1. Kommentit, muuttujat, tyyppimuunnokset

1.1. Yleistä

JavaScript on korkean tason ohjelmointikieli, jota kaikki modernit www-selaimet tukevat. Javascript on yksi World Wide Webin (myöh. webin) ydinteknologioista yhdessä HTML:n ja CSS:n kanssa. Tällä opintojaksolla käydään läpi javascript-ohjelmoinnin peruskonseptit, kuten muuttujat, aritmeettiset operaatiot, silmukat ja oliot.

JavaScript on alun perin Netscapen kehittämä pääasiassa Web-ympäristössä käytettävä dynaaminen komentosarjakieli. JavaScriptin tärkein sovellus on mahdollisuus lisätä Web-sivuille dynaamista toiminnallisuutta. Sitä käytetään tavallisimmin osana verkkoselaimia, joiden toteutukset sallivat asiakaspuolen skriptien interaktion käyttäjän kanssa, selaimen rajoitetun hallinnan, asynkronisen kommunikaation ja käyttäjälle näytettävän dokumenttisisällön muokkaamisen. JavaScriptiä käytetään myös palvelinten verkko-ohjelmoinnissa (esimerkiksi Node.js -ajoympäristössä), pelien kehityksessä ja työpöytä- sekä mobiilisovellusten luomisessa.

JavaScriptiä ei tule sekoittaa Java-kieleen.

1.2. Kommentointi

Kuten missä tahansa ohjelmointikielessä, myös JavaScriptissä voi koodia kommentoida siten, että niitä ei liitetä varsinaisen ajettavan ohjelman rakenteeseen, vaan kommentit ovat ohjeita ohjelmakoodia lukeville henkilöille, jotta he kykenevät ymmärtämään ohjelman rakenteen. Kommentit ovat siis koodin tekijän merkintöjä, jotka selkeyttävät tai selittävät koodia, sekä tekijälle itselleen, että mahdollisille ulkopuolisille koodin päivittäjille. Kommentoinnin päätarkoitus on säästää työtä, mutta myös jotkut ohjelmat voivat käyttää kommentteja hyväkseen. JS-tulkki kuitenkin ohittaa kaikki kommentit käsittelemättä niitä ollenkaan.

Kommentteja voi jättää kahdella tavalla:

```
// tämä on kommentti, joka on yhdellä rivillä

/* tämä on kommentti,
    joka
    on
    neljällä rivillä */
```

1.3. Muuttujien määrittely (1/4)

Määritelmän mukaan *data* on kaikkea sitä, mikä on merkityksellistä tietokoneelle. JavaScriptissä on seitsemän (7) tietotyyppiä erilaiselle datalle. Nämä tietotyypit ovat: `undefined`, `null`, `boolean`, `string`, `symbol`, `number` ja `object`.

Kuten ehkä olet aikaisemmin oppinut, tietokone erottelee esimerkiksi *numerot* (esim. 90) ja *merkkijonot* (esim. "90"), jotka ovat merkkien kokoelmia. Tietokone pystyy laskemaan numeroilla, mutta ei merkkijonoilla.

Muuttujat ovat niitä, mihin tietokone tallentaa ja manipuloi dataa dynaamisesti. Tämä tapahtuu käytännössä siten, että tietokone kutsuu muuttujien nimiä sen sijaan, että osoittaisi itse dataa. Mitä tahansa seitsemästä tietotyypistä voidaan tallentaa muuttujiin.

Muuttujilla (variable) on sama analogia matematiikasta tuttuihin muuttujiin, esimerkiksi `x` ja `y`. Toisin sanoen, muuttujat ovat vain viitteitä dataan, johon haluamme viitata. Ero matematiikan muuttujiin on kuitenkin se, että tietokoneella muuttujien sisältämät *arvot* voivat muuttua toisin kuin matematiikassa, jossa ne pysyvät jatkuvasti samana.

Seuraavassa määritellään muuttuja `omaNimi`:

```
var omaNimi;
```

Sana `var` tarkoittaa, että kyseessä on muuttuja ja sana `omaNimi` on muuttujan nimi. Huomaathan, että muuttujien nimillä on tiettyjä rajoituksia. Esimerkiksi nimen pitää olla yhtenäinen merkkijono. Näin ollen esimerkiksi `var oma Nimi` ei ole oikea muuttuja, koska `oma` ja `Nimi` ovat kirjoitettuna erikseen.

Nimeämisessä tulee huomioida että JavaScript on merkkiriippuvainen, eli isot- ja pienaakkoset ovat eri merkkejä.

Kaikki muuttujat, kahta poikkeusta lukuunottamatta, käyttäytyvät JavaScriptissä oliomaisesti. Nämä poikkeukset ovat `null` sekä `undefined`.

1.4. Muuttujien määrittely (2/4)

Muuttujiin tallennetaan arvot oikealta vasemmalle. Tämä tarkoittaa sitä, että kaikki operaattoria = oikealla puolella ratkaistaan ennen kuin arvo sijoitetaan ko. operaattorin vasemmalle puolelle. Esimerkki:

```
OmaMuuttuja = 5; // (1)
OmaNumero = OmaMuuttuja; // (2)
```

(1) kohdassa muuttujan `OmaMuuttuja` arvoksi tulee viisi, joka syötetään kohdassa (2) `OmaNumero`-muuttujan arvoksi. Myös seuraavantyyppiset määrittelyt ovat mahdollisia:

```
OmaMuuttuja = (100*6)/2 + OmaMuuttuja // (3)
```

(3) Kohdassa lasketaan ensin (100*6)/2, joka on 300 ja tämän jälkeen siihen liitetään `OmaMuuttuja`-muuttujan arvo, joka on tällä hetkellä kohdan (1) mukaisesti 5. Tästä seuraa, suorituksen jälkeen kohdassa (3) oleva `OmaMuuttuja`-muuttujan arvo on 305.

Kun useita operaattoreita sovelletaan yhtä aikaa, niillä on erilainen suoritusjärjestys, *presedenssi.* Tämä on matematiikasta tuttua, ja jakolaskulla (/) ja kertolaskulla (*) on korkeampi presedenssi kuin plus (+) tai (-) -laskulla. Jos kyseessä ovat samanarvoistet operaattorit, suoritusjärjestys määräytyy vasemmalta oikealle. Esimerkiksi 1+2-4 suoritetaan siten, että ensin lasketaan 1+2 ja tämän jälkeen vähennetään 4 eli tuloksena on -1. Vastaavasti tilanteessa 1*2+3*4/4 ensin suoritetaan 1*2, tämän jälkeen 3*4, seuraavaksi /4. Eli tuloksena on 2+3 eli 5.

Edellä kuvattujen operaattorien lisäksi on olemassa jakojäännös-operaattori (%) eli *modulo.* Modulo kertoo siis jakojäännöksen. Esimerkiksi 12 % 3 = 0. Vastaavasti 12 % 5 = 2.

JavaScriptissä on myös *erikoislukuja*, kuten ääretön (Infinity) ja ei-määritelty luku (NaN). Esimerkiksi ääretön - ääretön antaa tulokseksi NaN. Tai esimerkiksi Infinity/Infinity = NaN.

Useissa ohjelmointikielissä luvuille voidaan tehdä erilaisia *tilanvarauksia*. Esimerkiksi C-kielessä luvuille on olemassa 8-bittinen ja 16-bittinen muuttujamäärittely. Sen sijaan JavaScriptissä numerotietoa tallentavat muuttujat ovat aina 64 bittisiä. Tämä merkitsee sitä, että jos tallennetaan esimerkiksi luku 240, se vaatii oikeasti vain kahdeksan bitin tilanvarauksen, JavaScriptissa kuitenkin varataan aina 64 bitin verran.

JavaScript-muuttujien tietotyyppiä ei erikseen määritellä. Muuttujien arvojen tietotyyppiäkin voidaan vaihtaa dynaamisesti, ohjelman suorituksen aikana.

1.5. Muuttujien määrittely - merkkijonot (3/4)

Merkkijonot *(string)* on JavaScriptin tapa tallentaa merkkijonoja. Seuraavat merkinnät tallentavat merkkijonon:

```
'tämä on merkkijono'
"tämäkin on merkkijono"
´tässäkin on vielä merkkijono´
```

Toisin kuin monissa muissa ohjelmointikielissä, JavaScript hyväksyy merkkijonon tallentamiseen erilaisia lainausmerkkejä, kunhan vain ne ovat molemminpuolin merkkijonoa samanlaiset.

Merkkijonot voivat sisältää myös merkkausta, jolloin merkkauskielen sisältämät lainausmerkit voivat aiheuttaa harmeja. Yhdenmukaisuus- ja yhteensopivuussyistä kannattaa käyttää heittomerkkejä JavaScript-merkkijonojen rajoina, koska useimmiten merkkauskielessä käytetään lainausmerkkejä. Esimerkiksi seuraavassa koodissa merkkijonon sisältämät lainausmerkit eivät sekoitu merkkijonon rajoina toimiviin heittomerkkeihin:

```
'<h3 title="Esimerkki">Suositus</h3>'
```

Merkkijonoihin voidaan sisällyttää erikoismerkkejä (escape characters), kuten esimerksi rivinvaihdon. Erikoismerkit ilmaistaan "kenoviivalla" `\`, jota seuraa haluttu merkki. Esimerkiksi lainausmerkki merkkijonon sisään esitetään koodilla `\"` ja tabulointi koodilla `\t`. Rivinvaihto saadaan aikaan koodilla `\n`. Esimerkiksi merkkijono "tämä on merkkijono\njoka on kahdella rivillä", tulostuu näytölle seuraavasti:

```
tämä on merkkijono
joka on kahdella rivillä
```

kaksi `\`-merkkiä peräkkäin merkkijonon sisällä sulautuu yhdeksi. Jos merkkijonoon halutaan tulostuvan \n, voidaan se toteuttaa seuraavasti: "Uuden rivin merkki voidaan tulostaa seuraavasti: \"\\n\"." ,jolloin näytölle tulostuu:

```
Uuden rivin merkki voidaan tulostaa seuraavasti: "\n".
```

Merkkijonot tallennetaan Unicode-koodistoon, joka on 16 bittinen. Tämä merkitsee, että jokainen merkkijonon yksittäinen merkki, *elementti*, voi saada 2^16 arvoa.

Merkkijonoja ei voida kertoa, jakaa tai lisätä. Mutta niitä voidaan yhdistellä. Esimerkkinä seuraavat merkkijonot: "epä"+"järjestelmällis"+"tyttämättö"+"myydellän"+"säkään" voidaan oheisesti yhdistää + -operaattorilla sanaksi:

```
epäjärjestelmällistyttämättömyydellänsäkään
```

```
var first_name = "Matti";
var last_name = "Meikäläinen";
console.log("Hei" + first_name + " " + last_name);
```

JavaScript sisältää myös valmiita metodeja merkkijonojen käsittelyyn. Seuraavissa taulukoissa muutamia käyttökelpoisia metodeja, jotka helpottavat ohjelmointia.

Metodi	**Kuvaus**
length	merkkijonon pituus (ominaisuus)
charAt(paikka)	palauttaa merkin paikasta (ensimmäinen merkki on 0:s)
charCodeAt(paikka)	palauttaa merkin Unikoodin kohdasta paikka
substring(aloitusnro,lopetusnro)	hakee merkkijonon väliltä aloitusnro-lopetusnro (lopetusnro ei mukana)
toLowerCase()	muuttaa merkkijonon pieniksi kirjaimiksi
toUpperCase()	muuttaa merkkijonon isoiksi kirjaimiksi
split(jakajamerkki)	muuttaa merkkijonon taulukon alkioiksi annetun jakajamerkin kohdalta
indexOf(merkkijono)	palauttaa merkkijonon paikan (palauttaa -1, jos ei löydy)
lastIndexOf(merkkijono)	kuten yllä, mutta merkkijono käydään oikealta vasemmalle
substr(aloitusnro,pituus)	hakee halutun määrän merkkejä alkaen aloitusnro:sta
replace(mjono1,mjono2)	korvaa merkit mjono1 merkeillä mjono2 (vain ensimmäisen minkä löytää)
fontcolor()	palauttaa fontin värin
fontsize()	palauttaa fontin koon
search(merkkijono)	palauttaa annetun merkkijonon sijainnin toisesta merkkijonosta. Palauttaa -1:n, jos merkkijonoa ei löytynyt.

1.6. Muuttujien määrittely: unaari, binaari ja boolean operaattorit (4/4)

`console.log`-funktio tulostaa halutun tekstin ruutuun, tarkemmin konsoliin, joka on yleensä esimerkiksi komentotulkin ruutu tai selainikkuna. Esimerkiksi `console.log("Hello World!\n")` tulostaa ruutuun `Hello World!` ja tekee rivinvaihdon.

Operaattori (operator) on symbolinen merkki tai merkkiyhdistelmä lausekkeessa, joka suorittaa tietyn toimenpiteen.

Operaattorien avulla suoritetaan laskutoimituksia, verrataan lukuja toisiinsa ja tehdään sijoituksia muuttujiin. Yleisin operaattori on sijoitusoperaattori eli yhtäsuuruus-merkki (=). Sen avulla sijoitetaan muuttujalle uusi arvo. Sitä ei kuitenkaan missään nimessä tule sekoittaa vertailuoperaattoriin ==, jonka avulla tarkistetaan ovatko arvot yhtä suuria.

Sijoitus

```
var a = 2;
var b = 2;

// vertailu; onko a yhtäsuuri kuin b
if (a == b)
```

Vertailuoperaattorit ovat:

```
== Löysä yhtäläisyysoperaattori. Tarvittaessa yritetään tyyppimuunnosta.
!= Löysä erilaisuusoperaattori. Tarvittaessa yritetään tyyppimuunnosta.
=== Tiukka yhtäläisyysoperaattori. Ei tyyppimuunnosta.
!== Tiukka erilaisuusoperaattori. Ei tyyppimuunnosta.
> Suurempi kuin -operaattori. Tarvittaessa yritetään tyyppimuunnosta.
< Pienempi kuin -operaattori. Tarvittaessa yritetään tyyppimuunnosta.
<= Pienempi ja yhtäsuuri kuin -operaattori.  Tarvittaessa yritetään tyyppimuunnosta.
>= Suurempi tai yhtäsuuri kuin -operaattori. Tarvittaessa yritetään tyyppimuunnosta.
```

Aritmeettiset operaattorit ovat laskutoimituksia suorittavia operaattoreita, jotka palauttavat yksittäisen numeerisen arvon. Nämä operaattorit jaetaan:

- binäärisiin operaattoreihin - laskutoimitus suoritetaan kahden operandin välillä, jolloin operandeina toimivat luvut ja/tai lukuarvon sisältävät muuttujat
- unaarisiin operaattoreihin - laskutoimitus suoritetaan yhdelle operandille, jolloin operandina toimii lukuarvon sisältävä muuttuja

Unaarioperaattori on operaattori, joka kohdistuu vain yhteen operandiin. Unaarioperaattoreita voidaan testata `console.log()`-funktiolla. Esimerkiksi `typeof` on unaarioperaattori. `console.log(typeof 8)` tulostaa ruutuun, että merkki 8 on tässä tapauksessa luku (number). Vastaavasti `console.log(typeof "8")` tulostaa, että kyseessä on merkkijono (string).

Arvonmuunto-operaattoreiden avulla voidaan muuntaa arvoa yhdellä yksiköllä. Lukuun on mahdollista lisätä / vähentää yksi, lisätä/vähentää toinen luku tai tehdä luvusta negaatio:

```
++    lisää yksi
--    vähennä yksi
-     negaatio
+=    lisää luku
-=    vähennä luku
*=    kerro luku
/=    jaa luku
```

Esimerkiksi:

```
var b = 5;
b++;   // 6

var a += b   // tämä on sama kuin a = a + b
```

Sen sijaan - ja + ovat *binaarioperaattoreita.* Ne ottavat arvon molemmille puolille. Toisaalta, niitä voidaan käyttää sekä unaari että binaarioperaattoreina. Esimerkiksi `console.log(-(10-11)` antaa tulokseksi 1.

Merkkijonojen yhteydessä + -operaattori toimii liitosoperaattorina yhdistäen merkkijonoja toisiinsa.

Totuusarvot, Boolean-arvot ovat sellaisia, jotka antavat tulokseksi vain `true` (tosi) tai `false` (epätosi).

Esimerkiksi `console.log(10>9)` antaa tulokseksi `true`. Vastaavasti `console.log(9>10)` antaa tulokseksi `false`.

Vastaavalla tavalla voidaan verrata merkkijonoja. Esimerkiksi `console.log("Saippuakauppias" > "Saippuakauppias")` antaa tulokseksi `false`.

Edelleen `console.log("Saippuakauppias" != "Saippuakauppias")` antaa tulokseksi `false`.

Loogisia operaattoreita ovat:

```
!   EI-operaatio. Palauttaa käänteisen totuusarvon.
&&  JA-operaatio. Palauttaa true-arvon, mikäli kaikki
    operandit evaluoituvat tosiksi, muuten palauttaa false-arvon.
||  TAI-operaatio. Palauttaa true-arvon, mikäli vähintään
    yksi operandi evaluoituu todeksi, muuten palauttaa false-arvon.
```

Kun yhdistetään aritmeettisia ja boolean operaattoreita, ei ole aina selvää, milloin sulkeita tarvitaan. Yleensä selvitään sillä tiedolla, että `||`-operaattorilla on alhaisin presedenssi, sitten `&&` ja sitten vertailuoperaattorit (`>`, `==` jne.). Tämä järjestys on valittu sen vuoksi, että selvittäisi mahdollisimman pienellä sulkeiden määrällä.

Esimerkiksi seuraava palauttaa arvon `true`: `1*2==2 && 3*4 != 13 > 11`

JavaScriptissa on vielä *tertiaarinen* (ternary) operaattori, joka toimii seuraavasti. Esitellään muuttuja `henkilo`, ja testataan, onko hän ajokorttikelpoinen:

```
var henkilo= {
  nimi: 'jussi',
  ika: 20,
  driver: null
};henkilo.driver = henkilo.ika >= 18 ? 'Yes' : 'No';
```

Viimeisen lauseen toiminta voidaan esittää seuraavasti:

```
henkilo.driver = ((henkilo.ika >= 18) ? 'Yes' : 'No';)
```

Siinä siis testataan, onko ikä vähintään 18 ja palautetaan arvo 'Yes', jos näin on. Muuten palautetaan 'No'. Esimerkin tapauksessa ikä on 20 > 18 eli tosi ja toiminta on:

```
henkilo.driver = 'Yes';
```

1.7. Automaattinen tyyppimuunnos

Kun tietotyypit ovat epäselviä, JavaScript pyrkii monimutkaisten sääntöjen avulla saamaan tyypit yhteismitallisiksi. Katsotaan seuraavia esimerkkejä:

```
console.log("5" + 5) // (1)
console.log("5" - 1) // (2)
console.log(10 * null) // (3)
console.log("eight" * 8) // (4)
console.log(true == 0) // (5)
```

Kohdassa (1) järjestelmä saa arvoksi 55. Kohdassa (2) puolestaa arvoksi 4. Kohdassa (3) tulos on 0. Kohdassa (4) puolestaan `NaN`. Viimeisen eli kohdan (5) tulostus on `true`. Huomaa: `console.log(NULL = undefined)` tulokseksi tulee `true`, mutta `console.log(null == 0)` tulos on `false`.

Operaattorit `&&` ja `||` käsittelevät eri tyyppisiä arvoja eri tavalla, hieman omituisesti. Esimerkiksi `console.log(null || "Pekka")` palauttaa arvon `Pekka`.

Sääntö: `||` -operaattori palauttaa vasemmanpuolimmaisen arvon, jos se pystyy tekemään tyyppimuunnoksen. Muussa tapauksessa oikeanpuolimmaisen arvon. Tässä tapauksessa tyyppimuunnosta

ei voi tehdä, koska `null`-tyyppiin ei voi muuntaa merkkijonoa. Siksi palautetaan siis arvo `Pekka`.

`&&` toimii samantyylisesti kuin `||`, mutta täsmälleen päinvastoin. Eli `console.log(null && "Pekka")` palauttaisi arvon `null`.

1.8. Matemaattiset operaatiot

`Math`-olion avulla ratkaistaan "kehittyneitä" matemaattisia ongelmia. Sen sijaan normaalit laskutoimitukset, kuten yhteen- ja vähennyslasku voidaan suorittaa joko perusoperaattoreilla tai `eval()`-funktion avulla.

`eval`-funktio muuttaa parametrina saamansa merkkijonotyyppisen lausekkeen matemaattiseen, suoritettavaan muotoon. Seuraava esimerkki havainnollistaa `eval`-funktion toimintaa.

```
<script type="text/javascript">
// Sijoitetaan muuttujaan laskutoimitus merkkijonona:
var laskutoimitus = "2 + 4 - 7 / 16";

// Ilman eval()-funktiota tulostus on "2 + 4 - 7 / 16":
console.log(laskutoimitus);

/* eval-funktio muuntaa merkkijonotyyppisen
laskulausekkeen matemaattiseen muotoon ja suorittaa
laskutoimitukset. Selaimeen tulostuu "5.5625": */

console.log(eval(laskutoimitus));
</script>
```

JavaScript käsittelee matemaattisia operaatioita `Math`-olion kautta. `Math`-olio sisältää muutaman vakion yleisistä matematiikassa käytettävistä arvoista:

VAKIO	KUVAUS
Math.E	Neperin luku e eli noin 2,72
Math.LN10	luvun 10 luonnollinen logaritmi
Math.LN2	luvun 2 luonnollinen logaritmi
Math.PI	piin arvo, noin 3,14
Math.SQRT1_2	neliöjuuri luvusta 1/2
Math.SQRT2	neliöjuuri luvusta 2

`Math`-olion `round`-metodilla voidaan parametrina annettava luku pyöristää normaalien pyöristyssääntöjen mukaisesti.

`floor`- ja `ceil`- metodit pyöristävät myös parametrina annettavan luvun, `floor` alaspäin ja `ceil` ylöspäin seuraavaan kokonaislukuun.

Metodi `max` palauttaa parametreina annetuista luvuista suuremman, kun taas `min` palauttaa pienemmän luvun.

`random` palauttaa satunnaisluvun väliltä 0 - 1. Luku on desimaaliluku, esimerkiksi 0.7378634... Jotta tästä saisi käyttökelpoisemman luvun, voidaan se kertoa vaikkapa kolmellakymmennellä, jolloin saadaan desimaaliluku väliltä 0 - 30. Kun tulos vielä pyöristetään `round`-metodilla, saadaan desimaaliluku muunnettua kokonaisluvuksi.

Muita olion metodeja ovat:

METODI	KUVAUS
Math.abs(param)	palauttaa parametrina annetun luvun itseisarvon
Math.acos(param)	palauttaa parametrina annetun luvun arkuskosinin
Math.asin(param)	palauttaa parametrina annetun luvun arkussinin

METODI	KUVAUS
Math.atan(param)	palauttaa parametrina annetun luvun arkustangentin
Math.cos(param)	palauttaa parametrina annetun luvun kosinin
Math.exp(param)	palauttaa e potenssiin parametrina annettu luku
Math.log(param)	palauttaa parametrina annetun luvun luonnollisen logaritmin
Math.pow(param1, param2)	palauttaa luvun param1 korotettuna potenssiin param2 (vaihtoehtoisesti operaattori **, esim. param1 ** param2)
Math.sin(param)	palauttaa parametrina annetun luvun sinin
Math.sqrt(param)	palauttaa parametrina annetun luvun neliöjuuren
Math.tan(param)	palauttaa parametrina annetun luvun tangentin

2. Ohjelmarakenteet

2.1. Lauseet, sidokset, paluuarvot

Koodilla tarkoitetaan ohjelmoinnissa tietokoneohjelman tekstimuotoista ohjelmointikielistä listausta.

Ilmaisu, lauseke (*expression*) on koodin osa, joka tuottaa arvon.

Lause (*statement*) on yleensä omalle rivilleen kirjoitettava yksittäinen toiminto ohjelmassa.

Javascriptin *ilmaisu* tarkoittaa siis lauseen osaa ja *lause* täyttä lausetta. Esimerkiksi:

```
!TRUE; // (1)
2; (2)
```

(1) ja (2) ovat toimivia Javascriptin lauseita: niissä on ilmaisu sekä ne päättyvät puolipisteeseen.

Lauseita voidaan ryhmitellä lohkoiksi (blocks). *Lauselohko* on yhden tai useamman lauseen sisältävä eroteltu kokonaisuus, jota käytetään muun muassa silmukoissa ja funktioissa. Lauselohkot merkitään aaltosulkumerkeillä:

```
{ //lohko alkaa
  //lause tai lauseet lohkon sisällä;
} //lohko päättyy
```

Sidokset (bindings) tai muuttujat (variables) ovat komponentteja, joiden avulla Javacript ylläpitää ohjelman sisäistä tilaa. Tarkastellaan seuraavaa esimerkkiä:

```
var Lampo = 10 * 10; // (3)
console.log(Lampo * Lampo); // (4)
```

(3) on toimiva Javascriptin lause, joka muodostaa sidoksen seuraavasti: 10 * 10 = 100, ja tämä tulos (100) "sidotaan" muuttujaan Lampo. Tämän jälkeen tätä sidosta käytetään ja (4) tulostaan tulos eli tässä tapauksessa 100 * 100 eli 10 000.

Kun muuttuja määritellään *var*-sanalla, sen näkyvyysalueeksi tulee se funktio, jonka sisällä määrittely sijaitsee. Kyseessä on funktion paikallinen muuttuja. `var`-muuttuja voidaan määritellä useampaankin kertaan eri puolilla funktiota, mutta kyseessä on tällöin kuitenkin aina sama muuttuja.

JavaScriptin versioon 1.7 lisättiin mahdollisuus määritellä myös lohkoihin paikallisia muuttujia määrittelemällä ne varatulla sanalla *let*.

On luonnollista, että muuttujien sidoksia haluataan vaihtaa kesken. Katso seuraavaa esimerkkiä (5-8):

```
var Olotila = "iloinen"; // (5)
console.log(Olotila); // (6)
Olotila = "surullinen"; // (7)
console.log(Olotila); // (8)
```

Rivillä (5) määritellään muuttuja Olotila ja siihen sidotaan arvo "iloinen". Tämä tulostetaan rivillä (6) ja saadaan tuloste "iloinen". Tämän jälkeen Olotila-muuttuja alustetaan uudelleen rivillä (7) siten, että saadaan sidoksen eli muuttujan arvoksi surullinen. Tämän jälkeen tulostetaan tämä uudelleenalustettu muuttuja puolestaan rivillä (8).

Muuttujia voidaan määritellä myös useita peräkkäin, kunhan ne erotellaan toisistaan pilkuilla (9):

```
var muuttuja1 = 1, muuttuja2 = 2, muuttuja3 = 3; // (9)
console.log(muuttuja1+muuttuja2+muuttuja3); // (10)
```

Rivillä (10) tulostuu 6.

Lisäksi voidaan määritellä muuttuja siten, että se on koko elinaikansa samassa vakioarvossa. Esimerkiksi const laji = "lintu"; määrittelisi muuttujan laji arvoksi koko ohjelman suorituksen ajaksi lintu.

Joitain varattuja sanoja ei voi käyttää muuttujien nimissä. Näitä ovat esimerkiksi: *`break case catch class const continue debugger default delete do else enum export extends false finally for function if implements import interface in instanceof let new package private protected public return static super switch this throw true try typeof var void while with yield`.*

Muuttujien nimet kannattaa valita kuvaaviksi. Nimissä ei saa olla välilyöntejä, mutta niissä voi ja kannattaa käyttää isoja kirjaimia ja alaviivaa seuraavan esimerkin mukaisesti (ensimmäinen nimi kannattaa korvata joillain seuraavista):

```
karvainenpienilelu
karvainen_pieni_lelu
karvainenPieniLelu
```

Funktiot ovat aliohjelmia, joiden avulla voidaan pakata toiminnallisuutta myöhemmin ja/tai uudelleen käytettäväksi. Samoin kuin muuttujat, funktiotkin on esiteltävä ennen kuin niitä voidaan käyttää. Funktioilmoituksilla on myös samat nimeämiskäytännöt ja näkyvyysalueet kuin muuttujilla.

Funktioilmoituksen syntaksi on seuraavanlainen:

```
function funktioNimi(/*parametriluettelo*/)
 {// funktion runko alkaa
  /* Suoritettava(t) lause(et) */
 }// funktion runko päättyy
```

Javascriptissä on jo edellisessä luvussa esitetty console.log-funktio, joka tulostaa johonkin järjestelmän määrittelemään oletustulostusvirtaan. Sidosten nimissä ei voi olla pisteitä, mutta console.log-funktiossa voi olla. Tämä johtuu siitä, että ilmaisu (expression) hakee log-*ominaisuuden* (property), joka on console-sidoksessa. Tämä on voi olla vielä tässä vaiheessa aika vaikeasti ymmärrettävä, mutta asia selviää jäljempänä luvussa 4. Rivillä (11) on esimerkki *paluuarvosta* (return value):

```
console.log(Math.max(2, 4, 5) + 100); // (11)
```

Järjestelmä tulostaa arvon 105. `Math.max`-*funktio* valitsee syötearvioista 2, 4, 5 suurimman ja palauttaa sen funktion *paluuarvona*. Tämän jälkeen siihen lisätään vielä arvo 100. Lopputuloksena siis arvo 105.

Funktioista lisää luvussa 4.

Käskyjen suoritus

Ohjelman käskyt suoritetaan peräkkäin. Seuraavassa esimerkissä on kaksi lausetta, joista ensimmäisessä luetaan arvo, ja toisessa lasketaan ja tulostetaan syötetyn arvon neliö.

```
var Luku = Number(prompt("Syötä luku")); //(12)
console.log("Luvun neliö on " + Luku * Luku);
```

Rivin (12) funktio `Number` muuntaa syötetyn (merkkimuotoisen) luvun numeeriseksi.

`if`-lauseet (jos-lauseet)

Jos ohjelman suorituksen halutaan haarautuvan jonkin ehdon perusteella (ehdollinen haarautuminen), käytetään `if`-lausetta (jos-lause):

```
if (ehto) {// lauselohko alkaa
  /* Suoritettava(t) lause(et)
     mikäli ehto evaluoituu todeksi.
     Muuten ohjelman suoritus jatkuu lauselohkon jälkeen. */
}// lauselohko päättyy
```

Tarkastellaan seuraavaa esimerkkiä

```
var Luku= Number(prompt("Syötä luku"));
if (!Number.isNaN(Luku))  //(13)
{
console.log("Luvun neliö on " + Luku * Luku);
}
```

Rivillä (13) testataan syötteen muoto käyttäen Javascript-funktiota `isNaN`. Se palauttaa arvon `true` (tosi), jos sen argumentti ei esitä numeerista arvoa, eli on tyyppiä `NaN`. Tässä tapauksessa ehtolause siis suoritetaan, jos syöte esittää ei-numeerista arvoa.

Lauselohko sijoitetaan {- ja } -merkkien väliin. Ohjelman selkeyden vuoksi on suositeltava käyttää tätä notaatiota, vaikka lauselohko olisi yhden käskyrivin mittainen.

Tarvittaessa `if`-lauseita voidaan asettaa peräkkäin, niin monta kuin on tarpeen.

`if`-lauseita voidaan myös asettaa sisäkkäin, mikä mahdollistaa useamman eri ehdon tarkistamisen seuraavasti

```
   if (ehto2) {//siirrytään seuraavaan mikäli ehto täyttyy
    if (ehto3) {
      /* suoritettava(t) lause(et) vain kaikkien edellisten ehtojen täyttyessä */
    }
  }
}
```

Johdonmukaiset sisennykset parantavat syvälle tasolle menevien sisäkkäisyyksien luettavuutta ja tulkintaa.

`else`-lause (muuten-lause), on ehtolause, joka voidaan asettaa (aina tarvittaessa) `if`-lauseelle kuuluvan lauselohkon jälkeen. else-lausetta seuraavan lauselohkon lause(et) suoritetaan, mikäli välittömästi edellinen `if`-lause evaluoituu epätodeksi:

```
   /* Suoritettava(t) lause(et)
     mikäli ehto evaluoituu todeksi. */

} else {//muussa tapauksessa
  /* Suoritettava(t) lause(et)
     mikäli ehto evaluoituu epätodeksi. */
}
```

Seuraava esimerkki kuvaa `else`-lauseen käyttöä:

```
var Luku = Number(prompt("Syötä luku"));
if (!Number.isNaN(Luku)) {
console.log("Luvun neliö on " + Luku * Luku);
} else {
```

```
console.log("Virheellinen luku");
}
```

`if/else`-pareja voidaan ketjuttaa seuraavasti:

```
if (ehto1) {//jos ehto on tosi
/* Suoritettava(t) lause(et)
   mikäli ehto evaluoituu todeksi. */
} else if (ehto2) {//muuten jos ehto on tosi
  /* Suoritettava(t) lause(et)
     mikäli ehto evaluoituu todeksi. */
  } else if (ehto3) {//muuten jos ehto on tosi
    /* Suoritettava(t) lause(et)
       mikäli ehto evaluoituu todeksi. */
    } else {//valinnainnen lause
         /* Suoritettava(t) lause(et)
            mikäli mikään ehto ei toteudu. */
  }
```

Seuraava esimerkki valaisee `if/else`-ketjutusta:

```
var luku = Number(prompt("Syötä luku"));
if (luku < 10) {
  console.log("Pieni");
} else if (luku < 100) {
  console.log("Keskisuuri");
} else {
  console.log("Suuri");
}
```

Esimerkin ohjelma lukee luvun ja tulostaa tekstin "Pieni", jos se on alle 10. Jos luku on suurempi kuin 10, mutta pienempi kuin 100, tulostetaan "Keskisuuri". Muussa tapauksessa tulostetaan "Suuri".

2.2. Lauseet, sidokset, paluuarvot

Toistolauseet

Toistolauseet, tai *silmukat (loop)*, ovat ohjelmarakenteita, joilla toistetaan toimenpiteitä niin kauan kuin tietty toistoehto on voimassa.

`while`-toistolausetta suoritetaan niin kauan kuin ehto on tosi. Lause koostuu `while`-käskystä ja sitä seuraavasta ehtolauseesta.

`while`-lauseen syntaksi on seuraavanlainen:

```
/* Seuraavassa toistoehto on mikä tahansa arvioitava
   lauseke kaarisulkujen välissä: */

while (toistoehto) {// silmukan runko alkaa
  /* Suoritettava(t) lause(et)
     mikäli toistoehto evaluoituu todeksi.
     Muuten ohjelman suoritus jatkuu lauselohkon jälkeen. */
}// silmukan runko päättyy
```

Seuraava ohjelma tulostaa luvut 1, 3, 5, ... , 11 käyttäen `while`-lausestta:

```
var luku = 1; //(14)
while (luku <= 12) { //(15)
  console.log(luku); //(16)
  luku = luku + 2; //(17)
}
```

Rivillä (14) alustetaan kierrosmuuttuja. Rivillä (15) annetaan suoritusehto. Silmukan lauseet (16) ja (17) suoritetaan, kun ehto on tosi. Lausessa (17) inkrementoidaan kierrismuuttujaa. Lauseen (17) suorituksen jälkeen `while`-ehto testataan uudelleen. Jos se on edelleen tosi, lauseet (16) ja (17) suoritetaan uudelleen. Silmukasta poistutaan, kun ehto tulee epätodeksi.

Edellä kierrosmuuttujan alkuarvoksi annettiin 0, mikä on suositeltava käytäntö.

Seuraava ohjelma laskee arvon 2^{10}. Siinä käytetään kahta kierrosmuuttujaa. Toinen (`laskuri`) säätelee kierrosten lukumäärän ja toiseen (`tulos`) lasketaan tulos.

```
var tulos = 1;
var laskuri = 0;
while (laskuri < 10) {
  tulos = tulos * 2;
  laskuri = laskuri + 1;
}
console.log(tulos);
```

`do-while`-lause on `while`-lauseen muunnelma, jossa lauselohkon lauseet suoritetaan ainakin kerran, minkä jälkeen toistoehto evaluoidaan (loppuehtoinen silmukka).

`do-while`-lauseen syntaksi on:

```
do {
/* Vähintään kerran suoritettava(t) lause(et).
   Toistetaan niin kauan kuin toistoehto evaluoituu todeksi.
   Muista vaikuttaa toistoehtoon tämän lohkon sisältä! */
} while (toistoehto);
```

Seuravassa esimerkissä do-while-lausetta käytetään lukemaan nimi niin, että pyyntöön on syötettävä jotakin:

```
var Nimesi;
do {
  Nimesi = prompt("Syötä nimesi"); //(18)
} while (!Nimesi); //(19)
console.log(Nimesi);
```

Lauseessa (19) testataan, onko syötetty tyhjä rivi. Jos on, silmukan lause (18) suoritetaan uudelleen. Muussa tapauksessa poistutaan lohkosta.

Edellä olevissa esimerkeissä koodirivejä on sisennetty ohjelman selkeyden parantamiseksi. Tavallisesti uuden sisemmän lohkon käskyt sisennetään kahdella välilyönnillä seuraavan esimerkin mukaisesti:

```
if (false != true) {
  console.log("Pitää paikkansa");
  if (1 < 2) {
    console.log("Ei epäilystäkään");
  }
}
```

`for`-lauseessa alustetaan kierroslaskuri, minkä jälkeen evaluoidaan valinnainen toistolauseke. Jos ehto on tosi, suoritetaan silmukan rungon lauseet. Silmukasta poistutaan, kun ehto on epätosi.

`for`-lauseen syntaksi on:

```
for (/*alkuasetus*/;
     /*toistoehto*/;
```

```
       /*päivitys*/) {//silmukan runko alkaa

  /* Suoritettava(t) lause(et)
     mikäli toistoehto evaluoituu todeksi.
     Muuten ohjelman suoritus jatkuu lauselohkon jälkeen. */

}//silmukan runko päättyy
```

Seuraava ohjelma laskee arvon 2^{10} käyttäen `for`-silmukkaa:

```
var tulos = 1;
for (var laskuri = 0; laskuri < 10; laskuri = laskuri + 1) //(20)
{
  tulos = tulos * 2;
}
console.log(tulos);
```

Lauseessa (20) annetaan kierrosmuuttujalle alkuarvo, asetetaan kierrosmuuttujan yläraja ja määritellään inkrementointi. Silmukan lauseet suoritetaan niin monka kierrosta, kun kiierrosmuuttujan arvo on pienempi kuin 10.

`break`-lauseella voidaan katkaista kaikki toistolauseet silmukoiden rungon sisältä, vaikka toistoehto olisikin vielä voimassa.

Seuraava esimerkki valaisee `break`-lauseen toimintaa. Ohjelmassa etsitään ensimmäinen luku, joka on suurempi kuin 20 ja jaollinen 7:llä.

```
for (var luku = 20; ; luku = luku + 1) { //(21)
  if (luku % 7 == 0) { //(22)
    console.log(luku);
    break;
  }
}
```

Lauseessa (21) määritellään silmukka, joka käy 20:sta alkaen, ja jossa jokaisella kierroksella inkrementoidaan kierroslaskuria yhdellä. Lopetusehto on jätetty tyhjäksi. Lauseessa (22) testataan modulo-operaattorilla, onko luvun sen hetkinen arvo jaollinen 7:llä. Jos näin on, silmukasta poistutaan `break`-komennolla.

Jos `break`-lause poistetaan, silmukka muuttuu äärettömäksi, ei pääty milloinkaan. Tämä on tavallisesti haitallinen vikatilanne.

Javascriptissa voidaan inkrementointilause:

```
laskuri = laskuri + 1;
```

korvata lyhennettynä:

```
laskuri += 1;
```

Rakenne on yleinen ja toimii kaikilla lukuarvoilla.

Aiemman esimerkin `for`-lause voidaan kirjoittaa muotoon:

```
for (var laskuri = 0; laskuri < 12; laskuri += 1)
```

Vastaavasti

```
laskuri = laskuri - 2;
```

voidaan korvata lyhennettynä:

```
laskuri -= 2;
```

ja

```
tulos = tulos * 2;
```

voidaan korvata lyhennettynä:

```
tulos *= 2;
```

Yhdellä inkrementointi ja dekrementointi voidaan kirjoittaa vielä lyhyemmin:

```
laskuri ++
laskuri --
```

`switch`-lause eli valintalause sopii tilanteisiin, joissa pitää tarkastaa monta toisensa poissulkevaa vaihtoehtoa. Sen toiminta on seuraavanlainen:

1. Evaluoidaan switch-lauseen argumenttina annettu lauseke.
2. Verrataan lausekkeen arvoa jokaiseen (case-avainsanalla esiteltyyn) valintavakioon järjestyksessä kunnes lausekkeen arvoa vastaava valintavakio löytyy.
3. Suoritetaan kaikki "natsannutta" valintavakiota seuraavat switch-lohkon lauseet kunnes kohdataan katkaisulause (break) tai lauselohko päättyy.
4. Lohkoa voidaan täydentää valinnaisella default-lauseella, jonka lause(et) suoritetaan mikäli mikään valintavakio ei "tärppää".

`switch`-lauseen syntaksi on:

```
switch (lauseke) {//usein argumenttina muuttuja
  case valintavakio1: lause(et); break;
  case valintavakio2: lause(et); break;
  case valintavakio3: lause(et); break;
  case valintavakio4: lause(et); break;
  case valintavakio5: lause(et); break;
             default: lause(et);
}
```

`break`-lause on teoriassa valinnainen, mutta käytännössä pakollinen, koska muuten suoritetaan seuraavienkin valintavakioiden ja/tai `default`-lause(et) riippumatta niiden soveltuvuudesta.

Seuraava esimerkki valaisee `switch`-lauseen käyttöä:

```
switch (prompt("Millainen sää on tänään?")) { //(23)
  case "sateinen":
    console.log("Muista sateenvarjo.");
    break;
  case "aurinkoinen":
    console.log("Pukeudu kevyesti.");
  case "pilvinen":
    console.log("Mene ulkoilemaan.");
    break;
  default:
    console.log("Tuntematon säätyyppi!");
    break;
}
```

Rivillä (23) luetaan säätyyppi. Jos arvo on tunnettu, siihen sopiva ohje annetaan `case`-lauseissa. "aurinkoinen"-`case`-lauseen perässä ei ole `break`-lausetta, joten siihen kohdistunut valinta jatkuu seuraavaan lauseeseen.

Muuttujien nimet kannattaa valita kuvaaviksi. Nimissä ei saa olla välilyöntejä, mutta niissä voi ja kannattaa käyttää isoja kirjaimia ja alaviivaa seuraavan esimerkin mukaisesti (ensimmäinen nimi kannattaa korvata joillain seuraavista):

```
karvainenpienilelu
karvainen_pieni_lelu
karvainenPieniLelu
```

3. CSS, Web & HTML

3.1. HTML

HTML

HTML on kuvauskieli web-sivustojen luomiseen. Sen avulla kuvataan sekä web-sivun rakenne että sivun sisältämä teksti. HTML-sivujen rakenne määritellään HTML-kielessä määritellyillä elementeillä, ja yksittäinen HTML-dokumentti koostuu sisäkkäin ja peräkkäin olevista elementeistä.

Sivujen rakenteen määrittelevät elementit erotellaan pienempi kuin (<) ja suurempi kuin (>) -merkeillä. Elementti avataan elementin nimen sisältävällä <-merkillä alkavalla ja >-merkkiin loppuvalla merkkijonolla, esim. `<html>`, ja suljetaan merkkijonolla jossa elementin <-merkin jälkeen on kauttaviiva, esim `</html>`. Yksittäisen elementin sisälle voidaan laittaa muita elementtejä.

Suurin osa elementeistä tulee sulkea lopuksi. Osa HTML5:n elementeistä – esimerkiksi `<br>` – on kuitenkin ns. tyhjiä ("void"), eikä niille kirjoiteta erillistä lopetusta. Halutessaan tyhjät elementit voi lopettaa X(HT)ML-tyyliseen `/`-merkkiin, esimerkiksi seuraavasti: `<br />`

Seuraavassa on esimerkki HTML-sivusta:

- Ensimmäinen rivi ilmoittaa selaimelle, että tämä on HTML-dokumentti.
- Rivillä 2 aloitetaan html-koodi, joka loppuu viimeisellä rivillä (lopetusmerkki /html).
- Itse HTML-sivu rakentuu kahdesta osiosta: `head` ja `body`.
- `head`-osioon tulee yleensä esimerkiksi ohjelmakoodia, sivulle tulevia tyylimäärityksiä ja sivulle tilattavia ohjelmia muualta netistä.
- `body`-osioon kirjoitetaan kaikki selaimen ikkunassa näkyvät asiat. Selaimen ikkunassa näkyviä asioita voidaan sitten muunnella, poistaa tai lisätä ohjelmakoodeilla.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>

    Tässä välissä on nettiselaimen ikkunaan tuleva osuus

  </body>
 </html>
```

Mitkään muut muotoilut kuin välilyönnit eivät välity selaimelle. Kaikki muut muotoilut on tehtävä tageilla.

HTML-sivuille halutaan usein jonkinlaista toiminnallisuutta pelkän informaation näyttämisen lisäksi. Tähän liittyy monesti käyttäjän syötteiden lukeminen esimerkiksi painikkeilla, valintaruuduilla ja tekstisyötekentillä. Kaikki tämäntyyppiset elementit ovat <input>-tyyppisiä, ja niiden *type*-ominaisuus määrittää miten ne toimivat. Yleisimmät näistä ovat edellämainitut *button, checkbox*, ja *text*.

```
<input type="button" value="Click!">
```

Click!

```
<input type="checkbox" checked>
```

```
<input type="text" value="HTML & JavaScript">
```

HTML & JavaScript

Esimerkiksi valintaruudun tilan ja tekstisyötteen sisällön voi tarkistaa JavaScriptillä, kun taas painikkeeseen on yleensä sidottu jokin funktio. Niistä opitaan lisää ensi luvussa.

Lisäresursseja

HTML on ollut käytössä jo modernin Internetin alkuajoista asti, joten verkosta löytyy paljon lisäinformaatiota.

- Suuri osa HTML-elementeistä käsitellään esimerkiksi W3Schoolsin sivuilla.
- Tarkemmin asiat käsitellään muunmuassa Mozillan tarjoamassa dokumentaatiossa.

- Molemmat resurssit ovat englanniksi.
- W3Schools tarjoaa myös apua CSS:n kanssa - aiheen jota käsittelemme seuraavalla sivulla.

3.2. CSS

CSS

CSS (cascading style sheets)-tyylitiedostot ovat tiedostoja, joissa määritellään miten web-sivun elementit tulee näyttää käyttäjälle. HTML-kuvauskielellä määritellään web-sivun rakenne ja sisältö, tyylitiedostoilla sen ulkoasu.

Tyylitiedostoilla voidaan määritellä, miltä sivu näyttää. Tyylitiedosto on HTML-dokumentista erillinen tiedosto, joka sisältää erilaisia tyylimäärittelyjä. Tyylitiedostoja voi olla useita. Jotta HTML-dokumentti saa tyylitiedoston käyttöönsä, tulee tyylitiedoston sijainti määritellä `head`-elementin sisälle tyylitiedoston lataavaan elementtiin `link` seuraavasti:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" >
    <link rel="stylesheet" type="text/css" href="stylesheets/style.css">
    <title>Sivun otsikko (näkyy selaimen palkissa)</title>
  </head>
  <body>

        <!-- sivun sisältö: näin sivuille saa kommentin -->

  </body>
</html>
```

Elementille `link` kerrotaan viitattavan tiedoston tyyli (`rel="stylesheet"`), tyyppi (`type="text/css"`) ja sijainti `(href="sijainti.css"`). Sijainnista tulee selvitä tyylitiedoston nimi. Tyylitiedostojen päätteenä on `.css`. Esimerkiksi jos tyylitiedosto on tämän tiedoston sisältämän kansion sisällä olevassa kansiossa `stylesheets` ja tyylitiedoston nimi on `style.css`, asetetaan elementin `link` attribuutin `href` arvoksi `"stylesheets/style.css"`.

Alla on esimerkki yksinkertaisesta tyylitiedostosta.

```
body {
    background-color: rgb(200, 200, 200);
    margin: 0;
    padding: 0;
}
```

Yllä olevassa tyylitiedostossa kerrotaan, että elementin body (eli HTML-dokumentin rungon) taustaväri on rgb-arvolla kerrottuna 200, 200, 200, eli vaaleahko. Väriarvo rgb tulee sanoista red, green, ja blue, ja jokaisella arvolla kerrotaan värin määrän. Kunkin värin määrä ilmaistaan numerolla nollan ja 255 välillä. Jos jokaisen värin arvo on 0, on väri musta, ja jos jokaisen värin arvo on 255, on väri valkoinen. Lisäksi body-elementin ympärille jätetty alue (margin) sekä sen sisällön ympärille jätetty alue (padding) ovat koiltaan 0 pikseliä.

Seuraavassa esitetään muutamia CSS-määritteitä. Jos halutaan kaikkiin tekstikappaleisiin kaksinkertainen riviväli ja vihreä teksti, CSS-tiedostoon tehdään seuraavanlainen sääntö:

```
p {
  line-height: 2;
  color: green;
}
```

Nyt kaikki teksti joka on `<p></p>` -tagien sisällä näytetään kaksinkertaisella rivivälillä ja vihreällä tekstivärillä.

Mille tahansa elementille voidaan antaa ID-attribuutti, joka määrittelee sen yksilöllisesti. Sama ID-arvo voi esiintyä samalla sivulla vain yhden kerran. Jos esimerkiksi halutaan vain yhden kappaleen rivinväliksi 2 ja tekstinväriksi vihreä, toimitaan seuraavasti. Annetaan ensin halutulle elementille ID `korostus`. ja määritellään vastaava CSS-sääntö.

```
<p id="korostus">Korostettava tekstikappale</p>
```

```
#korostus {
  line-height: 2;
  color: green;
}
```

Huomaa #-merkki, jota käytetään CSS:ssä (mutta ei HTML:ssä) id:n merkitsemiseen.

Luokat (class) toimivat samalla tavoin kuin id:t, sillä erotuksella että luokan voi määritellä useammalle elementille seuraavan esimerkin mukaisesti.

```
<p class="korostus">Yksi korostettu tekstikappale.</p>
<p class="korostus">Toinenkin korostettu tekstikappale.</p>
```

```
.korostus {
  line-height: 2;
  color: green;
}
```

Huomaa piste (.) jota käytetään CSS:ssä luokan merkitsemiseen.

Rakennusvinkit

Tee web-sivut käyttökelpoisiksi huolimatta siitä, mitä selainta käytetään ja mitä laitetta tai näyttöä käyttäjä käyttää. Tämä tarkoittaa, että sivujen tulee olla luettavat huolimatta näytön koosta, resoluutiosta tai värien määrästä. Sivujen tulee olla käyttökelpoiset myös tulosteena, kuunneltuina lukijaohjelmalla tai käyttäen brailleselainta.

Miten voimme suunnitella näin mukautuvat ja käyttökelpoiset sivut?

1. Ensimmäiseksi ajattele sivujesi tehtävää, älä ulkonäköä. Palveluja jotka sivujesi on tarkoitus tuottaa käyttäjille. Anna sivujesi muodon paremminkin seurata toiminallisuutta kuin, että yrität tehdä tietyn näköistä ja laittaa sen toimimaan.
2. Älä käytä HTML-merkintää sivun ulkoasun määrittämiseen. Käytä tähän erillisiä tyylisivuja.
3. Käytä CSS-tyylisivuja oikein ehdottamaan sivun ulkoasua, ei kontrolloimaan sitä, silloin sivusi toimivat hienosti missä tahansa selaimessa myös tulevaisuudessa. Käytännössä voit tehdä näin esimerkiksi ehdottamalla tyylisivuilla mahdollisimman monia fontteja. Älä myöskään määritä fontin kokoa kiinteinä numeroina. Mukautuvampi tapa on määrittää eri elementeissä käytettyjen fonttien koko suhteessa toisiinsa prosentteina. Esimerkiksi pääotsikko 30% isompana kuin leipäteksti. Tai samoin sivun asettelussa voit määrittää sivun marginaalit prosentteina tai muina suhdelukuina käyttämättä kiinteitä lukuja.
4. Varo nojautumasta yksin värikoodien käyttämiseen välittämään tiettyä tarkoitusta. Monet meistä ovat värisokeita.

3.3. JavaScript

JavaScript

HTML on siis kuvauskieli web-sivujen rakenteen ja sisällön luomiseen ja CSS kieli web-sivustojen tyylin määrittelyyn. JavaScript on puolestaan kieli dynaamisen toiminnan lisäämiselle. JavaScriptillä voidaan saada aikaan Web-sivuja, jotka "vastaavat" käyttäjän toimenpiteisiin kuten hiiren klikkauksiin, syöttötietojen kirjoittamiseen ja sivulla liikkumiseen. Yleisemmin JavaScriptillä voidaan *ohjelmoida* erilaisia toimintoja, jotka selain suorittaa.

Alunperin ja perinteisesti JavaScript-ohjelmat sisältyvät HTML-sivuihin, joilla ne toteuttavat paikallisesti sivujen toiminnallisuuksia. Selaimissa on siis JavaScript-toteutus, tavallisesti tulkki. Kielen käyttö myös itsenäisesti selaimesta riippumatta on myös mahdollista. Erityisesti viime aikona sen käyttö myös palvelinohjelmistojen toteuttamiseen on yleistynyt.

JavaScript-koodi liitetään osaksi HTML-dokumenttia `script`-elementin avulla:

- Ohjelmointikäskyillä `<script>`-elementin (tagin) sisällä joko `head`- tai `body` -osioissa.
- Viittaamalla ulkoiseen JavaScript-lähdetiedostoon
- Määrittelemällä JavaScript-käsky HTML-attribuutin arvoksi.
- Tilannekäsittelijällä lomaketageissä.

Seuraavassa on esimerkki skriptin sijoittamisesta `<script>`-elementin sisään:

```
<html>
 <head>
 <title>JavaScript-esimerkki</title>
   <script type="text/javascript">
   // koodia....
   </script>
 </head>
<body>
</body>
</html>
```

Aivan kuten CSS-tyylimäärittelyt, JavaScript-lähdekoodit kannattaa erottaa HTML-dokumentista. JavaScript-tiedoston pääte on yleensä `.js` ja siihen viitataan elementillä `script`. Elementillä `script` on attribuutti `src`, jolla kerrotaan lähdekooditiedoston sijainti. Jos lähdekoodi on kansiossa `javascript` olevassa tiedostossa `code.js`, käytetään script-elementtiä seuraavasti: `<script src="javascript/code.js"></script>`. On huomattava, että `script`-elementti suljetaan poikkeuksellisesti erikseen vaikka se ei sisälläkään tekstiä. Seuraavassa esimerkissä skripti on sijoitettu ulkoiseen tiedostoon `JavaScript-koodit.js`, johon viitataan HTML-dokumentissa:

```
<html>
 <head>
 <title>JavaScript-esimerkki</title>
 <script type="text/javascript" src="JavaScript-koodit.js"></script>
 </head>
<body>
</body>
</html>
```

`script`-elementin sijoittamiselle dokumenttiin ei ole yhtä oikeata paikkaa, vaan se riippuu hyvin pitkälti suoritettavan koodin käyttötarkoituksesta. Mikäli koodi pitää suorittaa heti sivun latautuessa, on se sijoitettava dokumentin ylätunnisteeseen `head`-elementin sisään. Muuten sen voi sijoittaa myös dokumentin runkoonkin (eli `body`-elementin sisään). Myöskään `script`-elementtien lukumäärää ei ole rajoitettu millään tavoin.

Yleinen käytänne JavaScript-lähdekoodien sivulle lisäämiseen on lisätä ne sivun loppuun. Tämä johtuu mm. siitä, että selain lähtee hakemaan JavaScript-tiedostoa kun se kohtaa sen määrittelyn HTML-dokumentissa, jolloin kaikki muut toiminnot odottavat latausta odottamaan. Jos lähdekooditiedosto ladataan vasta sivun lopussa, käyttäjälle näytetään sivun sisältöä jo ennen Javascript-lähdekoodin latautumista, sillä selaimet näyttävät usein sivua käyttäjälle sitä mukaa kun se latautuu. Tällä luodaan tunne nopeammin reagoivista ja latautuvista sivuista.

Seuraavassa on esimerkki JavaScript-koodin lisäämisestä verkkosivulle sekä vastaava sivu selaimessa. Kansiossa `javascript` tiedostossa `code.js` on funktio `sayHello`, joka tulostaa konsolille viestin.

```
function sayHello() {
    console.log("BAD = browser application development");
}
```

HTML-dokumentti on seuraava:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" >
    <link rel="stylesheet" type="text/css" href="stylesheets/style.css">
    <title>Sivun otsikko (näkyy selaimen palkissa)</title>
  </head>
  <body>
    <header>
      <h1>Sivulla näkyvä otsikko</h1>
    </header>

    <article>
      <p>Sivuilla näytettävä normaali teksti on p-elementin sisällä. Alla on nappi,
         jota painamalla kutsutaan funktiota "sayHello".</p>
      <input type="button" value="Tervehdi" onclick="sayHello();" />
    </article>

    <!-- ladataan JavaScript-koodit tiedoston lopussa! -->
      <script src="javascript/code.js"></script>
  </body>
</html>
```

Ponnahduslaatikot

Ponnahduslaatikko on sivua ladattaessa tai toiminnolla ruudulle tulostettava laatikko, joka pysäyttää koodin suorittamisen kunnes käyttäjä kuittaa sen.

Huomautuslaatikko (Alert Box) vaatii käyttäjän kuittauksen eli "OK"-painikkeen klikkauksen. Sitä käytetään huomauttamaan käyttäjää jostain asiasta.

```
alert("sometext");
```

Vahvistuslaatikko (Confirm Box) vaatii käyttäjän kuittauksen, joka hyväksynnän ("OK") tai hylkäämisen ("Cancel"). Sitä käytetään, kun halutaan käyttäjältä kyllä tai ei vastaus. "OK" palauttaa arvon `true` ja "Cancel" arvon `false`.

```
confirm("sometext");
```

Kyselylaatikko (Prompt Box) pyytää käyttäjää syöttämää merkkejä ja hyväksymään tai hylkäämään sen. Sitä käytetään kun käyttäjältä halutaan jotain tietoa. Kyselylaatikossa on painikkeet "OK" ja "Cancel". "OK" palauttaa syötetyn merkkijonon ja "Cancel" arvon `null`.

```
prompt("sometext","oletusarvo");
```

HTML-dokumentin elementtien hallinta (Document Object Model, DOM)

Jotta JavaScriptin avulla voitaisiin kontrolloida tiettyjä www-sivun elementtejä, pitää elementteihin pystyä viittamaan jotenkin. Tätä varten selainikkuna (tai välilehti) sekä ikkunassa näytettävä dokumentti ovat edustettuina ohjelmaa suorittavassa Javascript-ympäristössä omina objekteinaan: `window` ja `document`.

window-objekti

`window`-objekti on selaimessa suoritettavan Javascriptin globaali objekti, eli objekti, jonka ominaisuutena kaikki globaalit muuttujat ovat. Siis, mikä tahansa kaikkien funktioiden ulkopuolella luotu muuttuja, eli globaali muuttuja, ilmestyy window-objektin ominaisuudeksi. Tai päin vastoin, jos window-objektiin luodaan uusi ominaisuus, siitä tulee globaali muuttuja, joka on käytettävissä kaikkialla, missä sitä ei ole “peitetty” saman nimisellä lokaalilla muuttujalla.

document-objekti

Objekti `document` edustaa koko selaimeen ladattua dokumenttia ja sen kautta Javascriptillä pääsee tutkimaan ja muokkaamaan mitä tahansa kohtaa dokumentista. Esimerkiksi web-sivuilla oleviin elementteihin tulee pystyä asettamaan arvoja, ja niitä tulee myös pystyä hakemaan. Dokumentissa oleviin elementteihin päästään käsiksi komennolla `document.getElementById("tunnus")`, joka palauttaa elementin, jonka id-attribuutti on "`tunnus`".

HTML-elementin arvon käsittely

Seuraavassa esimerkissä on tekstikenttä, jonka HTML-koodi on `<input type="text" id="tekstikentta"/>`. Kentän tunnus on siis `tekstikentta`. Elementtiin `tekstikenttä` päästään komennolla `document.getElementById("tekstikentta")`. Tekstikenttäelementillä on attribuutti `value`, joka voidaan tulostaa seuraavasti:

```
var arvo = document.getElementById("tekstikentta").value;
console.log("Tekstikentän arvo oli " + arvo);
```

Seuraavassa esimerkissä haetaan edellisen esimerkin tekstikenttä, ja asetetaan sille arvo 5:

```
document.getElementById("tekstikentta").value = 5;
```

Lopuksi tehdään ohjelma, jolla pyydetään käyttäjältä uusi kentän arvo:

```
document.getElementById("tekstikentta").value = prompt("Kirjoita jotain!");
```

Kaikilla elementeillä ei ole `value`-attribuuttia, vaan joillain näytetään niiden elementin sisällä oleva arvo. Elementin sisälle asetetaan arvo muuttujaan liittyvällä attribuutilla `innerHTML`.

HTML-elementtiin voi lisätä onclick-attribuutin, jolle määritellään arvoksi elementin klikkauksen yhteydessä suoritettava JavaScript-koodi:

```
<element onclick="ohjelmakoodi tai funktiokutsu">
```

4. Funktiot

4.1. Funktiot, sulkeumat

Funktiot ovat aliohjelmia, joiden avulla voidaan toiminnallisuus mahdollistaa myöhemmin käytettäväksi kirjoittamatta samaa logiikkaa useampaan paikkaan. Funktiot on esiteltävä ennen kuin niitä voidaan käyttää. Funktioilmoituksilla on myös samat nimeämiskäytännöt ja näkyvyysalueet kuin muuttujilla.

Funktioilmoituksen syntaksi on seuraavanlainen:

```
function funktioNimi(/*parametriluettelo*/) {// funktion runko alkaa

  /* Suoritettava(t) lause(et) */

}// funktion runko päättyy
```

Funktioilmoitus alkaa `function`-avainsanalla, jota seuraavat:

- `functionName,` ohjelmoijan nimeämä tunniste funktiolle
- `()` - kaarisulut, joiden sisälle valinnaisia (0 - 255) pilkulla erotettuja parametreja (funktiolle kutsuttaessa välitettäviä tietoja)
- `{}` - aaltosulut, joiden sisälle tulee funktion runko, eli funktion toiminnallisuus.

Seuraavassa esimerkissä esitellään funktio, joka palauttaa kutsuargumentin neliön.

```
const nelio = function(x) {
  return x * x;
};
```

Funktion kutsu argumentin arvolla 12:

```
console.log(nelio(12));
```

Toinen tapa funktion määrittelyyn on *funktiolausekkeen* (*function expression*) käyttö. Esimerkiksi seuraavassa ohjelmassa käytetään anonyymiä funktiota lukuvälin lukujen summan laskemiseen. Mielenkiintoista alla olevassa koodissa on se, että funktio on olemassa vain sen suorituksen ajan:

```
var lopputulos = (function(alku, loppu) {
    var summa = 0;
    for (var i = alku; i < loppu; i++) {
       summa += i;
    }
    return summa;
    })(1, 3);

console.log(lopputulos); // 3
```

Anonyymit funktiot ovat siis funktioita, joita ei kiinnitetä muuttujiin, jolloin ne eivät saa nimeä.

Funktiolla voi olla nolla tai useampia kutsuparametreja seuraavien esimerkkien mukaisesti.

```
const kilauta = function() {
  console.log("Pling!");
};

kilauta(); //(1)
```

```
const potenssi = function(kanta, eksponentti) {
  var tulos = 1;
  for (var laskuri = 0; laskuri < eksponentti;   laskuri++) {
    tulos *= kanta;
  }
  return tulos; //(2)
};

console.log(potenssi(2, 10)); //(3)
```

Ensimmäinen funktio "kilauttaa" tekstin "Pling" eikä sitä kutsuttaessa anneta kutsuparametria (1).

Jälkimmäiselle funktiolla annetaan kutsussa kantaluku ja eksponentti, ja se tulostaa luvun korotettuna eksponentin mukaiseen potenssiin (3). Funktio palauttaa tuloksen käyttäen `return`-lausetta (2).

Jos `return`-lauseelle ei anneta argumenttia tai `return`-lause puuttuu, funktio palauttaa hiljaisesti ja näkymättömästi arvon `undefined`.

Funktio muodostaa näkyvyysalueen (scope). Tämä tarkoittaa, että funktion muodolliset parametrit, paikalliset muuttujat ja paikalliset funktiot näkyvät – ovat käytettävissä – vain funktion sisällä. Tarkemmin sanoen noiden nimet eli tunnukset näkyvät vain funktion sisällä, eli ne ovat paikallisia (local).

Funktion ulkopuolella tehdyt määrittelyt ovat näkyvissä funktion sisällä, ne ovat julkisia (global).

Sisäkkäisten näkyvyysalueiden perusidea on, että "sisältä näkee ulos, mutta ulkoa ei näe sisään".

Sisemmällä näkyvyysalueella ympäröivän näkyvyysalueen tunnukselle voi antaa uuden merkityksen, joka peittää vanhan merkityksen seuraavan esimerkin mukaisesti.

```
function f() {
  var a=1, b=2;
  function ff() {
    var a = 11; // peittää ympäröivän funktion a:n
    write(a+" "+b); // 11 2 // ja tämä koskee siis myös var-muuttujia
  }
  ff();
  write(a+" "+b); // 1 2
}

f();
```

Funktion *sidotuiksi muuttujiksi* kutsutaan funktion muodollisia parametreja sekä funktion sisällä määriteltyjä paikallisia muuttujia. Toisin sanoen muuttujia, joilla on merkitys vain funktion sisällä ja jotka suoritusaikana ovat olemassa vain funktion suorituksen ajan.

Funktion *vapaiksi muuttujiksi* kutsutaan sellaisia funktioon sisältymättömiä, mutta funktiossa viitattuja muuttujia, jotka funktiosta näkyvyyssääntöjen sallimana nähdään.

```
function f(x) {
  var a=1, b=2;

  var g = function(y) {
    var c=3;
    return a+b+x+ // vapaita muuttujia
    y+c; // sidottuja muuttujia
  }
  return g(4);
}

write(f(5)); // 15
```

Tässä esimerkissä funktioliteraalissa määritellään muuttujien tunnukset y ja c. Nämä muuttujat on sidottu funktioliteraaliin ja niillä on merkitys vain literaalin sisällä. Vapaiden muuttujien tunnukset x, a ja b puolestaan on määritelty funktioliteraalin ulkopuolella Näkyvyyssäännöt sallivat näiden muuttujien käytön koska "sisältä näkyy ulos".

Funktion määritys voi myös tapahtua sen kutsun jälkeen seuraavan esimerkin mukaisesti:

```
console.log("Tulevaisuudessa:", tulevaisuus());
function tulevaisuus() {
  return "Ei ole lentäviä autoja!";
}
```

Peräkkäisten ja sisäkkäisten funktioiden suorituksessa käytetään *pinoa* (stack) tallettamaan paluutila ja -osoite. Jos esimerkiksi rekursiiviset funktiot kuluttavat liikaa tilaa pinosta, seuraa virhetilanne: “out of stack space” tai “too much recursion”. Seuraavan esimerkin funktiokutsut kuluttavat pinon:

```
function kana() {
  return muna();
}
function muna() {
  return kana();
}
console.log(kana() + " oli ensin.");
```

Funktion kutsussa voi olla eri määrä kutsuargumentteja, kuin funktion esittelyssä. Ylimääräiset argumentit hylätään ja puuttuvat korvataan `undefined`-määrittelyllä. Seuraava esimerkki palauttaa kutsuargumentin neliön huolimatta ylimääräisistä kutsuargumenteista:

```
function nelio(x) { return x * x; }
console.log(nelio(4, tosi, "siili"));
// → 16
```

`undefined`-määritettä voidaan käyttää seuraavan esimerkin mukaisesti;

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}
console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

Jos funktion `minus` kutsussa annetaan vain yksi argumentti, funktio palauttaa sen vastaluvun. Jos kutsussa on kaksi argumenttia, funktio palauttaa niiden erotuksen.

Mikäli funktion esittelyssä argumentille annetaan =-operaattorin jälkeen arvo, funktio käyttää sitä, jos argumentti puuttuu funktiota kutsuttaessa. Seuraavan esimerkin funktio palauttaa ensimmäisen parametrin arvon korotettuna toisen kutsuparametrin mukaiseen potenssiin.

```
function potenssi(kanta, eksponentti = 2) {
  var tulos = 1;
  for (var laskuri = 0; laskuri < eksponentti; laskuri++) {
    tulos *= kanta;
  }
  return tulos;
}
```

Jos funktion `potenssi` toinen kutsuparametri puuttuu, funktio palauttaa ensimmäisen argumenti toisen potenssin:

```
console.log(potenssi(4));
// → 16
```

Funktio voi myös olla rekursiivinen eli kutsua itseään seuraavan esimerkin mukaisesti.

```
function potenssi(kanta, eksponentti) {
  if (eksponentti == 0) {
    return 1;
  } else {
    return kanta * potenssi(kanta, eksponentti -1);
  }
}
```

JavaScript-implementaatiossa edellinen rekursiivinen funktio on kolme kertaa hitaampi kuin aiemmin silmukalla toteutettu. Kuitenkin rekursion käyttö on monessa tapauksessa järkevää.

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

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

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

Sulkeuma ei ole funktiokohtainen vaan kutsukohtainen. Jokaisella ulomman funktion kutsukerralla muodostuu uusi sulkeuma, jossa suuri sen kutsukerran sisäfunktio elää.

Kuten jo edellä nähtiin, JavaScriptissä funktioita voi ohjelmoida myös ilman nimeä, **anonyymeinä funktioina**, ohjelmatekstissä näitä kutsutaan **funktioliteraaleiksi**:

```
// funktioliteraali muuttujan arvoksi
var sum = function(a,b) {return a+b}
console.log(sum(1,2));
toinensumma = sum;  // kopioidaan viite funktio-olioon
console.log(toinensumma(3,4));
```

JavaScript sisältää näppärän tavan kirjoittaa funktioliteraaleja seuraavan esimerkin mukaisesti:

```
var sum = (a,b) => a+b
console.log(sum(1,2));
toinensumma = sum;
console.log(toinensumma(3,4));
```

JavaScriptin funktiot ovat ns. *first-class*-arvoja, mikä tarkoittaa, että niitä voidaan esimerkisi numeeristen arvojen tapaan sijoittaa muuttujiin ja vaikkapa taulukon alkoiden arvoiksi, funktioita voidaan välittää parametreina ja palauttaa funktioiden arvona, ...

Funktioliteraalit ovat erityisen käyttökelpoisia välitettäessä funktioita parametreina toisille funktioille. Seuraavassa esimekissä sarjan summan laskevalle funktiolle annetaan termin laskentakaava parametrina:

```
function sarjanSumma(termi, raja) {
  var summa = 0;
  for (var i=1; i<=raja; ++i)
    summa += termi(i);
  return summa;
}

aritmeettinen = sarjanSumma(i => i, 10);
console.log(aritmeettinen);  // 55

harmoninen = sarjanSumma(i => 1/i, 10);
console.log(harmoninen);    //  2.9289682539682538
```

5. Objektit ja taulukot

5.1. Oliot (objektit)

JavaScript tukee *olioajattelua*, vaikkei olekaan täysiverinen oliokieli. Yksinkertaistettuna **olio** voidaan ajatella jonkin asian kantailmentymänä tai yleistyksenä. Olioilla on kenttiä eli attribuutteja, jotka muodostuvat tunnuksesta ja tunnuksen tarkoittamasta asiasta, tietokentästä tai metodista. JavaScriptissä näitä attribuutteja kutsutaan usein ominaisuuksiksi (*property*). Olio on yksinkertaisesti *assosiaatiotaulukko*, joukko pareja *avain-arvo*. Avaimet ovat yksikäsitteisiä ja kuhunkin avaimeen liittyy arvo, joka voi olla tietokenttä, funktio, olio, ...

Olio luodaan kirjoittamalla *olioliteraali* ohjelmatekstiin. JavaScriptissa tätä ilmausta kutsutaan nimellä *object initializer*. Olion määrittely alkaa aaltosululla `{`, jota seuraa muuttujan nimi ja sille annettava arvo. Arvon asetus oliomuuttujalle tapahtuu kaksoispisteellä, esimerkiksi `nimi: "Minttu"`. Useampia muuttujia voi määritellä pilkulla eroteltuna. Olion määrittely lopetetaan sulkevaan aaltosulkuun `}`. Olion muuttujiin päästään käsiksi piste-notaatiolla.

```
var retki =
  {matka: 150,
   aika: 2,
   nopeus: function() {return this.matka/this.aika}
  }
console.log(retki.nopeus());  // 75
```

Kun olion sisällä viitataan olion kenttään, tarvitaan viiteavain "tähän olioon", eli `this`.

Seuraavan esimerkin ohjelma käyttää olion ulkopuolella määriteltyjä muuttujia, ja funktion palauttama arvo poikkeaa edellisestä.

```
var matka=20, aika=10
var retki =
  {matka: 150,
   aika: 2,
   nopeus: function() {return matka/aika}
```

```
  }
console.log(retki.nopeus());  // 2 !!
```

Olion ominaisuuteen voidaan viitata pistenotaation sijasta myös hakasuljenotaatiolla: `nimi['ominaisuus']`. Hakasuljemerkinnässä hakasulkeiden väliin tulee ominaisuuden nimi merkkijonona. Tästä on ainakin kahdenlaista hyötyä. Ensinnäkin, jos ominaisuuden nimi sisältää erikoismerkkejä, kuten `-, .` tai välilyönti, ei niitä voida käyttää pistenotaatiolla. Toiseksi, hakasuljemerkintä tekee mahdolliseksi viitata ominaisuuteen, jonka nimi on tallennettu muuttujaan. Tämä tuo objektien käyttöön dynaamisuutta. Tämäkään ei ole mahdollista pistenotaatiolla

Olio voidaan luoda myös seuraavasti:

```
var retki = new Object();
retki.matka = 150;
retki.aika = 2;
retki.nopeus = function() {return this.matka/this.aika}

console.log(retki.nopeus());  // 75
```

Kolmas tapa olion luontiin on *konstruktorifunktion* käyttö. Omien olioiden luonti tapahtuu funktioiden avulla. Oliot ovat funktioiden ilmentymiä, jotka luodaan new-avainsanalla. Määreellä `this` kerrotaan, että käsitellyn muuttujan arvo liittyy juuri tähän olioon.

```
function Matkailu(matka,aika) {
  this.matka = matka;
  this.aika = aika;
  this.nopeus = function() {return this.matka/this.aika}
}

var retki = new Matkailu(150,2);
console.log(retki.matka);     // 150
console.log(retki.nopeus());  // 75

var f1 = new Matkailu(150,0.5);
console.log (f1.aika);        // 0.5
console.log(f1.nopeus());     // 300
```

Edellä esitettytapa kirjoittaa aksessorit konstruktorifunktioon on kuitenkin huono: näin jokainen olio saa oman kopionsa `nopeus`-funktiosta!

Parempi on liittää nopeuden laskemisen taito `Matkailu`-funktion ns. *prototyyppiolioon*:

```
function Matkailu(matka,aika) {
  this.matka = matka;
  this.aika = aika;
}
Matkailu.prototype.nopeus = function() {return this.matka/this.aika}

var retki = new Matkailu(150,2);
console.log(retki.nopeus());           // 75

var f1 = new Matkailu(150,0.5);
console.log(f1.nopeus());              // 300
```

Prototyyppi sisältää tiedon funktioon liittyvistä attribuuteista. Prototyypin kautta lisättävien funktioiden avulla päästään käsiksi olioiden `this`-viitteeseen, mikä mahdollistaa olion sisäisen tilan muuttamisen.

Moni muu olio-ohjelmointiin perustuva kieli tukee luokkia (class). JavaScriptiin on myöhemmin lisätty luokalle tuki, tosin oliot seuraavat pitkälti samoja sääntöjä vaikka käytettäisiinkin luokkaa konstruktorifunktion sijasta. Esimerkiksi olioiden taustalla on edelleen prototyyppi, vaikka ne luotaisiin `class`-sanaa käyttäen kuten alla olevassa esimerkissä. JavaScriptin luokkaa kannattaa siis ajatella vain

vaihtoehtoisena tapana kirjoittaa konstruktorifunktio ja olion prototyypin metodit. Luokka tarvitsee aina sisäisen konstruktorimetodin `constructor`.

```
class Matkailu {
  constructor(matka, aika) {
    this.matka = matka;
    this.aika = aika;
  }
  nopeus() {
    return this.matka / this.aika;
  }
}

var retki = new Matkailu(150,2);
console.log(retki.nopeus()); // 75

var f1 = new Matkailu(150,0.5);
console.log(f1.nopeus()); // 300
```

5.2. Taulukot

Taulukko (array) on objekti, joka on tarkoitettu erityisesti järjestetyn alkiojoukon luomiseen ja käsittelyyn. Taulukko eroaa tavallisesta objektista siinä, että sen avaimina käytetään kokonaislukuja ja sillä on `length`-ominaisuus, joka kertoo aina sen pituuden. (Pituus on yhtä pienempi kuin suurin käytössä oleva indeksi.) Kuten tavallisilla objekteilla, taulukkoon tallennetut arvot voivat olla mitä tahansa tyyppiä, myös funktioita.

Taulukkotyyppisiä muuttujia voidaan luoda komennolla `[]`, joka on lyhenne komennolla `new Array();`. Esimerkiksi seuraavassa luodaan 3 paikkaa sisältävä taulukko.

```
var salasanat = ["salasana", "alasanas", "lasagna"];
```

Taulukon avaimina olevia numeroita kutsutaan *indekseiksi* ja tietyllä indeksillä olevaan paikkaa viitataan hakasuljemerkinnällä aivan samoin kuin objekteillakin.

Taulukoille ovat käytettävissä metodit, jotka käsittelevät niitä jonon tai pinon tapaan. Tämä tarkoittaa alkioiden lisäämistä ja poistamista taulukon päihin tai päistä. Nämä operaatiot muuttavat taulukon pituutta.

Taulukon käsittelyn metodit ovat:

Metodi	Toiminto	Palauttaa	Esimerkki
`push()`	Lisää alkioita taulukon loppuun	Taulukon uusi pituus	`lista.push('Aku')`
`pop()`	Poistaa alkion taulukon lopusta	Taulukon viimeinen alkio	`lista.pop()`
`unshift()`	Lisää alkion taulukon alkuun	Taulukon uusi pituus	`lista.unshift('Mikki')`
`shift()`	Poistaa alkioita taulukon alusta	Taulukon ensimmäinen alkio	`lista.shift()`

Taulukkoon voidaan sijoittaa uusia arvoja indeksin avulla tai käyttäen metodeja `push` ja `unshift`.

Seuraavassa esimerkissä käytetään `push`-metodia taulukon neljännen alkion lisäämiseen.

```
var tiedot = ["Minttu", 1983];
tiedot[2] = "Ville";
tiedot.push("Saara");

console.log(tiedot[3]); // Saara
```

Jos taulukkoon lisätään uusia arvoja yli taulukon nykyisen pituuden, taulukon pituus kasvaa ja väliin jäävillä indekseillä olevat paikat jäävät tyhjiksi, arvoina määrittelemätön (`undefined`). Taulukon `length`-

kentän arvoksi tulee (normaalisti!) tuo uusi indeksi plus yksi.

```
var lista = ['a', 'b', 'c'];
console.log(lista.length);        // 3
lista[9] = 'j';
console.log(lista.length);        // 10
console.log(lista);               // ['a', 'b', 'c',,,,,,,'j']
```

Taulukolta voi kysyä, monentenako kysytty alkio on taulukossa, `indexOf()`-metodilla. Jos alkio on taulukossa, vastaus on ensimmäinen indeksi, josta se löytyy. Jos alkiota ei ole taulukossa, on vastaus `-1`. Jos alkio on taulukossa useasti, vastauksena on ensimmäinen indeksi, josta se löytyy. Löydettävän alkion tulee olla operaattorin `===` kannalta sama kuin haettavan. Ei siis riitä, että se on vain samanlainen, vaan sen on oltava sama.

```
var lista = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'a', 'b', 'c',];
lista.indexOf('c');          // 2
lista.indexOf('h');          // -1

var arvoja = [1, 2, 3, 0, 5, false, true];
arvoja.indexOf(false);       // 5
```

Taulukosta voidaan kopioida keskeltä osia `slice()`-metodilla. Sen syntaksi on: `taulukko.slice(alku, loppu)`. Metodi palauttaa uuden taulukon, jossa ovat alkuperäisen taulukon alkiot indeksistä `alku` alkaen indeksiin `loppu-1` saakka (`loppu` on siis ensimmäinen indeksi, joka ei tule mukaan!). Jos loppu-indeksiä ei anneta, kopiointi tapahtuu alkuperäisen taulukon loppuun saakka. Indeksit voivat olla myös negatiivisia, jolloin niiden sijainti lasketaan taulukon lopusta. Alkuperäinen taulukko pysyy muuttumattomana.

Taulukosta voidaan poistaa tai siihen voidaan lisätä alkoita keskelle `splice()`-metodilla. Sen syntaksi on muotoa: `taulukko.splice(indeksi, montako, alkio1, ... , alkioX)`. Näistä `indeksi` kertoo, mitä kohtaa taulukosta operoidaan. Parametri `montako` kertoo, kuinka monta alkiota taulukosta poistetaan, alkaen kohdasta `indeksi`. Loput parametrit ovat alkioita, jotka laitetaan tämän jälkeen taulukkoon kyseiseen kohtaan. Metodi palauttaa listan poistetuista alkioista.

Kaksi taulukkoa voidaan liittää peräkkäin `concat()`-metodilla. Metodi ei muuta alkuperäistä taulukkoa vaan palauttaa uuden taulukon, jossa molempien taulukoiden alkiot ovat peräkkäin.

```
var lista = [1, 2, 3, 4];
var taulukko = ['a', 'b', 'c'];
var uusi = lista.concat(taulukko);     // [1, 2, 3, 4, 'a', 'b', 'c']
```

Taulukoita voidaan järjestää `sort()`-metodiin avulla. Sort-metodi järjestää alkuperäisen taulukon uudelleen ja palauttaa järjestetyn taulukon. Oletuksena `sort()` järjestää alkiot merkkijonoina akkosjärjestykseen.

```
var nimet = ['Eemeli', 'Celcius', 'Aarne', 'Daavid','Bertta',  'Frans'];
nimet.sort();    // ["Aarne","Bertta","Celcius","Daavid","Eemeli","Frans"]

var numerot = [6, 3, 1, 7, 10, 21];
numerot.sort();  // [1,10,21,3,6,7]
```

6. Korkeamman asteen funktiot

6.1. Korkeamman asteen funktiot

Abstrahointi

Verrataan seuraavia kahta funktiota keskenään.

```
var total = 0, count = 1;
while (count <= 10) {
```

```
  total += count;
  count += 1;
}
console.log(total);
```

ja

```
console.log(sum(range(1, 10)));
```

Molemmat suorittavat saman tehtävän, muttä jälkimmäinen on huomattavasti yksinkertaisempi, mikä vähentää ohjelmointivirheiden mahdollisuutta. Ohjelmointitekniikassa käytetään termiä *abstrahointi* (*abstraction*), kun turhat yksityiskohdat piilotetaan ja keskitytään varsinaiseen ongelmaan.

Yksittäisten funktioiden lisäksi abstrahointia voidaan käyttää silmukoissa seuraavan esimerkin mukaisesti. Siinä funktion kutsuargumenttina on toinen funktio.

```
function repeat(n, action) {
  for (var i = 0; i < n; i++) {
    action(i);
  }
}
repeat(3, console.log);
// → 0
// → 1
// → 2
```

Korkeamman asteen funktiot

Funktioita, joiden argumenttina on toinen funktio, tai jotka palauttavat toisen funktion, kutsutaan *korkeamman asteen funktioiksi* (*higher-order functions*).

Funktio voi esimerkiksi luoda uusia funktioita seuraavan esimerkin mukaisesti:

```
function greaterThan(n) {
  return m => m > n;
}
var greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

Funktio voi myös muuttaa toista funktiota:

```
function noisy(f) {
  return (...args) => {
    console.log("calling with", args);
    var result = f(...args);
    console.log("called with", args, ", returned", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1] , returned 1
```

Seuraava esimerkki valaisee korkeamman asteen funktion käyttöä.

```
// Korkeamman asteen funktio not() palauttaa
// uuden funktion, joka vie argumenttinsa
// funktiolle f ja palauttaa f:n paluuarvon negaation
function not(f) {
  return function() { // Palauttaa uuden funktion
                      // joka kutsuu f:ää
    var result = f.apply(this, arguments);
```

```
    return !result;   // ja palauttaa negaation.
  };
}
// Funktio parillisuuden tarkastamiseksi
var even = function(x) {
  return x % 2 === 0;
}
// Funktio, joka toimii päinvastoin
var odd = not(even);
[1,1,3,5,5].every(odd); // => true: jokainen alkio on pariton
```

Yleisimmin käytössä oleva korkeamman asteen funktio JavaScriptissä on taulukon `forEach`-metodi, joka ottaa parametrina yhden funktion ja suorittaa sen jokaiselle taulukon solulle.

Funktio `map()` luo uuden taulukon, jonka soluihin kohdistetaan annettu funktio:

```
var new_array = arr.map(function callback(currentValue[, index[, array]]) {
    // Return element for new_array
}[, thisArg])
```

Uusi taulukko on saman mittainen kuin alkuperäinen, mutta sen alkiot on kuvattu funktion avulla. Seuraava esimerkki valisee `map()`-funktion toimintaa:

```
const users = [
  {
    name: 'Ville',
    age: 25
  },
  {
    name: 'Kalle',
    age: 30
  },
  {
    name: 'Jussi',
    age: 1
  }
];

const newData = users.map( user => {
  return {
    ...user,
    onLapsi: user.age < 2 ? true: false
  }
});

console.log(newData);
```

Funktio `filter()` luo uuden taulukon, jonka alkioksi tulevat annetun funktion ehdon täyttävät alkiot:

```
var newArray = arr.filter(callback(element[, index[, array]])[, thisArg])
```

Seuraava esimerkki valaisee `filter()`-funktion toimintaa:

```
const users = [
  {
    name: 'Ville',
    age: 25
  },
  {
    name: 'Kalle',
    age: 30
  },
  {
```

```
    name: 'Jussi',
    age: 1
  }
];

function onAikuinen(user) {
  return user.age >= 18;
}

const aikuisetUsers = users.filter(onAikuinen);

console.log(aikuisetUsers);
```

7. Olioiden ominaisuuksia (enkapsulointi, metodit, prototyypit, luokat, ...)

7.1. Oliot, perintä, luokat

Olion voidaan ajatella olevan jonkin asian yleinen käsite tai esimerkiksi kokoelma tietoja. Olio sisältää ominaisuuksia eli attribuutteja jotka tallennetaan muuttujiin sekä metodeja, joilla käsitellään olion sisältämää tietoa. Olio voidaan luoda yksinkertaisesti aaltosulkujen avulla seuraavasti:

```
var noora = {nimi: "Noora", ika: 35};
```

Olion ominaisuudet voidaan määritellä luonnin jälkeen seuraavasti:

```
var milla = {}; milla.nimi = "Milla"; milla.ika = 18;
```

Olio voidaan luoda myös käyttäen object-määritettä:

```
var noora = new Object(); noora.nimi = "Noora"; noora.ika = 35;
```

Se millä tavalla oliota lähdetään luomaan, riippuu aina käyttötarkoituksesta. Yllä esitetyt tavat ovat nopeita ja niiden avulla voidaan luoda yksittäisiä kertakäyttöoliota. joilla ei ole yhteisiä ominaisuuksia.

Yksi tapa luoda olioita JavaScriptissa on käyttää *konstruktorifunktiota*. Seuraavassa on esimerkki olion määrittelystä konstruktorifunktion avulla ja olion ilmentymän luonnista:

```
function Auto(merkki, malli){
  this.merkki = merkki
  this.malli = malli
}

kosla = new Auto('Ford', 'Focus')
```

Konstruktorifunktiot on syytä nimetä isolla alkukirjaimella erottamaan ne normaaleista funktioista. Konstruktorifunktiota, jossa käytetään this-toimintoa, ei saa kutsua normaalin funktion tapaan, koska tällöin "`this`" viittaa globaaliin olioon, joka voi aiheuttaa kummallisia bugeja.

Perintä vähentää koodin turhaa kopioimista. JavaScriptissa olioiden välinen periytyminen hoidetaan prototyyppien avulla. Käyttämällä funktion `call`-toiminnallisuutta pystytään perintä tekemään luokkapohjaisten kielten tapaan:

```
function Auto(merkki, malli){
  this.merkki = merkki
  this.malli = malli
}
function Rekka(merkki, malli) [
  Auto.call(this, merkki, malli);
  this.vaihteita = 6;
}
Rekka.prototype = new Auto();
biili = new Rekka("Sisu", "Kontio");
```

```
console.log(biili.merkki);    // "Sisu"
console.log(biili.vaihteita); // 6
```

Konstruktorifunktio `Rekka` kutsuu aluksi konstruktorifunktiota `Auto`, jolloin saadaan tässä oliossa määritellyt kentät käyttöön myös `Rekka`-olioille. `Rekka`-olio laajentaa sitten `Auto`-oliota lisäämällä oman vaihteita-kentän.

Periaatteena periytymisen hyödyntämisessä on se, että olioon laitetaan vain kyseiselle oliolle yksilölliset muuttujat ja hyödynnetään "yläolion" ominaisuuksia muuten. Näin vältytään koodin kopioimiselta.

Periytymishierarkiassa funktioiden prototyyppiolioihin voidaan liittää kenttien lisäksi *aksessorimetodeita*. Jos peritty aksessori asettaa arvon perittyyn kenttään, `this` viittaa perineeseen olioon. Jos perineellä olioilla ei vielä ole omaa versiota peritystä kentästä, sellainen luodaan. Seuraava esimerkki valaisee tätä:

```
function Elain(kg) {
  this.paino = kg || 0; // kelvoton paino nollaksi
}
Elain.prototype.syo =
   function (maara) {this.paino += maara;};
Elain.prototype.kuluta =
   function (maara) {this.paino -= maara;};

function Tuotantoelain(kg, litr) {
  Elain.call(this, kg);
  this.tuotto = litr || 0; // kelvoton tuotto nollaksi
}
Tuotantoelain.prototype = new Elain();
Tuotantoelain.prototype.tuota =
   function () {return this.tuotto;};

function Lehma(nimi, kg, litr) {
  Tuotantoelain.call(this, kg, litr);
  this.nimi = nimi || "nimetön";   // kelvoton nimi oletusarvoksi
}
Lehma.prototype = new Tuotantoelain();
Lehma.prototype.toString =
   function () { return this.nimi+
                 ": "+this.paino+" kg, "+
                  this.tuotto+" litraa"
               };

var m = new Lehma("Mansikki", 560, 12);
console.log(m.toString()); // Mansikki: 560 kg, 12 litraa
```

8. Regular expressions

8.1. Regular expressions

Regular expressions ovat mallineita (pattern), joita käytetään etsittäessä ja korvattaessa merkkikombinaatioita merkkijonoista. JavaScriptissä ne ovat myös olioita.

Mallineita käytetään RegExp `exec`- ja `test` -metodeissa, ja String `match`-, `matchAll`-, `replace`-, `search`- ja `split` -metodeissa.

Regular expression luodaan käyttäen kauttaviivojen väliin sijoitettua literaalia, tai käyttäen RegExp-funktion konstruktoria:

```
var re = /ab+c/;
var re = new RegExp('ab+c');
```

Edellinen määrittely käännetään skriptin latauksen yhteydessä ja jälkimmäinen ajoaikana.

Yleisessä muodossa määrittely on:

```
/pattern/modifiers;
```

Modifiers-kytkimiä käytetään tarkentamaan hakuja. Ne ovat:

```
i      Haussa ei erotella pieniä ja isoja kirjaimia
g      Haetaan kaikki esiintymät
m      Haetaan useammilta riveiltä
```

Tavallisimmin käytetyt string-metodit ovat `match()`, `search()` ja `replace()`.

`match()` etsii ja palauttaa annetun merkkijonon:

```
var str = "Welcome to GEEKS for geeks!";
var res = str.match(/eek/i); // EEK
```

`search()` etsii annetun merkkijonon ja palauttaa sen sijainnin seuraavan esimerkin mukaisesti:

```
var str = "Visit MetSchools";
var n = str.search(/metschools/i); // 6
```

`replace()` korvaa merkkijonon toisella. Seuraava esimerkki valaisee tätä:

```
var str = "Visit Microsoft!";
var res = str.replace(/microsoft/i, "MetSchools"); // Visit MetSchools
```

Mallinteissa voidaan käyttää seuraavankaltaisia tarkenteita:

```
[abc]  Haetaan mitä tahansa hakasuluissa olevista
       merkeistä
[^abc] Haetaan muita kuin hakasuluissa olevia merkkejä
[0-9]  Haetaan mitä tahansa hakasuluissa olevista
       numeroista
[^0-9] Haetaan muita kuin hakasuluissa olevia
       numeroita (ei-numeerinen)
(x|y)  Haetaan |-merkillä erotettuja vaihtoehtoja
+      Yksi tai useampia, esim. [0-9]+ voi olla
       123, 000
*      Nolla tai useampia, esim. [0-9]* voi olla
       tyhjä, 123, 000
?      Nolla tai yksi
```

Mallinteissa voidaan käyttää seuraavia metamerkkejä:

```
.      Haetaan yhtä merkkiä (ei rivinvaihto/newline)
\n     Haetaan rivinvaihtoa
\d     Haetaan numeroa
\s     Haetaan ei-kaiuttuvaa merkkiä (whitespace)
\b     Haetaan sanan alusta tai lopusta
\uxxxx Haetaan Unicode-merkkiä xxxx
```

Mallinteissa voidaan käyttää seuraavia määritteitä (quantifier):

```
n+     Haetaan merkkijonoa, jossa on vähintään yksi n
n*     Haetaan merkkijonoa, jossa nolla tai useampi n
n?     Haetaan merkkijonoa, jossa nolla tai yksi n
n{X}   Haetaan merkkijonoa, jossa X kpl n
n{X,Y} Haetaan merkkijonoa, jossa X-Y kpl n
n{X}   Haetaan merkkijonoa, jossa vähintään X kpl n
n$     Haetaan merkkijonoa, joka päättyy n:ään
^n     Haetaan merkkijonoa, joka alkaa n:llä
?=n    Haetaan merkkijonoa, jonka perässä on jono n
?!n    Haetaan merkkijonoa, jonka perässä ei ole jono n
```

Seuraavat esimerkit valaisevat kytkinten ja määritteiden toimintaa.

```
// Haetaan vähintään yksi "o"
var str = "Hellooo World! Hello MetSchools";
var patt1 = /o+/g;
var result = str.match(patt1); // ooo,o,o,oo

// Haetaan "l", jota seuraa nolla tai useampi "o"
var str = "Hellooo World! Hello MetSchools!";
var patt1 = /lo*/g;
var result = str.match(patt1); // l,looo,l,l,lo,l

// Haetaan "1", jota seuraa nolla tai yksi "0"
var str = "1, 100 or 1000?";
var patt1 = /10?/g;
var result = str.match(patt1); // 1,10,10
```

RegExp-metodi `test()` etsii merkkijonoa, ja palauttaa arvon true, jos se löytyy:

```
var patt = /e/;
patt.test("The best things in life are free!"); // true
```

Edellinen ohjelma voidaan kirjoittaa lyhyemmin:

```
/e/.test("The best things in life are free!"); // true
```

RegExp-metodi `exec()` etsii merkkijonoa ja palauttaa sen oliona:

```
var obj = /e/.exec("The best things in life are free!");
```

Seurava esimerkki valaisee RegExp-käyttöä.

```
<!DOCTYPE html>
<!-- JSRegexNumbers.html -->
<html lang="en">
<head>
<meta charset="utf-8">
<title>JavaScript Example: Regex</title>
<script>
var inStr = "abc123xyz456_7_00";

// RegExp.test(inStr):lla testataan sisältääkö mallineen:
// palautetaan true tai false
console.log(/[0-9]+/.test(inStr));  // true

// String.search(regex):lla teststaan sisältääkö mallineen:
// palautetaan alkuindeksi tai -1
console.log(inStr.search(/[0-9]+/));  // 3

// String.match():lla ja RegExp.exec():lla etsitään
// alijoukko, viittaus ja indeksi
console.log(inStr.match(/[0-9]+/));
  // ["123", input:"abc123xyz456_7_00", index:3, length:"1"]
console.log(/[0-9]+/.exec(inStr));
  // ["123", input:"abc123xyz456_7_00", index:3, length:"1"]

// Käytetään g (global) -optiota
console.log(inStr.match(/[0-9]+/g));
  // ["123", "456", "7", "00", length:4]
// RegExp.exec() g-kytkimellä toistettuna:
// Haku toistuu, kunnes viimeinen vastaavuus löytyy.
// Sijoitetan RegExp.lastIndex:iin.
var pattern = /[0-9]+/g;
var result;
while (result = pattern.exec(inStr)) {
  console.log(result);
  console.log(pattern.lastIndex);
```

```
    // ["123"],  6
    // ["456"], 12
    // ["7"],   14
    // ["00"],  17
}

// String.replace(regex, replacement):
console.log(inStr.replace(/\d+/, "**"));
  // abc**xyz456_7_00
console.log(inStr.replace(/\d+/g, "**"));
  // abc**xyz**_**_**
</script>
</head>
<body>
  <h1>Hello,</h1>
</body>
</html>
```

9. Moduulit

9.1. Moduulit

Moduulit toteutetaan anonyymien funktioiden avulla. Muuttujien funktionäkyvyyden takia muuttujat voidaan kapseloida anonyymin funktion sisään, jolloin niihin ei pääse käsiksi funktion ulkopuolelta. Anonyymin funktion sisälle määritellyt funktiot pääsevät käsiksi muuttujiin, jolloin sisäfunktioissa voidaan muokata muuttujien arvoja. Moduuli palauttaa moduulissa määritellyn rajapinnan, jossa on viittaukset sisäfunktioihin.

Seuraava esimerkki valaisee moduulien käyttöä. Esimerkissä rakennetaan kaupan hallinnointiin tarvittava järjestelmä. Siinä luodaan ostoskorimoduuli, joka tarjoaa julkisen rajapinnan tuotteiden lisäämiseen ja tuotteiden lukumäärän laskemiseen.

```
var kauppa = {};

kauppa.ostoskori = (function() {
  var ostokset = [];
  function lisaaOstos(tuotteenNimi) {
    if(!ostokset[tuotteenNimi]) {
      // jos tuotetta ei ole lisätty ostoskoriin, lisätään se sinne
         ostokset[tuotteenNimi] = 0;
    }
    // kasvatetaan tuotteen lukumäärää yhdellä
    ostokset[tuotteenNimi]++;
  }

  function tuotteitaYhteensa() {
    var lukumaara = 0;
    for(var tuotteenNimi in ostokset) {
      lukumaara += ostokset[tuotteenNimi];
    }
    return lukumaara;
  }

  // rajapinta
  return {
    lisaa: lisaaOstos,
    tuotteidenLukumaara: tuotteitaYhteensa
  };
})();
```

Ostoskoria käytetään nyt seuraavasti:

```
kauppa.ostoskori.lisaa("keksi");
kauppa.ostoskori.lisaa("keksi");
```

```
kauppa.ostoskori.lisaa("omena");
console.log(kauppa.ostoskori.tuotteidenLukumaara()); // 3
```

10. Asynkroninen ohjelmointi

10.1. Asynkroninen ohjelmointi

Web-sovellusten ja työpöytäsovellusten ohjelmointi eroavat toisistaan paljon. Toisin kuin työpöytäympäristössä, selaimilla on tarjota yksi säie kaikille, jotka tarvitsevat pääsyä käyttöliittymään. Yksittäinen säikeistysmalli rajoittaa käyttöliittymän elementtejä muokkaavaa sovelluskoodia, koska se rajoittaa samalla muun koodin suorittamista. Toisin sanoen funktiot ja säikeet estävät käyttöliittymän käytön, kunnes säie on suoritettu. Tästä syystä on tärkeää hyödyntää kaikkia selaimen tarjoamia asynkronisia toimintoja.

Ohjelman synkroninen suoritus tarkoittaa sitä, että jokainen koodirivi suoritetaan järjestyksessä, eikä seuraavan rivin suoritus voi alkaa ennen kuin aiempi rivi on suoritettu. Funktiokutsuissa ohjelman suoritus odottaa, että funktiosta palataan, jotta voidaan jatkaa suoritusta seuraavalta riviltä. Jos em. funktiossa tehdään aikaa vieviä toimintoja, kuten datan noutamista palvelimelta, pääohjelma odottaa koko tämän ajan ennen kuin jatkaa oman koodin suoritusta.

Asynkronisessa suorituksessa funktiosta palaamista ei jäädä odottamaan, vaan jatketaan välittömästi pääohjelman suoritusta.

JavaScript käyttää ohjelman suorituksessa tapahtumapohjaista mallia, jolla on yksi suoritussäie. Tässä mallissa *event loop* käsittelee tehtäväjonon tehtäviä yksi kerrallaan, luoden jokaiselle suorituksessa olevalle tehtävälle kutsupinon. Vasta kun kutsupino on suoritettu kokonaan, voidaan tehtäväjonosta ottaa uusi tehtävä käsittelyyn.

Kun kutsupinossa suoritetaan asynkronisia toimintoja, ne siirtyvät selaimen API:lle joko ajastettavaksi tai palvelinkutsujen tapauksessa odottamaan vastausta palvelimelta. Sieltä ne siirretään tehtäväjonoon kun kyseinen operaatio on suoritettu. Seuraava esimerkki havainnollistaa tapahtumapohjaista mallia JavaScriptissä.

```
function main() {
  function firstFunction() {
    setTimeout(function cb() {
      secondFunction();
    }, 0}
  }
  function secondFunction() {
    console.log('Im second function');
  }
  firstFunction();
  console.log('Done');
}
main();
```

Kun esimerkin mukaista koodia suoritetaan, menee kutsupinoon ensin viimeisen rivin funktiokutsu `main()`. Tämän jälkeen kutsupinoon menevät funktiokutsu `firstFunction()` ja sen sisältä `setTimeout cb`. Kun `setTimeout`-kutsua suoritetaan, se siirretään asykronisena toimintona selaimen API:lle ajastuksen ajaksi, josta sen callback-funktio `cb` siirtyy tehtäväjonon perälle kun aika on kulunut (tässä tapauksessa annettu aika on 0, joten se siirtyy jonon perälle heti). Tehtäväjonossa seuraavana oleva tehtävä voidaan ottaa käsittelyyn vasta sen jälkeen, kun suorituksessa oleva kutsupino on saatu käsitellyksi.

Kun `setTimeout cb` on poistettu kutsupinosta, voidaan siitä poistaa `firstFunction()`, koska funktio on nyt suoritettu. Seuraavaksi kutsupinoon menee `console.log('Done')`, joka suoritetaan heti. Konsoliin tulostuu `Done` ja se voidaan poistaa kutsupinosta. Funktio `main()` on suoritettu ja sekin voidaan poistaa kutsupinosta.

Suorituksessa ollut kutsupino on suoritettu, joten tehtäväjonosta otetaan seuraava tehtävä suoritukseen. Kutsupinoon tulee siis järjestyksessä `cb()`, `secondFunction()` ja `console.log('Im second Function')`.

Lopuksi suoritetaan `console.log` joka tulostaa konsoliin `Im second function` ja kutsupinosta voidaan poistaa siellä olevat päältäpäin yksi kerrallaan, sillä ne on suoritettu. Kuten esimerkistä voidaan myös päätellä, `setTimeout`-metodi ei siis takaa sitä, että sen callback-funktio suoritettaisiin täsmälleen parametrina annetun ajan päästä, vaan tämä funktio siirretään annetun ajan kuluttua tehtäväjonon perälle.

Callback-funktioksi kutsutaan sellaista funktiota, joka annetaan parametrina toiselle funktiolle. Callback-funktio suoritetaan joko heti ennen kuin funktio palaa tai vasta sen jälkeen kun funktiosta ollaan jo palattu. Jos callback-funktio suoritetaan heti, on kyseessä synkroninen callback, kuten esimerkiksi taulukon `forEach`-metodi missä taulukon jokaiselle alkiolle suoritetaan sama koodilohko. Asynkroninen callback-funktio suoritetaan tulevaisuudessa, kuten esimerkiksi silloin kun HTTP-pyyntö saa vastauksen palvelimelta.

11. ES6 ja ES7

11.1. Ecmascript (ES)

Ecmascript on standardi johon JavaScript perustuu. Ecmascriptin ensimmäinen versio julkaistiin vuonna 1997. ES6 (tunnetaan myös nimellä Ecmascript 2015) julkaistiin vuonna 2015 ja lisäsi paljon hienoja ominaisuuksia JavaScript-kieleen. Nykyään suurten ja monimutkaisten ohjelmien kirjoittaminen on paljon helpompaa. ES6 oli valtava päivitys JavaScriptiin pitkän tauon jälkeen, mutta sen jälkeen on tullut useampia pieniä päivityksiä. Tässä luvussa käsitellään näiden päivitysten mukana tulleista ominaisuuksista tärkeimpiä, kuten luokkien määritelmät ja nuolifunktiot.

11.2. let ja const

JavaScriptissä on myös avainsana `let` muuttujien määrittelemiseksi. `Var` -avainsana on jo kuvattu aiemmin.
Muuttujan `var` näkyvyysalue on funktio, joka tarkoittaa sitä, että muuttujaa voidaan käyttää vain sen funktion sisällä, jossa se oli määritelty.

Esimerkiksi seuraavassa koodissa muuttujia `a` ja `b` voidaan käyttää vain `myFunction`- funktion sisällä.

```
function myFunction() {
  var a = 5;
  var b = a * 10;
}
```

Jos yrität tulostaa muuttujan `a` arvon konsolille kuten alla olevassa koodissa, saat virheilmoituksen.

```
function myFunction() {
  var a = 5;
  var b = a * 10;
}

console.log(a);
```

Alla tulostuva virheilmoitus. Muuttujaa `a` voi käyttää vain `myFunction`- toiminnon sisällä.

```
ReferenceError: a is not defined
```

Jos määrität `var` -muuttujan minkä tahansa funktion ulkopuolella, sitä voidaan käyttää kaikkialla JavaScript-tiedostossasi. Silloin se on niin kutsuttu globaali muuttuja. Se on kuitenkin riskialtista ja voi aiheuttaa odottamattomia virheitä, joten sinun kannattaisi ennemmin käyttää niin kapeaa näkyvyysaluetta kuin mahdollista.

Muuttajan `let` näkyvyysalue on lohkoalue, jolloin muuttujaa voidaan käyttää vain sen lohkon sisällä, jossa se oli määritelty. Seuraavassa koodissa meillä on sama funktio kuin aikaisemmin, mutta nyt muuttujat määritellään let-avainsanalla. Mikään ei itse asiassa muutu, koska muuttujia voidaan käyttää vain funktiolohkon sisällä.

```
function myFunction() {
  let a = 5;
  let b = a * 10;
}
```

Mutta jos meillä on esimerkiksi `if` -ehto toiminnon sisällä ja muuttujat määritellään siellä, voidaan muuttujia käyttää vain `if`- lohkon sisällä. Seuraavassa koodissa muuttujaa `b` voidaan käyttää vain `if`- lohkossa, jossa se määritettiin.

```
function myFunction() {
  let a = 5;
  if (a > 3) {
    let b = a * 10;
  }
  // variable a can be used here
  // variable b can't be used here
}
```

Avainsanalla `let` voi siis määritellä suppeamman näkyvyysalueen kuin käyttämällä `var` avainsanaa. Siksi sinun olisi aina käytettävä ennemmin `let-` kuin `var` -avainsanaa, jos vain mahdollista, koodisi ylläpidon helpottamiseksi.

`Const` avainsanalla voidaan määritellä vakioita. Avainsanan `const` näkyvyysalue on lohko. Seuraavassa esimerkissä määritellään vakio, jota kutsutaan PI: ksi.

```
const PI = 3.14159
```

Vakioarvot ovat muuttumattomia ja saat seuraavan virheen, jos yrität muuttaa niiden arvoja.

```
TypeError: Assignment to constant variable.
```

11.3. Nuolifunktiot

Nuolifunktiot (arrow functions) ovat vaihtoehtoinen tapa määritellä JavaScript-funktioita. Nuolifunktion perusajatus on seuraava: nuolen (`=>`) vasemmalla puolella ovat toimintoparametrit. Nuolen oikealla puolella on funktion lauseke.

```
parameters => expression
```

Jos meillä on seuraava nimetön toiminto,

```
function() {
  return "I am a function";
}
```

nuolifunktiona se voidaan kirjoittaa kuten alla.

```
() => "I am a function"
```

Kuten näet, toiminnon koodi lyhenee paljon. Jos parametreja ei ole, sinun on käytettävä tyhjiä sulkeita `()`, kuten yllä olevassa esimerkissä on esitetty.

Jos sinulla on vain yksi funktion argumentti, et tarvitse sulkeita. Seuraavassa esimerkissä yksi parametri välitetään funktiolle, joka palauttaa parametrin arvon plus kymmenen.

```
x => x + 10
```

Jos parametreja on useita, tarvitaan sulkeet kuten alla olevassa esimerkissä on esitetty.

```
(x, y) => x * y
```

Nuolifunktion lauseke voi sisältää useita rivejä, jolloin sinun on määriteltävä funktion lohko aaltosulkumerkkien `{}` ja `return` avainsanan avulla.

```
(x, y) => {
  let a = 10;
  return x + y + a;
}
```

Kaikki yllä olevat toiminnot ovat nimettömiä toimintoja, joita ei voi kutsua. Näitä käytetään enimmäkseen takaisinkutsutoimintoina. Takaisinkutsutoiminto on toiminto, joka välitetään toiselle toiminnolle parametrina. Käydään läpi yksi käytännön esimerkki. JavaScript-funktio `map()` on todella kätevä taulukon iteroinnissa. `Map()`-funktio hyväksyy takaisinkutsutoiminnon parametrina ja kutsuu välitettyä funktiota kullekin taulukon alkiolle. Seuraava esimerkki kuvaa `map()` -toiminnon käyttöä. Takaisinkutsutoiminto on muotoa `x => x * 2` ja kaikki `arrayA` -taulukon elementit kerrotaan kahdella. Takaisinkutsutoiminto suoritetaan kullekin taulukon arvolle ja se palauttaa uudet arvot tuloksena olevassa taulukossa (`arrayB`).

```
let arrayA = [1, 2, 3, 4, 5];

// arrayB = [2, 4, 6, 8, 10]
let arrayB = arrayA.map(x => x * 2);
```

Tämä voidaan kirjoittaa myös perinteisenä funktiona seuraavalla tavalla.

```
let arrayA = [1, 2, 3, 4, 5];

// arrayB = [2, 4, 6, 8, 10]
let arrayB = arrayA.map(function(x) {
  return x * 2;
});
```

Edellä olevasta voit nähdä, että nuolifunktioilla saat paljon lyhyemmän ja luettavamman koodin.

Voit myös tallentaa nuolifunktion muuttujaan ja kutsua sitä kuten alla olevassa koodissa.

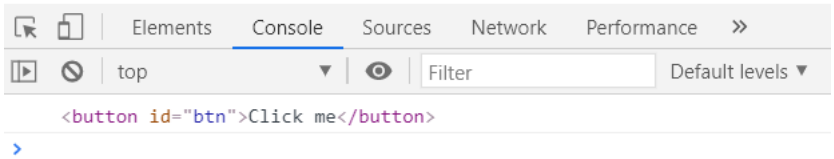
```
const sayHello = name => console.log("Hello " + name)

// Call function
sayHello("John");
```

On tärkeää tietää, että `this` -avainsana käyttäytyy eri tavalla nuolifunktioiden kanssa kuin perinteisten funktioiden. Nuolifunktioissa this -avainsana ei ole sidottu. Perinteisissä funktioissa `this`- avainsana on sidottu objektiin, joka kutsuu funktiota. Seuraava esimerkki osoittaa eron.

Meillä on seuraava HTML-tiedosto, jossa meillä on yksi painike (button). Painike käynnistää `hello`- toiminnon, kun sitä painetaan. Hello-toiminto tulostaa `this` -objektin konsolille. Ensimmäisessä tapauksessa funktio on määritelty käyttämällä perinteistä funktiota.

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="btn">Click me</button>

  <script>
    function hello() {
      console.log(this);
    }

    document.getElementById("btn").addEventListener("click", hello);
  </script>
</body>
</html>
```

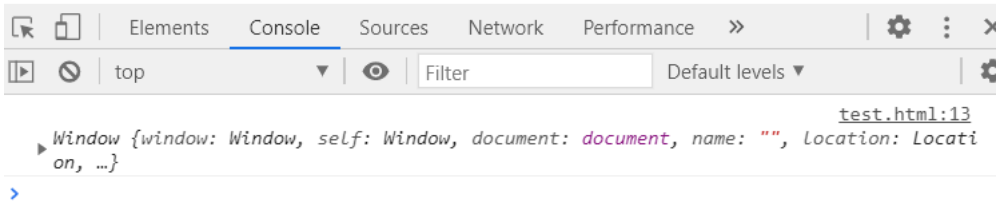
Jos nyt painamme painiketta, voimme nähdä painikkeen konsolissa kuten alla olevassa kuvassa. Joten `this` avainsana on nyt sidottu painikkeeseen (button).



Seuraavaksi muutamme toiminnon nuolifunktioksi, kuten alla olevassa koodissa.

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="btn">Click me</button>

  <script>
    hello = () => {
      console.log(this);
    }

    document.getElementById("btn").addEventListener("click", hello);
  </script>
</body>
</html>
```

Kun nyt painamme painiketta, voimme nähdä, että `this` on sidottu ikkunaobjektiin (Window), kuten alla olevassa kuvassa.



Nuolifunktiot ovat käytännöllisiä takaisinkutsutoiminnoissa, mutta sinun on oltava tietoinen tästä erosta käyttäessäsi niitä muihin tarkoituksiin.

11.4. Funktioparametrit

Funktion parametreilla voi olla oletusarvot, kuten alla olevassa koodissa.

```
function calcSum(x, y = 5) {
  return x + y;
}
```

Jos nyt kutsutaan yllä olevaa funktiota `calcSum(2)`, se palauttaa arvon 7, koska x = 2 ja y = 5. Seuraavaksi, jos kutsumme funktion käyttämällä lausetta `calcSum(2,3)`, se palauttaa arvon 5, koska x = 2 ja y = 3. Jos et välitä arvoa y:lle, funktio käyttää oletusarvoa.

Voit myös käyttää rest-parametria (`...`) funktioparametreissa, jolloin funktio voi hyväksyä äärettömän määrän parametrejä taulukkona. Esimerkiksi seuraava koodi ottaa parametrin nimeltä `params` käyttäen rest-parametria. Funktion rungossa käydään läpi `params`- taulukko ja tulostetaan kaikki arvot konsolille.

```
function myFunction(...params) {
  params.forEach(param => console.log(param));
}
```

Voit nyt kutsua `myFunction` käyttämällä niin monta parametria kuin haluat.

```
myFunction("Hello", "World", "John");
```

Tässä tapauksessa tulos on seuraava.

```
Hello
World
John
```

11.5. Luokat & periytyminen

class-avainsanaa voidaan käyttää ES6:ssa määrittelemään luokka. Käyttö on samankaltainen kuin monissa muissakin olio-ohjelmointikielissä kuten esim. Java. Alla oleva koodi luo luokan, jonka nimi on Shape.

```
class Shape {
  // Class code
}
```

Luokalla täytyy aina olla konstruktorimetodi nimeltä `constructor`. Sitä käytetään, kun luokasta luodaan uusi olio. Luokalla voi olla myös luokan ominaisuuksia ja muita metodeita, kuten alla olevassa esimerkissä.

```
class Shape {
  // luokan konstruktori eli muodostin, jossa x ja y ovat luokan ominaisuuksia
  constructor (x, y) {
    this.x = x;
    this.y = y;
  }

  // Luokkametodit
  move(x, y) {
    this. x = x;
    this.y = y;
  }

  printLocation() {
    console.log(this.x + ", " + this.y);
  }
}
```

Voit luoda luokan objekteja `new`-avainsanalla. Voit kutsua luokan metodeita käyttämällä `object_name.method_name`-merkintää, kuten alla olevassa koodissa.

```
class Shape {
// Constructor where x and y are class properties
  constructor (x, y) {
    this.x = x;
    this.y = y;
  }

  // Luokkametodit
  move(x, y) {
    this. x = x;
    this.y = y;
```

```
  }

  printLocation() {
    console.log(this.x + ", " + this.y);
  }
}

// Luo uusi Shape-olio
let shape = new Shape(0, 0);

// Kutsu luokan metodeja
shape.move(10, 10);
shape.printLocation(); // prints 10, 10 to console
```

Luokan periytyminen voidaan toteuttaa käyttämällä `extends`-avainsanaa. Luodaan ensin yksi luokka, jonka nimi on Person (Henkilö).

```
class Person {
  constructor(first_name, last_name) {
    this.last_name = last_name;
    this.first_name = first_name;
  }

  printName() {
   console.log(first_name + " " + last_name);
  }
}
```

Seuraavaksi luomme luokan nimeltä Student (Opiskelija), joka perii Person-luokan.

```
class Student extends Person {
  constructor(first_name, last_name, student_nr, departement) {
    super(first_name, last_name);
    this.student_nr = student_nr;
    this.department = department;
  }
}
```

Student-luokka (aliluokka) perii kaikki ominaisuudet ja metodit Person-luokasta (superluokka). Opiskelijaluokan konstruktorissa meidän on kutsuttava superluokan konstruktoria `super`-avainsanalla, joka viittaa perittyyn luokkaan. Nyt voit luoda uuden opiskelijaolion, kuten alla olevassa koodissa.

```
let student = new Student("John", "Johnson", S23324, "Computer Science");
```

Käytännössä JavaScriptin luokka, siis luotuna käyttäen class-avainsanaa, ei varsinaisesti eroa lopputuloksesta jonka saa aiemmin käsitellyillä konstruktorifunktiolla ja prototyypillä. Sen voi ajatella olevan vaihtoehtoinen syntaksi JavaScript-olioiden luomiseen mikäli on jo tottunut luokkiin varsinaisissa olio-ohjelmointikielissä.

11.6. Malliliteraalit

Mallipohjan literaaleja eli **malliliteraaleja** (template literals) voidaan käyttää merkkijonojen yhdistämiseen. Muistutuksena, perinteinen tapa olisi käyttää plus-merkkejä, kuten alla olevassa esimerkissä.

```
let first_name = "John";
let last_name = "Johnson";
console.log("Hello " + first_name + " " + last_name);
```

Malliliteraalit voivat sisältää paikkamerkkejä merkkijonon korvaukselle `${ }`, jotka sisältävät Javascript-lausekkeita. Yllä oleva esimerkki voidaan kirjoittaa seuraavalla tavalla. **Huom!** Malliliteraarien kanssa on käytettävä backticks (``) -merkkejä lainausmerkkien sijasta.

```
let first_name = "John";
let last_name = "Johnson";
console.log(`Hello ${first_name} ${last_name}`);
```

Seuraava koodi tulostaa kahden muuttujan summan.

```
let x = 5;
let y = 8;
console.log(`Sum = ${x + y}`);
```

Voit kutsua myös funktioita mallililiteraaleja käyttämällä kuten seuraavassa esimerkissä. Esimerkki tulostaa *Hello World John* -tekstin konsolille.

```
function hello() {
  return "Hello World";
}

console.log(`${hello()} John`)
```

11.7. Potenssioperaattori

Potenssilaskuja varten löytyy Math-kirjastosta metodi Math.pow(kantaluku, eksponentti). JavaScriptiin on myöhemmin lisätty myös operaattori ** tätä varten, minkä pitäisi olla nopeampi ja helpompi käyttää. Kuten muillakin yleisimmistä matemaattisista operaattoreista, potenssioperaattorillekin löytyy sijoittamista varten lyhennetty versio **=.

```
x = 2 ** 2      // 2 potenssiin 2 on 4
x **= 2         // 4 potenssiin 2 on 16
console.log(x)  // 16
```

Ainoa varsinainen ero Math.pow() ja **-operaattorin välillä on, että hyvin vanhat versiot selaimista eivät välttämättä tue kuin vanhaa Math.pow()-muotoa. Potenssioperaattorin käyttö on suositeltavaa, koska se on selkeämpi ja lyhyempi muoto, sekä mahdollistaa edellä mainitun **= -operaation.

11.8. Includes

On jo olemassa metodi, jolla tarkistaa miltä kohdalta taulukkoa tietty arvo löytyy.

```
jokuLista = ["eka arvo", "toka arvo", "kolmas arvo"];
console.log(jokuLista.indexOf("kolmas arvo");  // 2
console.log(jokuLista.indexOf("muu arvo");     // -1
```

.indexOf() palauttaa arvon indeksin, tai jos ei kyseistä arvoa löydy, -1. Myöhemmin on kuitenkin lisätty lisäksi .includes(), mikä palauttaa joko true tai false, riippuen siitä löytyykö haluttua arvoa. Suurta eroa näillä ei ole, koska .indexOf() voidaan yksinkertaisella if-lauseella saada palauttamaan true tai false.

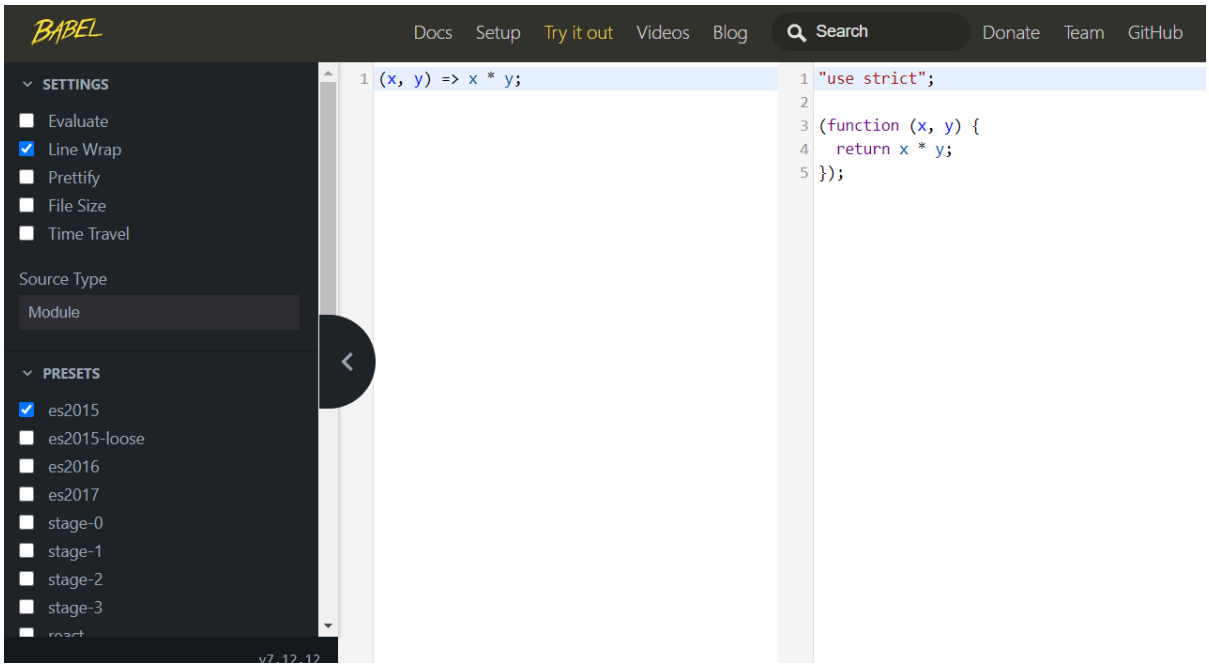
.includes() voi kuitenkin tuntua loogisemmalta käyttää. Lisäksi se osaa tunnistaa listalta NaN-arvon, toisin kuin .indexOf().

```
toinenLista = [1, 2, NaN, 3, 4];
toinenLista.indexOf(NaN);   // -1
toinenLista.includes(NaN);  // true
```

11.9. Babel

Ole varovainen uusimpien ECMAScript-ominaisuuksien kanssa, koska jotkin selaimet eivät välttämättä tue uusimpia ominaisuuksia. Babel (https://babeljs.io/) on JavaScript-kääntäjä, jota voidaan käyttää JavaScriptin kääntämiseksi vanhempaan selainyhteensopivaan versioon.

Voit testata Babelia käytännössä siirtymällä heidän verkkosivustolleen ja valitsemalla päävalikosta 'Kokeile' ('Try it out'). Seuraavassa kuvassa näet, kuinka ES6-nuolifunktio käännetään takaisin perinteiseksi funktioksi.



Voit käyttää Babelia selaimessa tuomalla Babel-kirjaston ja määrittelemällä `script type="text/babel"` kuten seuraavassa esimerkissä. Näin voit käyttää uusimpia ECMAScript-ominaisuuksia, jotka käännetään takaisin vanhempaan Javascript-versioon.

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

<!-- Your custom script here -->
<script type="text/babel">
  const getMessage = () => "Hello World";
</script>
```