöinä olivat muun muassa Google Maps ja Gmail [12].

Näiden ohjelmien viehättävyytenä pidettiin niiden laajaa ja rikasta interaktiivisuutta selaimessa, jota voitiin verrata työpöytäohjelmiin. Ajax itsessään ei ollut uusi teknologia eikä myöskään ohjelmointikieli. Se oli uusi ajattelutapa suunnitella ja toteuttaa web-ohjelmia joukolla olemassa olevia ja tunnettuja teknologioita:

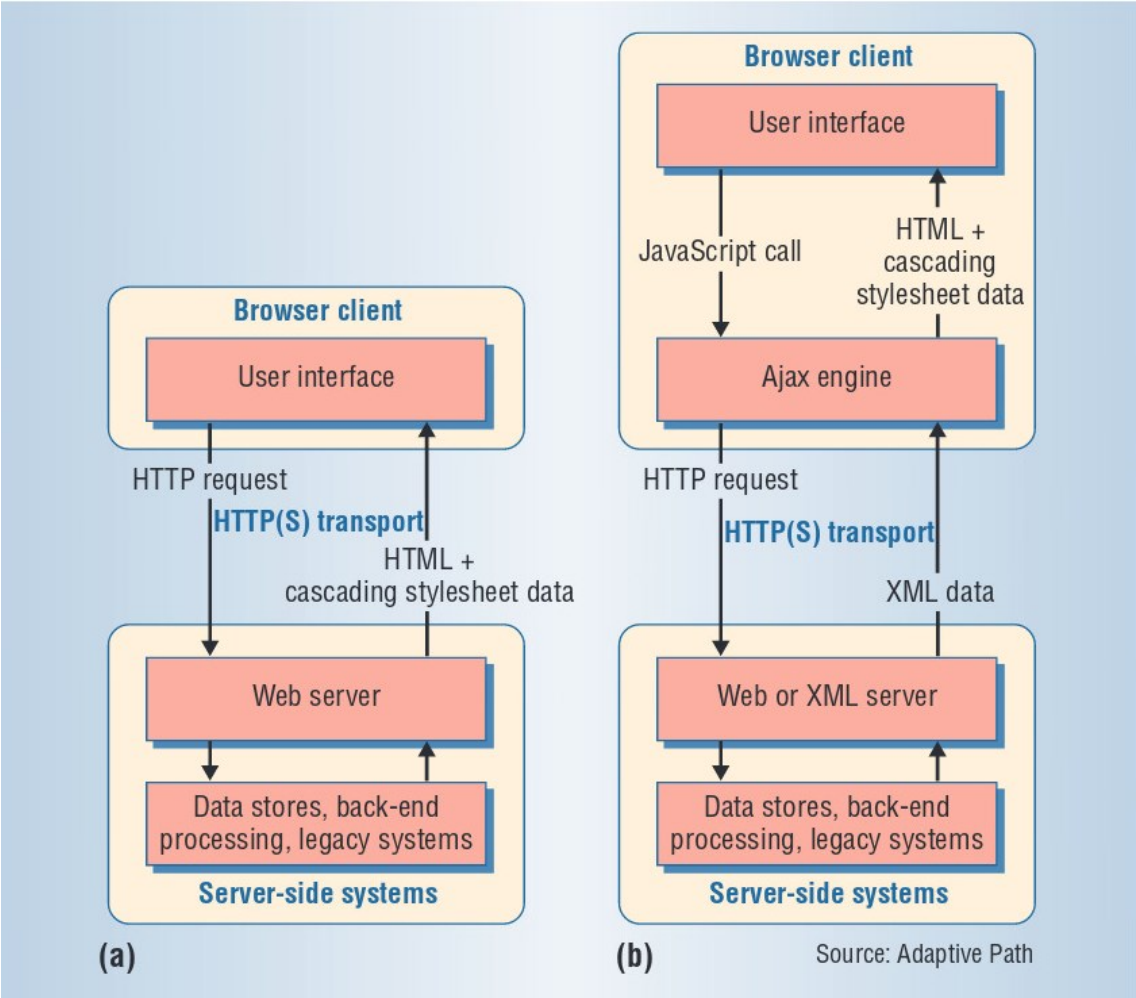
- HTML
- CSS
- DOM (Document Object Model): HTML mallintaminen olioina
- selainpuolen ohjelmointi: JavaScript
- asynkroninen tiedonsiirto ja kommunikointi palvelimen kanssa: XMLHttpRequestObject (XHR)
- tiedonsiirto-objektit: XML, XSLT, HTML, JSON
- tiedonsiirtoprotokollat: HTTP, HTTPS
- palvelinpuolen ohjelmointikielet: JSP, JSF, Perl, Ruby, PHP.

Nämä eivät ole uniikkeja teknologioita tai ohjelmointikieliä, vaan useimmin käytettyjä ja vapaan standardin omaavia ja ilmaisia käyttää. Ajax ei myöskään ole sidottu tiettyyn ohjelmointikieleen tai tiedonsiirtotapaan, vaan teknologioita voidaan vapaasti yhdistellä. [12]

Tärkeintä Ajaxissa on sen asynkroninen kommunikointi palvelimen kanssa. Tämä mahdollistaa pyyntöjen lähettämisen selaimesta palvelimme ja palvelimen vastauksen vastaanottamisen ilman sivun uudelleenlatausta. Tästä johtuen voidaan todeta Ajaxilla toteutettujen ohjelmien olevan suorituskyvyiltään tehokkaampia ja responsiivisempia käyttäjän toiminnoille kuin perinteisesti toteutetut ohjelmat [12]. Käyttöä nopeuttaa myös mahdollisuus kuljettaa vähemmän dataa, sillä Ajaxissa voidaan pyytää ja lähettää vain ja ainoastaan tarvittava määrä dataa. Ajaxin käyttöä kannustaa myös sen laaja tuki; samaa ohjelmaa voidaan ajaa webissä eri järjestelmillä riippumatta alustasta. [12] Julkaisunsa jälkeen, Ajax oli heti käyttövalmiina selaimissa, eikä sille tarvinnut odottaa selainvalmistajien lisäämää tukea, kuten tuleville HTML-, CSS- ja JavaScript-versioille.

Kuvassa 1 vasemman puoleisessa mallissa (a) käyttäjän toimenpiteet suorittavat HTTP-pyynnön web-palvelimelle, joka prosessoi pyynnön ja palauttaa HTML-sivun asiakkaalle. Ohjelma on lukitussa tilassa, kunnes vastaus saadaan. Oikealla puolella (b) Ajax-

ohjelmat luovat selaimessa ajettavan JavaScript-pohjaisen moottorin. Tämä moottori sieppaa käyttäjän syötteet, näyttää pyydetyt materiaalit ja hallitsee interaktiivisuutta selaimen puolella. Jos moottori tarvitsee lisää dataa, materiaali pyydetään palvelimelta tausta-ajona, jolloin käyttäjä voi samanaikaisesti jatkaa ohjelman käyttöä. [13]



Kuva 1 (*a) Perinteinen tapa kommunikoida asiakkaan ja palvelimen välillä. (b) Kommunikaatio käyttäen AJAX -arkkitehtuuria. [13]*

Kuvaa 1 katsomalla voisi virheellisesti luulla ylimääräisen välikäden lisäämisen asiakkaan ja palvelimen välille hidastavan ohjelman toimintaa, vaikka totuus on päinvastainen, kuten aiemmin jo alustavasti perusteltiin. Käyttäjän toiminnot, jotka normaalisti aiheuttaisivat HTTP-pyynnön, muodostavatkin JavaScript-kutsun Ajax-moottorille. Moottori voi itsenäisesti käsitellä sellaiset toiminnot, jotka eivät välttämättä vaadi palvelimelta toimintoja, kuten yksinkertainen validointi tai muistissa olevan datan muokkaaminen. Jos moottori tarvitseekin palvelimelta apua, kuten datan lähettämien prosessoitavaksi, se tekee nämä pyynnöt asynkronisesti ilman käyttöliittymän jumiuttamista. [14]

3.3 JavaScript

Alun perin JavaScript luotiin skriptikieleksi ja se sisällytettiin Netscape Navigator selaimeen vuonna 1995 melko yksinkertaisten animaatioiden ja skriptien tukemiseksi. Siitä lähtien JavaScriptin käyttö on laajentunut koskemaan huomattavasti isompia tehtäviä ja ohjelmistoja, joissa on satoja tuhansia rivejä ohjelmakoodia. [1]

Pian JavaScriptin julkaisun jälkeen, Microsoft kehitti täysin samanlaisen ohjelmointikielen ja nimesi sen JScriptiksi. Osittain tästäkin johtuen, Netscape halusi standardoida JavaScriptin ja pyysi standardointiorganisaatio Ecma Internationalia ylläpitämään standardia. Koska silloinen Sun, joka nykyisin tunnetaan yrityksenä Oracle, oli rekisteröinyt JavaScript-tavaramerkin, valittiin standardin nimeksi ECMAScript (ECMA-262). [15] JavaScript on siis ohjelmointikieli, joka noudattaa ECMAScript-standardia. Puhuttaessa JavaScriptin historiasta on viisaampaa viitata ECMAScript-standardin versioihin.

Vaikka JavaScriptin käyttö on ollut nousussa, sillä ei aina ole ollut kovinkaan hyvä maine ohjelmointikielenä. JavaScriptiä on pidetty ”leluohjelmointikielenä” [1] ja halveksuttu, koska se eroaa muista ohjelmointikielistä [16]. Kun perinteisillä ohjelmointikielillä ohjelmoinut ohjelmoija saa tehtäväkseen työskennellä JavaScriptillä toteutetun web-palvelun kanssa, voivat JavaScriptin eroavaisuudet ilmetä yllätyksenä ja ärsytyksenä. JavaScript esimerkiksi suorittaa tyyppimuuntelua (type coercion), mikä johtaa joidenkin operaatioiden yllättävään lopputulokseen. Yksinkertainen esimerkki tästä on operaation

```
false + true
```

jonka lopputulos on numero 1. Tässä operaatiossa suoritetaan implisiittinen muuntelu, eli automaattisesti tapahtuva tyyppimuuntelu. Ennen lausekkeen varsinaista suorittamista `false` muutetaan numeroksi 0 ja `true` numeroksi 1.

JavaScript sai alkunsa erittäin nopeasti, jotta se saatiin sisällytettyä Netscape Navigator-selaimeen mahdollisesti muiden huonompien kielien sijasta. JavaScript on saanut vaikutteita monesta eri ohjelmointikielestä, jotka ovat myös hyvin erityyppisiä. Esimerkiksi Javasta lainattiin syntaksi ja alkuperäistyypit, Perlistä ja Pythonista merkkijonot, taulukot ja säännölliset lausekkeet. JavaScript on sekoitus olio-ohjelmointia ja funktionaalista ohjelmointia. Funktionaalisia piirteitä ovat muun muassa korkeamman tason funktiot (higher-order functions) ja sisäänrakennetut funktiot kuten `map` ja `reduce`. Olio-ohjelmointia hyödynnetään olioita ja periytymistä käyttämällä. Voidaankin sanoa, muissa ohjelmointikielissä opetellaan kielen ominaisuuksia, mutta JavaScriptissä usein opitaan ohjelmointimalleja (patterns). [16]

Etenkin JavaScriptin aiemmissa versioissa on aina puuttunut ohjelmointikielen kannalta tärkeitä ominaisuuksia, kuten poikkeustenkäsittely ja lohkosidonnaiset muuttujat (block scoped variables), jotka on tuotu mukaan kieleen vasta myöhemmin. Toisaalta kielessä on paljon ominaisuuksia, joiden avulla ongelmat voidaan kiertää, ennen kuin ne korjataan

seuraavaan kielen julkaisuun. [16] JavaScriptin suosion kasvaessa, ECMAScript-standardiin on lisätty suuri määrä ominaisuuksia, jotka helpottavat suurten ohjelmistojen toteutusta ja ylläpidettävyyttä. Vuonna 2015 julkaistu ECMAScript 2015 (ES2015) versio lisäsi standardiin aiemmin vain lähinnä palvelinpuolen ja työpöytäohjelmoinnista tutut syntaksit kuten luokat ja moduulit. JavaScriptissä oli tätä aiemmin mahdollista käyttää olio-ominaisuuksia vain kielen prototyyppisen periytymisen avulla. Asynkronisen ohjelmoinnin helpottamiseksi ES2015 lisäsi myös promise-toiminallisuuden, jonka toimintaperiaate on samankaltainen kuin futuuri muissa ohjelmointikielissä. Lisää asynkronisia ominaisuuksia, kuten `async` ja `await`, tarjoaa ECMAScript 2017.

Ohjelmistojen kehittämistä helpottamaan ja JavaScriptin heikkouksia paikkaamaan on luotu uusia erillisiä ohjelmointikieliä, jotka tarjoavat JavaScriptistä puuttuvia ominaisuuksia. Tällaiset kielet, kuten Microsoftin kehittämä TypeScript ja Facebookin Flow, tarjoavat kehitysaikana muun muassa virheettömyyttä ja ylläpidettävyyttä parantavia ominaisuuksia, kuten staattisen tyypityksen. Nämä ohjelmointikielet kuitenkin käännetään ECMAScript-standardin mukaiseksi JavaScriptiksi, jolloin ohjelma toimii oikein myös ajonaikana.

4. TYPESCRIPT APUNA JAVASCRIPTIN ONGELMIIN

Kuten luvussa 3.3. todettiin, JavaScriptillä laajojen ohjelmistojen rakentaminen on hankalaa, johtuen luvussa 2 havaituista dynaamisen tyyppijärjestelmän heikkouksista. TypeScript-ohjelmointikieli on staattisesti tyypitetty laajennos JavaScriptille ja se pyrkii paikkaamaan JavaScriptin puutteita.

4.1 TypeScript

TypeScript rikastaa JavaScriptiä muun muassa moduulijärjestelmällä, luokilla, rajapinnoilla ja staattisella tyyppijärjestelmä. Moduuli- ja tyyppijärjestelmät on luotu joustaviksi ja helppokäyttöisiksi, sillä TypeScript pyrkii tarjoamaan kevytrakenteista avustusta ohjelmoijille. Lisäksi TypeScript säilyttää JavaScriptistä tutut ohjelmointitavat ja idiomit työskentelyn helpottamiseksi. Staattisesta tyyppijärjestelmästä johtuen, myös kehitysympäristöt pystyvät entistä paremmin auttamaan ohjelmoijaa tarjoamalla työkaluja, joita on totuttu käyttämään kielillä kuten C# ja Java. Esimerkiksi oliolle pystytään ehdottamaan, mitä metodeja sille voidaan kutsua. [17] Staattinen tyyppijärjestelmä tuo mukanaan kappaleessa 3.1.1. listattuja hyötyjä, kuten tyyppivirheiden ennaltaehkäiseminen. Ajonaikana tapahtuvat tyyppivirheet ovatkin olleet JavaScript-ohjelmoinnissa ongelmana, joten TypeScriptin tuoma tyyppitarkastelija on erittäin tervetullut. Toinen tärkeä TypeScriptin tuoma ominaispiirre on sen kääntäjä (transpiler), joka kääntää TypeScript-koodista JavaScript-koodia [18]. Toisaalta myös kaikki oikeellinen JavaScript -koodi on suoraan kelvollista TypeScript-koodia [19].

4.1.1 Suunnittelun tavoitteet

TypeScriptin kehittymistä ja muovautumista nykyisen kaltaiseksi ovat ohjanneet sen suunnittelutavoitteet ja arkkitehtoniset ratkaisut. Näitä ovat muun muassa:

- Tunnistaa sellaiset JavaScript-rakenteet, jotka ovat todennäköisesti virheitä. Microsoft päätyi staattiseen tyyppijärjestelmään ratkaisuna tähän vaatimukseen.
- Tarjota laaja yhteensopivuus olemassa olevan JavaScript-koodin kanssa. TypeScript on JavaScriptin ylijoukko (superset), eli toimivan JavaScript-ohjelman tulee olla myös toimiva TypeScript-ohjelma.
- Tarjota jäsentämismekanismi suurelle koodimäärälle. TypeScript lisää luokkapohjaisen olio-ohjelmoinnin, rajapinnat ja moduulit. Näiden avulla koodia pystytään rakenteellistamaan paremmin ja siitä tulee helpommin ylläpidettävää.
- Olla lisäämättä ajonaikasia kustannuksia (overhead) käännettyihin ohjelmiin.

TypeScriptillä ohjelmoitaessa on hyvä erottaa suunniteluaikainen ohjelmakoodi (design time code) ja ajonaikainen ohjelmakoodi (runtime code). Suunnitteluaikainen koodi on siis TypeScript-koodia, jota ei ole vielä käännetty. Ajonaikainen koodi on puolestaan TypeScriptistä käännettyä JavaScript-koodia. Moni TypeScriptin ominaisuus, esimerkiksi tyypit ja rajapinnat, ovat olemassa vain suunniteluaikaisessa koodissa. Kääntäjä pitää huolen siitä, että tuotettu koodi on laillista JavaScriptiä. [19]

4.1.2 Tyypit

Tyypit ovat TypeScriptissä nimensä mukaisesti olennaisessa roolissa. Ne ovat ohjelmoijalle suunnitteluaikaisen koodin apuväline. TypeScriptin staattisen tyyppijärjestelmän ansiosta muuttujille ja funktioille voidaan antaa eksplisiittiset tyypit, kuten `number` tai `string`. Tyyppien muodossa annetut rajoitteet mahdollistavat muun muassa aputyökalujen käytön kehityksessä ja kääntäjän tarjoaman varmennuksen ohjelman oikeellisuudesta. Ohjelmoija pystyy ilmaisemaan koodissaan aikeensa sekä itselleen että muulle kehitystiimille. [19]

Muuttujan tyyppi pystytään joissain tapauksissa päättelemään, vaikka sitä ei ole erikseen ilmoitettu. Tämä automaattinen tyyppipäättely on joissain tapauksissa mahdotonta, joten muuttujille voidaan eksplisiittisesti antaa tietty tyyppi. Tämä tapahtuu käytännössä kirjoittamalla muuttujan nimen perään kaksoispiste ja haluttu tyyppi.

```
let luku;                    // tuntematon (any) tyyppi
let luku = 0;                // number (tyyppi päätelty)
let luku : number;           // number
let luku : number = 0;       // number
```

Yllä olevasta esimerkistä huomataan, kuinka tyypitys toimii TypeScriptissä ja kuinka tyyppi voidaan päätellä esimerkiksi sille alustetun arvon perusteella. TypeScript siis käyttää muuttujien tyypin päättelemiseksi tyyppipäättelyä (type inference). [19] On myös tilanteita, jolloin ohjelmoija ei halua pakottaa muuttujalle mitään tyyppiä. Tähän TypeScriptissä on ratkaisuna tyyppi `any`. Tämä tyyppi ilmaisee mitä tahansa JavaScriptin arvoa ja se toimii samalla tavalla kuin (tyypittömät muuttujat) JavaScriptissä [19]. `Any`-tyyppiä käyttämällä, ohjelmoija voi halutessaan nauttia dynaamisen tyypityksen tuomista eduista. Yleisempi käyttötarkoitus on kuitenkin kuvata tällä tyypillä muuttujia, joiden tyyppiä ei voida tietää [20]. Tällaisia arvoja voi tulla dynaamisesta sisällöstä, kuten käyttäjän omat tai kolmannen osapuolen kirjastot. Tällaisia tapauksissa halutaan jättää tyyppitarkastelu tekemättä ja antaa arvojen ohittaa käännösaikainen tarkistus. `Any`-tyyppi on myös hyödyllinen, kun olemassa olevia JavaScript-ohjelmia muutetaan pala kerrallaan TypeScript-ohjelmiksi [19]. Myös siirtyminen perinteisestä JavaScript-maailmasta TypeScriptillä ohjelmien kehittämiseen helpottuu. Kehittäjä voi hiljalleen merkata tyyppejä yhä tarkemmin oppiessaan kieltä. Tyypitysten vapaaehtoisuudesta johtuen, TypeScriptiä voidaan kutsua myös vapaaehtoisesti tyypitetyksi ohjelmointikieleksi (optionally typed). Vapaaehtoinen

tyyppijärjestelmä (optional type system) on sellainen tyyppijärjestelmä, jolla ei ole vaikutusta ohjelmointikielen ajonaikaiseen semantiikkaan, eikä vaadi tyyppimerkintöjä syntaksissa [21].

4.1.3 Rajapinnat ja luokat

Olio-ohjelmointi helpottaa suurten ohjelmien suunnittelua ja ylläpitoa. JavaScriptillä toteuttavien ohjelmistojen kasvaessa, perinteinen luokkaperustainen olio-ohjelmointi olisi toivottu helpotus. Vaikka JavaScript onkin valmiiksi olio-ohjelmointikieli, se ei käytä luokkia vaan turvautuu olioihin, jotka perivät suoraan toisten olioiden ominaisuuksia. Tätä kutsutaan prototyyppiseksi periytymiseksi [16]. ECMAScript 2015 -standardiin lisättiin tuki luokille [19]. Tämän kaltaiset luokat ovat olleet TypeScriptissä käytettävissä jo siitä lähtien, kun kieli julkaistiin vuonna 2012. TypeScriptin kääntäjä osaa luoda ECMAScript-standardin mukaisista luokista rakenteita, jotka noudattavat vanhempaa ECMAScript-standardia käyttämällä olioiden prototyyppiominaisuutta [22]. Tämä oli hyödyllinen toiminto aikana ennen kuin selainvalmistajat lisäsivät ES2015-tuen. Tänä päivänä TypeScriptin luokat ovat hyödyllisiä, kun halutaan ohjelman tukevan myös vanhempia selaimia, joissa ei ole ES2015-ominaisuuksia.

Seuraavassa esimerkissä on ES2015-standardin mukainen luokka, joka koostuu kahdesta jäsenmuuttujasta, rakentajasta ja metodista. TypeScriptissä metodit ja jäsenmuuttujat ovat oletuksena julkisia.

```
class Dog implements IDog {
    name: string;
    color: string;
    constructor(name: string, color: string) {
        this.name = name;
        this.color = color;
    }
    bark() {
        console.log(this.name + “ gently barks: Woof woof!”);
    }
}
```

Periyttäminen on olennainen ja tehokas käsite olio-ohjelmoinnissa. TypeScriptissä luokkia voidaan laajentaa periyttämällä niistä uusia luokkia, jolloin aliluokat saavat käyttöönsä kantaluokan jäsenet ja metodit.

```
class HerdingDog extends Dog {
    stockType: string;
    constructor(name: string, color: string, stockType: string) {
        super(name, color);
        this.stockType = stockType;
    }
```

```
    bark() {
        super.bark();
        console.log(“The ” + this.stockType “ is now scared.”);
    }
    gatherStock() {
        console.log(this.name + “ gathers the stock into a group”);
    }
}
```

Periytyminen tapahtuu avainsanalla `extends`. `Super`-avainsanalla voidaan laajentaa kantaluokassa olevaa metodia, tässä tapauksessa `bark()`-metodia. Tätä kutsutaan kantaluokan metodin korvaamiseksi (method overriding). Tällä tavalla aliluokka voi siis tarjota oman toteutuksen metodille, joka on jo määritelty kantaluokassa. [19]

TypeScriptissä on myös käytettävissä rajapinnat (interface) [20], joita luokat voivat toteuttaa [19]. Rajapintaa voidaan pitää ”moduulin tyyppinä”, joka tarjoaa yhteenvedon moduulin tarjoamista palveluista – se on käyttäjän ja toteuttajan välinen sopimus. Laajan järjestelmän rakentaminen käyttäen selkeitä rajapintoja moduuleille johtaa abstraktin suunnittelun suuntaan, jossa rajapinnat suunnitellaan erillään niiden tulevista toteutuksista. [3] Aiemmin esitelty `Dog` -luokka toteuttaa seuraavanlaisen rajapinnan:

```
interface IDog {
    name: string;
    color?: string;
    bark(): void;
}
```

TypeScriptissä rajapinnat voivat toteuttaa toisia rajapintoja, sekä laajentaa toisia rajapintoja tai luokkia [19]. Kysymysmerkki muuttujan nimen perässä ilmaisee muuttujan olevan valinnainen; rajapinnan toteutuksessa sitä ei ole pakko määrittää. Tämä voi olla hyödyllinen, kun halutaan kertoa jonkin jäsenmuuttujan ehkä löytyvän ja samalla voidaan estää sellaisten esittelemättömien muuttujien käyttö.

```
let puppy = new Dog(”spot”, ”black”);
dog.clor = ”brownish”;
```

Tämä koodi johtaisi käännösaikaiseen virheeseen, sillä `IDog` -rajapinnassa (eikä `Dog`-luokassa) ei ole `clor` nimistä jäsenmuuttujaa. JavaScriptissä tämän kaltainen virheellinen koodi olisi hyvin voinut päätyä ajoon (jossa se ei kuitenkaan olisi aiheuttanut ajonaikaista virhettä, sillä JavaScriptissä olioon voidaan ajonaikana lisätä uusia jäsenmuuttujia).

Abstraktia suunnittelua voidaan TypeScriptissä harjoittaa myös käyttämällä abstrakteja luokkia. Nämä ovat käytännössä abstrakteja kantaluokkia, joista ei luoda olioita, vaan näistä periytetään muita luokkia. Abstrakti luokka voi sisältää metodeja ja abstrakteja

metodeja. Abstraktit metodit muistuttavat paljon rajapintoja, molemmat esimerkiksi määrittelevät metodin allekirjoituksen (signature of a method) mutta eivät sen runkoa. Tällaiset metodit on kuitenkin pakko määritellä abstraktin luokan aliluokissa. Normaalit metodit sen sijaan voivat sisältää toteutuksen, jota aliluokat sitten käyttävät tai uudelleenmäärittelevät.

4.1.4 Moduulit ja nimiavaruudet

ECMAScript 2015 version myötä JavaScriptissä on käytössä moduulit. TypeScriptin versiosta 1.5 alkaen, moduulit toimivat ECMAScript-standardin mukaisesti. Moduulit suoritetaan omalla näkyvyysalueellaan globaalin näkyvyysalueen sijaan. Moduulin sisällä esitellyt muuttujat, funktiot, luokat ja niin edelleen, eivät ole näkyviä moduulin ulkopuolelle, ellei niitä ole erikseen viety (export). Jotta toisen moduulin vietyjä palveluita voidaan käyttää, pitää moduuli tuoda (import). [20] Moduuleja ei ladata HTML:än `<script>` tagin avulla, vaan tätä varten käytetään erillistä moduulilatajaa (module loader). Tällaisen työkalun avulla saadaan paremmin hallittu kontrolli, miten moduuleja ladataan. Moduuleja voidaan ladata asynkronisesti, ja niitä voidaan yhdistää yhdeksi optimoiduksi tiedostoksi, mikä nopeuttaa ohjelman toimintaa. [19] Etenkin laajoissa JavaScript-ohjelmissa voidaan ohjelman latausta nopeuttaa, kun ohjelmakoodin hakeminen viivästytetään siihen asti, kunnes sitä todella tarvitaan.

Nimiavaruudet ovat moduulien kaltainen ominaisuus, jota käytetään lähinnä lähdekoodin järjestelyyn. Isoissa ohjelmistoissa koodimäärä on suuri ja uusille toiminnoille voi helposti tulla sama nimi kuin jollekin vanhalle toiminnolle (naming collision). Nimiavaruuksien avulla esimerkiksi saman nimisten luokkien törmääminen voidaan välttää. Samalla ohjelmakoodin luettavuus ja ymmärrettävyys paranevat. [19] Nimiavaruuden toiminnot voidaan jakaa useaan eri tiedostoon ja silti käyttää aivan kuin ne olisivat vain yhdessä paikassa. Jotta kääntäjä ymmärtäisi jaettujen tiedostojen väliset suhteet, pitää tiedostoihin lisätä `<reference>`-niminen tagi.

4.1.5 Kääntäjä

TypeScriptin kääntäjän kaksi päätehtävää ovat ohjelmakoodin kääntäminen (transpiling) ja tyyppitarkastelu. Koodi käännetään TypeScriptistä JavaScriptiksi, eli korkean tason ohjelmointikielestä toiseksi korkean tason ohjelmointikieleksi. Tyyppitarkastelijan tehtävänä on havaita koodista ristiriitoja, kuten epäsopivan tyyppisen muuttujan arvon sijoittaminen toisen tyyppiseen muuttujaan. Kääntäjä suorittaa useita säädettävien asetusten mukaisia toimintoja, kuten kommenttien poisto, ja muuttaa TypeScript-spesifiset ominaisuudet halutun ECMAScript-version mukaiseksi JavaScriptiksi. [18] Kääntäjän asetuksien määrittäminen tapahtuu `tsconfig.json`-nimisessä tiedostossa, joka sijoitetaan projektin juureen. Tässä tiedostossa voidaan myös määrittää käännettävät tiedostot tai kansiot josta tiedostot löytyvät.

Aiemmin esitelty Dog-luokka käännettynä JavaScriptiksi kohdentaen ECMAScript 5 -standardiksi on seuraavanlainen:

```
var Dog = /** @class */ (function () {
    function Dog(name, color) {
        this.name = name;
        this.color = color;
    }
    Dog.prototype.bark = function () {
        console.log(this.name + " gently barks: Woof woof!");
    };
    return Dog;
}());
```

Tästä huomataan, kuinka suunnitellun aikana voidaan käyttää ohjelmointityötä helpottavia käsitteitä, kuten luokkia, jotka kääntyvät JavaScriptiksi. Usein käännöksessä hyödynnetään JavaScriptin sulkeita ja prototyyppiä.

4.2 Kielten ominaisuuksien vertailu

TypeScriptiä ei ole mielekästä verrata vain JavaScriptiin yleisesti, sillä siihen kehitetään koko ajan uusia ominaisuuksia ECMAScript-standardin myötä. Taulukossa 1 on vertailtu kahta eri ECMAScript standardin mukaista JavaScript versiota ja TypeScriptiä. Taulukossa on pääosin keskitytty tyyppijärjestelmiin ja olio-ominaisuuksiin. ES5 julkaistiin vuonna 1999 ja ES2015 vuonna 2015. ES2015 toi mukanaan joukon uusia ominaisuuksia, joista merkittävimmät on listattu taulukkoon.

Taulukko 1. *Ominaisuuksien vertailua*

Ominaisuus	JavaScript (ES5)	JavaScript (ES 2015)	TypeScript
JSON tuki	x	x	x
Luokat		x	x
Moduulit		x	x
Nuoli-funktiot		x	x
Argumenttien oletusarvot		x	x
Tyypit			x
Rajapinnat			x
Geneerisyys			x

Nuoli-funktiot ovat lausekkeita funktioille, mutta lyhyemmällä syntaksilla. Nuoli-funktion määrittelyssä esiintyy nuolioperaattori (=>). Funktion rungossa sijaitseva this viit-

taa siihen luokkaan, jossa se sijaitsee. ES5-versiossa `this`-sanan viittaama kohde vaihtelee riippuen muun muassa suoritusympäristöstä (esimerkiksi selain vai Node). Eroa on selvennetty alla olevilla esimerkeillä.

```
// ES2015
class Car {
    manufacturer = “Ford”;
    model = “Mondeo”;
    let getFullModelName = () => {
        //this viittaa luokan sisään
        return this.manufacturer + “ “ + this.model;
    }
}

// ES5
function test() {
    return this;
}
// Selaimessa
console.log( test() ); // window
// Nodessa
console.log( test() ); // global
```

TypeScript sisältää geneerisyys-nimisen ominaisuuden, jota ei tyyppien puuttumisen vuoksi voida toteuttaa ECMAScript standardiin. Geneerinen komponentti voi toimia eri tyypeillä sen sijaan että näille tyypeille tehtäisiin erilliset toteutukset.

```
function getArrayType<T>(myArray: T[]) : T {
    console.log(“size is: “ + array.length);
    return myArray;
}

let numberArray = [1,2,6];
console.log( getArrayType<number>(numberArray) ); // number
```

Yllä olevassa esimerkissä on esitelty geneerinen funktio, joka palauttaa argumenttina annetun taulukon tyypin. Se myös kirjoittaa taulukon pituuden konsoliin. `Length`-jäsenmuuttujan käyttö tässä tapauksessa on turvallista, sillä sen tiedetään löytyvän varmasti minkä tahansa tyyppisestä taulukosta. [20]

4.3 TypeScriptin tarjoamat hyödyt

ECMAScript-standardin kehittyessä, sen uudet ominaisuudet helpottavat JavaScriptillä ohjelmien tuottamista. Yhä useammat ominaisuudet, jotka olivat aiemmin käytettävissä vain TypeScriptin kaltaisten työkalujen avulla, ovat nyt suoraan käytettävissä JavaScriptissä. Tämä vähentää hieman TypeScriptin tarjoamia hyötyjä, mutta ei missään nimessä

tee niistä turhia. TypeScriptin tarjoama staattinen tyyppijärjestelmä helpottaa virheiden havaitsemista [23] ja parantaa ohjelmistojen ylläpidettävyyttä [4].

Z. Gao, C. Bird, E.T. Barr suorittivat empiirisen tutkimuksen staattisen tyypityksen hyödyistä JavaScript-ohjelmakoodin virheiden ennaltaehkäisyssä. Tutkimuksessa käytettiin julkisesti saatavilla olevia ohjelmistoprojekteja. Tämä mahdollisti projektien versionhallinnasta bugikorjauksien löytämisen. Tutkimuksessa otettiin tarkasteltavaksi korjausta edeltävä versio ohjelmakoodista, jonka siis tiedettiin olevan viallinen. Tähän vialliseen koodin lisättiin muuttujille tyypitykset ja tutkittiin jääkö virhe kiinni käännettäessä. Staattisen tyypityksen apuvälineinä tutkimuksessa käytettiin TypeScript versiota 2.0 ja Facebookin kehittämää Flow versiota 0.30. Tutkimuksessa havaittiin, että molemmat käytetyt ohjelmointikielet havaitsevat helposti tyyppien yhteensopimattomuuden, mikä ei olisi JavaScriptissa mahdollista. Tutkimuksessa käytiin läpi 400 julkista bugia. Näistä TypeScript ja Flow molemmat havaitsivat lopulta onnistuneesti 60 bugia. [23]

Staattisen tyyppijärjestelmän lisäksi TypeScriptin hyöty ilmenee tulevissa ECMAScript-versioissa. Kehittäjät voivat käyttää toimintoja, jotka ovat julkaistussa tai vielä julkaisemattomassa ECMAScript -standardissa mutta eivät selainten tukemia.

5. YHTEENVETO

Webistä on tullut yhä suositumpi ympäristö uusille ohjelmistoille ja vanhoja työpöytäohjelmia siirretään jatkuvasti webiin. JavaScriptistä on tullut näiden ohjelmien selainpuolen de facto -ohjelmointikieli ja sen suosion kasvu on jatkunut jo yli kaksi vuosikymmentä, eikä merkkejä suosion laskulle ole. Tämä on kuitenkin samalla aiheuttanut kritisointia, sillä JavaScriptiä ja sen heikkouksia on (osittain turhaankin) moitittu.

Uudet ECMAScript-versiot tuovat mukanaan kehitystä helpottavia ja uusia mahdollisuuksia mahdollistavia ominaisuuksia. Nämä eivät kuitenkaan ratkaise ongelmia, jotka ovat JavaScriptin ideologiassa itsessään. Tällaisia ovat muun muassa dynaamisesta tyypityksestä johtuva automaattisesti tapahtuva tyyppimuuntelu ja hämmentävä sekoitus eri ohjelmointikielistä saatuja vaikutteita.

Tyyppijärjestelmä määrittelee suuresti ohjelmointikieltä. Dynaaminen tyyppitarkastelu mahdollistaa ohjelmien toteuttamisen vapaammin, toisin kuin staattinen tyyppitarkastelu, joka saattaa rajoittaa ohjelmoijaa jopa liikaa. Tutkimuksissa on kuitenkin havaittu staattisen tyypityksen tuovan tiettyjä tärkeitä etuja verrattuna dynaamisen tyypitykseen. Muun muassa ohjelmistojen ylläpidettävyys paranee, mikä on JavaScriptillä toteutettujen ohjelmistojen koon kasvaessa tärkeä yksityiskohta.

TypeScript on yksi vaihtoehto helpompaan, virheettömämpään ja ylläpidettävämpään JavaScript kehitykseen. Se tarjoaa JavaScript -kehitykseen staattisen tyyppijärjestelmän, jonka käyttö on kuitenkin vapaaehtoista. Ohjelmoijalla on näin mahdollisuus käyttää molempien tyyppijärjestelmien hyviä piirteitä ja ohittaa huonot piirteet.

6. LÄHTEET

[1] A. Taivalsaari, T. Mikkonen, M. Anttonen, A. Salminen, The death of binary software: End user software moves to the web, Proceedings - 9th International Conference on Creating, Connecting and Collaborating through Computing, C5 2011, pp. 17-23.

[2] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Computing Surveys (CSUR), Vol. 17, Iss. 4, 1985, pp. 471-523. Saatavissa (viitattu 20.2.2018): https://www.scopus.com/inward/record.uri?eid=2-s2.0-0022333721&doi=10.1145%2f6041.6042&partnerID=40&md5=505609351ba29884e20051a8c09de303.

[3] B.C. Pierce, Types and programming languages, MIT Press, Cambridge, Mass.[u.a.], 2002, pp. 1-648.

[4] S. Kleinschmager, S. Hanenberg, R. Robbes, É Tanter, A. Stefik, Do static type systems improve the maintainability of software systems? An empirical study, IEEE International Conference on Program Comprehension, pp. 153-162.

[5] L Tratt, Chapter 5 Dynamically Typed Languages Advances in Computers Vol. 77, 2009, pp. 149-184.

[6] K.C. Louden, K.A. Lambert, Programming languages, 3 ed. ed. Course Technology Cengage learning, Boston, 2012, pp-1-704.

[7] R.E. Johnson, B. Foote, Designing reusable classes, Journal of Object-Oriented Programming, Vol. 1, Iss. 2, 1988, pp. 30, 35. Saatavissa (viitattu 23.2.2018): https://www.scopus.com/inward/record.uri?eid=2-s2.0-0024028289&partnerID=40&md5=798f4863570d7886a55993b298c20426.

[8] M. Handley, Why the Internet only just works, BT Technology Journal, Vol. 24, Iss. 3, 2006, pp. 119-129. Saatavilla (viitattu 18.2.2018): https://www.scopus.com/inward/record.uri?eid=2-s2.0-33845227912&doi=10.1007%2fs10550-006-0084-z&partnerID=40&md5=00d3c1c94a7f1021ee2a7c712efda46a.

[9] G. Vossen, F. Schönthaler, S. Dillon, The Web from Freshman to Senior in 20+ Years (that is, A Short History of the Web), in: G. Vossen, F. Schönthaler, S. Dillon (ed.), The Web at Graduation and Beyond: Business Impacts and Developments, Springer International Publishing, Cham, 2017, pp. 1-52.

[10] G.J. Badros, A. Borning, K. Marriott, P. Stuckey, Constraint cascading style sheets for the web, UIST (User Interface Software and Technology): Proceedings of the ACM Symposium, pp. 73-82.

[11] M. Butler, F. Giannetti, R. Gimson, T. Wiley, Device independence and the web, IEEE Internet Computing, Vol. 6, Iss. 5, 2002, pp. 81-86. Saatavissa (viitattu 20.2.2018): https://www.scopus.com/inward/record.uri?eid=2-s2.0-0036755396&doi=10.1109%2fMIC.2002.1036042&partnerID=40&md5=e02a09055be682e2cb209de39b741e10.

[12] J.S. Zepeda, S.V. Chapa, From desktop applications towards ajax web applications, 2007 4th International Conference on Electrical and Electronics Engineering, ICEEE 2007, pp. 193-196.

[13] L.D. Paulson, Building rich Web applications with Ajax, Computer, Vol. 38, Iss. 10, 2005, pp. 14-17. Saatavissa (viitattu 10.2.2018): https://www.scopus.com/inward/record.uri?eid=2-s2.0-27344447954&doi=10.1109%2fMC.2005.330&partnerID=40&md5=8b056eb1a73d49f5d26e9a3318c92f18.

[14] Garret Jesse James Ajax: A New Approach to Web Applications, http://adaptivepath.org/ideas/ajax-new-approach-web-applications/.

[15] A. Rauschmayer, Speaking JavaScript : An in-Depth Guide for Programmers, 1st ed. O'Reilly Media, Incorporated, Sebastopol, 2014, pp. 1-460.

[16] D. Crockford, JavaScript: The Good Parts, illustrated edition ed. O'Reilly, Sebastopol, 2008, pp. 1-172.

[17] G. Bierman, M. Abadi, M Torgersen, Understanding TypeScript, ECOOP 2014 – Object-Oriented Programming, 2014, pp. 257-281.

[18] I.G.d. Wolff, TypeScript Blueprints, 1st ed. Packt Publishing, GB, 2016, pp. 1-288.

[19] R.H. Jansen, V. Vane, I.G.d. Wolff, V. Vane, I.G.d. Wolff, TypeScript: Modern JavaScript Development, 1st ed. Packt Publishing, Birmingham, 2016, pp. 1-841.

[20] TypeScript Handbook, verkkosivu Saatavissa (viitattu 21.2.2018): https://www.typescriptlang.org/docs/home.html

[21] G. Bracha, Pluggable Type Sytems, 2014, Saatavissa (viitattu 24.3.2018) http://bracha.org/pluggableTypesPosition.pdf, pp.1-6.

[22] Microsoft TypeScript Language Specification, verkkosivu Saatavissa (viitattu 21.2.2018): https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md#1.6.

[23] Z. Gao, C. Bird, E. T. Barr, To Type or Not to Type: Quantifying Detectable Bugs in JavaScript, 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 758-769.