1. Algoritmi

Sisällys

- Algoritmin määritelmä.
- Aiheen pariin johdatteleva esimerkki.
- Algoritmista ohjelmaksi.

Algoritmin määritelmä

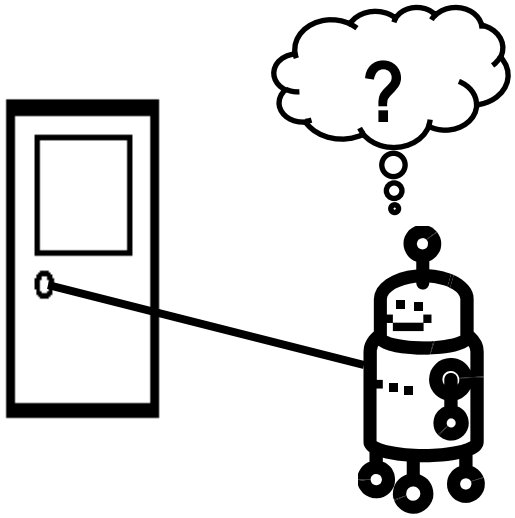
- Ohjelmointi vaatii abstraktia ajattelua, jonka apuvälineinä käytetään algoritmeja (algorithm).
- Algoritmi on vaiheittainen kuvaus jonkin tehtävän suorittamista varten äärellisessä ajassa.
- Algoritmilla on siis alku-, väli- ja loppuvaiheet.
- Mikäli äärellinen aika ei riitä, on kyseessä
  - algoritmin asemasta niin sanottu proseduuri tai
  - algoritmissa on virhe (ikuinen silmukka).

AvaaOvi-algoritmi

- Tehtävänä kertoa erityisen älykkäälle robotille kuinka avata ovi.
- Tehdään aluksi joitakin algoritmin muotoilemista helpottavia oletuksia: ovi on kunnossa, paikallaan, kiinni, lukitsematon, ovessa on kahva eikä kahvaa tarvitse painaa.

AvaaOvi-algoritmi

- Usein ensimmäisessä hahmotelmassa havaitaan virhe (niin sanottu bugi): Edellä ei huomioitu sitä, että jotkut ovet voi avata myös työntämällä.



Algoritmista ohjelmaksi

- Ohjelma on algoritmin konkreettinen toteutus jollakin ohjelmointikielellä.
- Algoritmeja hahmotellaan usein vuokaavioiden tai pseudokoodin avulla ennen varsinaista toteutusta.

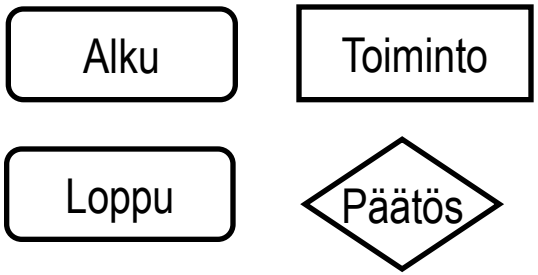
2. Vuokaaviot

Sisällys

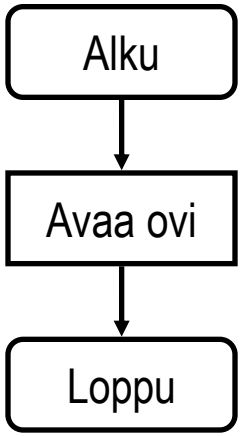
- Kaavioiden rakenne.
- Kaavioiden piirto symboleita yhdistelemällä.
- Kaavion osan toistaminen silmukalla.
- Esimerkkejä:
  - algoritmi oven avaamiseen vuokaaviona,
  - keskiarvon laskeminen ja
  - peli luvun arvaukseen.

Vuokaaviot

- Graafinen kieli algoritmien kuvaamiseen.
  - Ymmärrettäviä ja intuitiivisia.
  - Soveltuvat monimutkaistenkin algoritmien esittämiseen.
- Muodostetaan yhdistelemällä symboleja nuolilla.
- Symbolissa algoritmin vaihe.



- Kaavio suoritetaan (ajetaan) seuraamalla nuolia alkusymbolista alkaen ja loppusymboliin päätyen.
- Esimerkki:



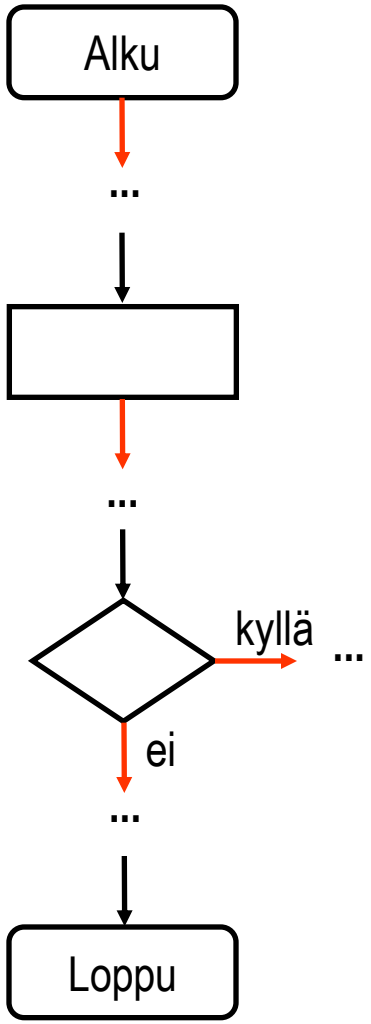
Yksinkertaisin kaavio koostuu alku- ja loppusymboleista sekä yhdestä toiminnosta.

Vuokaaviot

- Etenevät yleensä ylhäältä alas ja vasemmalta oikealle:
  - Suunta länsimaisesta kirjoituksesta.
  - Tilan loppuessa voi piirtää muutenkin.
- Aina yksi alku- ja yksi loppusymboli.
- Symboleista lähtevien nuolien lukumäärä on yksikäsitteinen.
- Symboleihin tulevien nuolien lukumäärässä tulkinnan varaa.

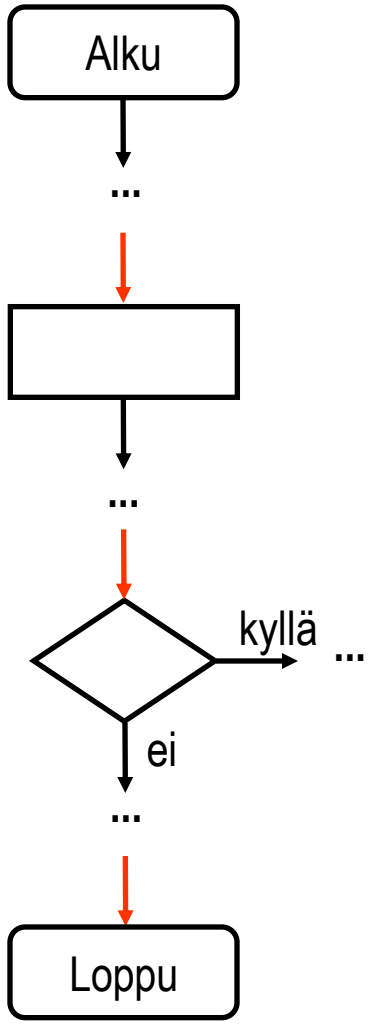
Lähtevät nuolet

- Alkusymbolista lähtee aina vain yksi nuoli.
- Loppusymbolista ei lähde nuolia.
- Toimintosymbolista lähtee aina vain yksi nuoli.
- Päätössymbolista lähtee aina kaksi nuolta, jotka vastaavat kyllä- ja ei-päätöksiä.



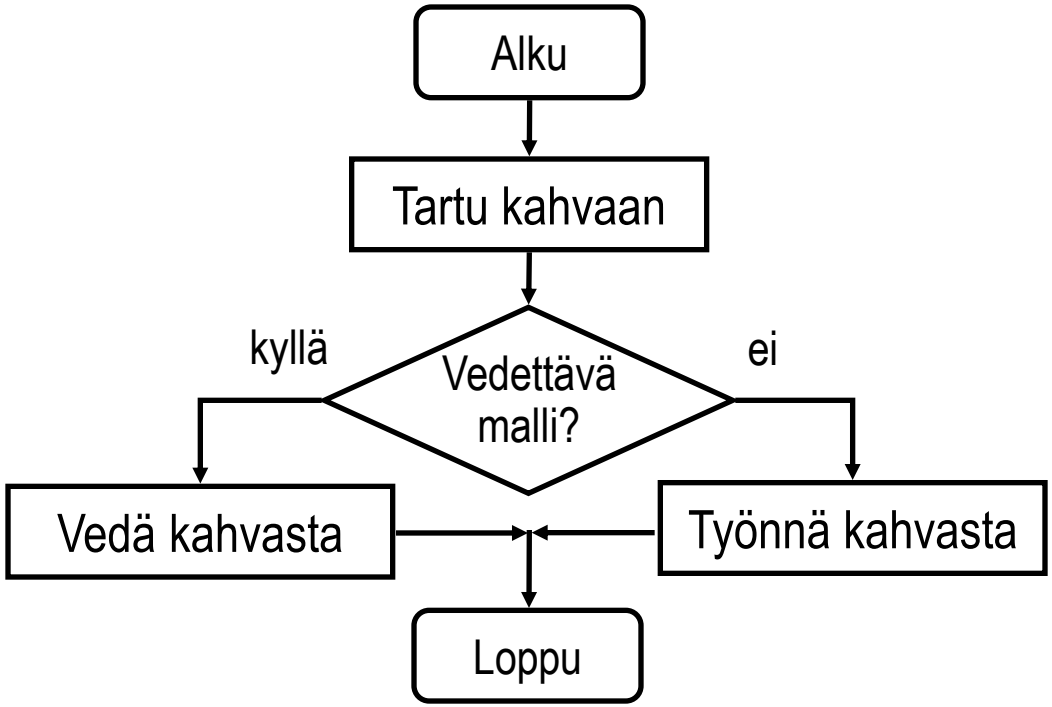
Tulevat nuolet

- Alkusymboliin ei tule nuolia.
- Muihin symboleihin tulee aina joko yksi tai useampi nuoli.
- Jos symboliin tulee useampi nuoli, voidaan
  - nuolet piirtää suoraan kiinni symboliin tai
  - symboliin piirtää yksi nuoli, johon muut nuolet liittyvät.
- Kalvoilla ja mallivastauksissa pyritään käyttämään selvyyden vuoksi jälkimmäistä piirtotapaa, jolloin tulevia nuolia on aina yksi.



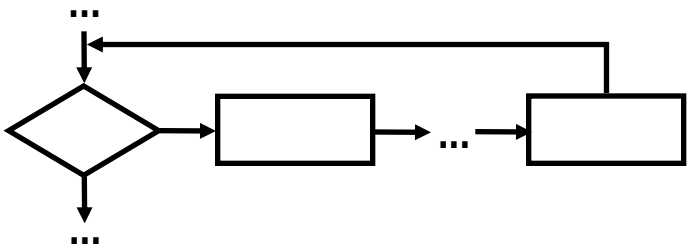
Oven avaaminen (versio 1)

- Esitetään AvaaOvi-algoritmi vuokaavioina.
- Ensimmäisessä versiossa algoritmin vaiheet kuvataan peräkkäisissä vaiheissa vapaa-muotoisena tekstinä.
- Toisessa versiossa avaaminen on kuvattu silmukkaa käyttäen.

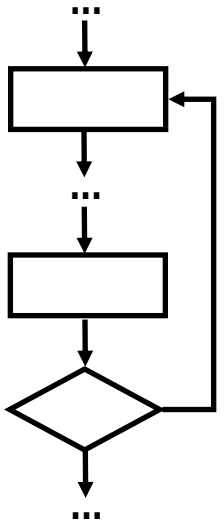


Silmukka

- Usein on tarpeen suorittaa uudelleen vuokaavion osa. Tämä onnistuu silmukan avulla.
- Koostuu päätöksestä, joka yhdistetään nuolella toistettavaan vuokaavion osaan, josta piirretään nuoli takaisin päätökseen.
- Päätös sijoitetaan usein siten, että se on silmukan ensimmäiseksi suoritettava osa (esiehto).



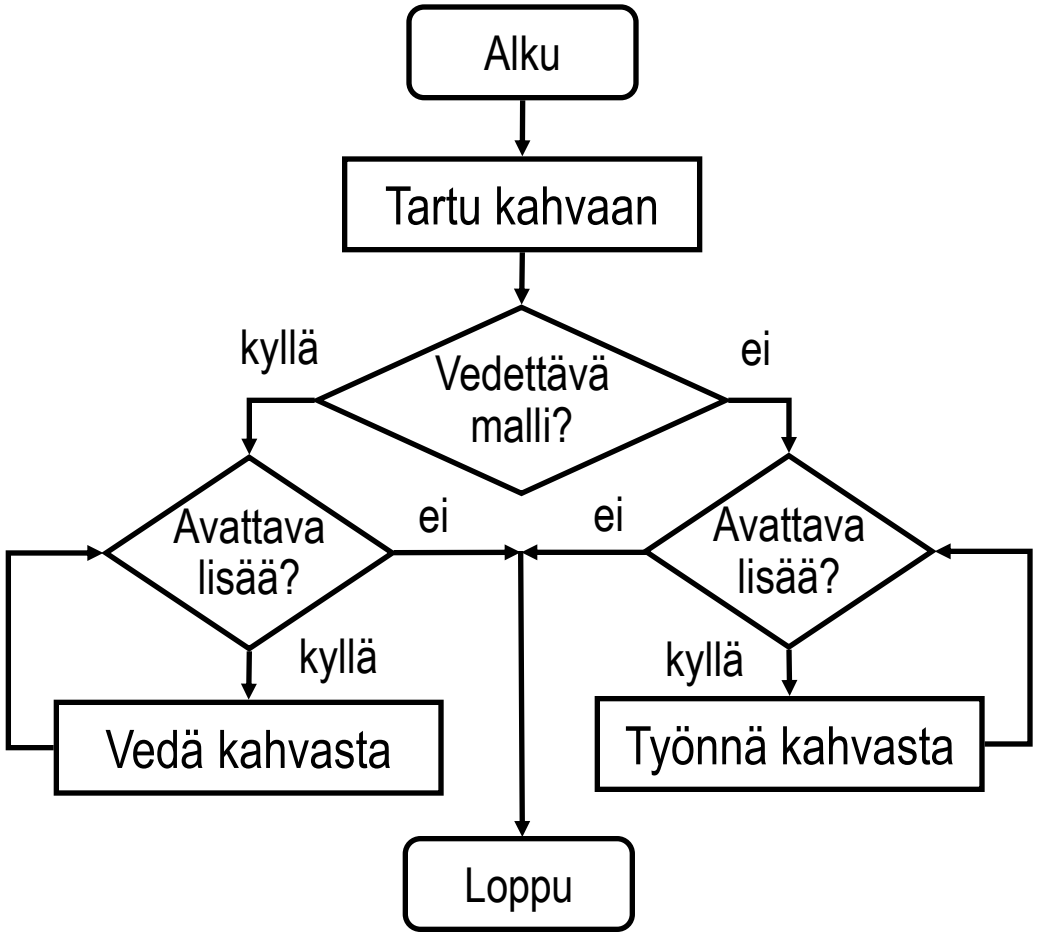
- Toisinaan on luontevampaa sijoittaa päätös silmukan loppuun (jälkiehto).



- Silmukointi jatkuu niin kauan kuin päätös on silmukkaan johtavaan nuolen suuntainen.
- Jos päätös on muotoiltu virheellisesti, algoritmi saattaa joutua ikuiseen silmukkaan.

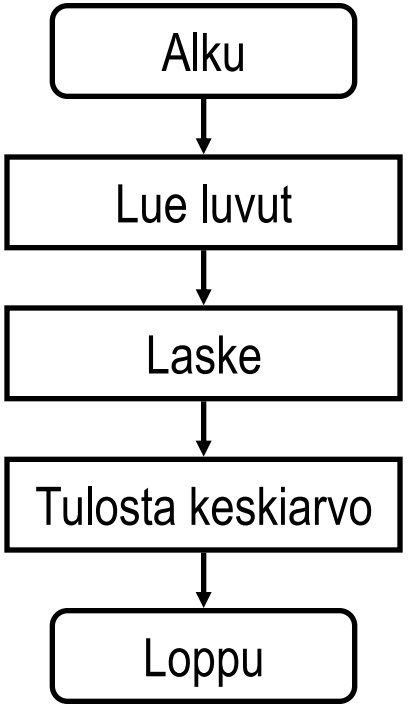
Oven avaaminen (versio 2)

- Algoritmin toiseen versioon on lisätty silmukat, joissa ovea joko vedetään tai työnnetään kahvasta kunnes ovi on auki.

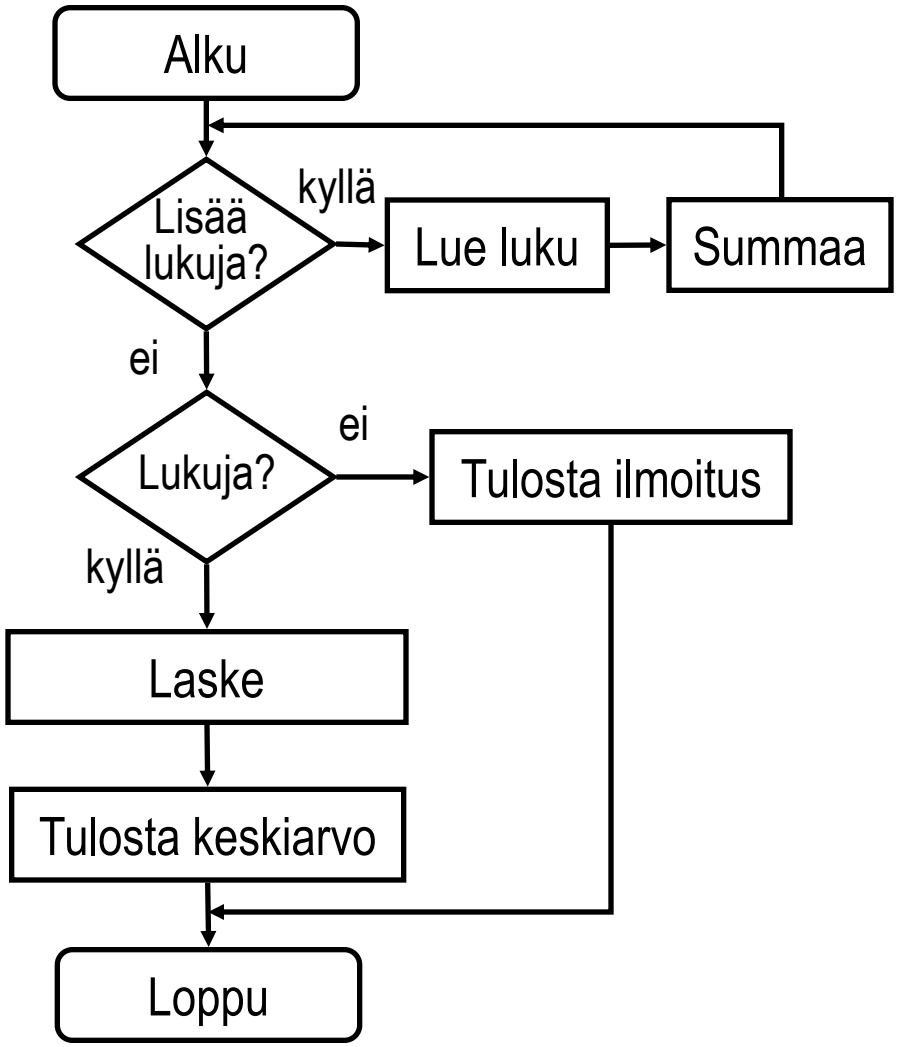


Keskiarvon laskeminen (versio 1)

- Ohjelma kommunikoi käyttäjänsä kanssa: luvut luetaan yksi kerrallaan ennen keskiarvon laskemista ja keskiarvo tulostetaan käyttäjälle.
- Oletetaan, että käyttäjä antaa aina luvun sitä pyydettäessä.
- Tarkennettavaa:
  - Lukujen lukemiseen ja summan laskemiseen tarvitaan silmukka.
  - Voi olla, että lukuja ei anneta. Tähän täytyy varautua.



Keskiarvon laskeminen (versio 2)

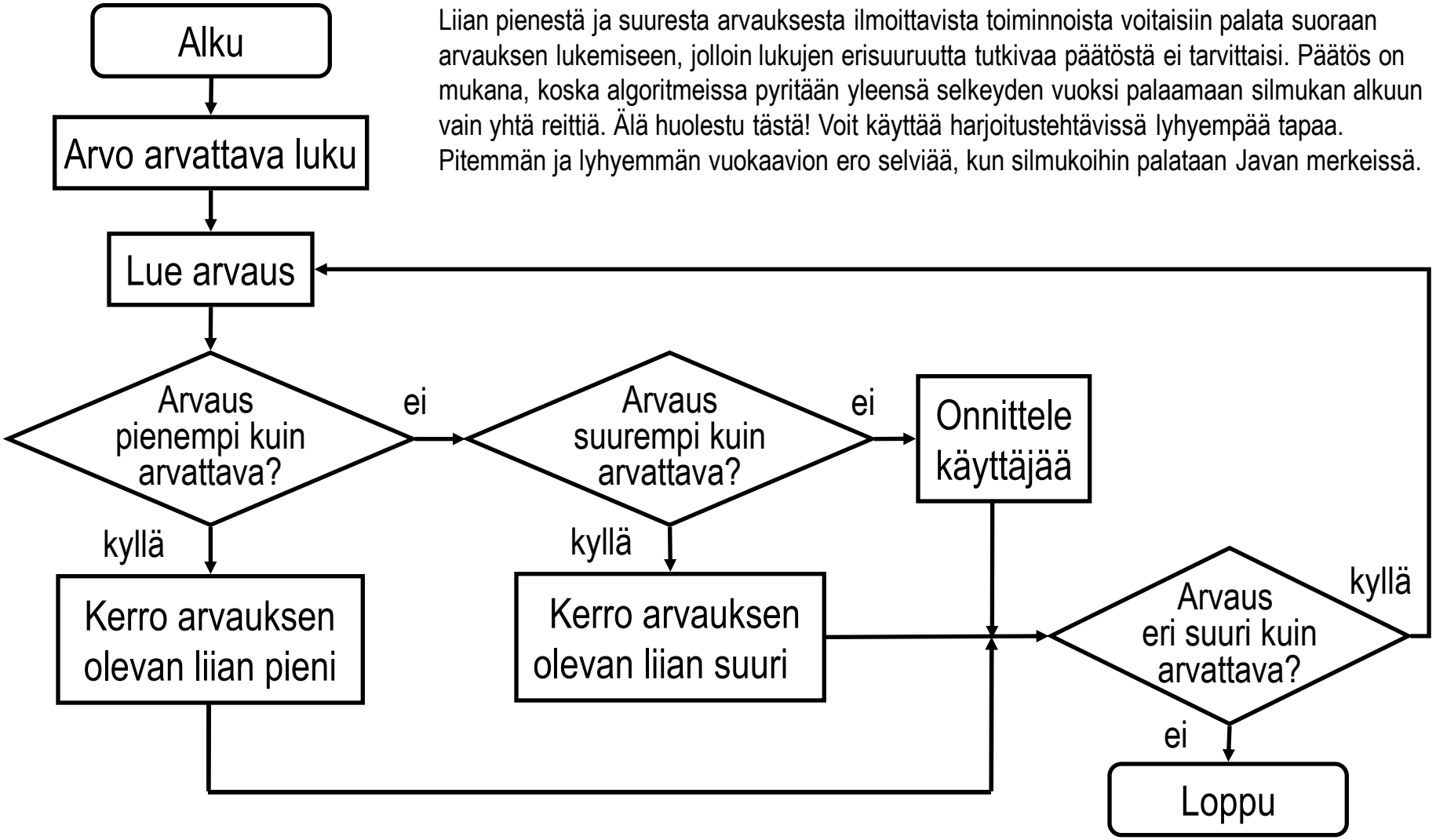


- Kaaviosta ei käy vielä ilmi laskemisen logiikka kuten keskiarvon kaavasta.
- Erityisesti pitää pohtia:
  - kuinka silmukka toteutetaan ja
  - kuinka Laske-vaiheen sisältö määritellään tarkemmin?

Peli luvun arvaukseen

- Peli arpoo kokonaisluvun käyttäjän tuntemalta väliltä. Käyttäjän tehtävänä on arvata lukua kunnes hän osuu oikeaan. Väärin menneen arvauksen osalta käyttäjälle kerrotaan oliko luku liian pieni tai suuri suhteessa arvattavaan lukuun. Oikeasta arvauksesta onnitellaan.
- Algoritmiin tarvitaan silmukka, koska on hyvin epätodennäköistä, että käyttäjä arvaa luvun heti.
  - Jälkiehto on luonteva, koska käyttäjä arvaa aina vähintään kerran.
  - Silmukassa luetaan arvaus ja tehdään päätöksiä, joiden avulla kerrotaan oliko arvaus mahdollisesti “ali”, “yli” tai oikein.
- Arpominen on tehtävä on ennen silmukkaa, jotta luvun arvaaminen on mahdollista vihjeiden avulla.

Peli luvun arvaukseen



Liian pienestä ja suuresta arvauksesta ilmoittavista toiminnoista voitaisiin palata suoraan arvauksen lukemiseen, jolloin lukujen erisuuruutta tutkivaa päätöstä ei tarvittaisi. Päätös on mukana, koska algoritmeissa pyritään yleensä selkeyden vuoksi palaamaan silmukan alkuun vain yhtä reittiä. Älä huolestu tästä! Voit käyttää harjoitustehtävissä lyhyempää tapaa. Pitemmän ja lyhyemmän vuokaavion ero selviää, kun silmukoihin palataan Javan merkeissä.

Pohdintaa

- Vuokaavioiden ongelmia:
  - Graafinen esitys poikkeaa paljon useimmista ohjelmointikielistä.
  - Algoritmin tarkentaminen kasvattaa kaaviota nopeasti.
  - Kaavioiden piirtäminen on työlästä.
- Vapaamuotoisesti tekstillä kuvaillut algoritmin vaiheet ovat liian monikäsitteisiä tietokoneelle.
- Kuinka algoritmin esitys voidaan tarkentaa tietokoneen ymmärtämälle tasolle?

3. Muuttujat ja operaatiot

Sisällys

- Muuttujat.
  - Nimi ja arvo.
  - Algoritmin tila.
  - Muuttujan nimeäminen.
  - Muuttujan tyyppi.
  - Muuttuja ja tietokone.
- Operaattorit.
  - Operandit.
  - Arvon sijoitus muuttujaan.
  - Aritmeetiikka.
  - Arvojen vertailu.
- Operaatiot.
  - Nimeäminen.
  - Parametrit.
  - Paluuarvo.
- Esimerkkejä:
  - algoritmi oven avaamiseen vuokaaviona,
  - keskiarvon laskeminen ja
  - peli luvun arvaukseen.

Muuttujat ja operaatiot

- Tietokoneet tarvitsevat vapaamuotoista tekstiä paljon yksityiskohtaisempia ohjeita.
  - Tietokone osaa vain laskea hyvin nopeasti. Arkijärkeä koneella ei ole.
- Algoritmia muuttujien ja operaatioiden avulla tarkentamalla päästään lähemmäs tietokonetta.
- Esimerkiksi AvaaOvi-algoritmi voidaan jakaa karkeasti toimintoihin ja toimintojen kohteisiin.
  - Vaiheessa “Tartu kahvaan” on toiminto “tartu” ja kohde “kahva”.

Muuttujat ja operaatiot

- Tarkemmissa algoritmeissa (ja ohjelmissa) kohteet esitetään muuttujina ja toiminnot operaatioina.
  - Myös kohteen osat (esim. oven kahva) ja ominaisuudet (esim. oven väri) voidaan ajatella muuttujiksi.
  - Algoritmin epämääräisemmät osat, kuten “ovi on vedettävää mallia”, voidaan ymmärtää tilanteen mukaan joko muuttujiksi tai toiminnoiksi.
- Ajattelussa siirrytään näin abstraktimmalle tasolle: algoritmissa käsitellään muuttujien sisältämiä tietoja operaatioita suorittamalla.

Muuttujat

- Muuttujalla on nimi (tunnus) ja nimeen liittyvä arvo.
- Muuttuja ei ole matematiikasta tuttu käsite, jolla symboloidaan usein tuntematonta arvoa.
  - Esimerkiksi yhtälössä $x + 1 = 0$ tuntematon arvo on x.
- Algoritmissa muuttujan arvo on yleensä tunnettu ja erityisesti erona on se, että arvo voi muuttua algoritmin edetessä vaiheesta toiseen.
  - Muuttuja muistuttaa hieman suuretta eli mitattavaa ominaisuutta. Esimerkiksi lämpötila (T) on suure, jolla mitataan aineen lämpöenergian määrää.

Muuttujat

- Usein vain yhden muuttujan arvo muuttuu, kun algoritmi etenee vaiheesta toiseen.
  - Kaikkien muuttujien arvot voivat säilyä myös entisellään.
- Algoritmin tila tarkoittaa sen muuttujien arvoja tietyssä algoritmin vaiheessa.
  - Esimerkiksi kurssin ilmoittautumistietoja käsittelevässä algoritmissa voisi olla muuttujat *etunimi*, *sukunimi* ja *opiskelijanumero* ja näillä arvot "Mikko", "Meikäläinen" ja 12345, kun Mikko Meikäläinen ilmoittautuu kurssille. Tiina Terävän ilmoittautuessa algoritmin tila muuttuu, koska muuttujien aikaisemmat arvot korvautuvat Tiinan arvoilla.

Muuttujat

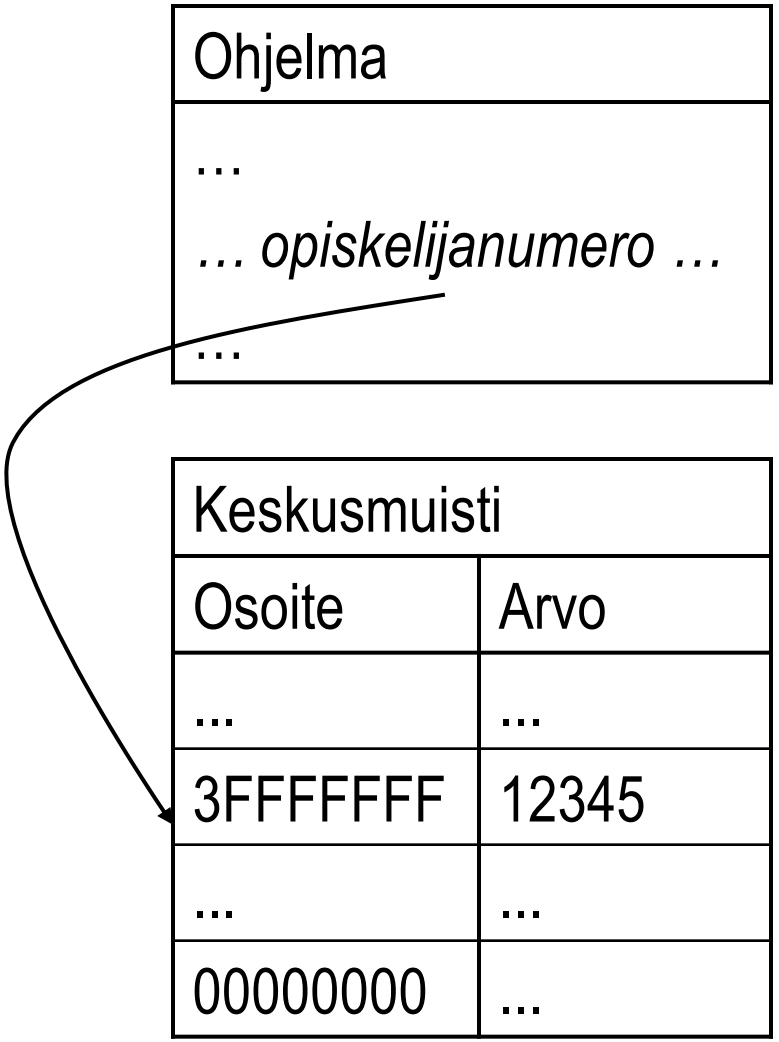
- Muuttuja nimetään sen arvoa kuvaavasti.
  - Erityisesti pitkien algoritmien ymmärrettävyyden kannalta on erittäin tärkeää, että nimistä nähdään minkä tiedon säilyttämiseen muuttujia käytetään.
- Yleensä hyvä nimi on riittävän pitkä.
  - Esimerkiksi syntymäpäivän sisältävälle muuttujalle on parempi antaa nimeksi vaikkapa *syntpvm* kuin *p*.
  - Nimeä lyhyesti vain, jos erehtymisen vaaraa ei ole.
- Nimissä käytetään enimmäkseen kirjaimia.
  - Kirjainten seurana voi olla numeroita.

Muuttujat

- Muuttujille määritellään lausekielissä yleensä tyyppi: muun muassa kokonais- ja liukuluvut sekä yksittäiset merkit ja merkkijonot ovat käytettävissä.
  - Esimerkiksi henkilön paino ja pituus on järkevää esittää lukuina, kun taas etu- ja sukunimi ovat luontevasti merkkijonoja.
- Muuttujalle ei voi antaa mittayksikköä.
  - Henkilön pituus esitetään muuttujana ilman metrin tunnusta (m), vaikka pituussuureen yksiköksi on sovittu metri.
- Muuttujat oletetaan alkeistyyppisiksi ja niiden tyyppejä ei toistaiseksi määritellä, jotta algoritmien kirjoittaminen ei olisi kurssin alkuvaiheessa liian vaikeaa.

Muuttujat

- Ohjelman suorituksen aikana muuttujien arvot tallennetaan johonkin paikkaan tietokoneen keskusmuistissa.
- Operaattorit ja operaatiot voivat lukea arvon ja muuttaa sitä muistipaikan osoitteen avulla.



Operaatiot

- Muuttujien arvojen käsittelyyn.
  - Operaatio saa muuttujan arvon käyttöönsä yksilöimällä arvon muuttujan nimen avulla.
- Perusoperaatioita kutsutaan usein operaattoreiksi ja niille on annettu omat symbolit.
- Operaattorin käsittelemät arvot ovat operandeja.
- Perusoperaatioiden lisäksi usein on käytettävissä monimutkaisempia operaatioita esimerkiksi arvojen tulostamiseen.

Sijoitusoperaattori

- Perusoperaatioista tärkein on sijoitusoperaattori (←), joka korvaa muuttujan vanhan arvon uudella.
- Yleisesti:

  muuttujanNimi ← arvo

  missä muuttujan nimi (1. operandi) on aina operaattorin vasemmalla puolella ja uusi arvo (2. operandi) operaattorin oikealla puolella.
  - Sijoitus ymmärretään yleensä algoritmin vaiheeksi.
- Uusi arvo voi olla arvo sellaisenaan, muuttujan arvo tai vaikkapa laskutoimituksen tulos.
- Esimerkki ilmoittautumis-algoritmista, jossa muuttujiin sijoitetaan arvoja sellaisenaan:

  etunimi ← "Tiina"

  sukunimi ← "Terävä"

  opiskelijanumero ← 54321

Sijoitusoperaattori

- Muuttujien arvot ovat samat, mutta säilyvät erillisinä, kun muuttujan arvo sijoitetaan toisen muuttujan arvoksi.
- Esimerkki: sijoituksia aikaa ja matkaa käsittelevässä algoritmissa:

matka ← 100

aika ← 2

matka2 ← *matka* (*matka* = 100, *matka2* = 100)

matka ← 30 (*matka* = 30, *matka2* = 100)

- 3. sijoituksen jälkeen muuttujilla *matka* ja muuttujalla *matka2* on edelleen oma erillinen arvo, vaikka molempien muuttujien arvo on 100.
- Koska arvot ovat erilliset, muuttujan *matka2* arvo ei muutu, kun muuttujalle *matka* annetaan uusi arvo 4. sijoituksessa.

Sijoitusoperaattori

- Esimerkki: sijoituksia elokuva-arvosteluja hallinnoivassa algoritmissa:

  nimi ← "Aliens" (Merkkijono suljetaan lainausmerkkeihin.)

  julkaisuvuosi ← 1986

  luokka ← 'A' (Yksittäinen merkki suljetaan suoriin yksinkertaisiin lainausmerkkeihin eli niin sanottuihin hipsuihin.)

Aritmeettiset operaattorit

- Myös tutut aritmeettiset operaattorit (**+** , **-** , **·** , **/**) ovat käytettävissä.
- Aritmeettisen operaattorin operandit ovat arvoja ja tulos on arvo, joka voidaan sijoittaa muuttujaan.
- Laskujärjestys koulusta tuttu.
- Sijoitusoperaattori on heikko: esimerkiksi aritmetiikka suoritetaan ennen sijoitusta.

- Esimerkki: aritmetiikkaa pelkillä arvoilla:

  tulosumma ← 1 · 2 + 2 · 2

  missä ensin lasketaan tulot 1 · 2 ja 2 · 2, toiseksi summataan tulot 2 + 4 ja lopuksi sijoitetaan summa muuttujan arvoksi.
  - Sijoitus muodostaa algoritmin vaiheen, vaikka ennen sijoittamista lasketaan aritmeettisilla operaatioilla.

Aritmeettiset operaattorit

- Esimerkki: nopeuden laskeminen muuttujien avulla:

  nopeus ← *matka* / *aika*

  Otetaan aluksi käyttöön muuttujien arvot:

  30 / 2

  Tämän jälkeen lasketaan osamäärä:

  15

  ja lopuksi sijoitetaan osamäärä muuttujan arvoksi:

  nopeus ← 15

- Sama muuttuja voi esiintyä myös sijoitusoperaattorin molemmin puolin.
- Esimerkki: lisätään aikaa:

  aika ← *aika* + 1

  jolloin selvitetään muuttujan nykyinen arvo:

  2 + 1

  lasketaan summa:

  3

  ja sijoitetaan se uudeksi arvoksi:

  aika ← 3

Vertailuoperaattorit

- Algoritmeissa tarvitaan usein myös vertailuoperaattoreita (**<**, **=**, **>**, **≤**, **≥**, **≠**).
- Oletetaan, että vertailu palauttaa kyllä- tai ei-arvon.
  - Nämä voidaan ajatella totuusarvoiksi tosi ja epätosi.
- Esimerkkejä:

luku1 ← 10

luku2 ← 13 - 3

samat ← *luku1* = *luku2* (*samat* = kyllä)

pienempi ← *luku1* < 1 (*pienempi* = ei)

Operaatiot

- Perusoperaatioiden lisäksi voidaan olettaa, että on käytettävissä myös muita operaatioita.
- Näille ei ole varattu symbolia – muut operaatiot tunnistetaan muuttujien tapaan nimen avulla.
- Erotetaan muuttujista nimen perään kirjoitettavan kaarisulkuparin (**()**) avulla.
- Esimerkiksi nimi *kiinni* on muuttujan tunnus, mutta nimi **kiinni**() tarkoittaa operaatiota, jolla voitaisiin vaikkapa tutkia onko ovi kiinni.
- Operaation nimi on yleensä pitempi kuin muuttujan nimi.
  - Näin **onkoKiinni**() olisi paremmin nimetty kuin **kiinni**().

Operaatiot

- Operaatio voi saada parametrina yhden tai useamman arvon.
- Parametriarvot välitetään operaatiolle kirjoittamalla ne sulkujen väliin.
  - Parametriarvot erotetaan toisistaan pilkulla.
- Operaatio voi myös palauttaa niin sanotun paluuarvon, joka sijoitetaan usein muuttujan arvoksi.
- Esimerkkejä:

tartu(*kahva*)

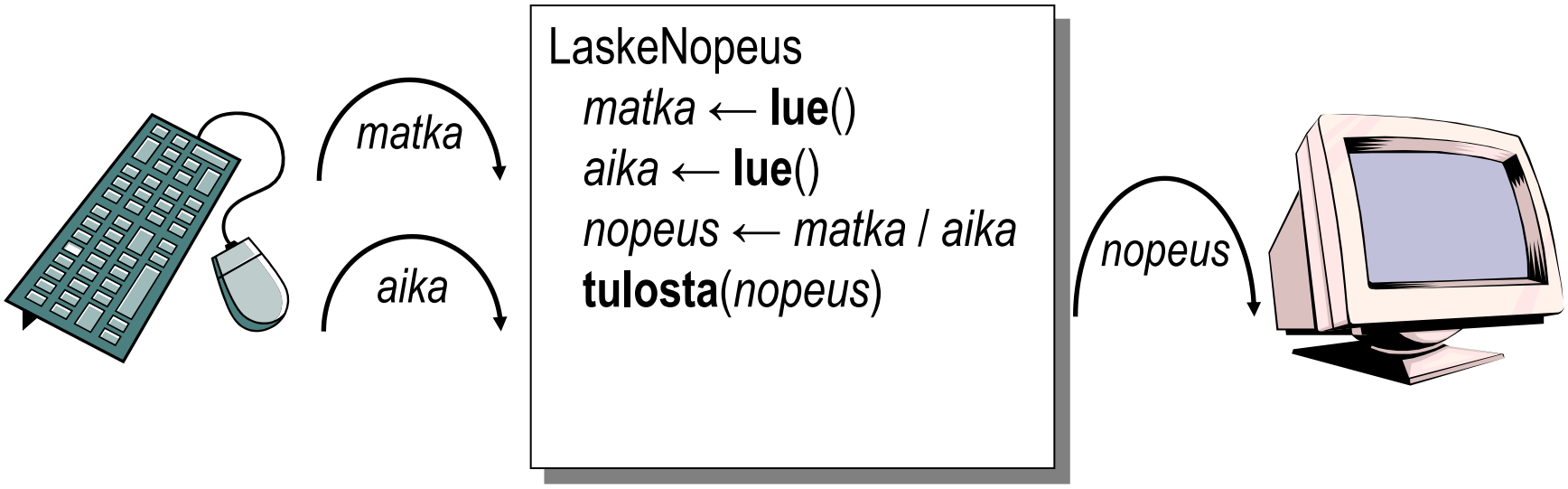
onkoVedettäväMalli ← **vedettävä**(*ovi*)

arvattava ← **arvoLuku**()

min ← **min**(400, *paino*)

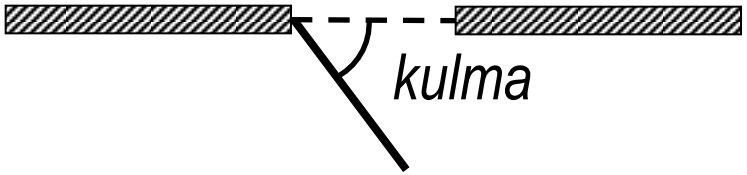
Operaatiot

- Sopivat myös kommunikointiin ympäristön kanssa. Algoritmin laatija voisi käyttää esimerkiksi **lue**() ja **tulosta**(*arvo*) -operaatioita tietojen lukemiseen näppäimistöltä ja tulostamiseen näytölle.
- Esimerkki:

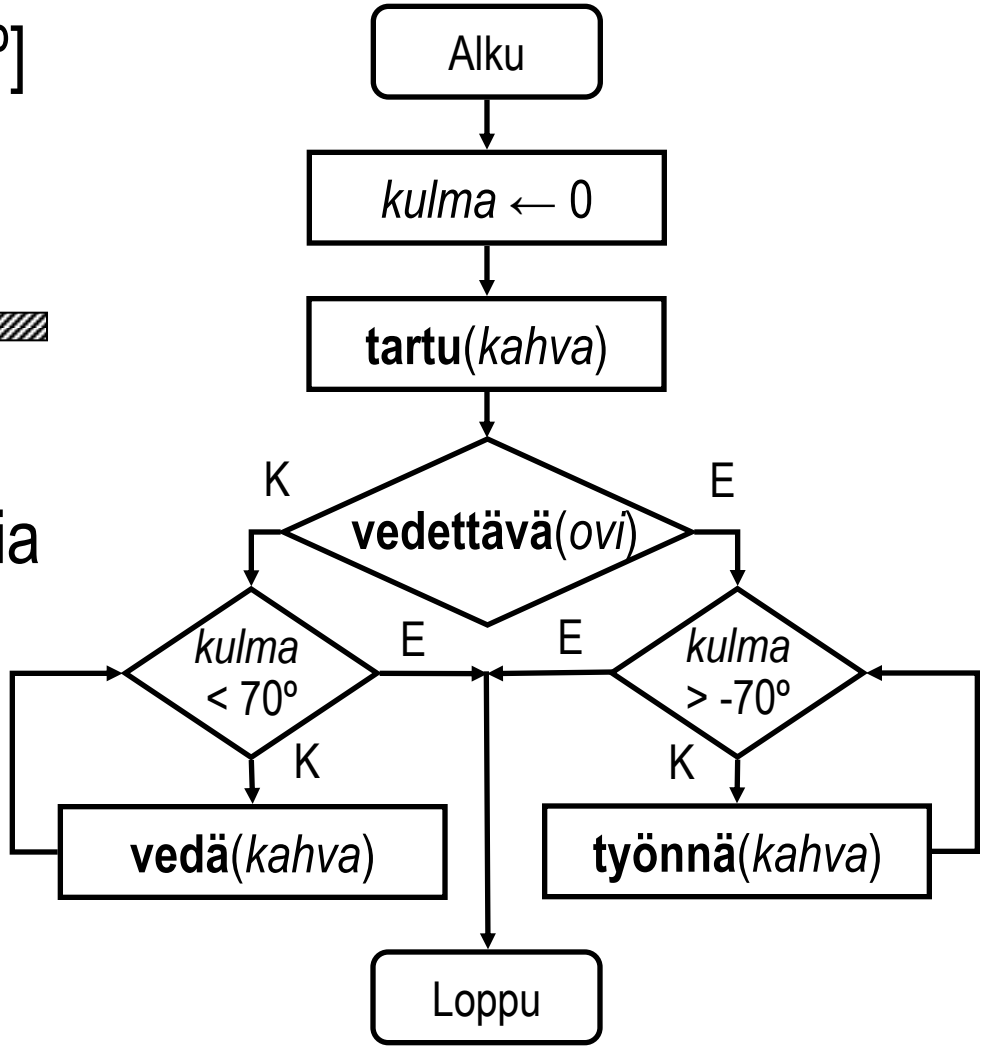


Oven avaaminen (versio 3)

- Muuttuja *kulma* $\in$ [-90º,90º] on oven ja seinän välinen kulma:



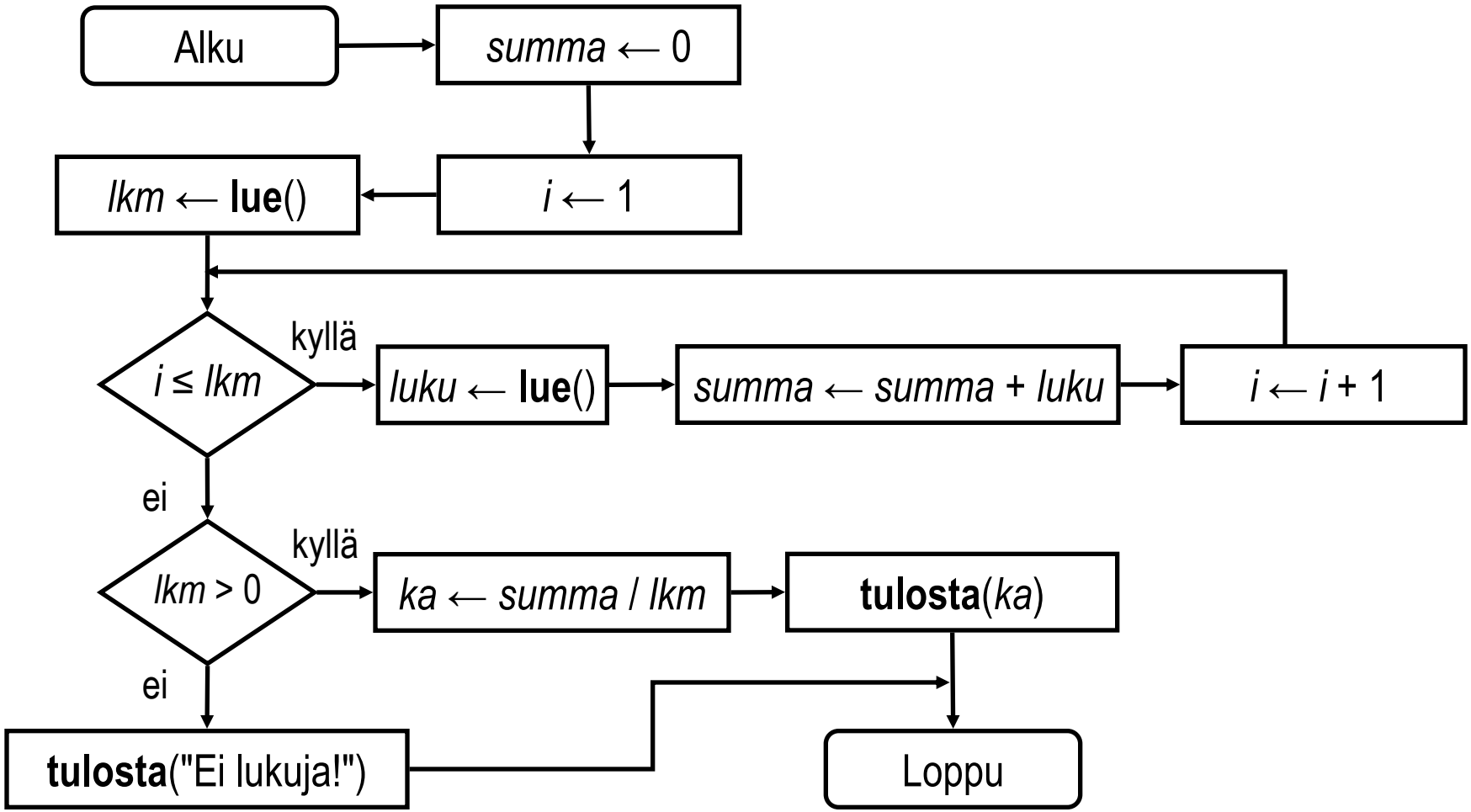
- Lisäksi käytetään muuttujia *ovi* ja *kahva*, joita käsitellään **tartu-**, **vedettävä-**, **vedä-** ja **työnnä-**operaatioilla.



Keskiarvon laskeminen (versio 3)

- Päätellään tarvittavat muuttujat kaavasta $\bar{x} = \sum_{i=1}^{n} x_i / n \ \ (n > 0)$.
  - *luku* on luettu luku (termi x_i),
  - *i* on laskuri, josta selviää monesko silmukan kierros on meneillään (termin indeksi),
  - *summa* vastaa lukujen summaa (Σx_i),
  - *lkm* vastaa lukujen lukumäärää (indeksin yläraja *n*) ja
  - *ka* on lukujen keskiarvo ($\bar{x}$).
- Silmukka voidaan toteuttaa usealla eri tavalla. Oletetaan nyt, että lukujen lukumäärä voidaan selvittää ennen silmukan aloitusta.

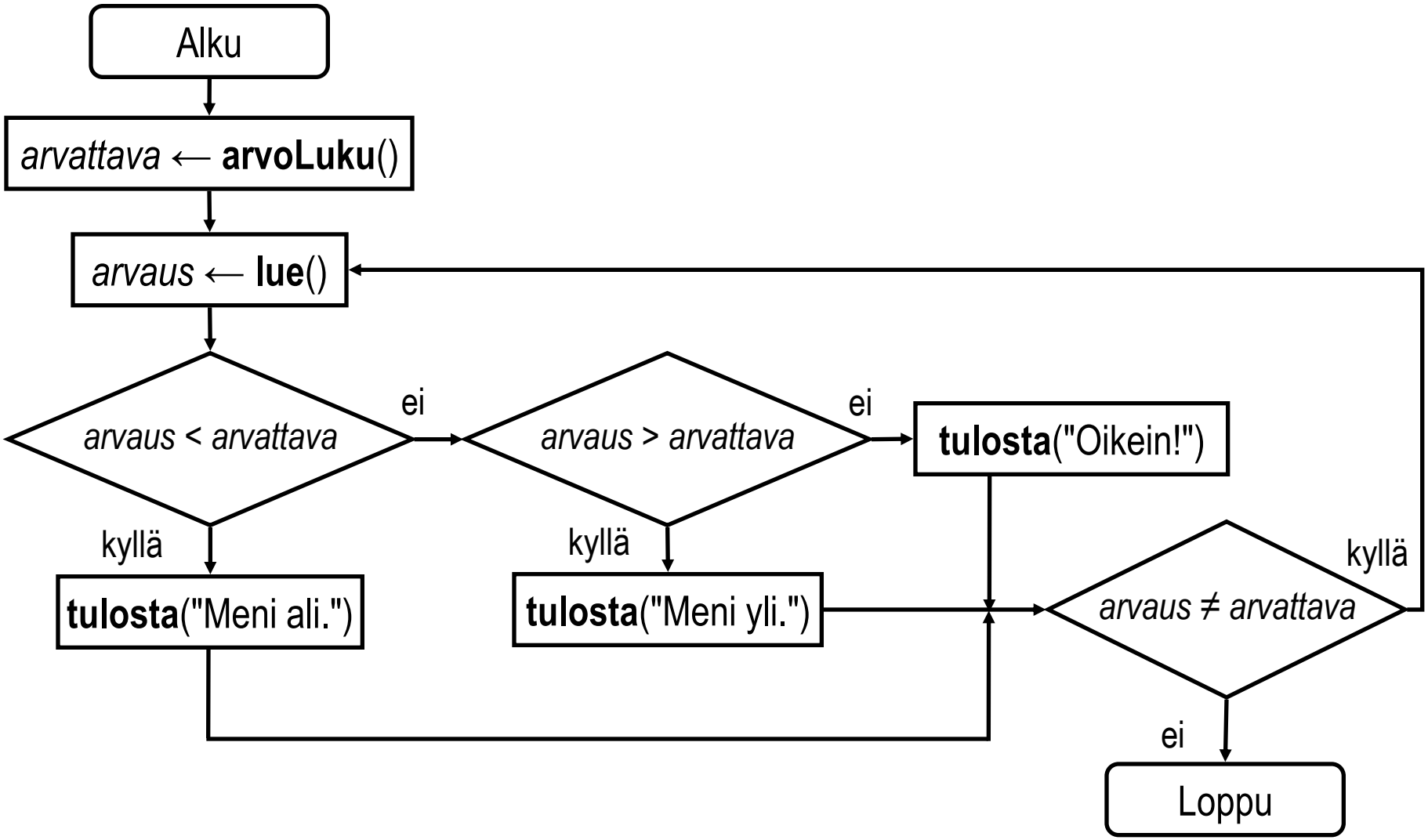
Keskiarvon laskeminen (versio 3)



Peli luvun arvaukseen (versio 2)

- Pelin vapaamuotoinen versio voidaan muuttaa melko suoraviivaisesti tarkemmalle tasolle.
  - Otetaan käyttöön muuttujat arvattavalle ja arvatulle luvulle.
    - Arvattava luku alustetaan satunnaisluvun tuottavalla operaatiolla **arvoLuku**.
    - Arvaus luetaan käyttäjältä **lue**-operaatiolla.
  - Lausutaan päätökset vertailuoperaattoreiden avulla.
  - Vinkit ja onnittelut viestitään **tulosta**-operaatiolla.

Peli luvun arvaukseen (versio 2)



4. Lausekielinen ohjelmointi

Sisällys

- Konekieli, symbolinen konekieli ja lausekieli.
- Hyvä ohjelmointitapa.
- Lausekielestä konekieleksi:
    - Lähdekoodi, tekstitiedosto ja tekstieditorit.
    - Kääntäminen ja tulkinta.
    - Kääntäminen, tulkinta ja Java.

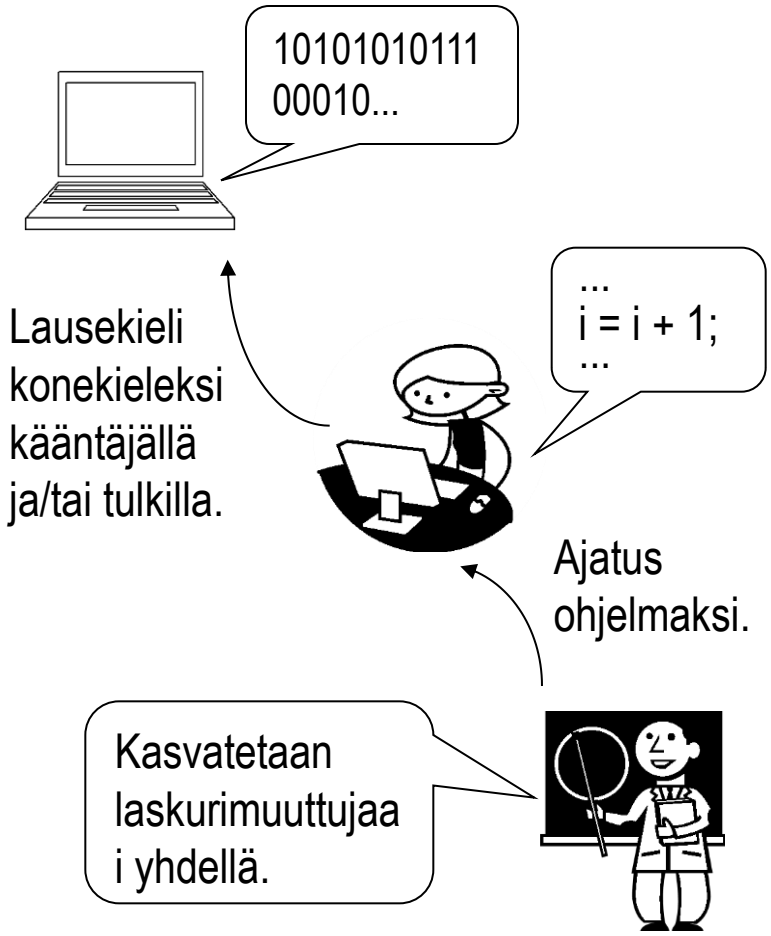
Konekieli ja symbolinen konekieli

- Aluksi ohjelmoitiin tietokoneiden sellaisenaan ymmärtämällä konekielellä (machine language), jossa suorittimen komennot ja niiden parametrit esitetään binäärilukuina.
- Symbolisessa konekielessä (assembly language) komennoille on annettu lyhyet nimet.
- Erästä suoritinta voitaisiin käskeä sijoittamaan akkuun (suorittimen rekisteri) luku yksi seuraavasti:
  - 10101001 00000001 tai
  - LDA #$01.
- Tällaisten kielten käyttö on vaikeaa ja eikä ohjelmia voi ajaa toisen tyyppisellä suorittimella.

Lausekieli

- Abstraktimpi kieli, hieman luonnollisen kielen kaltainen.
- Luonnollisella kielellä ohjelmointi vaikeaa kieliopin moni-mutkaisuuden sekä lauseiden ja sanojen moniselitteisyyden vuoksi.
- Lausekielinen ohjelmointi on mahdollista rajatun kieliopin ja sanaston sekä kääntäjä- ja tulkkiohjelmien ansiosta.
- Ohjelmat pitkälti suoritinriippu-mattomia ja hyvin tehtyinä siirrettä-vissä käyttöjärjestelmien välillä.

- Kielimuurin yli lausekielellä:



Lausekieli

- Sanasto koostuu pääosin tunnuksista (identifier),
  - esimerkiksi muuttujille ja operaatioille annettuja nimiä

  ja varattuista sanoista (reserved word),
  - joita käytetään muun muassa muuttujien tyyppien esittämiseen.
    - Valittu usein englannin kielestä (esimerkiksi **int** ja **double**).
    - Ei voi käyttää tunnuksina.
- Algoritmin vaiheet esitetään lauseina (statement).
  - Lause vastaa vuokaavion toimintoa.
  - Päätöksen ja silmukan kuvaavat lauseet aloitetaan tietyillä varatuilla sanoilla (esimerksi **if** ja **while**).
  - Päätökseen tai silmukkaan liittyvät lauseet kootaan yhteen kieliopillisella merkinnällä ja ohjelman ulkoasua muotoilemalla.

Lausekieli

- Kirjoitetun luonnollisen kielen yksiköitä:
  - Virke sisältää yhden tai useamman lauseen. Alkaa isolla alkukirjaimella ja päättyy isoon välimerkkiin.
  - Lause ilmaisee ajatuksen. Rakentuu verbin ympärille.
  - Lauseke muodostuu lauseen läheisesti yhteenkuuluvista sanoista.
- Lauseohjelmointikielissä virkkeet ovat lauseen mittaisia ja välimerkki on usein puolipiste (;).
- Ohjelmointikielissä lausekkeet (expression) ovat lauseiden arvon tuottavia osia.
- Esimerkiksi Java-kielen lause *i = i + 1;* kasvattaa laskurin arvoa yhdellä. Yhteenlasku *i + 1* on lauseke.

Hyvä ohjelmointitapa

- Ohjelman tulisi olla helposti ymmärrettävä, koska ohjelmia ei tehdä vain omaan käyttöön.
- Lausekielisen ohjelman toimintaa voi selkeyttää pienellä vaivalla käyttämällä kuvaavia nimiä, kommentoimalla ja koodin rivejä sisentämällä.
- Kommentit ovat luonnollisella kielellä kirjoitettuja, ohjelman toimintaa selventäviä lauseita, joita ei suoriteta.
- Usein kahdella kauttamerkillä (//) alkava rivi on kommentti.
- Kommentti liittyy yleensä seuraavaan lauseeseen. Esimerkki:

```
// Tulostetaan merkkijono näytölle.
System.out.println("Javan tulostusoperaation nimi on pitkä...");
```

Hyvä ohjelmointitapa

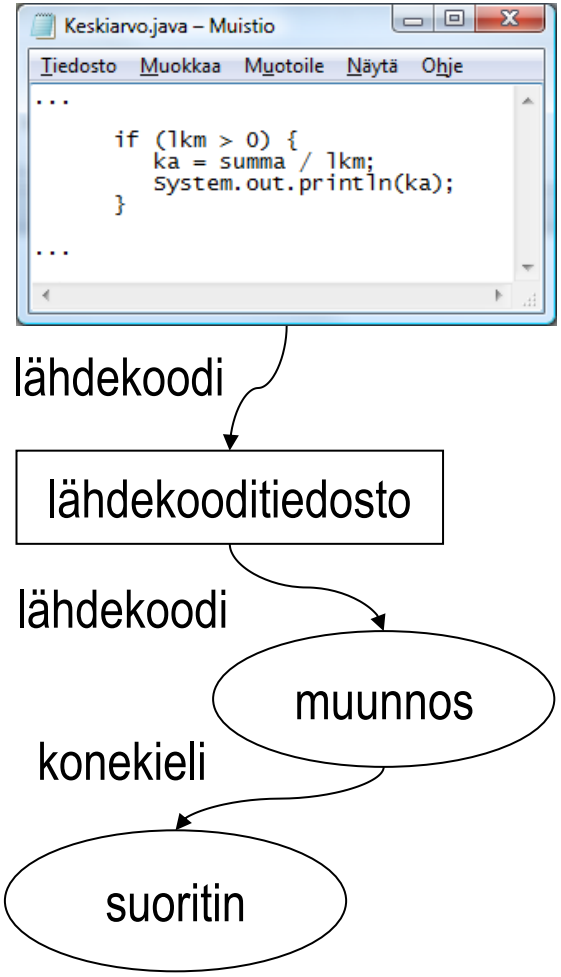
- Ohjelman lukijan tulee nähdä vaivatta mitkä lauseet liittyvät päätökseen tai silmukkaan.
  - Ulkoasun muotoilu sisentämällä on vakiintunut tapa avustaa lukijaa.
- Sisentäminen tarkoittaa muutaman välilyönnin sijoittamista koottujen lauseiden rivien alkuun siten, että rivit alkavat samalta tasalta.
  - Rivejä voidaan sisentää tabulaattorilla, mutta tällöin sisennyksen syvyyden määrää tabulaattorille asetettu pituus.
  - Välilyöntejä ja tabulaattoreja ei saa käyttää sekaisin!
- Lauseiden kieliopillinen yhteenkuuluvuus ilmaistaan usein aaltosuljeparilla ({ }).
- Esimerkki:

```
if (lkm > 0) {
  // Tämän kootun lauseen
  // sisällä jokaisen rivin
  // alussa on kolme välilyöntiä.
  ka = summa / lkm;
  System.out.println(ka);
}
```

- Kommenteissa voi säätellä, sisennystä ei saa koskaan jättää tekemättä.

Lausekielestä konekieleksi

- Aluksi algoritmi kirjoitetaan eli “koodataan” lausekielinen ohjelmaksi eli lähdekoodiksi (source code).
- Kirjoitustyö tapahtuu yksinkertaisella tekstinkäsittelyohjelmalla eli tekstieditorilla.
- Lähdekoodi tallennetaan tiedostoon.
- Lähdekoodi on tietokoneelle liian monimutkaista. Koodi on siksi muutettava apuohjelmalla tietokoneen ymmärtämään muotoon, konekielelle.
  - Kieliraja voidaan ylittää luonnollisen kielen tapaan lähdekoodia konekieleksi kääntämällä (compile) ja tulkitsemalla (interpret).



Lähdekoodi ja tekstitiedosto

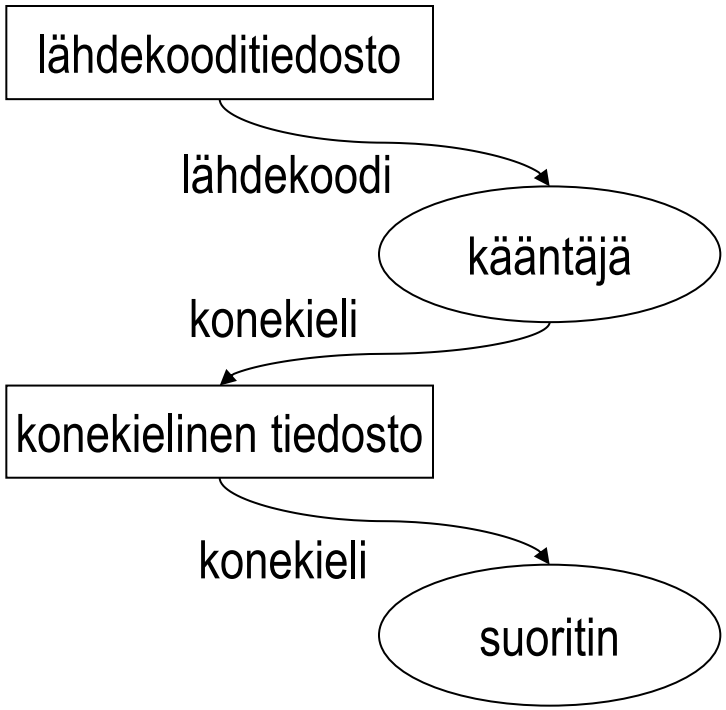
- Lähdekoodi, samoin kuin lähdekoodin sisältävä tekstitiedosto (text file), koostuu kirjaimista, numeroista, välimerkeistä ja näkymättömistä ohjausmerkeistä (muun muassa rivinvaihto- ja tabulaattori).
  - Lähdekoodiin ei voi sijoittaa kursivoinnin tai lihavoinnin tapaisia muotoiluja tai kuvia.
- Käyttöjärjestelmien merkistöt ovat erilaisia, jolloin järjestelmästä toiseen siirretyn lähdekooditiedoston
  - merkkejä saattaa näkyä eri merkkeinä ja
  - muunnos konekieleksi ei aina onnistu.
- Joskus lähdekoodin siirrettävyyttä parannetaan käyttämällä tunnuksissa vain alkuperäistä ASCII-merkistöä, joka on osa useampia muita merkistöjä.
  - Näin toimien å-, ö-, ä-, Å-, Ä- ja Ö-merkkejä voi käyttää vain kommenteissa.

Tekstieditorit

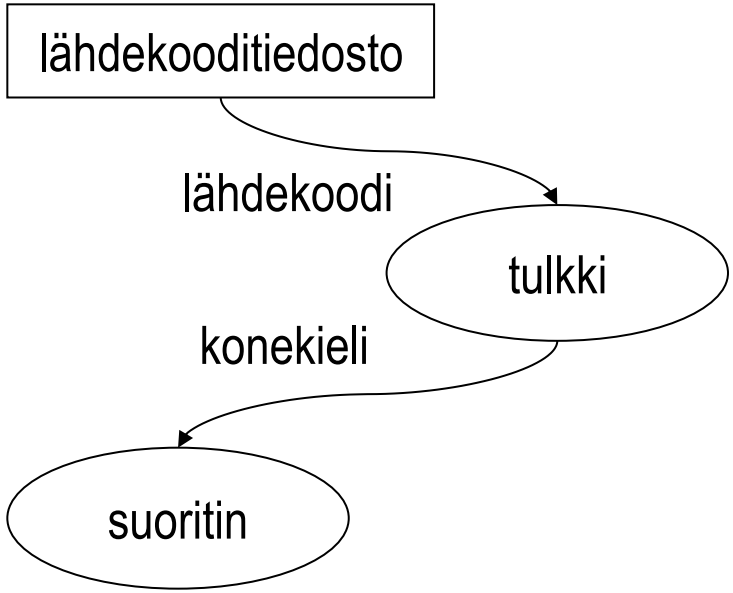
- Kurssilla riittää hyvin yksinkertainen editori.
  - Tärkeintä on, että editori osaa kertoa monennellako rivillä editorin kursori on, jolloin koodissa olevien virheiden korjaaminen on helpompaa.
  - Esimerkiksi Notepad eli Muistio (Windows) tai nano (UNIX).
- Kurssin kotisivuilla linkkejä parempiin editoreihin ja ohjeita Notepad++:n (Windows) ja TextWranglerin (Mac) käyttöön.
- Haasteita kaipaavat käyttävät Emacs-editoria.
- Laajempien ohjelmien tekoon löytyy ohjelmointi-ympäristöjä (esimerkiksi Eclipse ja NetBeans).
- Microsoft Word -ohjelma ei ole tarkoitettu ohjelmointiin vaan monipuoliseen tekstinkäsittelyyn.
  - Ohjelmia ei kannata edes yrittää tehdä Wordillä!

Kääntäminen ja tulkinta

- Kääntäjäohjelma muuttaa lähdekoodin suoraan konekieliseksi tiedostoksi eli suoritettavaksi ohjelmaksi.



- Tulkittavan kielen lähdekoodi suoritetaan erillisellä ohjelmalla eli tulkilla, joka muuttaa ajonaikaisesti lähdekoodin konekieleksi.

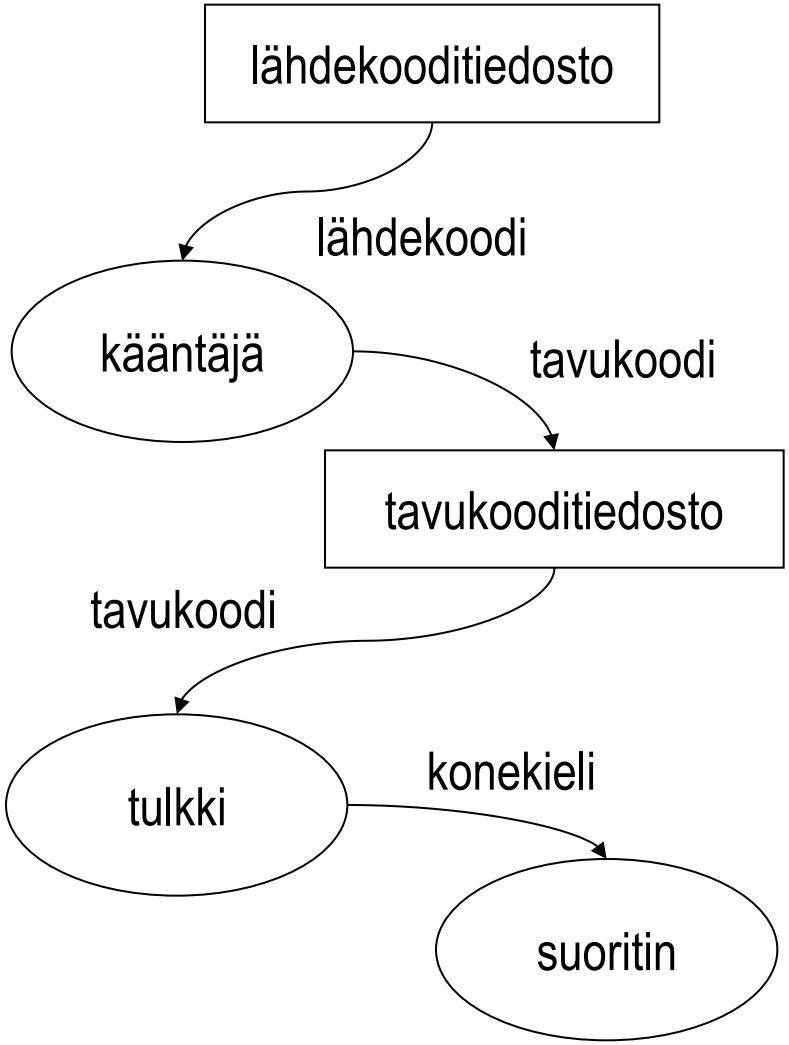


Kääntäminen ja tulkinta

- Ohjelman oltava kieliopillisesti täysin oikein, jotta tulkkaus tai kääntäminen onnistuisi:
  - Kielioppivirheistä ilmoitetaan yleensä virheen tyyppi ja virheen aiheuttaneen rivin numero.
- Käännetty koodi on yleensä tulkattavaa koodia nopeampaa, mutta tulkattavalla kielellä on helpompi tehdä pieniä testejä.
- Käännettäviä lausekieliä ovat esimerkiksi Pascal, C ja C++ ja tulkattavia kieliä muun muassa Basicin jotkut versiot.

Kääntäminen, tulkinta ja Java

- Javassa lähdekoodi käännetään ensin tavukoodiksi (bytecode).
- Tavukoodi suoritetaan sitten tulkin avulla virtuaali-koneessa (virtual machine).
- Tavukoodin ansiosta Java-sovellukset ovat siirrettäviä.
  - Sama tavukoodi voidaan suorittaa kaikkialla missä on tarjolla virtuaalikone (esim. Windows, Linux ja Mac).



5. HelloWorld-ohjelma

Sisällys

- Lähdekoodi.
- Lähdekoodin (osittainen) analyysi.
- Lähdekoodi tekstitiedostoon.
- Lähdekoodin kääntäminen tavukoodiksi.
- Tavukoodin suorittaminen.
- Virheiden korjaaminen

Lähdekoodi

```
/*
 * Laki-kurssin ensimmäinen Java-ohjelma.
 * Jorma Laurikkala, Tampereen yliopisto.
 */
public class HelloWorld {
   public static void main(String[] args) {
      // Tulostetaan näytölle Hello World -teksti.
      System.out.println("Hello World!");
   }
}
```

Lähdekoodin analyysi

- Java aito on oliopohjainen kieli $\Rightarrow$ suoritettava lähdekoodi sijoitetaan aina luokkaan (class).
- HelloWorld-luokan määrittely:

  public class HelloWorld { … }
- Luokka koostuu otsikosta ja kootusta lauseesta, jonka sisällä ovat luokkaan liittyvä osuu ohjelmasta.
- Ongelma: Ohjelmoinnin perusideoita voi oppia ilman olioajattelua!
- Ratkaisu: Jätetään luokkien ja olioiden käsittely myöhemmäksi ja hyväksytään, että koodissa on jonkin verran “magiaa”.

Lähdekoodin analyysi

- Suoritettavassa Java-luokassa on **main**-operaatio (pääohjelma), joka määritellään aina samalla tavalla:

  public static void main(String[] args) { … }

- Myös operaatioilla on otsikko ja runko.
  - Rungon sisään kootaan operaatioon kuuluvat lauseet.
- Lykätään operaationkin analyysi myöhemmäksi ja kirjoitetaan toistaiseksi koodi pääohjelman sisään.
- Ohjelman varsinainen toiminnallisuus on lauseessa

  System.out.println("Hello World!");

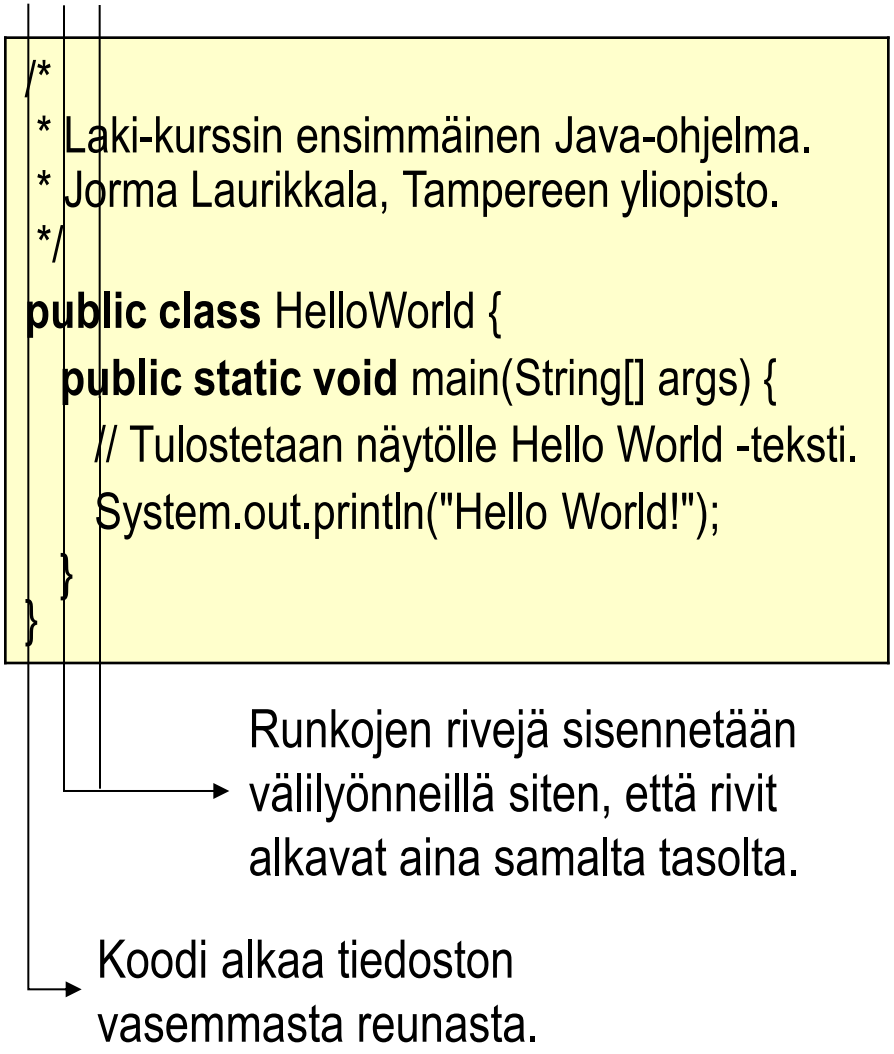
  joka tulostaa näytölle tekstin *Hello World*.

Lähdekoodin analyysi

- Java-kielen **System.out.println**-operaatio vastaa vuokaavioissa käytettyä **tulosta**-operaatiota.
- Kahdella kauttamerkillä (//) alkavat rivit ovat myös Javassa kommentteja.
- Laajempia kommentteja (niin sanotut lohkokommentit) on sujuvampaa kirjoittaa aloittamalla kommentti kauttamerkillä ja asteriskilla (/*) ja lopettamalla asteriskilla ja kauttamerkillä (*/).
- /*…*/ -tyylisiä kommentteja ei saa laittaa sisäkkäin.

Lähdekoodin analyysi

```
/*
 * Laki-kurssin ensimmäinen Java-ohjelma.
 * Jorma Laurikkala, Tampereen yliopisto.
 */
public class HelloWorld {
    public static void main(String[] args) {
        // Tulostetaan näytölle Hello World -teksti.
        System.out.println("Hello World!");
    }
}
```



- Lähdekoodin alussa kerrotaan kommentilla mitä ohjelma tekee.
- Ohjelman ja **main**-operaation rungot suljetaan aaltosulkeiden sisään kootuksi lauseeksi.
- Rungot sisennetään aina, jotta ohjelman osat erottuvat toisistaan.
  - Jokainen sisennyksen taso on saman syvyinen. (Ohessa on käytetty kolmea välilyöntiä.)

Lähdekoodi tekstitiedostoon

- Kirjoitetaan HelloWorld-ohjelman lähdekoodi esimerkiksi Notepad-editorilla ja tallennetaan koodi *HelloWorld.java*-nimiseen tiedostoon.
- Java-lähdekoodia sisältävä tiedosto:
  - Nimetään ohjelman (eli luokan) nimen mukaan.
  - Tunnistetaan *java*-päätteen avulla.
- Huomaa isot alkukirjaimet sekä ohjelman että tiedoston nimessä: Java-kielessä isot ja pienet kirjaimet eivät ole sama asia!

Lähdekoodin kääntäminen tavukoodiksi

- Avataan komentotulkki (command prompt). Kurssin kotisivulla tarkempia tietoja komentotulkin käytöstä.
- Siirrytään **cd**-komennolla hakemistoon, jossa lähdekooditiedosto sijaitsee.
- Kirjoitetaan komentotulkissa **javac HelloWorld.java** ja painetaan Enter-näppäintä.
- Mikäli kääntäminen onnistui, hakemistoon on ilmestynyt tavukooditiedosto *HelloWorld.class*
- *class*-tiedostopääte on varattu tavukoodille.

Lähdekoodin kääntäminen tavukoodiksi

- Kielioppivirhe tuottaa enemmän tai vähemmän selkeän virheilmoituksen. Tutki tarkkaan ilmoitettu rivi. Jos virhe ei ole rivillä, tarkista koko koodi.
  - Muista aina tallentaa korjattu koodi; kääntäjä lukee lähdekoodin tiedostosta, ei editorista.
- On myös mahdollista, että kääntäminen ei onnistu vaikka koodi on kirjoitettu oikein!
- Tällöin on usein kyse puutteellisista ympäristöasetuksista. Tarkempia tietoja löytyy kurssisivuilta.

Tavukoodin suorittaminen

- Kirjoita komentotulkissa **java HelloWorld** ja paina Enter-näppäintä.
- Komento on annettava hakemistossa, jossa tavukoodi-tiedosto sijaitsee.
- Tulkille annetaan ohjelman nimi *HelloWorld* tavukooditiedoston nimen asemasta. Jos tulkille tarjoaa *class*-päätteistä tiedostoa, saa luultavasti ilmoituksen
  - *Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld/class tai*
  - *Error: Could not find or load main class HelloWorld.class*.

Tavukoodin suorittaminen

- Virheelliset ympäristöasetukset voivat estää myös Java-tulkin käytön.
- **javac**- ja **java**-ohjelmat löytyvät Oraclen (aiemmin Sunin) Java Development Kitistä (JDK)
  - Ilmaisohjelmisto – saatavilla Oraclen sivuilta.
    - Usein tietokoneilla valmiiksi asennettuna.
  - Kurssilla tarvitaan Javan versio 1.7.0 tai uudempi.
  - Katso kurssisivujen Ohjelmointivälineitä-kohta, jossa muun muassa linkki JDK-asennukseen ja ohjeita.

HelloWorld-kertaus

HelloWorld.java - Notepad

File Edit Format View Help

```
/*
 * Laki-kurssin ensimmäinen Java-ohjelma.
 * Jorma Laurikkala, Tampereen yliopisto.
 */
public class HelloWorld {
   public static void main(String[] args) {
      // Tulostetaan näytölle Hello World -teksti.
      System.out.println("Hello World!");
   }
}
```

Ln 1, Col 1

Shortcut to CMD.EXE

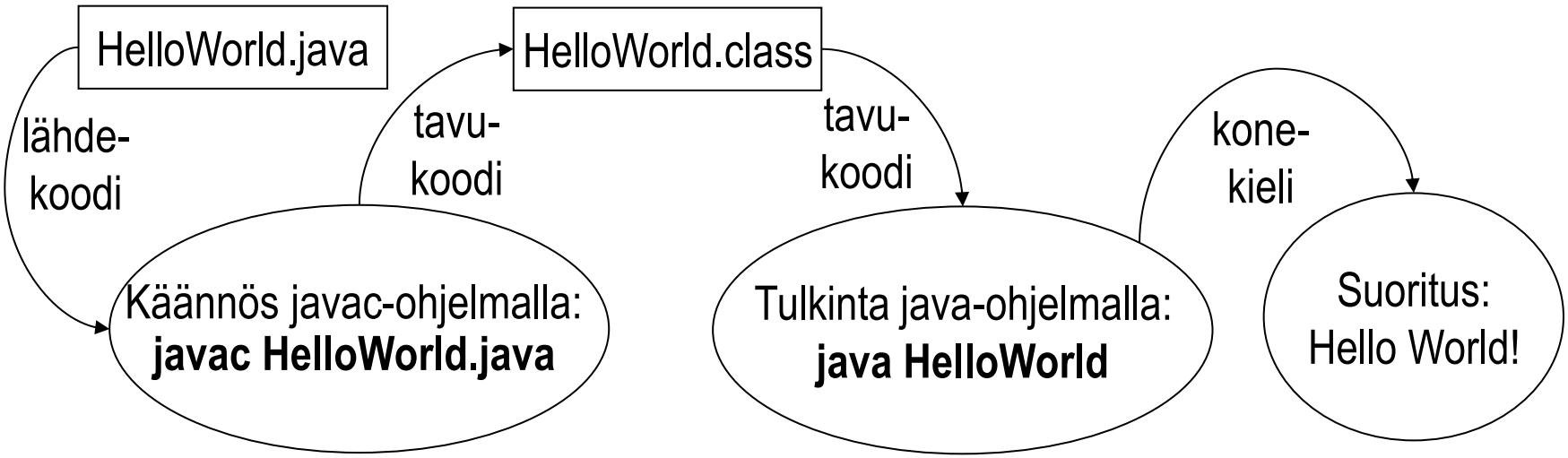
```
C:\laki>dir /b
HelloWorld.java

C:\laki>javac HelloWorld.java

C:\laki>dir /b
HelloWorld.class
HelloWorld.java

C:\laki>java HelloWorld
Hello World!

C:\laki>_
```



Kielioppivirheiden korjaaminen

```
/*
 * Laki-kurssin ensimmäinen Java-ohjelma.
 * Jorma Laurikkala, Tampereen yliopisto.
 */
public class HelloWorld {
   public static void main(string[] args) {
      // Tulostetaan näytölle Hello World -teksti.
      System.out.println("Hello World!");
   }
}
```

Kielioppivirhe:

String-tunnus alkaa pienellä kirjaimella.

Kääntäjän virheilmoitus:

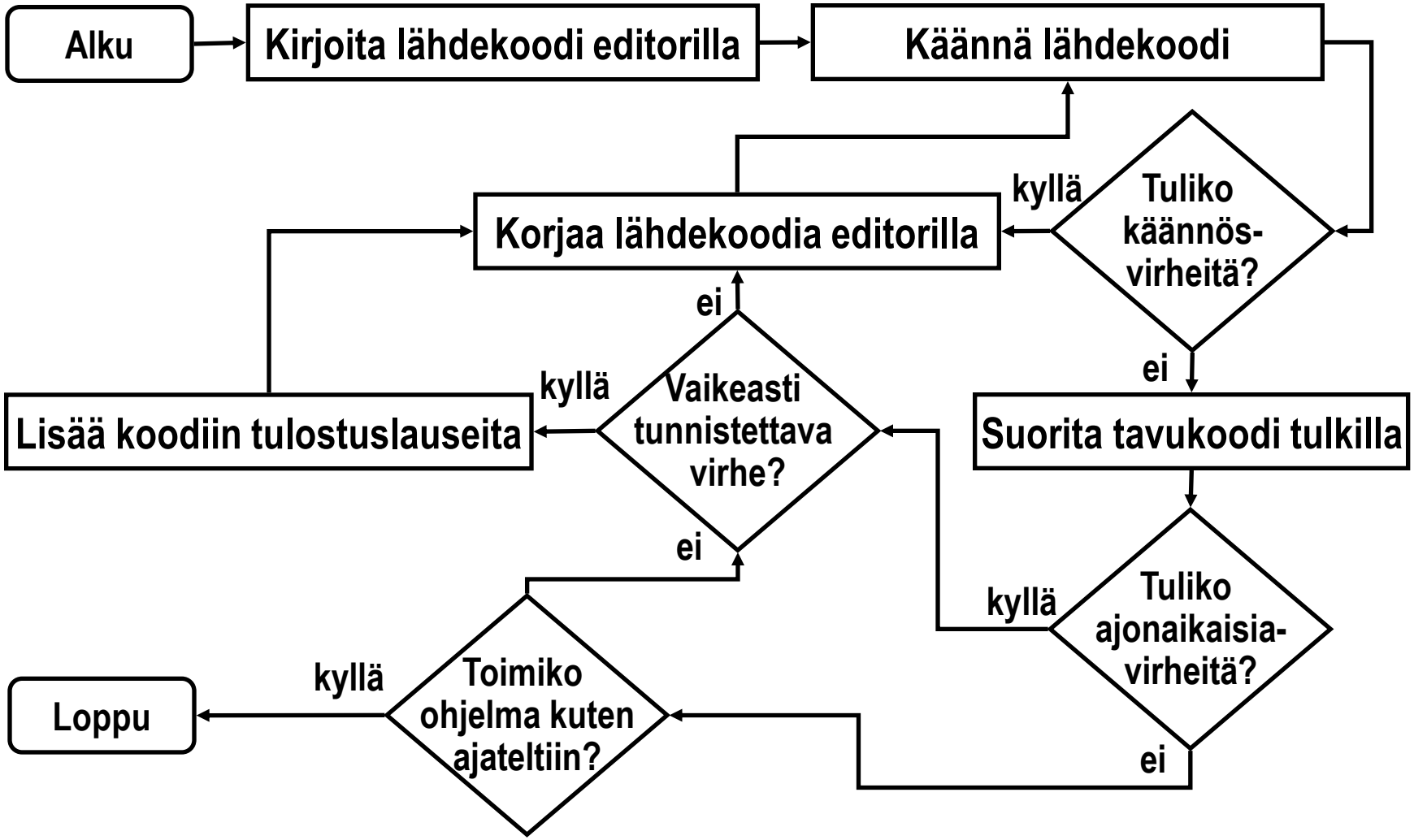
```
HelloWorld.java:6: cannot find symbol
symbol  : class string
location: class HelloWorld
   public static void main(string[] args) {
                      ^
```

- Java-kääntäjä (javac) antaa kielioppivirheestä ilmoituksen.
- Virhe on kääntäjän antamalla rivillä tai sen lähistöllä.
- Korjattu koodi tallennetaan ja käännetään uudelleen.

Suorituksen paljastamat virheet

- Java-ohjelman ajonaikainen virhe tapahtuu tavukoodia tulkilla (java) suoritettaessa.
- Ohjelmaa pysähtyy (eli “kaatuu”) ajonaikaisen virheen seurauksena.
- Ajonaikainen virhe on seurausta virheestä ohjelman logiikassa.
- Usein looginen virhe ei pysäytä ohjelmaa vaan ohjelma ei vain toimi ajatellulla tavalla.
- Ohjelman toimintaa voidaan tarvittaessa seurata tulostuslauseiden avulla.

Java-ohjelman kehitysprosessi karkealla tasolla



6. Muuttujat ja Java

Sisällys

- Muuttujien nimeäminen.
- Muuttujan tyypin määritys.
- Javan tietotyypit:
  - Kokonais- ja liukuluvut, merkit, totuusarvot.
- Tyyppien yhteensopivuus.
- Viitetietotyypit ja merkkijonotietotyyppi String.
- Vakiot.

Muuttujien nimeäminen

- Javan tunnuksissa, kuten muuttujien, operaatioiden ja ohjelman nimissä, voi käyttää kirjaimia, numeroita sekä alaviiva (_) ja dollari ($) -merkkejä.
- Java käyttää monipuolista Unicode-merkistöä, mutta koodaaminen on helpompaa, jos käyttää vain kirjaimia a…ö ja A…Ö sekä numeroita.
- Koska skandinaaviset merkit (å, ä, ö, Å, Ä, Ö) saattavat olla ongelmallisia tiedoston nimessä, on turvallisinta nimetä ohjelma ilman näitä merkkejä.
- Esimerkki: *Paattelya*-tunnus on huonoa suomea, mutta toisaalta *Päättelyä.java*-tiedosto voi olla ongelma käyttö-järjestelmälle.

Muuttujien nimeäminen

- Toisin kuin ohjelman tunnus, muuttujien tunnukset aloitetaan pienellä kirjaimella.
- Tunnuksessa ei voi käyttää välilyöntiä.
- Välilyöntiä ei korvata Java-kielessä alaviivalla, vaan useasta sanasta koostuvissa nimissä sanat kirjoitetaan yhteen.
- Sanat erotetaan toisistaan aloittamalla kukin sana ensimmäistä sanaa lukuun ottamatta isolla alkukirjaimella.
- Esimerkiksi *arvattuLuku*, *salinOvi* tai *syntymapaiva*.
- Tunnusta ei saa aloittaa numerolla.

Muuttujien nimeäminen

- Muuttuja tulee nimetä siten, että nimestä käy ilmi muuttujan käyttötarkoitus.
  - Esimerkiksi *syntymapaiva* on paljon informatiivisempi kuin *s*.
- Yleensä muuttujan nimi on kompromissi: nopeasti kirjoitettavissa, mutta vielä ymmärrettävissä.
- Hyvään ohjelmointitapaan kuuluu myös koodin kommentointi ja koodin sisentäminen asianomaisissa kohdissa.

Muuttujan tyypin määritys

- Vuokaavioissa tyypit on määritelty hyvin heikosti.
  - Muuttujan arvon tyyppi (luku, merkki tai merkkijono) on toistaiseksi päätelty epäsuorasti esimerkiksi muuttujan nimen, muuttujan arvon tai tehtävänannon avulla.
- Java on monien lausekielien tapaan vahvasti tyypitetty kieli, jossa muuttujille on aina erikseen määriteltävä tietotyyppi (data type) eli tyyppi.
- Javassa tyyppien ilmaisuun on varattu englanninkieliset sanat.
  - Esimerkiksi varattu sana **int** osoittaa, että kyseessä on kokonaislukutyyppisen arvon omaava muuttuja.
  - Tyypin nimeä ei voi käyttää esimerkiksi muuttujan nimenä.

Muuttujan tyypin määritys

- Muuttujan tyyppi annetaan muuttujan esittelyn (declaration) yhteydessä.
- Esittelylause yleisesti:

  tyyppi nimi;
- Esittely on ensimmäinen kerta, jolloin muuttuja esiintyy lähdekoodissa.
  - Muuttuja esitellään vain kerran.
  - Samannimiset muuttujat aiheuttavat käännösvirheen.
- Esimerkki: muuttujien *arvaus* ja *etunimi* esittely:
  - Aiemmin esittely tapahtui alkuarvon asettamisen (alustuksen) yhteydessä:

    arvaus ← 10;

    etunimi ← "Matti";
  - Javassa esittely on oma lauseensa, jolloin alustus voidaan tehdä myöhemmin:

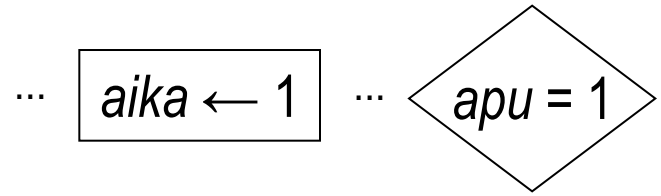
    int arvaus;

    String etunimi;

Muuttujan tyypin määritys

- Useissa lausekielissä, Java mukaan lukien, ei ole omaa merkintää sijoitusoperaatiolle, vaan
  - yhtäsuuruusmerkki (=) on varattu arvojen sijoittamiseen muuttujaan ja
  - arvojen yhtäsuuruuden vertailuun käytetään kahta yhtäsuuruusmerkkiä (==).

- Esimerkki: muuttujan *aika* esittely, alustus ja vertailu:
  - Vuokaavioissa:



  - Javassa:

```
...
int aika;
aika = 1;
...
if (aika == 1)
```

Muuttujan tyypin määritys

- Sijoitusoperaatio voidaan yhdistää myös esittelyyn, jolloin esittely ja alustaminen voidaan suorittaa yhdellä lauseella.
- Yleisesti:

  tyyppi nimi = arvo;
- Esimerkki: muuttujien *arvaus* ja *etunimi* esittely ja alustus:

  int apu = 1;

  String etunimi = "Matti";

- Muuttujat voidaan esitellä a) joko aivan ohjelman alussa tai b) sitä mukaan kuin muuttujia tarvitaan.
- Esitellään toistaiseksi ohjelman alussa; jälkimmäinen tapa vaatii näkyvyys-sääntöjen tuntemista. Esimerkki:

```
// Lasketaan nopeus matkan ja ajan avulla.
public class NopeusLaskuri {
   public static void main(String[] args) {
      // Muuttujien esittelyt. Arvot antaa käyttäjä.
      int matka;
      int aika;
      double nopeus;
      // Ohjelman muut lauseet tulisivat alle.
   }
}
```

Javan tietotyypit

- Javan tietotyypit voidaan jakaa alkeistietotyyppeihin (primitive data types):
  - kokonaisluvut: **byte**, **short**, **int** ja **long**
  - liukuluvut (desimaaliluvut): **float** ja **double**
  - merkit: **char**
  - totuusarvot: **boolean** (arvot **true** ja **false**)

  ja viitetyyppeihin (reference data types): luokka-, rajapinta- ja taulukkotietotyypit.

- Tehtävä määrää minkä tyyppisiä ja kokoisia lukuja on syytä käyttää.
  - Yleensä selviää hyvin **int**- ja **double**-tyypeillä.
- Muuttujilla ei voi olla rajapintatyyppistä arvoa.

Kokonaisluvut

- Koko ilmaistaan tavuina (byte, B).
  - Tavu: kahdeksan bittiä, $2^8 = 256$ arvoa.
  - Bitti (bit, b): tiedon pienin mittayksikkö. Kaksi toisensa poissulkevaa arvoa, usein 0 tai 1.
- Esimerkiksi **short**-tyyppisenä esitelty luku 273 on kooltaan 2 B eli 16 b ja sen bitit ovat: 00000001 00010001.
  - Usein eniten merkitsevä bitti on ensimmäinen vasemmalla.
  - $0 \cdot 2^{15} + 0 \cdot 2^{15} + \ldots + 1 \cdot 2^8 + \ldots + 1 \cdot 2^4 + \ldots + 0 \cdot 2^1 + 1 \cdot 2^0 = 273$
- Tietokone käsittelee tietoa sanan (word) kokoisina yksiköinä. Sana koostuu tavuista.
  - Nykyisin yleisimmät sanan pituudet ovat 4 B (32 b) ja 8 B (64 b).

Kokonaisluvut

Tyyppi	Koko	Pienin arvo	Suurin arvo
byte	1 B (8 b)	$-2^7 = -128$	$2^7 - 1 = 127$
short	2 B (16 b)	$-2^{15} = -32768$	$2^{15} - 1 = 32767$
int	4 B (32 b)	$-2^{31} = -2147483648$, Vakiona: Integer.MIN_VALUE	$2^{31} - 1 = 2147483647$ Vakiona: Integer.MAX_VALUE
long	8 B (64 b)	$-2^{63} =$ -9223372036854775808, Vakiona: Long.MIN_VALUE	$2^{63} - 1 =$ 9223372036854775807 Vakiona: Long.MAX_VALUE

Kokonaisluvut

- Luvun ensimmäinen (merkitsevin) bitti on positiivisilla luvuilla 0 ja negatiivisilla 1.
- Negatiiviset luvut saadaan positiivista niin sanottuna kahden komplementtina: käännetään bitit ja lisätään 1.
- Esim. 10-järjestelmän luku 14 on 8-bittisenä (**byte**) binäärilukuna 00001110 ($2^3 + 2^2 + 2^1 = 14$). Kahden komplementti:

```
  11110001
+ 00000001
----------
  11110010 = -14
```

Liukuluvut

- Desimaalilukujen esittämiseen.
- Desimaalierottimena käytetään pilkun sijasta pistettä.
- Likiarvoja, laskennassa käytetään pyöristystä.
  - Yhtäsuuruuden kanssa kannattaa siis varoa: voi olla, että joskus $a \cdot (b \cdot c) \neq (a \cdot b) \cdot c$.
- Liukuluvut voidaan esittää “tieteellisessä” muodossa $x \cdot 10^y$, jota merkitään ohjelmassa *x*e*y* tai *x*E*y*.
  - Esimerkiksi luku 120,123 = $1{,}20123 \cdot 10^2$, esitetään Javassa joko 120.123, 1.20123e2 tai 1.20123E2.

Liukuluvut

Tyyppi	Koko	Pienin itseisarvo	Suurin itseisarvo
float	4 B (32 b)	2^{-149} Float.MIN_VALUE	$(2 - 2^{-23}) \cdot 2^{127}$ Float.MAX_VALUE
double	8 B (64 b)	2^{-1074} Double.MIN_VALUE	$(2 - 2^{-52}) \cdot 2^{1023}$ Double.MAX_VALUE

- Liukulukujen esitystavan vuoksi niiden arvoalue on huomattavasti kokonaislukujen arvoaluetta suurempi.
- Liukulukuarvot ovat oletusarvoisesti **double**-tyyppisiä.
  - Tästä johtuen lause: **float** korkeus = 1.5; on virheellinen.
  - Arvon saa **float**-tyyppiseksi lisäämällä sen loppuun joko f- tai F-kirjaimen.
  - Esimerkkejä: 1.5f, 1.5F, 1.20123e2f , 1.20123e2F.

Totuusarvotyyppi boolean

- Tyypillä vain arvot **true** (kyllä, tosi) ja **false** (ei, epätosi).
  - Arvoilla ei ole järjestystä.
  - Kokonaisluvut 0 ja 1 eivät vastaa totuusarvoja.

- Esimerkki: totuusarvoisten muuttujien esittelyjä ja alustuksia:

```
boolean oviAuki;
oviAuki = true;
boolean lukuOK = false;
```

- Totuusarvon negaatiota merkitään huutomerkillä (**!**):
  - **!true == false**
  - **!false == true**

- Esimerkki:

```
// Alustetaan muuttuja todeksi.
boolean ekaKierros = true;
// Tästä tulostuu epätosi,
// koska NOT true = false.
System.out.println(!ekaKierros);
```

Merkkityyppi char

- Yksittäisten merkkien esittämiseen.
- Arvot esitetään yksinkertaisilla lainausmerkeillä (') eli niin sanotuilla hipsuilla.
- Esimerkki: **char** vastaus = 'k';
- Isot ja pienet kirjaimet ovat eri asia.
  - Esimerkiksi arvot 'e' ja 'E' eivät ole sama kirjain.
- Java käyttää Unicode-merkistöä, jossa merkille on varattu tilaa 16 bittiä.
- Merkki on myös luku välillä 0…65535.
  - Esimerkiksi luku 32 vastaa välilyöntimerkkiä, joka on ASCII-merkistön (osa Unicode-merkistöä) ensimmäinen tulostuva merkki.

Tyyppien yhteensopivuus sijoituksessa

- Sijoitusoperaatiossa operandien tyyppien on oltava yhteensopivat, jotta ohjelma kääntyisi.
- Kokonais-, liukuluku- ja merkkityyppejä voidaan sijoittaa toisiinsa, mikäli kohdetyyppi on kooltaan (arvoalueeltaan) riittävän suuri.
  - **boolean**-tyyppi on yhteensopiva vain itsensä kanssa.
- Suurempaan kokonaislukutyyppiin (liukulukuun) mahtuu pienempi kokonaislukutyyppi (liukuluku).
- Liukulukutyyppiseen muuttujaan voi sijoittaa mitä tahansa kokonaislukutyyppiä olevan arvon, koska kokonaislukujen arvoalue sisältyy liukulukujen arvoalueeseen.

Tyyppien yhteensopivuus sijoituksessa

- Esimerkki: yhteensopivia ja epäyhteensopivia sijoituksia.

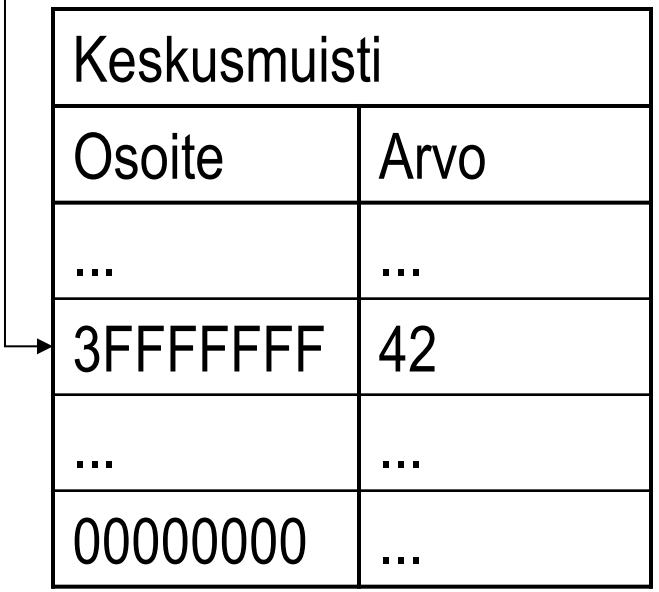
```
// Kokonaislukuja.
int i = 1; long l = 1;
// Liukulukuja.
float f = 1; double d = 1;
// Sijoituksia.
l = i;          // Oikein.
i = l;          // Väärin.
d = f;          // Oikein.
f = d;          // Väärin.
// Kokonaisluku mahtuu
// liukulukumuuttujaan.
f = l;          // Oikein.
d = i;          // Oikein.
// Liukuluku ei mahdu
// kokonaislukumuuttujaan.
l = f;          // Väärin.
i = d;          // Väärin.
```

Viitetietotyypit

- Alkeis- ja viitetyyppiset muuttujien arvot liitetään tunnuksiin eri tavoin.
- Muuttujien arvot tallennetaan johonkin paikkaan (osoitteeseen) keskusmuistissa.
- Alkeistietotyyppisten muuttujien arvot löytyvät suoraan tunnukseen liittyvästä muistiosoitteesta.

- Esimerkki:

```
public class OmaOhjelma { ...
    int kokoluku = 42; ...
}
```

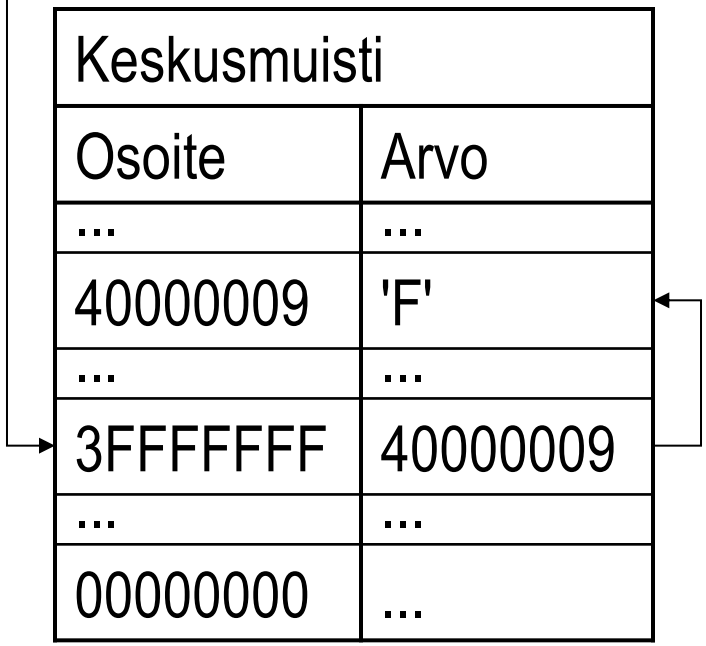


Viitetietotyypit

- Viitetyyppisten muuttujien voidaan ajatella liittyvän arvoonsa epäsuorasti: tunnus viittaa muistiosoitteeseen, jonka kautta arvo löytyy.
- Javan viitteet ovat eri asia kuin C++:n viitteet ja osoittimet.
- Viitteet turvallisia “osoittimia”, joilla ei voi harrastaa esimerkiksi aritmetiikka.

- Esim.

```
public class OmaOhjelma { ...
    String nimi = "Frodo"; ...
}
```



String-merkkijonotyyppi

- Yksittäisten merkkien yhdistelmää kutsutaan merkkijonoksi.
- Arvot esitetään lainausmerkkien (") avulla.
- Esimerkki: String sukunimi = "Meikäläinen";
- Myös yksittäiset lainausmerkkeihin suljetut merkit ovat merkkijonoja.
- Esimerkki:

```
// Yhdestä a-merkistä koostuva merkkijono.
String merkkijono = "a";
// Yksittäinen merkki a.
char merkki = 'a';
```

- String-tyyppi kuuluu viitetyyppeihin (luokkiin), mutta tätä tyyppiä voidaan käyttää alkeistyypin myös tapaan!

String-merkkijonotyyppi

- Merkkijonoon voidaan yhdistää yhteenlaskuoperaatiolla (**+**) alkeistyyppinen arvo tai toinen merkkijono.
- Esimerkki:

```
String ekaMjono = "opinto";
String tokaMjono = "piste";
// "opintopiste"
String kolmasMjono = ekaMjono + tokaMjono;
```

- Esimerkki:

```
String tulosTeksti = "Tulos oli ";
int tulos = 10;
// "Tulos oli 10."
tulosTeksti = tulosTeksti + tulos + ".";
System.out.println(tulosTeksti);
```

Vakiot

- Vakiot (constant) ovat muuttujia, joiden arvoa ei voi muuttaa alustamisen jälkeen.
- Vakiot esitellään Javassa **final**-määreellä.
- Vakioiden tunnukset kirjoitetaan aina ISOIN KIRJAIMIN, jotta ne erottuvat paremmin muuttujista.
- Vakio alustetaan yleensä esittelynsä yhteydessä.
- Esimerkki:

```
final double PII = 3.14;
// Kääntäjä ei hyväksy uutta sijoitusta.
PII = 3.1415;
```

- Kuuluvat hyvään ohjelmointitapaan.

Vakiot

- Hyödyllisiä, kun tiettyä arvoa käytetään monessa kohtaa lähdekoodissa: Arvo voidaan muuttaa kätevästi vain muuttamalla vakion alustuslausetta!
- Oheisessa esimerkissä piin likiarvon muuttuessa tarvitsee muuttaa vain yhtä lausetta.

```
// Arvo 3.14 kahdessa
// lauseessa.
double ala = 3.14 * r * r;
double keha = 2 * 3.14 * r;

// Arvo 3.14 vakiona.
final double PII = 3.14;
double ala = PII * r * r;
double keha = 2 * PII * r;
```

7. Näytölle tulostaminen

Sisällys

- System.out.println- ja System.out.print-operaatiot.
- Tulostus erikoismerkeillä.
- Edistyneempää tulosteiden muotoilua.

Tulostusoperaatiot

- System.out.println-operaatio tulostaa parametrina annetun arvon näytölle ja vaihtaa riviä.
- Lause System.out.println(); vaihtaa vain riviä.
- System.out.print-operaatio ei tulosta rivinvaihtoa.
- Esimerkki:

```
// Esitellään ja alustetaan String-tyyppiset muuttujat.
String hello = "Hello ";
String world = "World";
// Tulostetaan yhdelle riville ja vaihdetaan lopuksi riviä.
System.out.print(hello); System.out.println(world);
```

Tulostusoperaatiot

- Myös alkeistyyppien arvot voi tulostaa suoraan näytölle. Arvo sellaisenaan, muuttujan tai vakion arvo, operaation paluuarvo, lausekkeen tulos jne.
- Esimerkki:

```
/// Esitellään ja alustetaan alkeistyyppiset muuttujat.
int kluku = 10;
double lluku = 1.2345;
boolean tarvo = true;
```

Tulostusoperaatiot

- Esimerkki jatkuu:

```
// Tulostetaan riveittäin näytölle.
System.out.println(kluku);          // 10
System.out.println(lluku);          // 1.2345
System.out.println(tarvo);          // true
System.out.println(kluku + 3);      // 13
System.out.println(2015);           // 2015
```

Tulostusoperaatiot

- String-tyypin arvoille määritelty yhteenlaskuoperaatio helpottaa tulostamista.

- Esimerkki:

```
// Esitellään ja alustetaan muuttuja.
boolean lippu = false;
// Tulostetaan kahdella lauseella.
System.out.print("Lippu == ");
System.out.println(lippu);
// Tulostetaan yhdellä lauseella.
System.out.println("Lippu == " + lippu);
```

Tulostus erikoismerkeillä

- Erikoismerkkejä (escape characters) käytetään merkkeinä ja merkkijonon osina.
- Aloitetaan kenoviivalla (\).
- Koostuvat siis kahdesta merkistä, mutta käsitellään yhtenä merkkinä.
- Suljetaan yksinkertaisten lainausmerkkien sisään. Esim. '\n' ja '\t'.

- **\n** rivinvaihto
- **\r** telanpalautus (rivin alkuun)
- **\t** tabulaattori
- **\f** sivunvaihto
- **\b** merkki vasemmalle
- **\'** yksinkertainen lainausmerkki
- **\"** lainausmerkki
- **** kenoviiva

Tulostus erikoismerkeillä

```
public class Erikoismerkit {
   public static void main(String[] args) {
      System.out.println("\t*******\n\t* MOI *\n\t*******");
   }
}
```

```
*******
* MOI *
*******
```

Edistyneempää tulosteiden muotoilua

- Erityisesti liukulukuarvojen muotoiluun on tarvetta, koska Java tulostaa oletusarvoisesti liukuluvut yli 10 desimaalin tarkkuudella.
- Java-kielen versiosta 1.5.0 on tarjolla C-kielen **printf**-operaation tapainen tulosteiden muotoilu muun muassa
  - System.out.printf-operaation,
  - *Formatter*-luokan operaatioiden tai
  - String.format-operaation

  avulla.
- Muotoilu näillä keinoilla on valitettavan monimutkaista.

Edistyneempää tulosteiden muotoilua

- Ideana on antaa tulosteen muoto merkkijonona:

  "%[parametrin numero][lippu][kentän pituus][.tarkkuus]muunnos"

  missä hakasulkeiden sisään suljetut määreet ovat valinnaisia.
- Muunnosmääre kertoo tulosteen tyypin: esimerkiksi f-merkki on varattu liukuluvuille.
- Tarkkuusmääre on luonnollisesti käytettävissä vain liukulukujen yhteydessä.
- Lisätietoja esimerkiksi osoitteissa:

  http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

  https://docs.oracle.com/javase/tutorial/java/data/numberformat.html

Edistyneempää tulosteiden muotoilua

- Esimerkki: tulostetaan osamäärä kahden desimaalin tarkkuudella System.out.printf-operaatiolla.

```
// Huomaa tyyppimuunnos.
double osamaara = 7 / 3d;
// 2.3333333333333335
System.out.println(osamaara);
// 2.33
System.out.printf("%.2f", osamaara);
```

8. Näppäimistöltä lukeminen

Sisällys

- Arvojen lukeminen näppäimistöltä Java-kielessä.
- In-luokka.
- In-luokka, käännös ja tulkinta
- Scanner-luokka.

Yleistä

- Näppäimistöltä annettujen arvojen (syötteiden) lukeminen on periaatteessa helppoa:
  - Lukuoperaation kohdatessaan ohjelman jää odottamaan käyttäjän syötettä.
  - Näppäimistöltä annettu syöte lähetetään ohjelmalle Enter-näppäintä painamalla.
  - Ohjelman suoritus jatkuu syötteen lähetyksen jälkeen.
- Ongelma: lukuoperaatio odottaa saavansa tietyn tyyppisen arvon, mutta ohjelman käyttäjä voi antaa syötteeksi jotain muuta tyyppiä olevan arvon.
  - Esimerkiksi ollaan lukemassa kokonaislukua ja käyttäjä syöttää syystä tai toisesta ohjelmalle liukuluvun.

Yleistä

- Jos lukuoperaatio ei pysty muuttamaan syötettä halutun tyyppiseksi, tapahtuu ajonaikainen virhe (runtime error).
- Ajonaikainen virhe pysäyttää ohjelman suorituksen, ellei siihen ole varauduttu lukuoperaation yhteydessä.
- Lisävaikeuksia aiheuttaa itse Java-kieli, jossa tietojen vakavampi lukeminen näppäimistöltä vaatii olio-ohjelmointia ja ajonaikaisten virheiden käsittelyä.
- Laki 1 ja laki 2 -kursseilla asiat pyritään pitämään yksinkertaisina ja syötteiden lukemiseen käytetään pääasiassa omatekoista In-luokkaa, jonka operaatioihin virheidenkäsittely on koodattu valmiiksi.

In-luokka

- Luokka sisältää operaatiot:
  - **readInt**() **int**-tyyppisen kokonaisluvun lukemiseen
  - **readDouble**() **double**-tyyppisen liukuluvun lukemiseen
  - **readString**() merkkijonon (String) lukemiseen ja
  - **readChar**() merkin (**char**) lukemiseen.
- Nämä operaatiot eivät luovuta helpolla: lukemista jatketaan kunnes on saatu kelvollinen syöte.
- Luokka löytyy kurssin kotisivuilta.

In-luokka

- Operaatioita kutsutaan pistenotaatiolla, joka on tuttu jo tulostamisen yhteydessä: In.operaatio
- Esimerkki:

```
// Luetaan käyttäjältä desimaaliluku
// ja sijoitetaan syöte muuttujan arvoksi.
double korkeus = In.readDouble();
```

- Lukuoperaation palauttama arvo sijoitetaan usein muuttujaan, jotta arvoa voidaan käyttää myöhemmin.
  - Muuttujan on oltava operaation palauttaman arvon kanssa sopivaa tyyppiä.

In-luokka

```
// Lasketaan nopeus matkan ja ajan avulla.
public class NopeusLaskuri {
  public static void main(String[] args) {
    // Muuttujien esittelyt. Arvot antaa käyttäjä.
    int matka; // Matka kilometreinä.
    int aika; // Aika tunteina.
    double nopeus; // Kilometriä tunnissa.
    ...
    // Luetaan matka ja aika käyttäjää ohjeistaen.
    System.out.println("Anna matka (km):");
    matka = In.readInt();
    System.out.println("Anna aika (h):");
    aika = In.readInt();
    ...
  }
}
```

- Luetaan matka ja aika käyttäjältä In-luokan avulla.
  - On hyvä tapa viestiä käyttäjälle, että hänen tulisi antaa syöte.
- Kumpikin syöte sijoitetaan myöhempää käyttöä varten omaan muuttujaansa.
- Ohjelman lähdekoodi löytyy kokonaisuudessaan kurssin sivuilta.
- Kommentti on joskus rivin lopussa.
  - Harvinaisempaa kuin omalla rivillään kommentointi, koska lauseen ja kommentin sisältävästä rivistä tulee helposti liian pitkä.

In-luokka, käännös ja tulkinta

- In-luokka on käännettävä yhdessä oman ohjelman (luokan) kanssa. Tämä on tehtävissä eri tavoin.
- Helpointa on kopioida luokka samaan hakemistoon kuin oman ohjelma, jolloin käännettäessä In-luokka kääntyy ilman lisätoimia. Esimerkki:

  javac NopeusLaskuri.java
- Näin käännetty ohjelma on ajettavissa tutulla komennolla:

  java NopeusLaskuri
- In-luokan voi ottaa käyttöön ohjelmaa käännettäessä ja ajettaessa myös jostakin muusta hakemistosta.

In-luokka, käännös ja tulkinta

- Oletetaan esimerkiksi, että In-luokan lähdekoodi löytyy Windows-käyttöjärjestelmässä NopeusLaskuri-ohjelman sisältävän työhakemiston ylihakemistosta.
- Tällöin ohjelma voidaan kääntää vaihtoehtoisesti komennoilla:

  javac NopeusLaskuri.java ..\In.java

  (Linux ja Mac: javac Tehtavat.java ../In.java) tai

  javac -sourcepath .. NopeusLaskuri.java

- ja suorittaa komennolla:

  java -classpath .;.. NopeusLaskuri

  (Linux ja Mac: java -classpath .:.. NopeusLaskuri)

Scanner-luokka

```
import java.util.*; // Otetaan Scanner-luokka käyttöön import-lauseella.
public class ScannerDemo {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in); // Liitetään oletussyötevirtaan.
    try { // Yritetään lukea kokonaisluku.
      System.out.print("Anna luku: ");
      int luku = sc.nextInt();
    }
    catch (Exception e) { // Jos tapahtui virhe, niin se "siepataan" tänne.
      System.out.println("Tapahtui virhe: " + e);
    }
    // Suljetaan.
    sc.close();
  }
}
```

9. Aritmeettiset operaatiot

Sisällys

- Operaatiot ja operadit.
- Tietotyyppimuunnokset.
- Yli- ja alivuoto.

Aritmeettiset operaatiot

- Operandit a ja b ovat kokonais- tai liukulukutyyppisiä arvoja.
  - Operandin arvo voi olla pelkkä arvo tai lausekkeen tulos.
- Operaattori ja sen operadit muodostavat lausekkeen.
  - Lausekkeen tuloksen määrää tyyppimuunnos, jos operandit ovat erityyppisiä.
- a **+** b (yhteenlasku),
  a **-** b (vähennys),
  a * b (tulo),
  a **/** b (osamäärä), ja
  a **%** b (jakojäännös, modulo, mod).
- Esimerkki: 5 % 3 == 2, 15 % 5 == 0, -41 % 7 == -6

Aritmeettiset operaatiot

- Laskujärjestys on koulusta tuttu.
  - Esimerkiksi: 2 + 3 * 4 == 14.
- Suoritetaan vasemmalta oikealle.
- Ensin kerto- (*) ja jakolaskut (/, %), sitten yhteen- ja vähennyslaskut (+, -).
- Laskujärjestykseen voi vaikuttaa suluilla.
- Aritmeettiset operaatiot ovat sijoitusta vahvempia.
  - Esimerkki: i = i + 1; // Summa lasketaan ensin, sitten sijoitetaan.
- Sijoitusoperaatiossa aritmeettisen operaation tuloksen ja muuttujan tyyppien on oltava yhteensopivat.

Implisiittinen tyyppimuunnos

- Aritmeettisen operaation yhteydessä tapahtuu implisiittinen tyyppimuunnos (type conversion, casting), joka määrää automaattisesti operaation tuloksen tyypin.
- Esimerkki:

```
int a = 1; long b = 2; boolean c = true;
int summa1 = a + b;                 // Väärin.
long summa2 = a + b;                // Oikein.
int summa3 = c + 33;                // Väärin.
long osamaara1 = 7 / b;             // 3
double osamaara2 = 7 / 2.0;         // 3.5
```

Implisiittinen tyyppimuunnos

if (ainakin toinen operandi on **double**)
    Tulos on tyyppiä **double**;
else if (ainakin toinen operandi on **float**)
    Tulos on tyyppiä **float**;
else if (ainakin toinen operandi on **long**)
    Tulos on tyyppiä **long**;
else if (ainakin toinen operandi on **int**)
    Tulos on tyyppiä **int**;
else if (ainakin toinen operandi on tunnus)
    Tulos on tyyppiä **int**;
else
    Tulos on kokonaislukutyyppi;

- Jos aritmeettisen operaation toinen operandi on liukuluku, niin tulos on aina liukuluku.
- **long**- ja **int**-tyypit muita kokonais-lukutyyppejä “vahvempia”.

Implisiittinen tyyppimuunnos

```
byte b = 1; int i = 2; long l = 3; float f = 4; double d = 5;

double t1 = d * f;      // Oikein.
double t2 = l * d;      // Oikein.
int t3 = i * f;         // Väärin.
long t4 = f * d;        // Väärin.
int t5 = b * i;         // Oikein.
float t6 = d * f;       // Väärin.
```

Implisiittinen tyyppimuunnos

```
byte b = 1; int i = 2; long l = 3; float f = 4;  double d = 5;

float t7 = b * l;              // Oikein. Kokonaisluvut
                               // kuuluvat liukulukuihin.
float t8 = l * i;              // Oikein.
double t9 = l * i;             // Oikein.
byte t10 = b * b;              // Väärin. Tulos int-tyyppiä.
short t11 = 2 * 300;           // Oikein. Arvo mahtuu.
```

Eksplisiittinen tyyppimuunnos

- Tyyppi on muunnettavissa toiseksi myös ohjelmoijan toimesta eli eksplisiittisesti.
- Arvon tyyppi voidaan määrätä lisäämällä loppuun kirjain: **long** (l tai L), **float** (f tai F) ja **double** (d tai D).
- Esimerkki: 65L, 3.14f, 1D
- Muuttujien (ja arvojen) tyyppi vaihdetaan muunnos-operaatiolla (cast). Yleisesti:

  (kohdetyyppi)arvo

Eksplisiittinen tyyppimuunnos

- Esimerkki:

```
final float PII = (float)3.14;
double d = 0.0015;
float f = PII + (float)d;          // 3.1415
```

- Muunnosoperaatiota on käytettävä varoen.
- Operaatiolla voi “pakottaa” muuttujaan tai vakioon liian suuren tai liian pienen luvun.
- Esimerkki:

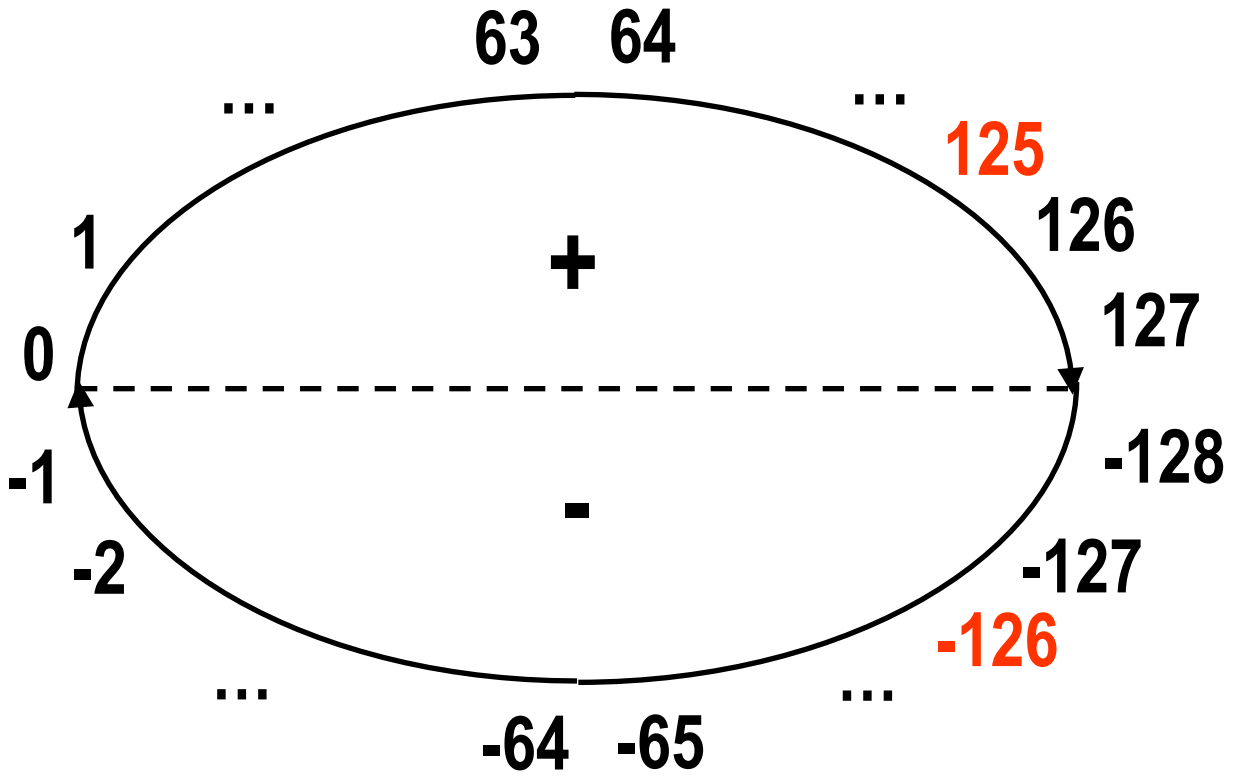
```
// Sijoituksen tulos on ylivuodon vuoksi 44
byte b = (byte)300;
```

Yli- ja alivuoto

- Ylivuodossa luvun arvoalueen yläraja ylitetään, jolloin “pyörähdetään” alarajan kautta negatiivisten lukujen puolelle.
- Alivuoto tapahtuu, kun arvoalueen alaraja alitetaan.
- Yli- ja alivuoto voi syntyä aritmeettisen operaation tuloksena tai huonosti tehdyn tyyppimuunnoksen vuoksi.
- Yli- ja alivuodot eivät aiheuta ajonaikaista virhettä Javassa.

Yli- ja alivuoto

- Esimerkki: **byte** b1 = (**byte**)(127 + 3); // -126
- Esimerkki: **byte** b2 = (**byte**)(-128 - 3); // 125



10. Javan ohjausrakenteet

Sisällys

- Ohjausrakenteiden koostumus.
- Vertailuoperaattorit.

Ohjausrakenteet

- Päätökset (eli valinta) ja toisto esitetään ohjausrakenteilla (control structures), jotka ovat tietyn muotoisia lauseita.
  - Peräkkäisyyteen lisätty valinta ja toisto riittävät minkä tahansa laskettavissa olevan tehtävän ratkaisuun.
- Ohjausrakenne koostuu:
  - Varatusta sanasta, joka kertoo mikä rakenne on kyseessä.
  - Yhdestä tai useammasta ohjausrakenteeseen liittyvästä lauseesta.
  - Ohjausrakenteen lauseet suoritetaan, jos ohjausrakenteeseen liittyvä ehtolauseke palauttaa totuusarvon tosi (kyllä).
    - Ehtolause sisältää vertailuoperaattorin.
    - Ehtolauseke sijoitetaan kaarisuljeparin sisään (()).

Ohjausrakenteet

- Rakenteelle varattu sana ja rakenteen ehto kirjoitetaan useimmiten riville.
- Rakenteeseen liittyvät lauseet alkavat seuraavalta riviltä.
- Usean lauseen ($n > 1$) liittyminen ilmaistaan kokoamalla yksittäiset lauseet kootuiksi lauseiksi aaltosulkeiden ({ }) avulla.

- Esimerkki:

```
if (ehto)
   lause 1;
```

```
if (ehto) {
   lause 1; …
   lause n;
}
```

- Ohjelman rakennetta selkeytetään sisentämällä kootun lauseen sisältö.
- Koottua lausetta ei päätetä puolipisteellä.

Vertailuoperaattorit

Operaatio	Ennen	Javassa
Pienempi kuin	<	**<**
Suurempi kuin	>	**>**
Pienempi tai yhtä suuri kuin	≤	**<=**
Suurempi tai yhtä suuri kuin	≥	**>=**
Yhtäsuuruus	=	**==**
Erisuuruus	≠	**!=**

Vertailuoperaattorit

- Kaikilla vertailuoperaattoreilla on kaksi operandia ja **boolean**-tyyppinen paluuarvo (**true** tai **false**).
- Lukuja ja merkkejä voi vertailla toisiinsa tiedon tyypistä riippumatta.
- Totuusarvoille ja merkkijonoille on käytettävissä vain eri- ja yhtäsuuruusvertailu.
  - Merkkijonojen vertailuun tarvitaan String-tietotyypin viiteluonteen vuoksi usein erillinen operaatio (equals).
- Vertailuoperaatiot aritmeettisia operaatioita heikompia: vertailut tehdään aritmetiikan jälkeen.

11. Javan valintarakenteet

Sisällys

- If- ja if-else-lauseet.
- Orpo else.
- Valintaa toisin: switch-lause.

Valintarakenteet

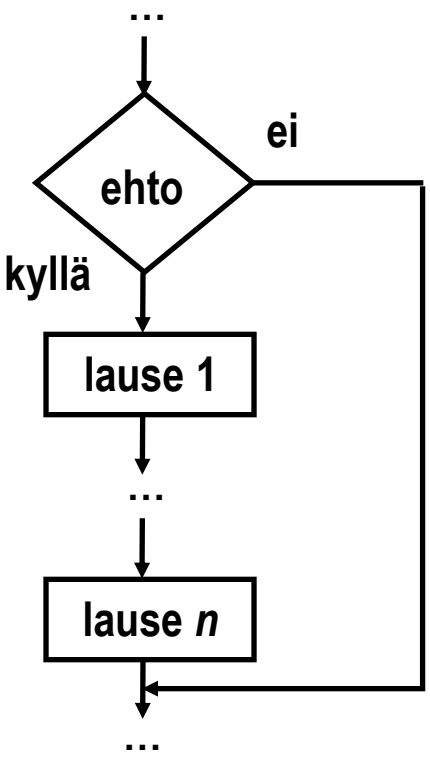
- Valintarakenteilla ilmaistaan formaalisti, kuinka algoritmin suoritus voi “haarautua” ehdosta riippuen.
- Tässä tärkein työkalu on if-lause, jolla kuvataan mitä täytyy tehdä, jos lauseen ehto on tosi.
- If-lauseen laajennos, if-else-lause, kuvaa kuinka algoritmi toimii, kun ehto on tosi ja epätosi.
- If-lause kuvaa yksi- ja else-lause kaksihaaraista päätöstä.
- Sisäkkäisillä valintarakenteilla voidaan kuvata näitä monimutkaisempia päätöksiä.

If-lause

- Kuvaa yksittäisen päätöshaaran: lauseet suoritetaan vain ehdon toteutuessa.
- Sulkujen sisällä olevan ehto palauttaa totuusarvon true toteutuessaan.
- Aaltosulkeet voidaan jättää pois, jos rakenteeseen liittyy yksi lause (*n* = 1).

- Yleisesti:

```
if (ehto) {
   lause 1; ...
   lause n;
}
```



- Esimerkki:

```
if (lkm > 0) {
   ka = summa / lkm;
   System.out.println(ka);
}
```

If-lause

```
// Tutkitaan ovatko luvut yhtä suuret.
public class Samatko1 {
   public static void main(String[] args) {
      // Esitellään ja alustetaan luvut.
      int luku1 = 1;
      int luku2 = 1;
      // Tulostetaan, jos yhtä suuret.
      if (luku1 == luku2)
         System.out.println("Yhtä suuret.");
   }
}
```

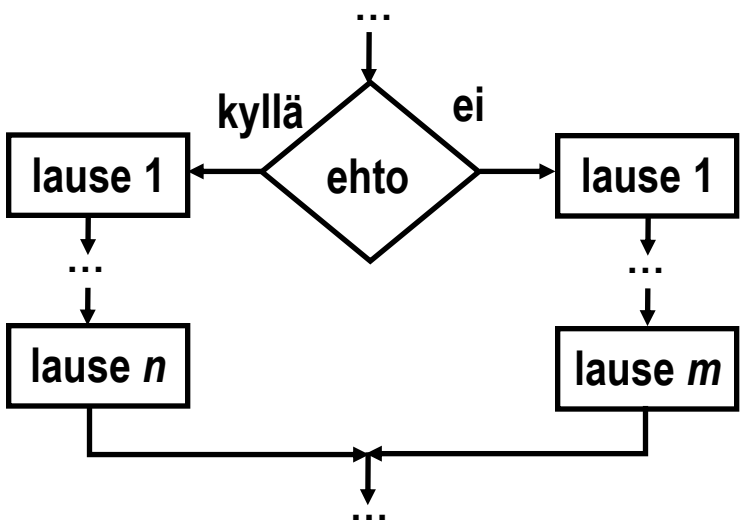
- Oheinen ohjelma tulostaa viestin vain, jos luvut ovat yhtä suuret.
- Eri suuruus jätetään huomiotta.
- Myös yksittäinen ohjausrakenteeseen liittyvä lause sisennetään.

If-else-lause

- Kuvaa kaksi-haaraisen päätöksen.
- Lauseet suoritetaan vaihto-ehtoisesti: ehdon ollessa tosi suoritetaan if-osa, muuten else-osa.
- Jos *n* = 1 tai *m* = 1, voidaan aalto-sulkeet jättää pois.

- Yleisesti:

```
if (ehto) {
   lause 1;
   ...
   lause n;
}
else {
   lause 1;
   ...
   lause m;
}
```



- Esimerkki:

```
if (lkm > 0) {
   ka = summa / lkm;
   System.out.println(ka);
}
else
   System.out.println("Ei lukuja!");
```

If-else-lause

```
// Tutkitaan ovatko luvut yhtä suuret.
public class Samatko2 {
   public static void main(String[] args) {
      // Esitellään ja alustetaan luvut.
      int luku1 = 1;
      int luku2 = 2;
      // Tulostetaan, jos yhtä suuret.
      if (luku1 == luku2)
         System.out.println("Yhtä suuret.");
      else
         System.out.println("Eri suuret.");
   }
}
```

- Oheinen ohjelma kertoo ovatko luvut yhtä vai eri suuret.
- Else-osa suoritetaan, kun lauseke *luku1* == *luku2* on epätotta (false).

Sisäkkäiset valintarakenteet

- If- ja if-else-lauseita voidaan yhdistää moni-mutkaisemmiksi valinnoiksi.
- Sisennys auttaa sisäkkäisyyden ymmärtämistä.

```
public class PolttaaPolttaa {
   public static void main(String[] args) {
      int lampo = 26;
      if (lampo > 15)          // > 15
         if (lampo < 25)       // 15 < lampo < 25
            System.out.println("Sopiva.");
         else                  //  >= 25
            System.out.println("Liian kuuma!");
      else                     // <= 15
          System.out.println("Liian kylmä!");
   }
}
```

Orpo else

- Valintarakenteita yhdistettäessä ei ole aina selvää mihin **if**-osaan **else**-osa liittyy.
- Orpo else -ongelma (dangling else): Onko **if** (ehto) **if** (ehto) lause x; **else** lause y;

```
if (ehto)                          if (ehto)
   if (ehto)                          if (ehto)
      lause x;          vai              lause x;
else                                  else
   lause y;                              lause y;
```

Orpo else

- Javassa orpo **else**-osa liittyy aina lähimpään vapaaseen **if**-osaan.

- **if** (ehto)
  if (ehto)
  lause x;
  else
  lause y;

- Sisentämällä valinta-rakenteista saadaan luettavampia.
- Koodia muotoilemalla ei kuitenkaan voida määrätä mihin **if**-osaan orpo **else** liittyy.
- Koottuja lauseita käyttämällä voidaan vaikuttaa oletussuoritus-järjestykseen.

Orpo else

```
if (ehto)
    if (ehto)
        lause x;
    else
        lause y;
```

- **Else** liittyy jälkimmäiseen **if**-osaan.

```
if (ehto) {
    if (ehto)
        lause x;
}
else
    lause y;
```

- **Else** liittyy ensimmäiseen **if**-osaan.

switch-lause

- **switch** (arvo) {
  case vakio 1:
  lause *k*; ...
  break;
  case vakio 2:
  lause *l*; ...
  break;
  ...
  default:
  lause *m*; ...
  break;
  }

- Valinta arvojen tai vakioiden avulla.
- Arvon oltava **char**, **byte**, **short** tai **int**.
  - Javan 1.7-versiosta alkaen String-tyyppi sallittu.
- Jos **break**-lause puuttuu jostakin valinnasta, niin tästä kohdasta alkanut suoritus jatkuu, kunnes kohdataan seuraava **break**-lause tai valintarakenne loppuu.
  - **break**-lauseen unohtuminen jostakin valinnasta on varsin yleinen ohjelmointivirhe.
- **default**-osa suoritetaan, jos mitään muuta lauseen osaa ei suoriteta.
- Lause sopii hyvin erilaisten valikoiden toteuttamiseen.

switch-lause

```
// Ohjelma, jossa on valikko.
public class Rekisteri {
    public static void main(String[] args) {
        // Ohjelman tuntemat
        // komennot vakioina
        final char LISAA = 'l';
        final char POISTA = 'p';
        final char HAE = 'h';
        final char LOPETA = 'e';
        // Muuttuja, johon luetaan käyttäjän
        // näppäimistöltä antama valinta.
        char valinta;
        …
        // Valitaan toiminto if-else-rakenteilla.
        if (valinta == LISAA) {
            System.out.println("Lisään..."); …
        }
        else if (valinta == POISTA) {
            System.out.println("Poistan..."); …
        }
        else if (valinta == HAE)
            System.out.println("Haen..."); …
        }
        else if (valinta == LOPETA) {
            System.out.println("Lopetan..."); …
        }
        else
            System.out.println("Virhe.");
        …
    }
}
```

switch-lause

```
…
// Valitaan toiminto switch-rakenteella.
switch (valinta) {
    case LISAA:
        System.out.println("Lisään..."); …
        break;
    case POISTA:
        System.out.println("Poistan..."); …
        break;
    case HAE:
         System.out.println("Haen..."); …
         break;
    case LOPETA:
        System.out.println("Lopetan..."); …
        break;
    default:
        System.out.println("Virhe.");
        break;
}
…
```

12. Javan toistorakenteet

Sisällys

- Yleistä toistorakenteista.
- Laskurimuuttujat.
- While-, do-while- ja for-lauseet.
- Laskuri- ja lippumuuttujat.
- Tyypillisiä ohjelmointivirheitä.
  - Silmukan rajat asetettu kierroksen verran väärin.
  - Ikuinen silmukka.
- Silmukoinnin lopettaminen break-lauseella.
- Sisäkkäiset silmukat.

Yleistä

- Toistorakenteilla formalisoidaan algoritmin osien toistuva suoritus.
- Toisto voidaan ilmaista yhtäpitävästi while-, do-while- tai for-lauseilla, mutta yleensä jokin näistä on tietyssä tilanteessa muita luontevampi.
  - **goto** on varattu sana vaikka goto-rakenne ei ole Javassa käytettävissä
- Toistorakennetta kutsutaan lyhyesti silmukaksi (loop).
- Laskurimuuttujalla pidetään kirjaa siitä, monesko silmukan kierros on meneillään, kun silmukan kierrosten määrä tunnetaan.
- Silmukkaa ohjataan lippumuuttujalla, kun silmukan kierrosten lukumäärää ei tunneta ennen silmukan käynnistymistä.
- Virheellisestä logiikasta voi seurata muun muassa
  - silmukan toisto yhden kerran liikaa tai liian vähän tai
  - silmukan loputon suorittaminen (ikuinen silmukka).

Laskurimuuttujat

- Silmukan toistoa ohjaava ehtolause muotoillaan usein laskurimuuttujan (counter) avulla.
  - Tyypillisesti laskurilla on kokonaislukuarvo, joka kasvaa yhdellä kullakin kierroksella.
  - Voidaan nimetä lyhyesti ellei ilmeistä parempaa nimeä löydy.
- Esimerkki:

```
int i = 0;
while (i < 10) {
  i = i + 1; ...
}
```

- Laskurin arvoa voidaan päivittää myös tehtävään paremmin sopivin tavoin.
- Esimerkki:

```
int i = 10;
while (i > 0) {
  i = i - 1; ...
}
```

- Esimerkki:

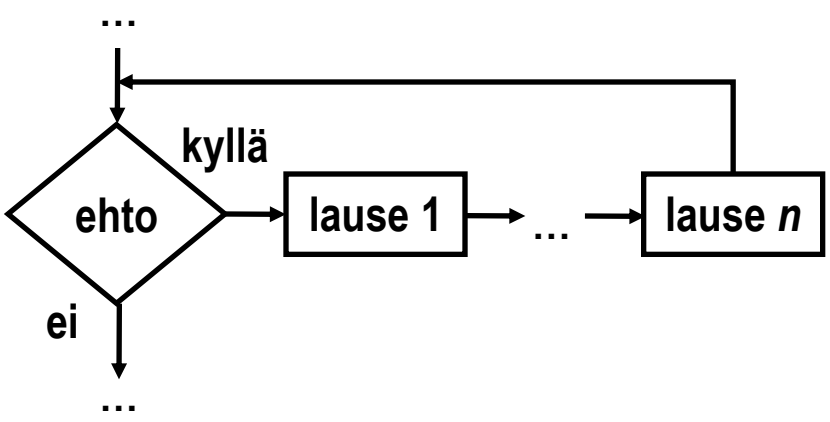
```
int i = 1;
while (i < 7) {
  i = i + 2; ...
}
```

While-lause

- Lauseen sisältöä suoritetaan niin kauan kuin ehto on tosi (**true**).
  - Sisältö merkitään Javalle kootulla lauseella, ohjelman lukijaa autetaan sisentämällä .
- Esiehtoinen – ehto on lauseen alussa.
  - Silmukan lauseet ohitetaan, jos ehto on epätosi jo while-lauseeseen tultaessa.
- Kootun lauseen voi poistaa, kun *n* = 1.

- Yleisesti:

```
while (ehto) {
  lause 1; ...
  lause n;
}
```

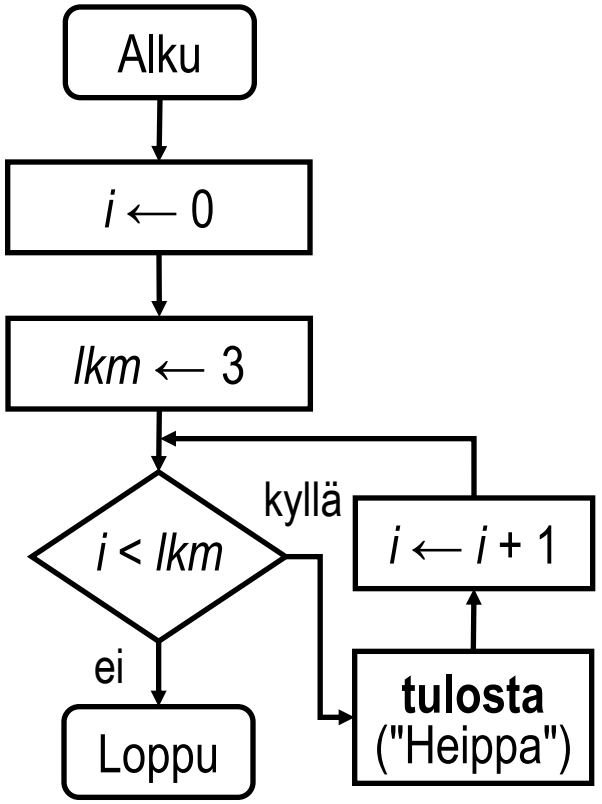


- Esimerkki:

```
while (i <= lkm) {
  luku = ln.readDouble();
  summa = summa + luku;
  i = i + 1;
}
```

While-lause

```
// Tervehtimistä esiehtoisella silmukalla.
public class Heippa1 {
   public static void main(String[] args) {
      // Laskuri ja kierrosten lukumäärä.
      int i = 0;
      int lkm = 3;
      // Tervehditään lkm kertaa.
      while (i < lkm) {
         System.out.println("Heippa!");
         i = i + 1;
      }
   }
}
```



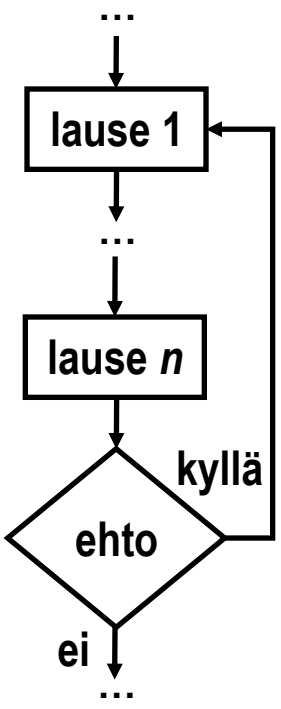
Oletetaan, että lkm on aina nollaa suurempi.

Do-while-lause

- Jälkiehtoinen – ehto on lauseen lopussa.
  - Lauseet suoritetaan ainakin kerran, mikäli tätä ei estetä valintarakenteella.
  - Käytä, kun silmukoidaan vähintään kerran.
- Koottua lausetta ei pitäisi poistaa, kun *n* = 1.
  - Lukija voi ymmärtää lauseen while-osan while-lauseeksi.
- Päätetään puolipisteeseen.
  - Älä lisää puolipistettä while-lauseen ehdon jälkeen!
- Muista sisennys.

- Yleisesti:

```
do {
  lause 1; ...
  lause n;
}
while (ehto);
```

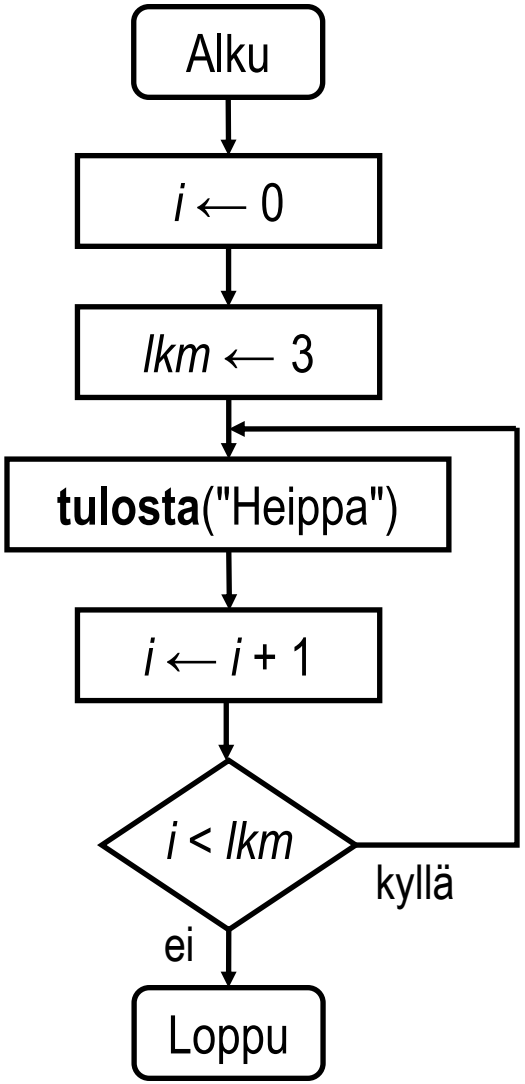


- Esimerkki:

```
do {
   luku = In.readDouble();
   summa = summa + luku;
   i = i + 1;
}
while (i <= lkm);
```

Do-while-lause

```
// Tervehtimistä jälkiehtoisella silmukalla.
public class Heippa2 {
   public static void main(String[] args) {
      // Laskuri ja kierrosten lukumäärä.
      int i = 0;
      int lkm = 3;
      // Tervehditään lkm kertaa.
      do {
         System.out.println("Heippa!");
         i = i + 1;
      }
      while (i < lkm);
   }
}
```



For-lause

- Lauseiden esiehtoinen toisto laskurimuuttujan avulla.
- Voidaan korvata helposti while-lauseella.
- Jos *n* = 1, aaltosulkeet voidaan jättää pois.
- Muista edelleen sisentää sisältö.

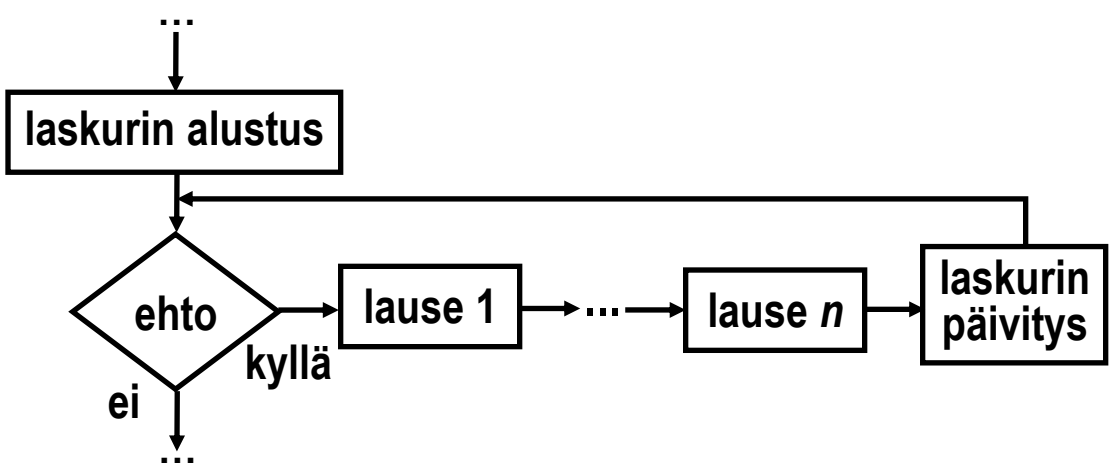
- Yleisesti:

for (laskurin alustus; ehto; laskurin päivitys) {

  lause 1; ... ...

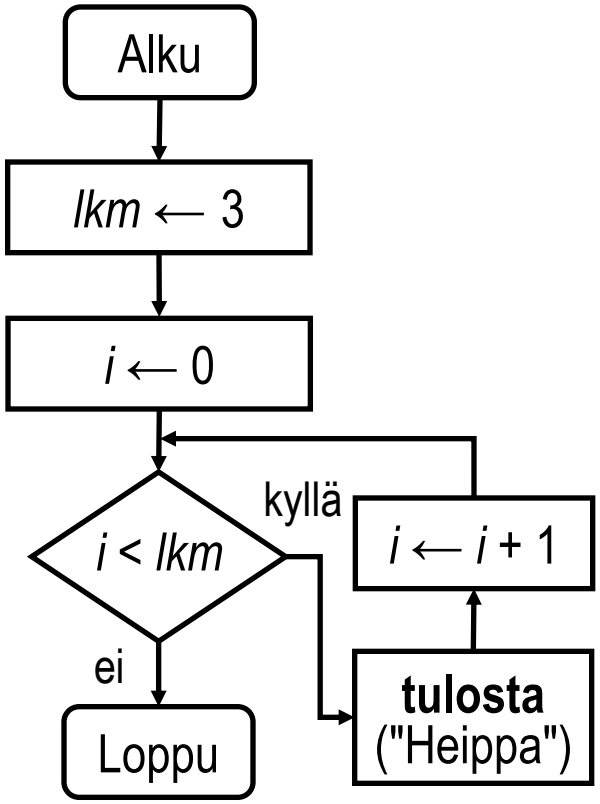
  lause *n*;

}



For-lause

```
// Tervehtimistä esiehtoisella silmukalla.
public class Heippa3 {
   public static void main(String[] args) {
      // Laskuri ja kierrosten lukumäärä.
      int i;
      int lkm = 3;
      // Tervehditään lkm kertaa.
      for (i = 0; i < lkm; i = i + 1)
         System.out.println("Heippa!");
   }
}
```



Vuokaaviona lähes kuin while-lause.

Tyypillisiä ohjelmointivirheitä

- Yleisin silmukoihin liittyvä ohjelmointivirhe on se, että toistojen todellinen lukumäärä poikkeaa tavalla tai toisella ajatellusta lukumäärästä.
- Tällaisen looginen virhe voi olla
  - hankalasti löydettävä, jos silmukan viimeinen kierros jää suorittamatta tai silmukkaa suoritetaan yhden ylimääräisen kerran tai
  - helposti löydettävä, mutta ohjelman suorituksen kannalta katastrofaalinen, jos ohjelma joutuu ikuiseen silmukkaan (eternal loop).

Kierroksen liian lyhyt tai pitkä silmukka

- Jos silmukka jää kierroksen “lyhyeksi” tai menee kierroksen “pitkäksi”, on todennäköistä, että laskurin alkuarvo ja silmukan lopetusehto eivät ole keskenään linjassa.
- Jos tavoitteena on esimerkiksi suorittaa *n* kierrosta alkuarvoa yksi käyttäen, ei lopetusehdoksi käy arvolle nolla sovitettu ehto, koska tällöin suoritetaan vain *n* - 1 kierrosta.
- Esimerkki:

```
int i = 1; int n = 3;
// Suoritetaan kaksi kierrosta kolmen asemasta.
while (i < n) {
   ...
   i = i + 1;
}
```

Kierroksen liian lyhyt tai pitkä silmukka

- Esimerkki: olkoon kierroksia *n* ja *i* laskurimuuttuja.

```
// n kierrosta.
for (i = 0; i < n; i = i + 1) ...

// n - 1 kierrosta.
for (i = 1; i < n; i = i + 1) ...

// n kierrosta.
for (i = 1; i <= n; i = i + 1) ...

// n + 1 kierrosta
for (i = 0; i <= n; i = i + 1) ...
```

Ikuinen silmukka

- Silmukan suoritus ei pääty koskaan virheellisen logiikan vuoksi.
- Kun Java-ohjelma joutuu ikuiseen silmukkaa, se voidaan pysäyttää CTRL-C-näppäinyhdistelmällä tai sulkemalla komentoikkuna.

- ```
  int i = 0;
  while (i < 10); {
    i = i + 1;
    System.out.println(i);
  }
  ```
- ```
  int i;
  for (i = 10; i > 0; i = i + 1)
    System.out.println(i);
  ```
- ```
  int i = 0;
  while (i < 10) {
    System.out.println(i);
  }
  ```
- ```
  char valinta; boolean jatketaan = true;
  do {
    ...
  }
  while (jatketaan = true);
  ```

Lippumuuttujat

- Silmukan suoritusta voidaan ohjata myös niin sanotun lippumuuttujan (flag tai sentinel) avulla.
  - Useimmiten boolean-tyyppiä.
- Hyödyllinen apuväline, kun toistojen määrää ei ole kiinnitetty eikä laskurimuuttujasta siten ole hyötyä.
- Käytetään while- ja do-while-lauseissa.
- Esimerkiksi tekstipohjaisten ohjelmien käyttöliittymässä on yleensä “pääsilmukka”, josta poistutaan vasta, kun käyttäjä antaa ohjelmalle lopetuskomennon jotain näppäintä painamalla.

Lippumuuttujat

- Esimerkki:

```
char valinta;
boolean jatketaan = true;
do {
   // Suoritetaan lauseita ja luetaan käyttäjän
   // komento valinta-muuttujaan.
   …
   // Asetetaan uusi arvo lippumuuttajalle, kun lopetetaan.
   if (valinta == LOPETA)
      jatketaan = false;
}
while (jatketaan);
```

- Boolean-tyyppistä lippua ei tarvitse verrata silmukan ehdossa **true**- tai **false**-arvoihin, koska lippu jo itsessään sisältää totuusarvon.

Break-lause

- Keskeyttää (sisimmän) silmukan suorituksen.
- Käytetään aina ehdollisena, koska silmukka pysähtyy välittömästi.
- Käyttöä syytä välttää, koska break-lause (**break**)
  - tekee koodista vaikeaselkoista ja
  - on korvattavissa lippumuuttujalla.
- Continue-lause (**continue**) pysäyttää kierroksen, mutta silmukan suoritusta jatketaan, mikäli kierroksia on jäljellä.

- // Erittäin huonoa koodia,
  // jossa käytetään ikuista
  // silmukkaa ja break-
  // lausetta.

```
while (true) {
   ...
   // Katkaistaan silmukka.
   if (...)
      break;
   ...
}
```

Sisäkkäiset silmukat

- Joissakin tehtävissä – kuten arvojen lajittelussa ja taulukoiden käsittelyssä – toistorakenteita yhdistetään siten, että toistorakenne sisältyy toiseen toistorakenteeseen.
- Tällainen rakenne tunnetaan sisäkkäisinä silmukoina (nested loops).
- Silmukoita voi olla sisäkkäin kaksi tai useampia.
  - Tällä kurssilla rajoitutaan pääasiassa yksinkertaisimpaan sisäkkäiseen toistorakenteeseen, jossa on yksi ulompi ja yksi sisempi silmukka.
- Jokaisella ulomman silmukan kierroksella suoritetaan kaikki sisemmän silmukan kierrokset.
  - Jos ulommassa silmukassa on m kierrosta ja sisemmässä silmukassa n kierrosta, niin sisempään silmukkaan liittyvät lauseet suoritetaan $m \cdot n$ kertaa.

Sisäkkäiset silmukat

```
// Tulostaa näytölle suorakaiteen.
public class SuorakaiteenTulostus {
   public static void main(String[] args) {
      // Vakiot rivien ja sarakkeiden lukumäärille sekä tulostusmerkille.
      final int RIVEJA = 3; final int SARAKKEITA = 7;
      final char MERKKI = '*';
      // Rivi- ja sarakelaskurit.
      int rivi; int sarake;
      // Tulostus kahden silmukan avulla.
      for (rivi = 0; rivi < RIVEJA; rivi = rivi + 1) {
         for (sarake = 0; sarake < SARAKKEITA; sarake = sarake + 1)
            System.out.print(MERKKI);
         System.out.println();
      }
   }
}
```

Sisäkkäiset silmukat

- Peräkkäiset ja sisäkkäiset silmukat ovat eri asia.
- Peräkkäiset silmukat ovat toisistaan “riippumattomia”, kun taas ulompien silmukoiden suoritus ohjaa sisempien silmukoiden suoritusta.

- Peräkkäisiä silmukoita:

```
for (i = 0; i < LKM; i = i+1) { ...
} ...
while (j > LKM) { ...
} ...
```

- Sisäkkäisiä silmukoita:

```
for (i = 0; i < A; i = i+1)
    for (j = 0; j < B; j = j+1) { ...
} ...
```

Sisäkkäiset silmukat

- Joskus ulompi silmukka voi sisältää useampia peräkkäisiä silmukoita, jolloin nämä silmukat suoritetaan kukin vuorollaan aina kullakin ulomman silmukan kierroksella.

```
// Pääsilmukka.
do { ...
    // 1. sisempi silmukka.
    for (i = 0; i < A; i = i + 1) { ...
    } ...
    // 2. sisempi silmukka.
    while (j > B) { ...
    } ...
}
while (valinta != LOPETUS);
```

13. Loogiset operaatiot

Sisällys

- Loogiset operaatiot AND, OR, XOR ja NOT.
- Operaatioiden ehdollisuus.
- Bittioperaatiot.
- Loogiset operaatiot ohjausrakenteissa.
- Loogiset operaatiot ja laskentajärjestys.

Yleistä

Operaatio	Lyhyesti	Merkintä
Negaatio (looginen ei)	NOT	!
Konjunktio (looginen ja)	AND	&& ja &
Disjunktio (looginen tai)	OR	|| ja |
Poissulkeva disjunktio	XOR	^

- Operandit totuusarvoja (**true**, **false**) tai ehtolausekkeita.
- Operaatioiden tulos myös **boolean**-tyyppiä.
- Operaatiot määritellään totuustauluina, joissa annetaan operandien arvojen yhdistelmät ja arvoyhdistelmän tulos.

AND (&&)

- Olkoon A ja B totuusarvoja tai totuusarvoisia lausekkeita.
- Lauseke A && B on tosi, vain kun A ja B ovat tosia.

A	B	A && B
false	**false**	**false**
false	**true**	**false**
true	**false**	**false**
true	**true**	**true**

- // Muuttujan y arvo epätosi.
  boolean x = **true**;
  boolean y = x && **false**;

- // Syödään karkkia vain,
  // jos on oikea päivä ja hampaat
  // käyttökelpoisia.
  if (karkkipaiva && reikia < 2)
  nameja = nameja - 1;

OR (||)

- Olkoon A ja B totuusarvoja tai totuusarvoisia lausekkeita.
- Lauseke A || B on tosi, kun lausekkeista toinen tai molemmat ovat tosia.

A	B	A || B
false	**false**	**false**
false	**true**	**true**
true	**false**	**true**
true	**true**	**true**

- // Muuttujan y arvo tosi.

```
boolean x = true;
boolean y = x || false;
```

- // Syödään karkkia lähes aina:

```
// karkkipäivänä ja/tai kun
// on yksi reikä hampaissa.
if (karkkipaiva || reikia < 2)
   nameja = nameja - 1;
```

XOR (^)

- Olkoon A ja B totuusarvoja tai totuusarvoisia lausekkeita.
- Lauseke A ^ B on tosi, kun vain toinen lausekkeista on tosi.

A	B	A ^ B
false	**false**	**false**
false	**true**	**true**
true	**false**	**true**
true	**true**	**false**

- // Muuttujan y arvo tosi.
  boolean x = **true**;
  boolean y = x ^ **false**;

- // Syödään karkkia,
  // kun ei ole karkkipäivä
  // ja hampaissa on nolla tai yksi
  // reikää tai kun on karkkipäivä
  // ja hampaissa on useita reikiä.
  if (karkkipaiva ^ reikia < 2)
  nameja = nameja - 1;

NOT (!)

- Olkoon A totuusarvo tai totuusarvoinen lauseke.
- Lauseke !A on tosi silloin, kun lauseke A on epätosi.
- Lauseke !A on epätosi silloin, kun lauseke A on tosi.

A	!A
true	**false**
false	**true**

- // Muuttujan y arvo epätosi.
  boolean x = **true**;
  boolean y = !x;

- // Jos hampaissa on reikiä neljän
  // tai useamman kappaleen verran,
  // niin "pääsee" juurihoitoon.
  // Huom! Oletuslaskujärjestystä
  // muutettu suluilla.
  if (!(reikia < 4))
  System.out.println("Juurihoito!");

Operaatioiden ehdollisuus

- Edellä määritellyt loogiset AND- (&&) ja OR- (||) operaatiot ovat ehdollisia: jos totuusarvo saadaan selville ensimmäisestä operandista, niin toista operadia ei arvioida.
- // Karkkipäivänä ei lisäreikiä.
  // Huom! Huonoa koodia.
  if (karkkipaiva || ++reikia < 2)
  nameja--;

- Ehdottomat AND- (&) ja OR- (|) -operaatiot laskevat molemmat lausekkeet. (Toimivat muuten kuin ehdolliset operaatiot.)
- Näin voidaan varmistaa, että ehtolausekkeessa annettu sivuvaikutus toteutuu.
- // Karkkipäivänäkin lisäreikiä.
  // Huom! Huonoa koodia.
  if (karkkipaiva | ++reikia < 2)
  nameja--;

Bittioperaatiot (~, &, | ja ^)

- AND, OR, XOR ja NOT voidaan kohdistaa totuus-arvojen asemasta bitteihin.
- Operandit ja operaation tulos kokonaislukuja.
- Esimerkki:

  byte k = 16;

  byte l = 17;

  $16_{10} = 00010000_2$

  $17_{10} = 00010001_2$
- ~00010000 = 11101111 = -17

- 00010000
& 00010001
00010000 = 16
- 00010000
| 00010001
00010001 = 17
- 00010000
^ 00010001
00000001 = 1
- Ei pidä käyttää, jollei tiedä, mitä tekee.
- Ei tarvita tällä kurssilla.

Ohjausrakenteissa

- Loogisilla operaatioilla yhdistetyistä lausekkeista voidaan muodostaa monimutkaisempia ehtoja, jotka vastaavat sisäkkäisiä valintarakenteita.
- Esimerkki:

```
if (x > 10)
  if (x < 20) { …
```

⇔

```
if ((10 < x) && (x < 20)) { …
```

- Sulut parantavat loogisen lausekkeen luettavuutta.

Ohjausrakenteissa

- Ehtojen yhdistäminen mahdollista myös silmukoissa.
- Esimerkiksi:

```
while ((i > 0) || (valinta == KYLLA)) { ...
```

- **Break**-lause voidaan korvata lippumuuttujan ja loogisen operaation avulla. Esimerkiksi:

```
do { …                          ⇔        do {
   if (ehto1)                                …
      break; …                           }
}                                        while (!ehto1 && ehto2);
while (ehto2);
```

Laskentajärjestys

1) NOT (!).
2) Aritmeettiset operaatiot.
3) Vertailut: ensin (<, <=, >, >=) sitten (==, !=).
4) Loogiset operaatiot järjestyksessä AND (&) XOR (^), OR (|), AND (&&) ja OR (||).
5) Lopuksi mahdollinen sijoitus.

- Vasemmalta oikealle.
- Laskentajärjestystä voi muuttaa suluilla.
- Sulkuja on myös joskus syytä käyttää selvyyden vuoksi.
- Huomaa, että lista ei sisällä kaikkia Java-kielen operaatioita.

Esimerkkejä

- **boolean** a = 1 != 1; // false
- **boolean** b = **true** | **false** ^ **true** && !**false**; // true
- **int** i = 1; **int** j = 2;

  boolean c = **true** ^ i == j && !**false** || 2 * i != j; // true
- **int** k = 4; **int** l = 4; **int** m = 8;

  if (2 * k < m || k == l && k != m) { ... // true
- // Edellinen lauseke selkeämpi näin.

  if ((2 * k < m) || (k == l) && (k != m)) { ...

Esimerkkejä

```
// Päätellään onko vuosi karkausvuosi.
public class Karkausvuosi {
    public static void main (String[] args) {
        int vuosi = 1900;
        // Karkausvuosi, jos vuosi on neljällä jaollinen ja ei ole vuosisata tai
        // on 400:lla jaollinen vuosisata. Sulut mukana selvyyden vuoksi.
        boolean karkausvuosi = ((vuosi % 4 == 0) &&
        (vuosi % 100 != 0)) || (vuosi % 400 == 0);
        if (karkausvuosi)
            System.out.println(vuosi + " on karkausvuosi.");
        else
            System.out.println(vuosi + " ei ole karkausvuosi.");
    }
}
```

14. Hyvä ohjelmointitapa

Yleistä

- Ohjelman elinkaari ei tyypillisesti pääty sen toteuttamiseen; ohjelmaa voidaan käyttää ja ylläpitää jopa vuosikymmeniä.
- Jotta koodin muuttaminen on mahdollista, sen on oltava muidenkin kuin tekijänsä ymmärrettävissä.
- Hyvää ohjelmointitapaa noudattamalla saadaan aikaiseksi ymmärrettäviä ja hallittavia ohjelmia.
- Perusasioita:
  - Nimeä tunnukset järkevästi.
  - Kommentoi riittävästi ja oikeissa paikoissa.
  - Sisennä koodia.
  - Rivitys: vältä liian pitkiä rivejä, käytä välirivejä ja käytä välilyöntejä riveillä.
  - Käytä vakioita.

Nimeä järkevästi

- Tunnusten (nimien) tulee olla järkeviä.
- Nimestä tulisi voida päätellä muuttujan tehtävä ohjelmassa ja muuttujan sisältämän tiedon luonne.
- Usein hyvä nimi on yhtä kuin riittävän pitkä nimi.
- Noudata nimeämiskäytäntöäsi johdonmukaisesti.
- Vakiintuneita käytäntöjä Javassa:
  - Muuttujien nimet alkavat pienellä kirjaimella.
    - Esimerkiksi: **double** keskiarvo;
  - Luokkien nimet alkavat isolla kirjaimella.
    - Esimerkiksi: **public class** HelloWorld { …
  - Vakiot kirjoitetaan isoin kirjaimin.
    - Esimerkiksi: **final char** EROTIN = '/';

Kommentoi

- Kommentit kannattaa kohdistaa erityisesti koodin keskeisiin osiin ja vaikeasti ymmärrettäviin osiin.
  - Pitemmässä ohjelmassa ei tarvitse kommentoida kaikkea.
- Ohjelman alkuun kannattaa kirjoittaa kommentti, josta käy ilmi mitä ohjelma tekee ja kuka ohjelman teki.

```
/*
 * Muutetaan suomenkielinen viesti morsekoodiksi ja päinvastoin.
 *
 * Lausekielinen ohjelmointi I, Jorma Laurikkala, jorma.laurikkala@uta.fi.
 *
 * Viimeksi muutettu 20.9.2015 15:11:45.
 *
 */
public class Morse { …
}
```

Sisennä

- Sisennys auttaa hahmottamaan kokonaisuuksia.
- Sisennyksellä osoitetaan lauseiden looginen ja kieliopillinen yhteenkuuluvuus.
- Välilyönnein tai tabulaattorilla:
  - Kukin sisennyksen taso ainakin kaksi välilyöntiä.
  - Sekä tavalliset lauseet että kommentit samalle tasolle.
  - Huomaa, että tabulaattorilla sisennettäessä koodi näyttää melko varmasti erilaiselta muissa editoreissa.
  - Välilyöntejä ja tabulaattoreita ei saa käyttää sekaisin.

Sisennä

- Käytännössä kaikkien koottujen lauseiden sisältö sisennetään.
- Tasot: ohjelman, *main*-operaation, ohjausrakenteen, sisemmän ohjausrakenteen koottu lause jne.
  - Myös ohjausrakenteeseen liittyvä yksittäinen lause sisennetään.
- Ole johdonmukainen sisennyksissä: kullakin tasolla aina sama määrä sisennystä.
- Esimerkiksi:

```
while (jatketaan) {
  // Tulostetaan
  System.out.println(…);
  …
  // Päivitetään lippumuuttuja.
  if (…) {
    jatketaan = false;
    …
  }
  else
    jatketaan = true;
}
```

Sisennä

- Sisäkkäiset **if-else**-rakenteet esitetään joskus tilanpuutteen vuoksi siten, että sisemmän lauseen **if**-osan otsikkorivi kirjoitetaan ulomman lauseen **else**-osan otsikkoriville.
- Tällöin rakenteen otsikkorivit ja koottujen lauseiden sisältö alkavat samalta tasolta.

- Esimerkki:

```
if (valinta == LISAA) {
   // Lisätään.
   ...
}
else if (valinta == HAE) {
   // Haetaan.
   ...
}
...
else {
   // Virhe.
   ...
}
```

Rivitys

- Osoita lauseiden looginen yhteenkuuluvuus kootun lauseen sisällä välirivejä käyttämällä.
- // Erota "teemoja" näin:

```
while (jatketaan) {
  // Luetaan syöte.
  System.out.println(…);
  int luku = In.readInt();

  // Tulostetaan
  System.out.println(…);
}
```

- Älä siis sisennä eri tasoille kootun lauseen sisällä, ellei kyseessä ole sisemmän kootun lauseen sisältö.
- // Ei _koskaan_ näin:

```
while (jatketaan) {
   // Luetaan syöte.
   System.out.println(…);
   int luku = In.readInt();
      // Tulostetaan
      System.out.println(…);
}
```

Rivitys

- Pitkät rivit ovat vaikeaselkoisia ja näkyvät vain osin pienessä ikkunassa.
- Jaa liian pitkä rivi kahdeksi lauseeksi kahdelle riville tai katkaise rivi, jolloin yksi lause on kahdella rivillä.
  - Riviä ei saa katkaista arvon tai tunnuksen keskeltä.
- Yli 100 merkin mittaiset rivit alkavat olla liian pitkiä.
- Paranna yksittäisten rivien luettavuutta käyttämällä välilyöntejä.
  - Esimerkiksi lause

    ```
    int i=(j+1)*2;
    ```

  on luettavampi muodossa

    ```
    int i = (j + 1) * 2;
    ```

Käytä vakioita

- Vakioiden tunnus kirjoitetaan ISOIN KIRJAIMIN.
- Esimerkki: **final** String LOPPU = "Ohjelma lopetettu.";
- Vakiot määritellään yleensä ohjelman alussa.
- Helpottavat ohjelmien ylläpitoa, esimerkiksi tilanteissa, joissa
  - arvo on suojattava muutoksilta,
  - sama arvo esiintyy ohjelmassa useassa kohtaa ja
  - ohjelmaa käytetään näppäimistöltä annettavilla komennoilla.

15. Lohkot

Sisällys

- Tutustutaan lohkoihin.
- Muuttujien ja vakioiden näkyvyys sekä elinikä erityisesti operaation lohkossa.
- Nimikonfliktit.
- Muuttujat operaation alussa vai myöhemmin?

Lohkot

- Aaltosulkeet ({ }) ovat tuttuja ohjausrakenteiden yhteydestä. Kaarisuluilla kootaan yhteen tiettyyn valintaan tai toistoon liittyvät lauseet kootuksi lauseeksi.
- Koottuja lauseita kutsutaan usein lohkoiksi (block), kun aaltosulje-parien ajatellaan kokoamisen asemasta jakavan ohjelmaa osiin.
- Lohkot voivat olla sisäkkäin ja peräkkäin.

```
if (ehto1) {
    if (ehto2) {
        lause 1; ...
        lause n;
    }
}

else {
    lause 1; ...
    lause m;
}
```

Lohkot

- Aaltosulkeita käytetään myös ohjelman (luokan) ja operaation (metodin) sisällön merkitsemiseen.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

- Ulompi lohko sisältää *HelloWorld*-ohjelman ja sisempi lohko *main*-operaation.
  - Operaation otsikon tunnus *args* kuuluu *main*-operaation lohkoon.

Lohkot

- Lohkon voi esitellä ilman avainsanaa tai otsikkoa pelkästään aaltosulkuja käyttämällä.
- Esimerkki:

```
{
    int i = 1;
}
```

- Joskus kätevää, mutta yleisesti ottaen tällaiseen ei ole tarvetta.

Näkyvyys

- Lohkot säätelevät muuttujien (ja vakioiden) elinikää, joka puolestaan ilmenee näkyvyytenä (visibility).
- Seuraavassa tarkastellaan näkyvyyttä vain operaatioiden sisällä, koska ohjelman (luokkien) mittakaavassa näkyvyyssäännöt ovat monimutkaisemmat.
- Koska vakiot pyritään esittelemään aina operaation alussa, keskitytään seuraavassa lähinnä muuttujiin.

Näkyvyys operaation sisällä

- Ulomman lohkon muuttuja näkyy sisempään lohkoon, mutta sisemmän lohkon muuttuja ei näy ulompaan lohkoon.
  - Ulomman lohkon muuttuja näkyy sisempään lohkoon vain, mikäli se on esitelty ennen sisempää lohkoa.

// Oikein.

```
int i = 0;
do {
  i = i + 1; …
}
while (i < 10);
i = 1;
```

// Väärin.

```
do {
  int i = 0; …
  i = i + 1; …
}
while (i < 10);
i = 1;
```

// Väärin.

```
do {
  i = i + 1; …
}
while (i < 10);
int i = 0;
i = 1;
```

Näkyvyys operaation sisällä

```
// Muuttuja x näkyy käyttöpaikkaansa.
public class Lohko1 {
   public static void main(String[] args) {
      int y = 1;
      int x = 0;
      if (y > 0) {
         x = 2;
         y = y + 1;
      }
   }
}
```

- Kun muuttuja x esitellään ulommassa eli operaation lohkossa, x näkyy myös sisemmässä eli **if**-lauseen lohkossa.

Näkyvyys operaation sisällä

```
// Muuttuja x ei näy käyttöpaikkaansa.
public class Lohko2 {
   public static void main(String[] args) {
      int y = 1;
      if (y > 0) {
         int x = 0;
         y = y + 1;
      }
      x = 2;
   }
}
```

Lohko2.java:9: cannot find symbol
symbol : variable x

- Kun muuttuja x esitellään sisemmässä eli **if**-lauseen lohkossa, x ei näy ulommassa eli operaation lohkossa. Tästä syystä oheinen ohjelma ei käänny.

Näkyvyys operaation sisällä

- Lohkossa esitelty muuttuja ei näy lohkoa seuraavaan lohkoon, mikäli lohkot ovat peräkkäisiä, vaikka lohkot ovat “samalla” tasolla.

```
// Väärin.
int i = 0;
while (i < 10) {
   int j = 0;
   i = i + 1; …
}
do {
   j = j + 1; …
}
while (j < 10);
```

Näkyvyys operaation sisällä

```
// Muuttuja x ei näy käyttöpaikkaansa.
public class Lohko3 {
   public static void main(String[] args) {
      int y = 1;
      if (y > 0) {
         int x = 0;  // Muuttujan x esittely
         y = y + 1;  // ensimmäisessä lohkossa.
      }
      while (y < 10) {
         y = y + 1;  // Virhe! Muuttuja x ei näy
         x = x + 1;  // toiseen lohkoon.
      }
   }
}
```

Elinikä

- Muuttujien (ja vakioiden) näkyvyys liittyy kiinteästi eliniän (lifetime) käsitteeseen.
- Muuttujan voidaan ajatella “syntyvän”, kun se esitellään. Tällöin muuttujalle varataan muistialue keskusmuistista.
  - Alkeis- ja viitetyyppisten muuttujien syntymä on kuitenkin hieman erilainen.
- Muuttujan elinikä vaihtelee ohjelman eri tasoilla, mutta operaation sisällä esittelylohko ja elinikä ovat erottamattomat.

Elinikä operaation sisällä

- Operaation mittakaavassa alkeistyyppinen muuttuja “elää” (on käytettävissä) niin kauan kuin ohjelman suoritus viipyy sen esittelylohkossa.
- Muuttuja “kuolee” (hävitetään), kun lohkon suoritus loppuu. Tällöin muuttujalle varattu muisti vapautuu eikä sen tunnus ole enää käytettävissä.
- Tästä syystä sisemmässä lohkossa esitelty muuttuja ei ole käytettävissä ulommassa lohkossa eikä edellisen lohkon muuttuja ole käytettävissä seuraavassa lohkossa.

```
public … main(…) {
…
// Esittely synnyttää
// muuttujan.
double paino = 80.5;
// Muuttuja elää.
…
// Muuttuja häviää,
// kun operaatio on
// suoritettu.
 }
```

Elinikä operaation sisällä

- Näkyvyyssääntöjä noudattamalla operaation sisällä voidaan esitellä samannimisiä muuttujia, koska edellinen muuttuja ehtii hävitä ennen seuraavan muuttujan esittelyä.
- Jos sisäkkäisissä lohkoissa “elää” yhtä aikaa samannimisiä muuttujia (tai vakioita), tapahtuu niin sanottu nimikonflikti (name conflict), joka estää ohjelman kääntämisen.
  - Nimikonflikti vältetään, jos ulomman lohkon muuttuja (tai vakio) esitellään vasta sisemmän lohkon jälkeen, koska sisempi muuttuja ehtii hävitä ennen ulompaa esittelyä.
  - Samannimisten muuttujien käyttö operaatiossa ei ole suositeltavaa, koska nimen jakavat muuttujat on helppo sekoittaa keskenään.

Nimikonflikti operaation sisällä

```
// Toisilleen näkyvät x-muuttujat aiheuttavat nimikonfliktin.
public class Lohko4 {
   public static void main(String[] args) {
      // Esittely pääohjelman lohkossa.
      int x = 0;
      if (x == 0) {
         // Esittely if-lauseen lohkossa ei onnistu,
         // koska operaation lohkossa on esitelty
         // samanniminen muuttuja ennen if-lohkoa!
         int x = 1;
         System.out.println(x);
      }
   }
}
```

```
Lohko4.java:10: x is already defined in main(java.lang.String[])
        int x = 1;
            ^
```

Vältetty nimikonflikti

```
// Kaksi x-muuttujaa mahtuu olemaan, koska vain yksi elää kerrallaan.
public class Lohko5 {
   public static void main(String[] args) {
      int y = 0;
      if (y == 0) {
         // Esittely if-lauseen lohkossa onnistuu, koska operaation lohkossa on
         // esitelty samanniminen muuttuja if-lohkon jälkeen.
         int x = 1;
         // Tulostaa if-lauseen muuttujan arvon (1).
         System.out.println(x);
      }
      // Esittely pääohjelman lohkossa.
      int x = 0;
      // Tulostaa operaation muuttujan arvon (0).
      System.out.println(x);
   }
}
```

Muuttujien paikka operaatiossa?

- Esittely operaation alussa.
  - Perinteinen tapa, jota on käytetty toistaiseksi näkyvyysongelmien välttämiseksi.
  - Näkyvyys taattu; muuttuja saatavilla kaikkialla operaation lohkossa.
  - Muuttujat löytyvät helposti yhteen paikkaan koottuina.
  - Muuttujan esittely ja käyttöpaikka saattavat olla kaukana toisistaan.
    - Koodia vaikeampi seurata.
    - Ohjelmasta pitempi kuin tarpeen, koska alustaminen ei ole aina mahdollista esittelyn yhteydessä.
- Esittely käyttöpaikan lähellä.
  - Modernimpi tapa.
  - Koodi selkeämpää.
  - Esittely ja alustus voidaan tehdä yhdellä lauseella.
  - Näkyvyyden kanssa pitää olla tarkkana.
  - Muuttujat hajallaan koodin seassa.
  - Joissakin kielissä vaarana tehottomuus silmukan lohkon sisäisissä esittelyissä.
    - Java-kääntäjä osaa optimoida jonkin verran.

16. Ohjelmoinnin tekniikkaa

Sisällys

- For-lause lyhemmin.
- Vaihtoehtoisia merkintöjä aritmeettisille lauseille.
- Useiden muuttujien esittely ja alustaminen yhdellä lauseella.
- If-else-lause vaihtoehtoisesti merkiten.
- Ohjelman optimointi.

For-lause lyhemmin

- Laskurin esittely voidaan tehdä alustuksen yhteydessä:

```
for (laskurin esittely ja alustus; ehto; laskurin päivitys) {
        lause 1; ...
        lause n;
}
```

- Esimerkki:

```
// Tervehditään x kertaa.
for (int i = 0; i < x; i = i + 1)
    System.out.println("Heippa!");
```

- Laskuri käytettävissä nyt vain silmukan sisällä.

Aritmetiikkaa toisin merkiten

- Aritmeettisten operaattoreiden (+, -, *, /, %) tuloksen sijoitus voidaan merkitä lyhyemmin yhdistetyillä sijoitusoperaattorilla **+=**, **-=**, ***=**, **/=** ja **%=**.
- Yleisesti:

  muuttuja = muuttuja operaatio arvo; ⇔
  muuttuja operaatio= arvo;

  missä arvo voi olla myös muuttujan, vakion tai lausekkeen arvo.
- Esimerkki:

```
int i = 10;
i = i + 2;      ⇔    i += 2;      // 12
i = i - 2;      ⇔    i -= 2;      // 10
i = i * 2;      ⇔    i *= 2;      // 20
```

Aritmetiikkaa toisin merkiten

- Esimerkki: i = i / 2; ⇔ i /= 2; // 10
  i = i % 2; ⇔ i %= 2;// 0
- Esimerkki: **int** i = 10;
  i = i + 13 / 3; ⇔ i += 13 / 3; // 14
- Lyhyemmillä merkinnöillä saadaan helposti aikaiseksi vaikeaselkoista koodia.
- Esimerkki: **int** i = 10;
  int j = 2;
  i *= i + j; // 120

  Edellinen lause ei ole i = i * i + j; vaan i = i * **(**i + j**)**;

Aritmetiikkaa toisin merkiten

- Muuttujan arvoa voidaan kasvattaa yhdellä (**++**) tai vähentää (**--**) yhdellä nopealla, mutta vaarallisella tavalla.
  - Erittäin kätevää erityisesti silmukoiden yhteydessä.
- Yleisesti:

  muuttuja = muuttuja + 1; ⇔ **++muuttuja;** tai **muuttuja++;**

  muuttuja = muuttuja - 1; ⇔ **--muuttuja;** tai **muuttuja--;**
- Esimerkki:

```
int i = 1;
i = i + 1;      ⇔    i++;  tai  ++i;
```

- Esimerkki:

```
// Tervehditään x kertaa.
for (int i = 0; i < x; i++)
   System.out.println("Heippa!");
```

Aritmetiikkaa toisin merkiten

- **++** ja **--** operaattoreiden paikka säätelee muuttujan arvon lisäystä tai vähennystä suhteessa lausekkeeseen, jossa operaatio esiintyy.
- Muuttujaa edeltävä operaatio suoritetaan ennen lausekkeen laskemista.
- Muuttujan jälkeen oleva operaatio suoritetaan lausekkeen laskemisen jälkeen.
- Jos lauseen ainoa operaatio on **++** tai **--**, niin lopputulos on sama.

Aritmetiikkaa toisin merkiten

- Esimerkki:

```
int i = 1;
i++;   ⇔   ++i;   // 2
i--;   ⇔   --i;   // 1
```

- Operaattoreita +=, -=, *=, /=, %=, ++ ja -- voidaan ajatella lausekelauseiksi: niitä voi käyttää sekä lauseina että lausekkeina.
- Esimerkki:

```
i++;        // lause
i++ * 2     // lauseke
```

Aritmetiikkaa toisin merkiten

- Esimerkki:

```
int i = 1;
// Kasvatetaan ensin muuttujaa i yhdellä
// ja kerrotaan vasta sitten.
int j = ++i * 2; // i == 2, j == 4
```

- Esimerkki:

```
int i = 1;
// Kerrotaan ensin ja kasvatetaan
// muuttujaa i yhdellä kertomisen jälkeen.
int k = i++ * 2;          // i == 1, k == 2
```

Aritmetiikkaa toisin merkiten

- Esimerkki:

```
final int YLARAJA = 3;
int i = 0;
// Laskurin arvo muuttuu ennen lauseketta.
while (++i < YLARAJA)
    System.out.print(i + " ");    // 1 2
System.out.println();
// Laskurin arvo muuttuu lausekkeen jälkeen.
int j = 0;
while (j++ < YLARAJA)                    // 1 2 3
    System.out.print(j + " ");
```

Esittely ja alustaminen yhdellä lauseella

- Samassa lauseessa voidaan esitellä ja alustaa useita samantyyppisiä muuttujia pilkulla erottaen.
- Yhtä lausetta käyttäen voi ohjelmoida hieman nopeammin, mutta erillisillä lauseilla ohjelma on usein selkeämpi.

- Esim.

```
// Esitellään ja alustetaan.
int i;
int j;
int k = 1;
int l = 2;
int m;
```

⇔

```
// Esitellään ja alustetaan.
int i, j, k = 1, l = 2, m;
```

Esittely ja alustaminen yhdellä lauseella

- Alustaminen on mahdollista merkitä lyhemmin, kun samantyyppisen muuttujien alkuarvo on sama.
- Esimerkki:

```
double a = 3.14, b = 3.14, c = 3.14;
```

⇔

```
double a, b, c;
a = b = c = 3.14;
```

- Javassa ei voi esitellä ja alustaa samaan arvoon yhdessä lauseessa.
  - Esimerkiksi lause **double** a = b = c = 3.14; on virheellinen, koska muuttujia b ja c ei ole esitelty.

If-else-lause toisin merkiten

- Yksinkertaiseen arvonvalintaan kirjoitettu **if-else**-lause voidaan esittää ehto-operaattorin (**?:**, conditional operator) avulla.
- Yleisesti: **ehtolauseke ? lauseke 1 : lauseke 2**
- Javan ainoa kolmioperadinen operaattori.
- Jos ehto on tosi (**true**), suoritetaan ensimmäinen lauseke ja palautetaan sen arvo.
- Jos ehto on epätosi (**false**), suoritetaan toinen lauseke ja palautetaan sen arvo.

If-else-lause toisin merkiten

- Esimerkki: kahden luvun minimin päättely if-else-lauseella.

```
// Apumuuttuja minimille.
int min;
// Vertailtavat luvut.
int luku1 = 1;
int luku2 = 2;
// Minimin päättely.
if (luku1 < luku2)
   min = luku1;
else
   min = luku2;
```

- Esimerkki: kahden luvun minimin päättely ehto-operaattorilla.

```
// Muuttujat minimille
// ja vertailtaville luvuille.
int min;
int luku1 = 1, luku2 = 2;
// Minimin päättely.
min = luku1 < luku2 ? luku1 : luku2;
```

If-else-lause toisin merkiten

- Esimerkki: lämpötilan luonnehdinta kahdella tavalla.

```
// Lämpötila Celsius-asteina.
double lampo = 24.2;
// Päätellään if-else-lauseen avulla.
if (lampo > 25)
   System.out.println("Helle");
else
   System.out.println("Kylmempää");
// Päätellään ehto-operaattorin avulla.
System.out.println(lampo > 25 ? "Helle" : "Kylmempää");
```

If-else-lause toisin merkiten

- Ehto-operaatio joskus hankalasti ymmärrettävissä, joten sen käyttöä syytä välttää lausekkeiden osana.
- Melko heikko operaatio ja tästä syystä suljetaan usein lausekkeissa sulkujen sisään:
  - Aritmeettisia, vertailu- ja loogisia operaatioita heikompi.
  - Sijoitusoperaatiota ja yhdistettyä sijoitusta vahvempi.

Ohjelman optimointi

- Optimoimalla pyritään yleensä nopeuttamaan ohjelman suoritusta tai vähentämään muistin käyttöä.
  - Nopeuttaminen voi vaatia lisää muistia tai päinvastoin.
  - Pitkälle optimoitu ohjelma on vaikeaselkoinen.
- Perustekniikoita:
  - Pysäytä silmukka heti, kun on selvää ettei lisälaskentaa tarvita.
  - Kutsu laskennallisesti raskaita operaatioita harvoin.
  - Vältä tarpeettomia muistivarauksia.
- Kääntäjät osaavat optimoida lähdekoodin pieniä yksityiskohtia hyvin.

Ohjelman optimointi

- Optimoimaton ohjelma:

```
// Silmukan ehdossa
// kutsutaan tarpeettomasti
// (raskasta) operaatiota.
while (i < laskeYlaraja()) {
   ...
}
```

- Optimoitu ohjelma:

```
// Lasketaan yläraja
// ennen silmukkaa,
// jolloin ohjelma nopeutuu.
int ylaraja = laskeYlaraja();
while (i < ylaraja) {
   ...
}
```

17. Javan omat luokat

Sisällys

- Application Programming Interface (API).
- Pakkaukset.
- Merkkijonoluokka *String*.
- *Math*-luokka.
- Käärelуokat.

Java API

- Java-kielen Application Programming Interface (API) on kokoelma Javan omia luokkia, joista osa on valmiiksi ohjelmoitu ja osa määrittelee rajapintoja.
  - API-dokumentaatio on luettavissa Internetissä.
- API on jaettu pienempiin kokonaisuuksiin eli pakkauksiin (package), koska Javan omia luokkia on satoja.
- Pakkaukset sijaitsevat nimeään vastaavissa alihakemistoissa: esimerkiksi *java.lang*-paketin luokat ovat Java-ympäristön alihakemistossa *java\lang*

Pakkaukset

- Pakkauksen luokat tuodaan ohjelman käyttöön **import**-lauseella, jolla voidaan sisällyttää useampia pakkauksia.
- **import**-lauseet lisätään tiedoston alkuun ennen luokkaa.
- // Otetaan käyttöön java.util-pakkauksen kaikki luokat.

  import java.util.*;

  public class PakettiTesti { …
- *java.lang*-pakkauksen luokkia (esimerkiksi *String* ja *Math*) voidaan käyttää suoraan ilman eri pyyntöä.
- Tällä kurssilla riittää tietää mitä pakkauksen ovat – pakkauksia ei tarvitse osata tuoda ohjelmaan tai tehdä itse.

String-luokka

- Vaikka merkkijonot ovat *String*-luokan olioita, tämän tyyppiset tunnukset voidaan esitellä, ja niihin voidaan myös sijoittaa, alkeistyyppien tapaan.
- Esimerkiksi:
```
String tervTre = "Moro!";
String tervKuo = new String("Päevee!");
```
- Merkkijonoarvot suljetaan lainausmerkkeihin.
- Pienet ja isot kirjaimet ovat eri asia.
- Merkkijonoihin voidaan yhdistää **+**-operaatiolla alkeistyyppisiä arvoja ja merkkijonoja.
- Luokan palvelut ovat saatavilla ilman **import**-lausetta.

String-luokka

- Koska *String*-luokan operaatiot eivät ole osa omaa ohjelmaa (luokkaa), on niitä kutsuttava kutsuttava pistenotaatiolla.
- Joitain *String*-luokan operaatioista kutsutaan *In*-luokan operaatioiden tapaan luokan nimen kautta.
- Useimpia operaatioita kutsutaan olion (muuttujan) nimen kautta, jolloin operaatio kohdistuu olioon.
- Esimerkki: String mjono = "ABC - kissa kävelee.";

```
// Pyydetään merkkijonoa kertomaan pituutensa.
int jononPituus = mjono.length();   // 20
// Muutetaan kokonaisluku merkkijonoksi.
String lukuJonona = String.valueOf(123);
```

String-luokka

- Merkkijononon merkkeihin viitataan nollasta alkavalla indeksillä. Ensimmäisen kirjaimen indeksiarvo on 0, toisen 1 ja viimeisen *n* – 1, missä *n* on jonon pituus.
- Esimerkki: String kielenNimi = "Java";

Indeksi	0	1	2	3
Merkki	J	a	v	a

- Indeksiarvoja annetaan monien *String*-luokan operaatioiden parametreiksi.
- Javassa merkkijonot ja taulukot ovat eri asia.

Joitakin *String*-luokan operaatioita

- Operaatio length() palauttaa merkkijonon pituuden (**int**).
- Esimerkki: String kielenNimi = "Java";

  int jononPituus = kielenNimi.length(); // 4
- Tietty merkkijonon kirjain voidaan lukea charAt(**int** i)-opraatiolla, missä parametri i on indeksiarvo ja operaation tuloksen tyyppi on **char**.
- Esimerkki: String kielenNimi = "Java";

  char toinenMerkki = mjono.charAt(1); // 'a'

Joitakin *String*-luokan operaatioita

- Operaatio indexOf(**int** c) palauttaa merkin c ensimmäisen esiintymän indeksiarvon merkkijonon alusta lukien.
  - Jos merkkijono ei sisällä merkkiä, paluuarvo on -1.
  - Java muuttaa parametrin tyypin automaattisesti merkistä luvuksi.
- Esimerkki:

```
String kielenNimi = "Java";
// Merkkijonossa on kaksi pientä a-kirjainta,
// joista ensimmäinen on toisessa paikassa.
int inda = kielenNimi.indexOf('a'); // 1
// Merkkijonossa ei ole pientä q-kirjainta.
int indq = kielenNimi.indexOf('q'); // -1
```

Joitakin *String*-luokan operaatioita

- Operaation equals(String s) tutkii ovatko merkkijonojen merkit samat ja kertoo vertailun tuloksen **boolean**-tyyppisellä paluuarvolla.
- Yhtä- ja erisuuruusoperaatiot (==, !=) eivät sovellu merkkijonomuuttujien vertailuun, koska kyseessä ovat oliot.
- Esimerkki:

```
String mjono1 = new String("Java");
String mjono2 = new String("Java");
// true
boolean samat1 = mjono1.equals(mjono2);
// false
boolean samat2 = mjono1 == mjono2;
```

Joitakin *String*-luokan operaatioita

- Operaatio valueOf(**int** c) muuntaa alkeistyyppisen parametrinsa merkkijonoksi.
- Operaatio on kuormitettu: parametri voi olla esimerkiksi **int**- tai **double**-tyyppinen.
- Esimerkki: String klmjono = String.valueOf(123);
  String llmjono = String.valueOf(1.23);
- *String*-tyyppistä merkkijonoa ei voi muuttaa.
  - Ei operaatiota esimerkiksi yksittäisen merkin muuttamiseen.
  - *StringBuffer*- ja *StringBuilder*-tyyppiset jonot muuttuvat.

String-luokan operaatiot toimessa

```
public class VaihdaKirjain {
   public static void main(String[] args) {
      String sana = "Saari";                    // Muutettava merkkijono.
      char vanhaMerkki = 'a';                   // Vanha merkki.
      char uusiMerkki = 'i';                    // Uusi merkki.
      String apu = "";                          // Väliaikainen muuttuja.
      for (int i = 0; i < sana.length(); i = i + 1)
         if (sana.charAt(i) == vanhaMerkki)  // Löytyi vaihdettava merkki.
            apu = apu + uusiMerkki;
         else                                   // Jokin muu merkki.
            apu = apu + sana.charAt(i);
      sana = apu;
      System.out.println(sana);
   }
}
```

Math-luokka

- Operaatioita kutsutaan luokan nimen kautta.
- Esimerkiksi: // Satunnaisluku väliltä [0, 1[.

```
double satluku = Math.random();
```

- Luokan operaatiot ovat kuormitettuja.
  - Esimerkiksi parametrinsa itseisarvon palauttavar abs-operaatio hyväksyy **double**-, **float**-, **int**- tai **long**-tyyppisiä parametreja.
- Esimerkki:

```
double lluku = -1.23;
lluku = Math.abs(lluku); // 1.23
int kluku = -10;
System.out.println(Math.abs(kluku)); // 10
```

Joitakin *Math*-luokan operaatioita

min(a, b)	Palauttaa pienemmän parametreistaan a ja b.
max(a, b)	Palauttaa suuremman parametreistaan a ja b.
round(a)	Pyöristää liukulukutyyppisen parametrinsa. Palauttaa pyöristetyn arvon.
pow(a, b)	Potenssiin korotus a^b . Palauttaa tuloksen.
sqrt(a)	Neliöjuuren lasku $\sqrt{a}$. Palauttaa tuloksen.

18. Testaus

Testaus

- Testauksella pyritään löytämään virheitä, jotka sitten korjataan.
- Yksittäinen testi on yleensä ohjelman suoritus (tietyillä syötteillä).
- Kun virheet on korjattu, kaikki testit uusitaan. Vanhojen virheiden korjaus voi tuottaa uusia virheitä.
- Vaikka ohjelma läpäisee testit, se ei välttämättä toimi oikein. Siinä voi olla vikoja, joita suoritettavaksi valitut testit eivät paljasta.
- Voidaan ja osin pitäisikin suorittaa ohjelmoinnin kuluessa.

Testaus

- Testi voidaan suunnitella sen perusteella, mitä ohjelmalta on vaadittu: yleensä pohjana on jokin kirjallinen dokumentti ohjelmalle asetetuista vaatimuksista.
  - Niin sanottu musta laatikko (black box) -testaus, jossa ohjelman sisältöä ei tunneta.
- Testi voidaan myös suunnitella ohjelman toteutuksen jälkeen siten, että käytetään hyväksi tietoa sen toteutuksesta.
  - Voidaan esimerkiksi tietää, että jokin erityinen syötearvo on erikoistapaus.
  - Niin sanottu valkoinen laatikko (white box) -testaus, jossa ohjelman sisältö tunnetaan.

Testaus

- Kaikkia mahdollisia syötteitä ei voida testata, niitä on yleensä aivan liikaa.
- Tyypilliset oikeelliset syötteet.
- Raja-arvot ja lähes oikeelliset syötteet.
- Virheelliset syötteet, joista ohjelman olisi hyvä toipua järjellisesti.
- Testausta voidaan automatisoida.

Lausekielinen ohjelmointi II

- Harjoitustöiden toiminnallisuus testataan automaattisesti WETOssa, opettajat tarkistavat töiden rakenteen ja tyylin.
- Harjoitustöiden automaattinen testaus mustavalkoista.
  - Työn tulee toimia tehtävänannon mukaisesti.
  - Testaajalla ennakkotietoa tyypillisestä ratkaisusta.
- Testaus jaettu julkiseen ja salaiseen osuuteen.
  - Julkisten testien läpäisy antaa vahvoja viitteitä salaisten testien läpäisystä, mutta ei täyttä varmuutta tästä.
- Kussakin testissä verrataan tiedostoihin ohjattuja harjoitustyö- ja malliohjelman tulosteita.
  - Testi hyväksytään, jos tiedostot ovat merkilleen samat.

19. Tietovirrat

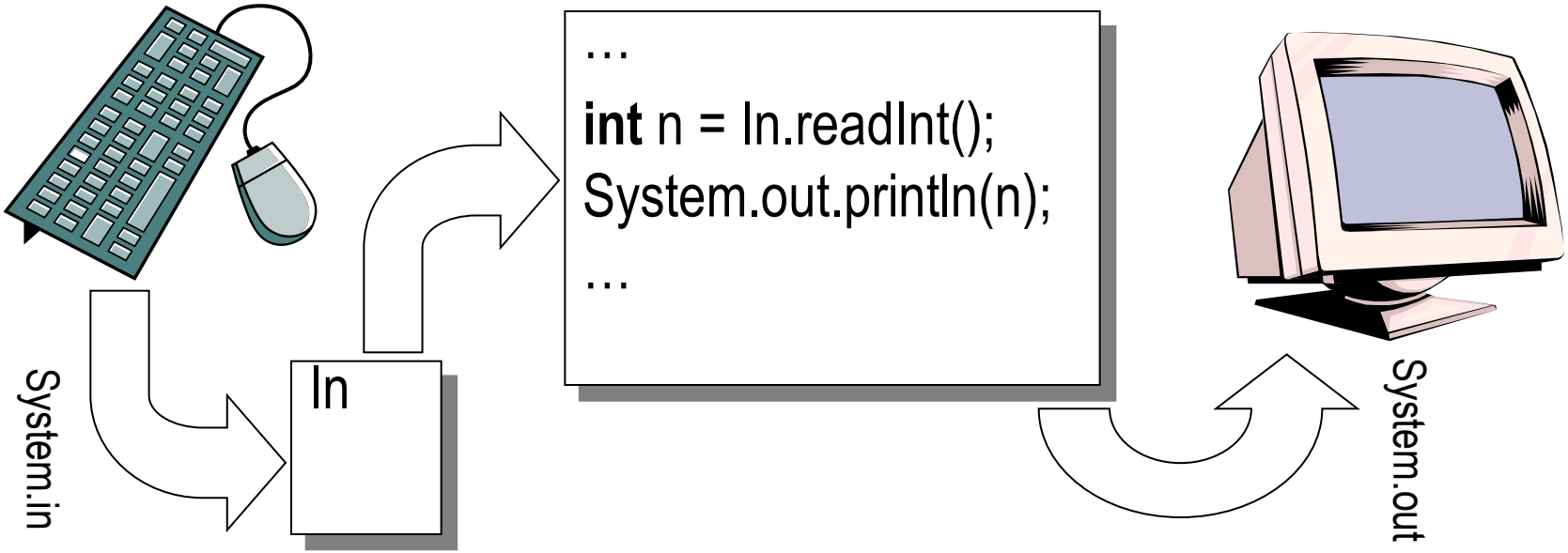
Sisällys

- Tietovirrat.
- Tietovirrat ja *In*-luokka.
- Tietovirtojen uudelleenohjaus.
- Uudelleenohjaus testauksessa.

Tietovirrat

- Usein ajatellaan abstraktisti, että ohjelma kommunikoi ympäristönsä kanssa tietovirtojen (data stream) avulla.
- Tyypillisesti käytettävissä ovat standarditulos-, standardisyöte- ja -standardivirhevirrat.
- Tulos- ja virhevirrat liittyvät oletusarvoisesti näyttöön ja syötevirta näppäimistöön.
- Javassa standardivirtojen tunnukset ovat System.out, System.in ja System.err.

Tietovirrat



- System.out-virta on tuttu tulostamisoperaatioiden yhteydestä.
- *In*-luokan operaatiot liittyvät System.in-virtaan.

Uudelleenohjaus

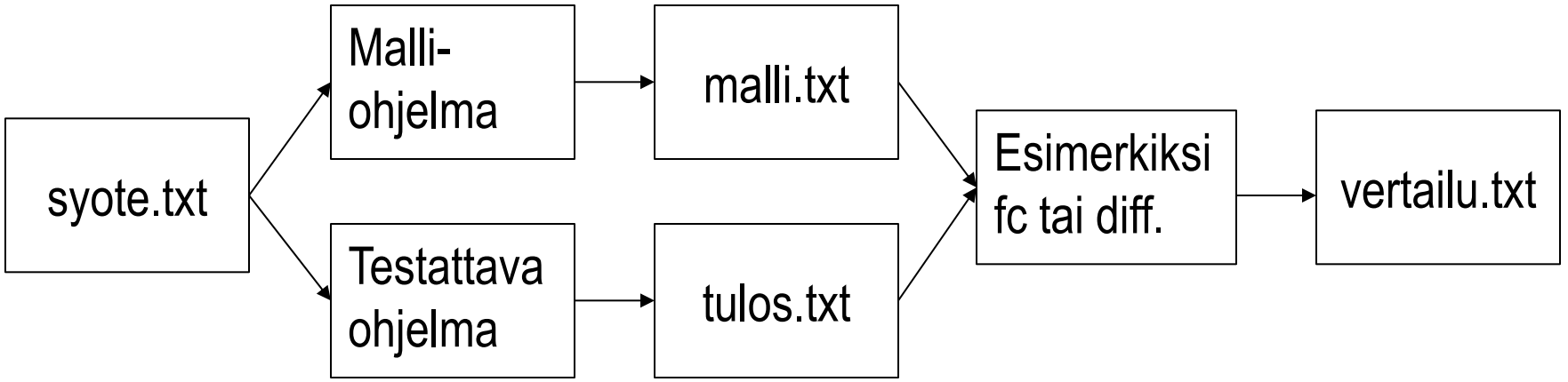
- Komentoikkunassa standardisyöte- ja standarditulostevirrat voidaan liittää väliaikaisesti tiedostoihin ohjausmerkeillä.
- Uudelleenohjaus toimii Windows- ja UNIX-komentoikkunoissa.
- Pienempi kuin -merkillä (**<**) ohjelma lukee syötteet tiedostosta näppäimistön asemasta.
- Esimerkiksi: **java OmaOhjelma < syote.txt**
- Syötetiedostossa kukin syöte on omalla rivillään.

Uudelleenohjaus

- Suurempi kuin -merkillä (**>**) ohjelma tulostaa tiedostoon näytön asemasta.
- Esimerkiksi: **java OmaOhjelma > tulos.txt**
- Kannattaa tarkastaa, että tiedostossa ei ole mitään tärkeää, koska tiedoston vanha sisältö menetetään!
- Ohjausmerkkejä voidaan myös käyttää yhdessä, jolloin syötteet tulevat tiedostosta ja tulosteet menevät tiedostoon.
- Esimerkiksi: **java OmaOhjelma < syote.txt > tulos.txt**

Testaus

- Ohjelmia voidaan testata vertailemalla niiden tulosteita.
- Tulosteiden vertailu voi olla silmämääräistä tai ohjelmallista.
- Eräs keino ohjelmalliseen vertailuun on ohjata testin syötteet tiedostosta ohjelmille, ohjata tulosteet tiedostoihin ja vertailla tiedostoja tähän sopivalla komennolla.
  - Lausekielinen ohjelmointi II -kurssin harjoitustyöt testataan näin.



1. Omat operaatiot

Sisällys

- Yleistä operaatioista.
- Mihin operaatioita tarvitaan?
- Oman operaation määrittely.
  - Yleisesti, nimeäminen ja hyvä ohjelmointitapa, määreet, parametrit ja näkyvyys.
- *HelloWorld*-ohjelma tarkemmin.
- Oman operaation paikka ohjelmassa, operaation kutsuminen ja paluuarvo esimerkkien avulla tarkasteltuna.
- Operaatioiden kutsuminen yleisesti.

Yleistä operaatioista

- Operaatio (metodi, aliohjelma, funktio jne.) eristää jonkin luokan osan nimettyyn lohkoon.
  - *Main*-operaatio tunnistetaan nimestä, johon liittyy lohko.
- Operaatiota kutsutaan luokan ulkopuolelta operaation nimellä, pistenotaatiota käyttäen.
- Kaarisulkupari erottaa operaation ja muuttujan tunnukset.
- Operaatiolle voidaan tarjota tietoja eli parametreja.
- Kutsun kohdalla ohjelman suoritus siirtyy operaatioon, josta palataan kutsukohtaa seuraavaan ohjelman vaiheeseen.
- Esimerkiksi: System.out.println("Moikka!");

Yleistä operaatioista

- Operaatio voi palauttaa arvon.
- Esimerkiksi: **int** luku = In.readInt();
- Näkyvyyssääntöjen mukaan operaation lohkossa esitellyt tunnukset eivät näy operaation ulkopuolelle.
- Operaatiotkaan eivät välttämättä näy luokan ulkopuolelle.
  - Operaatio voidaan kätkeä tai julkaista näkyvyysmääreillä.
  - *Main*-operaatio on julkinen (**public**), jotta Java-tulkki voi kutsua sitä.

Mihin operaatioita tarvitaan?

- Operaatioilla pitkä ohjelma voidaan jakaa “palasiin”, jolloin koodia on helpompi hallita, ymmärtää ja ylläpitää.
  - Monimutkaisuuden hallinta modulaarisella ohjelmoinnilla.
- Ohjelman eri kohdissa samana toistuva osa voidaan sijoittaa operaatioon, jolloin koodia ei tarvitse “monistaa” eri kohtiin, vaan pelkkä operaation kutsu riittää ohjelman osan suoritukseen.
  - Ohjelman ylläpito helpottuu.
- Mahdollistavat tiedonkätkennän.
  - Tarkemmin Olio-ohjelmoinnin perusteet -kurssilla.

Operaatioiden määrittely

- Koostuu otsikosta (header, signature) ja rungosta (body):

```
// Otsikossa määreitä, operaation nimi (tunnus) ja parametrilista.
määreet nimi(parametrilista) {
   // Runko on koottu lause (lohko), joka kirjoitetaan
   // luokkamäärittelyn sisään. Sulkeita ei voi jättää pois.
}
```

-
```
// Tervehditään ohjelman käyttäjää.
public static void tulostaTervehdys() {
    System.out.println(Moi!); …
}
```

Operaatioiden määrittely: nimet

- Nimet (tunnukset) ovat yleensä muuttujien nimiä pitempiä.
  - Esimerkiksi: *syote* ja *tarkistaSyote*.
- Kirjoitetaan samoin kuin muuttujat.
  - Nimi aloitetaan pienellä alkukirjaimella.
  - Useasta sanasta koostuvassa nimessä ensimmäinen sana pienellä ja muut isolla alkukirjaimella. (Yhdyssanat poikkeus.)
- Operaatiot kuvaavat toimintoja – tunnuksessa on usein käskymuotoinen verbi.
- Hyvä ohjelmointitapa koskee myös operaatioita.
  - Otsikkoon liittyvä kommentti kertoo suppeimmillaan mitä operaatio tekee.
  - Koottu lause sisennetään ja sen sisältö kommentoidaan.

Operaatioiden määrittely: määreet

- Määreillä otetaan kantaa muun muassa operaation näkyvyyteen ja operaation mahdollisesti palauttamiin tietoihin.
- **Private**-määreellä operaatio on yksityinen eli sitä voi kutsua vain luokan omissa operaatioissa.
- **Public**-määreellä operaatio on julkinen. Sitä voidaan kutsua pistenotaatiolla myös luokan ulkopuolelta.
- Arvon palauttavalle operaatiolle määritellään tyyppi, joka voi olla mikä tahansa Javan tietotyyppi.
  - Otsikon kommentissa selitetään minkä tiedon paluuarvo sisältää.
- Jos operaatiolla ei ole paluuarvoa, tietotyypin asemasta käytetään **void**-määrettä.
- Tällä kurssilla tarvitaan **static**-määre, jotta ohjelma kääntyisi.

Operaatioiden määrittely: parametrit

- Parametrilista määrittelee operaatiolle välitettävät tiedot.
  - Lista pilkuin erotettuja parametreja, jotka nimetään ja esitellään kuten muuttujat.
  - Toisinaan käytetään lyhyitä tunnuksia, koska parametrien merkitys tulee selittää otsikkoon liittyvässä kommentissa.
  - Pelkillä suluilla ilmaistaan ettei operaatiolle välitetä tietoja.
- /* Korottaa kantaluvun a potenssiin b ja palauttaa tuloksen.
  */

```
public static int laskePotenssi(int a, int b) {
  …
}
```

Operaatioiden määrittely: näkyvyys

- Parametrit ja muuttujat eivät näy muihin operaatioihin.
- Mikäli operaation parametrilla ja muuttujalla on sama nimi, tapahtuu nimikonflikti, eikä ohjelma käänny.
- // Korottaa kantaluvun a potenssiin b ja palauttaa tuloksen.

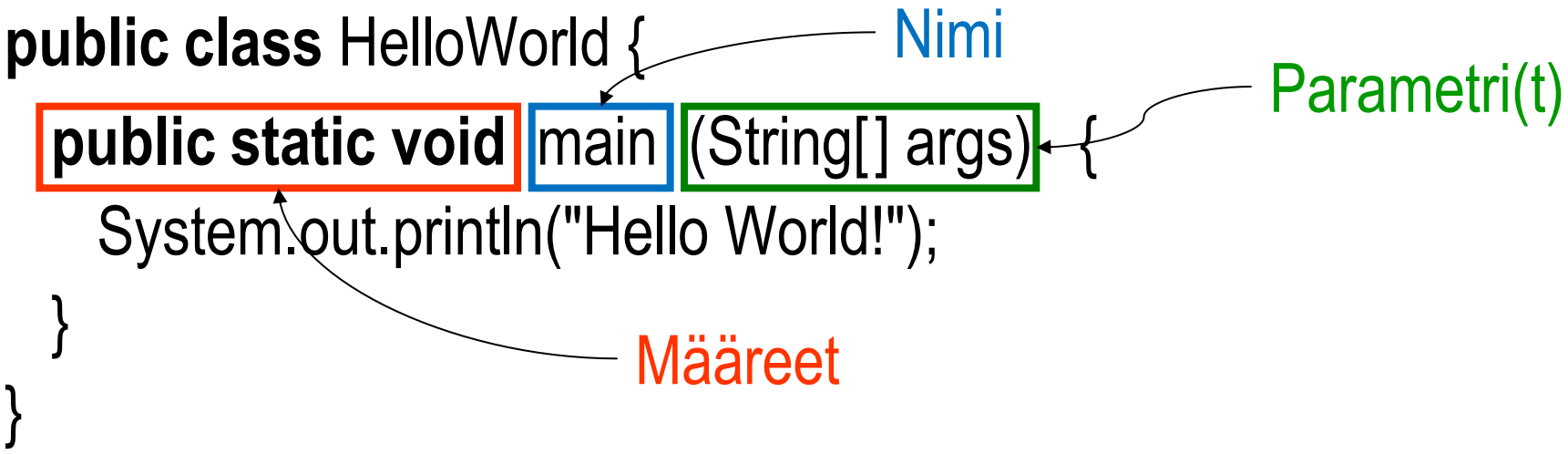
```
public static int laskePotenssi(int a, int b) {
  // Toisen operaation parametri ja  muuttuja eivät näy tänne.
  int tulos = 0; …
}

  // Tarkistaa syötteen. Paluuarvo on true, jos syöte on kunnossa.
  public static boolean tarkistaSyote(String syote) {
    // Ensimmäisen operaation parametrit ja muuttuja eivät näy tänne.
    boolean tulos = true; …
```

HelloWorld-ohjelma

- Jatketaan *HelloWorld*-ohjelman analyysiä.
- Aikaisemmalta kurssilta muistetaan, että
  - ajokelpoisessa Java-ohjelmassa on oltava *main*-operaatio eli niin sanottu pääohjelmaoperaatio,
  - *main*-operaation otsikon on oltava aina tietyn muotoinen, jotta ohjelman suoritus onnistuisi,
  - *main*-operaatio sijaitsee luokkamäärittelyn sisässä ja
  - *main*-operaation lohko sijaitsee luokan lohkon sisässä.

HelloWorld-ohjelma



- *Main*-operaatio on julkinen (**public**) eikä sillä ole paluuarvoa (**void**).
- *Main*-operaatiota kutsutaan automaattisesti. Tästä nimitys pääohjelmaoperaatio.
- Operaation parametreista kerrotaan myöhemmin.

Omat operaatiot: *sayHello*-operaatio

```
/* Tervehditään maailmaa
 * englanniksi. Huomaa static-
 * määre operaation otsikossa.
 */
public static void sayHello() {
    // Tekstiä näytölle.
    System.out.println("Hello World!");
}
```

- Ennen oman operaation otsikkoa kirjoitetaan aina yleisluonteinen kommentti operaation tarkoituksesta sekä mahdollisista parametreista ja paluuarvoista.
- Oheinen operaatio ei palauta arvoa (**void**).
- Operaatiolle ei myöskään voi antaa parametreja (parametrilista on tyhjä).

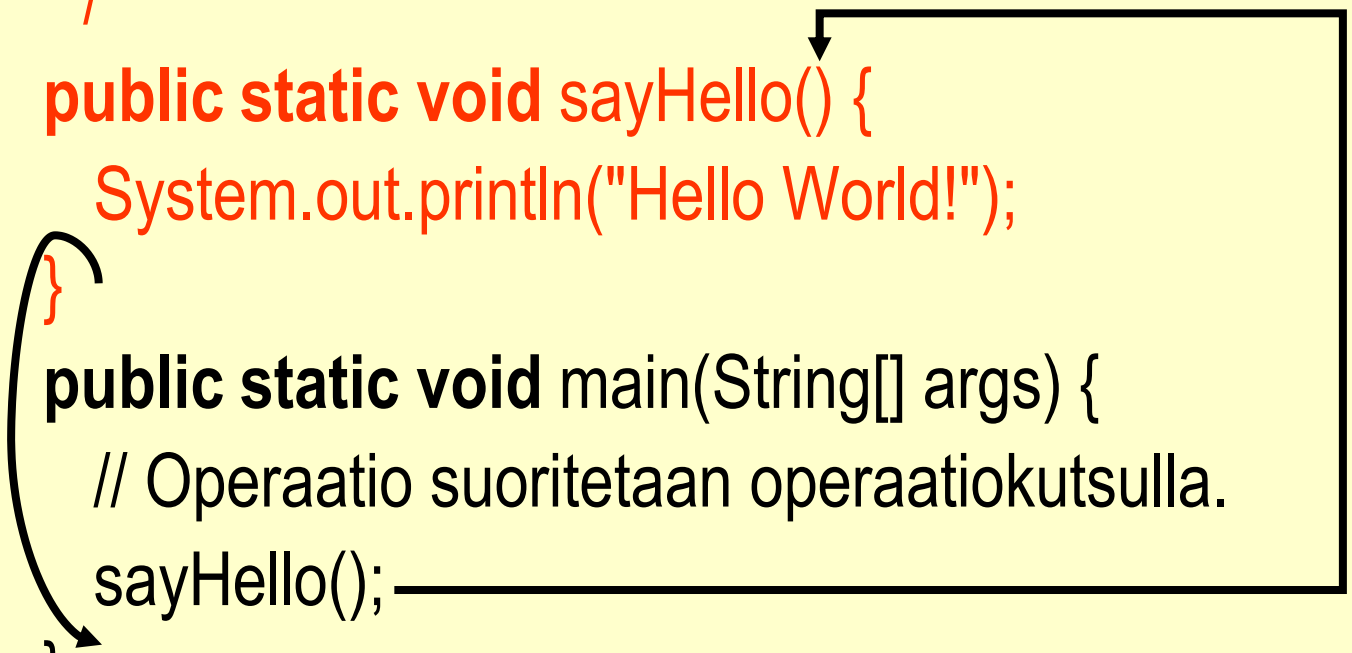
Omat operaatiot: paikka luokassa

```
public class HelloWorld {
  /* Tervehditään maailmaa englanniksi.
   */
  public static void sayHello() {
    System.out.println("Hello World!");
  }
  public static void main(String[] args) {
    // Kuinka sayHello voidaan suorittaa?
  }
}
```

- Omat operaatiot kirjoitetaan tällä kurssilla samaan luokkaan kuin *main*-operaatio.
- Otsikoihin on siksi muistettava liittää **static**-määre. Koodi ei käänny, jos **static**-määre puuttuu.

Omat operaatio: kutsuminen

```
public class HelloWorld {
   /* Tervehditään maailmaa englanniksi.
    */
   public static void sayHello() {
      System.out.println("Hello World!");
   }
   public static void main(String[] args) {
      // Operaatio suoritetaan operaatiokutsulla.
      sayHello();
   }
}
```



Omat operaatiot: kutsuminen

- *SayHello*-operaatiota kutsuttiin luokan sisällä toisesta operaatiosta nimellä ilman pistenotaatiota.
- Operaatiolle ei välitetty parametreja: kutsulause koostuu operaation nimestä ja “tyhjistä” suluista.
- Ohjelman suoritus palasi *main*-operaatioon automaattisesti.
- Operaatio ei palauttanut arvoa (**void**-määre).
- Operaatio olisi voitu sijoittaa myös *main*-operaation jälkeen.
  - Javassa operaatiota ei ole pakko määritellä ennen kuin sitä kutsutaan.

Omat operaatiot: kutsuminen

- Operaatiolle välitettävät tiedot määritellään parametrilistan avulla.
- Parametri saa arvonsa operaation kutsun yhteydessä.
- Parametriksi voidaan antaa arvo sellaisenaan (literaali), muuttuja, vakio, lauseke, toisen operaation paluuarvo jne.
- Arvoja on oltava oikea määrä, arvojen on oltava oikean tyyppisiä ja arvojen on oltava oikeassa järjestyksessä, jos järjestyksellä on väliä.
- Kutsun yhteydessä ei pidä kirjoittaa näkyviin arvon tyyppiä.
- Kutsussa käytettäviä muuttujia ei tarvitse nimetä samoin kuin parametreja, koska kutsuvan operaation tunnukset eivät näy kutsuttavaan metodiin ja päinvastoin.
- Operaation paluuarvo voidaan sijoittaa muuttujaan, jos tyypit ovat yhteensopivat.

Omat operaatiot: kutsuminen

- Esimerkiksi potenssin a^b laskevan operaation otsikossa:

  public static int laskePotenssi(**int** a, **int** b)

  määritellään **int**-tyyppiset parametrit *a* ja *b*.
- *LaskePotenssi*-operaatiota esimerkiksi *main*-operaatiosta kutsuttaessa on annettava kaksi **int**-tyyppistä arvoa (myös **byte**- ja **short**-arvot käyvät) siten, että kantaluku on ensimmäisen parametrin arvo arvo ja eksponentti toisen parametrin arvo.

  … laskePotenssi(2, 3) … // Kaksi potenssiin kolme.

  … laskePotenssi(3, 2) … // Kolme potenssiin kaksi.

  … laskePotenssi(2.5, 2) … // Virhe, 1. parametriarvo väärää tyyppiä.

  … laskePotenssi(**int** 2, **int** 3) … // Virhe, arvojen tyypit annettu.

Omat operaatiot: kutsuminen

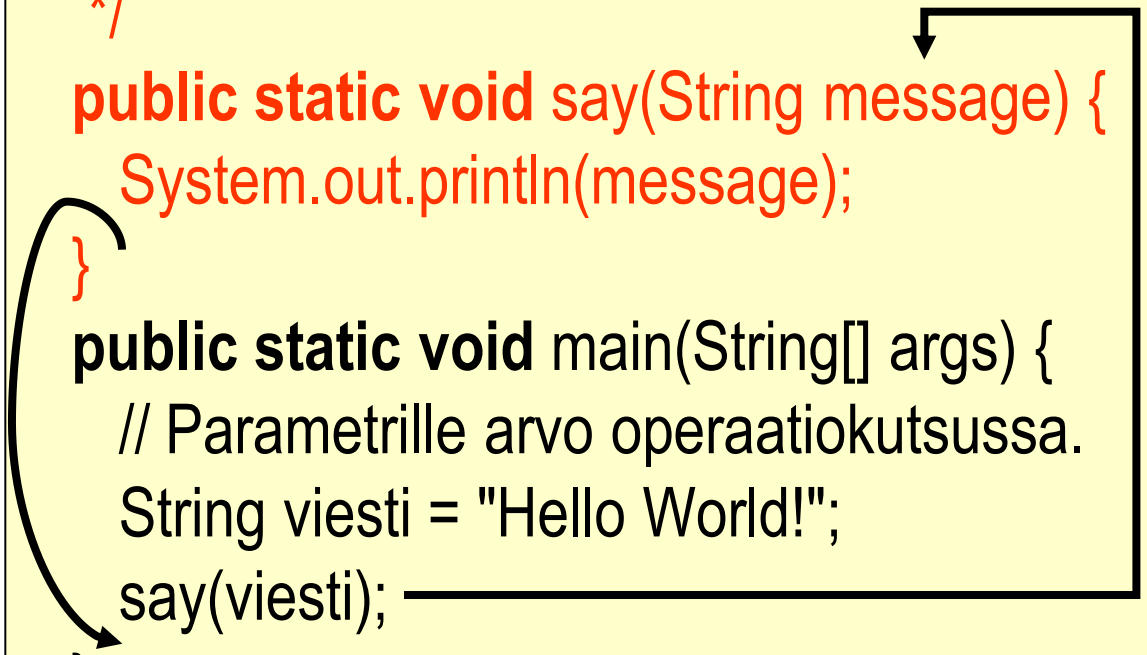
- Edellä määriteltyä operaatiota kutsuttaessa voi muuttujien nimet valita vapaasti. Esimerkiksi *main*-operaatiossa:

```
int kantaluku = 2, eksponentti = 3;
// Parametrien tunnuksia a ja b ei tarvitse käyttää, koska ne eivät näy
// potenssin laskevasta operaatiosta tänne.
… laskePotenssi(kantaluku, eksponetti);  …
```

- Java luo parametrille annetusta arvosta kopion ja sijoittaa sen automaattisesti parametrin tunnukseen liittyväksi arvoksi (pass-by-value, call-by-value).
  - Koska operaatio käsittelee kopiota, eivät operaation saamaansa arvoon tekemät muutokset välity alkuperäiseen arvoon.

Omat operaatiot: kutsuminen

```
public class HelloWorld {
  /* Tervehditään maailmaa operaation
   * kutsujan antamalla viestillä .
   */
  public static void say(String message) {
    System.out.println(message);
  }
  public static void main(String[] args) {
    // Parametrille arvo operaatiokutsussa.
    String viesti = "Hello World!";
    say(viesti);
  }
}
```



- Otsikossa määritellään, että operaatiolla on String-tyyppinen parametri.
- Operaatiolle annetaan sitä kutsuttaessa String-tyyppinen muuttuja, jonka arvo "Hello World!" kopioidaan ja sijoitetaan parametrin arvoksi automaattisesti.

Omat operaatiot: parametrit

```
…                                                    3
// Tervehditään n (> 0) kertaa.
public static void sayHello(int n) {
   while (n > 0) {
      System.out.println("Hello World!");
      n--;
   }
}
public static void main(String[] args) {
   int moilkm = 3;
   sayHello(moilkm);
   System.out.println(moilkm);  // 3
} …
```

- Muuttujan arvo kopioidaan ja sijoitetaan parametrin arvoksi automaattisesti.
- Parametrin *n* arvoon tehdyt muutokset eivät vaikuta muuttujan *moilkm* arvoon, koska arvot ovat erilliset.

Omat operaatiot: parametrit

```
/* Tulostaa näytölle annetulla merkillä n riviä korkean
 * ja m saraketta leveän suorakaiteen.
 */
public static void piirraSuorakaide(int n, int m, char
    merkki) {
  // Parametrien oltava järkevät, jotta edes yritetään
    piirtämistä.
  if (n > 0 && m > 0)
    for (int i = 0; i < n; i++) {                  // Tulostetaan
    n riviä.
      for (int j = 0; j < m; j++)       // Tulostetaan
    yksittäinen
        System.out.print(merkki);  // rivi eli m merkkiä.
      System.out.println();
  }
```

- Parametrien arvot on syytä tarkistaa operaatiossa, jos virheelliset arvot ovat mahdollisia, jotta ohjelmasta saadaan vakaampi.

Omat operaatiot: paluuarvo

- Operaation tyyppi määrää palauttaako operaatio arvon vai ei.
- **Void**-tyyppinen operaatio ei palauta arvoa.
  - Ohjelman suoritus palaa automaattisesti kutsupaikkaan, kun operaation viimeinen lause on suoritettu.
- Muun tyyppiset operaatiot palauttavat arvon **return**-lauseella.
  - **Return**-lause palauttaa ohjelman suorituksen kutsupaikkaan.
  - Yleisesti: **return** arvo;
- Paluuarvo voidaan tallentaa normaaliin tapaan muuttujaan.
  - Paluuarvon ja muuttujan tyyppien tulee olla yhteensopivat.

Omat operaatiot: paluuarvo

- Esimerkiksi potenssin a^b laskevan operaation otsikossa:

  public static int laskePotenssi(**int** a, **int** b)

  määritellään operaation tyypiksi **int**, jolloin operaation on aina palautettava **return**-lauseella **int**-tyyppinen arvo.

- Sijoitetaan eri tavoin kutsutun operaation paluuarvo muuttujaan esimerkiksi *main*-operaatiossa.

```
int kantaluku = 2, eksponentti = 3;
short i = 10;
int potenssi = laskePotenssi(2, 3);
potenssi = laskePotenssi(kantaluku, eksponetti);
potenssi = laskePotenssi(i - 1, i + 1);
```

Max-operaatio

```
/* Palauttaa suurimman luvuista a, b ja c.
 */
public static int max(int a, int b, int c) {
      int m;
      if ((a > b) && (a > c))
         m = a;
      else if (b > c)
         m = b;
      else
         m = c;
      return m;
}
```

- Parametrilistassa määritellään, että operaatiolle on annettava kolme **int**-tyyppistä arvoa. Arvojen järjestyksellä ei ole väliä.
- Operaatio on **int**-tyyppinen, jolloin operaatiossa on oltava arvon palauttava **return**-lause.

Max-operaatio

```
/* Palauttaa suurimman luvuista a, b ja c.
 */
public static int max(int a, int b, int c) {
    if ((a > b) && (a > c))
        return a;
    else if (b > c)
        return b;
    else
        return c;
}
```

- Tässä *max*-operaation versiossa arvo palautetaan jokaisessa päätös-haarassa saman tien.

Max-operaation kutsuja *main*-operaatiosta

```
public static void main(String[] args) {
   int k = 1, l = 3, m = 2;
   // Parametriarvoiksi muuttujien arvot, paluuarvo muuttujaan.
   int n = max(k, l, m);
   System.out.println(n);
   // Parametriarvoina literaaleja ja muuttujan arvo.
   // Paluuarvo katoaa, koska sitä ei sijoiteta muuttujaan.
   max(111, 222, l);
   // Paluuarvo annetaan toisen operaation parametriarvoksi.
   System.out.println(max(111, 222, 333));
}
```

Operaation kutsuminen yleisesti

- Luokan *A* operaatioita voi kutsua kolmella tavalla:
  1) Toisesta luokasta *A*-tyyppisen olion kautta.
  2) Toisesta luokasta luokan *A* nimen avulla.
  3) Luokan *A* toisista operaatioista.
- Kohdissa 1 ja 2 voidaan ajatella abstraktisti, että ympäristöstä lähetetty viesti käynnistää operaation.
- Käytännössä operaation kutsuminen tapahtuu tutulla pistenotaatiolla.
- Omia operaatioita kutsutaan tällä kurssilla vain kohdan 3 tavalla, koska kutsuminen luokan ulkopuolelta ei onnistu, kun käytettävissä on vain yksi itse kirjoitettu luokka.

Operaation kutsuminen olion kautta

- Yleisesti:

  olio.operaatio(parametrit);

  missä parametrit ovat operaation parametrilistassa annettua tyyppiä olevia arvoja (literaaleja, vakioita, muuttujia jne.).
- Parametreja ei luonnollisesti anneta, jos operaation parametrilista on tyhjä.
- Esimerkiksi:

```
String nimi = "Minni";
char ekaMerkki = nimi.charAt(0);
```

Operaation kutsuminen luokan kautta

- Yleisesti:

  Luokka.operaatio(parametrit);

  missä parametrit ovat kuten edellä.
- Operaatio on esiteltävä **static**-määreellä, jotta luokan nimellä kutsuminen onnistuisi.
- Tällaiset luokkaoperaatiot ovat hyödyllisiä, kun
  - kootaan yhteen toimintoja, joiden käyttö sujuvampaa suoraan ilman olion esittelyä ja
  - on tarpeen määritellä koko luokalle yhteisiä vakioita.

Operaation kutsuminen luokan kautta

- Luokkaoperaatiot ovat tuttuja jo *Math*- ja *In*-luokkien yhteydestä.
- Esimerkiksi:

```
double hinta = In.readDouble();
int itseisarvo = Math.abs(luku);
```

- Luokkaoperaatioista voidaan kutsua vain toisia luokkaoperaatioita.
- Tästä syystä *main*-operaation sisältävässä luokassa operaatioiden (ja attribuuttien) esittelyihin on lisättävä **static**-määre.

2. Taulukot

Sisältö

- Yleistä.
- Esittely ja luominen.
- Alkioiden käsittely.
- Kaksiulotteinen taulukko.
- Taulukko operaation parametrina.
- Taulukko ja *HelloWorld*-ohjelma.
- Taulukko paluuarvona.

Yleistä

- Suurten tietomäärien esittäminen erillisinä arvoina ei ole käytännöllistä.
- On helpompaa koota arvot yhdeksi kokonaisuudeksi eli tietorakenteeksi (data structure), joka mahdollistaa tietojen helpomman organisoinnin keskusmuistiin.
- Tietorakenteita on paljon ja ohjelmointiongelma ratkaisee pitkälti mitkä ovat sopivia tai parhaita.
  - Tarkemmin Tietorakenteet-kurssilla.
- Nyt käsitellään vain taulukko (array), joka on saman tyyppisten muuttujien eli alkioiden (element) kokoelma.

Yleistä

- Taulukon alkioilla on järjestys.
- Alkiot indeksoidaan 0, ... , *n* - 1, missä *n* on alkioiden lukumäärä (vertaa merkkijono).
- Esimerkki. **Int**-tyyppiset luvut 12, 56, 34 ja 78 sisältävä yksiulotteinen taulukko:

Indeksi	0	1	2	3
Alkio	12	56	34	87

Esittely ja luominen

- Esitellään hakasulkuja ([]) käyttäen.

  Yleisesti: **tyyppi[] taulukonNimi**;
- Taulukko on viitetyyppiä.
  - Java varaa esittelyn yhteydessä automaattisesti muistia viitteelle, mutta viitteeseen liittyvä olio on luotava **new**-operaatiolla käyttäjän toimesta.
- Luomisen yhteydessä määritellään myös taulukon koko (alkioiden lukumäärä).
- Yleisesti: **taulukonNimi = new tyyppi[koko]**;
- Taulukon alkiot alustuvat automaattisesti.

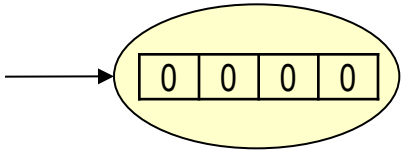
Esittely ja luominen

- **int**[] satunnaisluvut;

satunnaisluvut ⟶ **null**

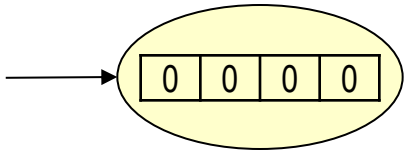
1) Java varaa esittelyn yhteydessä muistia viitteelle ja alustaa viitteen **null**-arvolla.

- satunnaisluvut = **new int**[4];



2) Lauseke **new int**[4]; luo taulukko-olion, alustaa alkiot ja palauttaa olioon liittyvän tunnuksettoman viitteen.

satunnaisluvut



3) Sijoituksen seurauksena *satunnaisluvut*-viite liittyy samaan olioon kuin paluuarvona saatu viite.

Esittely ja luominen

- Taulukko voidaan alustaa luonnin yhteydessä, jolloin koko määräytyy automaattisesti. Yleisesti:

  taulukonNimi = new tyyppi[] { arvo1, … , arvo*N* };

- Esimerkki. // Esitellään taulukko.

  int[] satunnaisluvut;

  // Luodaan 4 alkion kokoinen taulukko.

  satunnaisluvut = **new int**[4];

- Esimerkki. **int**[] satunnaisluvut;

  satunnaisluvut = **new int**[] { 12, 56, 34, 87 };

Esittely ja luominen

- Taulukon esittely, luominen ja mahdollinen alustus voidaan yhdistää samaan lauseeseen.
- Esimerkki. **int**[] satunnaisluvut = **new int**[4];

  int[] satunnaisluvut = **new int**[] { 12, 56, 34, 87 };

  // Esittely, luominen ja alustus lyhemmin.

  int[] satunnaisluvut = { 12, 56, 34, 87 };
- Taulukon kokoa ei voi muuttaa jälkeenpäin.
- Kullakin taulukolla on julkinen *length* attribuutti, joka ilmoittaa taulukon koon.
  - Hyödyllinen erityisesti, kun taulukko saadaan parametrina.

Alkioiden käsittely

- Hakasulkeilla ([]) taulukosta voidaan myös “erottaa” tietty alkio (muuttuja), jonka avulla alkion arvo voidaan lukea tai muuttaa.
- Alkion arvon lukeminen yleisesti:

  taulukonNimi[indeksiarvo]
- Alkion arvon muuttaminen yleisesti:

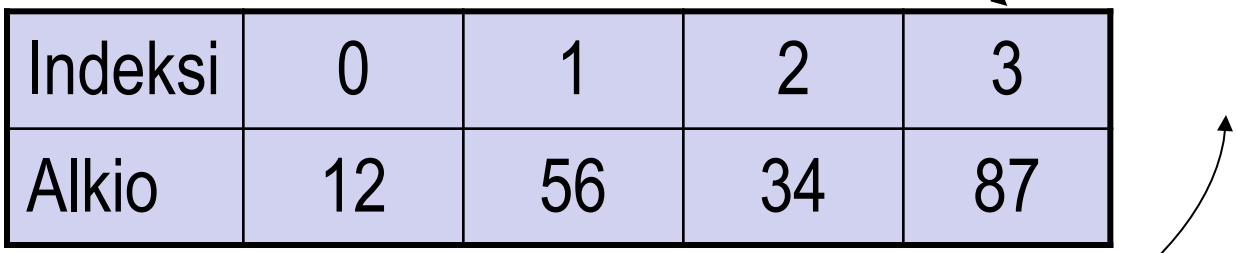
  taulukonNimi[indeksiarvo] = arvo;
- Indeksiarvo on **int**-tyyppinen literaali, muuttuja, vakio tai lauseke.

Alkioiden käsittely

- Indeksointi alkaa nollasta, jolloin 0 ≤ indeksiarvo < taulukon koko.
- Virheellinen indeksin arvo aiheuttaa ajonaikaisen virheen *ArrayIndexOutOfBoundsException*.
- Erityisesti taulukon ylärajan kanssa on syytä olla tarkkana: *n* alkion kokoisen taulukon viimeinen paikka on *n* - 1.
- **For**-silmukka on erityisen kätevä, kun on tarpeen käydä läpi kaikki taulukon indeksiarvot.

Esimerkki: arvon lukeminen

- System.out.println(satunnaisluvut[3]); // 87

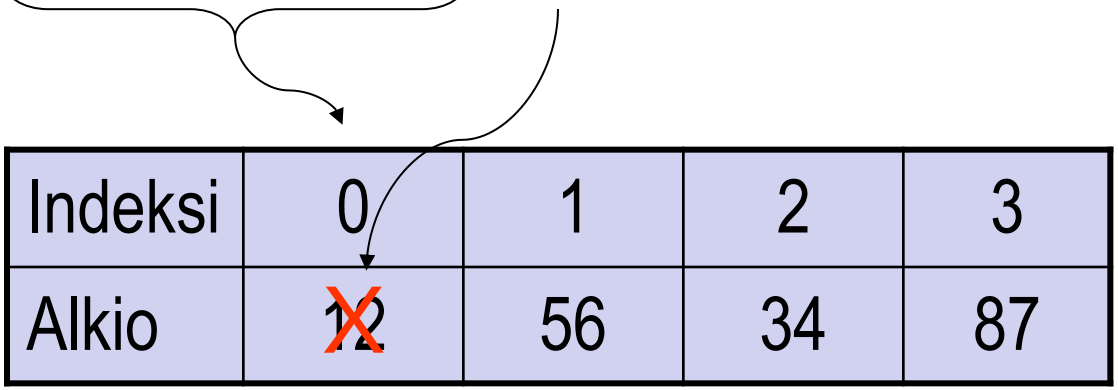


Indeksi	0	1	2	3
Alkio	12	56	34	87

- **int** l = satunnaisluvut[4] + 1; // Ohjelma kaatuu

Esimerkki: arvon muuttaminen

- satunnaisluvut[0] = 6;



Indeksi	0	1	2	3
Alkio	~~12~~	56	34	87

Indeksi	0	1	2	3
Alkio	6	56	34	87

- System.out.println(satunnaisluvut[0]); // 6

Esimerkki: käsittelyä silmukoilla

```
/* Taulukon käsittelyä silmukoilla: täytetään taulukko satunnaisluvulla
 * ja tulostetaan luvut näytölle.
 */
public class TaulukonKasittelya {
   public static void main(String[] args) {
      // Taulukon koko ja sen suurin luku vakioina.
      final int KOKO = 100; final int MAX = 10;
      // Taulukon esittely ja luominen.
      int[] satunnaisluvut = new int[KOKO];
      // Täytetään taulukko satunnaisluvuilla väliltä [0, MAX].
      for (int i = 0; i < KOKO; i++)
         satunnaisluvut[i] = (int)((MAX + 1) * Math.random());
      // Tulostetaan taulukon alkiot.
      for (int j = 0; j < KOKO; j++)
         System.out.print(satunnaisluvut[j] + " ");
   }
}
```

Esimerkki: haku taulukosta

```
/* Taulukon käsittelyä silmukoilla:
 * täytetään taulukko satunnaisluvulla ja haetaan
 * käyttäjän antamaa lukua taulukosta.
 */
public class SatunnainenTaulukko1D {
  public static void main(String[] args) {
    // Taulukon suurin luku.
    final int MAX = 100;
    // Taulukon esittely ja luominen.
    int[] satunnaisluvut = new int[10];
    // Täytetään satunnaisluvuilla väliltä [0, MAX].
    for (int i = 0; i < satunnaisluvut.length; i++) {
      satunnaisluvut[i] =
      (int)((MAX + 1) * Math.random());
      System.out.print(satunnaisluvut[i] + " ");
    }
    // Luetaan haettava luku.
    System.out.println("\nAnna haettava luku:");
    int luku = In.readInt();
    // Haetaan.
    boolean hukassa = true;
    int j = 0;
    while (j < satunnaisluvut.length && hukassa) {
      if (satunnaisluvut[j] == luku)
        hukassa = false;
      j++;
    }
    if (!hukassa)
      System.out.print("Löytyi!");
    else
      System.out.print("Ei löytynyt!");
  }
}
```

Kaksi ulottuvuutta

- Taulukossa voi olla useampia ulottuvuuksia.
- Kahden ulottuvuuden esittämiseen tarvitaan kaksi indeksiä:

toinen ulottuvuus

ensimmäinen ulottuvuus

Indeksi	0	1	2	3
0	23	44	5	33
1	3	32	0	11
2	12	56	34	87

Kaksi ulottuvuutta

- Esittely yleisesti: **tyyppi[][] nimi**;
  - Esim. // Esitellään taulukko.
    int[][] satunnaisluvut;
- Luominen yleisesti:
  taulukonNimi = new tyyppi[rivejä][sarakkeita];
  - Esim. // Luodaan 12 alkion kokoinen taulukko.
    satunnaisluvut = **new int**[3][4];
- Esittely ja luominen samassa lauseessa:
  tyyppi[][] taulukonNimi = new tyyppi[rivejä][sarakkeita];
  - Esim. // Esitellään ja luodaan 12 alkion kokoinen taulukko.
    int[][] satunnaisluvut = **new int**[3][4];

Kaksi ulottuvuutta

- Alustaminen luomisen yhteydessä:

  tyyppi[][] taulukonNimi = { rivi1, … , **rivi*N* }**

  missä **rivi = { arvo1, … , arvo*M* }**

  (myös **tyyppi[][] taulukonNimi =**

  new tyyppi[][] { rivi1, … , **rivi*N* }**

  Edellä *N* rivien lukumäärä ja *M* sarakkeiden lukumäärä.
- Esimerkki. **int**[][] satunnaisluvut = { { 23, 44, 5, 33 },
  { 3, 32, 0, 11 }, { 12, 56, 34, 87 } };

Kaksi ulottuvuutta

- Arvoon viittaaminen yleisesti: **taulukonNimi[rivi][sarake]**
  - Esim.
```
System.out.println(satunnaisluvut[0][0]);    // 23
System.out.println(satunnaisluvut[1][2]);    // 0
System.out.println(satunnaisluvut[2][3]);    // 87
```

Indeksi	0	1	2	3
0	23	44	5	33
1	3	32	0	11
2	12	56	34	87

- *Length*-attribuutti ilmoittaa rivien lukumäärän.
  - Sarakkeiden lukumäärä saadaan selville tutkimalla jonkin rivin pituus tai rivin pituus.

Esimerkki: käsittelyä silmukoilla

```
/* Taulukon käsittelyä silmukoilla: täytetään taulukko satunnaisluvulla ja tulostetaan sisältö.
 */
public class SatunnainenTaulukko2D {
   public static void main(String[] args) {
      // Taulukon suurin luku.
      final int MAX = 100;
      // Taulukon esittely ja luominen.
      int[][] satunnaisluvut = new int[3][4];
      // Täytetään taulukko satunnaisluvuilla väliltä [0, MAX].
      for (int rivi = 0; rivi < satunnaisluvut.length; rivi++)
         for (int sarake = 0; sarake < satunnaisluvut[0].length; sarake++)
            satunnaisluvut[rivi][sarake] = (int)((MAX + 1) * Math.random());
      // Tulostetaan taulukon alkiot.
      for (int rivi = 0; rivi < satunnaisluvut.length; rivi++) {
         for (int sarake = 0; sarake < satunnaisluvut[0].length; sarake++)
            System.out.print(satunnaisluvut[rivi][sarake] +  "\t");
         System.out.println();
      }
   }
}
```

Taulukko parametrina

- Kaiken tyyppiset parametrit ovat operaation lohkon tunnuksia:
  - Parametrit näkyvät vain lohkossaan: luodaan operaatioon tultaessa ja tuhotaan operaatiosta poistuttaessa.
- Alkeistyyppisen parametrin arvoon tehdyt muutokset eivät jää voimaan, koska arvo on kopio.
- Viitetyyppinen parametri on kopio alkuperäisestä viitteestä ja liittyy samaan olioon kuin operaation kutsupaikassa.
- Kopioidun viitteen kautta taulukko-olion sisältöön operaatiossa tehdyt muutokset säilyvät operaatiosta poistumisen jälkeen, koska sisältö ei ole kopio.
  - Operaatiosta ei tarvitse palauttaa viitettä, kun sisältöä muutetaan.

Taulukko parametrina

- Koska viitetyyppiselle parametrille pitää varata muistia **new**-operaatiolla, ja ihmisen tiedetään olevan erehtyväinen, on viitetyyppisiä parametreja sisältävissä operaatioissa hyvä tarkistaa, että muistia on todella varattu.
- Tähän tehtävään sopii **null**-arvo, jonka Java antaa alkuarvoksi jokaiselle viitteelle. **Null**-arvon voidaan ajatella olevan keskusmuistin ulkopuolella.
- **if** (viitetyyppisenParametrinTunnus != **null**) { … }

Taulukko parametrina

```
/* Täytetään taulukko t satunnaisluvuilla.
 */
public static void tayta(int[] t) {
   // Satunnaisluvut välillä [0, MAX[.
   final int MAX = 10;
   // Täytetään, jos on varattu muistia.
   if (t != null) {
      for (int i = 0; i < t.length; i++)
         t[i] = (int)(MAX * Math.random());
   }
}
```

Ilman **if**-lausetta ohjelman suoritus keskeytyy ajonaikaiseen virheeseen (*NullPointer-Exception*) aina kun parametrille ei ole varattu muistia. Huomaa ettei paluuarvoa ole.

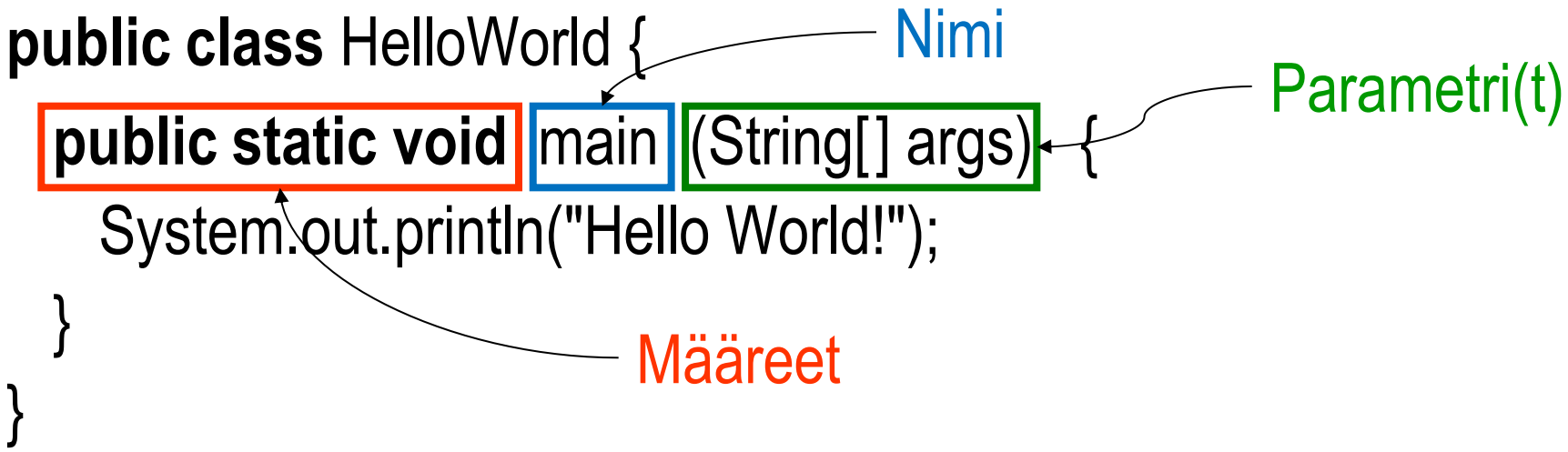
Taulukko parametrina

```
/* Tulostetaan taulukko t.
 */
public static void tulosta(int[] t) {
   // Tulostetaan, jos on varattu muistia.
   if (t != null) {
      for (int i = 0; i < t.length; i++)
         System.out.print(t[i] + " ");
      System.out.println();
   }
}
```

Taulukko parametrina

```
public static void main(String[] args) {
    final int KOKO = 5;
    int[] satunnaisluvut = new int[KOKO];
    // Java alustaa taulukon sisällön automaattisesti.
    tulosta(satunnaisluvut);            // 0 0 0 0 0
    // Täytetään taulukko satunnaisluvuilla.
    tayta(satunnaisluvut);
    // Operaatiossa taulukkoon tehdyt muutokset säilyvät.
    tulosta(satunnaisluvut);            // 8 0 5 7 8
 }
```

HelloWorld-ohjelma



- Parametri *args* on *String*-tyyppisten alkioiden taulukko.

Taulukko paluuarvona

- Operaatiossa luotu taulukko voidaan palauttaa operaation paluuarvona.
  - Operaatiosta palautetaan taulukko-olioon liittyvä viite.
  - Kutsuvassa operaatiossa taulukko-olioon täytyy kiinnittää uusi viite, jotta oliota voidaan käyttää.
- Joskus yksiulotteista taulukkoa käytetään kahden tai useamman paluuarvon palauttamiseen.
  - Eri tyyppisten arvojen palautus hankalaa.
  - Useampia arvoja voidaan palauttaa joustavammin luokkatyyppisen olion avulla.

Taulukko paluuarvona

```
/* Luodaan n alkion taulukko, täytetään se satunnaisluvuilla sekä
 * palautetaan viite taulukko-olioon. Paluuarvo on null, jos n < 1.
 */
public static int[] luoJaTayta(int koko) {
   final int MAX = 10;   // Satunnaisluvut välillä [0, MAX[.
   if (koko > 0) {   // Koko on järkevä.
      int[] taulu = new int[koko];   // Luodaan olio ja liitetään siihen viite.
      for (int i = 0; i < taulu.length; i++)   // Täytetään satunnaisluvuilla.
         taulu[i] = (int)(MAX * Math.random());
      return taulu;   // Palautetaan viite taulukko-olioon.
   }
   else   // Virheellinen parametri.
      return null;   // Palautetaan virheen merkkinä null-arvo.
}
```

Taulukko paluuarvona

```
public static void main(String[] args) {
    // Taulukon alkioiden lukumäärä vakiona.
    final int KOKO = 5;
    // Luodaan taulukko, täytetään se satunnaisluvuilla
    // ja liitetään siihen viite, jotta taulukkoa ei hukata.
    int[] satunnaisluvut = luoJaTayta(KOKO);
    // Operaatiossa taulukkoon sijoitetut arvot säilyvät.
    tulosta(satunnaisluvut);        // 5 6 2 1 4
}
```

3. Olio-ohjelmoinista lyhyesti

Sisällys

- Yleistä.
- Oliot ja luokat.
- Attribuutit.
- Olioiden esittely ja alustus.
- Rakentajat.
- Olion operaation kutsuminen.

Yleistä

- Olio-ohjelmointia käsitellään hyvin yleisellä tasolla:
  - Tarkastellaan vain yhtä omaa luokkaa.
  - Olio-ohjelmoinnissa keskeiset attribuutit käsitellään lyhyesti.
  - Keskitytään operaatioihin.
- Aihepiiriin palataan tarkemmin kolmannessa periodissa järjestettävällä Olio-ohjelmoinnin perusteet (Oope, 10 op) -kurssilla.

Oliot ja luokat

- Oliot (object) ja luokat (class) ovat keskeisiä olio-ohjelmoinnin käsitteitä.
- Olio-ohjelmointi on ohjelmointiparadigma, jossa
  - ohjelma kuvataan keskenään kommunikoivina olioina,
  - oliot ajatellaan luokkansa ilmentymiksi (instance) ja
  - luokille voidaan määritellä periytymissuhteita.
- Oliot ja luokat liittyvät siis kiinteästi toisiinsa, mutta ovat kuitenkin eri asia!

Oliot ja luokat

- Olio-ohjelmoinnissa ohjelman käyttökohteen (ja sen ympäristön) eli sovellusalueen käsitteet (concept) pyritään mallintamaan formaalisti luokkien avulla.
- Luokka vastaa useimmiten sovellusalueen käsitettä hyvin karkealla tasolla.
- Luokkia ei voida yleensä määritellä suoraan, vaan ensin pitää analysoida sovellusalueen kohteita, joiden voidaan ajatella olevan käsitteiden ilmentyminä myös omanlaisiaan olioita.

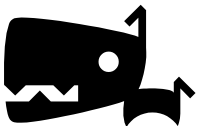
Oliot ja luokat

- Luokkia rakennettaessa edetään siis usein yksityiskohdista yleiseen määrittelyyn.
- Luokkiin kootaan sovellusalueen olioille yhteisiä
  - tietoja (attribuutteja) ja
  - toiminnallisuutta (operaatioita eli metodeja).
- Oliolla on kaksi roolia:
  - Sovellusalueen käsitteen edustaja.
  - Käsitettä (karkeasti) vastaavan luokan edustaja.

Koira-luokka

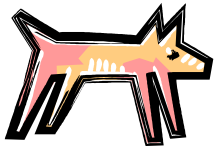
- Oletetaan, että sovellusalueella on koiria. Mallinnetaan ensin luokaksi ja toteutetaan sitten Javalla.

Teukka
- kiltti
- sekarotuinen
- "Hau!"
- metsästää rottia



Hiski
- rohkea
- bullterrieri
- "RÄYH!"
- repii sohvaa

Viivi
- aristokraattinen
- villakoira
- "vuh"
- antaa tassua



Koira
- luonne
- rotu
- haukkuu

Koira-luokka

- Kun luokan sisältö on selvillä, se voidaan toteuttaa: Esitellään attribuutit ja toteutetaan metodit.
- Ohessa hyvin alkeellinen *Koira*-luokan toteutus Java-kielellä.

```
public class Koira {
   // Attribuutit.
   private int luonne;
   private String rotu;
   // Metodit.
   public void hauku(String s) {
      System.out.println(s);
   }
}
```

Attribuutit

- Luokan lohkossa esiteltyjä muuttujia tai vakioita.
- Esitellään lähes samalla tavoin kuin “tavalliset” muuttujat tai vakiot.
- Lisänä määreet. Pyritään yleensä kätkemään luokan ympäristöltä **private**-määreellä.
- Luetaan ja muutetaan aksessorimetodeilla.
- Käytettävä varoen: näkyvät kaikkiin metodeihin ja rikkovat siten modulaarisuusperiaatetta.
  - Älä käytä tällä kurssilla ellei lupaa ole annettu.
  - Vakioituja attribuutteja voi käyttää vapaammin.

```
public class Koira {
  // Attribuutit.
  private int luonne;
  private String rotu;
  // Aksessorit.
  public int luonne() {
    return luonne;
  }
  // Metodit.
  public void hauku(String s) {
    System.out.println(s);
  }
}
```

Olioiden esittely ja alustus

- Javan oliot eivät ole oliotyyppiä vaan viitetyyppisiä (tarkemmin luokkatyyppisiä) muuttujia.
- Luokkatyyppisiä muuttujia ei voi yleensä ottaen käsitellä kuten alkeistietotyyppisiä muuttujia.
- Viite “osoittaa” jonkin muistipaikan kautta varsinaiseen dataan keskusmuistissa.
- Tästä syystä luokkatyyppisen muuttujan esittely varaa muistia vain viitteen tallentamiseen, ei olion tietojen tallentamiseen.

Olioiden esittely ja alustus

- Olion tarvitsema muistitilavaraus tehdään alustuksen yhteydessä **new**-operaatiolla, joka palauttaa viitteen ja varaa keskusmuistista muistialueen olion tiedoille.
- Yleisesti:

```
LuokanNimi olionNimi;
olionNimi = new LuokanNimi();
```

tai

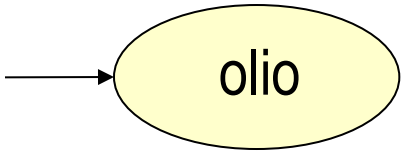
```
LuokanNimi olionNimi = new LuokanNimi();
```

Olion luominen

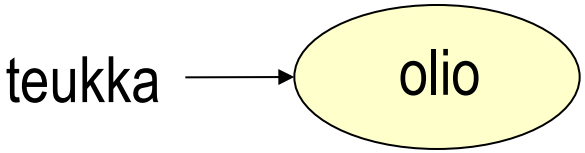
- Koira teukka = **null**;

teukka ⟶ **null** Olion viitteen esittely ja alustaminen tyhjäksi.

- teukka = **new** Koira();



Lauseke **new** Koira(); luo olion ja palauttaa paluu-arvona olioon liittyvän tunnuksettoman viitteen.



Sijoituksen seurauksena *teukka*-viite viittaa samaan olioon kuin paluuarvona saatu viite.

Rakentajat

- Attribuuttien arvot alustetaan luokan rakentajassa (constructor), jonka nimi on aina sama kuin luokan nimi. Rakentajalla ei ole tyyppiä.
- Kutsu yleisesti:

  - Oletusrakentaja **LuokanNimi();** tai

  - Parametrillinen rakentaja **LuokanNimi(parametrit);**
- Rakentajaa kutsutaan aina, kun luodaan olio.
- Esimerkiksi: Koira hiski = **new** Koira();

Rakentajat

- Java luo automaattisesti tyhjän oletusrakentajan.
- Itse kirjoitetulle luokalle voidaan luonnollisesti kirjoittaa myös rakentaja.
- Entä jos on tarvetta sekä parametrittomalle oletusrakentajalle että parametriselle rakentajalle?
- Javassa on kuormittamisena (overloading) tunnettu mekanismi, joka sallii samannimisten metodien esittelyn ja siten myös samannimisen rakentajien esittelyn.

Olion operaation kutsuminen

- Olion luokan julkisia operaatioita voidaan kutsua olion luokan ulkopuolelta muuttujan nimen ja pistenotaation avulla.
- Esimerkiksi:
```
Koira hiski = new Koira();
hiski.hauku("RÄYH!");
```
- Esimerkiksi:
```
String syote = In.readString();
int merkkienLkm = syote.length();
```
- Esimerkiksi:
```
Scanner lukija = new Scanner(System.in);
String syote = lukija.nextLine();
```
- Esimerkiksi:
```
Random generaattori = new Random();
int arvottu = generaattori.nextInt(10);
```

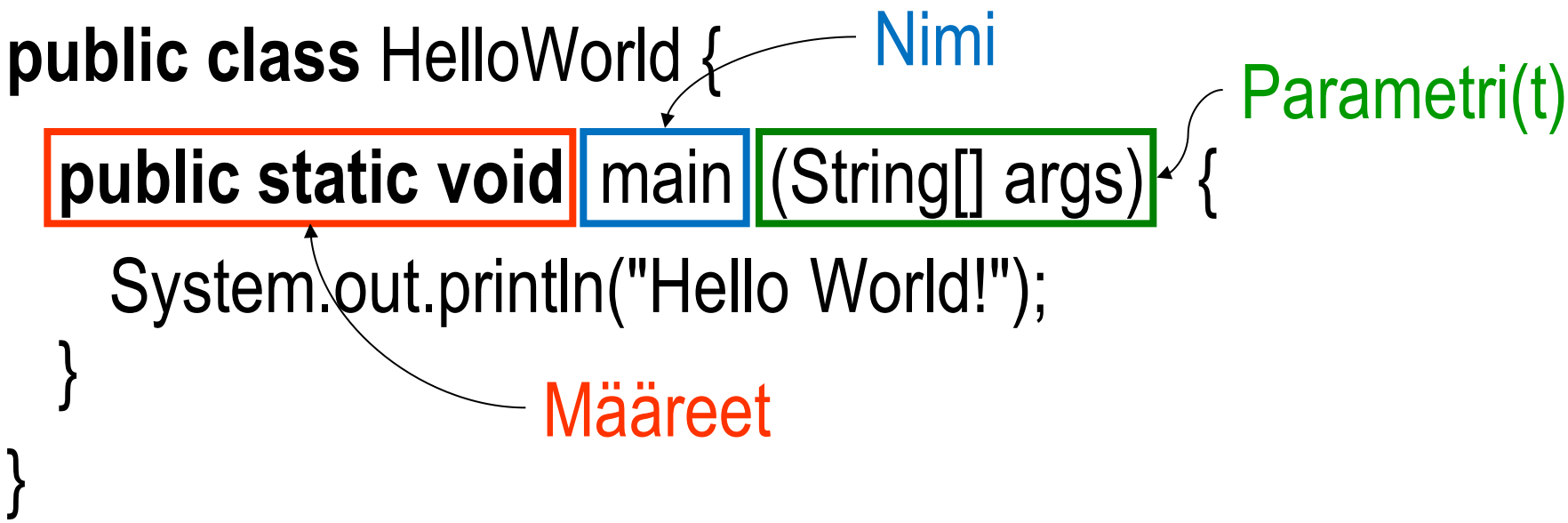
4. Komentoriviparametrit

Komentoriviparametrit

- Ohjelman nimen perään kirjoitettavia merkkijonoja.
- Erotetaan välilyönnein.
- Tuttuja Windows- ja UNIX-komentotulkeista.
- Esimerkkejä komentojen parametreista

Komento	Ohjelman nimi	Parametri(t)
copy In.java c:\temp	copy	In.java c:\temp
dir *.java	dir	*.java
cat -v teksti.txt	cat	-v teksti.txt

HelloWorld: main-operaatio



- Parametri *args* on String-tyyppisten alkioiden taulukko, jonka avulla komentoriviparametri(t) välitetään Java-ohjelmalle.

Komentoriviparametrit

- Komentorivi: java SayHello World
- ```
  public class SayHello {
      public static void main(String[] args) {
          // Tulostetaan, jos (vähintään) yksi parametri.
          if (args.length > 0)
              System.out.println("Hello " + args[0]);
      }
  }
  ```
- Näyttö: Hello World
  ```

Komentoriviparametrit

```
public class KRPt {
   public static void main(String[] args) {
      System.out.print("Komentoriviparametrejä ");
      System.out.print(args.length);
      System.out.println(" kpl.");
      // Tulostetaan komentoriviparametrit näytölle.
      for (int i = 0; i < args.length; i++)
         System.out.println(args[i]);
   }
}
```

Komentoriviparametrit

- Komentorivi:

  java KRPt Tämä on testi

- Näyttö:

  Komentoriviparametrejä 3 kpl.

  Tämä

  on

  testi

5. Poikkeukset

Yleistä

- Virheeseen varautuminen tarkoittaa sitä, että ohjelmoija huomioi koodia kirjoittaessaan ajonaikaisen virheen mahdollisuuden.
- Perinteiset tavat käsitellä virheitä:
  - Virhe tunnistetaan ja vältetään, mutta siihen ei reagoida muuten.
  - Virhe tunnistetaan ja vältetään ja sen tapahtumisesta ilmoitetaan operaation paluuarvon avulla.

Yleistä

- Edellä mainitut keinot antavat riittävät keinot virheiden käsittelyyn, mutta niissä on omat ongelmansa:
  - Paluuarvot on helppo jättää huomiotta.
  - Paluuarvoa ei voi aina käyttää virheen ilmaisemiseen.
  - Virheidenkäsittelyyn käytetty ohjelmakoodimäärä saattaa olla yllättävän suuri ja ohjelman luettavuus saattaa kärsiä.
- Java tarjoaa poikkeuksina (exceptions) tunnetun mekanismin ajonaikaisten virheidenkäsittelyyn.

Poikkeusten käsittely

- Mikäli jossakin operaatiossa tapahtuu ajonaikaisen virheen aiheuttama poikkeus, on sen käsittelyyn kaksi mahdollisuutta:
  - poikkeus käsitellään paikallisesti pois päiväjärjestyksestä tai
  - poikkeus heitetään (throw) kutsuvalle operaatiolle.
- Ohjelman suoritus pysähtyy, mikäli poikkeusta ei käsitellä viimeistään pääohjelmassa.
- Poikkeusta ei siis voida jättää koskaan huomiotta.
- Tällä kurssilla tutustutaan vain paikalliseen käsittelyyn, jotta tiedostojenkäsittelty olisi mahdollista.

Poikkeusten käsittely

- Poikkeuksen käsittelyyn tarvitaan **try-catch**-lause:

```
try {
    // Mahdollisesti poikkeuksen tuottavat lauseet.
}
catch (Tyyppi1 tunnus1) {
     // Tässä käsitellään tyypin1 virheet.
}

…
catch (Tyyppi N tunnus N) {
    // Tässä käsitellään tyypin N virheet.
}
```

Poikkeusten käsittely

- Poikkeusten tyypit ovat Javan luokkia – poikkeukset ovat siis olioita.
- Joitakin poikkeuksia:
  - *NullPointerException*: viitteelle ei ole varattu muistia.
  - *ArrayIndexOutOfBoundsException*: viitataan virheelliseen paikkaan taulukossa.
  - *OutOfMemoryError*: ei riittävästi muistia olion luomiseksi.
  - *Exception*: Mikä tahansa poikkeus.

Poikkeusten käsittelyä paikallisesti

```
/* Täytetään taulukko t satunnaisilla luvuilla.
 */
public static void tayta(int[] t) {
   try {
      // Satunnaisluvut välillä [0, MAX[.
      final int MAX = 10;
      // Mahdollista virhettä ei vältetä vaan se käsitellään.
      for (int i = 0; i < t.length; i++) {
         t[i] = (int)(MAX * Math.random());
      }
   }
   catch (NullPointerException e) {
      System.out.println("Taulukolle pitää varata muistia!");
   }
}
```

6. Tiedostot

Sisältö

- Johdanto.
- Tiedostojen lukeminen.
- Tiedostojen kirjoittaminen.

Johdanto

- Tiedostoja on käsitelty uudelleenohjattujen standardisyöte- ja tulostusvirtojen avulla.
- Tiedostoja voidaan käyttää myös ohjelmasta käsin.
- Tutustutaan tekstitiedostojen lukemiseen ja kirjoittamiseen *java.io*-pakkauksen luokkien avulla.
  - Javan versiosta 1.5 alkaen käytettävissä myös *java.util*-pakkauksen *Scanner*- ja *Formatter*-luokat.
  - **import** java.io.*; (tai **import** java.util.*;).
- Lukeminen ja kirjoittaminen tapahtuu vaiheittain: avataan, luetaan tai kirjoitetaan, suljetaan.

Avaaminen lukemista varten

- Tiedoston avaaminen tarkoittaa yleisesti syötevirran liittämistä tiedostosta ohjelmaan.
- Javassa syötevirta avataan esimerkiksi luomalla *FileInputStream*-luokan olio. Yleisesti:

  FileInputStream olionNimi

  = **new** FileInputStream(tiedostonNimi);
- Esimerkiksi: FileInputStream syotevirranNimi

  = **new** FileInputStream("in.txt");

Lukeminen

- Syötevirrasta lukemiseen tarvitaan olio *InputStreamReader*-luokasta.
- Esimerkiksi: InputStreamReader lukijanNimi =

  new InputStreamReader(syotevirranNimi);
- Usein on kätevämpää käyttää hienostuneempaa lukijaa. Tällaisen tarjoaa *BufferedReader*-luokka.
- Esimerkiksi: BufferedReader puskuroituLukija =

  new BufferedReader(lukijanNimi);

Lukeminen

- Kun syötevirtaan on saatu liitettyä *BufferedReader*-lukija, voidaan tekstitiedostoa lukea riveittäin *readLine*-operaation avulla.
- *Ready*-operaatio kertoo voidaanko lukea uusi rivi.
- Esimerkiksi:

```
while (puskuroituLukija.ready()) {
    String rivi = puskuroituLukija.readLine();
    System.out.println(rivi);
}
```

Rivien käsittely

- Usein yhdellä rivillä on useampia jollakin merkillä (esimerkiksi pilkku tai välilyönti) erotettuja tietoja, jotka voivat vieläpä olla eri tyyppisiä.
- *String*-luokan *split*-operaatio on tässä tilanteessa hyödyllinen – se pilkkoo rivi osiinsa.
- Esimerkiksi:

```
String rivi = "this is a test";
String[] osat = rivi.split("[ ]");
for (int i = 0; i < osat.length; i++)
   System.out.println(osat[i]);
```

Sulkeminen lukemisen jälkeen

- Tiedoston lukemisen jälkeen siihen liitetty syötevirta suljetaan *close*-operaatiolla.
- Esimerkiksi: puskuroituLukija.close();
- Tiedoston avaamisen, lukemisen ja/tai sulkemisen yhteydessä voi tapahtua poikkeus.
- Tämän vuoksi Java pakottaa sulkemaan useimmat tiedostonkäsittelyyn liittyvät lauseet **try-catch**-lauseen sisään.

Esimerkki

```
import java.io.*;
public class Lukeminen1 {
  public static void main(String[] args) {
    String TIEDNIMI = "in.txt";
    try {
      FileInputStream syotevirta =
      new FileInputStream(TIEDNIMI);
      InputStreamReader lukija =
      new InputStreamReader(syotevirta);
      BufferedReader puskuroituLukija =
      new BufferedReader(lukija);
      while (puskuroituLukija.ready()) {
        String rivi = puskuroituLukija.readLine();
        System.out.println(rivi);
      }
      puskuroituLukija.close();
    }
    catch (FileNotFoundException e) {
      System.out.print("Tiedosto hukassa!");
    }
    catch (Exception e) {
      System.out.print("Lukuvirhe!");
    }
  }
}
```

Esimerkki

```
import java.io.*; import java.util.*;
public class Lukeminen2 {
  public static void main(String[] args) {
    String TIEDNIMI = "in.txt";
    try {
      // Luodaan tiedosto-olio.
      File tiedosto = new File(TIEDNIMI);
      // Luodaan lukija.
      Scanner lukija = new Scanner(tiedosto);
      // Luetaan ja tulostetaan rivit.
      while (lukija.hasNextLine()) {
        String rivi = lukija.nextLine();
        System.out.println(rivi);
      }
      // Suljetaan lukija.
      lukija.close();
    }
    catch (FileNotFoundException e) {
      System.out.print("Tiedosto hukassa!");
    }
    catch (Exception e) {
      System.out.print("Lukuvirhe!");
    }
  }
}
```

Avaaminen kirjoittamista varten

1) Luodaan tiedosto-olio *File*-luokasta:

```
File tiedostoOlionNimi =
    new File(tiedostonNimi);
```

2) Liitetään tulostusvirta ohjelmasta tiedostoon:

```
FileOutputStream virtaOlionNimi =
    new FileOutputStream(tiedostoOlionNimi);
```

Kirjoittaminen

- Tulostusvirtaan kirjoittamiseen tarvitaan kirjoittajaolio *PrintWriter*-luokasta. Yleisesti:

  PrintWriter kirjoittajaOlionNimi =

  new *PrintWriter*(virtaOlionNimi, **true**);

- Kirjoittaminen tapahtuu *PrintWriter*-luokan *print*- ja *println*-operaatioiden avulla.
- Nämä operaatiot on kuormitettu kuten *System.out*-attribuutin kautta kutsuttavat operaatiot, jotka tulostavat esimerkiksi lukuja ja merkkijonoja.

Sulkeminen kirjoittamisen jälkeen

- Tiedoston kirjoittamisen jälkeen siihen liitetty syötevirta suljetaan kirjoittajan *close*-operaatiolla.
- Myös kirjoittamisen yhteydessä voi tapahtua poikkeus: tarvitaan poikkeuksen käsittelyä **try-catch**-lauseen avulla.

Tiedostot: esimerkki

```
import java.io.*;
public class Kirjoittaminen {
  public static void main(String [] args) {
    final String TIEDNIMI = "out.txt";
    final int HEIPPALKM = 10;
    try {
      // Luodaan tiedosto-olio
      File tiedosto =
      new File(TIEDNIMI);
      // Luodaan tulostusvirta ja
      // liitetään se tiedostoon
      FileOutputStream tulostusvirta =
      new FileOutputStream(tiedosto);
      // Luodaan virtaan kirjoittaja
      PrintWriter kirjoittaja =
      new PrintWriter(tulostusvirta, true);
      // Kirjoitetaan tiedostoon
      for (int i = 0; i < HEIPPALKM; i++)
        kirjoittaja.println("Heippa " + i);
      // Suljetaan tiedosto
      kirjoittaja.close();
    }
    catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

7. Hyvä ohjelmointitapa.

Yleistä

- Jaa pitkä koodi osiin.
- Käytä attribuutteja säästeliäästi.
- Kommentoi operaatiot ja attribuutit.
- Varaudu operaatioissa virheisiin.
- Tunne ohjelmointikieli.

Jaa pitkä koodi osiin

- Satoja tai tuhansia rivejä sisältävää ohjelmaa ei ole järkevää kirjoittaa yhdeksi “pötköksi”.
- On hyvä jakaa ohjelma helposti ymmärrettäviksi ja hallittavaksi osiksi (modulaarinen ohjelmointi).
- Operaatiot ovat pääasiallinen ositusmenetelmä.
  - Myös luokat ja pakkaukset ovat keinoja hallita isoja kokonaisuuksia.
- Osat ideaalisesti ympäristöstään riippumattomia.
  - Operaatioiden tulisi vuorovaikuttaa muiden operaatioiden kanssa vain parametrien ja paluuarvojen avulla.

Käytä attribuutteja säästeliäästi

- Olio-ohjelmoinnissa tärkeät attribuutit ovat ristiriidassa modulaarisuusperiaatteen kanssa, koska ne ovat “globaaleina” muuttujina käytettävissä kaikissa luokan operaatioissa.
  - Attribuutteja on käytettävä varoen.
  - Erityisesti toisessa harjoitustyössä ei saa olla (turhia) attribuutteja.
  - Vakiomuotoisia attribuutteja saa käyttää vapaammin, koska niiden arvoja ei voi muuttaa operaatioissa.

Kommentoi operaatiot ja attribuutit

- Operaatioihin liitetään yleisluontoiset kommentit operaation tarkoituksesta, parametreista ja paluuarvoista:

```
/* Tulostetaan kaksiulotteinen taulukko t.
 */
public static void tulosta(int[][] t) { … }
```

  - Lyhyitä parametritunnuksia käytettäessä parametrien tarkoitus on ehdottomasti kerrottava kommentissa.
  - Operaatioiden nimet ovat muuttujien nimiä pitempiä ja alkavat usein käskymuotoisella verbillä.
- Attribuutit kommentoidaan muuttujien tapaan:

```
// Ohjelman lopettava komento.
private static final String KOMENTO_LOPETA = "quit";
```

Varaudu operaatioissa virheisiin

- Maailma ei ole täydellinen – ohjelmasi ei esimerkiksi saa syötteitä juuri siten kuin toivoisit.
- Usein muutama yksinkertainen tarkistus tekee ohjelmasta huomattavasti vakaamman ja samalla miellyttävämmän käyttää.
- Hyvässä ohjelmassa on varauduttu virhetilanteisiin ja ohjelman toimintaa näissä tilanteissa on vieläpä testattu.
- Erityisesti operaatioiden viitetyyppiset parametrit on syytä tarkistaa.
  - Operaation kutsuminen **null**-arvoisen viitteen kautta aiheuttaa ohjelman pysäyttävän ajonaikaisen virheen.

Varaudu operaatioissa virheisiin

```
/* Täytetään taulukko t satunnaisluvuilla.
 */
public static void tayta(int[] t) {
   // Satunnaisluvut välillä [0, MAX[.
   final int MAX = 10;
   // Täytetään, jos on varattu muistia.
   if (t != null) {
      for (int i = 0; i < t.length; i++)
         t[i] = (int)(MAX * Math.random());
   }
}
```

Ilman **if**-lausetta ohjelman suoritus keskeytyy ajonaikaiseen virheeseen (*NullPointer-Exception*) aina, kun parametrille ei ole varattu muistia.

Varaudu operaatioissa virheisiin

```
// Tutkitaan ovatko merkkijonon m ensimmäinen
// ja viimeinen merkki sama merkki.
public static boolean ekaJaVikaSamat(String m) {
   // Arvataan, että merkit eivät ole samoja.
   boolean ovatSamat = false;
   // Edetään, jos viitteeseen liittyy olio
   // ja merkkijonossa on vähintään yksi merkki.
   if ((m != null) && (m.length() > 0))
      // Käännetään lippu, jos merkit ovat samat.
      if (m.charAt(0) == m.charAt(m.length() - 1))
         ovatSamat = true;
   // Palautetaan tulos.
   return ovatSamat;
}
```

Ilman ulompaa **if**-lausetta ohjelman suoritus keskeytyy ajonaikaiseen virheeseen, kun parametrille ei ole varattu muistia (*NullPointer-Exception*) tai kun merkkijonon pituus on nolla (*StringIndexOutOf-BoundsException*).

8. Rekursio

Johdanto

- Rekursiivinen operaatio kutsuu itseään.
- Rekursio etenee askelittain “alaspäin” kunnes kohdataan alkeistapaus (base case), joka voidaan ratkaista.
- Tämän jälkeen palataan “ylöspäin” yhdistäen kussakin askeleessa tehtyjen rekursiivisten kutsujen tulokset.
- Rekursion avulla voidaan ratkaista vaikeitakin ongelmia yksinkertaisesti.
  - Käytetään esimerkiksi lajittelualgoritmeissa.
- Rekursio on haastava ohjelmointitekniikka.
  - Ikuinen rekursio mahdollinen.
- Rekursio on laskennallisesti tehoton menetelmä.
- Luonteva ratkaisu vain osaan ongelmista.

Esimerkki

- Tarkastellaan esimerkkinä luvun n kertoman $n!$ ($n \geq 0$) laskemista sekä iteratiivisesti että rekursiivisesti .
- Määritellään erikoistapaukset: $0! = 1$ ja $1! = 1$.
- Iteratiivinen ratkaisu:
  - $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$
  - $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$
- Rekursiivinen ratkaisu:
  - $n! = n \cdot (n - 1)!$
  - $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1!$

  $= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

Esimerkki

```
/* Lasketaan luvun n kertoma iteratiivisesti.
 */
public static int laskeKertoma(int n) {
   // 0! = 1 ja 1! = 1.
   int kertoma = 1;
   // Silmukoidaan, jos n > 1.
   while (n > 1) {
      // Päivitetään tuloa ja tuotetaan seuraava kerrottava.
      kertoma = kertoma * n;
      n = n - 1;
   }
   // Palautetaan tulos.
   return kertoma;
}
```

Esimerkki

```
/* Lasketaan luvun n kertoma rekursiivisesti.
 */
public static int laskeKertoma(int n) {
   // Käytetään perustapauksena
   // luvun 1 kertomaa.
   if (n <= 1)
      return 1;
   // Palautetaan rekursiivisen kutsun tulos
   // luvulla n kerrottuna.
   else
      return n * laskeKertoma(n - 1);
}
```