Sisällys

Tehtävät:

Kuvat:

Malliohjelmat:

Lukijalle

Alkuun annan sulle vinkin,
joutavia on juorut muiden:

Luppo loppui, alkoi arki,
kutsuu koulu - niinkö luulet?
Uskoppas: YLIOPISTO
vaatii työtavat totiset!

Ostolla oppi ei tulene
eikä kauhan kaadannalla,
myös ei vastuun välttämällä,
työnsä muilla teettämällä

Jos sun mielesi tekevi,
aivosi odottelevi,
vilauttaa vinkin voinen,
opastukset ongelmille.

Pakko tehdä on demoja,
harjoitukset harjoitella,
itse illoin ihmetellä,
kovin koodailla kotona.

Harjoitustyö haastavasta,
syntyy aiheesta omasta,
kokonaisuuden kuvaksi,
metsän puilta mieltäväksi.

Luontune ei kurssi yksin:
moni meitä auttamassa,
ohjaajat opastamassa,
valistaen vaadittaissa.

Toki saarnailen salissa,
kerron joukolle jotakin,
esimerkkejä esitän
ynnä vaiheita valotan.

Tämä moniste on tarkoitettu oheislukemistoksi Ohjelmointi 2-kurssille. Vaikka monisteen yksi teema onkin *Java*–kieli, ei kieli ole monisteen päätarkoitus. Päätarkoituksena on esitellä ohjelmointia. Esitystavaksi on valittu yhden ohjelman suunnitteleminen ja toteuttaminen alusta lähes loppuun saakka. Tämä *Top-Down* –metodi tuottaa varsin suuren kokonaisuuden, jonka hahmottaminen saattaa aluksi tuntua vaikealta.

Kunhan oppii kuitenkin katsomaan kokonaisuuksiin yksityiskohtien sijasta, asia helpottuu. Yksityiskohtia harjoitellaan monisteen esimerkeissä (*Bottom-Up*), joista suuri osa liittyy monisteen malliohjelmaan, mutta jotka silti voidaan käsittää mallista riippumattomina palasina.

Monisteen ohjelmat on saatavissa myös elektronisesti, jotta niiden toimintaa voidaan kokeilla kunkin vaiheen jälkeen.

Java–kieltä ja sen ominaisuuksia on monisteessa sen verran, että lukijalla on juuri ja juuri erittäin pienet mahdollisuudet selvitä ilman muuta kirjallisuutta.

Lukijan kannattaakin ilman muuta hankkia ja seurata tämän monisteen rinnalla jotakin varsinaista *Java*–ohjelmointikirjaa. Olio-ohjelmoinnista on kansantajuinen teos: **Timothy A. Budd:** *Understanding Object Oriented Programming with Java*, Updated Edition - Addison Wesley, 2000. Usein myös ohjelmointiympäristön mukana olevasta *OnLine*–avustuksesta (*Help*) saa tarvittavaa lisätietoa.

Monisteen esimerkkiohjelmat löytyvät elektronisessa muodossa:

Mikroluokka:	hakemisto:	n:\kurssit\ohj2\moniste\esim
WWW:	URL:	http://www.mit.jyu.fi/vesal/kurssit/ohj2/moniste/esim

Edellä mainittuun polkuun lisätään vielä ohjelman yhteydessä mainittu polku.

Monisteessa on lukuisia esimerkkitehtäviä, joiden tekeminen on oppimisen kannalta lähes välttämätöntä. Vaikka lukija saattaa muuta kuvitellakin, on monisteen vaikeimmat tehtävät monisteen alussa. Mikäli loppupuolen tehtävät tuntuvat vaikeilta, ei monisteen alkuosa olekaan hallinnassa. Siksi kehotankin lukijaa aina vaikeuksia kohdatessaan palaamaan monisteen alkuosaan; siitä ei monikaan voi sanoa, ettei asioita ymmärtäisi.

Lopuksi kiitokset kaikille työtovereilleni monisteen kriittisestä lukemisesta. Erityisesti Tapani Tarvainen on auttanut suunnattomasti C–kieleen tutustumistani ja Jonne Itkonen vastaavasti tutustumista Java ja C++–kieleen ja olio–ohjelmointiin.

Alkuperäinen versio allekirjoitettu Palokassa 28.12.1991

Monisteen 3. korjattuun painokseen on korjattu edellisissä monisteissa olleita painovirheitä sekä lisätty lyhyt C–kielen "referenssi". Lisäksi kunkin esimerkkiohjelman alkuun on laitettu kommentti siitä, mistä tiedostosta lukija löytää esimerkin. Myös hakemistoa on parannettu vahventamalla määrittelysivun sivunumero.

Palokassa 28.12.1992

Monisteen 4. korjattuun painokseen on jopa vaihdettu monisteen nimi: *Ohjelmointi++*, kuvaamaan paremmin olio–ohjelmoinnin ja *C++*:n saamaa asemaa. Tätä kirjoittaessani moniste ei ole vielä kokonaan valmis ja kaikkia siihen tulevia muutoksia en vielä tässä pysty luettelemaan.

Joka tapauksessa olen monisteeseen lisännyt tekstiä – valitettavasti nimenomaan ohjelmointikieleen liittyvää – jota vuosien varrella olen huomannut opiskelijoiden jäävän kaipaamaan. Lisäksi kunkin luvun alkuun on lisätty suppea luettelo luvun pääteemoista ja luvussa esiintyvistä kielen piirteistä sekä niiden syntaksista. Tämä syntaksilista on helppolukuinen "lasten" syntaksi, varsinainen tarkka ja virallinen syntaksi pitää katsoa kielen määrityksistä.

Ohjelmalistauksiin on lisätty *syntax-highlight*, eli kielen sanat on korostettu ja näin lukijan toivottavasti on helpompi löytää mitkä termit on itse valittavissa ja mitkä täytyy kirjoittaa juuri kuvatulla tavalla. Myös joitakin vinkkejä on lisätty. Pedagogisesti on vaikea päättää saako esittää virheellisiä tai huonoja ohjelmia lainkaan, mutta vanha viidakon sananlasku sanoo että *"Viisas oppii virheistä, tavallinen kansa omista virheistä ja tyhmä ei niistäkään*". Siis mahdollisuus "viisaillekin" ja nämä virheelliset ohjelmat on merkitty surullisella naamalla: ☹. Näin aivan jokaisen ei tarvitse rämpiä jokaista sudenkuoppaa pohjia myöten ominpäin.

Olio-ohjelmointi on kuvattu esimerkkien avulla ja varsinainen oliosuunnittelu - joka on erittäin tärkeää – on jätetty erittäin vähälle. Suosittelenkin lukijalle jonkin oliosuunnitteluun liittyvän kurssin käymistä tai kirjan lukemista.

Myös tätä kurssia edeltävä kurssi *Ohjelmoinnin alkeet* on kokenut muutoksia ja vaikka kurssi meneekin nykyisin entistä pitemmälle, ei tästä monisteesta ole kuitenkaan poistettu kaikkea päällekkäisyyttä *Ohjelmoinnin alkeet* –monisteen kanssa. Toimikoon nämä päällekkäisyyden kertauksena ja kohtina, joissa luennoilla voidaan asia sivuttaa nopeammin. Joka tapauksessa lukijan kannattanee pitää myös *Ohjelmoinnin alkeet* –moniste tämän monisteen rinnalla.

Palokassa 5.1.1997

Monisteen 5. korjattuun painokseen on korjattu pieniä kirjoitusvirheitä ja epätäsmällisyyksiä. Samalla on poistettu hieman C-esimerkkejä ja yritetty enemmän jättää jäljelle esimerkkejä siitä, kuinka käytännössä kannattaa tehdä. Nyt erityisesti kurssia myös pitänyt Antti-Juhani Kaijanaho on antanut merkittävästi palautetta monisteesta.

Palokassa 30.12.2001

Monisteen uusi painos on kirjoitettu C++:n sijasta Java-ohjelmointia silmällä pitäen.

Monisteessa olevat Kalevala-mittaiset runot ovat syntaksin mukaisia. Lukija voi itse päättää riittääkö se. Sama pätee ohjelmoinnissa. Syntaktisesti oikea ohjelma on vielä kaukana toimivasta ohjelmasta.

Palokassa 30.12.2002

Vesa Lappalainen

1. Johdanto

Alkoi kurssi, alkoi uusi
tuska tuli, moni jo huusi:
Javaa jankuttaa tuo ukko
syntaksia sammaltaapi.

Tokko tavalla tuollasella
ohjelmoimaan oppimahan
Java kieltä pänttämähän
Ceetä kalloon taikomahan.

Arvelee, ajattelevi,
pitkin päätänsä pitävi:
Ei oo ulkoo oppimista,
kieli väkisin vääntämistä.

Pohtimaan pitää heretä
ongelmia oikomahan
sulamahan suunnittelu
pohja vankaksi valaman.

Tämän monisteen tarkoituksena on toimia tukimateriaalina opeteltaessa sekä algoritmisen että olio–ohjelmoinnin alkeita. Aluksi meidän tulee ymmärtää mitä kaikkea ohjelmointi pitää sisällään. Aivan liian usein ohjelmointi yhdistetään päätteen äärellä tapahtuvaan jonkin tietyn ohjelmointikielen koodin naputtamiseen. Tämä on ehkä ohjelmoinnin näkyvin osa, mutta myös toisaalta mekaanisin ja helpoin osa.

Ohjelmointi voidaan jakaa esimerkiksi seuraaviin vaiheisiin:

- tehtävän saaminen
- tehtävän tarkentaminen ja tarvittavien toimintojen hahmottaminen
- ohjelman toimintojen ja tietorakenteiden suunnittelu, oliosuunnittelu
- yksityiskohtaisten olioiden ja algoritmien suunnittelu
- OHJELMOINTITYÖKALUN VALINTA
- algoritmien/luokkien metodien tarkentaminen valitulle työkalulle
- ohjelmakoodin kirjoittaminen
- ohjelman testaus
- ohjelman käyttöönotto
- ohjelman ylläpito

Huomattakoon, että edellisessä listassa varsinaisesti tietokoneella tehtävä työ on vain aivan listan viimeisissä kohdissa. Tietenkin nykyisin suunnittelun alkuvaiheessakin tarvittava dokumentointi ja ideoiden sekä vaihtoehtojen kirjaaminen tehdään käyttäen tekstinkäsittelyohjelmia ja/tai kaavioiden piirtoa piirto-ohjelmilla. Varsinaisesta koodauksesta ei kuitenkaan alkuvaiheessa ole kysymys.

Ohjelman kehityksen eri vaiheissa saatetaan tarvittaessa palata takaisin alkumäärityksiin. Kuitenkin ohjelman valittujen toimintojen muuttaminen oman laiskuuden tai osaamattomuuden takia ei ole suotavaa. Ei saa lähteä ompelemaan kissalle takkia ja huomata, että kangas riittikin lopulta vain rahapussiin.

1.1 Ohjelman suunnittelu

Usein ohjelmointikursseilla unohdetaan itse ohjelmointi ja keskitytään valitun työkalun – ohjelmointikielen – esittelyyn. Ajanpuutteen takia tämä onkin osin ymmärrettävää. Kuulijat kuitenkin hämääntyvät eivätkä ymmärrä luennoitsijan tekevän edellä kuvatun listan kaltaista suunnittelutyötä myös kunkin pienen malliesimerkin kohdalla. Kokenut ohjelmoija saattaa pystyä hahmottamaan ongelman ratkaisun ja tarvittavat erikoistapaukset päässään silloin, kun on kysy erittäin lyhyistä malliesimerkeistä. Jossain vaiheessa ohjelmoinnin oppimista suunnittelu ja koodin kirjoittaminen tuntuvat sulautuvan yhteen.

Opiskelun alkuvaiheessa on kuitenkin syytä keskittyä nimenomaan ongelman analysointiin ja ohjelman suunnitteluun. Tässä paras apu on usein terve maalaisjärki. Mitä vähemmän ymmärtää itse ohjelmointikielistä, sitä vähemmän kielet rajoittavat luovaa ajattelua.

Usein ohjelman suunnittelu voidaan aloittaa jopa käyttöohjeen kirjoittamisella! Tällöin tulee tutkituksi ohjelmalta vaaditut ominaisuudet ja toimintojen loogisuus sekä helppokäyttöisyys! Nykytyökaluilla voidaan myös rakentaa suhteellisen helposti ensin ohjelman käyttöliittymä ilman oikeita toimintoja. Tätä "protoa" voidaan sitten tutkia yhdessä asiakkaan kanssa ja päättää toimintojen loogisuudesta ja riittävyydestä.

1.2 Työkalun valinta

Kun ohjelmaan on suunniteltu halutut toimenpiteet ja päätetty mitä tietorakenteita tarvitaan, on edessä työkalun valinta. Nykypäivänä ei ole itsestään selvää, että valitaan työkaluksi jokin perinteinen ohjelmointikieli. Vastakkain pitää asettaa erilaiset sovelluskehittimet, valmisohjelmat kuten tietokannat ja taulukkolaskennat, ehkä jopa tavallinen tekstinkäsittely sekä ohjelmointikielet. Matemaattisissa ongelmissa jokin symbolisen tai numeerisen laskennan paketti saattaa olla soveltuva.

Ratkaisu voi koostua myös useiden eri ohjelmien toimintojen yhdistelemisestä: CAD – ohjelmalla piirretään/digitoidaan kartan pohjakuva, tietokantaohjelmalla pidetään kirjaa paikoista ja pienellä C/C++ tai Java-kielisellä ohjelmalla suoritetaan ne osat, joita CAD-ohjelmalla tai tietokantaohjelmalla ei voida suorittaa.

Joskus työkaluksi valitaan prototyyppiä varten jokin sovelluskehitin tai tietokantaohjelmisto. Kun halutut toiminnot on perusteellisesti testattu ja tuotetta tarvitsee edelleen kehittää, voidaan ohjelmointi toteuttaa uudelleen vaikkapa Java–kielellä. Prototyyppi on rinnalla toimivana ja uudessa ohjelmassa käytetään samoja tietoja ja toimintoja.

1.3 Koodaus

Mikäli työkalun valinnassa päädytään olio/lausekieleen (esim. C++ tai Java), ei pyörää kannata keksiä uudelleen. Nelikulmioon nähden kolmikulmiossa on yksi poksaus vähemmän kierroksella, mutta kyllä silti ympyrä on paras. Siis käytetään toisten kirjoittamia valmiita olioita ja/tai aliohjelmapaketteja "likaisessa" työssä.

Aina tietenkin puuttuu joitakin alemman tason palasia. Nämä tietysti koodataan JA TESTATAAN ERILLISINÄ ennen varsinaiseen ohjelmaan liittämistä.

Siis itse koodaus on pienten aputyökalujen etsimistä, tekemistä, testaamista ja dokumentointia. Lopullinen koodaus on näiden aputyökaluista muodostuvan palapelin yhteen liittäminen.

Jo koodausvaiheessa kannattaa miettiä ongelman yleisiä ominaisuuksia. Jos ollaan kirjoittamassa telinevoimistelun pistelaskua naisten sarjaan, niin koodissa ei mitenkään tulisi estää ohjelman käyttöä myös miesten sarjassa. Siis telineiden nimet ja määrät pitäisi olla helposti muutettavissa.

Koodausta voidaan tehdä joko *BOTTOM–UP* periaatteella, jolloin ensin rakennetaan työkalut (=olioluokat/aliohjelmat) jotka sitten kasataan yhteen. Toinen mahdollisuus on koodaus *TOP–DOWN* periaatteella, jolloin päätoiminnat kirjoitetaan ensin ja alatoiminnoista tehdään aluksi tyhjiä laatikoita. Myöhemmin valmiita ja testattuja alitoimintoja liitetään tähän runkoon. Valitulla menetelmällä ei ole vaikutusta lopputulokseen ja joskus voikin olla hyvää vaihtelua siirtyä näpertelemään pikkuasioiden kimpussa isojen kokonaisuuksien sijasta tai päinvastoin.

Missään tapauksessa ohjelma ei synny siten kuin se kirjallisuudessa näyttää olevan: alkumäärittelyt, aliohjelmat ja päämoduuli.

Koodaajan on osattava hyvin käytettävä työkalu, esim. ohjelmointikieli. Kuitenkin jonkin ohjelmointikielen hyvän osaamisen avulla on suhteellisen helppo kirjoittaa myös muunkielisiä ohjelmia.

Koodaus on pääosin tekstinkäsittelyä ja 10–sormijärjestelmä nopeuttaa koodin syntymistä oleellisesti. Myös hyvä tekstinkäsittelytaito valmiiden palasten siirtelemisineen ja kopioimisineen helpottaa tehtävää.

1.4 Testaus

Ohjelman testaus alkaa jo suunnitteluvaiheessa. Valitut algoritmit ja toiminnot pitää pöytätestata teoriassa ennen niiden koodaamista. Suunnitteluvaiheessa täytyy miettiä kaikki mahdolliset erikoistapaukset ja todeta algoritmin selviävän niistäkin tai ainakin määritellä miten erikoistapauksissa menetellään. Testitapaukset kirjataan ylös myöhempää käyttöä varten.

Koodausvaiheessa kukin yksittäinen aliohjelma/luokka testataan kaikkine mahdollisine syötteineen pienellä testiohjelmalla. Aliohjelman kommentteihin voidaan kirjata suunnitteluvaiheessa todettu testiaineisto ja testausvaiheessa ruksataan testatut toiminnot ja erikoistapaukset. Nykyisin on jopa testausta automatisoivia työkaluja, joilla koko projekti voidaan testata uudelleen kun sen johonkin osaan tehdään muutoksia.

Lopullisen ohjelman toimivuus riippuu hyvin paljon siitä, miten hyvistä palasista se on kasattu.

Testauksessa apuna on aluksi pöytätestit. Sitten käytetään pieniä testiohjelmia. Ennen testiohjelmiin lisättiin tulostuslauseita. Nykyisin tehokkaat debuggerit helpottavat testausta huomattavasti: ohjelman toimintaa voidaan seurata askel kerrallaan ja epäilyttävien muuttujien arvoja voidaan tarkistaa kesken suorituksen. Voidaan myös laittaa ohjelma pysähtymään jonkin muuttujan saadessa virheellisen arvon.

Testaus on vaihe, missä hyvä koneenkäyttörutiini ja epäluulo ovat suureksi avuksi.

1.5 Käyttöönotto

Valmiin ohjelman käyttöönotto tapahtuu yleensä aina liian aikaisin. Paineet keskeneräisesti testatun ohjelman käyttämiseksi ovat suuret. Lisäksi testaajat ja erityisesti ohjelman koodaajat ovat sokeita tavallisen käyttäjän (usein myös omilleen) virheille.

Käyttöohjeen olisi syytä olla valmis viimeistään tässä vaiheessa.

1.6 Ylläpito

Käyttöönotetusta ohjelmasta paljastuu aina virheitä tai puuttuvia toimintoja. Virheet pitää korjata ja puuttuvat toiminnot mahdollisesti lisätä ja ollaan jälleen ohjelmansuunnittelun alkuvaiheessa. Hyvin suunniteltuun ohjelmaan saattaa olla helppo lisätä uusia toimintoja ja vastaavasti huonosti suunnitellussa saattaa jopa tietorakenteet mennä uusiksi.

Myös ohjelman alkuperäiset kirjoittajat ovat saattaneet häipyä ja kesätyöntekijä joutuu ensitöikseen paikkaamaan toisten huonosti dokumentoitua sotkua.

1.7 Yhteenveto

Ohjelmointi ei yleensä ole yhden henkilön työtä. Eri henkilöt voivat tehdä eri vaiheita ohjelmoinnissa. Lähes aina tulee tilanne, missä jonkin toisen kirjoittamaa koodia joudutaan korjailemaan.

Oli ohjelmaa tekemässä kuinka monta henkilöä tahansa (vaikka vain yksi), pitää ohjelmointi jakaa vaiheisiin. Oikeaa ohjelmaa on mahdoton "nähdä" valmiina Java-kielisinä lauseina heti tehtävän määrityksen antamisen jälkeen. Aloitteleva ohjelmoija kuitenkin haluaisi pystyä tähän (koska hän "näkee" määrityksestä: Kirjoita ohjelma joka tulostaa "*Hello world*", heti myös Java-kielisen toteutuksen). Tämän takia ohjelmoinnin helpoin osa, eli koodaus koetaan ohjelmoinnin vaikeimmaksi osaksi – suunnittelu on unohtunut!

Valitulla ohjelmointikielellä ei ole suurtakaan merkitystä ohjelmoinnin toteuttamiseen. Jokin kieli saattaa soveltua paremmin johonkin tehtävään, mutta pääosin *BASIC, Fortran, Pascal, C, Modula–2, ADA* jne. ovat samantyylisiä lausekieliä. Samoin oliokielistä esimerkiksi *C++, Java, Delphi (Pascal) ja Python* ovat hyvin lähellä toisiaan. Kun yhden osaa, on toiseen siirtyminen jo helpompaa.

Jos joku kuvittelee ettei hänen tarvitse koskaan ohjelmoida *C/C++* tai *Java*–kielellä, voi hän olla aivan oikeassakin. Nykyisin kuitenkin jokaisessa tietokantaohjelmassa, taulukkolaskentaohjelmassa ja jopa tekstinkäsittelyohjelmissakin (vrt. esim. T_EX, joka on tosin ladontaohjelma) on omat ohjelmointikielensä. Osaamalla jonkin ohjelmointikielen perusteet, voi saada paljon enemmän hyötyä käyttämästään valmisohjelmasta. Ja joka väittää selviävänsä nykymaailmassa (ja sattuu lukemaan tätä monistetta) esimerkiksi ilman tekstinkäsittelyohjelmaa on suuri valehtelija!

2. Kerhon jäsenrekisteri

Nyt tavuja taikomahan,
koodia kokoamahan?
Tuosta tokkopa tulisi
ohjelmaapa oivallista.

Ongelma jo täytyy olla
suunnitelma siivitellä
aikeet aina aatostella
toki tarpeet tarkastella.

Saatatko tuon jo sanoa
tieto kusta tarvitahan
ohjelman ositeltavan
jo bitteiksi pilkottavan.

Nyt liimaile liittymätä
sitä silmälle suotavaksi
käyttäjälle nähtäväksi
muille mutristeltavaksi.

Mitä tässä luvussa käsitellään?

- tehtävän "analysointi"
- ohjelman vaatimien aputiedostojen sisällön suunnittelu
- ohjelman suunnittelu ohjelman tulosteiden avulla
- suunnitelman korjaus
- tarvittavien algoritmien hahmottaminen
- relaatiotietomalli

2.1 Tehtävän tarkennus

Ohjelman suunnittelu aloitetaan aina tehtävän tarkastelulla. Annettua tehtävää joudutaan usein huomattavasti tarkentamaan.

Olkoon tehtävänä suunnitella kerhon jäsenrekisteri. Onko kerho iso vai pieni? Mitä tietoja jäsenistä tallennetaan? Mitä ominaisuuksia rekisteriltä halutaan?

Mikäli sovitaan, että kerho on kohtuullisen pieni (esim. alle 500 jäsentä), ei meidän heti alkuun tarvitse miettiä parhaita mahdollisia hakualgoritmeja eikä tiedon tiivistämistä.

Mitä tietoja jäsenistä tarvitaan?

```
- nimi
- sotu
- katuosoite
- postinumero
- postiosoite
- kotipuhelin
- työpuhelin
- autopuhelin
- liittymisvuosi
- tämän vuoden maksetun jäsenmaksun suuruus
- lisätietoja
jne...
```

Mitä ominaisuuksia rekisteriltä halutaan?

```
- kerholaisten lisääminen
- kerholaisten poistaminen
- tietyn kerholaisen tietojen hakeminen
- tietyn kerholaisen tietojen muuttaminen
- postitustarrat postinumerojärjestyksessä
- nimilista nimen mukaisessa järjestyksessä
- lista jäsenmaksua maksamattomista jäsenistä
jne...
```

2.2 Työkalun valinta

On varsin selvää, ettei tätä nimenomaista tehtävää kannattaisi nykypäivänä lähteä itse ohjelmoimaan, vaan turvauduttaisiin tietokantaohjelmaan. Joissakin erikoistapauksissa saatetaan vaatia ominaisuuksia, joita tietokantaohjelmasta ei saada. Tällöin työkaluksi valittaisiin lausekieli ja tietokantaohjelmiston aliohjelmakirjasto, joka hoitelee varsinaiset tietokannan ylläpitoon yms. liittyvät toimenpiteet.

Edellinen analyysi on kuitenkin tehtävä työkalusta riippumatta! Esimerkin vuoksi jatkamme tehtävän tutkimista hieman pidemmälle tavoitteena ohjelmoida jäsenrekisteri jollakin lausekielellä.

2.3 Tietorakenteet ja tiedostot

Mikäli työkalun valinnassa on päädytty johonkin lausekieleen, on jossain vaiheessa päätettävä käytettävistä tietorakenteista. Esimerkin tapauksessa meillä on selvästikin joukko yhden henkilön tietoja. Mikäli yhden henkilön tietoa pidetään yhtenä yksikkönä (tietueena), on koko tietorakenne taulukko henkilöiden tiedoista. Taulukko voidaan tarvittaessa toteuttaa myös lineaarisena listana tai jopa puurakenteena. Mikäli kyseessä on pieni rekisteri, mahtuu koko tietorakenne ohjelman ajon aikana muistiin.

Missä tiedot tallennetaan kun ohjelma ei ole käynnissä? Tietenkin levyllä tiedostona. Minkä tyyppisenä tiedostona? Tiedoston tyyppinä voisi olla binäärinen tiedosto alkioina henkilötietueet. Tällaisen tiedoston käsittely hätätapauksessa on kuitenkin vaikeata. Varmempi tapa on tallentaa tiedot tekstitiedostoksi, jota tarvittaessa voidaan käsitellä millä tahansa tekstinkäsittelyohjelmalla. Tällöin on lisäksi usein mahdollista käsitellä tiedostoa taulukkolaskentaohjelmalla tai tietokantaohjelmalla ja näin joitakin harvinaisia toimintoja voidaan suorittaa rekisterille vaikkei niitä olisi alunperin edes älytty laittaa ohjelmaan mukaan.

Minkälainen tekstitiedosto? Ehkäpä yhden henkilön tiedot yhdellä rivillä? Miten yhden henkilön eri tiedot erotetaan toisistaan? Mahdollisuuksia on lähinnä kaksi: erotinmerkki tai tietty sarake. Valitaan erotinmerkki. Usein on mukavaa lisäksi laittaa joitakin

huomautuksia eli kommentteja tiedostoon. Siis tallennustiedoston muoto voisi olla vaikkapa seuraava:

kelmit.dat - ensimmäinen ehdotus tiedostoksi

```
; Tässä tiedostossa on KELMIT Ry:n jäsentiedot
; Kenttien järjestys tiedostossa on seuraava:
; sukunimi etunimi|sotu|katuosoite|postinumero|postiosoite
;|kotipuhelin|työpuhelin|autopuhelin|liittymisvuosi|jmaksu|lisätietoja
Ankka Aku|010245-123U|Ankkakuja 6|12345|ANKKALINNA|12-12324|||1991|50|velkaa Roopelle
Susi Sepe|020347-123T||12555|Takametsä||||1990|50|jäsen myös kelmien kerhossa
Ponteva Veli|030455-3333||12555|Takametsä||||50|1989|
```

Tällaisenaan tiedosto on varsin suttuinen luettavaksi. Vaikka valitsimmekin erotinmerkin erottamaan tietoja toisistaan, voimme silti kirjoittaa vastaavat tiedot allekkain sopimalla, ettei loppuvälilyönneillä ole merkitystä.

kelmit.dat - sarakkeet linjaan

```
; Tässä tiedostossa on KELMIT Ry:n jäsentiedot
; Kenttien järjestys tiedostossa on seuraava:
; sukunimi etunimi |sotu       |katuosoite  |postinumero|postiosoite|kotipuhelin|työpuhelin|
Ankka Aku          |010245-123U|Ankkakuja 6 |12345      |ANKKALINNA |12-12324   |          |
Susi Sepe          |020347-123T|            |12555      |Takametsä  |           |          |
Ponteva Veli       |030455-3333|            |12555      |Takametsä  |           |          |
```

Nyt tiedostoa on helpompi lukea ja tyhjien kenttien jättäminen ei ole vaikeaa. Tiedosto vie kuitenkin levyltä enemmän tilaa kuin ensimmäinen versio. Lisäksi yhden henkilön tiedot eivät mahdu kerralla näyttöön. Onneksi kuitenkin lähes kaikki nykyiset tekstieditorit suostuvat rullaamaan näyttöä myös sivusuunnassa. Mikäli saman henkilön tietoja jaettaisiin eri riveille, tarvitsisi meidän valita vielä tietueen loppumerkki (nytkin se on valittu: **rivinvaihto**).

2.4 Käyttöohje tai käyttöliittymä

Jatkosuunnittelu on ehkä helpointa tehdä suunnittelemalla ohjelman toimintaa käyttöohjeen tai käyttöliittymän tavoin. Erityisesti graafisella käyttöliittymällä varustettuja ohjelmia suunnitellaan nykytyökaluilla nimenomaan "piirtämällä" käyttäjälle näkyvä käyttöliittymän osa. Tähän osaan sitten lisätään heti tai jälkeenpäin itse toiminnallisuus. Tällaisia työkaluja on esimerkiksi *JBuilder*, *NetBeans*, *C++Builder*, *Delphi*, *Visual Basic* ja myös muiden ohjelmointikielten resurssityökalut.

Graafisen käyttöliittymän suunnittelu ja toteutus on kuitenkin jo edistyneempää - ja usein laiteriippuvaa - puuhaa, joten tällä kurssilla tyydymme aluksi "vanhanaikaiseen" keskustelevaan tekstikäyttöliittymään. Jos koodausvaiheessa riittävästi erotetaan käyttöliittymään liittyvä koodi tietorakennetta ylläpitävästä koodista, voidaan ohjelma kohtuullisella työllä muuttaa myös graafisessa käyttöliittymässä toimivaksi.

Suunnittelussa toimitaan käyttäjän ja helppokäyttöisyyden (= myös nopea käyttö, ei aina välttämättä hiiri) ehdoilla.

Ohjelmassa on kahdenlaisia vastauksia. Toisiin riittää painaa pelkkä yksi kirjain tai numero (kuten menut ja K/e tyyppiset valinnat). Mikäli vastaukseen on mahdollista kirjoittaa enemmän kuin yksi merkki, pitää vastaus lopettaa [RET]–näppäimen painamisella (Return, Enter). Jatkossa hoputteilla (*prompt*) on seuraavat merkitykset

: odotetaan pelkkää yhtä merkkiä. Mikäli vaihtoehdot on lueteltu ja jokin niistä on isolla kirjaimella, valitaan tämä painettaessa [RET]–näppäintä.

> odotetaan 0 – useata merkkiä ja [RET]–näppäintä. Mikäli hoputteen edessä on suluissa jokin arvo, tulee tämä arvo vastauksen arvoksi painettaessa pelkkää [RET]–näppäintä. Mikäli oletusvastaus on epätyhjä ja halutaan antaa vastaukseksi tyhjä merkkijono, painetaan välilyönti ja [RET].

Ohjelman päävalintaan päästään usein vastaamalla pelkkä [RET] uuden kierroksen alussa tai painamalla q[RET] missä tahansa ohjelman kohdassa.

Kun ohjelma käynnistyy, tulostuu näyttöön:

```
      ###################################
      #   J Ä S E N R E K I S T E R I   #
      #     versio 9.95                 #
      #     Hannu Hanhi                 #
      ###################################

Tällä ohjelmalla ylläpidetään kerhon jäsenrekisteriä.
Anna kerhon nimi>_
```

Kerhon tiedot on tallennettu vaikkapa tiedostoon nimi.DAT. Näin voimme ylläpitää samalla ohjelmalla useiden eri kerhojen tietoja. Mitäpä jos tiedostoa ei ole? Tällöin voi syynä olla kirjoitusvirhe tai se, ettei rekisteriä ole vielä edes aloitettu! Miten ohjelman tulee tällöin menetellä?

```
Tällä ohjelmalla ylläpidetään kerhon jäsenrekisteriä.
Anna kerhon nimi>KERMIT[RET]
Kerhon KERMIT tietoja ei ole!
Luodaanko tiedot (K/e):e
Anna kerhon nimi>KELMIT[RET]
Odota hetki, luetaan tietoja...

Jäsenrekisteri
==============

Kerhossa KELMIT on 52 jäsentä.

Valitse:
   0  = lopetus
   1  = lisää uusi jäsen
   2  = etsi jäsenen tiedot
   3  = tulosteet
   4  = tietojen korjailu
   :_
```

Edellä on edetty siihen saakka, kunnes ohjelmassa on päädytty päävalikkoon (*main menu*). Seuraavaksi voimme lähteä tarkastelemaan eri alakohtien toimintaa. Näissä eri erikoistapaukset on otettava huomioon:

2.4.1 Lisää uusi jäsen

```
1. Uuden jäsenen lisäys
=======================
Jäseniä on nyt 52.
Anna uusi nimi muodossa sukunimi etunimi etunimi
Jäsenen nimi      ()            >Ankka Aku[RET]
Rekisterissä on jo jäsen Aku Ankka!  Mikäli haluat muuttaa
hänen tietojaan, valitse tietojen korjailu päävalikosta!
Jäsenen nimi      ()            >ANKKA TUPU[RET]
Sotu              ()            >012356-1257[RET]
Sotu mieletön tai tarkistusmerkki väärin (N)!
Sotu              (012356-125N) >010356-125J[RET]
Katuosoite        ()            >Ankkakuja 6[RET]
Postinumero       (12345)       >[RET]
Postiosoite       (ANKKALINNA)  >[RET]
Kotipuhelin       ()            >12-12324[RET]
Työpuhelin        ()            >[RET]
Autopuhelin       ()            >[RET]
Liittymisvuosi    (91)          >91[RET]
Jäsenmaksu mk     (50)          >10[RET]
Lisätietoja       ()            >Aku Ankan veljenpoika[RET]

Lisätäänkö
  Ankka Tupu 010356-125J
  Ankkakuja 6 12345 ANKKALINNA
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 10 mk.
  Aku Ankan veljenpoika
:k

Jäseniä on nyt 53.

Anna uusi nimi muodossa sukunimi etunimi etunimi
Jäsenen nimi      ()            >[RET]

Jäsenrekisteri
==============

Kerhossa KELMIT on 53 jäsentä.

Valitse:
   0  = lopetus
   1  = lisää uusi jäsen
   2  = etsi jäsenen tiedot
...
```

Edellä on suluissa esitetty oletusarvo, joka tulee kentän arvoksi, mikäli painetaan pelkkä [RET]. Mikäli oletusarvon tilalle halutaan vaikkapa tyhjä merkkijono, vastataan välilyönti ja [RET]. Näin voidaan nopeuttaa tiettyjen samojen asioiden syöttöä. Oletusarvo voidaan antaa joko ohjelmasta väkisin (kuten tyhjä sotuksi) tai se voidaan ottaa edellisen syötön perusteella.

2.4.2 Etsi jäsenen tiedot

Tietoa voidaan hakea usealla eri tavalla. Voidaan haluta etsiä nimellä, osoitteella tai jopa puhelinnumerolla. Myös lisätietokentästä voidaan haluta etsiä tiettyä sanaa.

Siis aluksi pitää valita minkä kentän mukaan haetaan. Kun kenttä on selvitetty, voi olla mielekästä voida käyttää myös jokerimerkkejä (esim. ? ja *, vrt. MS-DOS.).

```
2. Etsi jäsenen tiedot
======================

  Nykyinen henkilö:

  Ankka Lupu 010356-127L
  Ankkakuja 6 12345 ANKKALINNA
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 10 mk.
  Aku Ankan veljenpoika, sudenpentu

Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.) :?

  1 = nimi
  2 = sotu
  3 = katuosoite
  4 = postinumero
  5 = postiosoite
  6 = kotipuhelin
  7 = työpuhelin
  8 = autopuhelin
  9 = liittymisvuosi
  A = jäsenmaksu mk
  B = lisätietoja

Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.) :1
Jäsenen nimi      (Ankka Lupu)  >*ANKKA*[RET]
Tähän täsmää 3 jäsentä:

  Ankka Aku 010245-123U
  Ankkakuja 6 12345 ANKKALINNA
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 50 mk.
  velkaa Roopelle

Lisää (K/e):[RET]

  Ankka Lupu 010356-127L
  Ankkakuja 6 12345 ANKKALINNA
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 10 mk.
  Aku Ankan veljenpoika, sudenpentu

Lisää: (K/e):e

2. Etsi jäsenen tiedot
======================

Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.) :[RET]
```

Korjailua ja muuta varten on mukavaa, että viimeksi löydetyn henkilön tiedot jäävät muistiin. Näin esimerkiksi korjailu tai poisto voidaan tehdä suoraan tälle henkilölle.

2.4.3 Tulosteet

Tulosteita varten voisi tulla menu siitä, mitä tulostetaan. Lisäksi voisi olla kysymys siitä, millä ehdoilla tulostus tehdään (jäsenmaksu maksamatta jne.) sekä tulostusjärjestys. Tulostusjärjestys on tärkeä, sillä esimerkiksi postitusta varten lehdet yms. pitää lajitella postin antamien ohjeiden mukaisesti postinumeroittain nippuihin, jotka sitten menevät tietylle postialueelle. Jäsenrekisteriä varten taas aakkosjärjestetty lista on kätevin.

Tulostuskohdan suunnittelu jätetään lukijalle harjoitustehtäväksi. Tulosteiden ulkonäkö kannattaa kuitenkin suunnitella tarkasti, koska tämä on muille kerholaisille näkyvin osa ohjelmastamme!

Tietokantaohjelmissa tulosteista käytetään myös nimitystä "raportit".

Tehtävä 2.1 Tulosteet

Suunnittele tulostusmenu ja kunkin kohdan alta mahdollisesti saatavat kysymykset sekä tulosteiden ulkonäkö.

2.4.4 Tietojen korjailu

```
4.  Tietojen korjailu
=====================

  Korjailtava henkilö:

  Ankka Lupu 010356-127L
  Ankkakuja 6 12345 ANKKALINNA
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 10 mk.
  Aku Ankan veljenpoika, sudenpentu

Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.)
tai poisto (P) tai korjailu (K) :1

Jäsenen nimi      (Ankka Lupu)  >*aku*[RET]
Tähän täsmää 1 jäsentä:

  Ankka Aku 010245-123U
  Ankkakuja 6 12345 ANKKALINNA
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 50 mk.
  velkaa Roopelle

Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.)
tai poisto (P) tai korjailu (K) :p

Haluatko todellakin poistaa jäsenen Ankka Aku (k/E):e

Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.)
tai poisto (P) tai korjailu (K) :k

Jäsenen nimi      (Ankka Aku)   >[RET]
Sotu              (010245-123U) >[RET]
Katuosoite        (Ankkakuja 6) >Ankkakuja 13[RET]
Postinumero       (12345)       >[RET]
Postiosoite       (ANKKALINNA)  >[RET]
Kotipuhelin       (12-12324)    >12-12325[RET]
Työpuhelin        ()            >12-33333[RET]
Autopuhelin       ()            >[RET]
Liittymisvuosi    (91)          >[RET]
Jäsenmaksu mk     (50)          >[RET]
Lisätietoja       (velkaa Roopelle) >[RET]

  Ankka Aku 010245-123U
  Ankkakuja 13 12345 ANKKALINNA
  k: 12-12325 t: 12-33333 a:
  Liittynyt -91.  Jäsenmaksu 50 mk.
  velkaa Roopelle

Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.)
tai poisto (P) tai korjailu (K) :[RET]
```

Huomattakoon, että edellä LUPU ANKKA jäi vielä asumaan osoitteeseen Ankkakuja 6. Tietysti ohjelma voitaisiin tehdä myös siten, että joidenkin tiettyjen arvojen muuttaminen muuttaisi haluttaessa myös muita (esim. kirjoitettu sama osoite).

2.4.5 Lopetus

Ohjelman lopetuksessa tulee huolehtia siitä, että ohjelman aikana mahdollisesti rekisteriin tehdyt muutokset tulevat tallennetuiksi. Tämä voidaan tehdä automaattisesti tai tallennus voidaan varmistaa käyttäjältä. Automaattisen tallennuksen tapauksessa alkuperäinen tiedosto on ehkä syytä tallentaa eri nimelle.

```
      ##################################
      #   J Ä S E N R E K I S T E R I  #
      #     versio 9.95                #
      #     Hannu Hanhi                #
      ##################################


Tiedot tallennettu tiedostoon KELMIT.DAT
Vanhat tiedot tiedostossa     KELMIT.BAK

  KIITOS KÄYTÖSTÄ JA TERVETULOA UUDELLEEN
```

2.5 Käyttöohjeen testaus

Kirjoitettu käyttöohje kannattaa antaa jollekin luotettavalle henkilölle "apinatestiin". Onko toimintoja riittävästi? Keksitäänkö erikoistapauksia, joista saattaisi seurata hankaluuksia?

Miksi jäsenen tietojen etsintä ja korjailu ovat eri paikoissa? Turvallisuussyistä!

2.5.1 Testaus

Annetaanpa käyttöohje vaikkapa kerhon sihteerin testattavaksi. Hänelle on juuri vuodenvaihteessa tullut iso kasa pankkisiirtokuitteja uuden vuoden jäsenmaksujen maksamisesta. Niinpä seuraakin ehkä keskustelu:

Sihteeri: Miten jäsenmaksut korjataan. Ensin kaikki viimevuotiset pitää poistaa. Jaa ehkä jos joku on ainaisjäsen, niin hänen merkintäänsä ei poisteta. No muilta kuitenkin. Sitten pitäisi nämä kuitit saada naputeltua sinne!

Ohjelmoija: No poisto ensiksi. Valitse 4 eli korjailu. Sitten valitse 1 Jäsenen nimi ja vastaa ensimmäisen jäsenen nimi. Kun se löytyy, niin paina K niin kuin korjailu ja sitten vain RET kunnes olet jäsenmaksun kohdalla ja sitten välilyönti ja RET jäsenmaksuun. Sitten taas 1 Jäsenen nimi jne. Helppoa!

Sihteeri: Tuohan vie AINAKIN vuoden!!! MINÄ KÄYTÄN EDELLEEN VANHAA käsikirjanpitoani. Tästä vaan kumilla vanha pois ja uusi tieto tilalle...

Ohjelmoija: Öh, tuota mutta kun minä...

2.5.2 Korjaus

Siis parasta mennä takaisin miettimään. Onneksi itse ohjelmaa ei ole ehditty kirjoittamaan. Jäsenmaksut?

Miten jäsenmaksukentän käyttö? Ehkä tämä kenttä pitää säilyttää tiedoksi siitä, paljonko henkilön pitäisi maksaa jäsenmaksua. Tarvitsemme siis uuden kentän:

```
Maksettu maksu    (10)          >
```

Ainais-– ja kunniajäsenet hoidetaan siten, että heidän jäsenmaksukseen, joka pitäisi maksaa, annetaan vaikkapa 0 mk.

Lisäksi meille ilmiselvät asiat eivät aina olleetkaan sihteerille selviä. Ohjelmaamme täytynee lisätä myös avustus.

Lisätään kaksi uutta valintaa päävalikkoon:

```
...
Valitse:
   ?  = avustus
   0  = lopetus
   1  = lisää uusi jäsen
   2  = etsi jäsenen tiedot
   3  = tulosteet
   4  = tietojen korjailu
   5  = päivitä jäsenmaksuja
   :5

5. Päivitä jäsenmaksuja
=======================

Valitse:
   ?  = avustus
   0  = takaisin päävalintaan
   1  = poista kaikki edellisen vuoden maksut
   2  = kysy maksettu maksu nimen mukaan
   :1

Poistetaan kaikki edellisen vuoden jäsenmaksut!
Poistetaan siis kaikki maksetut jäsenmaksut (K/e):K
Odota hetki...  Jäsenmaksut poistettu!

Valitse:
   ?  = avustus
   0  = takaisin päävalintaan
   1  = poista kaikki edellisen vuoden maksut
   2  = kysy maksettu maksu nimen mukaan
   :2

Muutetaan maksettuja jäsenmaksuja!
==================================

Jos haluat kaikki jäsenet, anna *
Kysely loppuu, mikäli annat maksuksi  q
Jäsenen nimi      ()            >*[RET]

  Ankka Aku 010245-123U
  Ankkakuja 13 12345 ANKKALINNA
  k: 12-12325 t: 12-33333 a:
  Liittynyt -91.  Jäsenmaksu 50 mk.  Maksettu mk.
  velkaa Roopelle
Maksettu maksu mk ()            >[RET]

  Ankka Lupu 010356-127L
  c/o Aku Ankka (Ankkakuja 13 12345 ANKKALINNA)
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 10 mk.  Maksettu mk.
  Aku Ankan veljenpoika, sudenpentu
Maksettu maksu    ()            >10[RET]
... Jatkuu näin kunnes kaikki käyty läpi ...
... Tai vastataan ...
Maksettu maksu mk ()            >q[RET]

Jäsenen nimi      ()            >*aku*[RET]
  Ankka Aku 010245-123U
  Ankkakuja 13 12345 ANKKALINNA
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 50 mk.  Maksettu mk.
  velkaa Roopelle
Maksettu maksu    ()            >40[RET]
Pitäisi maksaa 50 mk ja on maksanut 40 mk.
Onko oikein (K/e):e
Maksettu maksu mk ()            >50[RET]

Jäsenen nimi      ()            >[RET]
```

```
Valitse:
   ?  = avustus
   0  = takaisin päävalintaan
   1  = poista kaikki edellisen vuoden maksut
   2  = kysy maksettu maksu nimen mukaan
   :0
...
```

2.5.3 Muita korjauksia

Käyttäjällä saattaa tulla vaikeuksia myös kesken jonkin kysymyksen. Tällöin voisi olla hyvä, että käyttäjä voi painaa ?–merkkiä ja saada avustusta siitä, mitä tähän kohti pitäisi syöttää (sisältöriippuva avustus, *context sensitive help*).

Jäsenmaksun päivittämisessä jokerimerkin käyttö tuntui varsin kätevältä tavalta saada joko yksi tai useampi henkilö (tai vaikkapa kaikki) päivitettäväksi.

Samaa ajatusta voitaisiin täydentää myös muuhun päivittämiseen. Muutetaankin korjailussa esiintynyt kysymys

```
Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.)
tai poisto (P) tai korjailu (K) :
```

muotoon

```
Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.),
poisto (P), korjailu (K), seuraava (+), edellinen (-):
```

silloin, kun hakuehtoon täsmääviä on löytynyt useita. Tekstit `seuraava` ja `edellinen` voidaan varmaankin jättää pois, jos seuraavaa tai edellistä ei ole.

Kenttälistaan hakuehdossa voitaisiin lisätä lisäkohta, jossa kaikille kentille voidaan antaa ehto (ja/tai):

```
  1 = nimi
...
  9 = liittymisvuosi
  A = jäsenmaksu mk
  B = maksettu maksu mk
  C = lisätietoja
  & = JA ehto kaikille kentille
  | = TAI ehto kaikille kentille

Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.):&

Kirjoita niihin kenttiin ehto, joiden mukaan haluat etsiä.
== tarkoittaa, että kentän TÄYTYY olla tyhjä.
Jäsenen nimi      ()            >*ankka*[RET]
Sotu              ()            >[RET]
Katuosoite        ()            >[RET]
Postinumero       ()            >[RET]
Postiosoite       ()            >[RET]
Kotipuhelin       ()            >[RET]
Työpuhelin        ()            >[RET]
Autopuhelin       ()            >[RET]
Liittymisvuosi    ()            >[RET]
Jäsenmaksu mk     ()            >[RET]
Maksettu maksu mk ()            >==[RET]
Lisätietoja       ()            >[RET]
```

```
Tähän täsmää 1 jäsentä:

  Ankka Tupu 010356-125J
  Ankkakuja 6 12345 ANKKALINNA
  k: 12-12324 t: a:
  Liittynyt -91.  Jäsenmaksu 10 mk. Maksettu mk.
  Aku Ankan veljenpoika
...
```

Edellä siis etsittiin kaikkia niitä Ankkoja, joilla maksettu maksu on tyhjä. Näin sihteeri voisi aina tutkia kenellä maksut on maksamatta (tässä tapauksessa erityisesti Ankoista). Hakuehtoihin voitaisiin vielä liittää epäyhtälöt:

```
<    <=    >   >=    ==   !=
```

Siis hakuehto voisi olla esimerkiksi

```
Jäsenen nimi      ()            >!=*ankka*[RET]
Sotu              ()            >[RET]
...
Jäsenmaksu mk     ()            ><30[RET]
Maksettu maksu mk ()            >==[RET]
Lisätietoja       ()            >[RET]
```

Eli etsitään niitä jäseniä, joiden nimi EI OLE `*ankka*` ja joiden jäsenmaksu on alle `30` sekä maksettu maksu on tyhjä.

Samalla tietojen etsimisessä kysymys

```
Lisää (K/e):[RET]
```

voitaisiin muuttaa selaukseksi:

```
Valitse kenttä jonka mukaan etsitään (?=kenttälista uudel.),
seuraava (+), edellinen (-):
```

Kerhon nimi saattaa olla varsin pitkä. Sen antaminen aina ohjelman käynnistämisen yhteydessä voi olla työlästä. Siksi käynnistämisessä kannattaakin antaa vain lyhenne, jolla tiedosto on tallennettu. Varsinainen nimi täytyy tallentaa jonnekin muualle. Minne?

Nimi voitaisiin tallentaa vaikkapa jäsenrekisteritiedoston ensimmäiselle riville:

kelmit.dat - kerhon nimikin talteen

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
; sukunimi etunimi |sotu       |katuosoite  |postinumero|postiosoite|kotipuhelin|työpuhelin|
Ankka Aku          |010245-123U|Ankkakuja 6 |12345      |ANKKALINNA |12-12324   |          |
Susi Sepe          |020347-123T|            |12555      |Takametsä  |           |          |
Ponteva Veli       |030455-3333|            |12555      |Takametsä  |           |          |
```

Kerhon nimi kysytään käyttäjältä uutta tiedostoa luotaessa.

Jos halutaan vielä suurempaa yhteensopivuutta valmiiden tietokantaohjelmien kanssa, voidaan kerhon nimi tallentaa erilliseen tiedostoon muiden kerhoon liittyvien lisätieto-

jen kanssa. Esimerkiksi jos jäsenet on tiedostossa kelmit.dat, voisi lisätiedot olla tiedostossa kelmit.opt. Tällöin myös kommentit (;) kannattaa jättää pois tiedostosta ja tiedoston ensimmäinen rivi on kenttien nimiä kuvaava rivi.

kelmit.dat - yhteensopivuus muihin ohjlemiin

```
nimi               |sotu       |katuosoite  |postinumero|postiosoite|kotipuhelin|työpuhelin|
Ankka Aku          |010245-123U|Ankkakuja 6 |12345      |ANKKALINNA |12-12324   |          |
Susi Sepe          |020347-123T|            |12555      |Takametsä  |           |          |
Ponteva Veli       |030455-3333|            |12555      |Takametsä  |           |          |
```

kelmit.opt - yleiset tiedot tänne

```
[Tiedot]
nimi=Kelmien kerho ry
maxjaseniä=100
```

2.5.4 Uusi testaus

Koska käyttöohjeemme on tietenkin kirjoitettu tekstinkäsittelyohjelmalla (tai ohjelmaeditorilla), on muutokset helppo tehdä. Menemme uudelleen sihteerin luokse uudistetun käyttöohjeen kanssa ja pyydämme jos hän armeliaasti vielä kerran suostuisi katsomaan sitä.

Paranneltavaa saattaa löytyä lisää nyt kun kiinnostus ehkä herää. Korjaamme selvät kohdat mutta joistakin asioista täytyy neuvotella ja ainakin tinkiä ettei niitä ehkä toteuteta ohjelman ensimmäiseen versioon.

Yksi selvä ongelma ilmenee. Entä jos lisäyksessä todella on kaksi samannimistä jäsentä? Homma täytynee korjata seuraavasti:

```
Jäsenen nimi      ()            >Ankka Aku[RET]
Rekisterissä on jo jäsen Aku Ankka!  Mikäli haluat muuttaa
hänen tietojaan, valitse tietojen korjailu päävalikosta!
Lisätäänkö sama nimi (l) vai kysytäänkö toinen nimi (T):T
Jäsenen nimi      ()            >ANKKA TUPU[RET]
```

Jatkamme testauskierrosta kunnes potentiaaliset käyttäjät on saatu tyytyväiseksi.

2.6 Tarvittavien algoritmien hahmottaminen

Nyt olemme selvillä ohjelman toiminnasta. Edellisestä käyttöohjeesta voimme etsiä mitä työkaluja (aliohjelmia) tarvitsemme ohjelman toteutuksessa. Ainakin seuraavat tulevat helposti mieleen:

2.6.1 Ylemmän tason aliohjelmat

1. tiedoston lukeminen
2. tiedoston tallentaminen
3. henkilön tietojen kysyminen päätteeltä
4. tiedoston lajittelu haluttuun järjestykseen
5. tiedon etsiminen tiedostosta tietyllä hakuehdolla
6. uuden henkilön lisääminen tiedostoon
7. henkilön poistaminen tiedostosta

2.6.2 Alemman tason aliohjelmat

Mikäli tutkimme yo. palasia tarkemmin, tarvitsemme ehkä seuraavia pienempiä ohjelman palasia (apualiohjelmia):

1. yhden merkin vastauksen lukeminen mahdollisen oletusarvon kanssa
2. merkkijonon lukeminen päätteeltä siten, että sille voidaan jättää oletusarvo
3. pitkän merkkijonon pilkkominen osamerkkijonoihin annetun merkin kohdalta
4. loppuvälilyöntien poistaminen merkkijonosta
5. isojen ja pienien kirjainten muuttaminen merkkijonossa esimerkiksi:

```
AKU ANKKA  ->  Aku Ankka
aku ankka  ->  Aku Ankka
aKU ANkKa  ->  AKU ANKKA
```

6. sotun oikeellisuuden tarkistus
7. ovatko merkkijonot "`*aku*`" ja "`AKU ANKKA`" samoja?

2.6.3 Ohjelman yhteiset osat

Kannattaa myös etsiä onko ohjelmassa samanlaisina toistuvia osia. Edellä meillä selvästikin tiedon haku on samanlainen sekä etsimis- että korjailukohdassa. Samoin henkilön tietojen luku on samanlainen sekä lisäämisessä että korjailussa. Yhden henkilön tietojen tulostaminen näytölle esiintyy useammassa kohdassa.

Mikäli löytyy likipitäen samanlaisuuksia, kannattaa harkita voidaanko ne käyttämisen yksinkertaistamiseksi ja/tai ohjelmoinnin helpottamiseksi muuttaa samanlaisiksi. Mikäli voidaan, korjataan äkkiä käyttöohjetta tältä osin.

2.7 Ikkunoinnit ja muut hienostelut

Ohjelma voitaisiin suunnitella myös nykyaikaisen ikkunoidusti toimivan käyttöliittymän mukaiseksi. Kuten edellä todettiin, tämä on kuitenkin ohjelmoinnin oppimisen tässä vaiheessa liian työlästä ja tähän paneudutaan vasta myöhemmillä kursseilla.

Tässäkin ohjelmassa korjailua voitaisiin parantaa siten, että meillä olisi käytössä aliohjelma, jolle korjailtava merkkijono vietäisiin parametrinä. Palautuksena tulisi korjattu merkkijono ja korjailun aikana toimisivat nuolinäppäimet yms. hienoudet. Tosin suurtakaan muutosta ohjelmaan ei tarvitse tehdä mikäli em. aliohjelman tilalla käytämme aluksi vain merkkijonon lukemiseen kykenevää aliohjelmaa. Myöhemmin tätä voitaisiin parantaa.

Menut voisivat olla nykytyyliin alasvetovalikoita, mutta aluksi meille riittää vallan hyvin käyttöohjeessa esitetyn kaltainen "näyttö tyhjäksi ja uusi menu ruutuun" –tyyli. Myös hiirtä voitaisiin käyttää, mutta jälleen ohjelmointityö kasvaisi vastaavasti.

Hakuehdot voisivat olla monipuolisempia ja niille voitaisiin yrittää keksiä jokin hienompi menetelmä. Esimerkiksi tarvitsisi hakea seuraavilla ehdoilla

```
nimi on "*ankka*" tai jäsenmaksu on "<50"
postiosoite "ankka*" ja lisätiedoissa "*sudenpentu*"
```

Tätä varten hakemiseen voitaisiin kehitellä vaikkapa seuraavanlainen kieli:

```
Hakuehto >(nimi=*ankka*) || (jmaksu<50)[RET]
...
Hakuehto >(postiosoite=ankka*) && (lisätiedot=*sudenpentu*)[RET]
```

Saattaa tulla myös tarve lisätä uusia kenttiä henkilön tietoihin. Tämä on hallittavissa huolellisella ohjelman suunnittelulla, jossa käytettyjen kenttien määrä ja nimet esiintyisivät vain yhdessä paikassa ohjelmaa.

Kun näitä haluttuja lisäominaisuuksia silmäillään, ei ole ihme että on kehitetty tietokantaohjelmia; eli halutut ominaisuudet tarvitaan lähes jokaisessa vastaavassa sovelluksessa. Hieman muuttamalla oman ohjelmamme toimintaa, voisimme saada siitäkin yleiskäyttöisen tietokantaohjelman, mutta jätettäköön tämäkin lukijalle harjoitustehtäväksi.

Etsimisissä voisi olla oletuksena lisätä * kummallekin puolelle etsittävää jonoa, jolloin kun nimeen vastataan

```
aku
```

täydennetään tämä muotoon

```
*AKU*
```

ja näin löydetään Ankka Aku.

2.8 Koodaus ohjelmointikielelle

Seuraava vaihe olisi suunnitelman koodaaminen valitulle ohjelmointikielelle. Voisimme kirjoittaa aluksi löytämiämme alimman tason aliohjelmia (*BOTTOM-UP*–suunnittelu) ja testata ne toimiviksi. Voisimme myös kirjoittaa pääohjelman ja tyhjiä aliohjelmia testataksemme ohjelman rungon (*TOP-DOWN*). Menujen alavalinnat voitaisiin laittaa vain sanomaan:

```
TOIMINTAA EI OLE VIELÄ TOTEUTETTU!
```

Emme kuitenkaan osaa vielä riittävästi ohjelmointikieltä, jotta voisimme aloittaa koodauksen. Huomattakoon, ettei yllä olevassa suunnitelmassa ole missään kohti vedottu käytettävään ohjelmointikieleen. Palaamme myöhemmin takaisin ohjelman osien koodaamiseen.

2.9 Varautuminen tulevaan, eli relaatiotietomalli

Vaikka sihteerimme ei juuri nyt huomannutkaan, saattaa hän tulevaisuudessa esimerkiksi kysyä miten rekisterillä pidettäisiin yllä tietoja jäsenten harrastuksista. Mietitäänpä?

Ensin miten harrastukset muuttaisivat tiedostomuotoamme?

2.9.1 Kaikki samassa tietueessa

Eräs mahdollisuus olisi lisätä kunkin rivin loppuun jollakin erotinmerkillä harrastukset:

kelmit.dat - harrasteet samalle riville

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
; sukunimi etunimi |sotu       |…|harrastukset
Ankka Aku          |010245-123U|…|kalastus,laiskottelu,työn pakoilu
Susi Sepe          |020347-123T|…|possujen jahtaaminen,kelmien kerho
Ponteva Veli       |030455-3333|…|susiansojen rakentaminen
```

Ratkaisu toimisi tietyissä erityistapauksissa. Ongelmia tulisi esimerkiksi jos pitäisi kuhunkin harrastukseen liittää esimerkiksi harrastuksen aloitusvuosi, viikoittain harrastukseen käytetty tuntimäärä jne.

2.9.2 Erimalliset tietueet

Edellinen ongelma ratkeaisi esimerkiksi laittamalla henkilön tietojen rivin perään jollakin tavalla eroavia rivejä, joilla harrastuksen on lueteltu:

kelmit.dat - harrasteet omalle riville

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
; sukunimi etunimi |sotu       |katuosoite  |postinumero|postiosoite|kotipuhelin|työpuhelin|
Ankka Aku          |010245-123U|Ankkakuja 6 |12345      |ANKKALINNA |12-12324   |          |
- kalastus                | 1955 | 20
- laiskottelu             | 1950 | 20
- työn pakoilu            | 1952 | 40
Susi Sepe          |020347-123T|            |12555      |Takametsä  |           |          |
- possujen jahtaaminen    | 1954 | 20
- kelmien kerho           | 1962 |  2
Ponteva Veli       |030455-3333|            |12555      |Takametsä  |           |          |
- susiansojen rakentaminen | 1956 | 15
```

Ratkaisu olisi aivan hyvä ja tämän ratkaisun valitsemiseksi meidän ei tarvitsisi tehdä mitään muutoksia tiedostomuotoomme vielä tässä vaiheessa.

Huono puoli on kuitenkin se, että tämän muotoisen tiedoston siirrettävyys muihin järjestelmiin on varsin huono.

2.9.3 Relaatiomalli

Suurin osa tämän hetken valmiista järjestelmistä käyttää relaatiotietokantamallia. Tämä tarkoittaa sitä, että koko tietokanta koostuu pienistä tauluista, jossa kukin rivi (=tietue) on samaa muotoa. Eri taulujen välillä tiedot yhdistetään yksikäsitteisten avainkenttien avulla. Meidän esimerkissämme kelmit.dat olisi yksi tällaisen taulu ja sosiaaliturvatunnus kelpaisi yhdistäväksi avaimeksi (relaatioksi).

Kuitenkin sosiaaliturvatunnus on varsin pitkä kirjoittaa ja välttämättä sitä ei saada kaikilta. Jos tällainen pelko on olemassa, täytyy avain luoda itse. Itse ohjelman käyttäjän ei tarvitse tietää mitään tästä uudesta muutoksesta, vaan ohjelma voi itse generoida avaimet ja käyttää niitä sisäisesti.

Valitaan vaikkapa juoksevasti generoituva numero. Jos jäseniä poistetaan jää ko. jäsenen numero vapaaksi eikä sitä yritetäkään enää käyttää. Uuden jäsenen numero olisi sitten aina suurin jäsenen numero +1.

kelmit.dat - relaatiokannan päätaulu

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
;id|sukunimi etunimi   |sotu       |katuosoite  |postinumero|postiosoite|kotipuhelin|työpuhelin|
1  |Ankka Aku          |010245-123U|Ankkakuja 6 |12345      |ANKKALINNA |12-12324   |          |
2  |Susi Sepe          |020347-123T|            |12555      |Takametsä  |           |          |
4  |Ponteva Veli       |030455-3333|            |12555      |Takametsä  |           |          |
```

Harrastukset kirjoitetaan toiseen tiedostoon, jossa tunnusnumerolla ilmaistaan kuka harrastaa mitäkin harrastusta.

harrastu.dat - harrasteet relaation avulla

```
id|harrastus                |aloit |viikossa
1 |kalastus                 | 1955 | 20
1 |laiskottelu              | 1950 | 20
1 |työn pakoilu             | 1952 | 40
2 |possujen jahtaaminen     | 1954 | 20
2 |kelmien kerho            | 1962 |  2
4 |susiansojen rakentaminen | 1956 | 15
```

Nyt esimerkiksi kysymykseen "Mitä Sepe Susi harrastaa" saataisiin vastus etsimällä ensin Sepe Suden tunnus (2) tiedostosta kelmit.dat. Sitten etsittäisiin ja tulostettaisiin kaikki rivit joissa tunnus on 2 tiedostosta harrastu.dat.

Myös vastaus kysymykseen "Ketkä harrastavat laiskottelua" löytyisi suhteellisen helposti.

Tämä ratkaisu vaatii muutoksen tiedostomuotoomme jo suunnitelman tässä vaiheessa, mutta toisaalta mikäli ratkaisu valitaan, voidaan sen ansiosta lisätä jatkossa vastaavia "monimutkaisia" kenttiä rajattomasti tekemällä kullekin oma "taulu".

Valitsemmekin siis tämän ratkaisun, eli annamme kullekin jäsenelle tunnusnumeron heti alusta pitäen. Itse ohjelman käyttösuunnitelmaan ei tässä vaiheessa tarvita muutoksia.

Tehtävä 2.2 Ketkä harrastavat?

Kirjoita algoritmi joka relaatiomallin tapauksessa vastaa kysymykseen "Ketkä harrastavat harrastusta X".

2.9.4 XML-muotoinen tiedosto

Nykyisin on kovasti muotia että jokainen ohjelma osaa lukea ja kirjoittaa *XML*-muotoista tiedostoa (*Extensible Markup Language*). Meidän ohjelmamme tiedosto voisi olla vaikka seuraavan näköinen *XML*-muotoisena:

kelmit.xml – kerho XML-muodossa

```
<?xml version="1.0"?>
<kerho>
<kerhonnimi>Kelmien kerho ry</kerhonnimi>
<maxjasenia>13</maxjasenia>
<jasen>
  <id>1</id>
  <nimi>Ankka Aku</nimi>
  <hetu>010245-123U</hetu>
  <katuosoite>Ankkakuja 6</katuosoite>
  <postinumero>12345</postinumero>
  <postiosoite>ANKKALINNA</postiosoite>
  <kotipuhelin>12-12324</kotipuhelin>
```

```
  <harrastukset>
     <harrastus>kalastus</harrastus>
     <aloit>1955</aloit>
     <viikossa>20</viikossa>
  </harrastukset>
  <harrastukset>
     <harrastus>laiskottelu</harrastus>
     <aloit>1950</aloit>
     <viikossa>20</viikossa>
  </harrastukset>
  <harrastukset>
     <harrastus>tyon pakoilu</harrastus>
     <aloit>1952</aloit>
     <viikossa>40</viikossa>
  </harrastukset>
</jasen>
<jasen>
  <id>2</id>
  <nimi>Susi Sepe</nimi>
  <hetu>020347-123T</hetu>
  <postinumero>12555</postinumero>
  <postiosoite>Takametsa</postiosoite>
  <harrastukset>
     <harrastus>possujen jahtaaminen</harrastus>
     <aloit>1954</aloit>
     <viikossa>20</viikossa>
  </harrastukset>
  <harrastukset>
     <harrastus>kelmien kerho</harrastus>
     <aloit>1962</aloit>
     <viikossa>2</viikossa>
  </harrastukset>
</jasen>
<jasen>
  <id>4</id>
  <nimi>Ponteva Veli</nimi>
  <hetu>030455-3333</hetu>
  <postinumero>12555</postinumero>
  <postiosoite>Takametsa</postiosoite>
  <harrastukset>
     <harrastus>susiansojen rakentaminen</harrastus>
     <aloit>1956</aloit>
     <viikossa>15</viikossa>
  </harrastukset>
</jasen>
</kerho>
```

Kuten edeltä nähdään, on *XML* varsin tuhlaileva tallennusmuoto. Sen käyttöä puoltaa lähinnä sen standardinmukaisuus. Tuon tiedoston voi lukea tulevaisuudessa vaikka millä ohjelmalla. Haittapuolena on työläämpi lukeminen omassa ohjelmassa. Tosin jos on tarkoitus selvitä vain ylläkuvatun mukaisesta tiedostosta, ei koodaus ole kovin paljon monimutkaisempaa kuin muidenkaan tiedostomuotojen kanssa. Lisäksi esim. Java-kieleen löytyy useita *XML*-jäsentimiä valmiiksi käytettävinä luokkina.

Tehtävä 2.3 Mikä on tilaa säästävin tallennusmuoto

Laske mikä edellä esitetyistä vaihtoehtoisista tiedostomuodoista on tilaa säästävin kun rivinvaihtomerkin lasketaan vievän yhden merkin verran tilaa ja välilyönnit "unohdetaan". Laske karkeasti "merkkejä/jäsen".

3. Yksinkertainen tulkkiohjelma

Anna aihe aivan toinen
jos siitäkin koodin voinen?
Olkoon ukko ulkomailla
sanoja sen taarviis saada.

Mielipä sen jo tekevi
apetta ajattelevi.
Polo kuinka tuon sanoisi
Sanakirjan saa avuksi

.

3.1 Tehtävän tarkennus

Olkoon meillä tehtävänä tehdä satunnaisen matkaajan käyttöön soveltuva tulkkiohjelma, joka kääntää sana kerrallaan. Mitään valmiita lauseita ja taivutusmuotoja hallitsevaa ohjelmaa emme edes yritä tehdä. Tavoitteena on siis sanakirjan yksinkertaistettu elektroninen versio.

Ohjelman yksi tavoite täytyisi olla sanaston helppo täydentäminen. Sanasto on jälleen luonnollisinta säilyttää tiedostossa. Mikäli sanasto on normaali tekstitiedosto, on sen täydentäminen tekstinkäsittelyohjelmalla helppoa. Nykyisin JOKAISEN on osattava käyttää jotakin tekstinkäsittelyohjelmaa!

Tehtäväksemme jää siis sanastotiedoston muodon suunnittelu sekä itse ohjelman toimintojen suunnittelu.

3.2 Tietorakenteet ja tiedostot

Sanasto voisi näyttää vaikkapa seuraavalta:

```
Suomi     Ruotsi    Englanti
;-----------------------------------------
minä      jag       i
sinä      du        you
hän       han       he
```

Tässä tuleekin nyt ongelmia. Tarvitaan synonyymejä. Esimerkiksi "*hän*" täytyisi voida sanoa "*he*" tai "*she*" suvun mukaan. Miten esittäisimme synonyymit? Tarvitaan ehkä myös kommentteja! Ratkaisu voisi olla seuraava:

```
Suomi         | Ruotsi          | Englanti
;--------------------------------------
minä          | jag             | i
sinä          | du              | you (yks.)
hän           | han (mask.)     | he  (mask.)
hän           | hon (fem.)      | she (fem.)
me            | vi              | we
te            | ni              | you (mon.)
he            | dom             | they
hernekeitto   | ärtsoppa        | pea soup
sukellusvene  | ubåt            | submarine
suklaa        | chokolade       | chocolate
sukka         | socka (lyhyt)   | sock (lyhyt)
sukka         | strumpa (pitkä)| stocking (pitkä)
```

Itse ohjelman on sitten huolehdittava siitä, että synonyymien tapauksessa myös synonyymit tulostetaan.

Ohjelma lukee käynnistyessään koko tiedoston. Tiedoston 1. riviltä selviävät käytettyjen kielten nimet. Nimessä iso kirjain tarkoittaa kirjainta, jolla kielen nimi voidaan lyhentää.

Tehtävä 3.1 Kielen lisääminen

Lisää sanastoon Savon kieli!

3.3 Käyttöohje

Ohjelman suunnittelu on tässäkin tapauksessa parasta tehdä käyttöohjeen avulla. Suunnittelemme aluksi kuitenkin ohjelmalta halutut toiminnot.

Satunnaisen matkaajan tarpeet ovat varmaankin seuraavat:

- Hän istuu ruokapöydässä Ruotsissa ja lukee menua. Pitäisi saada selville mitä "*ärtsoppa*" tarkoittaa.
- Hän haluaa "*hernekeittoa*". Mitä se on ruotsiksi?
- Hän näkee sukellusveneen Tukholman rannikolla ja haluaa kertoa tästä vieressään seisovalle merisotilaalle.
- Ruotsalainen merisotilas ei hetkauta korvaansa "*UBÅT–UBÅT*" –huudoille. Takana seisoo englantilainen kenraali ja satunnainen matkaaja haluaisi tiedottaa näkemästään myös hänelle.

Suomalaisen ollessa Ruotsissa tarvitaan käännöksiä ruotsi–suomi ja suomi–ruotsi. Siis molemmat kielet tulee voida valita.

Hätätilassa pitää suhteellisen helposti päästä käsiksi myös muunkielisiin sanoihin.

Periaatteessa, mikäli kielet on jo valittu, riittäisi käyttö tyyliin

```
Sana >ärtsoppa[RET]
ruotsi: ärtsoppa = suomi: hernekeitto
```

Entäpä mikäli kielinä olisi suomi ja englanti:

```
Sana >he[RET]
suomi   : he         = englanti: they
englanti: he (mask.) = suomi   : hän
```

Tämä voisi olla aivan hyvä ratkaisu. Miten kielen vaihto kävisi kätevästi (nopeasti)?

```
Sana >sukellusvene/s-e[RET]
suomi: sukellusvene = englanti: submarine
```

Voisi olla tarpeen myös saada käännös samanaikaisesti usealla eri kielellä:

```
Sana >sukellusvene/s-re[RET]
suomi: sukellusvene = ruotsi  : ubåt
                    = englanti: submarine
```

Sovimme siis, että sanan perään kirjoitetaan kauttaviivan jälkeen mistä kielestä käännetään ja mihin. Mikäli kumpiakin kieliä on vain yksi, käännös voi olla kumpaankin suuntaan. Mikäli toista kieltä on useita kappaleita, tehdään käännös vain kuten on pyydetty. Mikäli toista kieltä ei anneta lainkaan, tarkoitetaan kaikkia mahdollisia kieliä.

Kerran annetut käännöskielet säilyvät, kunnes käännöskielet annetaan uudelleen.

Siis ohjelman toiminta voisi olla vaikkapa seuraavanlainen:

```
Terve! Olen Satunnaiselle matkaajalle suunniteltu Tulkki!
Sanastoni on tiedostossa SANASTO.DAT.  Mikäli haluat lisätä
sanastoa, käytä tekstinkäsittelyohjelmaa sanaston
päivittämiseksi.

Avustusta saat vastaamalla ? kun sinulta kysytään sanaa.
Tulkattavina kielinä on nyt suomi-ruotsi.
Sana>?[RET]
------------------------------------------------------------
Avustus:

Kirjoita sanakysymyksen perään sana jonka haluat kääntää.
Mikäli painat pelkän [RET], ohjelman toiminta loppuu.

Mikäli haluat vaihtaa kieliä, kirjoita sanan perään
kauttaviiva ja kielten nimien lyhennekirjaimet, joita
haluat kääntää.  Mikäli haluat käännöksen samalla
usealle eri kielelle/kieleltä, kirjoita kaikki haluamasi
kielet.  Mikäli haluat käännöksen kaikille mahdollisille
kielille, kirjoita vain lähtökieli.

Tunnen kielet Suomi Ruotsi Englanti.
Esimerkki:
Sana>kissa/s-e
suomi: kissa = englanti: cat
Sana>car
englanti: car = suomi: auto
Sanassa voi käyttää myös jokerimerkkejä * ja ?
Esimerkki:
Sana>suk*
------------------------------------------------------------
```

```
Sana>hän/s-e[RET]
suomi: hän = englanti: he  (mask.)
           = englanti: she (fem.)
Sana>he[RET]
suomi   : he          = englanti: they
englanti: he (mask.) = suomi   : hän

Sana>hän/s-[RET]
suomi: hän = englanti: he  (mask.)
           = ruotsi  : han (mask.)
           = englanti: she (fem.)
           = ruotsi  : hon (fem.)

Sana>suk*/s-r[RET]
suomi: sukellusvene = ruotsi : ubåt
       sukka        = ruotsi : strumpa (pitkä)
       sukka        = ruotsi : socka (lyhyt)
       suklaa       = ruotsi : chokolade

Sana>käyrätorviorkesteri[RET]
Ei löydy sanastosta!
Sana>[RET]
Kiitos käytöstä!
```

Satunnainen matkaaja voi olla huono/hidas konekirjoittaja. Siksi valitsimme hänelle lisäksi mahdollisuuden antaa sanoja jokerimerkkien avulla. Näin hänen ei välttämättä tarvitse osata kirjoittaa koko sanaa vaan hän voi lyhentää sen haluamastaan kohdasta. Tätä sanotaan käyttäjäystävällisyydeksi.

Mikäli kieliä ei anneta lainkaan, sanaa etsittäisiin kaikista mahdollisista kielistä. Tällaisen option tarve olisi silloin, kun ei ole harmaintakaan aavistusta siitä, millä kielellä sana on kirjoitettu.

```
Sana >h*/-[RET]
suomi   : hän         = ruotsi  : han (mask.)
                      = englanti: he (mask.)
ruotsi  : han (mask.) = suomi   : hän
                      = englanti: he (mask.)
englanti: he (mask.)  = suomi   : hän
                      = ruotsi  : han (mask.)
suomi   : hän         = ruotsi  : hon (fem.)
                      = englanti: she (fem.)
ruotsi  : hon (fem.)  = suomi   : hän
                      = englanti: she (fem.)
suomi   : he          = ruotsi  : dom
                      = englanti: they
suomi   : hernekeitto = ruotsi  : ärtsoppa
                      = englanti: pea soup
Sana >h*/-s[RET]
ruotsi  : han (mask.) = suomi   : hän
englanti: he (mask.)  = suomi   : hän
ruotsi  : hon (fem.)  = suomi   : hän
```

3.4 Algoritmien hahmottaminen

3.4.1 Ylemmän tason aliohjelmat

Voimme jälleen havaita seuraavat suuremmat ohjelman kokonaisuudet:

1. tiedoston lukeminen
2. sanan lukeminen ja kielten erottaminen
3. sanan etsiminen sanastosta sekä synonyymien tarkistus

Miten sanaa etsitään sanastosta? Mikäli sanasto ei ole järjestetty (voidaan myöhemmin järjestää mikäli sanasto kasvaa) riittää varmaankin raaka peräkkäishaku.

Miten löydetään sanaa vastaavat muunkieliset tulkinnat? Ne ovat löytyneen sanan kanssa samalla rivillä.

Käännöksenä on esim. "`s-e`" (suomi–englanti) ja annetaan sana he. Mitä tehdään? Etsitään suomi –sarakkeesta sanan "*he*" –rivi ja tulostetaan rivin englanti –sarakkeessa oleva sana. Etsitään englanti –sarakkeesta "*he*" –rivi ja tulostetaan rivin suomi –sarakkeessa oleva sana.

3.4.2 Alemman tason aliohjelmat

Yllättäen alemman tason aliohjelmissa tarvitaan samoja ominaisuuksia kuin jäsenrekisteriäkin tehtäessä:

1. merkkijonon lukeminen päätteeltä siten, että sille voidaan jättää oletusarvo (laitetaan oletusarvoksi tyhjä)
2. pitkän merkkijonon pilkkominen osamerkkijonoihin annetun merkin kohdalta
3. loppuvälilyöntien poistaminen merkkijonosta
4. isojen ja pienien kirjainten muuttaminen merkkijonossa. Esimerkiksi:

```
AKU ANKKA     ->  Aku Ankka
aku ankka     ->  Aku Ankka
aKU ANkKa     ->  AKU ANKKA
```

5. ovatko merkkijonot "`*aku*`" ja "`AKU ANKKA`" samoja?

Tehtävä 3.2 Aliohjelmien käyttö

1. Mikä edellisistä sopii sanan perässä olevan kommentin poistoon (suluissa oleva sana)?
2. Voidaanko jotain edellistä soveltaa sanan ja kielten erottamiseen?
3. Entä kun käännös (esim. s–e) on löytynyt, niin voidaanko jotain edellä olevista soveltaa mistä– ja mihin– kielten erottamiseen?

Mikäli käännösetsiminen osataan tehdä esim. s–sarakkeesta ja tulos ottaa e–sarakkeesta ohjeella "`s-e`", niin miten sama saataisiin tehtyä päinvastoin? Ohjeella "`e-s`"?. Siis mikäli käännösohjeeksi valittaisiin "`s-e`", niin se ehkä kannattaisi muuttaa muotoon "`se-es`". Näin kahden kielen tapausta ei tarvitsisi käsitellä minään erikoistapauksena. Samalla ajatuksella ohje "–" muutettaisiin muotoon "`sre-sre`" (eli kaikkiin tunnettuihin kieliin). Siis koko kääntäminen muodostuisi siten, että annettua sanaa etsitään ohjeessa vasemmalla annetuista sarakkeista ja mikäli se jostakin niistä löytyy, tulostettaisiin oikealla annetut sarakkeet paitsi se sarake josta sana löytyi!

Tehtävä 3.3 Erikoistapauksia

Miten muutetaan seuraavat käännöskielet:

`-s`
`r-`

Tehtävä 3.4 Algoritmi muuttamiselle

Kirjoita selkeät säännöt (=taulukko), miten käännösohje muutetaan kussakin erikoistapauksessa.

```
syöttömuoto     muunnettu muoto
-----------------------------------
s-e             se-es
e-s             es-se
s-r             sr-rs
s-              s-sre
...
-               sre-sre
...
```

Eräs ongelma on se, miten tulos muotoillaan siistiksi, eli löydetään pisimmät mahdolliset sanat ja kielet jotta vastinsanat saadaan tulostettua siististi allekkain.

3.5 Koodaus ohjelmointikielelle

Koodaus jätetään jälleen oppimisen myöhempään vaiheeseen. Taaskaan suunnittelun alkuvaihe ei sisältänyt (eikä tarvinnut sisältää) mitään tietoa siitä, millä ohjelmointikielellä ohjelma toteutettaisiin.

4. Algoritmin suunnittelu

Kirjettä jos kirjoittelet,
ulkomaille viestittelet,
tokko Ruohtia viskomassa,
turhaan sanoja kiskomassa?

Aloittanet aatoksilla,
kotokielellä pohtimalla,
viestin vääntöö valmistellen,
siistimiseksi sisällön.

Sama kaava koodatessa
kääntäjätä käskiessä
kotokieltä alkuun käytä
vasta sitten ruutuun täytä.

Algoritmit alkuun teeppä
koneen kimppuun vasta meeppä
kun selvillä on tarkka kaava
jopa kääntyy Cee ja Jaava.

Mitä tässä luvussa käsitellään?

- mikä on algoritmi
- vertailu ja lajittelu
- algoritmin kompleksisuus
- alkion etsiminen joukosta

Kun ohjelman suunnittelu on edennyt siihen pisteeseen, että tarvitaan yksityiskohtaisia algoritmeja, meneekin jo monella sormi suuhun.

Vaikeudet johtuvat taas liian hankalasta ajattelutavasta ja siitä, että algoritmi yritetään nähdä osana koko ohjelmaa. Tästä ajattelutavasta on luovuttava ja osattava määritellä tarvittava algoritmi omana kokonaisuutenaan, jota suunniteltaessa sitten unohdetaan kaikki muu.

4.1 Algoritmi

Algoritmi on se joukko toimenpiteitä, joilla annettu tehtävä saadaan suoritettua. Mieti esimerkiksi miten selostat kaverillesi ohjeet juna–asemalta opiskeluboxiisi.

Voit tietysti antaa ohjeet myös muodossa "Tule osoitteeseen Ohjelmoijankuja 17 B 5". Tämäkin on varsin hyvä algoritmi. Kaverin vain oletetaan nyt osaavan enemmän. Kaverin oletetaan osaavan etsiä katuluettelosta kadun paikka ja keksivän itse menetelmän tulla asemalta sinne.

Toisaalta kaverisi saattaa hypätä taksiin ja sanoa kuskille osoitteen. Tämä on hyvä ja helppo algoritmi, mutta ehkä liian kallis opintovelkaiselle opiskelijalle. Mikäli algoritmia tarvitaan useasti, voidaan sitä myöhemmin parantaa tyyliin:

```
- kävele asemalta sinne ja sinne
- hyppää bussiin se ja se
- jne
```

Tarkennettu algoritmi voisi olla myös seuraavanlainen:

```
Valitse seuraavista:
1.  Kello 7-20:
     - kävele kirkkopuistoon
     - nouse bussiin no 3 joka lähtee 15 yli ja 15 vaille
2.  Sinulla on rahaa tai saat kimpan:
     - ota taksi
3.  Ei rahaa tai haluat ulkoilla:
     - kävele
```

Edellä eri kohdat eivät ole toisiaan poissulkevia. Kello voi olla 9 ja rahaakin voi olla, mutta siitä huolimatta halutaan kävellä. Hyvässä algoritmissa ei saa olla tällaisia epätäsmällisyyksiä, vaan ohjelmoijan tulee etukäteen jo päättää mitä missäkin tapauksessa tehdään. Esimerkiksi:

```
1.  Jos haluat ulkoilla, niin
     - kävele.
2.  Muuten jos kello 7-20:
     - kävele kirkkopuistoon
     - nouse bussiin no 3 joka lähtee 15 yli ja 15 vaille
3.  Muuten jos sinulla on rahaa tai saat kimpan:
     - ota taksi
4.  Muuten
     - kävele
```

Tässäkin algoritmissa jää vielä kaverillekin tehtävää: Miten kävellään? Miten astutaan bussiin jne..

No tätä ei kaverille ehkä enää selostetakaan. Lapsille nämä asiat on aikanaan opetettu ja myöhemmin ne kuitataan yhdellä tai kahdella sanalla. Sama pätee ohjelmoinnissakin. Kerran tehtyä ei joka kerran pureksita uudelleen (vrt. aliohjelma)!

Tehtävä 4.1 Kävelyohjeet

Yritä kirjoittaa ohjeet siitä miten kävellään.

Kirjoita kaverillesi kävelyohjeet (missä käännytään, ei miten kävellään) rautatieasemalta asunnollesi.

4.2 Lajittelu

Kerhon jäsenrekisteriä suunniteltaessa tulee jossakin kohtaa vastaan tilanne, jossa nimet tai osoitteet pitää pystyä lajittelemaan jollakin tavalla.

4.2.1 Nimien ja numeroiden vertaus

Jos osaamme lajitella numeroita, niin osaammeko lajitella nimiä? Vastaus on KYLLÄ. Mikä numeroiden lajittelussa on oleellista? Oleellista on tietää onko numero A pienempi kuin numero B. Miten tämä sitten soveltuu nimille? Jos osaamme päättää onko nimi A aakkosissa ennen kuin nimi B, on ongelma ratkaistu.

Verrataanpa erilaisia nimiä:

```
A: Kassinen Katto
B: Ankka Aku
```

B on ensin aakkosissa. Miksi? Koska B:n ensimmäinen kirjain (A) on ennen nimen A ensimmäistä kirjainta (K).

```
A: Kassinen Katto
B: Karhukopla 701107
```

Nytkin B on ensin. Siis miten vertaamme kahta nimeä?

```
Vertaamme nimiä merkki kerrallaan kunnes vastaan tulee eri-
suuret kirjaimet.  Kumpi erisuurista kirjaimista on aakko-
sissa ennen, määrää sen kumpi nimistä on aakkosissa ennen.
```

Siinä meillä on algoritmi joka on varsin selvä. Jos algoritmi haluttaisiin vielä kirjoittaa "lausekieliseen" muotoon, niin se olisi suurin piirtein seuraavanlainen:

```
1. siirry kummankin nimen ensimmäiseen kirjaimeen
2. jos kummankin nimen viimeinen merkki on ohitettu, niin nimet ovat samat
3. jos toisessa nimessä viimeinen merkki on ohitettu, niin se on ennen
   aakkosissa
4. verrataan vuorossa olevia kirjaimia kummastakin nimestä
 - jos samat, niin siirrytään seuraaviin kirjaimiin ja jatketaan kohdasta
   2.
 - jos erisuuret, niin se ensin aakkosissa, jonka kirjain on ensin
```

Tähän vielä pieni "viilaus enemmän strukturoidummaksi", niin meillä olisikin valmis (ali)ohjelma nimien vertaamiseksi.

4.2.2 Algoritmin sanallinen versio on kuvaavampi!

Vaikka esitimmekin algoritmin "lausekielisenä" kohdittain numeroituna, ei koskaan pidä unohtaa sitä ennen ollutta sanallista versiota, joka on selkeämpi kuvaus siitä ideasta, mitä tehdään!

Siis kirjoita aina ensin sanallinen kuvaava kuvaus algoritmista ja vasta sitten sen yksityiskohtainen "lausekielinen" versio!

4.2.3 Numeroiden järjestäminen

Näin ollen on aivan yksi lysti opettelemmeko järjestämään nimiä vai numeroita. Siksi paneudummekin seuraavassa numeroiden järjestämiseen. Kuulostaako vaikealta?

Otapa käteesi korttipakka ja ota sieltä esiin vaikkapa vain kaikki padat. Nyt sinulla on joukko "numeroita" (A=14, K=13, Q=12, J=11), yhteensä 13 kappaletta. Sekoita kortit! Yritä järjestää kortit suuruusjärjestykseen siten, ettet tarvitse pöytätilaa kuin yhden kortin verran, loput kortit pidät kädessäsi.

Millaisen algoritmin saat? Ehkäpä seuraavan (*insertion sort*):

```
Pöydällä on lajiteltujen kasa.  Aluksi tietysti tyhjä.  Ota
kädestäsi seuraava kortti ja laita pöydällä olevaan kasaan
omalle paikalleen.  Jatka kunnes kädessä ei enää kortteja.
```

"Lausekielisenä":

```
1. ota kädessä olevan kasan päällimmäinen kortti
2. sijoita se pöydällä olevaan kasaan paikalleen
3. mikäli kortteja vielä jäljellä, niin jatka kohdasta 1.
```

Algoritmisi voi olla myös seuraava (*selection sort*):

```
Etsitään aina pienin kortti ja laitetaan se pöydälle olevan
kasan päällimmäiseksi.  Jatketaan kunnes kädessä olevat
kortit on loppu.
```

Eli "lausekielisenä":

```
1.  etsi kädessäsi olevista korteista pienin
2.  laita se pöydällä olevan pinon päällimmäiseksi
3.  mikäli vielä kortteja jäljellä, niin jatka kohdasta 1.
```

Tehtävä 4.2 Muita lajittelualgoritmeja

Mitä muita mahdollisia "lajittelumenetelmiä" keksit?

Siinä eräitä ratkaisuja tähän "hirveän vaikeaan" ongelmaan. Ratkaisuissa on tiettyjä huonoja puolia, mutta ratkaisut ovat todella yksinkertaisia ja jokaisen itse keksittävissä.

Tehtävä 4.3 Algoritmin kompleksisuus

Mikäli kahden kortin vertaaminen lasketaan yhdeksi "operaatioksi", niin kuinka monta "operaatiota" joudumme tekemään, jotta pakka on lajiteltu *Selection Sortilla*?

Edellisen tehtävän vastausta sanotaan algoritmin kompleksisuudeksi.

Tehtävä 4.4 Lajittelujärjestys

Edellinen algoritmi (*selection sort*) toimi siten, että kortit jäivät pöydälle suurin päällimmäiseksi. Miten algoritmia pitää muuttaa, jotta pienin saataisiin päällimmäiseksi?

Ei siis ole suurtakaan väliä pitääkö lajitella nouseva vai laskeva järjestys!

4.2.4 Kuplalajittelu

Kokeillaanpa vielä erästä algoritmia: Sotke kortit kädessäsi uudelleen.

Bubble sort:

```
Vertaa aina kahta peräkkäistä korttia keskenään.  Mikäli ne
ovat väärässä järjestyksessä, vaihda ne keskenään.  Kun koko
pakka on käyty lävitse, aloita alusta ja jatka kunnes yhtään
kertaa ei tarvitse vaihtaa peräkkäisiä kortteja.
```

Tehtävä 4.5 Kuplalajittelu

Tuleeko pakka järjestykseen tällä algoritmilla? Voidaanko algoritmia nopeuttaa mitenkään? Kirjoita algoritmista "lausekielinen" versio.

4.2.5 Lajittelu avaimen mukaan

Kirjoita nyt joukko pahvilappuja, joissa kussakin on henkilön nimi, osoite ja puhelinnumero.

Sekoita laput ja kokeile toimiiko edelliset algoritmit mikäli laput järjestetään nimien mukaan. Ai tyhmä ehdotus! Tässä se onkin ohjelmoinnin vaikeus. Asiat ovat yksinkertaisia! Eiväthän ne osoitteet siellä lajittelua sotke.

Mikäli laput järjestetään nimen mukaan, sanotaan nimen olevan lajitteluavaimena. Lajitteluavaimeksi voitaisiin valita myös osoite tai puhelinnumero. Mikäli kahdella henkilöllä olisi sama nimi, voitaisiin nämä kaksi järjestää osoitteen perusteella. Tällöin lajitteluavain muodostuisi merkkijonosta johon olisi yhdistettynä nimi ja osoite.

4.2.6 Algoritmin parantaminen

Kaikki edelliset algoritmit ovat kompleksisuudeltaan normaalitapauksessa samanlaisia.

Tehtävä 4.6 Loppuminen erikoistapauksessa

Mikä edellisistä algoritmeista loppuu nopeasti, mikäli kortit jo olivat järjestyksessä?

Ohjelman toimintaan saattamisen kannalta olisi riittävää löytää jokin toimiva algoritmi. Myöhemmin, mikäli ohjelman toiminta todetaan hitaaksi ko. algoritmin kohdalta, voidaan algoritmia yrittää tehostaa. Lajittelussa tehostus saattaisi olla vaikkapa *QuickSort* (mukana mm. C–kielen standardikirjastossa).

Tehtävä 4.7 QuickSortin kompleksisuus

Jos algoritmin kompleksisuus on esimerkiksi $2n^2+n$, sanotaan että kompleksisuus on $O(n^2)$, eli usein kiinnostaa vain kompleksisuuden suurin "potenssi". *QuickSortin* keskimääräinen kompleksisuus on $O(n \log_2 n)$. On olemassa myös erikoistapauksissa toimivia lajitteluja, joissa kompleksisuus on O(n). Piirrä kuva jossa on *Selection Sortin*, *QuickSortin* ja lineaarisen lajittelun käyttämä "aika" piirrettynä lajiteltavien alkioiden (n=10,100,1000,10000,1000000) funktiona.

Tehtävä 4.8 Lisäys oikealle paikalleen vaiko lisäys loppuun ja lajittelu?

Tutki kumpiko on työmäärältään edullisempaa jos järjestettyyn taulukkoon tulee lisättäväksi suuri määrä uusia alkiota

1) lisätä alkio aina taulukkoon oikealle paikalleen
2) lisätä alkio aina taulukon loppuun ja kun kaikki alkiot on lisätty, niin lajitella taulukko

4.3 Algoritmin tarkentaminen

Edellisissä lajittelualgoritmeissa oli vielä muutamia aukkopaikkoja! Etsi pienin? Laita oikealle paikalleen?

4.3.1 Pienimmän etsiminen

Miten kädessä olevista korteista voidaan etsiä pienin. Yksi mahdollisuus on kuljettaa "pienin ehdokasta" läpi koko pakan. Mikäli matkan varrelta löytyy parempi ehdokas,

otetaan tämä tilalle. Edellä mainittu kuplalajittelu korjattuna perustuu nimenomaan tähän ideaan.

Entä jos kädessä olevien korttien järjestystä ei haluta muuttaa? Voisimme menetellä esimerkiksi seuraavasti (alkuarvaus ja arvauksen korjaaminen):

```
0.  vedä kädessä olevan pakan ylin kortti hieman esille
    ota ensimmäinen kortti tutkittavaksi
1.  vertaa tutkittavaa korttia ja esiinvedettyä korttia
2.  mikäli tutkittava on pienempi, vedä se esiin ja työnnä
    edellinen takaisin
3.  siirry tutkimaan seuraavaa korttia ja jatka kohdasta 1.
    kunnes olet tutkinut koko pakan
```

4.3.2 Paikalleen sijoittaminen

Miten kortti sijoitetaan paikalleen jo lajiteltuun kasaan? Esimerkiksi seuraavasti:

```
0. laita uusi kortti päällimmäiseksi lajiteltuun kasaan
1. vertaa uutta ja seuraavaa
2. mikäli väärässä järjestyksessä, niin vaihda ne keskenään
   ja jatka kohdasta 1.
```

4.4 Haku järjestetystä joukosta

Usein tulee vastaan myös tilanne, jossa tietyn henkilön tiedot pitäisi hakea esimerkiksi nimen mukaan. Mikäli valittu tietorakenne on järjestetty nimen mukaan, voidaan hakemisessa käyttää vaikkapa puolitushakua.

Nimen hakeminen ei taas poikenne kortin etsimisestä järjestetystä korttipakasta vai mitä?

4.4.1 Suora haku

Kun kortit ovat järjestämättä, niin miten löydät haluamasi kortin?

```
Ota seuraava kortti.  Mikäli etsittävä niin lopeta, muuten ota taas seuraava.
```

Algoritmi on OK 13 kortille, mutta kokeilepa *Äystön* etsimistä puhelinluettelosta tällä algoritmilla (muista lukea jokainen nimi ennen *Äystöä*)!

4.4.2 Puolitushaku

Mikäli 13 korttiasi on järjestyksessä ja sinun pitäisi mahdollisimman vähällä pläräämisellä löytää vaikkapa pata 4, niin miten voisit menetellä?

```
1. laita pakka pöydälle kuvapuolet ylöspäin
2. laita pakka puoliksi
3. laita molemmat pakat pöydälle kuvapuolet ylöspäin
4. kummassako kasassa etsittävä on?
5. heitä se pakka pois jossa etsittävä ei ole
6. jos etsittävä ei päällimmäinen, niin jatka kohdasta 1.
```

Vaikuttaa tyhmältä 13 kortille, mutta kokeilepa 1000 kortilla! Tai kokeile nyt etsiä ÄYSTÖÄ puhelinluettelosta tällä algoritmilla.

Tehtävä 4.9 Puolitushaku

Kirjoita puolitushausta kunnon "lausekielinen versio" kun meillä on sivunumeroitu kirja, jonka kullakin sivulla on täsmälleen yhden henkilön tiedot. Sivunumeroita kirjassa on N-kappaletta. Aloitat sivuista S1=0 ja S2=N+1. Miten jatkat mikäli pitää etsiä nimi `NIMI`?

Tehtävä 4.10 Puolitushaun kompleksisuus

Mikä on puolitushaun kompleksisuus?

4.5 Yhteenveto

Tätä on ohjelmointi! Kykyä (ja rohkeutta) sanoa selvät asiat täsmällisesti. Jossain vaiheessa vaihdamme vain täsmällisyyden astetta ja "lausekielen" sijasta siirrymme käyttämään oikeata lausekieltä, esim. Java–kieltä. Nämä omatekoiset algoritmit kannattaa kuitenkin säilyttää ja kirjata näkyviin todellisen ohjelman kommentteihin. Arviot algoritmin nopeudesta kannattaa myös laittaa kommentteihin, jotta jälkeenpäin on helpompi etsiä jo tekovaiheessa hitaaksi epäiltyjä kohtia. Miksi jättää seuraavalle lukijalle sama tehtävä ihmeteltäväksi, jos olemme sen toteutuksen jo jonnekin kirjanneet.

Algoritmit kannattaa testata huolellisesti jossain tutussa ympäristössä. Hyvin moni ohjelmointiongelma vektoreiden (=taulukko, =kasa kortteja, =ruutupaperi, =sivunumeroitu kirja jne.) kanssa samaistuu johonkin jokapäiväiseen ilmiöön. Kuten etsiminen puhelinluettelosta, korttipakan järjestäminen jne. Yritä etsiä näitä yhteyksiä ja kokeile ensin ratkaista ongelma tällä tavoin. Siirrä ratkaisu sitten "lausekielelle" ja lopulta ohjelmointikielelle.

Äläkä yritä liikaa, vaan jaa aina ongelma pienempiin osiin, kunnes tulee vastaan sen kokoisia osaongelmia, jotka osataan ratkaista! Tällaista osaongelman ratkaisijaa sanotaan ohjelmointikielessä aliohjelmaksi.

Kun osaongelma on ratkaistu, unohda se miten sen ratkaisija toimii ja käsittele ratkaisijaa vain yhtenä yksinkertaisena toimenpiteenä (vrt. aikaisempi kävelyesimerkki). Tämä on myös eräs ohjelmoinnin "vaikeus". Kirjoittaja haluaa nähdä kaikkien osien toiminnan yhtäaikaisesti. Tämä on kuitenkin mahdotonta. Siis kun jokin osa tekee hommansa, niin tehköön se sen miten tahansa.

Huono on johtaja joka kyttää koko ajan alaisiaan, eikä luota siihen, että nämä tekevät heille annetun tehtävän. Tässä mielessä ohjelmointia voisi verrata yrityksen johtamiseen: Johtaja jakaa koko yrityksen pyörittämisessä tarvittavia tehtäviä alaisilleen (aliohjelmille). Nämä saattavat edelleen jakaa joitakin osatehtäviä omille alaisilleen (aliohjelma kutsuu toista aliohjelmaa) jne. Johtaja (=ohjelmoija ja pääohjelma) kokoaa alaisten tekemän työn toimivaksi kokonaisuudeksi ja firma tuottaa.

Tehtävä 4.11 Kumin paikkaus

Kirjoita algoritmi polkupyörän kumin paikkaamiseksi.

Tehtävä 4.12 Sunnuntai–ilta

Kirjoita algoritmi sunnuntai–illan viettoa varten (muista että ohjelmoinnin demot on maanantaina).

Tehtävä 4.13 Onkiminen

Kirjoita algoritmi 10 ei–alimittaisen kalan onkimiseksi mato–ongella.

Tehtävä 4.14 Järjestyksen kääntäminen päinvastaiseksi

Kirjoita algoritmi pöydälle levitetyn 13 kortin kääntämiseksi päinvastaiseen järjestykseen.

5. Algoritmeissa tarvittavia rakenteita

Tarvitaan nyt silmukoita,
kaiken maailman taulukoita,
eri ehtoja kummastella,
aliohjelmia aavistella.

Mitä tässä luvussa käsitellään?

- silmukat ja valintalauseet
- totuustaulut
- pöytätesti
- muuttujat
- taulukot
- osoittimet

Vaikka jatkossa keskitymmekin oliopohjaiseen ohjelmointiin, tarvitaan yksittäisen olion metodin toteutuksessa algoritmeja. Riippumatta käytettävästä ohjelmointikielestä, tarvitaan algoritmeissa aina tiettyjä samantyyppisiä rakenteita.

Käsittelemme seuraavassa tyypilliset rakenteet nopeasti lävitse. Tarvitsisimme asioille enemmänkin aikaa, mutta otamme asiat tarkemmin esille käyttämämme ohjelmointikielen opiskelun yhteydessä. Lukijan on kuitenkin asioita tarkennettaessa syytä muistaa, ettei rakenteet ole mitenkään sidottu ohjelmointikieleen. Vaikka ne näyttäisivät kielestä täysin puuttuvankin (esim. *assembler*), voidaan ne kuitenkin lähes aina toteuttaa.

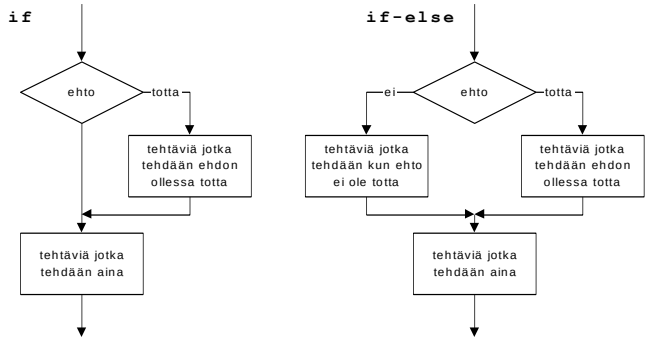
5.1 Ehtolauseet

Triviaaleja algoritmeja lukuun ottamatta algoritminen suoritus tarvitsee ehdollisia toteutuksia:

```
Jos kello yli puolenyön ota taksi
muuten mene linja-autolla
```

Ehtolauseita voi ehtoon tulla useampiakin ja tällöin on syytä olla tarkkana sen kanssa, mihin ehtoon mahdollinen `muuten`–osa liittyy:

```
Jos kello 00.00-07.00
  Jos sinulla on rahaa niin ota taksi
  muuten kävele
muuten mene linja-autolla
```



Kuva 5.1 Ehtolauseet

Tehtävä 5.1 Ajanlisäys

Jos sinulla on muuttujassa `t` tunnit ja muuttujassa `m` minuutit, niin kirjoita algoritmi miten lisäät `n` minuuttia kellonaikaan `t:m`.

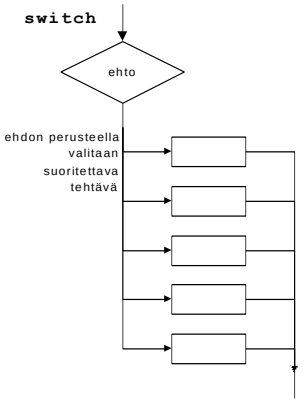
Tehtävä 5.2 Postimaksu

Kirjoita algoritmi `g`–painoisen kirjeen postimaksun määräämiseksi (saat keksiä hinnaston itse).

5.2 Valintalauseet

Usein ehtoja kasaantuu niin paljon, että peräkkäiset ja sisäkkäiset ehtolauseet muodostavat varsin sekavan kokonaisuuden. Tällöin voi olla helpompi käyttää valintalausetta:

```
Auto oli enenvanhaan rekisterinumeron 1. kirjaimen mukaan rekisteröity
seuraavassa läänissä:
  X     Keski-Suomen lääni
  K     Kuopion lääni
  M     Mikkelin lääni
  A,U   Uudenmaan lääni
```



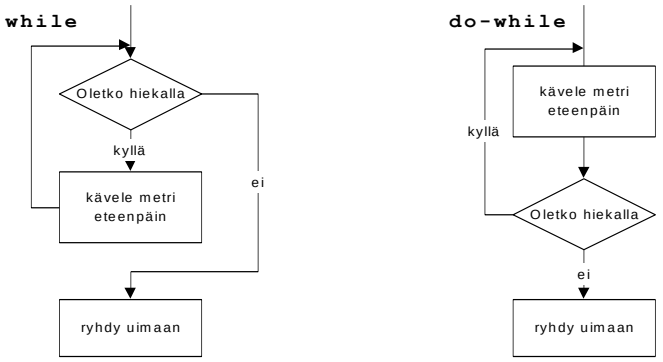
Kuva 5.2 `switch`–valintalause

Tehtävä 5.3 Korvaaminen ehtolauseilla

Esitä auton rekisteröintipaikan riippuvuus rekisterin ensimmäisestä kirjaimesta sisäkkäisten ehtolauseiden avulla.

5.3 Silmukat

Hyvin usein algoritmi tarvitsee toistoa: Esimerkiksi ohjeet (vuokaavio) hiekkarannalla toimimiseksi jos nenä näyttää merelle päin:



Kuva 5.3 `do`–silmukka ja `do-while`–silmukka

Ehtolause voi olla silmukan alussa, tällöin on mahdollista ettei silmukan runkoa tehdä yhtään kertaa. Ehto voi olla myös silmukan jälkeen, jolloin silmukan runko tehdään vähintään yhden kerran. Joissakin kielissä on lisäksi mahdollisuus silmukan rungon keskeltä poistuminen.

Silmukoihin liittyy aina ohjelmoinnin eräs klassisimmista vaaroista: päättymätön silmukka! Tämän takia silmukoita tulee käsitellä todella huolella. Eräs oleellinen asia on aina muistaa suorittaa silmukan rungossa jokin silmukan lopetusehtoon vaikuttava toimenpide. Mitä tapahtuu muuten?

Myös silmukan suorituskertojen lukumäärän kanssa tulee olla tarkkana. Silmukka tulee helposti suoritettua yhden kerran liikaa tai yhden kerran liian vähän.

Tehtävä 5.4 Uiminen

Mitä eroa on kahdella edellä esitetyllä "uimaan–meno" –algoritmilla? Mitä ehtoja algoritmiin voisi vielä lisätä?

Tehtävä 5.5 Ynnää luvut 1–100

Kirjoita algoritmi lukujen 1–100 yhteenlaskemiseksi sekä `do-while`– että `while` –silmukan avulla.

5.4 Muuttujat

Algoritmeissa tarvitaan usein muuttujia.

```
kellonaika
rahan määrä
```

5.4.1 Yksinkertaiset muuttujat

Yksinkertaisessa tapauksessa muuttuja voi olla yksinkertaista tyyppiä kuten kellonaika (jos ilmaistu minuutteina), rahasumma jne.

Yksinkertainen luvun jaollisuuden testausalgoritmi voisi olla vaikkapa seuraavanlainen:

```
Jaetaan tutkittavaa lukua jakajilla 2,3,5,7...luku/2.
Jos jokin jako menee tasan, niin ei alkuluku:

0. Laita jakaja:=2, kasvatus:=1,
   Jos luku=2 lopeta, alkuluku
1. Jaa luku jakajalla.  Meneekö jako tasan?
   - jos menee, on luku jaollinen jakajalla, lopeta
2. Kasvata jakajaa kasvatus arvolla (jakaja:=jakaja+kasvatus)
3. Kasvatus:=2; (koska parillisilla ei kannata enää jakaa)
4. Onko jakaja<luku/2?
   - jos on, niin jatka kohdasta 1
   - muuten lopeta, luku on alkuluku
```

Tehtävä 5.6 Vuokaavio

Piirrä jaollisuuden testausalgoritmista vuokaavio.

5.4.2 Pöytätesti

Hyvin usein algoritmi kannattaa pöytätestata. Pöytätesti alkaa kirjoittamalla sarakkeiksi kaikki algoritmissa esiintyvät muuttujat. Muuttujiksi voidaan kirjoittaa myös algoritmissa esiintyviä ehtoja. Tällainen muuttuja voi saada arvon `kyllä` tai `ei`. Pöytätestin riveiksi kirjoitetaan algoritmin eteneminen vaiheittain. Sarakkeisiin muuttujille kirjoitetaan uusia arvoja vain niiden muuttuessa.

Testataan esimerkiksi edellisen esimerkin algoritmi:

askel	Luku	Jakaja	Kasvatus	Luku/Jakaja	Jako tasan?	Jakaja<Luku/2?	Tulostus
0	25	2	1				
1				12.500	ei		
2		3					
3			2				
4						3<12.5	
1				8.333	ei		
2		5					
3			2				
4						5<12.5	
1				5.000	kyllä		Jaollinen 5:llä

askel	Luku	Jakaja	Kasvatus	Luku/Jakaja	Jako tasan?	Jakaja<Luku/2?	Tulostus
0	23	2	1				
1				11.500	ei		
2		3					
3			2				
4						3<11.5	
1				7.667	ei		
2		5					
3			2				
4						5<11.5	
1				4.600	ei		
2		7					
3			2				
4						7<11.5	
1				3.286	ei		
2		9					
3			2				
4						9<11.5	
1				2.556	ei		
2		11					
3			2				
4						11<11.5	
1				2.091	ei		
2		13					
3			2				
4						13>11.5	Alkuluku

Usein pöytätesti antaa hyviä vinkkejä myös algoritmin jatkokehittelylle. Käytännön työssä osa pöytätestistä voidaan suorittaa debuggereiden avulla. Joskus kuitenkin voi olla niin paljon esitietoa algoritmille, että tarvittavan testiohjelman rakentaminen voi olla työlästä. Pöytätestihän voidaan aloittaa minkälaisesta alkutilasta tahansa. Samoin yksi pöytätestin etuja on siitä jäävä historia. Usein debuggerit näyttävät vain yhden ajanhetken tilanteen, siis yhden pöytätestin rivin kerrallaan.

Tehtävä 5.7 Algoritmin parantaminen

Tarvitsisimmeko algoritmin kohtaa 4 lainkaan? Voitaisiinko algoritmin lopetus hoitaa muuten?

Tehtävä 5.8 Pöytätesti

Pöytätestaa edellinen algoritmi kun syöttönä on luku 121.
Pöytätestaa molemmat *Ynnää luvut 1–100* –algoritmisi versiona *Ynnää luvut 1–6*.

5.4.3 Yksiulotteiset taulukot

Tutkikaamme aikaisempia korttipakkaesimerkkejämme! Nyt tietorakenteeksi ei enää riitäkään pelkkä yksi muuttuja. Mikäli pakasta on otettu esiin pelkät padat, tarvitsisimme 13 muuttujaa. Näiden kunkin nimeäminen erikseen olisi varsin työlästä.

Tarvitsemme siis jonkin muun tietorakenteen. Mahdollisuuksia on useita: listat, jonot, pinot ja taulukot. Ohjelmoinnin alkuvaiheessa taulukot ovat tietorakenteista helpoimpia, joten keskitymme niihin aluksi.

Varataan pöydältä tilaa leveyssuunnassa 13 kortille. Varattua tilaa voimme nimittää taulukoksi tai vektoriksi. Taulukon yksi alkio on yhdelle kortille varattu paikka. Taulukon yhden alkion sisältö on se kortti, joka on siinä paikassa.

Mikäli numeroimme varatut paikat vaikkapa 0:sta alkaen vasemmalta oikealle, on meidän korteillamme osoitteet 0–12:

0	1	2	3	4	5	6	7	8	9	10	11	12
♠ 7	♠ 3	♠ K	♠ 2	♠ 5	♠ 9	♠ 4	♠ 6	♠ Q	♠ 10	♠ J	♠ A	♠ 8

Nyt voimme käsitellä yksittäisiä kortteja aivan kuin ne olisivat yksittäisiä muuttujia. Viittaamme tiettyyn korttipaikkaan (taulukon alkioon) sen indeksillä (olkoon taulukon nimi kortit):

```
paikassa kortit[5] meillä on pata 9
paikassa kortit[8] meillä on pata akka
```

Minkälaisia algoritmeja tulee vastaan taulukoita käsiteltäessä? Esim. ♠9:n siirtäminen taulukon viimeiseksi vaatisi ♠4:en siirtämistä paikkaan 5. ♠6:en siirtämistä paikkaan 6, ♠Q:n siirtämistä paikkaan 7 jne. Näin loppuun saataisiin raivatuksi paikka ♠9:lle.

Lajittelun ilman valtaisaa korttien siirtelyä voisimme hoitaa seuraavasti:

```
0. laita alku paikkaan 0
1. etsi alku paikasta lähtien pienin kortti
2. vaihda pienin ja paikassa alku oleva kortti
3. alku:=alku+1
4. mikäli alku<suurin indeksi, niin jatka 1
```

Sovitaan, että `ässä=1`. Nyt pienimmän kortin etsimisalgoritmi voisi olla seuraava:

```
0.  Alkuarvaus:  pien.paikka:=alku,  tutki:=alku
1.  Jos kortit[tutki] < kortit[pien.paikka]
      niin pien.paikka:=tutki
2.  tutki:=tutki+1
3.  Jos tutki<=suurin indeksi, niin jatka 1.
```

Voisimme vielä pöytätestata algoritmin:

askel	pien. paikka	tutki	Kortit 0	1	2	3	4	5	6	7	8	9	10	11	12	[tutki]< [pp]	tutki< suur.ind
0	0	0	♠7	♠3	♠K	♠2	♠5	♠9	♠4	♠6	♠Q	♠10	♠J	♠A	♠8		
1			⇑t													7<7 ei	
2&3		1															juu
1	1		⇑	t												3<7 juu	
2&3		2															juu
1				⇑	t											K<3 ei	
2&3		3															juu
1	3			⇑		t										2<3 juu	
2&3		4															juu
1						⇑	t									5<2 ei	
2&3		5															juu
1						⇑		t								9<2 ei	
2&3		6															juu
1						⇑			t							4<2 ei	
2&3		7															juu
1						⇑				t						6<2 ei	
2&3		8															juu
1						⇑					t					Q<2 ei	
2&3		9															juu
1						⇑						t				10<2 ei	
2&3		10															juu
1						⇑							t			J<2 ei	
2&3		11															juu
1	11					⇑								t		A<2 juu	
2&3		12															juu
1														⇑	t	8<A ei	
2&3		13															ei

Tehtävä 5.9 Lajittelun testaus

Oletetaan, että pienimmän etsimisalgoritmi toimii. Pöytätestaa edellä esitelty lajittelualgoritmi edellisen pöytätestin mukaisella korttien järjestyksellä.

Onko tämä *insertion sort*? Missä on lajiteltujen kasa ja missä lajittelemattomien?

Tehtävä 5.10 Korttien poisto

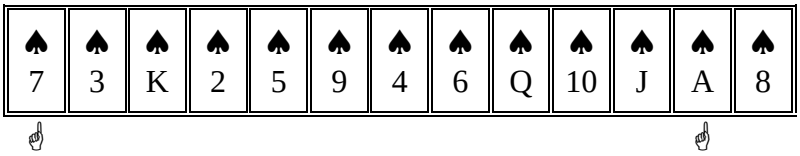
Kirjoita algoritmi kuvakorttien poistamiseksi taulukosta käyttäen indeksejä.

5.4.4 Osoittimet

Edellisessä pöytätestissä merkitsimme pienen merkin niiden korttien kohdalle, joita kunakin hetkenä tutkimme. Kun tutkimme esimerkiksi paikoissa 3 ja 10 olevia kortteja (`P2` ja `PJ`) voisimme sanoa, että muuttuja `pien.paikka` osoitti korttiin pata 2 ja muuttuja `tutki` korttiin pata jätkä. Näin ollen voisimme oikeastaan sanoa, että (indeksi)muuttujat `pien.paikka` ja `tutki` ovat osoittimia korttipakkaan. Niiden osoittamassa paikassa (indeksit 3 ja 10) on tietyt kortit (`P2` ja `PJ`).

Lajittelualgoritmi voitaisiin lausua esimerkiksi:

```
0. levitä kortit rinnakkain pöydälle
   osoita vasemman käden etusormella ensimmäiseen korttiin
1. etsi vasemman käden osoittamasta kohdasta alkaen oikealle
   pienin kortti ja osoita sitä oikean käden etusormella
2. vaihda etusormien kohdalla olevat kortit keskenään
3. siirrä vasemman käden etusormea yksi kortti oikealle päin
4. mikäli vasen sormi vielä kortin päällä, jatka kohdasta 1.
```



Osoittimen ja indeksin ero on siinä, että osoittimen tapauksessa emme yleensä koskaan ole kiinnostuneita itse osoittimen arvosta (osoitteesta, siitä indeksistä missä kohdin olemme menossa), vaan osoittimen osoittaman paikan sisällöstä (sormen kohdalla olevan kortin arvo tai ei korttia). Indeksejä käsitellessämme tutkimme monesti myös itse indeksin arvoa (`tutki=3 –>kortit[tutki]=P2`).

Osoitin voi tarvittaessa osoittaa myös itse taulukon ulkopuolelle. Mikäli kirjoittaisimme pöydälle numeroita, voisimme osoittaa sormella yhtä hyvin pöydälle kirjoitettuja numeroita (älkää hyvät ihmiset nyt töhrätkö pöytää!) kuin pöydälle levitettyjä kortteja (taulukon alkioita).

Siis indeksit liittyvät kiinteästi taulukoihin ja osoittimet voivat liittyä mihin tahansa tietorakenteisiin alkaen yksinkertaisesta muuttujasta päätyen monimutkaisen lista– tai puurakenteen alkioon.

Javassa tällä tavalla käyttäytyviä osoittimia vastaavat iteraattorit. Itse asiassa Javan kaikki "oliomuuttujat" ovat osoittimia, niitä sanotaan vaan viitteiksi. Erona esimerkiksi C++:n osoittimiin on se, että Javan viitteitä ei voi muuttaa muuta kuin osoittamaan toista alkiota. Eli Javan viitteillä käsky "*siirry yksi alkio eteenpäin*" on mahdotonta. Javan iteraattoreilla tämä sen sijaan onnistuu. C++:ssa on aidot osoittimet - joiden kanssa voi helposti myös möhliä laittamalla osoittimen osoittamaan paikkaan johon se ei saisi osoittaa). C++:ssa on myös viitteet (*reference*), joita tosin ei voi siirtää mihinkään luomisen jälkeen. C++:n iteraattorit muistuttavat jopa syntaksiltaan C++:n osoittimia ja itse asiassa C++:n osoitin käy algoritmissa paikkaan, johon tarvitaan iteraattori.

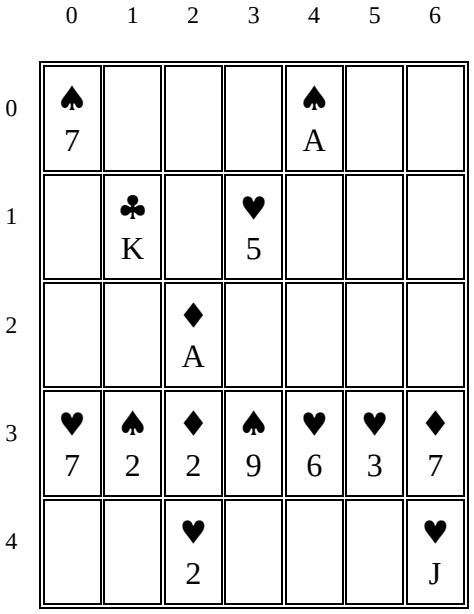
Tehtävä 5.11 Korttien poisto osoittimia käyttäen

Kirjoita algoritmi kuvakorttien poistamiseksi taulukosta käyttäen osoittimia.
Pöytätestaa algoritmi.

5.4.5 Moniulotteiset taulukot

Yksiulotteista taulukkoa voidaan verrata rivitaloon tai ruutupaperin yhteen riviin. Kaksiulotteinen taulukko on vastaavasti kuten kapea kerrostalo tai koko ruutupaperin yksi sivu. Tarvitsemme vastaavasti useampia osoitteita (indeksejä) osoittamaan millä rivillä ja missä sarakkeessa liikumme.

Alla on esimerkki 5x7 taulukosta (♠=pata, ♣=Risti, ♦=ruutu, ♥=hertta):



Jos taulukon nimi on `peli`, niin paikassa 3,1 on kortti pata 2:

```
peli[3][1] = ♠2
```

Tehtävä 5.12 Kaksiulotteisen taulukon indeksit

Kirjoita kaikkien esimerkissä olevien korttien osoitteet em. muodossa.

Kaksiulotteista taulukkoa nimitetään usein matriisiksi.

Usein taulukoiden indeksit ilmoitetaan eri järjestyksessä kuin koordinaatiston `(x,y)`–koordinaatit. Tämä johtuu siitä ajattelutavasta, että taulukon rivi sinänsä voidaan kuvitella yhdeksi alkioksi (rivityypiksi) ja tällöin ilmaisu

```
peli[3]
```

tarkoittaa koko riviä (♥`7, ♠2, ♦2, ♠9, ♥6, ♥3, ♦7`), jonka indeksi on kolme. Mikäli tämän perään laitetaan vielä `[1]`, niin tarkoitetaan ko. tietorakenteen alkiota jonka indeksi on yksi (♠`2`).

Tarvittaessa moniulotteiset taulukot voidaan muodostaa yksiulotteisenkin taulukon avulla. Esimerkin taulukko voitaisiin muodostaa yksiulotteisesta taulukosta siten, että

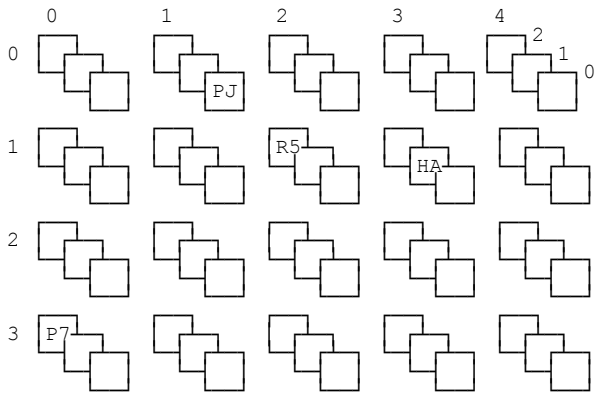
yksiulotteisen taulukon 7 ensimmäistä alkiota kuvaisivat matriisin 0:ta riviä, 7 seuraavaa matriisin ensimmäistä riviä jne.

Siis mikäli yksiulotteisen taulukon nimi olisi `pakka`, niin voisimme käyttää samaistuksia:

```
peli[3][1] = pakka[7*3+1]
peli[j][i] = pakka[7*j+i]
```

Olemme siis numeroineet kaksiulotteisen taulukon alkiot juoksevasti. Voimmehan tehdä näin myös kerrostalon huoneistoille tai teatterin istumapaikoille.

Taulukot voivat olla myös useampiulotteisia, esimerkiksi 3x4x5 taulukko:




```
isopeli[0][0][1]=PJ
isopeli[2][1][2]=R5
isopeli[1][1][3]=HA
isopeli[2][3][0]=P7
```

Tehtävä 5.13 Sijoitus 3–ulotteiseen taulukkoon

Esitä 5 muuta sijoitusta taulukkoon.

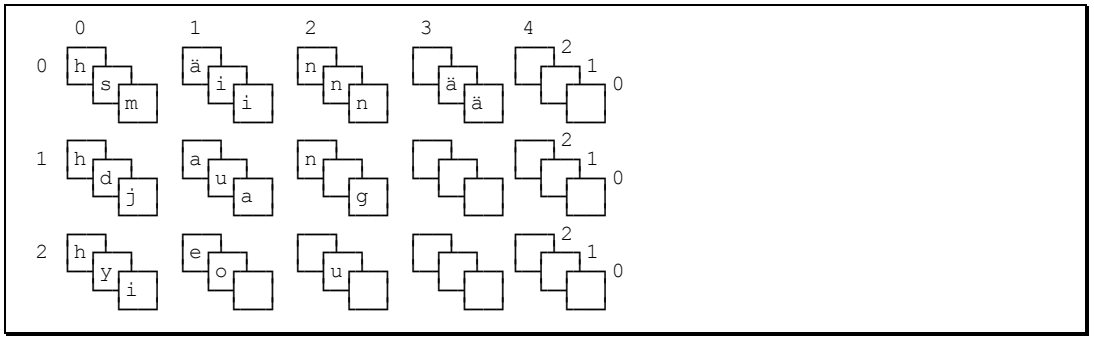
Tehtävä 5.14 3–ulotteinen taulukko 1–ulotteiseksi

Esitä kaava miten edellä oleva 3–ulotteinen taulukko voitaisiin esittää yksiulotteisella taulukolla.

Aikaisempi satunnaisen matkaajan sanastomme on oikeastaan myös kolmiulotteinen taulukko:

```
     0        1        2
0  minä     jag      i
1  sinä     du       you
2  hän      han      he
```

Se on kaksiulotteinen taulukko sanoista. Mitä sitten yksi sana on? Se on yksiulotteinen taulukko kirjaimista!



Siis "*you*" sanan indeksi on `[1][2]` ja sen kirjaimen "y" indeksi on `[0]`. Siis kaiken kaikkiaan "*you*"–sanan "y"-kirjaimen indeksi olisi `[1][2][0]`.

Taulukko voitaisiin järjestää 3–ulotteiseksi myös toisinkin. Esimerkiksi yhdessä "tasossa" olisi yksi kieli jne.

Tehtävä 5.15 3–ulotteinen taulukko

Esitä edellisessä esimerkissä kaikkien kirjainten indeksit.
Millaisella yhden kirjaimen sijoituksella muuttaisit sanan "*han*" sanaksi "*hon*"?

Tehtävä 5.16 4–ulotteinen taulukko

Mitenkä tavallinen kirja voitaisiin kuvitella 3–ulotteiseksi taulukoksi?
Miten kirja voitaisiin kuvitella 4–ulotteiseksi taulukoksi?
Piirrä edellisiin perustuen esimerkki 4–ulotteisesta taulukosta ja anna muutama esimerkkisijoitus.

Osoitinmuuttuja osoittaisi myös moniulotteisessa taulukossa yhteen alkioon kerrallaan. Esimerkiksi osoittamalla "*you*"–sanan "y"-kirjaimeen.

Moniulotteisen ja yksiulotteisen taulukon väliset muunnokset ovat tärkeitä, koska tietokoneen muisti (1997) on lähes aina yksiulotteinen. Siis loogisesti moniulotteiset taulukot joudutaan lopulta aina toteuttamaan yksiulotteisina. Onneksi useat kielet sisältävät moniulotteiset taulukot tietotyyppinä ja kääntäjät tekevät sitten muunnoksen. Tästä huolimatta esimerkiksi C–kielessä joudutaan usein muuttamaan moniulotteisia taulukoita yksiulotteisiksi.

5.4.6 Sekarakenteet

Taulukko voi olla myös taulukko osoittimista. Esimerkiksi sanastomme tapauksessa kaikki sanat voisivat olla yhdessä "möykyssä":

```
0         1         2         3
01234567890123456789012345678901234
minä jag i sinä du you hän han he
```

Itse sanasto voisi sitten olla taulukko osoittimia sanojen alkupaikkoihin:

	0	1	2
0	00	05	09
1	11	16	19
2	23	27	31

Siis taulukon paikasta `sanasto[1][0]` löytyy osoitin. Tämän osoittimen arvo on tässä esimerkissä 11. Siis osoitin viittaa sanan "sinä" alkuun. Tässä 2-ulotteinen taulukko osoittimista 1-ulotteiseen merkkitaulukkoon

```
// C++:lla
char *sanasto[3][3];
```

Tehtävä 5.17 Sanojen muuttaminen

Mitä ongelmia edellä olisi, mikäli yhdenkin sanan pituutta kasvatettaisiin?

Voitaisiinko edellä käyttää samoja sanoja uudestaan ja jos niin miten?

5.5 Osoittimista ja indekseistä

Osoitinmuuttujaa voitaisiin kuvitella myös seuraavasti: Olkoon meillä osoitekirja (`osoitteet`) jossa on sivuja:

```
sivu 0:                sivu 1:                sivu 2:
Kassinen Katto         Susi Sepe              Ankka Aku
Katto                  Takametsä              Ankkalinna
3452                   -                      1234
```

Meidän osoitekirjamme on tavallaan taulukko osoittimista (tässä tapauksessa osoitteita, älä sotke termejä!). Taulukon `osoitteet` paikassa 1 (sivu 1) on osoite "*Sepe Sudelle*". Mitä tapahtuu mikäli kirjoitamme kokonaan uuden henkilön osoitteen sivulla 1 olevan osoitteen päälle (sijoitetaan uusi arvo osoitinmuuttujalle `sivu[1]`)?

```
sivu 1:
Batman
Gotham City
999
```

```
C++:lla:
  sivu[1] = &Batman;
Javalla:
  Sivu[1] = Batman;
```

Mitä tapahtuu "*Sepe Sudelle*"? Tuskinpa sijoitus osoitekirjassamme siirtää "*Sepe Sutta*" yhtään mihinkään "*Perämetsästä*", tai tekee edes häntä murheelliseksi! Tämä on eräs tyypillinen virhekäsitys osoitinmuuttujia käytettäessä. Osoitinmuuttujaan sijoittaminen ei muuta tietenkään itse tiedon sisältöä. Mutta sijoittaminen siihen paikkaan johon osoitinmuuttuja osoittaa (esimerkissämme "*Sepe Suden*" asuntoon) muuttaa tietenkin myös itse tiedon sisältöä.

```
// C++:lla
sivu[1] = uusi_osoite;    // ei vaikuta Sepe Suteen
*sivu[1] = uusi_henkilo;  // laittaa uuden henkilön Sepen osoitteeseen
                          // = "tähdätään osoitetta pitkin"
// Javalla
sivu[1] = uusi_osoite;    // ei vaikuta Sepe Suteen
sivu[1].setNimi("Batman") // tämän lähemmäksi Javalla ei pääse
```

Vastaavasti jos meillä on indeksimuuttuja nimeltä `snro`, niin sijoitus muuttujalle

```
snro=2
```

ei muuta mitenkään itse sivun sisältöä. Vasta sijoitus

```
sivu[snro]=
```

muuttaisi sivun 2 sisältöä.

5.6 Aliohjelmat

Aliohjelma on tarkempi kuvaus tietylle asialle. Tämä kuvaus esitetään jossakin toisessa kohdassa ja sitä ei suinkaan tarvitse joka kerta lukea uudelleen.

Keittokirjassa lukee:

```
pyöritä lihapullataikinasta pyöreitä pullia
paista pullat kauniin ruskeiksi
```

Miten pullat paistetaan kauniin ruskeiksi. Tämä olisi edellisen algoritmin aliohjelma. Kokenut kokki ei välitä aina (eikä edes suostuisi) joka reseptin kanssa lukea itse paistamisohjetta. Aloittelija tätä saattaisi tarvita, jottei naapuri hälyttäisi palokuntaa paikalle liian savunmuodostuksen takia. Siis toivottavasti keittokirjasta jostakin kohti löytyisi myös itse paistamisohje.

Aliohjelmille on tyypillistä, että ne saattavat suoriutua vastaavista tehtävistä eri muuttujillekin. Näitä kutsukerroista riippuvia muuttujia sanotaan parametreiksi.

Esim. lihapullan paisto–ohje saattaa semmoisenaan kelvata myös tavalliselle pullalle:

```
paista("paistettava","c")
  Korvaa seuraavassa sana "paistettava" sillä mitä olet
  paistamassa ja "c" sillä lämpötilalla, joka keittokirjan
  ohjeessa ko. paistettavan kohdalla on:

0. laita "paistettavat" pellille
1. lämmitä uuni "c"-asteeseen
2. laita pelti uuniin
...
9. mikäli "paistettavat" ovat mustia mene ostamaan
   kaupasta valmiita
...
```

Tehtävä 5.18 Lihapullan paistaminen

Täydennä edellinen paistamisalgoritmi. Onko parametrejä tarpeeksi?

5.7 Vaihtoehtojen tutkiminen totuustaulun avulla

Usein helpostakin tehtävästä seuraa monia eri vaihtoehtoja, joiden täydellinen hallitseminen pelkästään päässä saattaa olla ylivoimaista. Tällöin avuksi tulee totuus– ja päätöstaulut. Päätöstaulu on totuustaulun pidemmälle viety versio.

Tutkikaamme aluksi muutamaa esimerkkiä jotka eivät suoranaisesti liity ohjelmointiin, mutta joissa esiintyy täsmälleen vastaava tilanne:

5.7.1 Kaikkien vaihtoehtojen kirjaaminen

Uusien opiskelijoiden lähtötasotesti 1991 (n. 20% oli osannut vastata oikein tähän tehtävään):

Tehtävä 3.4: Seisot tienhaarassa tuntemattomassa maassa. Toinen teistä vie pääkaupunkiin. Maan asukkaat joko valehtelevat aina tai puhuvat aina totta. Toisen tien alussa seisoskelee yksi heistä. Esität hänelle kyllä vai ei –kysymyksen ja saat vastauksen. Mitä kahta seuraavista neljästä kysymyksestä olisit voinut käyttää ollaksesi varma siitä, kumpi tie vie pääkaupunkiin – riippumatta siitä valehteleeko asukas vai ei?

1. Viekö tämä tie pääkaupunkiin?
2. Olisitko sanonut, että tämä tie vie pääkaupunkiin, jos olisin kysynyt sinulta sitä?
3. Onko niin, että tämä joko on tie pääkaupunkiin, tai sitten sinä puhut totta (mutta ei molemmin tavoin)?
4. Onko totta, että tämä tie vie pääkaupunkiin ja sen lisäksi sinä puhut totta?

Erotelkaamme eri kysymykset:

1 =	A	Viekö pääkaupunkiin?
	B	Puhutko totta? (mielenkiinnon vuoksi)
2 =	C	Olisitko sanonut että vie jos A?
3 <–	D	Tie pääk. XOR puhut totta?
4 <–	E	Tie pääk. AND puhut totta?

Nyt voimme kirjoittaa eri vaihtoehtojen mukaan seuraavan totuustaulun (`V=vastaus` kun valehteleminen otetaan huomioon):

Tie vie pääkaupunkiin	Puhuu totta	1		2	3		4	
		A	B	C	D	V	E	V
E	E	K	K	**E**	E	**K**	E	K
E	K	E	K	**E**	K	**K**	E	E
K	E	E	K	**K**	K	**E**	E	K
K	K	K	K	**K**	E	**E**	K	K

Siis 2 ja 3 vastauksissa on järkevä korrelaatio siihen viekö tie pääkaupunkiin vai ei.

5.7.2 Vaihtoehtojen lukumäärä

Mistä tiedämme milloin kaikki vaihtoehdot on kirjoitettu? Mikäli systeemiin vaikuttavia asioita on n kappaletta ja kukin on kyllä/ei tyyppinen (0/1), niin vaihtoehdot on helpointa saada aikaan kirjoittamalla kaikki n–bittiset binääriluvut järjestyksessä (esimerkissämme n=2) ja suorittamalla sitten tarvittavat samaistukset (esim. E=0 ja K=1). Vaihtoehtoja on tällöin 2^n.

```
00   -> E  E
01   -> E  K
10   -> K  E
11   -> K  K
```

Tehtävä 5.19 Kombinaatioiden lukumäärä

Olkoon meillä tehtävä, jossa yksi muuttuja voi saada arvot `K,E,tyhjä` ja toinen muuttuja arvot 5 ja 10. Kirjoita kaikki ko. muuttujien kombinaatiot.

Mikäli meillä on vaihtoehtoja n kappaletta ja kukin voi saada k_i eri arvoa, niin montako eri kombinaatiota saamme aikaiseksi?

5.7.3 Useita vaihtoehtoja samalla muuttujalla

Ottakaamme toinen vastaava tehtävä:

Tehtävä 4.3: Pekka valehtelee maanantaisin, tiistaisin ja keskiviikkoisin; muina viikonpäivinä hän puhuu totta. Paavo valehtelee torstaisin, perjantaisin ja lauantaisin; muina viikonpäivinä hän puhuu totta. Eräänä päivänä Pekka sanoi: "Eilen valehtelin!" Paavo vastasi: "Niin minäkin!" Mikä viikonpäivä oli?

Minä päivinä kaverukset saattaisivat sanoa ko. lausuman? Näitä päiviä on tietysti ne, jolloin joko eilen valehdeltiin ja tänään puhutaan totta TAI eilen puhuttiin totta ja tänään valehdellaan. (XOR)

päivä	Pekka	valehteli eilen	Paavo	valehteli eilen	
sunnuntai				k sanoo	
maanantai	V	sanoo			
tiistai	V	k			
keskiviikko	V	k			
torstai		k sanoo	V	sanoo	<==
perjantai			V	k	
lauantai			V	k	
sunnuntai				k sanoo	

Totuustaulun tavoitteena on siis kerätä kaikki mahdolliset vaihtoehdot ohjelmoijan silmien eteen, ja näin kaikki mahdollisuudet voidaan analysoida ja käsitellä.

Tehtävä 5.20 BAL=kyllä?

Eräällä saarella asuu luonnonkansa. Puolet kansan asukkaista aina valehtelevat ja toinen puoli puhuu aina totta. Lisäksi heidän kielensä on tuntematon. On saatu selville, että "BAL" ja "DA" tarkoittavat "kyllä" ja "ei", muttei sitä kumpiko tarkoittaa kumpaa. He ymmärtävät suomea mutta vastaavat aina omalla kielellään. Vastaasi tulee yksi saaren asukas.
a) Mitä saat selville kysymyksellä "Tarkoittaako BAL KYLLÄ"?
b) Millä kysymyksellä saat selville mikä sana on kyllä?

Tehtävä 5.21 Kuka valehtelee?

Jälleen maahan jossa asukkaat joko valehtelevat tai puhuvat aina totta. Tapaat kolme asukasta A:n, B:n ja C:n. He sanovat sinulla

A: Me kaikki kolme valehtelemme! **B:** Tasan yksi meistä puhuu totta!

Mitä ovat A, B ja C?

Entä jos asukkaat sanovat:

A: Me kaikki kolme valehtelemme! **B:** Tasan yksi meistä valehtelee!

Entä jos:

A: Minä valehtelen mutta B ei!

Entä jos:

A: B valehtelee! **B:** A ja C ovat samaa tyyppiä!

Vielä yksi:

A sanoo: B ja C ovat samaa tyyppiä. **C:ltä kysytään:** Ovatko A ja B samaa tyyppiä?

Mitä C vastaa?

5.7.4 Loogiset operaatiot

Ehtoja usein yhdistellään loogisten operaatioiden avulla:

```
Mikäli kello 7–20 ja et halua ulkoilla
– mene bussilla
...
Mikäli sinulla on rahaa tai saat kimpan
– ota taksi
```

Yksittäinen ehto antaa tulokseksi tosi (`T=true`) tai epätosi (`F=false`). Ehtojen tulosta voidaan usein myös kuvata 1 tai 0. Ehtojen yhdistämistä loogisilla operaatioilla kuvaa seuraava totuustaulu (myös *C++*:n loogiset operaattorit merkitty):

p	q	ja AND p && q	tai OR p || q	ehd. tai XOR p ^ q	ei NOT ! p
F 0	F 0	F 0	F 0	F 0	T 1
F 0	T 1	F 0	T 1	T 1	T 1
T 1	F 0	F 0	T 1	T 1	F 0
T 1	T 1	T 1	T 1	F 0	F 0

`^ toimii jos p ja q boolean`

Huomattakoon edellä, että AND operaatio toimii kuten kertolasku ja OR operaatio kuten yhteenlasku (mikäli määritellään 1+1=1). Siis loogisia operaattoreita voidaan käyttää kuten normaaleja algebrallisia operaattoreita ja niillä operoiminen vastaa tavallista algebraa. Loogisten operaatioiden algebraa nimitetään Boolen –algebraksi.

Ehtojen sieventämisessä käytettäviä kaavoja voidaan todistaa oikeaksi totuustaulujen avulla. Todistetaan esimerkiksi *de Morganin* kaava (vrt. joukko–oppi, `1=true`, `0=false`):

```
NOT (p AND q) = (NOT p) OR (NOT q)

Jaetaan ensin väittämä pienempiin osiin:
NOT   e1      =   e2    OR   e3
```

p	q	e1 p AND q	e2 NOT p	e3 NOT q	NOT e1	e2 OR e3
0	0	0	1	1	1	1
0	1	0	1	0	1	1
1	0	0	0	1	1	1
1	1	1	0	0	0	0

Koska kaksi viimeistä saraketta ovat samat ja kaikki muuttujien p ja q arvot on käsitelty, on laki todistettu!

Tehtävä 5.22 de Morganin kaava

Todista oikeaksi myös toinen *de Morganin* kaava:

NOT (p OR q) = (NOT p) AND (NOT q)

Tehtävä 5.23 Osittelulaki

Yhteenlaskun ja kertolaskun välillähän pätee osittelulaki:

```
p * (q + r) = (p * q) + (p * r)
```

Samaistamalla * <=> AND ja + <=> OR todetaan loogisille operaatioillekin osittelulaki:

```
p AND (q OR r) = (p AND q) OR (p AND r)
```

Todista oikeaksi toinen osittelulaki (toimiiko vast. yhteenlaskulla ja kertolaskulla):

```
p OR (q AND r) = (p OR q) AND (p OR r)
```

p	Q	r	e1 q AND r	e2 p OR q	e3 p OR r	p OR e1	e2 AND e3
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

Huomaa, että totuustauluun tulee nyt 8 riviä (koska kolme muuttujaa)!

Tehtävä 5.24 Ehtojen sieventäminen

Käytä *de Morganin* kaavoja tai osittelulakia seuraavien ehtojen sieventämiseen:

a) ei ole totta että hinta alle 5 mk ja paino yli 10 kg
b) NOT (kello<=7 OR rahaa>50 mk)
c) ((hinta < 5) tai (rahaa>10)) ja ((hinta < 5) tai (kello>9))

5.8 Muistele tätä

Mikäli edellä esitetyt asiat tuntuvat ymmärrettäviltä, niin ohjelmoinnissa ei tule olemaan mitään vaikeuksia. Jos vastaavat asiat tuntuvat vaikeilta ohjelmoinnin kohdalla, kannattaa palata takaisin tähän lukuun ja yrittää samaistaa asioita ohjelmointikieleen.

Taulukoiden samaistaminen ruutupaperiin, korttipakkaan tai muuhun tuttuun asiaan auttaa asian käsittelyä. Osoitinmuuttuja on yksinkertaisesti jokin (vaikkapa sormi) joka osoittaa johonkin (vaikkapa yhteen kirjaimeen).

Silmukat ja ehtolauseet ovat hyvin luonnollisia asioita.

Aliohjelmat ovat vain tietyn asian tarkempi kuvaus. Tarvittaessa tiettyä asiaa ei ongelmaa tarvitse heti ratkaista, vaan voidaan määritellä aliohjelma, joka hoitaa homman ja kirjoitetaan itse aliohjelman määrittely joskus myöhemmin.

Tehtävä 5.25 Merkkijonot

C–kielessä merkkijonot tullaan esittämään taulukoina kirjaimista. Merkkijonon loppu ilmaistaan kirjaimella NUL. Siis esimerkiksi `Kissa` olisi seuraavan näköinen

0	1	2	3	4	5
K	i	s	s	a	NUL

Kirjoita seuraavat algoritmit. Erityisesti kirjoita ensin algoritmin sanallinen versio

1. Välilyöntien poistaminen jonon alusta.
2. Välilyöntien poistaminen jonon lopusta.
3. Ylimääräisten (2 tai useampia) välilyöntien poistaminen jonosta.
4. Kaikkien ylimääräisten (alku-, loppu- ja monikertaiset) välilyöntien poistaminen.
5. Jonon muuttaminen siten, että kunkin sanan 1. kirjain on iso kirjain.
6. Tietyn merkin esiintymien laskeminen jonosta.
7. Esiintyykö merkkijono toisessa merkkijonossa (`kissatarha`, `sata` –> esiintyy; `kissatarha`, `satu` –> ei esiinny).

Tehtävä 5.26 Päivämäärät

Kirjoita seuraavat algoritmit:

1. Onko vuosi karkausvuosi vai ei. (Huom! 1900 ei, 2000 on)
2. Montako karkausvuotta on kahden vuosiluvun välillä.

3. Jos 1.1 vuonna 1 oli maanantai, niin mikä viikonpäivä on 1.1 vuonna `x`? (Oletetaan että kalenteri olisi ollut aina samanlainen kuin nytkin. Vihje! Tutki almanakkaa peräkkäisiltä vuosilta.)
4. Onko päivämäärä `pp.kk.vvvv` oikeata muotoa?

6. Esimerkkejä eri kielistä

Ompi Jaavaa ompi Ceetä,
Adaa ompi Pascalia.
Kieltä vanhaa, kieltä uutta
ne kaukaa sekä läheltä.

Monta kieltä monta mieltä,
kummajaisilla ku noilla,
voi koodia väänneskellä,
kaikellailla keikistellä.

Mitä tässä luvussa käsitellään?

- katsotaan mitä syntaktista eroa on eri ohjelmointikielillä yksinkertaisessa esimerkissä

Jossakin vaiheessa ohjelmoinnin opiskelua tullaan siihen, että ohjelma pitäisi toteuttaa jollakin olemassa olevalla ohjelmointikielellä (on tosin ohjelmointikursseja, joilla käytetään keksittyä ohjelmointikieltä).

Seuraavassa esitämme ohjelman jonka ainoa tehtävä on tulostaa teksti:

```
Terve!  Olen ??-kielellä kirjoitettu ohjelma.
```

Itse ohjelman suunnittelu on tällä kertaa varsin triviaali ja tehtävä ei tarvitse varsinaista tarkennustakaan, kaikki on sanottu tehtävän määrityksessä. Siis valitaan vain käytetyn kielen tulostuslause.

6.1 Esimerkkiohjelmat

Jotta lukija ymmärtäisi, ettei eri kielten välillä ole kuin pieni ero, esitämme ohjelman useilla eri kielillä. Ohjelman lopun jälkeen olevan poikkiviivan alapuolella on mahdollisesti esitetty miten ohjelmaa voitaisiin kokeilla *MS–DOS*–koneessa:

6.1.1 Pascal

```
{ Pascal-kieli }
PROGRAM olen(OUTPUT);
BEGIN
  WRITELN('Terve!  Olen Pascal-kielellä kirjoitettu ohjelma.');
END.
------------------------------------------------------------------------
- käynnistä vaikkapa Turbo Pascal
TP OLEN.PAS
- kirjoita ohjelma
- paina [Ctrl-F9]
```

6.1.2 C

```
/* C-kieli */
#include <stdio.h>
int main(void)
{
  printf("Terve! Olen C-kielellä kirjoitettu ohjelma.\n");
  return 0;
}
-----------------------------------------------------------------------
- käynnistä vaikkapa Turbo C
TC OLEN.C
- kirjoita ohjelma
- paina [Ctrl-F9]
```

6.1.3 C++

```
// C++ -kieli
#include <iostream.h>
int main(void)
{
  cout << "Terve!  Olen C++ -kielellä kirjoitettu ohjelma.\n";
  return 0;
}
-----------------------------------------------------------------------
- käynnistä vaikkapa Turbo C++
TC OLEN.CPP
- kirjoita ohjelma
- paina [Ctrl-F9]
- huomattakoon, että myös OLEN.C kelpaisi sellaisenaan C++ ohjelmaksi
```

6.1.4 Java

```
// Java -kieli
public class Olen {
   public static void main(String[] args) {
      System.out.println("Terve, Olen Java-kielella kirjoitettu ohjelma.");
   }
}
-----------------------------------------------------------------------
- Kirjoita Olen.java jollakin editorilla
- käännä: javac Olen.java
- aja: java Olen
```

6.1.5 Fortran

```
C Fortran-kieli
      PRINT*,'Terve! Olen Fortran-kielellä kirjoitettu ohjelma.'
      STOP
      END
```

6.1.6 ADA

```
-- ADA-kieli
with TEXT_IO; use TEXT_IO;
procedure ADA_MALLI is
  pragma MAIN;
begin
  PUT("Terve! Olen ADA-kielellä kirjoitettu ohjelma."); NEW_LINE;
end ADA_MALLI;
```

6.1.7 BASIC

```
REM BASIC-kieli
  PRINT "Terve! Olen BASIC-kielellä kirjoitettu ohjelma."
END
------------------------------------------------------------------------
- käynnistä vaikkapa Quick Basic
QBASIC OLEN.BAS
- kirjoita ohjelma
- paina [Alt-R][Return]
```

6.1.8 APL

```
⍝ APL-kieli
□ ← 'Terve! Olen APL-kielellä kirjoitettu ohjelma.'
```

6.1.9 Modula–2

```
(* Modula-2 -kieli *)
MODULE olen;
FROM InOut IMPORT WriteString, WriteLn;
BEGIN
  WriteString("Terve! Olen Modula-2 -kielellä kirjoitettu ohjelma.");
  WriteLn;
END olen.
```

6.1.10 Lisp

```
; AutoLisp
(prin1 "Terve! Olen AutoLisp-kielellä kirjoitettu ohjelma.")
------------------------------------------------------------------------
- Kirjoita rivi AutoCadissä (rivi tosin tulostuu 2x)
```

6.1.11 FORTH

```
 ( FORTH-kieli )
: Olen
  " Terve! Olen FORTH-kielellä kirjoitettu ohjelma." TYPE CR
;
```

6.1.12 Assembler

```
; 8086 assembler
DOSSEG
.MODEL TINY
.STACK
.DATA
viesti DB 'Terve! Olen 8086-assemblerilla kirjoitettu ohjelma.',0DH,0AH,'$'
.code
olen PROC NEAR
  MOV  AX,@@DATA
  MOV  DS,AX            ; Viestin segmentti DS:ään
  MOV  DX,OFFSET viesti ; Viestin offset osoite DX:ään
  MOV  AH,09H           ; Funktiokutsu 9 = tulosta merkkijono DS:DX
  INT  21H              ; Käyttöjärjestelmän kutsu
  MOV  AX,4C00H         ; Funktiokutsu 4C = ohjelman lopetus
  INT  21H              ; Käyttöjärjestelmän kutsu
olen ENDP
  END olen
-----------------------------------------------------------------
- kirjoita ohjelma jollakin ASCII-editorilla nimelle OLEN.ASM
- anna käyttöjärjestelmässä komennot (oletetaan että TASM on polussa):
TASM OLEN
TLINK OLEN
OLEN
```

Siis erot eri kielten välillä ovat hyvin kosmeettisia (Pascalin `BEGIN` on C:ssä `{` jne.). Jollakin kielellä asia pystyttiin esittämään hyvin lyhyesti ja jossakin tarvitaan enemmän määrittelyjä. Ainoastaan *assembler*–versio on sellaisenaan epäselvä, suoritettavia lauseita on täytynyt kommentoida enemmän.

6.2 Käytettävän kielen valinta

Kullakin ohjelmointikielellä on omat etunsa. Pascal on hyvin tyypitetty kieli ja sillä ei ole aloittelijankaan niin helppo tehdä eräitä tyypillisiä ohjelmointivirheitä kuin muilla kielillä. Standardi Pascal on kuitenkin hyvin suppea ja siitä puuttuu esim. merkkijonojen käsittely.

Turbo Pascalin laajennukset tekevät kielestä erinomaisen ja nopean kääntäjän ja `UNIT`–kirjastojen ansiosta se on todella miellyttävä käyttää. Nykyisin lisäksi *Delphi*–sovelluskehittimen kielenä on juuri Turbo Pascalista laajennettu *Object Pascal*. *Delphi* on eräs merkittävimmistä ja helppokäyttöisimmistä työkaluista *Windows*–ohjelmointiin. *Borlandin* julkaistua *Kylix*-sovelluskehittimen, tarjotaan myös *Linux*-ohjelmoijille samaa helppoutta..

BASIC–kieli on yleensä suppea kieli ja siksi helppo oppia. Lukuisten eri murteiden takia ohjelmien siirtäminen ympäristöstä toiseen on lähes mahdotonta. *Microsoftin Visual Basic* on kuitenkin *Windows*–ympäristössä nostanut *Basicin* jopa ohjelmankehittäjien työkaluksi. Alkuperäiset *Basic*-murteet olivat huonoja opiskelukieliä rajoittuneiden rakenteiden ja automaattisen muuttujanluomisen vuoksi. *Visual Basicin* uusimmissa versioissa on jo mukana yleisimmin tarvittavat rakenteet.

Fortran on luonnontieteellisissä sovelluksissa eniten käytetty kieli ja siihen on saatavissa laajat aliohjelmakirjastot useisiin numeriikan ongelmiin. Mikroissa kääntäjät ovat kuitenkin hitaita. Fortran–77 standardi on eräs parhaista standardeista, jota seuraamalla ohjelma toimii lähes koneessa kuin koneessa. Fortranin uusin standardi –90 tarjoaa ennen kielestä puuttuneet rakenteet.

ADA on Pascalin kaltainen vielä tarkemmin tyypitetty kieli, jossa on joitakin olio–ominaisuuksia. Se on Yhdysvaltain puolustusministeriön tukema kieli, joten se lienee tulevaisuuden kieliä. Sopii erityisesti raskaiden reaaliaikatoteutusten kirjoittamiseen (esim. ohjusjärjestelmät). GNU ADA tuo kääntäjän käyttämisen mahdolliseksi jokaiselle. Lisäksi ADA-95 tuo kieleen olio-ominaisuudet.

C–kieli on välimuoto Pascalista ja konekielestä. Ohjelmoijalle sallitaan hyvin suuria vapauksia, mutta toisaalta käytössä on hyvät tietotyypit ja rakenteet. Hyvä kieli osaavan ohjelmoijan käsissä, mutta aloittelija saattaa saada aikaan katastrofin. *ANSI–C* on suhteellisen hyvin standardoitu ja sitä seuraamalla on mahdollista saada ohjelma toimimaan pienin muutoksin myös toisessakin laiteympäristössä. Lisäksi *ANSI–C*:n tuoma funktioiden prototyypitys ja muutkin tyyppitarkistukset poistavat suuren osan ohjelmointivirheiden mahdollisuuksista, eivät kuitenkaan kaikkia. UNIX –käyttöjärjestelmän leviämisen myötä C on kohonnut erääksi kaikkein käytetyimmistä kielistä.

C++ on C–kielen päälle kehitetty oliopohjainen ohjelmointikieli. Aikaisemmin C–kielellä oli niin suuri merkitys, että se kannatti ehkä opetella aluksi. Nykyisin jokainen merkittävä C–kääntäjä on myös C++–kääntäjä. Oliopohjaisen ohjelmoinnin kannalta on parempi mitä aikaisemmin olio–ohjelmointi opetellaan. Valitettavasti C++ ei ole hybridikielenä (*multi paradigm*) paras mahdollinen ohjelmoinnin opetteluun. Kuitenkin paremmin opetteluun soveltuvat kielet ovat usein "leikkikieliä", kuten alkuperäinen Pascalkin oli. *Delphi* olisi mahtavan graafisen kirjastonsa ja kehitysympäristönsä ansiosta loistava opettelutyökalu, valitettavasti vaan lehti–ilmoituksissa harvoin haetaan *Delphi*–osaajia! Kohtuullisena kompromissina *C++*:kin voidaan valita opettelukieleksi, kunhan ei heti yritetä opetella kaikkia kielen kommervenkkeja.

Java on verkkoympäristössä tapahtuvaan ohjelmointiin kehitetty oliokieli. *Javan* erikoisuus on se, että se käännetään siirrettävään *Java*-tavukoodimuotoon. Tätä tavukoodia voidaan sitten suorittaa lähes missä tahansa ympäristössä *Java*-virtuaalikoneen avulla. *Javaa* on sanottu *C++:aa* yksinkertaisemmaksi, mutta kuitenkin *Java* kirjat ovat yhtä paksuja kuin *C++* kirjatkin. Nykyisin onkin monesti niin, ettei itse kieli ole ongelma, vaan sille kehitettyjen aliohjelmakirjastojen opettelu ja käyttö. *Javan* suurimpana etuna on sen lähes kaikissa koneissa toimiva graafinen kirjasto. *Java*-työkalut kehittyvät kovaa vauhtia, valitettavasti käyttöliittymän graafiseen suunniteluun tarkoitetut työkalut eivät ole keskenään yhteensopivia.

Tämän kurssin eri versioissa on käytetty esimerkkikielenä Pascalia, C:tä, C++:aa ja nyt tässä versiossa Javaa. Kielen valinta ei suuria merkitse, ohjelmointi on kuitenkin perusteiltaan samanlaista. Joitakin vivahde-eroja kuitenkin tulee valitun kielen mukana.

7. Java –kielen alkeita

Kommenttit jo käyttämäksi
muistiksipa merkit muille
selvennykseksi sepille
omaksikin ovat iloksi.

Vakioksi alkuun tiedot
kevenee koodin korjaaminen
mukavampi muutos aina
sulavampi säätäminen.

Koodi ensin käännettävä
syntaksikin syynättävä
tuo tulkilla tulkattava
siitä sitten suorittava.

Mitä tässä luvussa käsitellään?

- Java-kielisen ohjelman peruskäsitteet
- kääntämisen ja linkittämisen merkitys
- paketin käyttöönotto
- vakioarvot

Syntaksi:

```
kommentti:          /* vapaata tekstiä, vaikka monta riviäkin */
kommentti:          // loppurivi vapaata tesktiä
luokan ottaminen: import paketin_nimi.Luokka; import paketin_nimi.*;
vakio:            static final tyyppi nimi = arvo;
tulostus:         System.out.println(merkijono);
merkkijono:       "merkkejä"
```

Luvun esimerkkikoodit:

http://www.mit.jyu.fi/~vesal/kurssit/ohj2/moniste/esim/java-alk/

Ohjelman toteuttamista varten täytyy valita jokin todellinen ohjelmointikieli. Lopullisesta ohjelmasta ei valintaa toivottavasti huomaa. Valitsemme käyttökielen tällä kurssilla puhtaasti "markkinaperustein": paljon käytetyn ja työelämässä kysytyn - *Java*.

7.1 Hello World! Java –kielellä

java-alk\Hello.java - ensimmäinen Java ohjelma

```
// Ohjelma tulostaa tekstin Hello world!
class Hello {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Tehtävä 7.1 Nimi ja osoite

Kirjoita Java–ohjelma joka tulostaa:

```
Terve!
Olen Matti Meikäläinen 25 vuotta.
Asun Kortepohjassa.
Puhelinnumeroni on 603333.
```

7.2 Tekstitiedostosta toimivaksi konekieliseksi versioksi

7.2.1 Kirjoittaminen

Ohjelmakoodi kirjoitetaan millä tahansa tekstieditorilla tekstitiedostoon vaikkapa nimelle `Hello.java`. Yleensä tiedoston tarkennin määrää ohjelman tyypin.

7.2.2 Kääntäminen

Valmis tekstitiedosto käännetään ko. kielen kääntäjällä. Käännöksestä muodostuu usein objektitiedosto, joka on jo lähellä lopullisen ohjelman konekielistä versiota. Objektitiedostosta puuttuu kuitenkin mm. kirjastorutiinit. Kirjastorutiinien kutsujen kohdalla on "tyhjät" kutsut.

Java-kielen tapauksessa käännöksen tuloksena syntyy Java-virtuaalikoneen (JVM) ymmärtämää tavukoodia. Esimerkin tiedosto kääntyy esimerkiksi komennolla:

```
javac Hello.java
```

Käännöksen tuloksena syntyvässä Hello.class-tiedossa on siis Java-tulkin ymmärtämää tavukoodia. Kuitenkin siitäkin puuttuu itse kirjastorutiinit. Erona muihin kieliin on se, että käännetty tiedosto toimii niissä ympäristöissä, joissa on JVM ja nuo puuttuvat rutiinit.

Varsinaisissa käännettävissä kielissä käännös pitää suorittaa uudelleen jos ohjelma halutaan siirtää toiseen ympäristöön.

7.2.3 Linkittäminen

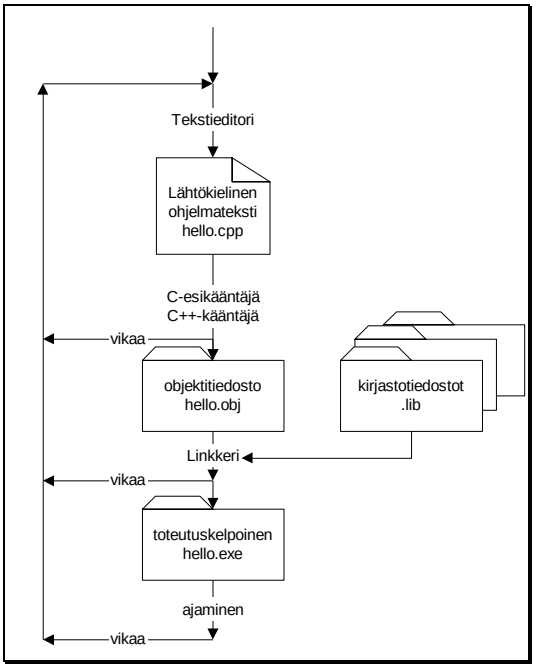
Linkittäjällä (kielestä riippumaton ohjelma) liitetään kirjastorutiinit käännettyyn objektitiedostoon. Linkittäjä korvaa tyhjät kutsut varsinaisilla kirjastorutiinien osoitteilla kunhan saa selville mihin kohti muistia kirjastorutiinit sijoittuvat. Näin saadaan valmis ajokelpoinen konekielinen versio alkuperäisestä ohjelmasta.

Javan tapauksessa varsinaista linkittämistä ei tarvita, vaan ohjelman suorituksen aikana etsitään tarpeellisia luokkia. Luokkien etsiminen voi tapahtua heti kun ensimmäistä luokkaa ladataan muistiin (*"static" resolution*) tai vasta kun luokkaan viitataan (*"laziest" resolution*) .

7.2.4 Ohjelman ajaminen

Käännetty ohjelma ajetaan käyttöjärjestelmästä riippuen yleensä kirjoittamalla ohjelman alkuperäinen nimi. Tällöin käyttöjärjestelmän lataaja–ohjelma lataa ohjelman konekielisen version muistiin ja siirtää prosessorin ohjelmalaskurin ohjelman ensimmäisenä suoritettavaksi tarkoitettuun käskyyn. Vielä tässäkin vaiheessa osa aliohjelmakutsujen osoitteista voidaan muuttaa vastaamaan sitä todellista osoitetta, johon aliohjelma muis-

tiin ladattaessa sijoittui. Tämän jälkeen vastuu koneen käyttäytymisestä on ohjelmalla. Onnistunut ohjelma päättyy aina ennemmin tai myöhemmin käyttöjärjestelmän kutsuun, jossa ohjelma pyydetään poistamaan muistista.



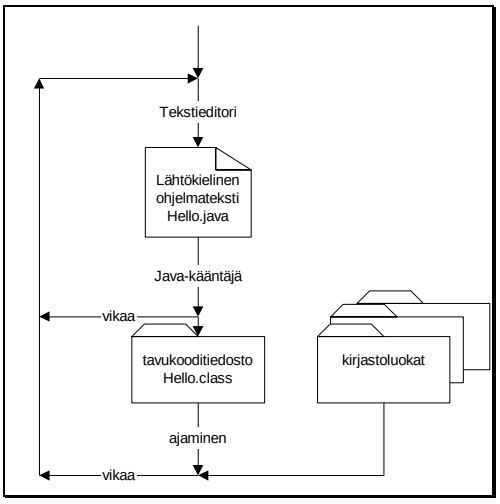
Kuva 7.1 Ohjelman kääntäminen ja linkittäminen

Javan tapauksessa ajaminen suoritetaan antamalla .class tai .jar tiedosto Java-virtuaalikoneelle (*Java Virtual Machine*, *JVM*). Esimerkkimme tapauksessa komennolla

```
java Hello
```

Jos luokasta `Hello` löytyy julkinen luokkametodi (staattinen metodi) nimeltä `main`, niin ohjelman suoritus aloitetaan siitä. Mikäli metodia ei löydy, tulee virheilmoitus:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```



Kuva 7.2 Ohjelman kääntäminen ja linkittäminen

7.2.5 Varoitus

Alkuperäisellä editorilla kirjoitetulla ohjelmakoodilla ei ole tavallista kirjettä kummempaa virkaa ennen kuin teksti annetaan kääntäjäohjelman tutkittavaksi. Käännöksen jälkeen alkuperäinen teksti voitaisiin periaatteessa vaikka hävittää - käytössä tietysti alkuperäinen teksti säilytetään ylläpidon takia! Siis me kirjoitamme tekstiä, joka ehkä (toivottavasti) muistuttaa Java–kielen syntaksin mukaista ohjelmaa. Vasta käännös ja linkkaus tekevät todella toimivan ohjelman.

7.2.6 Integroitu ympäristö

On olemassa ohjelmankehitysympäristöjä, joissa editori, kääntäjä ja linkkeri (sekä mahdollisesti debuggeri, virheenjäljitin) on yhdistetty käyttäjän kannalta yhdeksi toimivaksi kokonaisuudeksi. Esimerkkeinä *Borland Jbuilder, NetBeans, Microsoftin Visual–Studio* ja *Borland–C++ Builder* . Kaikissa listassa mainituissa kehittimissä on myös tuki käyttöliittymän suunnittelulle.

Esimerkiksi *Borlandin* ympäristöissä ohjelma kirjoitetaan tekstinä ja kun ohjelmakoodi on valmis, saadaan koodi käännettyä, linkitettyä ja ladattua ajoa varten vain painamalla `[F9]` (tai `[Ctrl-F9]` versiosta riippuen) .

Mahdollisia muita integroitujen ympäristöjen ominaisuuksia ovat mm: UML-kaavioiden ja muiden dokumenttien automaattinen tuottaminen (esim. *Delphi*n *ModelMaker*, *JBuilderin* luokkakaaviot, *JavaDoc*-yhteistoiminta Java-kehittimissä), jotka perinteisesti ovat olleet *CASE*-suunnitteluohjelmien aluetta. Lisäksi myös koodin generointi kaavioista onnistuu rajoitetusti.

7.3 Ohjelman yksityiskohtainen tarkastelu

Seuraavaksi tutkimme ohjelmaa lause kerrallaan:

java-alk\Hello2.java - malliohjelma

```
import java.lang.System;
/**
 *
 * Ohjelma tulostaa tekstin Hello world!
 * @author Vesa Lappalainen
 * @version 1.0, 03.01.2003
 */
class Hello2 {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

7.3.1 Tarvittavien luokkien esittely

Harvoin voi tehdä ohjelman joka tulee täysin toimeen ilman muiden apua. Javan tapauksessa ilman muiden luokkien apua. Jotta kääntäjä tietäisi mistä puhutaan, pitää kertoa mistä paketista luokka löytyy. Paketista java.lang löytyy `System`-niminen luokka, josta löytyy tarvitsemamme `out`-olio. Eli pitäisi oikeastaan kirjoittaa:

```
    java.lang.System.out.println("Hello world!");
```

Olioihin ja luokkiin paneudumme tarkemmin luvussa 9.

Jos kuitenkin samaan luokkaa tarvitaan useasti ja halutaan lyhentää kirjoittamista, voidaan `import`-lauseella kertoa ennen varsinaista koodin aloittamista apuna tarvittavat luokat.

```
import java.lang.System;
```

Jos haluttaisiin ottaa kaikki tietyn paketin luokat käyttöön, tämä voitaisiin tehdä rivillä:

```
import java.lang.*;
```

Poikkeuksen muodostaa paketti java.lang jota ei tarvitse välttämättä erikseen esitellä lainkaan. Näinhän oli tehty ensimmäisessä esimerkissämme.

7.3.2 Kommentti

```
// Ohjelma tulostaa tekstin Hello world!
```

tai

```
/* Ohjelma tulostaa tekstin Hello world! */
```

Ohjelman alussa on kommentoitu mitä ohjelma tekee. Yleensä ohjelmakoodit on hyvä varustaa kuvauksella siitä, mitä ohjelma tekee, kuka ohjelman on tehnyt, milloin ja miksi. Milloin ohjelmaa on viimeksi muutettu, kuka ja miten.

Lisäksi jokainen vähänkin ei–triviaali lause tai lauseryhmä kommentoidaan. Kommenttien tarkoituksena on kuvata ohjelmakoodia lukevalle lukijalle se mistä on kyse.

Lohkokommentti alkaa `/*` –merkkiyhdistelmällä ja päättyy `*/` –merkkiyhdistelmään. Lohkokommentteja voidaan sijoittaa Java–koodissa mihin tahansa mihin voitaisiin pistää myös välilyönti. Rivin loppuminen ei sinänsä lopeta lohkokommenttia. Kommentin sisällä SAA esiintyä `/` ja `*` –merkkejä yhdessä tai erikseen, muttei lopettavaa yhdistelmää `*/`.

Yleinen virhe on unohtaa lohkokommentin loppusulku pois. Mikäli esimerkissämme puuttuisi kommentin loppusulku, olisi koko loppuohjelma kommenttia ja mitään ohjelmaa ei siis olisikaan. Mikäli kääntäjä antaa vyöryn ihmeellisiä virheilmoituksia, kannattaa aina ensin tarkistaa kommenttisulkujen täsmäävyys. Tosin tähän auttaa nykyisten ohjelmointiympäristöjen värikoodien käyttö eri ohjelman osille, eli esimerkiksi kommentit näkyvät eri värisinä ja puuttuva komenttisulku paljastuu välittömästi.

Javassa yhden rivin kommentti voidaan ilmaista myös `//` –merkkiyhdistelmällä, jolloin rivinloppu lopettaa kommentin.

7.3.3 JavaDoc

```
/**
 *
 * Ohjelma tulostaa tekstin Hello world!
 * @author Vesa Lappalainen
 * @version 1.0, 03.01.2003
 */
```

Jos tiedot annetaan Javan dokumentoinnin standardimuodossa, niin tiedostoista saadaan sitten koostettua helposti HTML-muotoinen dokumentti. *JavaDoc*in mukainen kommentti alkaa ”sululla” `/**` ja päättyy normaaliin kommentin loppumerkkiin.

Kommentointi kannattaa käytännössä tehdä yksittäisten metodien tarkkuudella. Katso esimerkiksi tämän luvun viimeinen esimerkki.

Katso lisää *JavaDoc*:in käytöstä esimerkiksi:

```
http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
```

7.3.4 Luokan esittely

```
class Hello2 {
```

Jokainen Java-ohjelma sisältää vähintään yhden julkisen luokan. Kunkin tiedoston nimi on oltava sama kuin tiedostossa olevan julkisen luokan nimi + ’.java’. Palaamme luokkiin ja olioihin tarkemmin hieman myöhemmin. Usein olio-ohjelmoinnissa on tapana että luokkien nimet aloitetaan isolla kirjaimella.

Luokan esittely ja toteutus alkaa aaltosululla `{` ja päättyy toiseen lopettavaan aaltosulkuun `}`.

7.3.5 Pääohjelman esittely

```
public static void main(String[] args) {
```

Kun Java-tavukoodi ladataan muistiin, etsitään ensin ladatusta luokasta (tai muuten erikseen ilmoitetusta luokasta) pääohjelmaa, josta koodin suoritus aloitetaan. Pääohjelman nimi on aina oltava `main`. Oikeassa ohjelmassa on pääohjelman lisäksi useita luokkia ja metodeita (luokkien sisällä olevia aliohjelmia).

`main`-metodi voi olla myös useammassa luokassa, jolloin kullakin `main`-metodilla voidaan testata kyseisen luokan toiminta. Näin helpotetaan yksikkötestausta (modulitestausta). Tästä lisää kun pääsemme tarkemmin olioiden ja luokkien kimppuun.

Seuraavaksi esitellään ohjelman pääohjelma ("oikea" ohjelma koostuu isosta kasasta aliohjelmia ja yhdestä pääohjelmasta, jonka nimi on `main`).

`public` tarkoittaa, että metodi on julkisesti näkyvä. Muuten metodi ei näkyisi luokan ulkopuolelle eikä sitä voitaisi suorittaa.

`static` tarkoittaa että metodi on ns. luokkametodi, eli se voidaan suorittaa, vaikkei luokasta olisi olemassa yhtään esiintymää eli oliota. Luokkametodi ei voi käyttää luokan olioiden attribuutteja suoraan (koska oliota ei välttämättä ole).

`void` ilmoittaa, että metodi jota kirjoitamme ei palauta mitään arvoa (*eng. void* = mitätön).

`main` tarkoittaa pääohjelman nimeä. Tämä TÄYTYY aina olla `main`. Muut metodit voidaan nimetä vapaasti.

`(` Metodin parametrilistan (argumenttilistan) alkusulku.

`String[]` ilmoittaa että metodi saa parametrinaan taulukollisen (hakasulut tarkoittavat taulukkoa) merkkijonoja. Nämä ovat merkkijonot tulevat ohjelmaan käynnistyksen yhteydessä olevina parametreinä. Käynnistys parametrejä voi olla nolla tai useita.

`args` itse keksitty nimi jolla merkkijonotaulukkoon viitataan. Tämä nimi voi olla mikä tahansa.

`)` Metodin parametrilistan (argumenttilistan) loppusulku.

Ohjelma

java-alk\Hello3.java - tervehdys parametrina

```
/**
 * Ohjelma tulostaa kutsun mukana tulleet parametrit
 * @author Vesa Lappalainen
 * @version 1.0, 03.01.2003
 */
class Hello3 {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
      System.out.println("Parametri " + i + ": " + args[i]);
  }
}
```

Tulostaisi seuraavalla tavalla:

```
E:\kurssit\ohj2\moniste\esim\java-alk>java Hello3 eka toka kolmas
Parametri 0: eka
Parametri 1: toka
Parametri 2: kolmas
```

7.3.6 Lausesulut

`{ }` Javassa isompi joukko lauseita kootaan yhdeksi lauseeksi sulkemalla lauseet aaltosulkuihin. Metodin täytyy aina sisältää aaltosulkupari, vaikka siinä olisi vain 0 tai 1 suoritettavaa lausetta.

7.3.7 Tulostuslause

```
  System.out.println("Hello world!")
```

`System` on paketista java.lang löytyvä luokka, jossa on joukko hyödyllisiä oliota ja metodeja.

`out` on `System` luokan olio, joka sisältää mm. tulostukseen tarvittavia metodeja.

`println("?")` tulostaa ajonaikana sen tekstin, joka on lainausmerkkien välissä. Tulostuksen jälkeen vaihdetaan uudelle riville. Jos tarvitsee tulostaa useita eri tekstejä tai muuttujia välissä, voidaan niistä muodostaa +-operaatiolla uusi merkkijono, esimerkiksi:

```
System.out.println("Parametri " + i + ": " + args[i]);
```

7.3.8 Lauseen loppumerkki ;

`;` puolipiste lopettaa lauseen. Puolipiste voidaan sijoittaa mihin tahansa lopetettavaan lauseeseen nähden. Sen eteen voidaan jättää välilyöntejä tai jopa tyhjiä rivejä. Sen pitää kuitenkin esiintyä ennen uuden lauseen alkua. Näin Java–kieli ei ole rivisidonnainen, vaan Java–kielinen lause voi jakaantua usealle eri riville tai samalla rivillä voi olla useita Java-kielisiä lauseita.

Puolipisteen unohtaminen on tyypillinen syntaksivirhe. Ylimääräiset puolipisteet aiheuttavat tyhjiä lauseita, joista tosin ei ole mitään haittaa: *"Tyhjän tekemiseen ei kauan mene"* – sanoo tyhjän toimittaja.

7.3.9 Isot ja pienet kirjaimet

Isoilla ja pienillä kirjaimilla on Java–kielessä eri merkitys. Siis EI VOIDA KIRJOITTAA:

```
SysTem.Out.printLn("Hello!")              // VÄÄRIN!
```

☹

7.3.10 White spaces, tyhjä

Välilyöntejä, tabulointimerkkejä, rivinvaihtoja ja sivunvaihtoja nimitetään yleisesti yhteisellä nimellä "*white space*". Käännettäessä kommentit muutetaan yhdeksi välilyönniksi, joten myös kommenteista voitaisiin käyttää nimitystä "*white space*". Jatkossa käytämme nimitystä tyhjä tai tyhjä merkki, kun tarkoitamme "*white space*".

Java–koodi voi sisältää tyhjiä merkkejä missä tahansa, kunhan niitä ei kirjoiteta keskelle sanaa tai tekstiä määrittelevän `""`–parin ollessa auki. `""`–parin sisällä tyhjätkin merkit ovat merkityksellisiä.

Siis kääntäjän kannalta malliohjelmamme voitaisiin kirjoittaa myös seuraavillakin tavoilla:

☹

```
class
Hello4
{
public
static
void
main
(
String[]
args)
{
System
.
out
.
println
(
"Hello world!"
)
;
}
}
```

☹

```
class Hello5{public static void main
         (String[]                                          args){
                 System.out.println("Hello world!"              );}}
```

☹

```
class Hello6{public static void main(String[]args){System.out.println("Hello world!");}}
```

Yleinen tyyli on kuitenkin jakaa koodia riveihin ja sisentää lohkoja muutamalla pykälällä. Kunnes lukija on varma omasta tyylistään, kannattaa matkia tässä monisteessa (ei kuitenkaan edellisiä esimerkkejä) esitettyä kirjoitustapaa ohjelmille.

7.3.11 Vakiomerkkijonot

Voimme määritellä ohjelmaamme vakioita; eli arvoja jotka esiintyvät ohjelmassa täsmälleen yhden kerran. Näin ohjelmastamme saadaan helpommin muuteltava. Esimerkiksi seuraava ohjelma tulostaisi myös tekstin "`Hello world!`": . Vakioiden nimet on tapana kirjoittaa isoilla kirjaimilla.

java-alk\Hello7.java - tervehdys vakioksi

```
/**
 * Ohjelma tulostaa Hello World!  Tulostettava teksti on vakiona
 * @author Vesa Lappalainen
 * @version 1.0, 03.01.2003
 */
class Hello7 {
  static final String TERVE   = "Hello";
  static final String MAAILMA = "world!";

  public static void main(String[] args) {
    System.out.println(TERVE + " " + MAAILMA);
  }
}
```

Tehtävä 7.2 Terve maailma!

Kirjoita edellisestä ohjelmasta suomenkielellä tulostava versio (= suomenna ohjelma).

Tehtävä 7.3 Nimi ja osoite vakioksi

Kirjoita aikaisemmasta "Matti Meikäläinen asuu Kortepohjassa" –ohjelmasta versio, jossa nimi, osoite ja puhelin on esitelty vakioina.

7.3.12 Vakiolukuarvot

Vakiomäärittelyä voitaisiin käyttää esimerkiksi kokonaislukuvakioiden määrittelemiseen:

java-alk\Kuutio.java - monikulmion tiedot vakioksi

```
/**
 * Ohjelma tulostaa tietoja kuutiosta
 * @author Vesa Lappalainen
 * @version 1.0, 04.01.2003
 */
class Kuutio {
  static final String TAHOKAS    = "Kuutiossa";
  static final int    KARKIA     =  8;
  static final int    SIVUTASOJA =  6;
  static final int    SARMIA     = 12;

  public static void main(String[] args) {
    System.out.print  (TAHOKAS + " on " + KARKIA + " kärkeä,");
    System.out.print  (" " + SIVUTASOJA + " sivutasoa ja");
    System.out.println(" " + SARMIA + " särmää.");
  }
}
```

Tehtävä 7.4 Tetraedri

Muuta edellistä ohjelmaa siten, että tulostetaan samat asiat tetraedristä.

7.3.13 Tulostuksen muotoilu ja apumetodit

C:llä olisi edellisen esimerkin tulostus helppo muotoilla kutsulla:

```
  printf("%17s on %2d kärkeä,\n"     ,TAHOKAS,KARKIA);
```

Javassa joudumme kuitenkin tekemään saman asian eteen aluksi joukon apumetodeja:

java-alk\Kuutio4.java - monikulmion tulostus muotoillusti

```
/**
 * Ohjelma tulostaa tietoja kuutiosta "siistissä" muodossa
 * @author Vesa Lappalainen
 * @version 1.0, 05.01.2003
 */
class Kuutio4 {
  static final String TAHOKAS    = "Kuutiossa";
  static final int    KARKIA     =  8;
  static final int    SIVUTASOJA =  6;
  static final int    SARMIA     = 12;

  /**
   * Palauttaa jonon s muotoiltuna vähintään len-pituiseksi
   * <pre>
   * Esim:  fmt("2",3)  => "  2"
   *        fmt("2",-3) => "2  "
   * </pre>
   * @param s     muotoiltava jono
   * @param len   pituus, negatiivisella vasempaan laitaan, pos. oikeaan
   * @return      muotoiltu jono
   */
  static String fmt(String s,int len) {
    int needs = Math.abs(len) - s.length();
    if ( needs <= 0 ) return s;
```

```
    StringBuffer fill = new StringBuffer("                                    ");
    while ( fill.length() < needs ) fill.append("                             ");
    fill.delete(needs,1000);
    if ( len < 0 ) return s + fill;
    return fill + s;
  }

  /**
   * Tulostaa 2 merkkijonoa ja yhden kokonaisluvun siististi
   * @param s1 Ensimmäinen tulostettava jono
   * @param i  tulostettava kokonaisluku
   * @param s2 toinen tulostettava jono
   */
  static void tulosta(String s1, int i, String s2) {
    System.out.println(fmt(s1,20) + " " + fmt(String.valueOf(i),2) + " " + s2);
  }

  public static void main(String[] args) {
    tulosta(TAHOKAS + " on", KARKIA    , "kärkeä,");
    tulosta(""             , SIVUTASOJA, "sivutasoa ja");
    tulosta(""             , SARMIA    , "särmää.");
  }
}
```

Tehtävä 7.5 Apumetodit

Mieti miksi edellinen Java-ohjelma tulostaa tiedot seuraavassa muodossa:

```
0        1         2         3         4
12345678901234567890123456789012345678901234567890
------------------------------------------------
        Kuutiossa on  8 kärkeä,
                      6 sivutasoa ja
                     12 särmää.
```

8. Java–kielen muuttujista ja aliohjelmista

Turha koodi muuttujitta,
ompi onneton ohjelmaksi.
Parametri kutsuun pistä
aliohjelmalle argumentti.

Tarjoappa käyttöön tuota
metodia mielekästä
rutiinia riittävätä
itse tarkoin testattua.

Mitä tässä luvussa käsitellään?

- muuttujat
- malliohjelma jossa tarvitaan välttämättä muuttujia
- oliomuuttujat eli viitemuuttujat
- aliohjelmat, eli funktiot (metodit)
- aliohjelman testaaminen
- erilaiset aliohjelmien kutsumekanismit
- parametrin välitys
- lokaalit muuttujat
- pöytätesti

Syntaksi:

```
Seuraavassa muut = muuttujan nimi, koostuu kirjaimista,0-9,_, ei ala 0-9
muut.esittely:    tyyppi muut = alkuarvo;            // 0-1 x =alkuarvo
sijoitus:         muut = lauseke;
merkkijonon lukeminen,  ks. Syotto-luokka
aliohj.esittely:  tyyppi aliohj_nimi(tyypi muut, tyyppi muut);  // 0-n x muut
aliohj.kutsu      muut = aliohj_nimi(arvo, arvo);    // 0-1 x muut=, 0-n x arvo
olion luonti      Tyyppi olion_nimi = new Tyyppi(parametrit);
```

Luvun esimerkkikoodit:

http://www.mit.jyu.fi/~vesal/kurssit/ohj2/moniste/esim/java-muut/

8.1 Mittakaavaohjelman suunnittelu

Satunnainen matkaaja ajelee tällä kertaa kotimaassa. Autoillessa hänellä on käytössä Suomen tiekartan GT –karttalehtiä, joiden mittakaava on 1:200000. Viivoittimella hän mittaa kartalta milleinä matkan, jonka hän aikoo ajaa. Ilman matikkapäätä laskut eivät kuitenkaan suju. Siis hän tarvitsee ohjelman, jolla matkat saadaan muutettua kilometreiksi.

Millainen ohjelman toiminta voisi olla? Vaikkapa seuraavanlainen:

```
C:\OMA\MATKAAJA>matka[RET]
Lasken 1:200000 kartalta millimetreinä mitatun matkan
kilometreinä luonnossa.
Anna matka millimetreinä>35[RET]
Matka on luonnossa 7.0 km.
C:\OMA\MATKAAJA>matka[RET]
Lasken 1:200000 kartalta millimetreinä mitatun matkan
kilometreinä luonnossa.
Anna matka millimetreinä>352[RET]
Matka on luonnossa 70.4 km.
C:\OMA\MATKAAJA>
```

Edellisessä toteutuksessa on vielä runsaasti huonoja puolia. Mikäli samalla haluttaisiin laskea useita matkoja, niin olisi kätevämpää kysellä matkoja kunnes kyllästytään laskemaan. Lisäksi olisi ehkä kiva käyttää muitakin mittakaavoja kuin 1:200000. Muutettava matka voitaisiin tarvittaessa antaa jopa ohjelman kutsussa. Voimme lisätä nämä asiat ohjelmaan myöhemmin, kunhan kykymme siihen riittävät. Toteutamme nyt kuitenkin ensin mainitun ohjelman.

8.2 Muuttujat

Ohjelmamme poikkeaa aikaisemmista esimerkeistä siinä, että nyt ohjelman sisällä tarvitaan muuttuvaa tietoa: matka millimetreinä. Tällaiset muuttuvat tiedot talletetaan ohjelmointikielissä muuttujiin. Muuttuja on koneen muistialueesta varattu tarvittavan kokoinen "muistimöhkäle", johon viitataan käytännössä muuttujan nimellä.

Kone viittaa muistipaikkaan muistipaikan osoitteella. Kääntäjäohjelman tehtävä on muuttaa muuttujien nimiä muistipaikkojen osoitteiksi. Kääntäjälle täytyy kuitenkin kertoa aluksi minkä kokoisia 'möhkäleitä' halutaan käyttää. Esimerkiksi kokonaisluku voidaan tallettaa pienempään tilaan kuin reaaliluku. Mikäli haluaisimme varata vaikkapa muuttujan, jonka nimi olisi `matka_mm` kokonaisluvuksi, kirjoittaisimme seuraavan Java-kielisen lauseen (muuttujan esittely):

```
int matka_mm; // yksinkertaisen tarkkuuden kokonaisluku
```

Pascal –kielen osaajille huomautettakoon, että Pascalissahan esittely oli päinvastoin:

```
var matka_mm: integer;
```

Tulos, eli matka kilometreinä voitaisiin laskea muuttujaan `matka_km`. Tämän muuttujan on kuitenkin oltava reaalilukutyyppinen (ks. esimerkkiajo), koska tulos voi sisältää myös desimaaliosan:

```
double matka_km; // kaksinkertaisen tarkkuuden reaaliluku
```

On olemassa myös yksinkertaisen tarkkuuden reaaliluku `float`, mutta emme tarvitse sitä tällä kurssilla. Samoin kokonaisluvusta voidaan tehdä "tosi lyhyt", "lyhyt" tai "kaksi kertaa isompi":

```
int   matka_km;
short sormia;                // max 32767
byte  varpaita;              // max 127
long  valtion_velka_Mmk;     // Tarvitaan ISO arvoalue
```

Muuttujan määritys voisi olla myös

```
volatile static long sadasosia;
```

Tulemme kuitenkin aluksi varsin pitkään toimeen pelkästään seuraavilla perustyypeillä:

```
short    - kokonaisluvut –32 768 –  32 767,  16-bit
int      – kokonaisluvut –2 147 483 648 - 2 147 483 647, 32-bit
double   – reaaliluvut n. 15 desim. -> 1.7e308
char     – kirjaimet 16 bit Unicode
boolean  - true tai false
```

Katso lisää Javan tietotyypeistä linkistä:

```
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html
```

8.2.1 Matkan laskeminen

Ohjelman käyttämä mittakaava kannattaa sijoittaa ehkä vakioksi, tällöin ainakin ohjelman muuttaminen on helpompaa. Samoin vakioksi kannattaa sijoittaa tieto siitä, paljonko yksi km on millimetreinä (1 km = 1000 m, 1 m = 1000 mm). Ohjelmastamme tulee tällöin esimerkiksi seuraavan näköinen:

java-muut\Matka.java - mittakaavamuunnos 1:200000 kartalta

```
import java.io.*;
/**
 * Ohjelmalla lasketaan mittakaavamuunnoksia 1:200000 kartalta
 * @author Vesa Lappalainen
 * @version 1.0 / 05.01.2003
 */
class Matka {
  static final double MITTAKAAVA = 200000.0;
  static final double MM_KM      = 1000.0*1000.0;

  public static void main(String[] args) {
    int    matka_mm;
    double matka_km;

    // Ohjeet
    System.out.println("Lasken 1:" + MITTAKAAVA +
                       " kartalta millimetreinä mitatun matkan");
    System.out.println("kilometreinä luonnossa.");

    // Syöttöpyyntö ja vastauksen lukeminen
    System.out.print("Anna matka millimetreinä>");
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String s = "";
    try {
      s = in.readLine();
    } catch (IOException ex) {
    }
    if ( s == null ) return;
    if ( s.equals("") ) return;
    matka_mm = Integer.parseInt(s);

    // Datan käsittely
    matka_km = matka_mm*MITTAKAAVA/MM_KM;

    // Tulostus
    System.out.println("Matka on luonnossa " + matka_km + " km.");
  }
}
```

Lukija huomatkoon, että muuttujien ja vakioiden nimet on pyritty valitsemaan siten, ettei niitä tarvitse paljoa selitellä. Tästä huolimatta isommissa ohjelmissa on tapana kommentoida muuttujan esittelyn viereen muuttujan käyttötarkoitus. Mekin pyrimme tähän myöhemmin.

Valitettavasti Javan suoraan sanoen alkeiskäyttöön kelvottoman IO:n takia ohjelman kohta "Syöttöpyyntö ja vastauksen lukeminen" venähti toivottoman pitkäksi. No toisaalta näemme kohta sitäkin paremmin yleiskäyttöisten alirutiinien hyödyt.

Tehtävä 8.1 Vakion korvaaminen

Kokeile ottaa vakioiden edestä pois sana `static`. Mitä tällöin tapahtuu ja miksi? Onko `final`-sanan poistamisella sama vaikutus (palauta ensin `static`)?

8.2.2 Muuttujan nimeäminen

Muuttujien nimissä on sallittuja kaikki kirjaimet (myös skandit, itse asiassa kaikki Unicode-kirjaimet) sekä numerot (0–9) sekä alleviivausviiva (_). Muuttujan nimi ei kuitenkaan saa alkaa numerolla. Muuttujia saa esitellä (*declare*) useita samalla kertaa, kunhan muuttujien nimet erotetaan toisistaan pilkulla. Yleinen Java-tapa on että muuttujan nimi alkaa pienellä kirjaimella ja sen jälkeen jokainen muuttujan nimessä oleva alkava sana alkaa isolla kirjaimella (`parasTulos`).

Muuttujan nimi ei myöskään saa olla mikään vakioista (*literal*):

```
true  false  null
```

eikä mikään seuraavista avainsanoista (*keyword*):

abstract	double	int	strictfp **
assert	else	interface	super
boolean	extends	long	switch
break	final	native	synchronized
byte	finally	new	this
case	float	package	throw
catch	for	private	throws
char	goto *	protected	transient
class	if	public	try
const *	implements	return	while
continue	import	short	void
default	instanceof	static	volatile
do			

Tähdellä (*) merkityt sanat on varattu myöhempään käyttöön.

Vaikka muuttujan nimi saakin sisältää skandeja, kannattaa niiden käytöstä pidättäytyä toistaiseksi ainakin luokkien nimissä, koska luokan nimi on samalla tiedoston nimi ja skandit tiedostojen nimissä aiheuttavat edelleen ongelmia.

Javan nimeämiskäytännöistä katso lisää linkistä:

```
http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html
```

Tehtävä 8.2 Avainsanat

Merkitse edelliseen taulukkoon kunkin avainsanan viereen se, missä kohti monistetta ko. sana on selitetty.

Tehtävä 8.3 Muuttujan nimeäminen

Mitkä seuraavista ovat oikeita muuttujan esittelyjä ja mitkä niistä ovat hyviä:

```
int    o;
int    9_kissaa;
int    _9_kissaa;
double pitkä_matka, pitkaMatka;
int    i, j, kissojen_maara, kissojenMäärä;
int    auto, pyora, juna;
```

8.2.3 Muuttujalle sijoittaminen =

Muuttujalle voidaan antaa ohjelman aikana uusia arvoja käyttäen sijoitusoperaattoria = tai ++,−−,+=,−=,*= jne. –operaattoreilla.

Sijoitusmerkin = vasemmalle puolelle tulee muuttujan nimi ja oikealle puolelle mikä tahansa lauseke, joka tuottaa halutun tyyppisen tuloksen (arvon). Lausekkeessa voidaan käyttää mm. operaattoreita `+, −, *, /` ja funktiokutsuja. Lausekkeen suoritusjärjestykseen voidaan vaikuttaa suluilla `(` ja `)`:

```
kenganKoko   = 42;
pi           = 3.14159265358979323846;
// usein käytetään Math-luokan PI vakiota
pi           = Math.PI;
pinta_ala    = leveys * pituus;
ympyranAla   = pi*r*r;
hypotenuusa  = vastainen_kateetti/sin(kulma);
matka_km     = matka_mm*MITTAKAAVA/MM_KM;
```

Seuraava sijoitus on tietenkin mieletön:

```
r*r = 5.0; /* MIELETÖN USEIMMISSA OHJELMOINTIKIELISSA! */
```


Eli sijoituksessa tulee vasemmalla olla sen muistipaikan nimi, johon sijoitetaan ja oikealla arvo joka sijoitetaan.

Huom! Java–kielessä = merkki EI ole yhtäsuuruusmerkki, vaan nimenomaan sijoitusoperaattori. Yhtäsuuruusmerkki on = =.

Tehtävä 8.4 Muuttujien esittely

Esittele edellisissä sijoitus –esimerkeissä tarvittavat muuttujat.

8.2.4 Muuttujan esittely ja alkuarvon sijoittaminen

Muuttujan esittelyn (*declaration*) yhteydessä muuttujalle voidaan antaa myös alkuarvo (alustus, *definition*). Muuttujien alustaminen tietyllä arvolla on tärkeää, koska alustamattoman muuttujan arvo saattaa olla hyvinkin satunnainen. Alustamattoman muuttujan käyttö onkin jälleen eräs tyypillinen ohjelmointivirhe. Java-kääntäjä tosin ilmoittaa virheenä jos muuttujaa yritetään käyttää ennenkuin sille on annettu alkuarvo.

```
int    kengan_koko = 32, takin_koko = 52;
double pi = Math.PI, r = 5.0;
```

8.3 Muuttujan arvon lukeminen päätteeltä

Javassa tosiaan on tehty melkoisen vaikeaksi tietojen lukeminen päätteeltä. Monissa muissa kielissä esimerkiksi kokonaisluvun lukemista varten on huomattavasti yksinkertaisemmat rakenteet tarjolla:

```
scanf("%d",&matka_mm);   /*  C-kieli */
cin >> matka_mm;         // C++ -kieli
readln(matka_mm);        // Pascal-kieli
```

Rehellisyyden nimissä on kyllä sanottava ettei oikeassa elämässä mikään noistakaan ole hyvä käytännön ratkaisu. Jos käyttäjä syöttää muuta kuin kokonaisluvun, on virheestä toipuminen kaikissa esitetyissä kielissä varsin työlästä.

Usein helpoin ratkaisu onkin lukea tieto ensin merkkijonoon ja sitten "kaivaa" merkkijonosta tarvittava informaatio. Tästä saadaan lisäetuna samalla se, että voidaan käsitellä myös muita kuin numeerisia arvoja eikä ohjelmasta tarvitse tehdä sellaista että jokin tietty luku tarkoittaa ohjelman lopettamista:

```
Anna lukuja (-99 lopettaa) >
```


8.3.1 Lukeminen merkkijonoon

Javan IO-systeemi on varsin monimutkainen. Sitä ei olekaan suunniteltu aloittelevaa käyttäjää silmällä pitäen, vaan mahdollisimman laajennettavaksi. Sellaiseksi että samoilla luokilla voitaisiin hoitaa tiedon lukeminen tiedostosta ja verkosta.

```
// Syöttöpyyntö ja vastauksen lukeminen
System.out.print("Anna matka millimetreinä>");
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String s = "";
try {
  s = in.readLine();
} catch (IOException ex) {
}
```

Alkuun tarvitsemme olion, joka pystyy lukemaan kokonaisen rivin ja tunnistaa meidän puolestamme rivin lopun. Tämä saadaan aikaiseksi yhdistämällä `System`-luokan olio `in` lukijaan (`InputStreamReader`) ja yhdistämällä se puskuroituun lukijaan (`BufferedReader`):

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

Sama voitaisiin tehdä useammallakin lauseella:

```
InputStreamReader instream = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(instream);
```

Tässä tapauksessa emme kuitenkaan tarvitse itse käyttää apuluokkaa `instream`, joten tyydymme yhden rivin versioon.

Saatu uusi olio `in` pystyy lukemaan päätteeltä tietoa. Esimerkiksi metodi `readLine` lukee kokonaisen rivin. Eli käyttäjä syöttää merkkejä päätteelle ja painaa Enter. Jos tulee jokin ongelma syöttövirran kanssa olio heittää poikkeuksen `IOException`. Tässä tapauksessa emme välitä poikkeuksista muuta kuin, että se on otettava vastaan (`catch`).

Nyt lohkon

```
String s = "";
try {
  s = in.readLine();
} catch (IOException ex) {
}
```

jälkeen merkkijono-oliossa `s` on joko päätteeltä luettu arvo tai mikäli jokin meni vikaan, niin tyhjä merkkijono. Vielä on mahdollista että syöttövirta katkaistiin kesken kaiken. Windows-konsolilla tämä tapahtuu jos painetaan Ctrl-Z ja *Unix*/*Linux*-konsolilla Ctrl-C. Tällöin olioviite s ei viittaa mihinkään (sen arvo on `null`).

Siksipä tutkimmekin seuraavaksi mistä on kyse ja lopetamme ohjelman ilman sen suurempia mukinoita:

```
    if ( s == null ) return;
    if ( s.equals("") ) return;
```

Tuon voi kirjoittaa myös yhdelle riville, koska Javan `||`-operaattori (tai) suorittaa totuusarvoista lauseketta vain siihen saakka kunnes totuusarvo selviää:

```
    if ( ( s == null ) || ( s.equals("") ) ) return;
```

Huomattakoon että myös muoto

```
    if ( ( s == null ) | ( s.equals("") ) ) return;
```

on syntaktisesti oikein, mutta tarkoittaa hieman eri asiaa. Looginen lopputulos molemmissa on ehdon lausekkeelle sama. Mutta `|` -operaattorilla molemmat lausekkeet suoritetaan aina. Ja tässä tapauksessa tämä olisi virhe jos `s` olisi `null`.

8.3.2 Lukuarvon selvittäminen merkkijonosta

Kaiken edellä mainitun jälkeen meillä on käytössä oliossa `s` käyttäjän syöttämä merkkijono. Seuraava ongelma on saada tämä merkkijono muutettua numeroksi, jolla voidaan jopa jotakin laskeakin. Kokonaisluvun tapauksessa tämä onnistuu käyttämällä luokkaa `Integer` ja pyytämällä tätä selvittämään luvun arvon:

```
    matka_mm = Integer.parseInt(s);
```

Mikäli käyttäjä on kiltisti syöttänyt kokonaisluvun, niin kaikki menee hienosti. Mutta jos käyttäjä antaa merkkijonon, joka on jotakin muuta kuin kokonaisluku, niin silloin `parseInt` heittää poikkeuksen:

```
bash-2.05a$ java Matka
Lasken 1:200000.0 kartalta millimetreinä mitatun matkan
kilometreinä luonnossa.
Anna matka millimetreinä>kolme
Exception in thread "main" java.lang.NumberFormatException: kolme
        at java.lang.Integer.parseInt(Integer.java:414)
        at java.lang.Integer.parseInt(Integer.java:463)
        at Matka.main(Matka.java:32)
bash-2.05a$
```

Jos haluamme tästäkin siististi selvitä ja vielä ystävällisesti huomauttaa käyttäjälle, tarvitsee muunnoksen ympärille laittaa myös poikkeuskäsittely ja vielä koko lukeminen silmukkaan. Kaikkien näiden muutosten jälkeen pelkkä yhden kokonaisluvun lukeminen viekin jo likemmäksi 20 riviä ja "sotkee" muuten yksinkertaisen ohjelmamme rakenteen lähes täysin.

8.3.3 Apumetodit

Tämän takia onkin ilman muuta järkevää eristää lukemiskoodi omaksi metodikseen:

```
/**
 * Kysytään kokonaisluku. Jos annetaan ei-luku, kysytään uudelleen.
 * @param kysymys näytölle tulostettava kysymys
 * @param oletus arvo jota käytetään jos painetaanpelkkä Ret
 * @return käyttäjän kirjoittama kokonaisluku
 */
public static int kysy_int(String kysymys, int oletus)
{
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  while ( true ) {
    System.out.print(kysymys+" >");
    String s = "";
    try {
      s = in.readLine();
    } catch (IOException ex) {
      continue; // jatkaa silmukkaa
    }
    if ( ( s == null ) || ( s.equals("") ) ) return oletus;
    try {
      return Integer.parseInt(s);
    } catch (NumberFormatException ex) {
      System.out.println("Ei numero: " + s);
    }
  }
}
```

Nyt omassa ohjelmassamme voidaan korvata "koko hirveä sotku" vain yhdellä rivillä:

```
matka_mm = kysy_int("Anna matka millimetreinä",0);
```

Lisäsimme aliohjelmaamme vielä kutsuun yhden parametrin: `oletus`. Näin voidaan käyttäjälle antaa mahdollisuus painaa pelkästään Enter ja silti saadaan järkevä vastaus.

Tehtävä 8.5 Oletuksen tulostaminen

Lisää apumetodiin `kysy_int` vielä oletusarvon tulostaminen sulkuihin ennen väkäsen tulostamista. Eli tulostus olisi:

```
Anna matka millimeterinä (0) >
```

8.3.4 Apuluokat

Seuraava kysymys sitten onkin että mihin tuo apumetodi `lue_int` kirjoitetaan? Yksinkertainen vaihtoehto on kirjoittaa se joko ennen tai jälkeen `main`-metodia. Tässä ratkaisussa olisi se huono puoli, että tuo metodi voisi olla käyttökelpoinen vaikka missä ohjelmassa. Siksipä se kannattaa kirjoittaa omaan luokkaansa. Mutta mihin tämä luokka. Yleiskäyttöisyyden nimissä tuo luokka kannattaa kirjoittaa omaan tiedostoonsa.

Kirjoitammekin koodin vaikkapa tiedostoon Syotto.java:

java-muut\Syotto.java – kokonaisluvun lukeminen päätteeltä

```
import java.io.*;

/**
 * Aliohjelmia tietojen lukemiseen päätteeltä
 * @author Vesa Lappalainen
 * @version 1.0/08.01.2003
 */
public class Syotto {
  /**
   * Kysytään kokonaisluku. Jos annetaan ei-luku, kysytään uudelleen.
   * @param kysymys näytölle tulostettava kysymys
   * @param oletus arvo jota käytetään jos painetaanpelkkä Ret
   * @return käyttäjän kirjoittama kokonaisluku
   */
  public static int kysy_int(String kysymys, int oletus)
  {
...
  }

  public static void main(String[] args) {
    int i;
    i = kysy_int("Anna kokonaisluku",12);
    System.out.println("Luku oli: " + i);
  }
}
```

8.3.5 Luokan testaaminen

Olio-ohjelmoinnin - samoin kun minkä tahansa muun ohjelmoinnin - yksi tavoite on modulaarinen testaus. Eli jokainen palanen testataan - jos suinkin vain mahdollista - omana kokonaisuutenaan. Näin lopullinen ohjelma voidaan koostaa toimiviksi todetuista palikoista.

`Syotto`-luokkaan on myös kirjoitettu pääohjelma ja nyt testaus voidaan tehdä ensin pelkälle `Syotto`-luokalle ennen sen liittämistä muuhun ohjelmaan. Komentoriviltä tämä tapahtuisi nyt vaikkapa:

```
bash-2.05a$ javac Syotto.java
bash-2.05a$ java Syotto
Anna kokonaisluku >
Luku oli: 12
bash-2.05a$ java Syotto
Anna kokonaisluku >392
Luku oli: 392
bash-2.05a$ java Syotto
Anna kokonaisluku >kolme
Ei numero: kolme
Anna kokonaisluku >0
Luku oli: 0
bash-2.05a$
```

Vaikka tässä tapauksessa luokka testattiinkin lukemalla tieto päätteeltä, ei missään tapauksessa pidä tätä yleistää. Yleensä paras testiohjelma on sellainen, joka automaattisesti kokeilee testattavaa yksikköä (oliota, metodia) niillä arvoilla joilla sitä tulee kuormittaa. Hyvä testiohjelma sitten kertoo millä arvoilla yksikkö toimi kuten pitikin ja millä ei toiminut. Ihminen testaajana on kaikista testaajista huonoin, koska ihminen väsyy ja muutoksen jälkeen helposti laiskuuksissaan jättää testaamatta niillä arvoilla, jotka jo ennen muutosta oli testattu. Kuitenkin muutos saattaa tuottaa virheitä jo testattuun osaan ja siksi testi pitää aina aloittaa aivan alusta jokaisen muutoksen jälkeen.

8.3.6 Luokan käyttäminen

Nyt kun uusi luokka, tai oikeastaan tässä tapauksessa uusi apumetodi, on huolellisesti testattu, se voidaan ottaa käyttöön. Nyt kun yksinkertaisuuden vuoksi emme vielä käyttäneet paketteja, niin luokka löytyy jos se on samassa hakemistossa kuin sitä käyttävä luokka. Eli ainoa muutos ohjelmakoodiin on kertoa mistä luokasta metodi `kysy_int` löytyy.

```
matka_mm = Syotto.kysy_int("Anna matka millimetreinä",0);
```

Tehtävä 8.6 Muiden tyyppien lukeminen

Tee vastaavasti luokkaan `Syotto` metodit `kysy_double` ja `kysyString`. Tuleeko paljon samanlaista koodia? Kannattaisiko käyttää jotakin hyväksi? Lisää luokan testiohjelmaan testi uusillekin metodeille.

Tehtävä 8.7 Mittakaavan kysyminen

Muuta matka–ohjelmaa siten, että myös mittakaava kysytään käyttäjältä. Mikäli mittakaavaan vastataan pelkkä [RET] pitää mittakaavaksi tulla 1:200000.

8.4 Viitteet

8.4.1 Miksi viitteet?

C–kielessä osoittimet piti opetella heti ohjelmoinnin alussa, jos halusi tehdä minkäänlaisia järkeviä aliohjelmia. *C++*:ssa ongelmaa voidaan kiertää viitemuuttujien (*references*) avulla. Javassa on myös vastaava käsite, eli kaikki Javan olio-muuttujat ovat tosiasiassa viitemuuttujia. Ne ovat kuitenkin tietyssä mielessä perinteisen C:n osoittimen ja C++:n viitteen välimuoto. Javan viitemuuttujan voi laittaa osoittamaan toistakin oliota kesken koodin. C++:n viitemuuttuja osoittaa aina samaan olioon, mihin se luotiin osoittamaan.

Tutkimme seuraavaksi Javan viitemuuttujien käyttäytymistä. Tehdään ohjelma, jossa päällisin puolin näyttäisi olevan kaksi samanlaista merkkijonoa ja kaksi samanlaista kokonaislukuoliota. Merkkijonot ovat Javassa olioita ja merkkijonomuuttujat viitteitä noihin olioihin.

java-muut\Jonotesti.java - merkkijonoviitteet

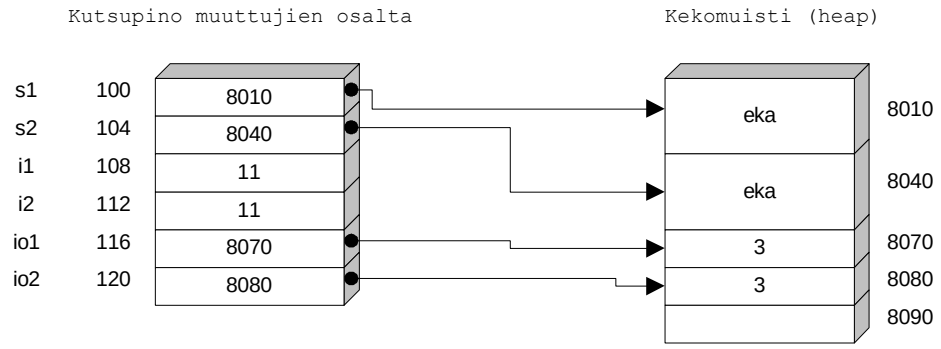
```
/**
 * Tutkitaan olioviitteiden käyttäytymistä
 * @author Vesa Lappalainen
 * @version 1.0, 08.01.20003
 */
class Jonotesti {

  private static void tulosta(boolean b) {
    if ( b ) System.out.println("Samat ovat");
    else System.out.println("Erilaiset ovat");
  }

  public static void main(String[] args) {
    String s1 = "eka";
    String s2 = new String("eka");

    tulosta(s1 == s2);       // Erilaiset ovat
    tulosta(s1.equals(s2));  // Samat ovat

    int i1 = 11;
    int i2 = 10 + 1;

    tulosta(i1 == i2);       // Samat ovat

    Integer io1 = new Integer(3);
    Integer io2 = new Integer(3);

    tulosta(io1 == io2);       // Erilaiset ovat
    tulosta(io1.equals(io2));  // Samat ovat
    tulosta(io1.intValue()== io2.intValue()); // Samat ovat

    s2 = s1;
    tulosta(s1 == s2);         // Samat ovat
  }
}
```

Koodiin on rivien viereen kommentoitu mitä mikäkin rivi tulostaisi.

Javassa on kahden tyyppisiä muuttujia, aikaisemmin lueteltuja perustyyppisiä (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`) muuttujia ja sitten oliomuuttujia. Oliomuuttujat Javassa ovat aina vain viitteitä todellisiin olioihin. Edellisessä esimerkissä muuttujat `s1,s2,io1,io2` ovat olioviitteitä. Silti olioviitteistä puhekielessä käytetään helposti nimitystä olio.

8.4.2 Lokaalit muuttujat

Ohjelman kaikki muuttujat ovat lokaaleja muuttujia. Eli ne on esitelty lokaalisti `main`-metodin sisällä eivätkä "näy" näin ollen `main`-metodin ulkopuolelle. Tällaisille muuttujille varataan tilaa yleensä kutsupinosta. Kun kaikki muuttujat on esitelty ja alustettu, pino voisi hieman yksinkertaistettuna olla näiden lokaalien muuttujien kohdalta suurin piirtein seuraavan näköinen:



Kuva 8.1 Olioviitteet

8.4.3 Dynaaminen muisti

Javassa itse olioiden tila varataan muualta dynaamisen muistinhallinnan hoitamalta alueelta. Usein tätä muistia nimitetään keko- tai kasamuistiksi (*heap*). Kun ohjelmoija pyytää `new`-operaattorilla uuden olion, muistinhallinta etsii sopivan vapaan muistipaikan ja palauttaa viitteen tähän muistipaikkaan. Todellisuudessa olioviitteet ovat hieman monimutkaisempia. Asiasta voi lukea lisää sivuilta:

```
http://java.sun.com/docs/books/vmspec/html/Overview.doc.html
```

Asian ymmärtämiseksi meille kuitenkin riittää yllä piirretty yksinkertaistettu malli.

8.4.4 Viitteiden vertaaminen

Vaikka molemmat viitteet `s1` ja `s2` osoittavat sisällöltään samanlaiseen olioon, palauttaa vertailu

```
( s1 == s2 )     // onko s1 sama kuin s2, => true tai false
```

epätoden arvon. Miksikö? Koska vertailussa verrataan muuttujien arvoja, ei niitä arvoja, joihin muuttujat viittaavat. Esimerkissä on kuviteltu että ensimmäinen "eka"-merkkijono olisi sijoittunut muistissa osoitteeseen `8010` ja toinen osoitteeseen `8040`. Siis itse asiassa kysytäänkin:

```
( 8010 == 8040 )
```

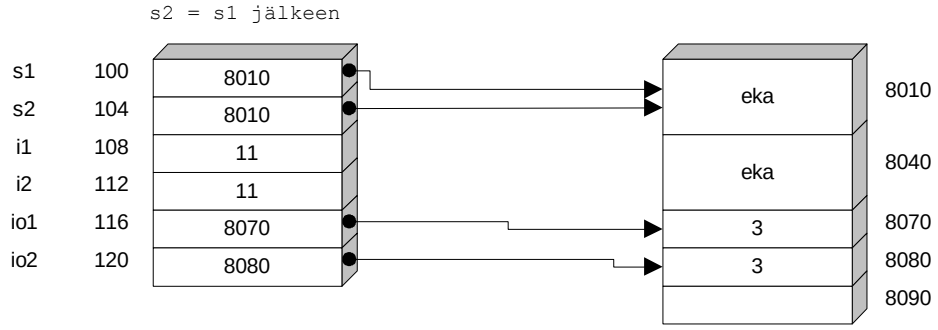
mikä ei ole totta. Javan primitiivityypit sen sijaan sijoittuvat suoraan arvoina pinomuistiin (tai myöhemmin olioiden attribuuttien tapauksessa oliolle varattuun muistialueeseen). Siksi vertailu

```
( i1 == i2 )
```

on totta. Merkkijonoja vastaavasti myös kokonaislukuoliot `io1` ja `io2` käyttäytyvät samalla tavalla. Javassa on kokonaislukuoliot sitä varten, että primitiivityyppejä ei voi tallentaa Javan tietorakenneluokkiin. Piilottamalla primitiivityyppejä "kääreeseen", voidaan näitä "kääreitä" sitten tallentaa tietorakenteisiin.

8.4.5 Viitteeseen sijoittaminen

Jos sijoitetaan "olio" toiseen "olioon", niin tosiasiassa sijoitetaan viitemuuttujien arvoja, eli sijoituksen `s2 = s1` jälkeen molemmat merkkijono-olioviitteet "osoittavat" samaan olioon.



Kuva 8.2 Kaksi viitettä samaan olioon

Sijoituksen jälkeen kuvassa muistipaikkaan 8040 ei osoita (viittaa) enää kukaan ja tuo muistipaikka muuttuu "roskaksi". Kun Javan roskienkeruu (*garbage-collection*, *gc*) seuraavan kerran käynnistyy, "vapautetaan" tällaiset käyttämättömät muistialueet. Tätä automaattista roskienkeruuta on pidetty yhtenä syynä Javan menestykseen. Samalla täytyy kuitenkin varoittaa että muisti on vain yksi resurssi ja Javassa on automatiikka vain muistin hoitamiseksi. Muut resurssit kuten esimerkiksi tiedostot ja tietokannat pitää edelleen hoitaa samalla huolellisuudella kuin muissakin kielissä. Jopa C++:aa huolellisemmin, koska Javassa ei ole C++:n tapaan automaattisia olioita.

Javan viitemuuttuja voidaan siis laittaa "osoittamaan" milloin tahansa toista oliota. Tämä tapahtuu sijoittamalla viitemuuttujaan joko olemassa olevan olion viite

```
s2 = s1;    // laitetaan s2 viittaamaan samaan paikkaan kuin s1
```

tai luomalla uusi olio,

```
String s2 = new String("eka");  // laitetaan s2 viittaamaan uuteen olioon
```

jolloin `new`-operaattorin palauttama viite sijoitetaan. Käytännössä Javan viitteet ovat siis oikeastaan osoittimia. Javan viitteillä ei kuitenkaan voi "edetä" C++:n osoittimien tapaan (esim. `s1++`). Tämä osoitinaritmetiikan puute on toinen Javan hyväksi puoleksi usein mainostettu ominaisuus (tosin ääneen tämä sanotaan "Javassa ei ole osoittimia", lisäksi on tosin totta että Javassa ei todellakaan ole viitteitä tai osoittimia primitiivityyppeihin).

8.4.6 null-viite

Viitemuuttujan arvo voi olla myös `null`. Tämä tarkoittaa ettei oliomuuttuja viittaa mihinkään todelliseen olioon ja tällaista viitemuuttujaa ei saa käyttää ennen kuin siihen on sijoitettu jonkin todellisen olion viite. Yksi Java-ohjelmien yleisimmistä virheistä onkin "`null pointer reference`" kun ohjelmoija ei ole huolellinen viitteiden kanssa.

Hyvin usein pitää siis testata

```
if ( s1 != null ) { // nyt voi käyttää s1 viitettä huoletta
```

8.5 Aliohjelmat (metodit, funktiot)

Eräs ohjelmoinnin tärkeimmistä rakenteista on aliohjelma. C–kielessä kaikkia eri tyyppisiä aliohjelmia nimitetään funktioiksi; joissakin muissa kielissä eri tyyppejä erotetaan eri nimille. Javassa oikeastaan aliohjelmia nimitetään metodeiksi. Kuitenkin kaikkia tähän asti käytettyjä metodeja voidaan suhteellisen hyvällä omallatunnolla nimittää aliohjelmiksi tai C:n tapaan funktioiksi. Aikaisempien esimerkkien metodit nimittäin kaikki ovat olleet `static`-määreellä varustettuja metodeja ja tällaisten metodien virallinen nimi on luokkametodi. Lisäksi kun esimerkkiemme luokkametodit eivät ole koskeneet mihinkään luokan ominaisuuteen, ei metodeilla ole oikeastaan ollut luokan kanssa muuta tekemistä kuin että ne ovat olleet luokan sisällä. Tällöin niitä voi aivan hyvin kutsua aliohjelmiksi. Luokan merkitys on toistaiseksi ollut vain pitää joukkoa metodeja omassa "nimiavaruudessaan". C++:ssa vastaava rakenne hoidetaankin yleensä käyttäen nimiavaruuksia.

Aliohjelmaa käytetään seuraavissa tapauksissa:

1. Haluttu tehtävä on valmiiksi jonkun toisen kirjoittamana aliohjelmana esimerkiksi standardikirjastossa (`y=Math.sin(x)`)
2. Haluttua tehtävää suoritetaan usein liki samanlaisena joko samassa ohjelmassa tai jossain toisessa ohjelmassa.
3. Haluttu tehtävä muodostaa selvän kokonaisuuden, jonka toiminta on ilmaistavissa muutamalla sanalla riittävän selkeästi (= aliohjelman nimi).
4. Haluttua tehtävää ei juuri sillä hetkellä osata tai viitsitä ohjelmoida. Tällöin määritellään millainen aliohjelma tarvitaan ja kirjoitetaan tarvittavaan kohtaan pelkkä aliohjelman kutsu. Itse aliohjelma voidaan aluksi toteuttaa varsin yksinkertaisena ja korjata myöhemmin tekemään sen varsinainen tehtävä.
5. Rakenne saadaan selkeämmän näköiseksi.

8.5.1 Parametriton aliohjelma

Aliohjelma esitellään vastaavasti kuin "pääohjelmakin", eli Javan `main`-metodi. Esimerkiksi satunnaisen matkaajan mittakaavaohjelmassa voisimme kirjoittaa käyttöohjeet omaksi aliohjelmakseen:

java-muut\Matka_a1.java - ohjeet aliohjelmaksi

```
import java.io.*;
/**
 * Ohjelmalla lasketaan mittakaavamuunnoksia 1:200000 kartalta
 * @author Vesa Lappalainen
 * @version 1.0 / 05.01.2003
 */
class Matka_a1 {
  static final double MITTAKAAVA = 200000.0;
  static final double MM_KM      = 1000.0*1000.0;

  /**
   * Tulostaa ohjelman käyttöohjeet
   */
  private static void ohjeet() {
    System.out.println("Lasken 1:" + MITTAKAAVA +
                       " kartalta millimetreinä mitatun matkan");
    System.out.println("kilometreinä luonnossa.");
  }
```

```
  public static void main(String[] args) {
    int    matka_mm;
    double matka_km;

    ohjeet();
    matka_mm = Syotto.kysy_int("Anna matka millimetreinä",0);

    // Datan käsittely
    matka_km = matka_mm*MITTAKAAVA/MM_KM;

    // Tulostus
    System.out.println("Matka on luonnossa " + matka_km + " km.");
  }
}
```

Tämän etu on siinä, että saimme pääohjelman selkeämmän näköiseksi.

8.5.2 Funktiot ja parametrit

Voisimme jatkaa pääohjelman selkeyttämistä. Tavoite voisi olla aluksi vaikkapa kirjoittaa pääohjelma muotoon:

```
    ohjeet();
    matka_mm = Syotto.kysy_int("Anna matka millimetreinä",0);
    matka_km = mittakaava_muunnos(matka_mm);
    tulosta_matka(matka_km);
```

Tällainen pääohjelma tuskin tarvitsisi paljoakaan kommentteja.

Edellä on käytetty kolmen eri tyypin aliohjelmia (funktioita)

1. `ohjeet();` – parametriton aliohjelma
2. `mittakaava_muunnos(matka_mm);` – funktio, joka palauttaa tuloksen nimessään
3. `tulosta_matka(matka_km);` – aliohjelma, jolle vain viedään arvo, mutta mitään arvoa ei palauteta

Valmis ohjelma, jossa myös aliohjelmat on esitelty, näyttäisi seuraavalta (rivien numerointi on myöhemmin esitettävää pöytätestiä varten):

java-muut\Matka_a3.java - erilaisia funktioita

```
1  /**
2   * Ohjelmalla lasketaan mittakaavamuunnoksia 1:200000 kartalta
3   * @author Vesa Lappalainen
4   * @version 1.0 / 05.01.2003
5   */
6  public class Matka_a3 {
7    static final double MITTAKAAVA = 200000.0;
8    static final double MM_KM      = 1000.0*1000.0;
9
10   /**
11    * Tulostaa ohjelman käyttöohjeet
12    */
13   private static void ohjeet() {
14     System.out.println("Lasken 1:" + MITTAKAAVA +
15                        " kartalta millimetreinä mitatun matkan");
16     System.out.println("kilometreinä luonnossa.");
17   }
18
19   /**
20    * Muuttaa mm mittakaavan mukaisesti kilometreiksi
21    * @param matka_mm muutettavat millit
22    * @return mittakavan mukaiset kilometrit
```

```
23          */
24        private static double mittakaava_muunnos(int matka_mm)
25        {
26          return matka_mm*MITTAKAAVA/MM_KM;
27        }
28
29        /**
30         * Tulostaa matkan kilometreinä
31         * @param matka_km tulostettava kilometrimäärä
32         */
33        private static void tulosta_matka(double matka_km)
34        {
35          System.out.println("Matka on luonnossa " + matka_km + " km.");
36        }
37
38        /**
39         * Varsinainen pääohjelma matka kysymiseksi ja laskemiseksi
40         * @param args ei käyttöä
41         */
42        public static void main(String[] args) {
43          int    matka_mm;
44          double matka_km;
45
46          ohjeet();
47          matka_mm = Syotto.kysy_int("Anna matka millimetreinä",0);
48          matka_km = mittakaava_muunnos(matka_mm);
49          tulosta_matka(matka_km);
50        }
51
52      }
```

Edellä olevasta huomataan, että aliohjelmat jotka eivät palauta mitään arvoa nimessään, esitellään `void`–tyyppisiksi.

`mittakaava_muunnos` on reaaliluvun palauttava funktio, joten se esitellään `double` –tyyppiseksi.

Seuraavaksi pöytätestaamme ohjelmamme toiminnan:

	main		**mi..muunnos**		**tulosta**	
lause	matka_mm	matka_km	matka_mm	tulos	matka_km	**tulostus**
46 ohjeet()	??	??				
13-17 System						Lasken 1:200000
47 matka_mm=	352					Anna matka ...
48 matka_km			352			
26 **return**				70.4		
48 matka_km		70.4				
49 tulosta					70.4	
33-36 System						Matka on luo..
50 }						

Emme enää käyneet läpi sitä, mitä `Syotto.kysy_int` tekee, koska se oli testattu erikseen ja sen jälkeen aliohjelma voidaan käsittää "valmiina kieleen kuuluvana käskynä".

Mikäli kukin "omatekoinen" aliohjelmakin olisi testattu erikseen, riittäisi meille pelkkä pääohjelman testi:

	main		
lause	matka_mm	matka_km	**tulostus**
46 ohjeet()	??	??	Lasken 1:200000
47 matka_mm=	352		Anna matka ..
48 matka_km		70.4	
49 tulosta			Matka on luo..
50 }			

Tämä on testaustapa, johon tulisi pyrkiä. Isossa ohjelmassa ei ole enää mitään järkeä testata sitä jokainen aliohjelma kerrallaan. Koodiin liitettyjen aliohjelmien tulee olla testattuja kukin erillisinä ja lopullinen testi on vain viimeisimmän mallin mukainen!

8.5.3 Parametrin nimi kutsussa ja esittelyssä

Huomattakoon, ettei parametrien nimillä aliohjelmien esittelyissä ja kutsuissa ole mitään tekemistä keskenään. Nimi voi olla joko **sama** tai **eri nimi**. Parametrien idea on nimenomaan siinä, että samaa aliohjelmaa voidaan kutsua eri muuttujien tai mahdollisesti vakioiden tai lausekkeiden arvoilla. Esimerkiksi nyt kirjoitettua `tulosta_matka` aliohjelmaa voitaisiin kutsua myös seuraavasti:

java-muut\Matka_a4.java - erilaisia tapoja kutsua funktiota

```
/**
 * Esimerkkejä kutsua aliohjelmaa eri tavoin
 * @author Vesa Lappalainen
 * @version 1.0 / 05.01.2003
 */
public class Matka_a4 {
  /**
   * Tulostaa matkan kilometreinä
   * @param matka_km tulostettava kilometrimäärä
   */
  private static void tulosta_matka(double matka_km)
  {
    System.out.println("Matka on luonnossa " + matka_km + " km.");
  }

  /**
   * Varsinainen pääohjelma matka kysymiseksi ja laskemiseksi
   * @param args ei käyttöä
   */
  public static void main(String[] args) {
    double d = 50.2;
    tulosta_matka(d);          // eri niminen muuttuja
    tulosta_matka(30.7);       // vakio
    tulosta_matka(d+20.8);     // lauseke
    tulosta_matka(2*d-30.0);   // lauseke
  }

}
```

Edellä aliohjelman kutsut voidaan tulkita seuraaviksi sijoituksiksi aliohjelman `tulosta_matka` lokaaliin parametrimuuttujaan `matka_km`:

```
matka_km = d;
matka_km = 30.7;
matka_km = d+20.8;
matka_km = 2*d-30.0
```

Aliohjelma jouduttiin edellä vielä kirjoittamaan uudestaan (käytännössä kopioimaan edellisestä ohjelmasta), mutta myöhemmin opimme miten aliohjelmia voidaan kirjastoida standardikirjastojen tapaan (ks. moduuleihin jako), jolloin kerran kirjoitettua aliohjelmaa ei enää koskaan tarvitse kirjoittaa uudestaan (eikä kopioida).

8.5.4 Nimessään arvon palauttavat funktiot

Funktion arvo palautetaan `return` –lauseessa. Jokaisessa ei-`void` –tyyppiseksi esitellyssä funktiossa tulee olla vähintään yksi `return` –lause. `void`–tyyppisessäkin voi olla `return`–lause. Tarvittaessa `return`–lauseita voi olla useampiakin:

```
public static int suurempi(int a, int b)
{
  if ( a >= b ) return a;
  return b;
}
```

Kun `return` –lause tulee vastaan, lopetetaan **HETI** funktion suoritus. Tällöin myöhemmin olevilla lauseilla ei ole mitään merkitystä. Näin ollen useat `return`–lauseet ovat mielekkäitä vain ehdollisissa rakenteissa. Siis seuraavassa ei olisi mitään mieltä:

```
public static int hopo(int a)
{
  int i;
  return 5;  /* Palauttaa aina 5!!! */
  i = 3 + a;
  return i+2;
}
```

`return`–lausetta ei saa sotkea siihen, että parametrina vietyjä olioita voidaan pyytää muuttamaan sisältöään funktion aikana:

java-muut\FunJaOlio.java - sivuvaikutuksellinen funktio

```
/**
 * Esimerkki funktiosta joka muuttaa myös parametriään
 * @author Vesa Lappalainen
 * @version 1.0 / 05.01.2003
 */
public class FunJaOlio {

  private static int pituus_ja_muuta(StringBuffer s)
  {
    int pit = s.length();
    s.delete(0,pit).append("toka"); // pääohjelman jono muuttuu nyt
    return pit;
  }

  public static void main(String[] args) {
    int i; StringBuffer jono = new StringBuffer("eka");
    i = pituus_ja_muuta(jono);
    System.out.println("i=" + i + ", jono="+jono); // tulostaa: i=3, jono=toka
  }

}
```

Edellä ei kutsusta näe millään tavalla että kutsun jälkeen `jono` on muuttunut. Yhtenä Java-kielen miinuksena voidaankin pitää sitä, että siitä puuttuu C++-kielessä oleva mekanismi suojata oliot muutoksilta aliohjelman suorituksen aikana (`const`).

Näin paljon jääkin ohjelmoijan vastuulle, eli ohjelmoijan pitää nimetä aliohjelmat siten, että niiden nimi jo paljastaa jos jotakin parametria muutetaan ohjelman suorituksen aikana. Ja sitten aliohjelmat on tehtävä huolellisesti, etteivät ne todellakaan muuta kutsuparametrejaan jollei se ole aliohjelmien tarkoitus.

Tehtävä 8.8 Funktio ja osoitin

Mitä pääohjelma `FunJaOlio` tulostaisi jos aliohjelma olisikin ollut:

```
  private static int pituus_ja_muuta(StringBuffer s)
  {
    s.append("toka");
```

```
    return s.length();
  }
```

Tehtävä 8.9 String vs. StringBuffer

Kirjoita edellisestä tehtävästä versio jossa muutat kaikki `StringBuffer => String` ja korvaat `append`-metodin `concat`-metodilla. Mitä tulostuu?

8.5.5 Ketjutettu kutsu

Koska funktio–aliohjelma palauttaa valmiiksi arvon, voitaisiin Matka_a3.java:n pääohjelma kirjoittaa myös muodossa:

```
public static void main(String[] args) {
  double matka_mm;
  ohjeet();
  matka_mm = Syotto.kysy_int("Anna matka  millimetreinä",0);
  tulosta_matka(mittakaava_muunnos(matka_mm));
}
```

Funktioita käytetään silloin, kun aliohjelman tehtävänä on palauttaa vain yksi täsmällinen arvo. `Math`–luokan funktioita ovat:

```
abs, acos, asin, atan, atan2, ceil, cos, exp, floor, IEEEremainder, log, max,
min, pow, random, rint, round, sin, sqrt, tan, toDegrees, toRadians
```

Funktioita käytetään kuten matematiikassa on totuttu:

```
double alpha = 1.32, a = 4, b=3;
double c = Math.sqrt(a*a+b*b) + Math.asin((Math.sin(alpha)+0.2)/2.0);
```

`kysy_matka` ja `kysy_mittakaava` voitaisiin kirjoittaa myös funktioiksi, ja tällöin niitä voitaisiin kutsua esim. seuraavasti:

```
matka_km = kysy_matka()*kysy_mittakaava()/MM_KM;
```

Vaarana olisi kuitenkin se, ettei voida olla aivan varmoja kumpiko funktiosta `kysy_matka` vai `kysy_mittakaava` suoritettaisiin ensin ja tämä saattaisi aiheuttaa joissakin tilanteissa yllätyksiä.

Tämän vuoksi pyrimmekin kirjoittamaan funktioiksi vain sellaiset aliohjelmat, jotka palauttavat täsmälleen yhden arvon ja jotka eivät ota muuta informaatiota ympäristöstä kuin sen mitä niille parametrinä välitetään. Eli tavoitteena on se, että funktioiden kutsuminen lausekkeen osana olisi turvallista. Tämä ei valitettavasti ole aina Javassa mahdollista, koska Javan aliohjelmakutsuista puuttuu muissa kielissä oleva muuttujaparametrin välitys (Pascal: var, C: osoitin *, C++ referenssi &).

Muissa kielissä aliohjelmat kirjoitamme siten, että arvot palautetaan osoitteen avulla. Hyvin yleinen C–tapa on kuitenkin palauttaa tällaisenkin aliohjelman onnistumista kuvaava arvo funktion nimessä (vrt. esim. `scanf` C-kielessä).

Tehtävä 8.10 Math-luokka

Katso SDK:n dokumenteista kunkin `Math`–luokan funktion parametrien määrä ja tyyppi sekä se mitä kukin todella tekee.

Tehtävä 8.11 Funktiot

Kirjoita edellä mainitut `kysy_matka` ja `kysy_mittakaava` nimessään arvon palauttavina funktioina.

Tehtävä 8.12 Ympyrän ala ja pallon tilavuus

Kirjoita funktiot, jotka palauttavat r–säteisen ympyrän pinta–alan ja r–säteisen pallon tilavuuden.

Kirjoita pääohjelma, jossa pinta–ala ja tilavuus –funktiot testataan.

Tehtävä 8.13 Pääohjelma yhtenä funktiokutsuna

Jatka edellä mainittua ketjuttamista siten, että koko pääohjelma on vain yksi lauseke (`ohjeet`–kutsu saa olla oma rivinsä jos haluat). Tosin tämä on C-hakkerismia eikä mikään tavoite helposti luettavalta ohjelmalta. Itse asiassa hyvä kääntäjä tekee automaattisesti tämän kaltaista optimointia (mitä muka voitiin säästää?).

8.5.6 Aliohjelmien testaaminen

Kuten aiemmin todettiin, kannattaa aliohjelmien testaamista varten kirjoittaa hyvin lyhyt testi–pääohjelma.

Esimerkiksi kerhon jäsenrekisterin päämenun tulostamista varten voisimme kirjoittaa aliohjelman nimeltä `paamenu`. Tämä päämenu voitaisiin sitten testata vaikkapa seuraavalla testipääohjelmalla:

java-muut\Paamenu.java - päämenun totetutus ja testi

```
/**
 * Testataan Kerho-ohjelman päämenun tulostamista
 * @author  Vesa Lappalainen
 * @version 1.0, 13.01.2003
 */
public class Paamenu {

  private static void tulosta(String s) {
    System.out.println(s);
  }

  /**
   * Tulostaa Kerho-ohjelman päämenun
   * @param jasenia kerhon jäsenten lukumäärä
   */
  public static void paamenu(int jasenia) {
    tulosta("\n\n\n\n");
    tulosta("Jäsenrekisteri");
    tulosta("===============");
    tulosta("");
    tulosta("Kerhossa on " + jasenia + " jäsentä.");
    tulosta("");
    tulosta("Valitse:");
    tulosta("   ?  = avustus");
    tulosta("   0  = lopetus");
    tulosta("   1  = lisää uusi jäsen");
    tulosta("   2  = etsi jäsenen tiedot");
    tulosta("   3  = tulosteet");
    tulosta("   4  = tietojen korjailu");
    tulosta("   5  = päivitä jäsenmaksuja");
    tulosta("   :");
  }

  public static void main(String[] args) {
```

```
    paamenu(10);
  }

}
```

Huomattakoon, että aliohjelma on saatu kopioiduksi suoraan aikaisemmasta ohjelman suunnitelmasta lisäämällä vain kunkin rivin alkuun `tulosta("` ja loppuun `");`. Tällaiset toimenpiteet voidaan automatisoida tekstinkäsittelyn avulla.

8.5.7 Yksinkertaisen aliohjelman kutsuminen

Valmiin aliohjelman kutsuminen on helppoa: etsitään aliohjelman esittely ja kirjoitetaan kutsu, jossa on vastaavan tyyppiset parametrit vastaavissa paikoissa.

Esimerkiksi funktion `Math.sin` esittely saattaa olla muotoa:

```
sin
public static double sin(double a)
  Returns the trigonometric sine of an angle. Special cases:
    - If the argument is NaN or an infinity, then the result is NaN.
    - If the argument is zero, then the result is a zero with the same
      sign as the  argument.
  A result must be within 1 ulp of the correctly rounded result.
  Results must be semi-monotonic.
Parameters:
   a - an angle, in radians.
Returns:
  the sine of the argument.
```

Funktion tyyppi on `double` ja sille viedään `double` tyyppinen parametri. Funktio ei muuta mitään parametrilistassa esiteltyä parametriaan (mistä tietää?). Siis funktiota ei ole mitään mieltä kutsua muuten kuin sijoittamalla sen palauttama arvo johonkin muuttujaan tai käyttämällä funktiota osana jotakin lauseketta. `x`:ää vastaava parametri voi olla mikä tahansa `double` tyyppisen arvon palauttava lauseke (tietysti mielellään sellainen joka tarkoittaa kulmaa radiaaneissa):

```
double kulman_sini,a,b,x,y;
...
kulman_sini = Math.sin(x);
...
y = Math.sin(x/2) + Math.cos(a/3);
...
```

Funktiota voitaisiin tietysti kutsua myös muodossa:

```
double x = 3.1;

Math.sin(x);
```

mutta kutsussa olisi yhtä vähän järkeä kuin kutsussa

```
double x=3.1;

x + 3.0;
```

tai jopa

```
3.0;
```


Mihin lausekkeiden arvot menisivät? Eivät minnekään! Tosin Javassa kääntäjäkään ei päästä lävitse kahta viimeksi mainittua vaihtoehtoa, eli pelkää vakioita tai muuttujia sisältävää lauseketta, jota ei sijoiteta mihinkään.

Usein aloittelijan näkee yrittävän kutsua muodoissa

```
y = double Math.sin(double a);
y =  Math.sin(double a)
```

mutta näissäkään ei ole järkeä, koska parametrin tyypin esittely kuuluu vain aliohjelman otsikon puoleiseen päähän, ei kutsupäähän.

8.5.8 Aliohjelmat tulostavat harvoin

Yksi yleinen aloittelijan virhe on tehdä paljon aliohjelmia, jotka tulostavat. Pikemminkin pitää toimia päinvastoin, eli aliohjelmien on tehtävä oma työnsä ja annettava sitten tulokset muille tulostettavaksi. Näin samoja aliohjelmia voidaan käyttää myös järjestelmässä, jossa varsinaista konsolitulostusta ei voi tehdä. Tällaisia ovat mm. graafiset käyttöliittymät.

Jos halutaan että aliohjelma kuitenkin tulostaa, niin useimmiten sille kannattaa siinä tapauksessa viedä parametrina tietovirta johon tulostetaan. Palaamme tähän esimerkin kanssa seuraavissa luvuissa. Alla kuitenkin pikainen esimerkki:

java-muut\Tulostustesti.java - tulostus näytölle ja tiedostoon

```
import java.io.*;
/**
 * Testataan tietovirran viemistä parametrina
 * @author  Vesa Lappalainen
 * @version 1.0, 19.01.2003
 */
public class Tulostustesti {

  private static void tulosta(OutputStream os,int h, int m) {
    PrintStream out = new PrintStream(os);
    out.println("" + h + ":" + m);
  }

  public static void main(String[] args) throws FileNotFoundException,
                                                IOException {
    int h=12, m=15;

    // Tulostaminen näyttöön
    tulosta(System.out,h,m);

    // Tulostaminen tiedostoon
    FileOutputStream f = new FileOutputStream("Tulostustesti.txt");
    try {
      tulosta(f,h,m);
    } finally {
      f.close();
    }

    // Tulostaminen tavutietovirtaan, joka voidaan muuttaa sitten merkkijonoksi
    ByteArrayOutputStream bs = new ByteArrayOutputStream();
    tulosta(bs,h,m);
    String s = bs.toString();
    System.out.println(s); // Lisätty, jotta nähdään tulos.
  }

}
```

8.6 Parametrin välitys

8.6.1 Useita parametrejä

Suuressa osassa edellisissä esimerkeissämme meillä on ollut vain 0 tai yksi parametria välitettävänä aliohjelmaan. Käytännössä usein tarvitsemme useampia parametrejä. Esimerkiksi edellisessä `paamenu`–aliohjelmassa pitäisi oikeastaan tulostaa myös kerhon nimi.

Ottakaamme esimerkiksi `mittakaava_muunnos` –funktio. Mikäli ohjelma haluttaisiin muuttaa siten, että myös mittakaavaa olisi mahdollista muuttaa, pitäisi myös mittakaava voida välittää muunnos–aliohjelmalle parametrinä. Kutsussa tämä voisi näyttää esim. tältä:

```
  matka_km = mittakaava_muunnos(32,10000.0);
```

Vastaavasti funktio–esittelyssä täytyisi olla kaksi parametria:

```
private static double mittakaava_muunnos(int matka_mm, double mittakaava)
{
  return matka_mm*mittakaava/MM_KM;
}
```

Kun kutsu suoritetaan, välitetään aliohjelmalle parametrit siinä järjestyksessä, missä ne on esitelty. Voitaisiin siis kuvitella aliohjelmakutsun aiheuttavan sijoitukset aliohjelman parametrimuuttujiin (tosin sijoitusjärjestystä ei taata, eli ei tiedetä kumpi sijoitus suoritetaan ensin):

```
mittakaava = 10000.0;
matka_mm   = 32;
```

Jos kutsu on muotoa

```
  matka_km = mittakaava_muunnos(matka_mm, MITTAKAAVA);
```

kuvitellaan sijoitukset:

```
matka_mm   = matka_mm;   // Pääohjelman muuttuja matka_mm sijoitetaan aliohjelman
                         // vastinpaikassa olevaan muuttujaan
mittakaava = MITTAKAAVA; // Ohjelman vakio toiseeen aliohjelman muuttujaan
```

Siis vaikka kutsussa ja esittelyssä esiintyykin sama nimi, ei nimien samuudella ole muuta tekemistä kuin mahdollisesti se, että turha on väkisinkään keksiä lyhennettyjä huonoja nimiä, jos kerran on hyvä nimi keksitty kuvaamaan jotakin asiaa.

Parametreista osa, ei yhtään tai kaikki voivat olla myös oliota.

Huom! Vaikka kaikilla aliohjelman parametreille olisikin sama tyyppi, täytyy jokaisen parametrin tyyppi mainita silti erikseen:

```
public static double nelion_ala(double korkeus, double leveys)
```

Tehtävä 8.14 Päämenuun kerhon nimi

Lisää Paamenu.java:n aliohjelmaan `paamenu` parametriksi myös kerhon nimi.

Tehtävä 8.15 Toisen asteen yhtälön juuri

Kirjoita funktio `root_1(a,b,c)`, joka palauttaa jomman kumman toisen asteen yhtälön $ax^2+bx+c=0$ juurista (oletetaan tällä kertaa, että `a<>0` ja `D = b`^{2}`–4ac >= 0`. Miksi oletetaan?).

Tehtävä 8.16 Toisen asteen polynomi, `root_1`

Kirjoita funktio `root_1` joka palauttaa toisen asteen polynomin $P(x) = ax^2+bx+c$ arvon (muista viedä parametrinä myös `a,b` ja `c`).

Tehtävä 8.17 root_1 testaus

Kirjoita pääohjelma, jolla voidaan testata `root_1` – aliohjelma (jotenkin myös se, että tulos toteuttaa yhtälön).

8.6.2 Parametrin paikka ratkaisee, ei nimi

Aloitteleva ohjelmoija sotkee yleensä aliohjelmakutsua tehdessään kutsuvan ja kutsuttavan parametrien nimiä keskenään. Parametrien nimillä ei ole Java-kielessä mitään merkitystä. Aliohjelmakutsussa ratkaisee vain parametrien paikka. Kunkin kutsussa

oleva arvo "sijoitetaan" vastinparametrilleen kun aliohjelmaan mennään. Seuraava esimerkki havainnollistaa tätä:

java-muut\Parampaikka.java - parametrin paikka kutsussa ratkaisee

```
/**
 * Esimerkki miten parametrin paikka ratkaisee, ei nimi
 * @author  Vesa Lappalainen
 * @version 1.0, 19.01.2003
 */
public class Parampaikka {

  private static void ali(int a, int b, int c) {
    System.out.println("a=" + a + " b=" + b + " c=" + c);
  }

  public static void main(String[] args) {
    int a=1,b=2,c=3;
    ali(a,b,c);  // Tulostaa: a=1 b=2 c=3
    ali(b,a,c);  // Tulostaa: a=2 b=1 c=3
    ali(c,a,b);  // Tulostaa: a=3 b=1 c=2
    ali(10,c,c); // Tulostaa: a=10 b=3 c=3
  }
}
```

On olemassa myös kieliä, joissa parametrit ovat nimettyjä. Tällainen on tarpeen jos parametreja on niin paljon, ettei niitä kaikkia välitetä joka kutsussa. Esimerkki tällaisesta kielestä on vaikkapa *Microsoft Visual Basic for Application* (*VBA*).

8.6.3 Metodin nimen kuormittaminen

Javassa - samoin kuin monessa muussakin nykykielessä - on mahdollista kuormittaa (*overload*) aliohjelman nimeä. Eli samassa näkyvyysalueessa saa esiintyä samannimisiä aliohjelmia kunhan niiden parametrit eroavat toisistaan määrältään ja/tai tyypiltään.

```
  private static double mittakaava_muunnos(int matka_mm, double mittakaava)
  {
    return matka_mm*mittakaava/MM_KM;
  }

  private static double mittakaava_muunnos(int matka_mm)
  {
    return matka_mm*MITTAKAAVA/MM_KM;
  }

...
  matka_km = mittakaava_muunnos(20);
...
  matka_km = mittakaava_muunnos(32,20000.0);
```

Kääntäjä pystyy kutsussa päättelemään oikean aliohjelman parametrien määrän ja tyypin mukaan.

Tehtävä 8.18 Toisiaan kutsuvat aliohjelmat

Kirjoita yhden parametrin `mittakaava_muunnos` siten, että se kutsuu kahden parametrin `mittakaava_muunnos`ta.

8.6.4 Muuttujien lokaalisuus

Kukin aliohjelma muodostaa oman kokonaisuutensa. Edellä olleissa esimerkeissä aliohjelmat eivät tiedä ulkomaailmasta mitään muuta, kuin sen, mitä niille tuodaan parametreinä kutsun yhteydessä.

Vastaavasti ulkomaailma ei tiedä mitään aliohjelman omista muuttujista. Näitä aliohjelman lokaaleja muuttujia on esim. seuraavassa:

```
  private static int pituus_ja_muuta(StringBuffer s)
  {
    int pit = s.length();
    s.delete(0,pit).append("toka"); // pääohjelman jono muuttuu nyt
    return pit;
  }
```

```
s    – aliohjelman parametrimuuttuja (tässä tapauksessa viite merkkijonoon).
pit  – aliohjelman lokaali apumuuttuja pituuden säilyttämiseksi
```

Yleensäkin Java–kielessä lausesulut { ja } muodostavat lohkon, jonka ulkopuolelle mikään lohkon sisällä määritelty muuttuja tai tyyppimääritys ei näy. Näkyvyysalueesta käytetään englanninkielisessä kirjallisuudessa nimitystä *scope*. Lokaaleilla muuttujilla voi olla vastaava nimi, joka on jo aiemmin esiintynyt jossakin toisessa yhteydessä. Lohkon sisällä käytetään sitä määrittelyä, joka esiintyy lohkossa:

java-muut\Lokaali.java - lokaalien muuttujien näkyvyys

```
/**
 * Testataan Javan muuttujien lokaalisuutta
 * @author  Vesa Lappalainen
 * @version 1.0, 13.01.2003
 */
public class Lokaali {

  static private char ch='A';

  static private void ali() {
    double ch = 4.5;
    System.out.println("Reaaliluku " + ch);
  }

  public static void main(String[] args) {
    System.out.println("Kirjain " + ch);
    {
      int ch = 5;
      System.out.println("Kokonaisluku " + ch);
      ali();
    }
    System.out.println("Kirjain " + ch);
  }

}
```

Ohjelma tulostaa:

```
Kirjain A
Kokonaisluku 5
Reaaliluku 4.5
Kirjain A
```

Saman tunnuksen käyttäminen eri tarkoituksissa on kuitenkin kaikkea muuta kuin hyvää ohjelmointia.

Tehtävä 8.19 Eri nimet

Korjaa edellinen ohjelma siten, että kullakin erityyppisellä muuttujalla on eri nimi.

8.6.5 Parametrinvälitysmekanismi

Ainoa Java–kielen tuntema parametrinvälitysmekanismi on parametrien välittäminen arvoina. Tämä tarkoittaa sitä, että aliohjelma saa käyttöönsä vain (luku)arvoja, ei muuta. Olkoon meillä esimerkiksi ongelmana tehdä aliohjelma, jolle viedään parametreinä tunnit ja minuutit sekä niihin lisättävä minuuttimäärä. Jos ensimmäinen yritys olisi seuraava:

java-muut\Aikalisa.java - yritys lisätä arvoja

☹

```
/**
 * Yritetään lisätä metodissa parametrien arvoja
 * @author  Vesa Lappalainen
 * @version 1.0, 18.01.2003
 */
public class Aikalisa {

  private static void lisaa(int h, int m, int lisa_min) {
    int yht_min = h*60 + m + lisa_min;
    h = yht_min / 60;
    m = yht_min % 60;
  }

  private static void tulosta(int h, int m) {
    System.out.println("" + h + ":" + m);
  }

  public static void main(String[] args) {
    int h=12,m=15;
    tulosta(h,m);
    lisaa(h,m,55);
    tulosta(h,m);
  }

}
```

Tämä ei tietenkään toimisi! Hyvä (C-) - kääntäjä jopa varoittaisi että:

```
Warn :  aikalisa.cpp(8,2):'m' is assigned a value that is never used
Warn :  aikalisa.cpp(7,2):'h' is assigned a value that is never used
```

Mutta miksi ohjelma ei toimisi? Seuraavan selityksen voi ehkä ohittaa ensimmäisellä lukukerralla. Tutkitaanpa tarkemmin mitä aliohjelmakutsussa oikein tapahtuu. Oikaisemme seuraavassa hieman joissakin kohdissa liian tekniikan kiertämiseksi, mutta emme kovin paljoa. Esimerkki on kirjoitettu vastaavasta C++-ohjelmasta. Javassa periaatteessa tapahtuu samalla tavalla. Katsotaanpa ensin miten kääntäjä kääntäisi aliohjelmakutsun (*Borland* C++ 5.1, 32-bittinen käännös, rekisterimuuttujat kielletty jottei optimointi tekisi konekielisestä ohjelmasta liian monimutkaista):

```
lisaa(h,m,55);

muistiosoite assembler         selitys
-------------------------------------------------------------------------
004010F9     push 0x37           pinoon 55
004010FB     push [ebp-0x08]     pinoon m:n arvo
004010FE     push [ebp-0x04]     pinoon h:n arvo
00401101     call lisaa          mennään aliohjelmaan lisää
00401106     add esp,0x0c        poistetaan pinosta 12 tavua (3 x int)
```

Kun saavutaan aliohjelmaan `lisaa`, on pino siis seuraavan näköinen:

```
muistiosoite  sisältö          selitys
-------------------------------------------------------------------------
064FDEC       00401106 <-ESP   paluuosoite kun aliohjelma on suoritettu
064FDF0       0000000C         h:n arvo, eli 12
064FDF4       0000000F         m:n arvo, eli 15
064FDF8       00000037         lisa_min, eli 55
```

Eli aliohjelmaan saavuttaessa aliohjelmalla on käytössään vain arvot 12,15 ja 55. Näitä se käyttää tässä järjestyksessä omien parametriensa arvoina, eli `m,h,lisa_min`.

Esimerkiksi Pascal ja C/C++ -kielissä olisi tarjota tähän sellainen ratkaisu, että aliohjelman parametrit olisivatkin viitteitä (tai osoittimia) kutsuvan ohjelman muuttujiin ja niihin tehty muutos muuttaisi suoraan kutsuvan ohjelman muuttujia. Javassa tämä on mahdollista vain olioille, koska oliot välitettiin viitteinä.

```
C++:    void lisaa(int &h, int &m, int lisamin);           kutsu: lisaa(h,m,55);
Pascal: procedure lisaa(var h,m:integer; lisamin:integer); kutsu: lisaa(h,m,55);
C:      void lisaa(int *h, int *m, int lisamin);           kutsu: lisaa(&h,&m,55)
```

Tehtävä 8.20 Muotoilu?

Kokeilepa laittaa ajaksi esim. 12:05. Mitä tulostuu? Miten vian voisi korjata?

Tehtävä 8.21 Tiedon lukeminen

Kirjoita aliohjelma `lue_kello`, joka kysyy ja lukee arvon kellonajalle, syöttö muodossa 12:15.

8.6.6 Aliohjelmien kirjoittaminen

Uuden aliohjelmien kirjoittaminen kannattaa aina aloittaa aliohjelmakutsun kirjoittamisesta vähintään testiohjelmaan. Näin voidaan suunnitella mitä parametrejä ja missä järjestyksessä aliohjelmalle viedään. Näinhän teimme mittakaava–ohjelmassakin.

8.6.7 Luokkamuuttujat ja suhde lokaaleihin muuttujiin

Muuttujat voidaan esitellä myös luokan kaikissa metodeissa näkyväksi. Mikäli muuttujat esitellään kaikkien ohjelman aliohjelmalausesulkujen ulkopuolella, näkyvät muuttujat koko luokan alueella. Jos muuttujat vielä varustetaan vaikkapa `public` määreellä, niin luokan ulkopuolisetkin luokat voivat niitä käyttää. Tällaista on syytä välttää. Seuraava ohjelma on kaikkea muuta kuin hyvän ohjelmointitavan mukainen, mutta pöytätestaamme sen siitä huolimatta:

java-muut\Alisotku.java - parametrin välitystä

```
/**
 * Mitä ohjelma tulostaa??
 * @author  Vesa Lappalainen
 * @version 1.0, 19.01.2003
 */
public class Alisotku {

  /**
   * Palauttaa merkkijonon kokonaislukuna
   * @param s muutettava merkkijono
   * @return merkkijonosta saatu kokonaisluku
   */
  private static int i(StringBuffer s) {
    return Integer.parseInt(s.toString());
  }

  /**
   * Sijoittaa kokonaisluvun arvon merkkijonoon
   * @param s merkkijono johon tulos sijoitetaan
   * @param i kokonaisluku joka sijoitetaan
   */
  private static void set(StringBuffer s,int i) {
    s.delete(0, s.length()).append(""+i);
  }

/* 01 */ static int a; static StringBuffer b; static int c;
/* 02 */
/* 03 */ private static void ali_1(StringBuffer a, int b)
/* 04 */ {
/* 05 */   int d;
/* 06 */   d  = i(a);
/* 07 */   c  = b + 3;
/* 08 */   b  = d - 1;
/* 09 */   a.append(""+(c - 5));
/* 10 */ }
/* 11 */
/* 11 */ static private void ali_2(StringBuffer a, StringBuffer b)
/* 13 */ {
/* 14 */   int c;
/* 15 */   c  = i(a) + i(b);
/* 16 */   set(a,9 - c);
/* 17 */   set(b,32);
/* 18 */ }
/* 19 */
/* 20 */ public static void main(String[] args) {
/* 21 */   StringBuffer d = new StringBuffer(); b = new StringBuffer();
/* 22 */   a=1; set(b,2); c=3; set(d,4);
/* 23 */   ali_1(d,c);
/* 24 */   ali_2(b,d);
/* 25 */   ali_1(d,3+i(d));
/* 26 */   System.out.println("" + a + " " + b + " " + c + " " + d);
/* 27 */ }
}
```

Käsittelemme (huonosti nimettyjä) metodeja `i` ja `set` "operaattoreina", eli oletamme niiden toiminnan tunnetuksi, eikä pöytätestissä askelleta niihin sisälle.

Pöytätestin tekeminen aloitetaan piirtämällä sarakkeet kutakin isompaa ohjelmassa olevaa kokonaisuutta varten. Esimerkissä näitä ovat

- suoritettava lause
- luokkamuuttujat
- main-metodi
- metodit ali_1 ja ali_2
- keko
- lisäksi kannattaa laskea välitulokset jonnekin auki

Sitten kukin sarake jaetaan vielä osiin siinä olevien muuttujien määrän mukaan. Kekoa varten tarvitaan karkeasti yhtä monta saraketta kuin ohjelmassa on suoritettavia `new`-operaattoreita (tai `String a = "kissa";` tyyppisiä lauseita) .

Lyhyyden vuoksi olemme seuraavassa merkinneet N1 = ensimmäinen `new`:llä luotu olio ja `N2` on toinen. Lisäksi on otettu c-mäinen merkintä &N1, eli viite olioon N1. Merkintä `L.c` tarkoittaa seuraavassa luokan `c` -muuttuja (jos on vaara sekaantua muuhun). Merkintää `:=` on käytetty välilaskutoimituksissa erottamaan sijoitusta = -merkistä. Merkintä `*` muuttujien yläpuolella on muistutuksena sitä, että kyseessä on viitemuuttujat ja niiden käsittely muuttaa aina jotakin muuta muistipaikkaa. Pöytätestissä siis sarakkeet ovat muistipaikkoja ja rivit muistipaikkojen arvo tiettynä ajanhetkenä. Muistipaikka on merkitty harmaalla jos se ei ole voimassa tiettynä ajanhetkenä.

	luokan			main	ali_1			ali_2			keko		apulaskut
		*			*			*	*		SB	SB	
lause	a	b	c	d	a	b	d	a	b	c	N1	N2	
01 int a;	0	null	0										
21 d = new		&N2		&N1							""	""	syntyy tyhjät merkkijonot
22 a=1; b=2	1	o->	3	o->							"4"	"2"	
23 ali_1(d,c					&N1	3							ali_1(&N1,c)
05 int d							?						
06 d = i(a);							4						d:=i(N1)=4
07 c = b+3;			6										L.c:= 3+3 = 6
08 b = d-1;						3							b:= 4-1 = 3
09 a.ap(c-5)					o->						"41"		L.c-5=1; N1:="4"+"1"="41"
24 ali_2(b,d								&N2	&N1				ali_2(&N2,&N1)
14 int c;										?			
15 c=i(a)+i(										43			c:=41+2 = 43
16 set(a,9-c								o->				"-34"	N2:=9-c=-34;
17 set(b,32)									o->		"32"		
25 ali_1(d,3					&N1	35							ali_1(&N1,3+32)
06 d = i(a)							32						d:=i(N1)=32
07 c = b+3;			38										L.c:= 35+3 = 38
12 b = d-1;						5							b:= 32-1 = 31
09 a.ap(c-5					o->								L.c-5=33; N1:="32"+"33"="3233"
26 printl											"3233"		Tulostus: 1 -34 38 3233
													==========

Luokkamuuttujat ovat rinnastettavissa globaaleihin muuttujiin. Samoin kun seuraavassa luvussa päästään käsiksi varsinaiseen olio-ohjelmointiin, niin myös julkiset attribuutit ovat rinnastettavissa globaaleihin muuttujiin. Globaaleiden muuttujien käyttöä tulee ohjelmoinnissa välttää. Tuskin mistään on tullut yhtä paljon ohjelmointivirheitä, kuin vahingossa muutetuista globaaleista muuttujista!

Käytännössä pöytätestiä voidaan monesti korvata hyvällä debuggerilla. Debuggerista valitettavasti ei useinkaan näe suorituksen historiaa. Ennen kun debuggerit eivät olleet niin yleisiä, korvattiin niitä sijoittamalla ohjelmakoodin sekaan muuttujien arvoja tulostavia lauseita. Joissakin tapauksissa tähänkin vielä joudutaan turvautumaan.

Tehtävä 8.22 Muuttujien näkyvyys

Pöytätestaa seuraava ohjelma:

java-muut\Alisotk2.java - parametrin välitystä

```
/**
 * Mitä ohjelma tulostaa??
 * @author  Vesa Lappalainen
 * @version 1.0, 19.01.2003
 */
public class Alisotk2 {
```

```
  private static int i(StringBuffer s) {
    return Integer.parseInt(s.toString());
  }
  private static void set(StringBuffer s,int i) {
    s.delete(0, s.length()).append(""+i);
  }

/* 01 */ private static StringBuffer b; private static int c;
/* 02 */
/* 03 */ private static void s_1(StringBuffer a, int b)
/* 04 */ {
/* 05 */   int d;
/* 06 */   d  = i(a);
/* 07 */   c  = b + 3;
/* 08 */   b  = d - 1;
/* 09 */   set(a,c - 5);
/* 10 */ }
/* 11 */
/* 12 */ private static void a_2(int a, StringBuffer b)
/* 13 */ {
/* 14 */   c  = a + i(b);
/* 15 */   { int c; c = i(b);
/* 16 */   a = 8 * c; }
/* 17 */   set(b,175);
/* 18 */ }
/* 19 */
/* 20 */ public static void main(String[] args) {
/* 21 */   StringBuffer a = new StringBuffer("4"); int d=9;
/* 22 */   System.out.println("" + a + " " + b + " " + c + " " + d);
/* 23 */   b=new StringBuffer("3"); c=2; d=1;
/* 24 */   s_1(b,c);
/* 25 */   a_2(d,a);
/* 26 */   s_1(a,3+d);
/* 27 */   System.out.println("" + a + " " + b + " " + c + " " + d);
/* 28 */ }
}
```

9. Kohti olio-ohjelmointia

Ohjat ottaako oliot?
Luokista luodut ilmentymät
kantaemosta perityt
rajapinnalla rajatut.

Itsestäänkö ilmaantuvat,
sanomatta siunaantuvat?
Viestejä hyö viskoviksi
kaiken koodin korvaajiksi.

Nyt on virhe pienen pieni
ei valta noin suuren suuri.
Taas työ itse tehtäväksi
oliot olkoonkin avuksi.

Luokat luotava lujiksi
vakaan vastuun kantajiksi
ylläpidon ystäviksi
tehtävien taitajiksi.

Oman homman hoitajaksi
tuodun tiedon taattajaksi
sisältö sen suojaamaksi
paljon piiloon pistäväksi.

Perintääkin pohdittava
katsottava koostamista
muodostajaa muotoiltava
rajapintoja raakattava.

Metodeja mietittävä
attribuutteja aateltava
viestejäkin viskeltävä
olioita ohjaillessa.

Mitä tässä luvussa käsitellään?

- yksinkertaiset luokat
- olioiden perusteet
- olioterminologia
- koostaminen
- perintä
- polymorfismi

Syntaksi:

```
luokan esittely:  class Nimi extends Isa implements Rajapinta { // 0-1 x Isa
                                                                 // 0-n x Rajapinta , erot.
                    private yksityinen_attribuutti    // vain itse näkee, 0-n x
                    private yksityinen_metodi         // 0-n x
                    protected suojattu_attribuutti    // perillinen näkee 0-n x
                    protected suojattu_metodi         // 0-n x
                    public julkinen_attribuutti_paha  // kaikki näkee 0-n x WWW
                    public julkinen_metodi            // 0-n x
                    package paketin_attribuutti       // paketissa näkee 0-n x
                    package paketin_metodi            // 0-n x, package on oletus
                  }
attr              kuten muuttuja
attrib.esitt.     tyyppi attr;
metodin esitt.    kuten aliohjelman esittely
viit.olion metod: olio.metodin_nimi(param,param)        // 0-n x param
isäluokan
metodiin viit.    super.metodin_nimi(param,param)
muodostaja        Nimi(param_lista)       // voi olla monta eri param_listoilla
```

Luvun esimerkkikoodit:

http://www.mit.jyu.fi/~vesal/kurssit/ohj2/moniste/esim/olioalk/

Tähän lukuun on kasattu suuri osa olioihin liittyvää asiaa yhden esimerkin valossa. Esimerkin yksinkertaisuuden takia se ei anna joka tilanteessa täyttä hyötyä esitetyistä ominaisuuksista. Lisäksi asiaa voi olla yhdelle lukukerralla liikaa ja esimerkiksi perintä, rajapinnat ja polymorfismi kannattaa ehkä jättää myöhemmälle lukukerralle.

9.1 Miksi olioita tarvitaan

Emme tässä ryhdy pohtimaan kovin syvällisiä siitä, miten olio-ohjelmointiin on päädytty. Todettakoon kuitenkin että olio-ohjelmointi on hyvin luonnollinen jatke rakenteelliselle ohjelmoinnille heti, kun huomataan siirtää käsiteltävä data ja dataa käsittelevä koodi samaan ohjelman osaan. Tämä toimenpide voidaan tehdä tietenkin myös perinteisillä ohjelmointikielilläkin. Puhtaat oliokielet eivät vaan jätä edes muuta mahdollisuutta. Lähestymme asiaa evoluutiomaisesti - niin kuin kehitys on olioihin johtanut. Loput ylilaulut olioista kannattaa lukea jostakin hyvästä kirjasta.

Aloitetaanpa tutkimalla `Aikalisa` esimerkkiämme. Pääohjelmassa esiteltiin muuttuja tunteja varten ja muuttuja minuutteja varten. Aluksi tämä saattaa tuntua hyvin luonnolliselta ja niin se onkin, niin kauan kuin ohjelman koko pysyy pienenä. Entäpä ohjelma jossa tarvitaan paljon kellonaikoja?

olioalk\Aikalis4.java - useita aika "muuttujia"

```
... alku kuten Aikalisa.java
  public static void main(String[] args) {
    int h1=12,m1=15;
    int h2=13,m2=16;
    int h3=14,m3=25;
    tulosta(h1,m1);
    tulosta(h2,m2);
    tulosta(h3,m3);
  }
```

Hyvinhän tuo vielä toimii? Tosin Javassa ei voitu tehdä aliohjelmaa, joka muuttaisi "kellonaikaa". Kiertotienä voisi tallentaa ajan taulukkoon, sillä taulukko välitetään Javassa viitteenä ja silloin taulukon arvoja voisi muuttaa aliohjelmassa. Mutta tämäkin

kiertotie lakkaisi toimimasta jos alkioiden pitäisi olla keskenään eri tyyppiä. Nykyversiossa on lisäksi ongelmana se, että jos joku tulee ja sanoo, että sekunnitkin mukaan! Tulee paljon työtä jos on paljon aikoja.

Tehtävä 9.1 Tulostus

Mitä ohjelma Aikalis4.java tulostaa?

9.2 Hynttyyt yhteen, eli muututaan olioksi

Olio-ohjelmoinnin tärkeimpiä ideoita on kasata tiedot (muuttujat) ja niitä käsitelevät koodit yhteiseksi "paketiksi", olioksi, joka osaa tehdä tiedoille tarvittavat käsittelyt. Lisäksi suojataan tiedot niin, ettei niitä pääse kukaan muu muuttamaan kuin itse olio.

Itse asiassa vanhalla C-kielelläkin pystyi kirjoittamaan "olioita", kirjoittamalla tietuetyypin esittelyn ja sitä käyttävät aliohjelmat yhdeksi aliohjelmakirjastoksi. Näin data ja sitä käsittelevät aliohjelmat on kapseloitu yhdeksi paketiksi.

9.2.1 Terminologiaa

Nyt astuu kuvan mukaan olio-ohjelmoijat ja he nimittävät sitten näin syntyneitä aliohjelmia **metodeiksi** (*method*), tai *C++*-kirjallisuudessa jäsenfunktioiksi (*member function*). Olion alkioita, kenttiä nimitetään sitten **attribuuteiksi**.

Itse "kokoelma" saakin nimen **luokka** (*class*) ja luokkaa vastaava muuttuja - luokan ilmentymä - on sitten se kuuluisa **olio** (*object*)

9.2.2 Ensimmäinen olio-esimerkki

Muutetaanpa Aikalisa kunnon luokaksi ja olioksi:

olioalk\Aika.java - kunnon olioksi

```
/**
 * Ensimmäinen kunnon olioesimerkki
 * @author  Vesa Lappalainen
 * @version 1.0, 01.02.2003
 */
public class Aika {

  private int h=0, m=0;

  /**
   * Alustaa ajan
   * @param h tunnit
   * @param m minuutit
   */
  public Aika(int h,int m) { // Muodostaja
    this.h = h;
    this.m = m;
  }

  /**
   * Tulostaa ajan muodossa 15:05
   */
  public void tulosta() {
    System.out.println("" + h + ":" + (m<10?"0":"")+m);
  }
```

```
  /**
   * Lisää aikaan valitun minuuttimäärän
   * @param lisa_min lisättävä minuuttimäärä
   */
  public void lisaa(int lisa_min)  {
    int yht_min = h * 60 + m + lisa_min;
    h = yht_min / 60;
    m = yht_min % 60;
  }

  public static void main(String[] args) {
    Aika a1 = new Aika(12,15);
    Aika a2 = new Aika(13,16);
    Aika a3 = new Aika(14,25);
    a1.lisaa(55);    a1.tulosta();
    a2.lisaa(27);    a2.tulosta();
    a3.lisaa(39);    a3.tulosta();
  }

}
```

Siinäpä se! Ovatko muutokset edelliseen nähden suuria? Siis iso osa koko olio-ohjelmoinnista (ja tietotekniikasta muutenkin) on markkinahenkilöiden huuhaata ja yleistä hysteriaa "kaiken ratkaisevan" teknologian ympärillä. No, tosin olio-ohjelmoinnissa on puolia, joita emme vielä ole nähneetkään, joiden ansiosta olio-ohjelmointia voidaan pitää uutena ohjelmointia ja ylläpitoa helpottavana teknologiana. Näitä ovat mm. perintä ja polymorfismi (monimuotoisuus), joihin emme valitettavasti tällä kurssilla ehdi perehtyä kovinkaan syvällisesti.

No takaisin esimerkkiimme. Uutta on lähinnä se, että metodien (no sanotaan tästä lähtien funktioita metodeiksi) parametrilistat ovat lyhentyneet. Itse olion tietoja ei tarvitse enää viedä parametrina, koska metodit ovat luokan sisäisiä ja tällöin luokkaa edustava olio kyllä tuntee itse itsensä.

```
...
  public void lisaa(int lisa_min)  {
    int yht_min = h * 60 + m + lisa_min;
    h = yht_min / 60;
    m = yht_min % 60;
  }
...
```

Metodia kutsutaan ilmoittamalla olion nimi ja metodi, jota kutsutaan

```
a1.lisaa(55);  a1.tulosta();
```

Tällekin on keksitty oma nimi: välitetään oliolle viesti "tulosta" (*message passing).* Tässä kuitenkin jatkossa voi vielä lipsahtaa ja vahingossa sanomme kuitenkin, että kutsutaan metodia `tulosta`, vaikka ehkä pitäisi puhua viestin välittämisestä.

9.2.3 Taas terminologiaa

Kerrataanpa vielä termit edellisen esimerkin avulla:

Vinkki

Älä hämäänny termeistä

```
                oliotermi                                        perinteinen termi
-------------------------------------------------------------------------------
Aika          - aika-luokka                                      tietuetyyppi
h,m           - aika-luokan attribuutteja                        tietueen alkio
lisaa,tulosta - aika-luokan metodeja                             funktio,aliohj.
a1,a2,a3      - olioita, jotka ovat aika-luokan ilmentymiä        muuttuja
a1.lisaa(55)  - viesti olioille a1: lisää 55 minuuttia           aliohjelman kutsu
```

9.2.4 Luokka (*class*) ja olio (*object*)

Luokka on tavallaan "piparkakkumuotti" kaikille samankaltaisille "olioille". Luokalla ei sinänsä tee mitään (ellei siinä ole `static`-aliohjelmia), ellei siitä luo luokkaa edustavaa oliota.

```
    Aika a1 = new Aika(12,15);
```

Javan "olio-muuttujathan" eivät olleet mitään muuta kuin pelkkiä viitteitä keossa sijaitseviin varsinaisiin olioihin. `new` -operaattori luo kekoon uuden olion ja palauttaa viitteen tähän olioon.

Pelkkä olion luominen ilman viitteen sijoittamista mihinkään on useimmiten hyödytöntä

```
   new Aika(12,15);  // Tähän olioon ei päästä käsiksi
```

Kerran luodun olion viite voidaan luonnollisesti sijoittaa toiseen viitteeseen:

```
a2 = a1;  // molemmat viitteet viittaavat samaan olioon.
```

Kun olioon ei ole enää yhtään viitettä, muuttuu olio Javassa roskaksi ja muistinsiivous (roskienkeruu, *garbage collection*, *gc*) vapauttaa ajallaan olion viemän muistitilan.

```
Aika a1 = new Aika(12,15);
...
a1 = null;    // a1 ei viittaa enää olioon => olio muuttuu roskaksi

tai

{ // lohkon alku, jonka sisällä viite esitelty
  Aika a1 = new Aika(12,15);
  ...
}  // Viite a1 lakkaa olemasta => olio muuttuu roskaksi
```

9.2.5 Suojaustasot ja kapselointi

Luokan attribuuteille ja metodeille on suojaustasot, jotka oletuksena ovat pakettikohtaisia, eli metodeja voi kutsua kuka tahansa samaan pakettiin kirjoitetun luokan metodi. Erityisesti kuka tahansa samassa paketissa oleva metodi voi muuttaa attribuuttien arvoja ilman että olio tätä itse huomaa.

	Kuka voi käyttää metodia/attribuuttia			
Suojaus	kaikki	aliluokan metodit	paketin metodit	luokan metodit
Private				x
Package			x	x
Protected		x	x	x
Public	x	x	x	x

Kuva 9.1 Suojaustasot

Kirjoitamme ensin testiluokan:

olioalk\Aikatesti.java - testiluokka Aika-luokalle

```
/**
 * Testiohjelma Aika-luokalle
 * @author Vesa Lappalainen
 * @version 1.0, 01.02.2003
 *
*/
public class Aikatesti {

  public static void main(String[] args) {
    Aika a1 = new Aika(12,15);
    Aika a2 = new Aika(13,16);
    Aika a3 = new Aika(14,25);
    a1.lisaa(55);    a1.tulosta();
    a2.lisaa(27);    a2.tulosta();
    a3.lisaa(39);    a3.tulosta();
  }
}
```

Jos esimerkkimme metodi `lisaa` esiteltäisiin:

```
  private void lisaa(int lisa_min)  {
```

niin testiohjelma lakkaisi toimimasta, koska esimerkiksi pääohjelman kutsu

```
a1.lisaa(55)
```

tulisi laittomaksi luokan jäsenen `lisaa` ollessa yksityinen (`private`).

Erityisen tärkeää on kuitenkin että ei voida kirjoittaa testiohjelmassa

```
a1.h = 28;  // private-attribuuttiin ei saa viitata
```

Käytännössä attribuutit kannattaa lähes poikkeuksetta kirjoittaa yksityisiksi. Kaikista pahinta mitä olio-ohjelmoija voi tehdä on kirjoittaa julkisia attribuutteja.

Nyt vasta alkaakin olio-ohjelmoinnin hienoudet! Aloittelijasta saattaa tuntua että mitä turhaa tehdään asioista monimutkaisempaa kun se onkaan! Väärinkäytetyt ja virheelliset arvot muuttujilla on ollut ohjelmoinnin kiusa alusta alkaen. Nyt meillä on mahdollisuus päästä niistä eroon kapseloinnin (jotkut sanovat kotelointi, *encapsulation*) ansiosta. Eli kaikki arvojen muutokset (eli olio tapauksessa olion tilojen muutokset) voidaan suorittaa kontrolloidusta, vain olion itse siihen suostuessa. Mutta miten sitten alustuksen tapauksessa?

9.2.6 Muodostajat (*constructor*)

*Java*n olioilla on yksi erityinen metodi: **muodostaja** (konstruktori, rakentaja, *constructor*), jota kutsutaan muuttujan syntyessä. Muodostajan tehtävä on alustaa olion tila ja luoda mahdollisesti tarvittavat dynaamiset muistialueet. Näin voidaan järjestää se, että olion tila on aina tunnettu olion syntyessä.

Joissakin oliokielissä konstruktori ilmoitetaan omalla avainsanallaan. *Java*ssa muodostaja on metodi, jolla on sama nimi kuin luokalla. Muodostajia voi olla useitakin. Muodostaja on aina tyypitön, siis ei edes `void`-tyyppiä.

olioalk\Aika.java - muodastaja alustamaan tiedot

```
  public Aika(int h,int m) { // Muodostaja
    this.h = h;
    this.m = m;
  }
```

Esimerkissämme muodostaja on esitelty 2-parametriseksi ja sitä ”kutsutaan” olion luonnin yhteydessä:

```
Aika a1 = new Aika(12,15);
```

9.2.7 Oletusmuodostaja (*default constructor*)

Nyt ei kuitenkaan voida esitellä oliota ilman alkuarvoa

```
Aika aika = new Aika();
```

Kääntäjä antaisi esimerkiksi virheilmoituksen:

```
"Aikatesti.java": Error #: 300 : constructor Aika() not found in class Aika at
line 16, column 21
```

Parametritonta muodostajaa sanotaan **oletusmuodostajaksi** (*default constructor*). Sellainen on luokalla aina ilman muuta, jos luokalle ei ole esitelty yhtään muodostajaa. Jos luokalle esitellään muodostaja, ei oletusmuodostaja enää tulekaan automaattisesti.

Meidän pitäisi päättää nyt paljonko kellomme on, jos sitä ei erikseen ilmoiteta. Olkoon kello vaikka 0:0, eli keskiyö. Esittelemme oletusmuodostajan

olioalk\Aika2.java - lisätään oletusmuodostaja

```
...
public class Aika2 {

  private int h=0, m=0;

  public Aika2() { // Oletusmuodostaja
    h = 0; m = 0;
  }

  public Aika2(int h,int m) { // Muodostaja
    this.h = h;
    this.m = m;
  }

...
```

```
  public static void main(String[] args) {
    Aika2 a1 = new Aika2(12,15);
...
    a3.lisaa(39);    a3.tulosta();
    Aika2 aika = new Aika2();
    aika.tulosta();
  }

}
```

Tässä tapauksessa oletusmuodostajaksi olisi kelvannut myös tyhjä lohko. Miksi?

```
  public Aika2() {  }
```

9.2.8 Sisäinen tilan valvonta

Emme edelleenkään ole ottaneet kantaa siihen, mitä tapahtuu, jos joku yrittää alustaa oliomme mielettömillä arvoilla, esimerkiksi:

```
Aika a1 = new Aika(42,175);
```

Toisaalta miten joku voisi muuttaa ajan arvoa muutenkin kuin `lisaa`-metodilla? Teemmekin aluksi metodin `aseta`, jota kutsuttaisiin

```
a1.aseta(12,15); a2.aseta(16,23);
```

Nyt pitää kuitenkin päättää mitä tarkoittaa laiton asetus! Jos sovimme että minuuteissa yli 59 arvot ovat aina 59 ja alle 0:n arvot ovat aina 0, voisi `aseta`-metodi olla kirjoitettu seuraavasti:

```
  public void aseta(int ih,int im) {
    h = ih; m = im;
    if ( m > 59 ) m = 59;
    if ( m <  0 ) m =  0;
  }
```

Jos taas haluaisimme, että ylimääräinen osuus siirtyisi tunteihin, voitaisiinkin tämä tehdä "kierosti" `lisaa`−metodin avulla

olioalk\Aika4.java - sisäinen tilan valvonta asetuksessa

```
...
public class Aika4 {

  private int h=0, m=0;

  /**
   * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
   * @param h asetettavat tunnit
   * @param m asetettavat minuutit
   */
  public void aseta(int h,int m) {
    this.h = h; this.m = m; lisaa(0);
  }

  public Aika4() { // Oletusmuodostaja
    aseta(0,0);
  }
```

```
  /**
   * Alustaa ajan niin että minuutit ovat aina 0
   * @param h tunnit
   */
  public Aika4(int h) {
    aseta(h,0);
  }

  public Aika4(int h,int m) { // Muodostaja
    aseta(h,m);
  }

  public void tulosta() {...
  public void lisaa(int lisa_min)  { ...

  public static void main(String[] args) {
    Aika4 a1 = new Aika4();
    Aika4 a2 = new Aika4(13);
    Aika4 a3 = new Aika4(14,25);

    a1.tulosta();  a2.tulosta();  a3.tulosta();
    a1.aseta(12,15); a2.aseta(16,-15);
    a1.tulosta(); a2.tulosta();
  }

}
```

Huomattakoon, että samalla kun tehdään `aseta`-metodi, kannattaa muodostajassakin kutsua sitä. Näin olion tilaa muutetaan vain muutamassa metodissa, jotka voidaan ohjelmoida niin huolella, ettei mitään yllätyksiä pääse koskaan tapahtumaan. Tämä rupeaa jo muistuttamaan olio-ohjelmointia!

Tehtävä 9.2 Negatiivinen minuuttiasetus

Mitä ohjelma Aika4.java tulostaisi? Miksi ohjelma toimisi halutulla tavalla?

Tehtävä 9.3 Tuntien tarkistus

Ohjelmoi myös tunneille tarkistus, missä pidetään huoli siitä, että tunnit ovat aina välillä 0-23.

Tehtävä 9.4 Päivämääräolio

Esittele luokka, jolla kuvataan päivämäärä. Kirjoita ainakin sopiva muodostaja ja metodi `tulosta`, joka tulostaa päivämäärän.

9.2.9 Metodien kuormittaminen (lisämäärittely, *overloading*)

Edellisessä esimerkissä oli kolme samannimistä metodia `Aika4`. Kussakin oli eri määrä parametrejä. Tätä sanotaan metodin kuormittamiseksi, eli mahdollisuudeksi määritellä lisää merkityksiä (eng. *overloading*) metodin nimelle. Varsinainen kutsuttava metodi tunnistetaan nimen ja parametrilistassa olevien lausekkeiden avulla. Metodin nimi koostuukin tavallaan nimen ja parametrilistan yhdisteestä. Siten jos olisi vaikka kutsut

```
a1.tulosta();             // Tulostaisi  14:15
a1.tulosta("Kello on ");  // Tulostaisi  Kello on 14:15
```

niin kumpikin `tulosta`-kutsu kutsuu eri metodia. Metodien kuormitus onkin varsin mukava lisä ohjelmointiin, se ei kuitenkaan ole varsinaisia olio-ohjelmoinnin piirteitä.

Huomattakoon että kuormitetuissa metodeissa ero on oltava parametreissa, pelkkä ero metodin paluuarvossa ei riitä erottelemaan mitä metodia tarkoitetaan.

Tehtävä 9.5 Mitäs me tehtiin kun ei ollut kuormitusta?

Miten asiat on hoidettava C-kielessä, kun siellä funktioiden nimien kuormitus ei ole mahdollista, vaan kunkin funktion nimen tulee olla yksikäsitteinen.

Tehtävä 9.6 Lisäys yhdellä

Tee vielä uusi `lisaa` metodi, jota voidaan kutsua `a1.lisaa();` jolloin metodi lisää aikaa yhdellä minuutilla.

Tehtävä 9.7 Vain tuntien asettaminen

Kirjoita vielä yksi `lisaa`-metodi, jolla voidaan asettaa pelkät tunnit.

9.2.10 `this`-osoitin

Jos verrataan aliohjelmaa

olioalk\Aika5.java - aliohjelma vastaan metodi

```
  public static void lisaa(Aika5 aika,int lisa_min) {
    int yht_min = aika.h * 60 + aika.m + lisa_min;
    aika.h = yht_min / 60;
    aika.m = yht_min % 60;
  }

...
  lisaa(a1,55);
```

ja metodia

```
  public void lisaa(int lisa_min)  {
    int yht_min = h * 60 + m + lisa_min;
    h = yht_min / 60;
    m = yht_min % 60;
  }
...
  a1.lisaa(55);
```

niin helposti näyttää, että ensin mainitussa funktiossa on enemmän parametrejä. Tosiasiassa kummassakin niitä on täsmälleen sama määrä. Nimittäin jokaisen metodin ensimmäisenä näkymättömänä parametrinä tulee aina itse luokan osoite, `this`. Voitaisiinkin kuvitella, että metodi onkin toteutettu:

```
"public void lisaa(Aika5 this, int lisa_min)  {"   // Näin EI SAA KIRJOITTAA!!!
{
  int yht_min = this.h * 60 + this.m + lisa_min;
  this.h = yht_min / 60;
  this.m = yht_min % 60;
}
...
"a1.lisaa(a1,55)";
```

Oikeasti `this` *-viitettä ei saa esitellä*, vaan se on ilman muuta mukana parametreissa sekä esittelyssä että kutsussa. Mutta voimme todellakin kirjoittaa:

```
public void lisaa(int lisa_min)  {
{
  int yht_min = this.h * 60 + this.m + lisa_min;
  this.h = yht_min / 60;
  this.m = yht_min % 60;
}
```

Jonkun mielestä voi jopa olla selvempi käyttää `this`-viitettä luokan attribuutteihin viitattaessa, näinhän korostuu, että käsitellään nimenomaan tämän luokan attribuuttia `h`, eikä mitään muuta muuttujaa `h`. Joskus `this`-osoitinta tarvitaan välttämättä palautettaessa oliotyyppisellä metodilla olion koko tila (esim. viite olioon). Lisäksi joissakin kielissä `this`-osoittimen vastinetta (usein `self`) on aina käytettävä.

Usein `this`-osoitinta käytetään, jos ei haluta antaa metodin parametrilistan muuttujille eri nimiä kuin vastaavilla attribuuteilla:

```
public void aseta(int h,int m) {
  this.h = h; this.m = m; lisaa(0);
}
```

Vaihtoehtonahan olisi esimerkiksi

```
public void aseta(int ih,int im) {  // i = initialize
  h = ih; m = im; lisaa(0);
}
```

9.3 Perintä

9.3.1 Luokan ominaisuuksien laajentaminen

Pidimme jo aikaisemmin toiveena sitä, että voisimme laajentaa luokkaamme käsittelemään myös sekunteja. Miksi emme tehneet tätä heti? No tietysti olisi heti pitänyt älytä laittaa mukaan myös sekunnit, mutta tosielämässäkin käy usein näin, eli hyvästäkin suunnittelusta huolimatta toteutuksen loppuvaiheessa tulee vastaan tilanteita, jossa alkuperäiset luokat todetaan riittämättömiksi.

Tämän laajennuksen tekemiseen on olio-ohjelmoinnissa kolme mahdollisuutta: Joko muuttaa alkuperäistä luokkaa, periä alkuperäisestä luokasta laajempi versio tai tehdä uusi luokka, jossa on alkuperäinen luokka yhtenä attribuuttina.

Tutustumme seuraavassa kaikkiin kolmeen eri mahdollisuuteen.

9.3.2 Alkuperäisen luokan muuttaminen

Läheskään aina ei voi täysin välttää sitäkään, etteikö alkuperäistä luokkaa joutuisi muuttamaan. Jos näin joudutaan tekemään, pitäisi tämä pystyä tekemään siten, että jo kirjoitettu luokkaa käyttävä koodi säilyisi täysin muuttumattomana (tai ainakin voitaisiin päivittää minimaalisilla muutoksilla) ja vasta uudessa koodissa käytettäisiin hyväksi luokan uusia ominaisuuksia.

Jos luokka on saatu joltakin kolmannelta osapuolelta, ei luokan päivittäminen edes ole mahdollista, vaan silloin täytyy turvautua muihin (parempiin) tapoihin.

Tehtävä 9.8 Luokan muuttaminen

Muuta ohjelmaa Aika4.java siten, että ajassa on mukana myös sekunnit. Kuitenkin niin, että alkuperäinen testiohjelma säilyy sellaisenaan toimivana. Voit lisätä testiohjelmaan uusia rivejä sekuntien testaamiseksi.

Tehtävä 9.9 Sekuntien tulostus aina tai oletuksena

Muuta edellistä ohjelmaa siten, että sekunnit tulostetaan aina.

Muuta edellistä ohjelmaa siten, että sekunnit tulostetaan oletuksena jos ne on != 0.

9.3.3 Koostaminen

Seuraava mahdollisuus olisi uuden luokan **koostaminen** (*aggregation*) vanhasta aikaluokasta ja sekunneista. Tämä mahdollisuus meillä on aina käytössä vaikkei alkuperäistä lähdekoodia olisikaan käytössä. Tätä vaihtoehtoa pitää aina vakavasti harkita.

Valitettavasti emme voi aivan täysin onnistua. Nimittäin alkuperäisen luokan `tulosta` oli niin tyhmästi toteutettu, että se tulosti aina rivinvaihdon. Tämmöistä hölmöyttä ei pitäisi mennä koskaan tekemään ja siksipä alkuperäinen luokka pitää joka tapauksessa palauttaa valmistajalle remonttiin. No luokan valmistaja muutti `tulosta`-metodia siten, että se oletuksena tulostaa rivinvaihdon (merk. `lf` = *line feed*), mutta pyydettäessä jättää sen tekemättä. Näin vanha koodi voidaan käyttää muuttamattomana.

Palaamme tulostusongelmaan myöhemmin ja keksimme silloin paremman ratkaisun, jota olisi pitänyt käyttää jo alunperin.

olioalk\Aika5.java - rivinvaihto ehdolliseksi

```
  /**
   * Tulostaa ajan muodossa 15:05
   * @param lf tulostetaanko rivinvaihto vai ei
   */
  public void tulosta(boolean lf) {
    System.out.print("" + h + ":" + (m<10?"0":"")+m);
    if ( lf ) System.out.println();
  }

  /**
   * Tulostaa ajan muodossa 15:05 sekä aina rivinvaihdon
   */
  public void tulosta() {
    tulosta(true);
  }
```

Nyt voimme kirjoittaa uuden luokan, joka koostetaan luokasta `Aika5` ja sekunneista:

olioalk\AikaSek7.java - laajentaminen koostamalla

```
/**
 * Luokan laajentaminen koostamalla
 * @author  Vesa Lappalainen
 * @version 1.0, 01.02.2003
 */
public class AikaSek7 {

  private Aika5 hm = new Aika5();
  private int s;

  /**
   * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
   * @param h asetettavat tunnit
   * @param m asetettavat minuutit
   * @param s asetettavat sekunnit
   */
  public void aseta(int h,int m, int s) {
    hm.aseta(h,m); this.s = s; lisaa(0,0);
  }

  public void aseta(int h,int m) { aseta(h,m,0); }
  public void aseta(int h)       { aseta(h,0);   }
```

```
  public AikaSek7() { }

  /**
   * Alustaa ajan
   * @param h tunnit
   * @param m minuutit
   * @param s sekunnit
   */
  public AikaSek7(int h,int m, int s) { // Muodostaja
    aseta(h,m,s);
  }

  public AikaSek7(int h,int m) { aseta(h,m,0); }
  public AikaSek7(int h)       { aseta(h,0,0); }

  /**
   * Tulostaa ajan muodossa 15:05
   * @param lf tulostetaanko rivinvaihto vai ei
   */
  public void tulosta(boolean lf) {
    hm.tulosta(false);
    System.out.print(":" + (s<10?"0":"")+s);
    if ( lf ) System.out.println();
  }

  /**
   * Tulostaa ajan muodossa 15:05 sekä aina rivinvaihdon
   */
  public void tulosta() { tulosta(true);  }

  /**
   * Lisää aikaan valitun minuuttimäärän
   * @param lisa_min lisättävä minuuttimäärä
   * @param lisa_sek lisättävä sekunttimäärä
   */
  public void lisaa(int lisa_min,int lisa_sek)  {
    s += lisa_sek;
    hm.lisaa(lisa_min+s/60);
    s %= 60;
  }

  public void lisaa(int lisa_min)  { lisaa(lisa_min,0); }

  public static void main(String[] args) {
    Aika5 a1 = new Aika5();
    Aika5 a2 = new Aika5(13);
    Aika5 a3 = new Aika5(14,175);

    a1.tulosta();  a2.tulosta();  a3.tulosta();
    a1.aseta(12,15); a2.aseta(16,-15);
    a1.tulosta(); a2.tulosta();

    AikaSek7 a4 = new AikaSek7(14,55,45); a4.tulosta();
    a4.lisaa(3,30); a4.tulosta();
  }
}
```

Luokassa on niin vähän ominaisuuksia, että uudessa luokassamme olemme joutunee itse asiassa tekemään kaiken uudelleen ja on kyseenalaista olemmeko hyötyneet vanhasta luokasta lainkaan. Tämä on onneksi lyhyen esimerkkimme vika, todellisilla luokilla säästö kokonaan uudestaan kirjoitettuun verrattuna olisi moninkertainen.

9.3.4 Perintä

Viimeisenä vaihtoehtona tarkastelemme perintää (*inheritance*). Valinta koostamisen ja perinnän välillä on vaikea. Aina edes olioasiantuntijat eivät osaa sanoa yleispätevästi kumpiko on parempi. Nyrkkisääntönä voisi pitää seuraavaa *is-a* -sääntöä:

```
Jos voi sanoa että lapsiluokka on (is-a) isäluokka, niin peritään.
Jos sanotaan että lapsiluokassa on (has-a) isäluokka, niin koostetaan
```

Kokeillaanpa: "*luokka jossa on aika sekunteina*" on "*aika-luokka*". Kuulostaa hyvältä. Siis perimään:

olioalk\AikaSek8.java - laajentaminen perimällä

```
/**
 * Luokan laajentaminen perimällä
 * @author  Vesa Lappalainen
 * @version 1.0, 01.02.2003
 */
public class AikaSek8 extends Aika5 { // perii kaikki luokan Aika5 ominaisuudet

  private int s = 0;

  /**
   * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
   * @param h asetettavat tunnit
   * @param m asetettavat minuutit
   * @param s asetettavat sekunnit
   */
  public void aseta(int h,int m, int s) {
    this.s = s;  super.aseta(h,m); lisaa(0,0);
  }

  public AikaSek8() { }

  /**
   * Alustaa ajan
   * @param h tunnit
   * @param m minuutit
   * @param s sekunnit
   */
  public AikaSek8(int h,int m, int s) { // Muodostaja
    aseta(h,m,s);
  }

  public AikaSek8(int h,int m) { aseta(h,m,0); }
  public AikaSek8(int h)       { aseta(h,0,0); }

  /**
   * Tulostaa ajan muodossa 15:05
   * @param lf tulostetaanko rivinvaihto vai ei
   */
  public void tulosta(boolean lf) {
    super.tulosta(false);
    System.out.print(":" + (s<10?"0":"")+s);
    if ( lf ) System.out.println();
  }

  /**
   * Lisää aikaan valitun minuuttimäärän
   * @param lisa_min lisättävä minuuttimäärä
   * @param lisa_sek lisättävä sekunttimäärä
   */
  public void lisaa(int lisa_min,int lisa_sek)  {
    s += lisa_sek;
    super.lisaa(lisa_min+s/60);
    s %= 60;
  }

  public static void main(String[] args) {
    Aika5 a1 = new Aika5();
    Aika5 a2 = new Aika5(13);
    Aika5 a3 = new Aika5(14,175);

    a1.tulosta();  a2.tulosta();  a3.tulosta();
    a1.aseta(12,15); a2.aseta(16,-15);
    a1.tulosta(); a2.tulosta();
```

```
    AikaSek8 a4 = new AikaSek8(14,55,45); a4.tulosta();
    a4.lisaa(3,30); a4.tulosta();
    AikaSek8 a5 = new AikaSek8();         a5.tulosta();
    AikaSek8 a6 = new AikaSek8(12);       a6.tulosta();
    AikaSek8 a7 = new AikaSek8(12,15);    a7.tulosta();
  }

}
```

Tässä tapauksessa kirjoittamisen vaiva oli melkein sama kuin koostamisessakin. Niitä `aseta`, `lisaa` ja `tulosta` metodeja, jotka löytyivät jo kantaluokasta `Aika5` ei tarvinnut kirjoittaa. Kuitenkin esimerkiksi kaikki eri versiot muodostajasta pitää kirjoittaa, sillä muodostaja ei periydy Javassa.

Joissakin tapauksissa perimällä pääsee todella vähällä. Otamme tästä myöhemmin esimerkkejä, kunhan pääsemme eroon syntaksin esittelystä.

Lapsiluokka, aliluokka (*child class*, *subclass*) on se joka perii (Javassa *extands*) ja isäluokka, yliluokka (*parent class*, *super*) se joka peritään. Käytetään myös nimitystä välitön ali/yliluokka, kun on kyseessä perintä suoraan luokalta toiselle, kuten meidän esimerkissämme.

*Java*ssa välitön yliluokka ilmoitetaan aliluokan esittelyssä:

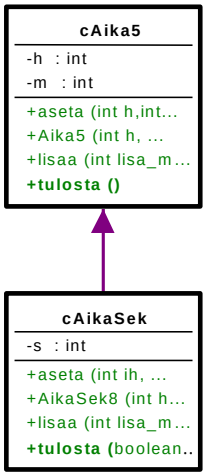
```
public class AikaSek8 extends Aika5 {
```

Jos täytyy viitata yliluokan metodeihin, joille on kirjoitettu aliluokassa oma määrittely, käytetään yliluokan viitettä `super`

```
    super.lisaa(lisa_min+s/60);
```

Yliluokan viitettä ei tarvita, mikäli samannimistä metodia ei ole aliluokassa.

Perintää kuvataan piirroksessa:



Kuva 9.2 Aika perinnällä

9.3.5 Polymorfismi, eli monimuotoisuus

Edellisestä esimerkistä ei oikeastaan paljastunut vielä mitään, mikä olisi puoltanut perintää. Korkeintaan snobbailu uudella syntaksilla. Mutta tosiasiassa pääsemme tästä kiinni olio-ohjelmoinnin tärkeimpään ominaisuuteen, ominaisuuteen jota on vaikea saavuttaa perinteisellä ohjelmoinnilla: polymorfismi (*polymorphism*) eli monimuotoisuus.

Lisätäänpä vielä testiohjelman loppuun:

```
    Aika5 a1 = new Aika5();  a1.aseta(12,15);
    AikaSek8 a4 = new AikaSek8(14,55,45); a4.tulosta();
...
    Aika5 aika = a1;  aika.tulosta();  // Esimerkki polymorfismista
    aika = a4;        aika.tulosta();
```

Tulostus:

```
12:15
14:59:15
```

Mistä tässä oli kyse? Viite `aika` on monimuotoinen, eli sama osoitin voi osoittaa useaan eri tyyppiseen luokkaan. Tämä on mahdollista, jos luokat ovat samasta perimähierarkiasta kuten tässä tapauksessa ja viite on tyypiltään näiden yhteisen kantaluokan olion viite.

9.3.6 Myöhäinen sidonta

Miksi edellä jälkimmäisessä `aika.tulosta()` kutsussa kutsuttiin luokan `AikaSek8` tulosta metodia eikä `Aika5` -luokan metodia `tulosta`. Tai tarkkaan ottaen kutsujärjestys on seuraava:

```
main aika=a4;
-> aika.tulosta();
-> Aika5.tulosta()
-> AikaSek8.tulosta(true)  // super.tulosta(false)
-> Aika5.tulosta(boolean lf)
```

Eli luokassa `AikaSek8` ei ole parametritonta tulosta metodia, siksi kutsutaan luokan `Aika5` `tulosta`-metodia. Tämä taas kutsuu `boolean`-tyyppisellä parametrilla varustettua `tulosta`-metodia. Sellainen löytyy siitä oliosta, jota `aika` tällä hetkellä edustaa, eli luokan `AikaSek8` metodi `tulosta(boolean lf)`. Vastaavasti tämä metodi kutsuu pakotetusti yliluokan metodia `super.tulosta(false)`. Jos sana `super` oli jäänyt pois, olisi kutsuta seurannut päättymätön rekursio.

Edellä mainittu on toteutettu siten, että alkuperäisessä luokassa kerrotaan että vasta ohjelman suoritusaikana selvitetään mistä luokasta todella on kyse, kun metodia kutsutaan. Tällaista ominaisuutta sanotaan myöhäiseksi sidonnaksi (*late binding*) (tälle monisteellekin tulee kyllä myöhäinen sidonta). Vastakohtana tälle on esimerkiksi C++:n oletustapa kutsua metodeja, eli aikainen sidonta (*early binding*). Sidonnan sisäisen mekanismin opettelun jätämme jollekin toiselle kurssille (ks. vaikkapa *Olio-ohjelmointi ja C++/VL*).

Javassa kutsutapana on onneksi aina myöhäinen sidonta, koska muuten perinnässä ei ole oikein mieltä.

Kun aliluokaan kirjoitetaan vastaava metodi kuin yliluokassa, käytetään tästä termiä uudelleenmäärittäminen (korvaaminen, syrjäyttäminen, *overriding*).

Tehtävä 9.10 Miksi vielä yksi lisaa-kutsu?

Osaatko selittää miksi `aseta`-metodissa pitää olla kutsut `this.s = s; super.aseta(h,m); lisaa(0,0);`? Jos osaat, olet jo melkein valmis `Java`-ohjelmoija!

Tehtävä 9.11 Ei turhaa lisaa-kutsua

Mitä tarvitsee muuttaa jotta viimeinen `lisaa`-kutsu saadaan pois?

9.3.7 Yliluokan muodostajan kutsuminen ennen muodostajaa

Joskus kirjoittamalla vähän enemmän voi saada aikaan nopeampaa ja turvallisempaa koodia. Niin tässäkin tapauksessa. Nimittäin säätäminen siinä, ettemme vaivautuneet kirjoittamaan muodostajia hirveän huolella, johtaa siihen että esimerkiksi alustus

```
AikaSek8 a4 = new AikaSek8(14,55,45);
```

kutsuu kaikkiaan seuraavia metodeja:

```
AikaSek8(14,55,45);
  Aika5();                // koska oli Aika5 pitää alustaa aluksi
    Aika5.aseta(0,0);
       Aika5.lisaa(0);       // Tässä Aika5, koska ei vielä "kasvanut" AikaSek8
  AikaSek8.aseta(14,55,45);
    Aika5.aseta(14,55);
      Aika5.lisaa(0);        //
        Aika.lisaa(0,0);
    AikaSek8.lisaa(0,0);
      Aika5.lisaa(0,0);
```

Olisitko arvannut! Enää ei tarvitse ihmetellä miksi olio-ohjelmat ovat isoja ja hitaita. Totta kai voitaisiin sanoa, että hyvän kääntäjän olisi pitänyt huomata tuosta optimoida päällekkäisyydet pois. Mutta tämä on vielä tänä päivänä kova vaatimus kääntäjälle. Mutta ehkäpä voisimme ohjelmoijina vähän auttaa:

olioalk\AikaSekB.java - tarkemmmin mietityt muodostajat

```
  public AikaSekB(int h,int m, int s) { // Muodostaja
    super(h,m); this.s = s; lisaa(0,0);
  }

  public AikaSekB(int h,int m) { super(h,m); s = 0; }
  public AikaSekB(int h)       { super(h);   s = 0; }
  public AikaSekB()            { super();    s = 0; }
```

olioalk\AikaB.java - tarkemmmin mietityt muodostajat

```
  public AikaB()            { this.h = 0; this.m = 0; }
  public AikaB(int h)       { aseta(h,0);             }
  public AikaB(int h,int m) { aseta(h,m);             }
```

Samalla on riisuttu pois alustukset

```
  private int h=0, m=0;
ja muutettu muotoon
  private int h,m;
```

Nyt on saatu ainakin seuraavat edut: oletustapauksessa kumpikin luokka alustuu pelkillä 0:ien sijoituksilla, ilman yhtään ylimääräistä kutsua.

ELI! Perityn yliluokan muodostajaa kutsutaan automaattisesti, jollei sitä itse tehdä. Nyt parametrillisessa muodostajassa kutsutaan yliluokan muodostajaa, ja näin voidaan välttää ainakin oletusmuodostajan kutsu:

```
public AikaSekB(int h,int m, int s) { super(h,m); this.s = s; lisaa(0,0);  }
```

Nyt alustuksesta

```
AikaSekB a4 = new AikaSekB(14,55,45);
```

seuraa seuraavat metodikutsut

```
AikaSekB(14,55,45);
  AikaB(14,55);                 // koska oli Aika5 pitää alustaa aluksi
    AikaB.aseta(14,55);
       AikaB.lisaa(0);        // Tässä AikaB, koska ei vielä "kasvanut" AikaSek8
    AikaSekB.lisaa(0,0);
       AikaB.lisaa(0,0);
```

Turhaahan tuossa on vieläkin, mutta tippuihan kuitenkin noin puolet pois! Joka tapauksessa periminen tuottaa jonkin verran koodia, aina kun yliluokan metodeja käytetään hyväksi. Ja jollei käytettäisi, kirjoitettaisiin samaa koodi uudelleen, eli palattaisiin 70-luvulle. Nykyisin kasvanut koneteho kompensoi usein tehottomamman oliokoodin ja olio-ohjelmoinnin ansiosta pystytään kirjoittamaan luotettavampia (?) ja monimutkaisempia ohjelmia. Eli etusija pitää antaa koodin selkeydelle ja ylläpidettävyydellä. Edellä esitetyt "kikkailut" huonontavat ylläpidettävyyttä jos jotakin oleellista muutetaan.

Tehtävä 9.12 Yliluokan alustajan kutsu

Pöytätestaa sekä AikaSek8.java että AikaSekB.java alustuksilla

```
AikaSek8 a4 = new AikaSek8(1,95,70);
AikaSekB a4 = new AikaSekB(1,95,70);
```

9.4 Kapselointi

Osa aikaisemmista ongelmista olisi voitu kiertää, mikäli olisimme päässeet käsiksi luokan yksityisiin tietoihin. Esimerkiksi alkuperäisen luokan `tulosta` olisi voitu jättää koskemattomaksi vaikka se olikin väärin tehty. Tätä olisi voitu helpottaa sillä, että kantaluokassa `Aika` olisi julistettu `h` ja `m` `protected` suojauksella. Tosin vain perityssä versiossa tästä olisi ollut apua.

9.4.1 Saantimetodit

Muutenkin saattaa tulla tilanteita, joissa luokan ulkopuolinen haluaa päästä käsiksi sisäisiin tietoihin. Ainakin lukemana niitä. Eihän ole ollenkaan tavatonta ajankaan kanssa, että joku haluaisi tietää tunnit, muttei tulostaa? Mikä ratkaisuksi? Julistetaanko kaikki attribuutit julkisiksi (`public`)? No ei sentään! Kirjoitetaan saantimetodi kullekin attribuutille, jonka perustellusti voidaan katsoa tarpeelliseksi jollekin ulkopuoliselle voitavan julkaista:

"Lopullinen" versio aikaluokastamme voisikin siis olla seuraava:

olioalk\AikaC.java - saantimetodit

```
public class AikaC {
  private int h,m;
...
  public int getH() { return h; }
  public int getM() { return m; }
...
}
```

Huomattakoon nyt, että perinnässä ei tarvitse määritellä uudestaan saantifunktioita `getH()` ja `getM()`, ainoastaan uudet, eli esimerkissämme `getS()`.

Nyt voitaisiin esimerkiksi kutsua:

```
System.out.println("Tunnit = " + a1.getH());
```

Mikä tässä sitten on erona attribuuttien julkaisemiseen verrattuna? Se että attribuutit ovat nyt tietyssä mielessä vain luettavissa (*read-only*), eli niitä voi lukea saantimetodien avuilla, mutta niitä voi asettaa vain `aseta`-metodin avulla, joka taas pystyy suorittamaan oikeellisuustarkistukset ja näin olion tila ei koskaan pääse muuttumaan olion itsensä siitä tietämättä.

Tehtävä 9.13 Saantimetodi sekunneille

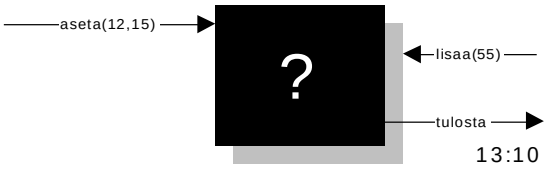
Täydennä AikaSekB.java:hen em. saantimetodit ja lisäksi `getS()` aliluokkaan `AikaSekB`.

Tehtävä 9.14 Saantimetodien käyttäminen

Muuta vielä edellisessä tehtävässä jokainen mahdollinen viittaus luokan sisälläkin saantimetodeja käyttäväksi suoran attribuuttiviittauksen sijasta.

9.4.2 Rajapinta ja sisäinen esitys

Kapseloinnin ansiosta luokan käyttämiseksi on tullut selvä rajapinta (*interface*): metodit, joilla olion tilaa muutetaan. Tämän rajapinnan ansiosta luokka muuttuu "mustaksi laatikoksi", jonka sisällöstä ulkomaailma ei tiedä mitään, mutta jonka kanssa voi kommunikoida metodien avulla.



Kuva 9.3 Musta laatikko

Tämä luokan sisustan piilottaminen antaa meille mahdollisuuden toteuttaa luokka oleellisesti eri tavalla. Voimme esimerkiksi toteuttaa ajan minuutteina vuorokauden alusta laskien:

olioalk\AikaD.cpp - sisäinen toteutus minuutteina

```
/**
 * Vaihettu sisäinen esitystapa
 * @author  Vesa Lappalainen
 * @version 1.0, 01.02.2003
 */
public class AikaD {

  private int yht_min;

  public void aseta(int h,int m)  { yht_min = 60*h + m; }

  public AikaD()                  { aseta(0,0); }
  public AikaD(int h)             { aseta(h,0); }
  public AikaD(int h,int m)       { aseta(h,m); }

  public void tulosta(boolean lf) {
    int m = getM();
    System.out.print("" + getH() + ":" + (m<10?"0":"")+m);
    if ( lf ) System.out.println();
  }

  public void tulosta() { tulosta(true); }

  public void lisaa(int lisa_min)  { yht_min += lisa_min;  }

  public static void lisaa(AikaD aika,int lisa_min) { aika.lisaa(lisa_min);  }

  public int getH() { return yht_min / 60; }
  public int getM() { return yht_min % 60; }

  public static void main(String[] args) {
    AikaD a1 = new AikaD();
    AikaD a2 = new AikaD(13);
    AikaD a3 = new AikaD(14,25);

    a1.tulosta();  a2.tulosta();  a3.tulosta();
    a1.aseta(12,15); a2.aseta(16,-15);
    a1.tulosta(); a2.tulosta();

    lisaa(a1,55); a1.tulosta();

    System.out.println("Tunnit = " + a1.getH());
  }

}
```

Tehtävä 9.15 `minuutteina()`

Lisää `AikaC`:hen ja `AikaD`:hen, eli molempiin sisäisiin toteutustapoihin saantimetodi `getMinuutteina`, joka palauttaa kellonajan vuorokauden alusta minuutteina laskettuna.

9.5 Rajapinta ja monimuotoisuus

Yksi perinnän tärkeimmistä ominaisuuksista on mahdollisuus monimuotoisuuteen, polymorfismiin. Esimerkiksi

olioalk\AikaSekB.java - esimerkki polymorfisesta taulukosta

```
    AikaB ajat[] = new AikaB[5];
    ajat[0] = a1; ajat[1] = a2; ajat[2] = a3; ajat[3] = a4;
    ajat[4] = new AikaSekB(23,59,59);

    for (int i=0; i < ajat.length; i++ ) {
      ajat[i].tulosta(false); System.out.print(" +" + i + " => ");
      ajat[i].lisaa(i); ajat[i].tulosta();
    }
```

Taulukko `ajat` koostuu viitteistä `AikaB`-luokan olioihin. Myös `AikaSekB` toteuttaa saman rajapinnan, koska se on perittyä samasta luokasta. Siksi taulukkoon voi laittaa mitä tahansa `AikaB`-luokan jälkeläisluokankin olioita.

Esimerkissä `AikaSek7` luokka koostettiin sekunneista ja luokan `Aika5` oliosta. Nyt valitettavasti vain polymorfismi ei toimi, eli `AikaSek7` ja `Aika5` eivät ole perimissuhteessa toisiinsa. Niillä on kyllä Javassa yhteinen kantaluokka `Object`, koska Javassa kaikki luokat periytyvät `Object`-luokasta. Mutta yhteistä aikaan liittyvää rajapintaa niillä ei ole. Kömpelö polymorfismi saataisiin aikaan seuraavasti:

olioalk\AikaSek7.java - kömpelö esimerkki polymorfisesta taulukosta

```
    Object ajat[] = new Object[5];
    ajat[0] = a1; ajat[1] = a2; ajat[2] = a3; ajat[3] = a4;
    ajat[4] = new AikaSekE(23,59,59);

    for (int i=0; i < ajat.length; i++ ) {
      if ( ajat[i] instanceof Aika5 ) {
        Aika5 aika = (Aika5)ajat[i];  // pakotettu tyypin muunnos
        aika.tulosta(false); System.out.print(" +" + i + " => ");
        aika.lisaa(i); aika.tulosta();
      }
      if ( ajat[i] instanceof AikaSek7 ) {
        AikaSek7 aika = (AikaSek7)ajat[i];  // pakotettu tyypin muunnos
        aika.tulosta(false); System.out.print(" +" + i + " => ");
        aika.lisaa(i); aika.tulosta();
      }
    }
```

Tavassa jossa joudutaan testaamaan olion tyyppiä, tulee uusien tyyppien lisääminen järjestelmään erittäin työlääksi.

Javassa avuksi tulee rajapintakäsite. Teemme ensin "mallin" siitä, minkälainen on vähintään kaikkien Aika-luokkien rajapinta:

olioalk\AikaRajapinta.java - malli kaikkien Aika-luokkien rajapinnasta

```
public interface AikaRajapinta {
  void aseta(int h,int m);
  void tulosta(boolean lf);
  public void tulosta();
  void lisaa(int lisa_min);
  int getH();
  int getM();
}
```

Seuraavaksi kaikki luokat, jotka halutaan kuuluvan "samaan kategoriaan", ilmoitetaan toteuttavan tämän rajapinnan:

olioalk\AikaE.java - luokka joka toteuttaa rajapinnan

```
public class AikaE implements AikaRajapinta {

  private int h, m;
  public void aseta(int h,int m) { this.h = h; this.m = m; lisaa(0);  }

  public AikaE()            { this.h = 0; this.m = 0; }
  public AikaE(int h)       { aseta(h,0);             }
  public AikaE(int h,int m) { aseta(h,m);             }

  public void tulosta(boolean lf) {
    System.out.print("" + h + ":" + (m<10?"0":"")+m);
    if ( lf ) System.out.println();
  }

  public void tulosta() { tulosta(true); }

  public void lisaa(int lisa_min)  {
    int yht_min = h * 60 + m + lisa_min;
    h = yht_min / 60;
    m = yht_min % 60;
  }

  public int getH() { return h; }
  public int getM() { return m; }

}
```

Sitten esim. koosteluokka ilmoitetaan toteuttamaan myös sama rajapinta:

olioalk\AikaE.java - luokka joka toteuttaa rajapinnan => polymorfismi

```
public class AikaSekE implements AikaRajapinta {

  private AikaE hm = new AikaE();
  private int s;

  public void aseta(int h,int m,int s) { hm.aseta(h,m); this.s = s; lisaa(0,0); }

  public void aseta(int h,int m)       { aseta(h,m,0); }
  public void aseta(int h)             { aseta(h,0);   }

  public AikaSekE()                    { aseta(0,0,0}; }
  public AikaSekE(int h,int m, int s)  { aseta(h,m,s); }
  public AikaSekE(int h,int m)         { aseta(h,m,0); }
  public AikaSekE(int h)               { aseta(h,0,0); }

  public void tulosta(boolean lf) {
    hm.tulosta(false);
    System.out.print(":" + (s<10?"0":"")+s);
    if ( lf ) System.out.println();
  }

  public void tulosta() { tulosta(true); }

  public void lisaa(int lisa_min,int lisa_sek)  {
    s += lisa_sek;
    hm.lisaa(lisa_min+s/60);
    s %= 60;
  }

  public void lisaa(int lisa_min)  { lisaa(lisa_min,0); }

  public int getH() { return hm.getH(); }
  public int getM() { return hm.getM(); }

  public static void main(String[] args) {
    AikaE a1 = new AikaE();
    AikaE a2 = new AikaE(13);
    AikaE a3 = new AikaE(14,175);

    a1.tulosta();  a2.tulosta();  a3.tulosta();
    a1.aseta(12,15); a2.aseta(16,-15);
    a1.tulosta(); a2.tulosta();

    AikaSekE a4 = new AikaSekE(14,55,45); a4.tulosta();
    a4.lisaa(3,30); a4.tulosta();

    // Rajapintaan perustuva esimerkki polymorfisesta taulukosta
    AikaRajapinta ajat[] = new AikaRajapinta[5];
    ajat[0] = a1; ajat[1] = a2; ajat[2] = a3; ajat[3] = a4;
    ajat[4] = new AikaSekE(23,59,59);

    for (int i=0; i < ajat.length; i++ ) {
      ajat[i].tulosta(false); System.out.print(" +" + i + " => ");
      ajat[i].lisaa(i); ajat[i].tulosta();
    }

  }

}
```

Näin voimme jälleen tehdä taulukon, johon voimme laittaa kaikkia `AikaRajapinta`-määrittelyn toteuttavien luokkien olioita.

9.6 `Object`-luokan metodien korvaaminen

Jos Javassa ei peritä luokkaa mistään, niin se periytyy aina `Object`-luokasta. Näin siksi, että kaikki oliot saadaan samaan hierarkiaan ja voidaan esimerkiksi tallentaa samaan tietorakenteeseen. Käytännössä tämä ei ole kovin kätevää, sillä silloin tietorakenteessa olevilla olioilla on käytössä vain `Object`-luokan metodit. Jotta olioilla voitai-

siin tehdäkin jotakin, pitää niiden tyyppi muuntaa vastaamaan niiden varsinaista luokkaa.

`Object`-luokassa on kuitenkin muutama tärkeä metodi, joiden olemassa olosta ohjelmoijan on hyvä olla tietoinen:

```
Object clone();             // tekee oliosta itsestään kopion
boolean equals(Object obj); // vertaa oliota toiseen olioon
int hashCode();             // palauttaa olioon liityvän "lajitteluavaimen"
String toString();          // palauttaa olion merkkijonona
```

olioalk\AikaF.java - luokka joka toteuttaa Object

```
/**
 * Luokka toteuttamaan sovitun julkisen rajapinnan ja Object-
 * luokan metodeja.
 * @author  Vesa Lappalainen
 * @version 1.0, 01.02.2003
 */
public class AikaF implements AikaRajapinta {

  private int h, m;

  /**
   * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
   * @param h asetettavat tunnit
   * @param m asetettavat minuutit
   */
  public void aseta(int h,int m) {
    this.h = h; this.m = m; lisaa(0);
  }

  public AikaF() { this.h = 0; this.m = 0; }

  /**
   * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
   * @param h asetettavat tunnit
   */
  public AikaF(int h) {
    aseta(h,0);
  }

  /**
   * Alustaa ajan
   * @param h tunnit
   * @param m minuutit
   */
  public AikaF(int h,int m) { // Muodostaja
    aseta(h,m);
  }

  /**
   * Tulostaa ajan muodossa 15:05
   * @param lf tulostetaanko rivinvaihto vai ei
   */
  public void tulosta(boolean lf) {
    System.out.print(toString());
    if ( lf ) System.out.println();
  }

  /**
   * Tulostaa ajan muodossa 15:05 sekä aina rivinvaihdon
   */
  public void tulosta() {
    tulosta(true);
  }
```

```
  /**
   * Lisää aikaan valitun minuuttimäärän
   * @param lisa_min lisättävä minuuttimäärä
   */
  public void lisaa(int lisa_min)  {
    int yht_min = h * 60 + m + lisa_min;
    h = yht_min / 60;
    m = yht_min % 60;
  }

  public static void lisaa(AikaF aika,int lisa_min) {
    int yht_min = aika.h * 60 + aika.m + lisa_min;
    aika.h = yht_min / 60;
    aika.m = yht_min % 60;
  }

  public int getH() { return h; }
  public int getM() { return m; }

  public String toString() {
    return "" + h + ":" + (m<10?"0":"")+m;
  }

  public boolean equals(Object o) {
    if ( !(o instanceof AikaRajapinta) ) return false;
    AikaRajapinta a = (AikaRajapinta)o;
    return getH() == a.getH() && getM() == a.getM();
  }

  public Object clone() {
    return new AikaF(getH(),getM());
  }

  public int hashCode() {
    return 3600*getH() + 60*getM();
  }

  public static void main(String[] args) {
    AikaF a1 = new AikaF();
    AikaF a2 = new AikaF(13);
    AikaF a3 = new AikaF(14,25);

    a1.tulosta();  a2.tulosta();  a3.tulosta();
    a1.aseta(12,15); a2.aseta(16,-15);
    a1.tulosta(); a2.tulosta();

    lisaa(a1,55); a1.tulosta();

    System.out.println("Tunnit = " + a1.getH());
    System.out.println(a1.toString());
  }

}
```

Esimerkiksi tekemällä metodi

```
  public String toString() {
    return "" + h + ":" + (m<10?"0":"")+m;
  }
```

vältämme kaikki tulostukseen liittyvät ongelmat. Jos halutaan verrata kahta Aika-oliota keskenään, kannattaa kirjoittaa `equals`-metodi.

```
  public boolean equals(Object o) {
    if ( !(o instanceof AikaRajapinta) ) return false;
    AikaRajapinta a = (AikaRajapinta)o;
    return getH() == a.getH() && getM() == a.getM();
  }
```

`equals`-metodia kirjoitettaessa on oltava huolellinen, sillä parametrina saattaa tulla oikean tyyppinen olio tai sitten väärän tyyppinen olio. `equals`-metodin pitää toteuttaa seuraavat ominaisuudet:

```
Olkoon seuraavassa a1,a2 ja a3 kolme luokan oliota.
reflektiivisyys:  a1.equals(a1) pitää olla aina tosi
symmetrisyys:     a1.equals(a2) == a2.equals(a1)
transitiivisuus:  jos a1.equals(a2) && a2.equals(a3) niin a1.equals(a3)
luonnollisesesti toistuvien equals kutsujen pitää palauttaa samoille olioille
sama arvo, mikäli olioiden samuuteen vaikuttava tila ei muutu
```

Jos luokkaan toteutetaan `equals`-metodi, on siihen toteutettava myös hajautusarvo. Javan tietorakenteet tarvitsevat hajautusarvoa. Hajautusarvon täytyy palauttaa sama luku olioille, jotka ovat `equals`-vertailussa saman arvoisia. Mutta kaksi eriarvoistakin oliota saa palauttaa saman hajautusarvon. Meidän tapauksessamme hajautusarvoksi voidaan valita vaikkapa sekunnit vuorokauden alusta:

```
public int hashCode() {
  return 3600*getH() + 60*getM();
}
```

Lisäksi monessa tilanteessa tarvitaan oliosta samanlainen kopio. Tätä varten toteutetaan `clone`-metodi:

```
public Object clone() {
  return new AikaF(getH(),getM());
}
```

Tehtävä 9.16 `equals` `toString` avulla

Toteuta `equals`-metodi `toString`-metodin avulla. Arvioi ratkaisun tehokkuutta.

Tehtävä 9.17 `equals` `AikaSek`-luokkaan

`equals`-metodiin tulee ongelmia toteutettaessa `AikaSek`-luokkaa. Mieti mitä.

Tehtävä 9.18 `AikaSek` perimällä.

Esimerkissä `AikaSekF` on toteutettu sekunnit sisältävä aikaluokka koostamalla. Kokeile miten nyt onnistuu perintä `AikaF`-luokasta ja mitä metodeja pitää korvata.

Tehtävä 9.19 Vertailu

Tutki dokumenteista rajapintaa `Comparable` ja muuta luokat `AikaF` ja `AikaSekF` toteuttamaan tuo rajapinta.

9.7 Mistä hyviä luokkia

Alunperin kirjoittamamme luokka `Aika` kokikin varsin kovia tarkemmassa tarkastelussa. Näistä muutoksista osa oli vielä aivan perusasioita; läheskään kaikkea emme vieläkään ole ottaneet huomioon (vertailu, syöttö päätteeltä, muuntaminen merkkijonoksi ja takaisin, lisäyksessä tapahtuvan ylivuodon luovuttaminen päivämäärälle, jne.). Miten sitten monimutkaisempien luokkien kanssa? Niin kauan pärjää, kun luokat on omaan käyttöön. Heti kun yritetään tehdä yleiskäyttöisiä luokkia (joka on yksi olio-ohjelmoinnin tavoite), tuleekin ongelmia vastaan.

Paremmalla suunnittelulla luokasta olisi heti voinut tulla yleiskäyttöisempi. Usein jopa joudutaan tekemään kahden luokan yläpuolelle abstrakti, tai muuten vaan yhteinen yliluokka, josta hieman toisistaan poikkeavat luokat peritään. Kuljimme tämän pitkän tien sen vuoksi, että lukija oppisi ymmärtämään miksi valmiit luokat eivät ole parin rivin koodinpätkiä.

Tulevaisuudessa ohjelmoijat jakaantunevatkin selvästi kahteen ryhmään: toiset käyttävät valmiita luokkia (mikä on helppoa, jos luokat ovat kunnossa, vrt. *Delphi* tai *Visual Basic*, ehkä myös osittain *Java* ja Jonnen pyynnöstä mainitaan tietysti *Python*). Ammattitaitoisempi ryhmä sitten suunnittelee ja tekee näitä yleiskäyttöisiä luokkia. Sitä mukaa kun luokkia saadaan valmiiksi eri "elämän aloille", siirtyy ammattilaiset yhä spesifimmille aloille.

Esimerkiksi *M$:in Windowsin* ohjelmointiin tarkoitetut luokkakirjastot ovat paisuneet niin suuriksi, että niiden käyttämistä tuskin kukaan enää hallitsee, ja ennen kuin entisen kirjaston on ehtinyt edes auttavasti oppia, tuleekin uusi versio. Tästä tulee oravanpyörä, jossa voi olla kova homma pysyä mukana, jollei ohjelmateollisuus keksi jotakin uutta ja mullistavaa avuksi.

Joka tapauksessa haave siitä, että näkee näyn ja keksii hyvän luokan, jota muut sitten yhtään muuttamatta voivat käyttää hyväkseen, kannattaa heittää Ylistönrinteen sillan alle. Mieluummin kannattaa alistua siihen, että opettelee käyttämään hyviä luokkia ja imee niitä käyttäessään ideoita siitä, miten parantaa omia luokkiaan seuraavalla kerralla.

9.8 Valmiita luokkia

Olio-ohjelmoinnin eräs tavoite on tuottaa ohjelmoijien käyttöön yleiskäyttöisiä komponentteja, jotta jokainen ei keksisi samaa pyörää uudelleen. Erityisesti graafisen ohjelmoinnin puolella ja sekä myös tietokantaohjelmoinnin puolella näitä komponentteja onkin varsin mukavasti. *Borlandin Delphillä* syntyy melkein *Kerho*-ohjelmaamme vastaava *Windows*-ohjelma lähes koodaamatta, pelkästään pudottelemalla komponentteja lomakkeelle.

9.8.1 Merkkijonot

Jos kerrankin pääsisin vastakkain nykykielten kehittäjien kanssa, niin tekisi kovasti mieli kysyä ovatko he koskaan tehneet oikeaa ohjelmaa. Nimittäin lähes kielestä riippumatta kunnolliset merkkijonot loistavat poissaolollaan. Ja ohjelmoijat ovat käyttäneet ääirettömästi työtunteja tehdessään itselleen aluksi edes auttavaa merkkijonokirjastoa. Ainoastaan "lelukielissä" - *Basicissä* ja *Turbo Pascalissa* on ollut hyvät ja turvalliset merkkijonot.

C-kielen `char jono[10]` on todellinen aikapommi, jonka aukkoisuuteen perustuu vielä tänäkin päivänä useat hakkereiden kikat murtautua vieraisiin tietojärjestelmiin. Katsotaanpa ensin mitä C-merkkijonoille voi/ei voi tehdä:

☹

```
char s1[10],s2[5],*p;
p = "Kana"           // Toimii!
p[0] = 'S';          // Toimii! Mutta jatkossa käy huonosti...
s1 = "Kissa";        // ei toimi!
strcpy(s2,"Koira");  // Huonosti käy!  Miksi?  Älä käytä koskaan...
if ( s1 < s2 ) ...   // Sallittu, mutta tekee eri asian kuin lukija arvaakaan...
gets(s1);            // Itsemurha, tämä on eräs kaikkein hirveimmistä funktioista
                     // lukee päätteeltä rajattomasti merkkejä ...
fgets(s1,sizeof(s1),stdin);  // Oikein!  Tosin rivinvaihto jää jonoon jos syöte
                             // on lyhyempi kuin 9 merkkiä
printf(s1);          // Ohohoh!   Tämä jopa toimii!!!
cout << s1;          // Ja jopa tämäkin!!!
cin >> s1;           // Taas itsemurha ....
```

Jos käytetään C-kieltä, pitää käyttää varsin paljon aikaa siihen miten C:n merkkijonoja voidaan kohtuullisen turvallisesti käyttää.

Onneksi *C++*:assa on kohtuullinen merkkijonoluokka. Nyt jo! Yli 10 vuotta kielen kehittämisen jälkeen...

Katso esimerkiksi: http://www.iki.fi/gaia/tekstit/cxxstring/

Javassa on vastaavasti kaksi merkkijonoluokkaa: `String` ja `StringBuffer`. Ensin mainittu koskee merkkijonoja, joita ei koskaan (*immutable*) tarvitse muuttaa, vaan riittää aina luoda uusi merkkijono. Jälkimmäistä käytetään, mikäli jonoon tulee paljon muutoksia (*mutable*).

Tehtävä 9.20 Ensimmäinen melkein järkevä olio

Täydennä seuraavat luokat ja testaa ohjelma.

olioalk\Henkilo.java - 1. järkevä olio

```
/**
 * Henkilöluokka
 * Täydennä luokka.
 * @author  Vesa Lappalainen
 * @version 1.0, 05.02.2003
 */
public class Henkilo {

  private String nimi;
  private int ika;
  private double pituus_m;

  Henkilo(String nimi, int ika, double pituus_m) {

  }

  public String toString() {
    return "";
  }

  public void kasvata(double cm) {

  }

  public static void main(String[] args) {

  }

}
```

olioalk\Opiskelija.java - 1. järkevä olio

```
/**
 * Opiskelija, joka on peritty henkilöstä
 * @author  Vesa Lappalainen
 * @version 1.0, 05.02.2003
 */
public class Opiskelija extends Henkilo {
  double keskiarvo;

  Opiskelija(String nimi, int ika, double pituus_m, double keskiarvo) {
     super(nimi,ika,pituus_m);
  }

  public String toString() {
    return "";
  }

  public static void main(String[] args) {
    Henkilo Kalle = new Henkilo("Kalle",35,1.75);
    System.out.println(Kalle);
    Kalle.kasvata(2.3);
    System.out.println(Kalle);
    Opiskelija Ville = new Opiskelija("Ville",21,1.80,9.9);
    System.out.println(Ville);
  }

}
```

10. Java–kielen ohjausrakenteista ja operaattoreista

Ihvilläpä ihmettele
silmukalla suorittele
lopetukset laskeskele
virityksii vierastele.

Alusta kun ehto jääpi
siit silmukka iänikuinen
aina suru ei surkeen suuri
joutaapa avuksi tääkin.

Katkoo saapi keskeltäkin
jatkaa vaikka muualtakin
paluu kelpo keino myöskin
kunhan kaikki katseltuna.

Mitä tässä luvussa käsitellään?

- `if-else` –lause
- loogiset operaattorit: `&&`, `||` ja `!`
- bittitason operaattorit: `&`, `|`, `^` ja ~
- silmukat `while`, `do-while` ja `for`
- silmukan "katkaisu" `break`, `continue`, `goto`
- sijoituslauseet: `=` `+=` `-=` jne.
- valintalause `switch`

Syntaksi:

```
lause  joko       ylause;                              // HUOM! Puolipiste
       tai        lohko                                // eli koottu lause
ylause            yksinkertainen lause
  esim              a = b + 4
                    vaihda(a,b)
lohko             { lause1 lause2 lause3 }             // lauseita 0-n
  esim              { a = 5; b = 7; }
ehto              lauseke joka tuottaa false tai true
  esim              a < 5
                    ( 5 < a ) && ( a < 10 )
                    !(a == 0)                          // jos a=0 => 1, muuten 0
  HUOM!           Vertailu a == 5

if-else           if ( ehto ) lause1
                  else lause2                          // ei pakollinen
while             while ( ehto ) lause;
do-while          do lause while ( ehto );
for               for ( ylause1a,ylause2a; ehto ; ylause1k,ylause2k ) lause
  esim              for ( i=0,s=0; i<10; i++ ) s += i;  // ylause1a
switch            switch ( lauseke ) {
                    case arvo1: lause1 break;          // valintoja 0-n
                    case arvo2:                        // arvolla 2 ja 3 sama
                    case arvo3: lause2 break;
                    default:    laused break;          // ei pakollinen
                  }
```

Luvun esimerkkikoodit:

```
http://www.mit.jyu.fi/~vesal/kurssit/ohj2/moniste/esim/java-silm/
```

Ohjelma jossa ei ole minkäänlaista valinnaisuutta tai silmukoita on varsin harvinainen. Kertaamme seuraavassa Java–kielen tarjoamat mahdollisuudet suoritusjärjestyksen ohjaamiseen. Samalla näemme kuinka suomenkielisen algoritmin kääntäminen ohjelmointikielelle on varsin mekaanista puuhaa.

10.1 if–lause

Mikäli meillä on kaksi lukua, jotka pitäisi olla suuruusjärjestyksessä, voisimme hoitaa järjestämisen seuraavalla algoritmilla:

```
1. Jos luvut väärässä järjestyksessä,
     niin vaihda ne keskenään
```

Tämän kirjoittamiseksi ohjelmaksi tarvitsemme ehto–lausetta:

```
if ( ehto ) ylause1;
lause2;
```

Huomattakoon, että tässä sulut ehdon ympärillä ovat pakolliset. `lause1` suoritetaan vain kun `ehto` on voimassa. `lause2` suoritetaan aina. Lause voitaisiin kirjoittaa myös muodossa

```
if(ehto) ylause1;
lause2;
```

muttei näin tehdä, jotta erottaisimme paremmin funktion ja `if`–lauseen toisistaan. Saa tulee koskemaan myös `for`, `while` ja muita vastaavia rakenteita.

10.1.1 Ehdolla suoritettava yksi lause

Olkoon meillä aliohjelma nimeltään `tulosta`, joka parametrina viedyn luvun:

```
if ( a > b ) tulosta(a);
```

10.1.2 Ehdolla suoritettava useita lauseita

Jos esimerkiksi luvut pitäisi vaihtaa keskenään, täytyisi meidän voida suorittaa useita lauseita muuttujien vaihtamiseksi. Java–kielessä voidaan lausesuluilla kasata joukko lauseita yhdeksi lauseeksi (lohko, koottu lause, *block*):

Vinkki

Sisennä kauniisti

```
if ( a > b ) {
  t = a;
  a = b;
  b = t;
}
```

Huomautus! Lauseiden kirjoittaminen samalle riville ei auttaisi mitään, sillä

```
if ( a > b ) t = a; a = b; b = t;
/*  vastaisi loogisesti rakennetta: */
if ( a > b ) t = a;
a = b;
b = t;
```

Koodia voidaan kuitenkin usein lyhentää kirjoittamalla asioita samalle riville:

```
if ( a > b ) {
  t = a; a = b; b = t;
}
/* tai joskus jopa */
if ( a > b ) { t = a; a = b; b = t; }
```

Niin kauan kuin todella hallitsee asian, voi olla helpointa laittaa aina `if`–lauseen ainoakin suoritettava lause lausesulkuihin

```
if ( a > b ) {
  tulosta(a);
}
```

Mikäli sulkuja ei olisi, täytyisi toisen lauseen lisäyksen yhteydessä muistaa lisätä myös sulut (tosin eihän hyvin suunniteltua ohjelmaa tarvinnut enää jälkeenpäin paikata?).

Tehtävä 10.1 vaihda

Esitä pöytätestin avulla miksei vaihtaminen onnistu pelkästään lauseilla:
a = b; b = a;

Tehtävä 10.2 abs

Kirjoita funktio
`int itseisarvo(int i),`
joka palauttaa `i`:n itseisarvon (negat. muutet. posit.).

Tehtävä 10.3 jarjesta2

Kirjoita aliohjelma
`void tulosta2(int a, int b),`
joka tulostaa luvut suuruusjärjestyksessä .

Tehtävä 10.4 maksimi ja minimi

Kirjoita funktio
`int maksimi(int a, int b),`
joka palauttaa suuremman kahdesta luvusta.

Kirjoita vastaava funktio `minimi`.

10.2 Loogiset lausekkeet

Java–kielessä vain boolean-arvoiset lausekeet käsitellään loogisina lausekkeina. Arvo false on epätosi ja true on tosi.

```
a = 4;
if ( a == 4 ) ...
boolean samat;
samat = ( a == 4 );
if ( samat ) ...
```

10.2.1 Vertailuoperaattorit

Vertailuoperaattorin käyttö muodostaa loogisen lausekkeen, jonka arvo on 0 tai 1. Vertailuoperaattoreita ovat:

```
==  yhtäsuuruus
!=  erisuuruus
<   pienempi kuin
<=  pienempi tai yhtä kuin
>   suurempi kuin
>=  suurempi tai yhtä kuin
```

Esimerkkejä vertailuoperaattoreiden käytöstä:

```
if ( a < 5 )  System.out.println("a alle viisi!");
if ( a > 5 )  System.out.println("a yli viisi!");
if ( a == 5 ) System.out.println("a tasan viisi!");
if ( a != 5 ) System.out.println("a ei ole viisi!");
```

10.2.2 Sijoitus palauttaa arvon!

Yhtäsuuruutta verrataan == operaattorilla, EI sijoituksella =. Tämä on eräs tavallisimpia aloittelevan (ja kokeneenkin) C–ohjelmoijan virheitä:

```
/* Seuraava tulostaa vain jos a == 5 */
if ( a == 5 ) tulosta("a on viisi!\n");  /* Kääntyy Javassa ja C:ssä */

/* Seuraava sijoittaa aina a = 5 ja tulostaa AINA! */
if ( a = 5 ) printf("a:ksi tulee AINA 5!\n");  /* Kääntyy vain C:ssä */
```

☹

Sijoitus `a=5` on myös lauseke, joka palauttaa arvon 5. Siis sijoitus kelpaa tästä syystä vallan hyvin loogiseksi lausekkeeksi C-kielessä. Onneksi Javassa tämä sijoituksen tuloksena synytynyt lausekkeen kokonaislukuarvo EI kelpaa boolean-arvoksi, joten kääntäjä ei hyväksy sijoitusta vahingossa yhtäsuuruuden vertailun tilalle..

Joskus ominaisuutta voidaan tarkoituksella käyttää hyväksikin. Esimerkiksi halutaan sijoittaa AINA `a=b` ja sitten suorittaa jokin lause, mikäli `b!=0`. Tämä voitaisiin kirjoittaa useilla eri tavoilla:

java-silm\Ifsij2.java - esimerkki tahallisesta sijoituksesta ehdossa

```
    int a,b=5;
    /*1*/  // a = b; if ( b ) tulosta("b ei ole nolla!");
    /*2*/  a = b; if ( b != 0 ) tulosta("b ei ole nolla!");
    /*3*/  // if ( a = b ) tulosta("b ei ole nolla!");
    /*4*/  if ( (a=b) != 0 ) tulosta("b ei ole nolla!");
```

Edellisistä tapa 3 on C–mäisin, mutta Java-kääntäjä ei onneksi hyväksy sitä. Jotta C–mäinen tapa voitaisiin säilyttää, voidaan käyttää tapaa 4 jonka kääntäjä hyväksyy. Oleellista on, että sijoitus on suluissa (muuten tulisi sijoitus `a =(b!=0)`). Mikäli asian toimimisesta on pieninkin epäilys kannattaa käyttää tapaa 2!

Tyypillinen esimerkki sijoituksesta ja testauksesta samalla on vaikkapa tiedoston lukeminen:

```
while ( ( rivi = f.readLine() ) != null ) { // jos sijoitus palauttaa null,
                                            // on tiedosto loppu
  ... käsitellään tiedoston riviä
}
```

Jos edellisen esimerkin tiedoston lukemisessa ei käytettäisi sijoitusta ja testiä samalla, pitäisi tämä kirjoittaa muotoon:

```
while ( true ) {
  rivi = f.readLine();
  if ( rivi == null ) break;
  ... käsitellään tiedoston riviä
}
```

10.3 Loogisten lausekkeiden yhdistäminen

Loogisia lauseita voidaan yhdistää loogisten operaatioiden avulla. Tietysti lauseita voidaan yhdistää myös normaaleilla operaatioilla (+,−,*,/), mutta tämä ei ole oikein hyvien tapojen mukaista.

10.3.1 Loogiset operaattorit &&, || ja !

```
&&  ja
||  tai
!   muuttaa ehdon arvon päinvastaiseksi (eli false->true, true->false)
```

Mikäli yhdistettävät ehdot koostuvat esimerkiksi vertailuoperaattoreiden käytöstä, kannattaa ehtoja sulkea sulkuihin, jottei seuraa turhia epäselvyyksiä.

```
if ( ( rahaa > 50 ) && ( kello < 19 ) )  tulosta("Mennään elokuviin!");
if ( ( rahaa < 50 ) || ( kello >3 ) )    tulosta("Ei kannata mennä kapakkaan!");
if ( ( 8 <= kello ) && ( kello <= 16 ) ) tulosta("Pitäisi olla töissä!");
if ( ( rahaa == 0 ) || ( sademaara < 10 ) )  tulosta("Kävele!");
```

Usein tulee vastaan tilanne, jossa pitäisi testata onko luku jollakin tietyllä välillä. Esimerkiksi onko

```
1900 <= vuosi <= 1999
```

palauttaisi C-kielisenä lauseena aina 1. Miksikö? Koska lause jäsentyy

```
( 1900 <= vuosi ) <= 1999
     0 tai 1      <= 1999  eli aina 1
```

Javassa onneksi lauseke ei edes käänny, koska totuusarvoa ja kokonaislukua ei voi verrata keskenään. Oikea tapa kirjoittaa väli olisi:

```
if ( ( 1900 <= vuosi ) && ( vuosi <= 1999 ) ) ...
```

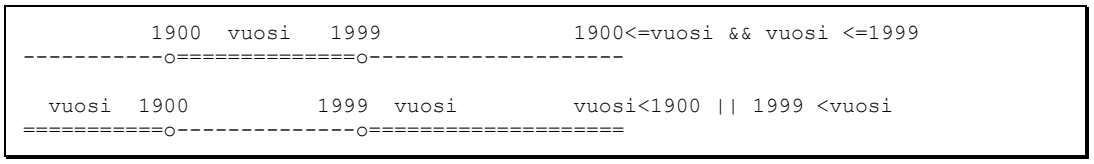
Huomattakoon edellä miten väliä korostettiin kirjoittamalla välin päätepisteet lauseen laidoille.

Java–kielen sidontajärjestyksen ansiosta lause toimisi myös ilman sisimpiä sulkuja, mutta ne kannattaa pitää mukana varmuuden vuoksi. Vertailtavat kannattaa kirjoittaa nimenomaan tähän järjestykseen, koska tällöin vertailu muistuttaa eniten alkuperäistä väliämme!

Vastaavasti jos arvon halutaan olevan välin ulkopuolella, kannattaa kirjoittaa:

```
if ( ( vuosi < 1900 ) || ( 1999 < vuosi ) ) ...
```

Tällöin epäyhtälöiden suuntaa ei joudu koskaan miettimään, vaan arvot ovat aina siinä järjestyksessä kuin lukusuorallakin:



10.3.2 Loogisen lausekkeen suoritusjärjestys

Loogiset lausekkeet suoritetaan AINA vasemmalta oikealle, kunnes ehdon arvo on selvinnyt.

Siis: Loogisen lausekkeen evaluoiminen lopetetaan heti kun ehdon arvo selviää (*boolean expression shortcut*).

Esimerkiksi:

```
if ( a != 0 || ( (b=c)==0 ) ) System.out.println("Kukkuu");
```

Tai-operaattorin (`||`) oikealla puolella oleva sijoitus suoritetaan vain mikäli `a==0`:

a	b	c	sij.suor	tulostetaan
0	?	0	kyllä	kyllä
0	?	3	kyllä	ei
5	?	0	ei	kyllä
5	?	3	ei	kyllä

Tätä ominaisuutta voidaan käyttää hyväksi esimerkiksi jos on vaara että olion arvo on null:

```
    if ( (jono != null) && jono.equals("kissa") ) tulosta("On kissa");
```

Tällöin testissä ei turhaan tule **null**-viittausta koska ehtoa `jono.equals` ei suoriteta muuta kuin jonon ollessa viite todelliseen olioon.

10.3.3 Loogiset operaattorit & ja |

Ja (`&&`) ja tai (`||`) -operaattoreista on myös versiot, joilla aina evaluoidaan (suoritetaan) kaikki lausekkeen osat, vaikka ehdon arvo selviäisi jo aikaisemminkin.

```
&  ja - suorittaa aina lauskekkeen molemmat puolet
|  tai - suorittaa aina lausekkeen molemmat puolet
```

Aikasempaa esimerkkiä mukaellen:

```
if ( a != 0 | ( (b=c)==0 ) ) System.out.println("Kukkuu");
```

Tai-operaattorin (`|`) oikealla puolella oleva sijoitus suoritetaan riippumatta `a`: n arvosta:

a	b	c	sij.suor	tulostetaan
0	?	0	kyllä	kyllä
0	?	3	kyllä	ei
5	?	0	kyllä	kyllä
5	?	3	kyllä	kyllä

Vastaavasti olisi paha virhe kirjoittaa:

```
if ( (jono != null) & jono.equals("kissa") ) tulosta("On kissa");
```

10.4 Bittitason operaattorit

Yksi C–kielen vahvoista piirteistä erityisesti alemman tason ohjelmoinnissa on mahdollisuus käyttää bittitason operaattoreita.

Loogisia operaattoreita `&&`,`||` ja `!` ei pidä sotkea vastaaviin bittitason operaattoreihin:

```
&   bittitason AND
|   bittitason OR
^   bittitason XOR
~   bittitason NOT
<<  rullaus vasemmalle, 0 sisään oikealta
>>  rullaus oikealle, 0 sisään vasemmalta (unsigned int ja int >=0)
                   , voi tulla 0 tai 1 sisään vasemmalta (int joka <0)
                     (laiteriippuva, esim. Turbo C:ssä tulee 1).
```

Bittitason operaattoreita voidaan käyttää vain kokonaisluvuiksi muuttuviin operandeihin.

Operaattoreiden toimintaa voidaan kuvata seuraavasti. Olkoon meillä sijoitukset `a=5;` `b=14;`. Kuvitellaan kokonaisluvut tilapäisesti 8 bitin mittaisiksi (oikeasti yleensä 16 tai 32 bittiä):

	Binäärisenä	desim.	
a	0000 0101	5	
b	0000 1110	14	
a & b	0000 0100	4	
a | b	0000 1111	15	
a ^ b	0000 1011	11	
~a	1111 1010	-6	
a<<2	0001 0100	20	
b>>3	0000 0001	1	
a && b	0000 0001	1	Toimii vain C:ssä
a || b	0000 0001	1	Toimii vain C:ssä
!a	0000 0000	0	Toimii vain C:ssä

Huomautus! Tyypillinen ohjelmointivirhe on sotkea keskenään loogiset ja bittitason operaattorit. Javassa onneksi kääntäjä tekee tämän vaikeammaksi.

Tehtävä 10.5 Loogiset/bittitason operaattorit

Mitä tulostaa seuraava ohjelman osa.

```
int a=5, b=2;
if ( a != 0 && b != 0 ) tulosta("On ne!");
if ( (a&b) != 0)  tulosta("Ei ne ookkaan!");
if ( a != 0 ) tulosta("a on!");
if ( ~b != 0) tulosta("b ehkä on!");
if ( !(b == 0) ) tulosta("b ei ole!");
```

Tehtävä 10.6 Luku parilliseksi

Kirjoita funktio `parilliseksi`, joka palauttaa parametrinään olevan kokonaisluvun pienemmäksi parilliseksi luvuksi "katkaistuna". Eli esim. 3 –> 2. 5 –> 4. 4 –> 4.

10.5 `if` – `else` –rakenne

`if` –lauseesta on myös versio, jossa jotakin voidaan tehdä ehdon ollessa epätosi:

```
if ( ehto ) ylause1;
else ylause2;
```

Jälleen, mikäli jommassa kummassa osassa tarvitaan useampia lauseita, suljetaan lausejoukko lausesuluilla. Tosin kannattaa taas harkita lausesulkujen käyttöä aina myös yhdenkin lauseen tapauksessa.

```
if ( a < 5 ) tulosta("a alle viisi!");
else tulosta("a vähintään viisi!");

// Eri riville:
if ( a < 5 )
  tulosta("a alle viisi!");
else
  tulosta("a vähintään viisi!");

// Lausesulkujen käyttö:
if ( a < 5 ) {
  tulosta("a alle viisi!");
}
else {
  tulosta("a vähintään viisi!");
}

// Seuraavaa tyyliä käytetään myös usein:
if ( a < 5 ) {
  tulosta("a alle viisi!");
} else {
  tulosta("a vähintään viisi!");
}
```

10.5.1 Sisäkkäiset `if`–lauseet

Meillä oli aikaisemmin tehtävänä kirjoittaa funktio, joka palauttaa toisen asteen yhtälön $ax^2+bx+c=0$ toisen juuren. Tällöin oletuksena oli, että `a<>0` ja `D>=0`. Mikäli ratkaisukaavaa sovelletaan sellaisenaan ja `a=0` tai `D<0`, niin tällöin ohjelman suoritus päättyy ajonaikaiseen virheeseen.

Voisimme muuttaa tehtävän määrittelyä siten, että kumpikin juuri pitää palauttaa ja funktion nimessä palautetaan tieto siitä, tuliko ratkaisussa virhe, eli jollei juuret olekaan reaalisia.

```
if ( a != 0 ) {
  D = b*b - 4*a*c;
  if ( D > 0 ) {
    ...
  }
  else {
    ...
  }
}
else {
  ...
}
```

Tosin yhtälö pystytään mahdollisesti ratkaisemaan myös kun `a==0`. Tällöin tehtävä jakautuu useisiin eri tilanteisiin kertoimien `a`,`b` ja `c` eri kombinaatioiden mukaan:

a	b	c	D	yhtälön muoto	juuret reaalisia	x1	x2
0	0	0	?	0 = 0	juu	0	0
0	0	c	?	c = 0	ei	0	0
0	b	?	?	bx - c = 0	juu	-c/b	-c/b
a	?	?	>=0	ax^2 + bx + c = 0	juu	(-b-SD)/2a	(-b+SD)/2a
a	?	?	<0	- " -	ei		

Algoritmiksi kirjoitettuna tästä seuraisi:

```
1.  Jos a=0, niin
      Jos b=0
         Jos c=0 yhtälö on muotoa 0=0 joka on aina tosi
           palautetaan vaikkapa x1=x2 =0
         muuten (eli c<>0) yhtälö on muotoa c=0 joka on
           aina epätosi, palautetaan virhe
      muuten (eli b<>0) yhtälö on muotoa bx=c
          joten voidaan palauttaa vaikkapa x1=x1=-c/b
2.  Jos a<>0, niin
      Jos D>=0 kyseessä aito 2. asteen yhtälö ja käytetään
        ratkaisukaavaa
      muuten (eli D<0) ovat juuret imaginaarisia
```

Funktio ja sen testiohjelma voisi olla esimerkiksi seuraavanlainen:

java-silm\P2_2.java - esimerkki 2. asteen yhtälön ratkaisemiseta

```
/**
 * Ohjelmalla testataan 2. asteen polynomin juurien etsimistä
 * @author Vesa Lappalainen
 * @version 1.0, 16.02.2003
 */
public class P2_2 {

  public static void testi(double a, double b, double c) {
    Polynomi2 p = new Polynomi2(a,b,c);
    System.out.print("Polynomi: " + p);
    if ( p.getReaalijuuria() <= 0 ) {
      System.out.println("  Ei yhtään reaalijuuria! ");
      return;
    }
    System.out.print("  juuret: ");
    System.out.print("x1 = " + p.getX1() + " => P(x1) = " + p.f(p.getX1()) );
    System.out.print("  ja  ");
    System.out.print("x2 = " + p.getX2() + " => P(x2) = " + p.f(p.getX2()) );
    System.out.println();
  }
```

```
  public static void main(String[] args)  {
    testi(1,2,1);
    testi(2,1,0);
    testi(1,-2,1);
    testi(2,-1,0);
    testi(2,1,1);
    testi(2,0,0);
    testi(0,2,1);
    testi(0,0,1);
  }
}



/**
 * Luokka toisen asteen polynomille ja sen nollakohdille
 * @author Vesa Lappalainen
 * @version 1.0, 16.02.2003
 */
class Polynomi2 {

  private double a,b,c,x1,x2;
  private int reaalijuuria;

  public Polynomi2(double a, double b, double c) {
    this.a = a; this.b = b; this.c = c;
    reaalijuuria = ratkaise_2_asteen_yhtalo();
  }

  private int ratkaise_2_asteen_yhtalo() {
    double D,SD;
    x1 = x2 = 0;
    if ( a==0 ) {                       /*       bx + c = 0 */
      if ( b==0 ) {                     /*            c = 0 */
        if ( c==0 ) {                   /*            0 = 0 */
          return 1;                     /* id. tosi         */
        }                   /* c==0 */
        else {              /* c!=0 */  /*       0 != c = 0 */
          return 0;                     /* Aina epät.       */
        }                   /* c!=0 */
      }                     /* b==0 */
      else {                /* b!=0 */  /*       bx + c = 0 */
        x1 = x2 = -c/b;
        return 1;
      }                     /* b!=0 */
    }                       /* a==0 */
    else {                  /* a!=0 */  /* axx + bx + c = 0 */
      D = b*b - 4*a*c;
      if ( D>=0 ) {                     /* Reaaliset juuret */
        SD  = Math.sqrt(D);
        x1 = (-b-SD)/(2*a);
        x2 = (-b+SD)/(2*a);
        return 2;
      }                     /* D>=0 */
      else {                            /* Imag. juuret     */
        return -1;
      }                     /* D<0  */
    }                       /* a!=0 */
  }

  public static double P2(double x, double a, double b, double c) {
    return (a*x*x + b*x + c);
  }

  public double f(double x) { return P2(x,a,b,c); }
  public double getX1() { return x1; }
  public double getX2() { return x2; }
  public int getReaalijuuria() { return reaalijuuria; }

  public String toString() { return a + "x^2 + " + b + "x + " + c; }

}
```

Edellinen metodi `ratkaise_2_asteen_yhtalo` on äärimmäinen esimerkki sisäkkäisistä `if`–lauseista. Jälkeenpäin sen luettavuus on erittäin heikko ja myös kirjoittaminen hieman epävarmaa. Parempi kokonaisuus saataisiin lohkomalla tehtävää pienempiin osasiin aliohjelmien tai makrojen avulla.

Sisäkkäisten `if`–lauseiden kirjoittamista voidaan helpottaa kirjoittamalla niitä sisenevästi, eli aloittamalla ensin tekstistä:

```
if ( a == 0 ) {                        /*       bx + c = 0 */
}                       /* a==0 */
else {                                 /* axx + bx + c = 0 */
  D = b*b - 4*a*c;
}                       /* a!=0 */
```

Sitten täydennetään vastaavalla ajatuksella sekä `if`–osan että `else`–osan toiminta.

Jos funktiosta karsitaan kaikki ylimääräinen (kommentit ja ylimääräiset lausesulut) pois, saamme seuraavan näköisen kokonaisuuden:

java-silm\P2_21.java - karsittu versio 2. asteen yhtälöstä

```
private int ratkaise_2_asteen_yhtalo() {
  double D,SD;
  x1 = x2 = 0;
  if ( a == 0 )
    if ( b == 0 ) {
      if ( c == 0 ) return 1;
      else return 0;
    }
    else {
      x1 = x2 = -c/b;
      return 1;
    }
  else {
    D = b*b - 4*a*c;
    if ( D >= 0 ) {
      SD  = Math.sqrt(D);
      x1 = (-b-SD)/(2*a);
      x2 = (-b+SD)/(2*a);
      return 2;
    }
    else return 0;
  }
}
```

Joskus kannattaa harkita olisiko luettavuuden kannalta paras esitystapa sellainen, että käsitellään "normaaleimmat" tapaukset ensin:

java-silm\P2_2n.java - normaalit tapaukset ensin ratkaisussa

```
  private int ratkaise_2_asteen_yhtalo() {
    double D,SD;
    x1 = x2 = 0;
    if ( a != 0 ) {
      D = b*b - 4*a*c;
      if ( D >= 0 ) {
        SD  = Math.sqrt(D);
        x1 = (-b-SD)/(2*a);
        x2 = (-b+SD)/(2*a);
        return 2;
      }
      else return -1;
    }
    else /* a==0 */
      if ( b != 0 ) {
        x1 = x2 = c/b;
        return 1;
      }
      else { /* a==0, b==0 */
        if ( c == 0 ) return 1;
        else return 0;
      }
  }
```

Usein aliohjelman `return`–lauseen ansiosta `else` osat voidaan jättää poiskin:

java-silm\P2_2r.java - else -osat pois

```
  private int ratkaise_2_asteen_yhtalo() {
    double D,SD;
    x1 = x2 = 0;
    if ( a == 0 ) {
      if ( b == 0 ) {
        if ( c == 0 ) return 1;
        return 0;
      }
      x1 = x2 = -c/b;
      return 1;
    }

    D = b*b - 4*a*c;
    if ( D < 0 ) return -1;

    SD  = Math.sqrt(D);
    x1 = (-b-SD)/(2*a);
    x2 = (-b+SD)/(2*a);
    return 2;
  }
```

Edellä oli useita eri ratkaisuja saman ongelman käsittelemiseksi. Liika kommenttien määrä saattaa myös sekoittaa luettavuutta kuten 1. esimerkissä. Toisaalta liian vähillä kommenteilla ei ehkä kirjoittaja itsekään muista jälkeenpäin mitä tehtiin ja miten. Jokainen valitkoon edellä olevista itselleen sopivimman kultaisen keskitien.

Huomattakoon vielä lopuksi, että rakenne

```
if ( c == 0 ) return true;
else return false;
```

voitaisiin korvata rakenteella

```
return ( c != 0 );
```

Tehtävä 10.7 else –osat pois

Kirjoita `ratkaise_2_asteen_yhtalo P2_2n.java` ilman `else` –osia.

10.5.2 Useat peräkkäiset ehdot

Vaikka rakenne

```
if (ehto1) lause1;
else
  if (ehto2) lause2;
  else
    if (ehto3) lause3;
    else lause4;
```

jossain mallissa sisennetäänkin ylläkuvatulla tavalla, on ajatus useimmiten lähempänä seuraavaa sisennystä:

java-silm\Postimaksu.java - esimerkki samanarvoisista ehtolauseista

```
  static double postimaksu(double paino)
  {
    if      ( paino < 50 )   return  0.60;
    else if ( paino < 100 )  return  0.90;
    else if ( paino < 250 )  return  1.30;
    else if ( paino < 500 )  return  2.10;
    else if ( paino < 1000 ) return  3.50;
    else if ( paino < 2000 ) return  5.50;
    else                     return  0.00;
  }
```

Sovimme siis, että rakenne onkin muotoa:

```
if      ( ehto1 ) lause1
else if ( ehto2 ) lause2
else if ( ehto3 ) lause3
else              lause4
```

Tehtävä 10.8 elset pois

Voiko aliohjelmasta `postimaksu` jättää `if`-lauseiden `else`-osat pois? Päteekö väittämä yleisesti?

Tehtävä 10.9 Lääni

Kirjoita aliohjelma

`void laani(string rekisteri)`

joka tulostaa missä läänissä auto on rekisteröity. (Ennen oli Suomessa monta lääniä ja rekisterinumeron 1. kirjain määräsi missä läänissä auto oli rekisteröity).

Kirjaimen yhtäsuuruutta testataan `if ( c == 'a' ) ...`

Merkkijonon 1. merkki saadaan `c = rekisteri[0];` edellyttäen tietysti että `rekisteri != ""`.

Tehtävä 10.10 if–else

Mitä ovat muuttujien arvot seuraavien ohjelmanpätkien jälkeen (pöytätesti!)?

```
if (a<5)
/*1*/ a=1; b=2; c=3;
  b=3;
  a=6;
c=7;
```

```
if (a<0) a=3; else
/*5*/ a=1; b=2; c=3;
if (a>2) b=3; a=6;
c=7;
```

```
/*2*/ a=1; b=2; c=3;
if (a<5) b=3; a=6; c=7;
```

```
/*6*/ a=1; b=2; c=3;
if (a<-5) if (a<0) a=6;
else a=2; c=7;
```

```
/*3*/ a=1; b=2; c=3;
if (a<5) {b=3; a=6;}
c=7;
```

```
/*7*/ a=1; b=2; c=3;
if (a<-5) b=3;
  if (a<5) a=6;
else a=2; c=7;
```

```
/*4*/ a=1; b=2; c=3; if (a<5)
b=3; else { a=6; c=7; }
```

```
/*8*/ a=1; b=2; c=3;
if (a<0) a=3; else;
if (a>2) b=3; a=6;
c=7;
```

Sisennä ohjelmanpätkät "asianmukaisesti".

10.6 `do-while` –silmukka

Aikaisemmin olemme tutustuneet erääseen algoritmiin selvittää onko luku alkuluku vai ei. Koska algoritmi on valmis, voimme kirjoittaa vastaavan ohjelman (% –operaattori antaa jakojäännöksen, `10 % 3 == 1`):

java-alk\Alkuluku.java - testataan onko luku alkuluku

```
**
 *  Ohjelmalla testataan onko_alkuluku-aliohjelmaa
 *  @author Vesa Lappalainen
 *  @version 1.0, 17.01.2002
 */
public class Alkuluku {

  /**
   * Aliohjelmalla tutkitaan onko parametrina tuotu
   * luku alkuluku vai ei<br>
   * Algoritmi: Jaetaan tutkittavaa lukua jakajilla 2,3,5,7...luku/2.
   * Jos jokin jako menee tasan, niin ei alkuluku:
   * @param luku tutkittava luku
   * @return tieto siitä, onko luku alkuluku vai ei
   */
  public static String onko_alkuluku(int luku)
  {
    int jakaja=2, kasvatus=1;
    if ( luku == 2 ) return "alkuluku";

    do {
      int jakojaannos = luku % jakaja;
      if ( jakojaannos == 0 )
        return "jaollinen";
      jakaja += kasvatus;
      kasvatus = 2;
    } while ( jakaja < luku/2 );

    return "alkuluku";
  }
```

```
  public static void main(String[] args) {
    String tulos;
    tulos = onko_alkuluku(25);
    System.out.println(tulos);
    tulos = onko_alkuluku(123);
    System.out.println(tulos);
    tulos = onko_alkuluku(7);
    System.out.println(tulos);
  }
}
```

Käytimme tässä silmukkaa:

```
do
  lause
while (ehto);
```

Koska esimerkin silmukassa oli useita suoritettavia lauseita, oli lauseet suljettu lausesuluilla. Jälleen voi olla hyvä tapa käyttää AINA lausesulkuja.

Huomautus! Silmukoiden kanssa on syytä olla tarkkana sekä 1. kierroksen että viimeisen kierroksen kanssa. Myös silmukan lopetusehdon on syytä muuttua silmukan suorituksen aikana.

Eräs tyypillinen esimerkki `do-while` silmukan käytöstä olisi seuraava:

java-silm\Dowhile.java - lukujen lukeminen kunnes halutulla välillä

```
import fi.jyu.mit.ohj2.Syotto;
/**
 * Ohjelmalla luetaan luk, kunnes se on halutulla välillä
 * @author Vesa Lappalainen
 * @version 1.0, 07.02.2003
 */
public class Dowhile {

  public static void main(String[] args)  {
    int luku;
    do {
      luku = Syotto.kysy("Anna luku väliltä [0-20]",0);
    } while ( luku < 0 || 20 < luku );
    System.out.println("Annoit luvun " + luku);

  }
}
```

10.7 `while` –silmukka

`do-while` –silmukka suoritetaan aina vähintään 1. kerran. Joskus on tarpeen silmukka, jonka runkoa ei suoriteta yhtään kertaa. Muutamme edellisen esimerkkimme käyttämään `while` –silmukkaa:

```
while ( ehto ) lause
```

Muutamme samalla algoritmia siten, että 2:lla jaolliset käsitellään erikoistapauksena. Näin pääsemme eroon "inhottavasta" `kasvatus`–muuttujasta.

java-silm\Alkuluku2.java - alkulukutesti while-silmukalla

```
public static int pienin_jakaja(int luku)
{
  int jakaja=3;
  if ( luku == 2 ) return 1;
  if ( luku % 2 == 0 ) return 2;

  while ( jakaja < luku/2 ) {
    if ( luku % jakaja == 0 ) return jakaja;
    jakaja += 2;
  }

  return 1;
}
```

10.8 `for` –silmukka, tavallisin muoto

Eräs C–kielen hienoimmista rakenteista on `for`–silmukka. Usein C–hakkereiden tavoite on saada kirjoitettua koko ohjelma yhteen `for`–silmukkaan. Tätä ei tietenkään tarvitse tavoitella, mutta se osoittaa `for`–silmukan mahdollisuuksia.

Tyypillisesti `for`–silmukkaa käytetään silloin, kun silmukan kierrosten lukumäärä on ennalta tunnettu:

java-silm\Valinsum.java - esimerkki for-silmukasta

```
/**
 * Lasketaan yhteen luvut 1..ylaraja
 * @param ylaraja summan yläraja
 * @return summa
 */
public static int valin_summa(int ylaraja)
{
  int i,summa=0;
  for (i=1; i<=ylaraja; i++)
    summa += i;
  return summa;
}
```

Tehtävä 10.11 valin_summa

Muuta `valin_summa` –aliohjelmaa siten, että myös alaraja viedään parametrinä. Kirjoita pääohjelma, jolla toiminta voidaan testata.

Käytännössä tällaisia silmukoita ei saa tehdä, koska ongelman ratkaisuun on valmis kaava. Millainen?

10.9 Java–kielen lauseista

10.9.1 Sijoitusoperaattori =

Olemme tutustuneet jo Java–kielen "normaaliin" sijoitusoperaattoriin =.

Sen ansiosta, että myös sijoitus palauttaa arvon, pystyimme tekemään mm seuraavia temppuja:

```
if ( (b=a) != 0 ) ... /* Suoritetaan jos a!=0 */
a = b = c = 0;
```

Sijoitus monelle muuttujalle yhtäaikaa onnistuu, koska sijoitus jäsentyy seuraavasti:

```
1.  a = ( b = (c = 0) ); -   sijoitus c=0 palauttaa arvon 0
2.  a = ( b =    0 );    -   sijoitus b=0 palauttaa arvon 0
3.  a =     0;
```

10.9.2 Sijoitusoperaattori +=

`valin_summa` aliohjelmassa meillä esiintyi myös kaksi uutta sijoitusoperaattoria, jotka ovat lyhenteitä tavallisille sijoituksille:

lyhenne	tavallinen sijoitus
`summa += i;`	`summa = summa + i;`
`i++`	`i = i + 1;`

+= sijoituksessa + voidaan korvata millä tahansa operaattoreista:

```
+   -   *   /   %   <<   >>   ^   &   |
```

Esimerkiksi luvun kertominen ja jakaminen 10:llä voitaisiin suorittaa:

```
luku *= 10;
luku /= 10;
```

Siis `muuttuja O= operandi` voidaan ajatella korvattavaksi seuraavasti:

```
0.  laita sulut operandin ympärille
      muuttuja O= (operandi)
1.  kirjoita muuttujan nimi kahteen kertaan
      muuttuja muuttuja O= (operandi)
2.  siirrä = -merkki muuttujien nimien väliin
      muuttuja = muuttuja O (operandi)
```

Tehtävä 10.12 +=

Mitä ovat muuttujien arvot seuraavien sijoitusten jälkeen:

```
int a=10,b=3,c=5;
a %= b;
b *= a+c;
b >>= 2;
```

10.9.3 Lisäysoperaattori ++

Erittäin tyypillisiä C–operaattoreita ovat ++ ja−−.

Nämä operaattorit lisäävät tai vähentävät operandin arvoa yhdellä. Operandin tyypin tulee olla numeerinen tai osoitin.

Operandeista on kaksi eri versiota: esilisäys ja jälkilisäys.

lyhenne	vastaa lauseita
a = i++;	a = i; i = i+1;
a = i–	a = i; i = i-1;
a = ++i;	i = i+1; a = i;
a = --i;	i = i-1; a = i;

Vaikka C–hakkerit rakentavatkin mitä ihmeellisimpiä kokonaisuuksia ++ –operaattorin avulla, kannattaa operaattorin liikaa käyttöä välttää. Esimerkiksi lauseet joissa esiintyy samalla kertaa useampia lisäyksiä samalle muuttujalle, saattavat olla jopa määrittelemättömiä:

java-silm\Plusplus.java - esimerkki ei-yksikäsitteisestä ++ operaattorin käytöstä

```
/**
 * Esimerkki epäselvästä ++-operaattorin käytöstä
 * @author Vesa Lappalainen
 * @version 1.0, 16.02.2003
 */
public class Plusplus {

  public static void main(String[] args)  {
    double i=1.0,a;
    a = i++/i++;
    System.out.println("(a = " + a + ", i = " + i);
  }
}
```

Ohjelma saattaa Java–kääntäjän toteutuksesta riippuen tulostaa mitä tahansa seuraavista `a`:n ja `i`:in kombinaatioista:

```
a: 0.5 1.0 2.0
i: 2.0 3.0
```

Aluksi ++ –operaattoria kannattaa ehkä käyttää vain yksinäisenä lauseena lisäämään (tai vähentämään) muuttujan arvoa.

```
i++;
```

Lisäysoperaattoria EI PIDÄ käyttää jos muuttuja johon lisäysoperaattori kohdistuu, esiintyy samassa lausekkeessa useammin kuin kerran.

Kiellettyjä on siis esimerkiksi:

```
a = ++i + i*i;
ali(i++,i);
```

10.10 for –silmukka, yleinen muoto

Yleensä ohjelmointikielissä `for`–silmukka on varattu juuri siihen tarkoitukseen, kuin ensimmäinen esimerkkimmekin; tasan tietyn kierrosmäärän tekemiseen.

Java–kielen `for`–silmukka on kuitenkin yleisempi:

```
/*     1.             2.  5.          4.  7.           3.  6.  */
for (alustus_lauseet; suoritus_ehto; kasvatus_lauseet) lause;
```

`for`–silmukka vastaa melkein `while`–silmukkaa (ero tulee `continue`–lauseen käyttäytymisessä):

```
alustus_lauseet;                /* 1.     */
while ( suoritus_ehto ) {       /* 2.  5. */
  lause;                        /* 3.  6. */
  kasvatus_lauseet;             /* 4.  7. */
}
```

Mikäli esimerkiksi alustuslauseita on useita, erotetaan ne toisistaan pilkulla:

java-silm\Valinsum.java - useita alustuslauseita for-silmukassa

```
  public static int valin_summa_2(int ylaraja) {
    int i,summa;
    for (summa=0, i=1; i<=ylaraja; i++)
      summa += i;
    return summa;
  }
```

Erittäin C:mäinen tapa tehdä yhteenlasku olisi:

java-silm\Valinsum.java - C:mäinen silmukka

```
  public static int valin_summa_3(int i) {
    int s;
    for (s=0; i >= 0; s += i--);
    return s;
  }
```

Tämä viimeinen esimerkki on juuri niitä C–hakkereiden suosikkeja, joita ehkä kannattaa osin vältellä.

Tehtävä 10.13 1+2+..+i

Miksi `valin_summa_3` laskee yhteen luvut 1..i?

10.11 `break` ja `continue`

10.11.1 `break`

Joskus kesken silmukan tulee vastaan tilanne, jossa silmukan suoritus haluttaisiin keskeyttää. Tällöin voidaan käyttää C–kielen `break`–lausetta, joka katkaisee **sisimmän** silmukan suorituksen.

java-silm\Break.java - silmukan katkaisu keskeltä

```
private static void break_testi1() {
  int summa=0,luku;
  System.out.println("Anna lukuja.  Summaan niitä kunnes annat 0 tai summa>20");
  do {
    luku = Syotto.kysy("Summa on " + summa + ".  Anna luku",0);
    if ( luku == 0 ) break;
    summa += luku;
  } while ( summa <= 20 );
  System.out.println("Lukujen summa on " + summa);
}
```

Koska 0:lla lisääminen ei muuta summaa, olisi tietenkin do-`while` –silmukan ehto voitu kirjoittaa muodossa

```
    do {
      luku = Syotto.kysy("Summa on " + summa + ".  Anna luku",0);
      summa += luku;
    } while ( luku != 0 && summa <= 20 );
```

mutta aina ei voida `break` –lausetta korvata näin yksinkertaisesti. Perus `break` –lauseen vika on lähinnä siinä, ettei siitä suoraan nähdä sisäkkäisten silmukoiden tapauksessa sitä, mihin saakka suoritus katkeaa. Epäselvissä tapauksissa silmukan katkaisu voidaan hoitaa nimeämällä silmukat ja ilmoittamalla `break`-lauseessa mikä silmukka katkaistaan:

java-silm\Break.java - ulomman silmukan katkaisu keskeltä

```
  private static void break_testi3() {
    int valisumma, loppusumma = 0,luku;
    System.out.println("Anna lukuja.");
    System.out.println("Summaan niitä kunnes annat 99.");
    System.out.println("Antamalla 0, näet välisumman");
    System.out.println("Välisumman näet myös jos välisumma > 20");
    laskeloppusummaa: do {
      valisumma = 0;
      do {
        luku = Syotto.kysy("Anna luku",0);
        if ( luku == 0 ) break;
        if ( luku == 99 ) break laskeloppusummaa;
        valisumma += luku;
      } while ( luku != 0 && valisumma <= 20 );
      System.out.println("Lukujen välisumma on " + valisumma);
      loppusumma += valisumma;
      System.out.println("Kaikkien summa on " + loppusumma);
    } while ( loppusumma < 100 );
    System.out.println("Lukujen loppusumma on " + loppusumma);
  }
```

Silmukka voidaan katkaista tietenkin myös muuttamalla silmukan lopetusehtoon vaikuttavia muuttujia. Varsinkin `for`–lauseen tapauksessa silmukan indeksin arvon muuttaminen muualla kuin kasvatus–lauseessa on todella väkivaltaista ja rumaa, eikä tällaista pidä mennä tekemään.

Hyvin usein aliohjelmassa `break` voidaan korvata `return`-lauseella.

Lisäksi näkyviä sisäkkäisiä silmukoita voidaan välttää tekemällä sisäsilmukasta oma aliohjelma:

```
while ( ulkoehto ) {
  while ( sisaehto ) {
    hommia();
  }
}
```

Eli sisäkkäisten silmukoiden tilalle kirjoitetaan:

```
void sisahommat() {
  while ( sisaehto ) {
    hommia();
  }
}
...
while ( ulkoehto ) {
  sisahommat();
}
```

Tehtävä 10.14 Tarvitaanko sisäkkäisiä silmukoita?

Tarvitaanko aliohjelmassa `break_testi3` todella sisäkkäisiä silmukoita? Esitä ratkaisu jossa on vain yksi silmukka.

10.11.2 `continue`

Vastaavasti saattaa tulla tilanteita, jolloin itse silmukan suoritusta ei haluta katkaista, mutta menossa oleva kierros halutaan lopettaa. Tällöin `continue` –lauseella voidaan suoritus siirtää suoraan silmukan loppuun ja näin lopettaa tämän kierroksen suoritus:

java-silm\Continue.java - silmukan lopun ohittaminen

```
/**
 * Esitellään continue-lauseen käyttöä
 * @author Vesa Lappalainen
 * @version 1.0, 07.02.2003
 */
public class Continue {

  public static void main(String[] args)  {
    int alku= -5, loppu=5,i;
    double inv_i;
    System.out.println("Tulostan lukujen " + alku + " - " + loppu +
                       "käänteisluvut");
    for (i = alku; i<=loppu; i++ ) {
      if ( i == 0 ) continue;
      inv_i = 1.0/i;
      System.out.println(i + ":n käänteisluku on " + inv_i);
    }
  }
}
```

Vastaavasti myös `continue`:n kanssa voi käyttää nimettyä silmukkaa, jos pitääkin siirtyä jatkamaan muuta kuin sisintä silmukkaa.

Tehtävä 10.15 continuen korvaaminen

Kirjoita käänteislukujen tulostusohjelma ilman `continue`–lausetta.

Tehtävä 10.16 Eri silmukoiden vertailu

Kirjoita lukujen alaraja-yläraja summausfunktio käyttäen

a) `while` -lausetta
b) `do-while` -lausetta
c) `goto` –lausetta

Muista, että alaraja saattaa olla suurempi kuin yläraja, eli summa väliltä `[3,0]` on `0`!

10.12 `switch` –valintalause

Jäsenrekisteriohjelmamme päävalinta olisi näppärintä toteuttaa `switch` –lauseella:

menut_3\Naytto.java - päävalinta switch -lauseella

```
/**
 * Silmukka jossa odotetaan näppäint ja suoritetaan vastaava toiminto.
 * 0:n painaminen lopettaa silmukan ja palaa kutsuvaan ohjelmaan.
 * @return palauttaa 0 jos kaikki meni hyvin, 1 jos tuli virhe
 */
public int paavalinta() {
  char nappain;

  while ( true ) {

    paamenu();

    nappain = IO.odota_nappain("?012345",IO.EI_OLETUSTA,IO.MERKKI_ISOKSI);

    switch (nappain) {
      case '?': avustus(nappain);               break;
      case '0': return 0;
      case '1': lisaa_uusi_jasen(nappain);      break;
      case '2': etsi_jasenen_tiedot(nappain);   break;
      case '3': tulosteet(nappain);             break;
      case '4': tietojen_korjailu(nappain);     break;
      case '5': paivita_jasenmaksuja(nappain);  break;
      default : tulosta("Näin ei voi käydä!"); return 1;
    }

  }
}
```

`switch` –lauseessa `case` osien lopuksi `break` on yleensä välttämätön. `break` estää suorittamasta seuraavia rivejä.

Joskus harvoin `break`in puuttumista voidaan käyttää hyväksi, mutta tällöin pitää olla todella tarkkana:

java-silm\Switch.java - switch, jossa break tahallaan jätetty pois

```
public static int switch_testi(int x,int operaatio) {
  switch (operaatio) {
    case 5:                 /* Operaatio 5 tekee saman kuin 4 */
    case 4: x *= 2; break;  /*            4 laskee x=2*x       */
    case 3: x += 2;         /*            3 laskee x=x+4       */
    case 2: x++;            /*            2 laskee x=x+2       */
    case 1: x++; break;     /*            1 laskee x=x+1       */
    default: x=0; break;    /* Muut nollaavat x:än             */
  }
  return x;
}
```

Lause `default` suoritetaan jos mikään `case`–osista ei ole täsmännyt (tai tietysti jos jokin `break` puuttuu). `default`–lauseen ei tarvitse olla viimeisenä, mutta tällöin vaaditaan taitavaa `break`in käyttöä, siis paras pitää `default` viimeisenä!

Yleistä `switch`–lausetta ei voi korvata joukolla `if`–lauseita käyttämättä `goto`–lausetta. Mikäli kuitenkin jokaisen `case` rakenteen perässä on `break`, voidaan `switch`– korvata sisäkkäisillä `if-else` –rakenteilla.

Tehtävä 10.17 switch –> if

Kirjoita `Switch.java` ohjelmanpätkä käyttäen `if`–rakenteita muuttamatta itse suoritettavia lauseita.

Tehtävä 10.18 Päävalinta

Kirjoita `paavalinta` käyttäen vain `if` ja `else` rakenteita.

Tehtävä 10.19 lääni, versio 2

Kirjoita `laani`–aliohjelma käyttäen `Switch`–rakennetta.

10.12.1 | ei toimi `switch` –lauseessa!

On huomattava, että jos halutaan suorittaa jokin `switch`–lauseen osista kahdella eri arvolla, EI voida käyttää rakennetta:

```
switch (operaatio) {  /* VÄÄRIN: */

  case 4 | 5: x *= 2; break;   /*    5 tai  4 laskee x=2*x       */
  case 3: x += 2;              /*           3 laskee x=x+4       */
  case 2: x++;                 /*           2 laskee x=x+2       */
  default: x=0; break;         /* Muut nollaavat x:än            */
}
```

☹

Kääntäjä ei tästä varoita, koska kaikki on aivan kieliopin mukaista. `4 | 5` on kahden bittilausekkeen OR eli `5`. Siis

```
case 4 | 8:
on sama kuin
case 12:
```

10.13 Ikuinen silmukka

Usein silmukat lipsahtavat tahottomasti sellaisiksi, ettei niistä koskaan päästä ulos. Ikuisen silmukan huomaa heti esimerkiksi siitä, ettei silmukan rungossa ole yhtään lausetta joka muuttaa silmukan ehdon totuusarvoa.

Joskus kuitenkin Java–kielessä tehdään tarkoituksella "ikuisia" –silmukoita:

```
for (;;) {
  ...
  if (lopetus_ehto) break;
  ...
}

while ( true ) {
  ...
  if (lopetus_ehto) break;
  ...
}

do {
  ...
  if (lopetus_ehto) break;
  ...
} while ( true );
```

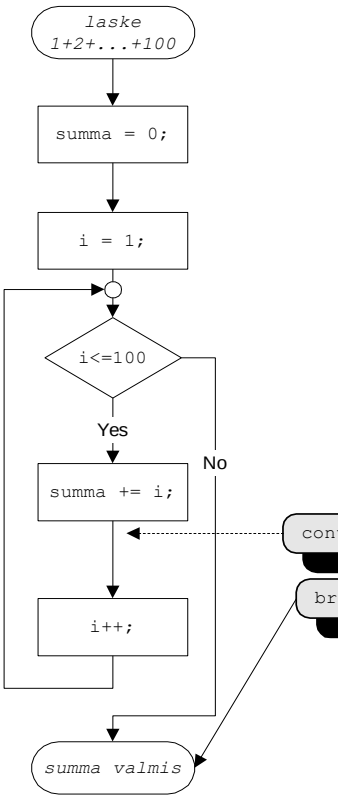
Näissä kahdessa ensimmäisessä korostuu silmukan ikuisuus. Viimeinen ei ole hyvä vaihtoehto.

Tällaiset ikuiset silmukat ovat hyväksyttävissä silloin, kun silmukan lopetusehto on luonnollisesti keskellä silmukkaa. Usein kuitenkin lauseiden uudelleen järjestelyllä lopetusehto voidaan sijoittaa silmukan alkuun tai loppuun, jolloin tavallinen `while–` , `do–while` – tai `for` –silmukka kelpaa.

10.13.1 Yhteenveto silmukoista

for

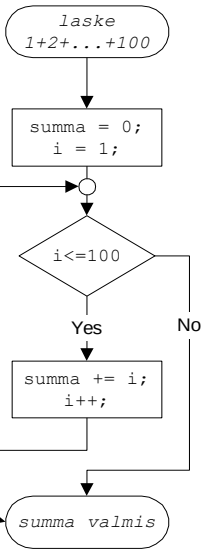
```
summa = 0;
for (i=1; i<=100; i++) {
  summa += i;
}
```




```
/* Huom järjestys:                    */
/*   1    2,5,8,11   4,7,10           */

for (i=1;    i<=3;     i++) {
  summa += i;                 /* 3,6,9 */
}
/* 1 -3*/ i=1;   1<=3 ? summa = 0+1;
/* 4 -6*/ i=1+1; 2<=3 ? summa = 1+2;
/* 7 -9*/ i=2+1; 3<=3 ? summa = 3+3;
/*10-11*/ i=3+1; 4<=3 ? ei => valmis
```

while

```
summa = 0;
i = 1;
while ( i <= 100 ) {
  summa += i;
  i++;
}
```

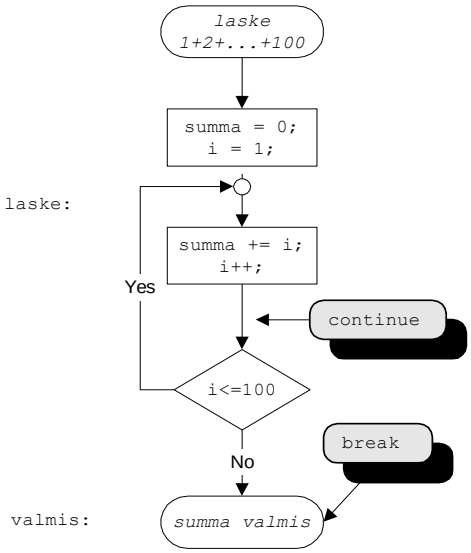



```
summa = 0;
i = 1;
laske:
  if ( i > 100 ) goto valmis
  summa += i;
  i++;
  goto laske;
valmis:
```

```
summa = 0;
i = 1;
laske: if ( i <= 100 ) {
  summa += i;
  i++;
  goto laske;
}
```

do-while

```
summa = 0;
i = 1;
do {
  summa += i;
  i++;
} while ( i <= 100 );
```




```
summa = 0;
i = 1;
laske:
  summa += i;
  i++;
if ( i <= 100 ) goto laske;
```

HUOM! Tässä esimerkki on vain silmukoiden esittämistä varten, oikeastihan lasku
1+2+3...+100
voidaan laskea ryhmittelemällä lasku uudelleen:
1+100 + 2+99 + 3+98 + ... + 50+51
= 101*50
eli

```
summa = 5050;
```

- **continue** on **goto** silmukan lopettavan sulun } vasemmalle puolella
- **break** on **goto** silmukan lopettavan sulun } oikealle puolelle
- sisäkkäisissä silmukoissa break ja continue on käytettävä varoen

Silmukan valinta:

- **for** -silmukka valitaan jos silmukan kierrosmäärä on "ennalta tiedossa", esim. taulukot
- **while**-silmukka valitaan jos for ei ole ilmeinen ja runkoa mahdollisesti ei suoriteta
- **do-while** -valitaan, mikäli runko on suoritettava vähintään 1. kerran
- joskus siistein rakenne saadaan **ikuisella silmukalla** (ehto esim true)

Käyttöesimerkkejä

```
// 3x3 matriisin alustus
    int r,s;
    for (r=0; r<3; r++)     // jokaiselle riville
      for (s=0; s<3; s++)   // jokainen sarake
        mat[r][s] = 0;      // nollataan
```

```
// Luvun kysyminen: ("ikuinen silmukka")
    int n;
    while ( true ) { /* 1 on aina tosi! */
      n = Syotto.kysy("Anna luku (1-10)",1);
      if ( 1 <= n && n <= 10 ) break;
      System.out.println(
        "Virhe: Luku ei sallitulla välillä!\n");
    }
```

11. Oliosuunnittelu

Kaikki korttiin kirjoittele
palaan pahvin piirrustele
lappuselle laita luokka
taakse tarpeet tarkastele.

Vastuut varmasti valitse
hommat huolella hajoita.
Apulaiset aatteleppa
kelle viestit viskomaksi.

Mitä tässä luvussa käsitellään?

- vaatimukset ohjelman toteutukselle
- olioiden etsiminen
- CRC–kortit
- kuka huolehtii harrastuksista?

11.1 Olio on luokan esiintymä

Ennen kuin voimme aloittaa ohjelman toteutuksen, pitää meidän suunnitella mitä luokkia ohjelmassamme tarvitaan. Ohjelmassa olevat oliot ovat sitten näiden luokkien esiintymiä (ilmentymiä, *instance*).

11.2 Tavoitteet

Asetamme ensin ohjelman toteutukselle ulkoisen toiminnan lisäksi tiettyjä lisätavoitteita:

- käyttöliittymä (tekstipohjainen vaiko ikkunoitu) on voitava muuttaa kohtuullisella ohjelmoinnilla
- ohjelma on voitava pienillä muutoksilla muuttaa muuksikin kuin jäsenrekisteriksi (puhelinluettelo, levyrekisteri)
- jäseneen voitava helposti lisätä kenttiä

11.3 Luokat

Tavoitteiden aikaansaamiseksi näyttäisi, että tarvitsemme ohjelmassa ainakin seuraavat kolme luokkaa:

- käyttöliittymää ylläpitävä luokka (`Naytto`)
- rekisteriä ylläpitävä luokka (`Kerho`)
- yksittäinen jäsen (`Jasen`)

Kullekin luokalle täytyy antaa selvät vastuualueet ja tieto siitä, miten kommunikoidaan muiden luokkien kanssa ja minkä luokan kanssa yleensäkään tarvitsee kommunikoida.

11.4 CRC–kortit

Timothy A. Budd [B] ehdottaa luokkasuunnittelun avuksi *CRC*-kortteja (*Class Responsibility Collaborator*, luokan vastuu ja avustajat). Kortti on 4"x6" (10 x15 cm) kooltaan ja se jaetaan 3 osaan: luokan nimi, vastuu (eli tehtävät) ja avustajat. Kortin

koko on perusteltu sillä, että se on riittävän iso, jotta luokan vastuualueet voidaan siihen kirjoittaa ja toisaalta jos vastuualueet ei mahdu korttiin, on luokka liian iso ja se pitää jakaa useammaksi osaluokaksi. Huomattakoon että luokkaa suunniteltaessa ei juurikaan oteta kantaa siihen, kuka luokkaa käyttää!

Korttien takapuolelle voidaan kirjoittaa luokan yksityiset tiedot, eli ne tiedot joita joudutaan käyttämään jotta luokka voi hoitaa sovitun vastuunsa.

CRC-kortteja on sitten tarkoitus tutkia työryhmän jäsenten kesken antamalla kortti aina yhdelle ryhmän jäsenelle joka näin voi tarkistaa saako hän ko. luokasta tarvitsemansa tiedot (kääntämättä korttia). Jollei saa, luokkia on vielä helppo muuttaa kun ohjelmaa ei ole kirjoitettu.

11.4.1 Jäsen-luokka (`Jasen`)

Luokan nimi: `Jasen`	**Avustajat:**
Vastuualueet: (- ei tiedä kerhosta, eikä käyttöliittymästä) - tietää jäsenen kentät (nimi, hetu, puhnro, jne.) - osaa tarkistaa tietyn kentän oikeellisuuden (syntaksin) - osaa muuttaa `|Ankka Aku|..|` - merkkijonon jäsenen tiedoiksi - osaa antaa merkkijonona i:n kentän tiedot - osaa laittaa merkkijonon i:neksi kentäksi	- merkkijonot

11.4.2 Kerho-luokka (`Kerho`)

Luokan nimi: `Kerho`	**Avustajat:**
Vastuualueet: - pitää yllä varsinaista rekisteriä, eli osaa lisätä ja poistaa jäsenen - lukee ja kirjoittaa kerhon tiedostoon - osaa etsiä ja lajitella	- `Jasen`

11.4.3 Käyttöliittymä-luokka (`Naytto`)

Luokan nimi: `Naytto`	**Avustajat:**
Vastuualueet: - hoitaa kaiken näyttöön tulevan tekstin - hoitaa kaiken tiedon pyytämisen käyttäjältä (- ei tiedä kerhon eikä jäsenen yksityiskohtia)	- `Jasen` - `Kerho`

11.4.4 Luokkajaon tarkastelua

Miksi näyttö ei saa tietää jäsenen yksityiskohtia? Jos näyttö tietäisi jäsenen yksityiskohdat, pitäisi myös `Naytto`–luokkaa muuttaa kun muutetaan jotakin jäsenen yksityiskohtaa, eli esimerkiksi lisätään fax–numero. Käytännössä näytön pitää kuitenkin saada tämä muutos tietoonsa. Miten?

Näyttö voi esimerkiksi aina kysyä jäseneltä montako kenttää tällä on. Samoin näyttö voi pyytää jäsentä antamaan 1. kentän (nimi) merkkijonona, 2. kentän merkkijonona jne. Näin voidaan tulostaa jäsenen tiedot näyttöön tietämättä tarkkaan mitä kenttiä jäsenessä on.

Entä tietojen lukeminen? Näyttö voi myös kysyä jäseneltä sen tekstin, joka täytyy käyttäjälle kirjoittaa, kun pyydetään käyttäjää antamaan 3. kentän tiedot ("Katuosoite"). Tämän jälkeen näyttö voi tulostaa tämän tekstin näyttöön ja jäädä odottamaan käyttäjältä merkkijonoa. Kun käyttäjä antaa merkkijonon, pyydetään jäsentä muuttamaan annettu merkkijono 3:nen kentän tiedoiksi.

Aikanaan saattaa tulla vastaan tilanne, jossa on edullista jakaa näyttöluokka useampaan alaluokkaan, eli esim. yleinen näyttö (`Naytto`), kerhon näyttö (`Kerhon_naytto`) ja jäsenen näyttö (`Jasenen_naytto`).

Lisäksi yksittäisten kenttien käsittelyä voi auttaa kenttä-luokan käyttö. Peruskenttäluokasta voidaan periä erikoistuneita kenttäluokkia. Tiedon säilyttämisen apuna kerholuokalla voi olla tietorakenne-luokka. Etsiminen voidaan ehkä jättää etsimis- ja selailu -luokan tehtäväksi. Alkuun päästään kuitenkin suunnitelman mukaisella kolmella luokalla.

11.5 Harrastukset mukaan

Entä sitten kun haluamme lisätä kerholaisille harrastuksia. Ainakin tarvitaan `Harrastus`–luokka lisää. Kuka huolehtii harrastuksista?

Jos valinta tehdään valitun tiedostomuodon mukaan, niin mahdollisuuksia on:

11.5.1 "Oliomalli"

Oletetaan aluksi että Jäsen huolehtii harrastuksistaan. Tällöin Kerho ei tarvitse mitään muutoksia ja Näyttökin vain sen verran, että tietää kysellä ja tulostaa Jäsenelle tulleita lisäominaisuuksia. Jäsenen ominaisuuksiin lisätään harrastuksien ylläpito vastaavasti kuin Kerho ylläpiti jäsenistöä.

11.5.2 Relaatiomalli

Jos valitaan relaatiomallin mukainen tiedostorakenne, niin ehkä myös luokat kannattaa suunnitella vastaavasti. Tässä mallissa Jäsen ei tiedä mitään harrastuksistaan ja `Jasen` pysyykin muuttumattomana. `Harrastus` tekee täsmälleen samat asiat kuin `Jasen`, paitsi tietenkin omille ominaisuuksilleen.

Jos mahdollisimman paljon vastuuta harrastusten ylläpidosta annetaan Kerholle, huomataan että toistetaan samat ominaisuudet, joita kirjoitettiin jo jäsenistöä varten. Tämän vuoksi Kerhon roolia kannattaakin hieman selventää siten, että tietorakenteiden ylläpitoa varten tehdään omat luokat ja Kerho komentaa näitä luokkia.

11.5.3 Harrastus-luokka (`Harrastus`)

Vertaa toimintaa `Jasen`–luokan toimintoihin.

11.5.4 Kerho-luokka (`Kerho`)

Luokan nimi: `Kerho`	**Avustajat:**
Vastuualueet: - huolehtii `Jasenet` ja `Harrastukset` –luokkien välisestä yhteistyöstä ja välittää näitä tietoja pyydettäessä - lukee ja kirjoittaa kerhon tiedostoon pyytämällä apua avustajiltaan	- `Jasenet` - `Harrastukset`

11.5.5 Jäsenet-luokka (`Jasenet`)

Luokan nimi: `Jasenet`	**Avustajat:**
Vastuualueet: - pitää yllä varsinaista jäsenrekisteriä, eli osaa lisätä ja poistaa jäsenen - lukee ja kirjoittaa jäsenistön tiedostoon - osaa etsiä ja lajitella	- `Jasen`

11.5.6 Harrastukset-luokka (`Harrastukset`)

Luokan nimi: `Harrastukset`	**Avustajat:**
Vastuualueet: - pitää yllä varsinaista harrasterekisteriä, eli osaa lisätä ja poistaa harrastuksen - lukee ja kirjoittaa harrastukset tiedostoon - osaa etsiä ja lajitella	- `Harrastus`

Koska Harrastukset ja Jäsenet ovat täsmälleen samanlaisia lukuun ottamatta sitä, mitä alkioita ne käsittelevät, voidaan käytännössä Javalla ensin tehdä kantaluokka, josta peritään kumpikin hieman eri versio.

Näin päästään siihen tilanteeseen, jossa myös rinnakkaisten rakenteiden lisääminen Harrastuksille vaatii vain hyvin vähän uutta ohjelmointia.

Huomattakoon, että sekä `Jasenet` että `Harrastukset` ovat pelkkiä abstrakteja tietorakenneluokkia, niiden sisäinen talletustapa voi olla mikä vaan (taulukko, lista, puu) ulkoisen rajapinnan ollessa silti edellisen suunnitelman kaltainen.

12. Jäsenrekisterin runko

Taulukkoko pienen pieni
vaiko lista suuren suuri
joku muuko miettimäksi
rakenne kerhoon katsomaksi.

Kuva tuosta piirtämäksi
tästä temput tutkimaksi
siitä selväksi sävelet
kohtapa jo koodamaksi.

Mitä tässä luvussa käsitellään?

- tietorakenteen valinta

12.1 Runko ilman kommentteja

Emme vielä täysin osaa tehdä edes runkoa jäsenrekisteriohjelmaamme, mutta esitämme tästä huolimatta jonkinlaisen toimivan rungon ohjelmalistauksen. Koodissa tulee lukuisia vielä käsittelemättömiä osia, joita käsittelemme tarkemmin myöhemmin. Samoin etsimme myöhemmin sopivan tavan kommentoida aliohjelmia.

Rungon tarkoituksena on tarjota näkyväksi ne toiminnot, jotka ohjelman suunnitelmassa päätettiin tehdä. Samoin tavoitteena on testata valitun tietorakenteen toimivuus. Jatkossa toimintoja lisätään tähän runkoon.

Tämä runko ei tietenkään ole syntynyt kerralla, vaan ensin on testattu tietorakenteet yksittäin ja sitten lisätty näiden käyttö valmiiseen menu-runkoon (vrt. menut_3\Naytto.java)

12.2 Valittava tietorakenne

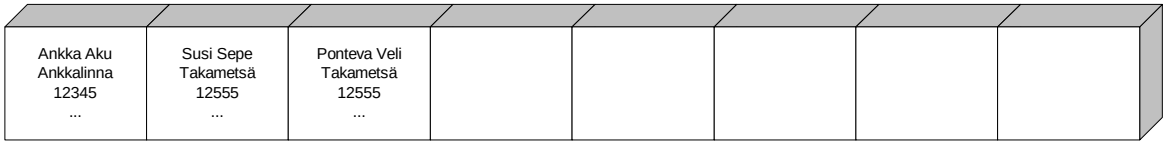
Minkälaisen tietotyypin voisimme valita? Vaihtoehtoja tulee ehkä lähes yhtä paljon kuin ohjelmoijiakin on. Voimme kuitenkin vertailla eräiden rakenteiden etuja ja haittoja. Jos mahdollista, tietorakenteiden tulisi olla yhtenäinen tehdyn oliosuunnitelman kanssa. Seuraavista ehdotuksista mikään ei ole ristiriidassa edellisessä luvussa tehdyn oliosuunnitelman kanssa. Vasta kun valitaan harrastusten talletustapaa, joudumme valinnan eteen.

Lähdemme siitä ajatuksesta, että koko käsiteltävä aineisto on kerralla keskusmuistissa. Voisimme tietenkin operoida myös suoraan levylle, mutta oppimisen tässä vaiheessa voi seuraava ratkaisu olla helpompi:

```
lue tiedosto muistiin
käsittele aineistoa muistissa
talleta aineisto takaisin levylle
```

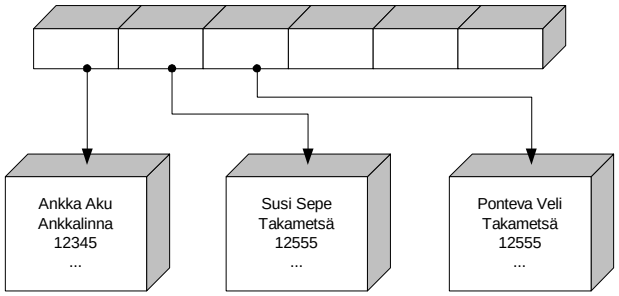
12.2.1 Taulukko

Taulukko on kiinteä tietorakenne, jota luotaessa täytyy jo tietää monelleko ihmiselle varaamme tilaa. Tässä tulee äkkiä varattua tilaa joko liikaa, jolloin tila ei riitä muille toiminnoille, tai liian vähän, jolloin kaikki henkilöt eivät mahdu rekisteriin. Esimerkissämme olemme varanneet n. 300 tavua/henkilö. Tilan varaaminen sadalle henkilölle veisi jo 30000 tavua. Usein sata ei edes riitä!



Kuva 12.1 Taulukko

Javassa asia ei tietysti ole ihan näin suoraviivaista. Javassahan oliot ovat vaan viitteitä, jolloin oliotaulukko onkin vain taulukollinen viitteitä. Näin "liian tilan varaaminen" ei ole kovin kohtalokasta, jos jokainen viite vie esim. 4 tavua, niin 100 hengen viitteet vievät 400 tavua. Edes tuhannen hengen viitteet eivät vie mitenkään katastrofaalisesti tilaa:



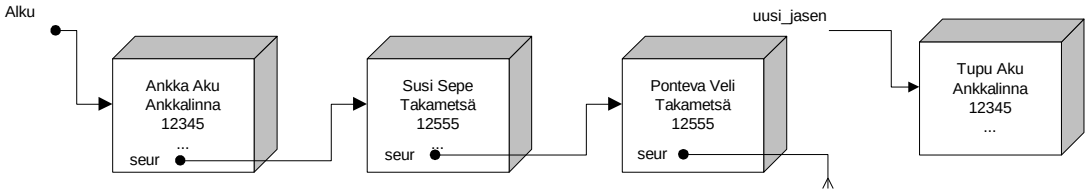
Kuva 12.2 Javan taulukko

Kuitenkin ohjelmoijan omalle vastuulle jää taulukon maksimikoon ja taulukon "käytettyjen" alkioiden lukumäärän ylläpitäminen. Maksimikokohan saadaan aina taulukon koosta, joten tämä ei ole Javassa kovin suuri vaiva. Käytettyjen alkioiden määrän ylläpitoon täytyy kuitenkin rakentaa jokin mekanismi.

Javassa on tarjota valmiitakin tietorakenteita, mutta niiden pienenä puutteena on se, että ne tallentavat `Object`-tyyppisiä olioita. Tällöin aina kun alkio otetaan tietorakenteesta, pitää sen tyyppi muuttaa vastaamaan todellista tyyppiä. Taulukossa aliota taas voivat suoraan olla omaa tyyppiään (eli oman tyypin viitteitä).

12.2.2 Linkitetty lista

Linkitetty lista on rakenne, jossa meillä on tieto vain listan 1. alkiosta. Tämän jälkeen kukin alkio tietää itseään seuraavan alkion, kunnes listan viimeinen alkio ei enää osoita minnekään.



Kuva 12.3 Linkitetty lista

Listan hyvänä puolena on se, ettei etukäteen tarvita mitään tietoa alkioiden lukumäärästä. Alkioita voidaan lisätä listaan joko alkuun, keskelle tai loppuun niin kauan kuin muistia riittää.

Oppimisen tässä vaiheessa kuitenkin linkitetyn listan käsittelyalgoritmit (lisäys, poisto, lajittelu jne..) saattavat olla liian työläitä.

Mikäli rakennamme ohjelman huolella, ei tietorakenteen vaihtaminen jälkeen päinkään ole mahdoton tehtävä. Tätä auttaa vielä aikaisemmin tekemämme valinta käyttää abstraktia rajapintaa (lisää, poista, etsi) tietorakenneluokan (`Kerho` tai `Jasenet`) ja käyttöliittymän (`Naytto`) välillä.

12.2.3 Sekarakenne

Valitsemme tähän toteutukseen tietorakenteeksi sekarakenteen:

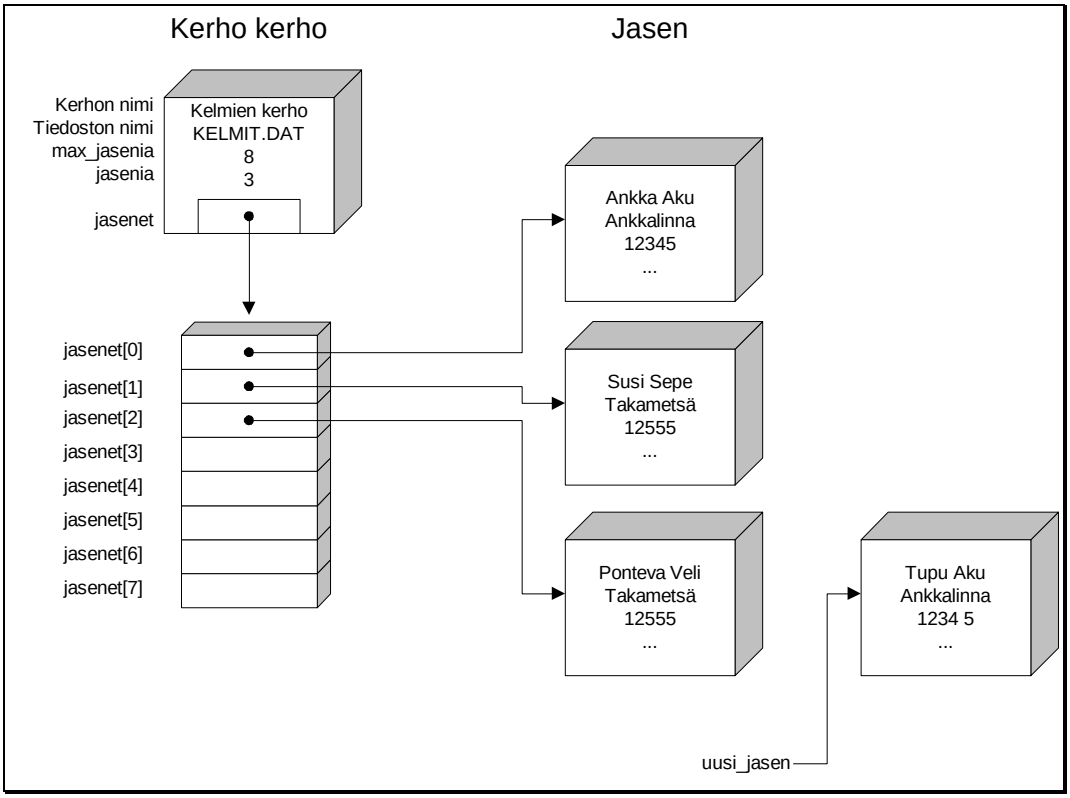
Siis perusrakenteena meillä on `Kerho`–tyyppi, joka pitää sisällään kerhon perustiedot. Kerhosta on osoitin taulukkoon, jossa on osoittimet varsinaisiin henkilöiden tietoihin (`Jasen`).

Henkilöiden tiedoille varattua tilaa ei ole olemassa ennen kuin sitä tarvitaan. Siis varataan kullekin kerhoon lisättävälle henkilölle hänen tiedoilleen tarvittava uusi n. 300 tavun "möykky" lisäyksen yhteydessä.

Osoitintaulukkoon sijoitetaan sitten vastaavaan paikkaan sen muistiosoitteen arvo, josta henkilölle tarvittava tila saatiin varattua.

Tässäkin rakenteessa on se huono puoli, että osoitintaulukon koko pitää päättää ennen kuin sinne voidaan sijoittaa osoitteita. Yksi osoite vie kuitenkin enimmilläänkin tilaa 4 tavua, joten kiinteää tilan varausta esim. 1000 henkilön taulukossa tulee vain 4000 tavua.

Hyvinä puolina rakenteessa on sen suhteellisen helppo käsittely sekä lisäyksen, poiston että lajittelun tapauksessa.



Kuva 12.4 Tietorakenne kun kerho tallettaa jäsenet

Tehtävä 12.1 Lisäys

Kirjoita algoritmi henkilön lisäämiseksi rakenteeseen.

Tehtävä 12.2 Etsiminen

Kirjoita algoritmi tietyn henkilön etsimiseksi (vaikkapa nimellä).

Tehtävä 12.3 Poisto

Kirjoita algoritmi löydetyn henkilön (miten löytö kannattaa säilyttää?) poistamiseksi rakenteesta.

Tehtävä 12.4 Lajittelu

Kirjoita algoritmi rakenteen lajittelemiseksi aakkosjärjestykseen. Mitä lajittelussa kannattaa vaihdella?

12.2.4 Ohjelman runko

Ohjelmalistaus löytyy listausmonisteesta tiedostosta

1. runko_5\Naytto.java

12.3 Harrastukset mukaan

Jos halutaan tallettaa myös kullekin jäsenelle vaihteleva määrä harrastuksia, on jälleen mahdollisuuksia useita. Tietorakennetta valittaessa voidaan käyttää samaa kriteeriä kuin tiedostoakin valittaessa. Välttämätöntä tämä ei kuitenkaan ole, vaan voidaan sisäisesti tietysti käyttää myös erilaista rakennetta kuin ulkoisesti.

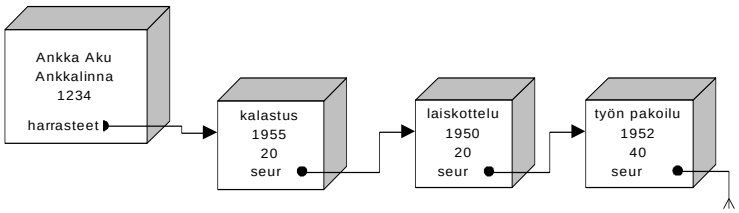
12.3.1 Tiedot jäsenen yhteyteen

Jos tiedoston muoto on sellainen että harrastukset on lueteltu jäsenen tietojen yhteydessä, kannattaa tietorakennekin valita vastaavasti.

```
Ankka Aku          |010245-123U|Ankkakuja 6 |12345      |ANKKALINNA |12-12324   |          |
- kalastus                 | 1955 | 20
- laiskottelu              | 1950 | 20
- työn pakoilu             | 1952 | 40
Susi Sepe          |020347-123T|            |12555      |Takametsä  |           |          |
- possujen jahtaaminen     | 1954 | 20
- kelmien kerho            | 1962 |  2
Ponteva Veli       |030455-3333|            |12555      |Takametsä  |           |          |
- susiansojen rakentaminen | 1956 | 15
```

Nyt tietorakenne voisi olla tilanteesta riippuen mikä tahansa edellä esitetyistä siten, että `kerho`sta alkava rakenne toistuu jäsenen kohdalla.

Esimerkiksi linkitetty lista:



Kuva 12.5 Harrastukset linkitettynä listana

12.3.2 Relaatiomalli

Jos tiedot on talletettu relaatiomallin mukaan, voi olla kannattavaa tehdä myös sisäinen tietorakenne vastaavaksi. Vaikka jatkossa emme toteutakaan kerhoon vielä harrastuksia, teemme tietorakenteen ja oliot sellaisiksi, että harrastusten käsittely jälkeenpäin olisi mahdollisimman helppoa.

Toteutettavaa ohjelmaa ajatellen tästä valinnasta seuraa yksi byrokratiaporras (`Kerho <-> Jasenet`) lisää, joka aluksi saattaa tuntua turhalta. Valinta maksaa itseään takaisin vasta ongelman monimutkaistuessa. Tähän samaan monimutkaistumisongelmaan tulemme törmäämään jatkossakin. Yksinkertaisin mahdollisuus, jolla vaadittu toimenpide juuri ja juuri voidaan suorittaa, johtaa usein jatkoa ajatellen umpikujaan.

Tehtävä 12.5 Lisää harrastus

Kirjoita algoritmi uuden harrastuksen lisäämiseksi. (Ks. alla oleva kuva)

Tehtävä 12.6 Mitä harrastaa

Kirjoita algoritmi, joka vastaa kysymykseen "Mitä henkilö N harrastaa?"

Tehtävä 12.7 Kuka harrastaa

Kirjoita algoritmi, joka vastaa kysymykseen: "Ketkä harrastavat harrastusta H?".

Tehtävä 12.8 Poista harrastus

Kirjoita algoritmi harrastuksen poistamiseksi.

Tehtävä 12.9 Jäsenen poistaminen

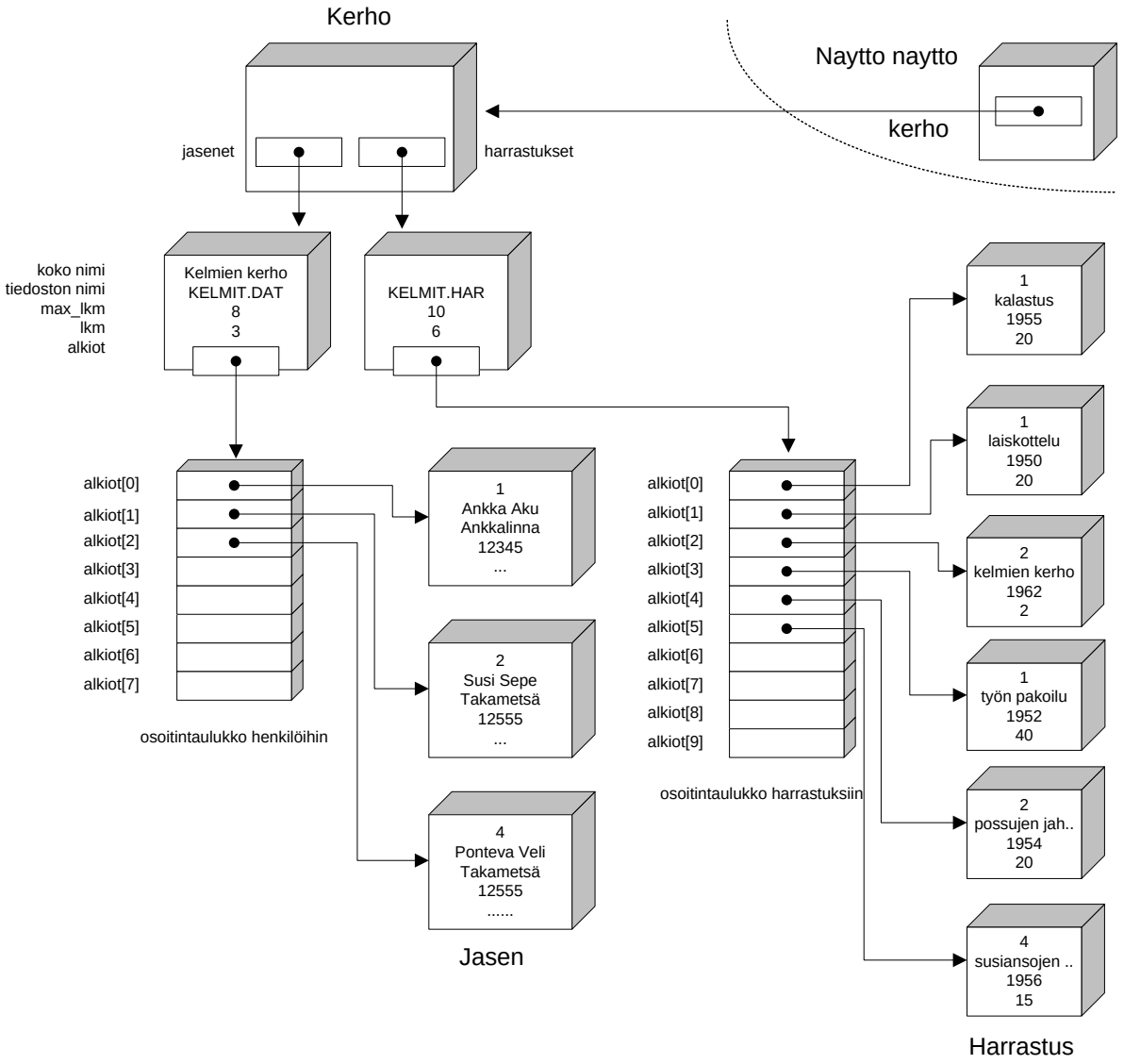
Kirjoita algoritmi, joka poistaa jäsenen jonkin nimi on N.

Tehtävä 12.10 "Roskaharrastukset"

Kirjoita algoritmi, joka poistaa "roskaharrastukset", eli ne harrastukset, joille ei löydy "omistajaa". Tällaiseen tilanteeseen hyvässä ohjelmassa ei tietenkään koskaan päädytä.

Tehtävä 12.11 Monta saman lajin harrastajaa

Jos harrastusten nimet ovat kovin pitkiä ja harrastuksista talletetaan vielä kuhunkin harrastukseen liittyvää lisätietoa, niin edellä mainittu rakenne käy tehottomaksi heti kun löytyy useita saman lajin harrastajia. Esitä ratkaisu, jossa hukkatilaa (mm. saman harrastuksen nimen toistamista) ei esiinny, mutta jolla voidaan tehdä kaikki samat tehtävät, kuin esitetyllä ratkaisulla.



Kuva 12.6 Harrastukset relaation avulla

12.3.3 Ohjelman runko harrastusten kanssa

Ohjelmalistaus, jossa harrastuksetkin ovat mukana, löytyy tiedostosta

1. runko_5\Nayttohar.java
2.

13. Java–kielen taulukoista

Taulukko se taasen tuttu
oiva säilö alkioille
maja muuttujalle monelle
silmäiltäväksi silmukalla.

Mitä tässä luvussa käsitellään?

- Java–kielen taulukot
- taulukoiden ja viitteiden yhteys
- moniulotteiset taulukot

Syntaksi:

```
Taulukon esittely:     alkiontyyppi taulukonnimi[];
Taulukon luominen:     taulukonnimi = new alkiontyyppi[koko_alkioina]
Alkioon viittaaminen:  taulukonnimi[alkion_indeksi]
Muista                 1. indeksi = 0
                       viimeinen  = koko_alkiona-1
Silmukoissa            for (i=0; i<taulukonnimi.length; i++) ...
2-ul.taulukon es:      alkiontyyppi taulukonnimi[][];
2-ul taul. luominen:   taulukonnimi = new alkiontyyppi[riveja][sarakkeita]
```

Luvun esimerkkikoodit:

http://www.mit.jyu.fi/~vesal/kurssit/ohj2/moniste/esim/java-taul/

13.1 Yksiulotteiset taulukot

C–kielessä taulukoita ei oikeastaan ole, tai ainakin ne ovat '2. luokan kansalaisia'. Lausuma tarkoittaa sitä, että taulukoista on käytettävissä vain 1. alkion osoite ja esimerkiksi taulukon sisällön sijoittaminen toiseen taulukkoon ei onnistu sijoitusoperaattorilla. Lisäksi taulukon rajoissa pysymiselle ei ole minkäänlaista valvontaa.

Javassa onneksi taulukot on tehty hieman paremmin. Erityisesti kriittisistä rajojen ylityksistä tulee poikkeus.

13.1.1 Taulukon määrittely

Taulukko määritellään kertomalla taulukon alkioiden tyyppi ja luomalla sitten varsinainen taulukko:

```
int k_pituudet[];          // saadaan vasta viite jolla voidaan viitata taulukkoon
kpituudet = new int[12];   // luodaan taulukko;
```

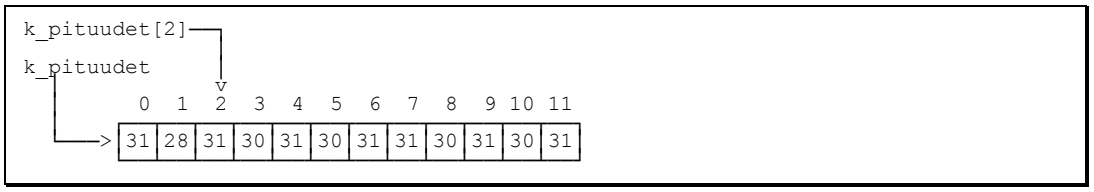
Tällöin taulukon 1. alkion indeksi on 0 ja 12. alkion indeksi on 11.

Määrittelyllä muuttujasta `k_pituudet` tulee osoitin kokonaislukuun; taulukon alkuun.

13.1.2 Taulukon alkioihin viittaaminen indeksillä

Taulukon alkioon voidaan viitata alkion indeksin avulla

```
k_pituudet[0]=31; /* tammikuu */
k_pituudet[1]=28; /* helmikuu */
```



Taulukon rajojen ylityksestä seuraa `IndexOutOfBoundsException`-poikkeus

```
k_pituudet[24]=31;
```

eli 2 paikkaa eteenpäin taulukon alusta lukien.

Taulukko voitaisiin nollata seuraavalla silmukalla:

```
for (int i=0; i<k_pituudet.length; i++)
  k_pituudet[i]=0;
```

Huomautus! Taulukoiden käsittelyssä on muistettava, että indeksi liikkuu välillä `[0,YLÄRAJA[`.

13.1.3 Taulukon alustaminen

Taulukko voidaan alustaa (vain) esittelyn yhteydessä:

```
                /*  1. 2. 3. 4. 5. 6. 7. 8. 9.10.11.12 */
int k_pituudet[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Tehtävä 13.1 Taulukon alkioiden summa

Kirjoita funktio–aliohjelma `taulukon_summa`, joka palauttaa taulukon alkioiden summan. Kirjoita pääohjelma, jossa aliohjelmaa kutsutaan valmiiksi alustetulla taulukolla `k_pituudet` ja tulostetaan vuoden päivien lukumäärä.

13.2 Merkkijonot

Merkkijonot ovat eräs ohjelmoinnin tärkeimmistä tietorakanteista. Valitettavasti tämä on lähes poikkeuksetta unohtunut ohjelmointikielten tekijöiltä. Heille riittää että kielellä VOI tehdä merkkijonotyypin. Tavallista käyttäjää kiinnostaa tietysti onko se tehty ja onko se hyvä. Usein vastaus on EI. Näin myös C–kielen kohdalla! *C++*:han on jo välttävä merkkijonoluokka. Javassa on kaksi merkkijonoluokkaa `String` ja `StringBuffer`.

13.2.1 Merkkityyppi

Yksittäinen merkki on Java–kielessä tyyppiä `char`:

```
char rek_1_merkki;
rek_1_merkki = 'X';
```

Merkkimuuttujiin voidaan vallan hyvin sijoittaa myös merkin koodiarvo

```
char m;
m = 65;
if ( m == 'A' ) ...
```

Lukuarvo tarkoittaa merkin (UNICODE–) koodia.

13.2.2 String

Javan `String`-luokka tarjoaa muuttumattoman merkkijonon (*immutable*). Merkkijonon "sisältö" voidaan vaihtaa vain luomalla uusi merkkijono.

13.2.3 StringBuffer

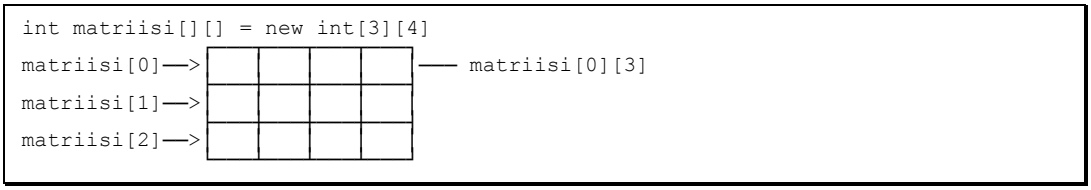
Jos halutaan merkkijono, jonka sisältöä voidaan muuttaa (*mutable*), pitää käyttää `StringBuffer`-luokkaa.

13.3 Moniulotteiset taulukot

Moniulotteiset taulukot ovat Javassa vain yksiulotteisia taulukoita taulukoista.

13.3.1 Kiinteä esittely

Kaikkein helpoin tapa esitellä moniulotteinen taulukko on aivan normaali esittely:



Taulukon nimi on vain viite taulukkoon. Taulukko on yksiulotteinen taulukko riveistä. Edellä

```
matriisi.length == 3
matriisi[1].length == 4
```

Taulukon alkioina voi tietysti olla mikä tahansa olemassa oleva tyyppi. Myös moniulotteinen taulukko voidaan alustaa esittelyn yhteydessä:

```
double yks[][] = {
  { 1.0, 0.0, 0.0 },
  { 0.0, 1.0, 0.0 },
  { 0.0, 0.0, 1.0 }
}
```

Tehtävä 13.2 Matriisit

Kirjoita seuraavat aliohjelmat, jotka saavat parametrinään 2 nxn matriisia ja palauttavat nxn matriisin:

1. Laskee yhteen 2 matriisia.
2. Kertoo kaksi matriisia keskenään. (Kirjoita avuksi funktio, joka kertoo matriisin rivin i toisen matriisin sarakkeella j).

13.3.2 Yksiulotteisen taulukon käyttäminen moniulotteisena

Toisaalta moniulotteinenkin taulukko voidaan toteuttaa 1–ulotteisena. Tästä muunnoksestahan puhuttiin jo monisteen alkuosassa. On makuasia kumpiko järjestys esimerkiksi matriisissa valitaan: sarakelista vaiko rivilista. Rivilista on C–kielen mukainen, mutta toisaalta maailma on pullollaan Fortran aliohjelmia, joissa matriisit on talletettu sarakelistana. Siis kumpikin tapa on syytä hallita.

Tehtävä 13.3 Matriisi 1–ulotteisena

Kirjoita aliohjelma `tee_yksikko`, jolle tuodaan parametrinä neliömatriisin rivien lukumäärä ja 1–ulotteisen taulukon viite, ja joka alustaa tämän neliömatriisin yksikkömatriisiksi.

13.3.3 Taulukko taulukoista

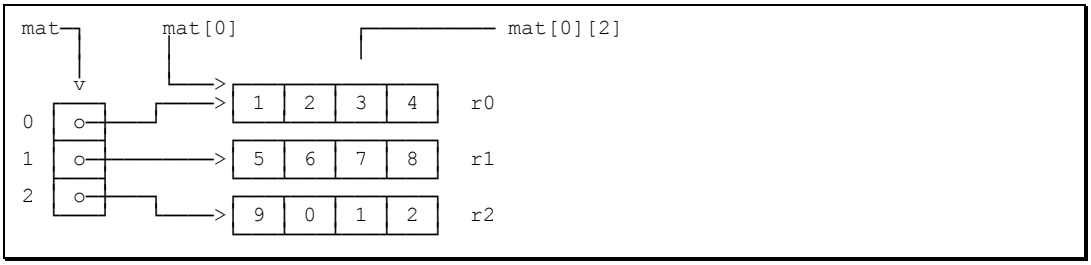
Javassahan moniuloitteinen taulukko on tosiasiassa taulukko taulukoista.

java-taul\Mat2.c - matriisi parametrina riviosoittimen avulla

```
/**
 * Matriisi parametrina
 * @author Vesa Lappalainen
 * @version 1.0, 04.03.2003
 */
public class Mat2 {

  public static double alkioiden_summa(double mat[][]) {
    double summa = 0; int riv = mat.length;
    for (int i=0; i<riv; i++) {
      int sar = mat[i].length;
      for (int j=0; j<sar; j++)
        summa += mat[i][j];
    }
    return summa;
  }

  public static void main(String[] args) {
    double s1,s2,s3;
    double yks[][] = {
      { 1.0, 0.0, 0.0 },
      { 0.0, 1.0, 0.0 },
      { 0.0, 0.0, 1.0 }
    };
    double mat2[][] = { {1,2,3,4},{5,6,7,8} },
           mat3[][] = { {1,0,0},{0,1,0},{0,0,1} };
    s1 = alkioiden_summa(yks);
    s2 = alkioiden_summa(mat2);
    s3 = alkioiden_summa(mat3);
    System.out.println("Summat ovat " + s1 + ", " + s2 + " ja " + s3);
  }
}
```

13.3.4 Taulukko viitteistä

Se että matriisi onkin vain taulukko viitteistä riveihin, mahdollistaa edellä olleen mielivaltaisen kokoisen matriisin käyttämisen aliohjelman parametrina. Matriisin rivit voidaan luoda myös erikseen:



java-taul\Mat3.java - matriisi osoitintaulukon avulla

```
/**
 * Matriisi kasattuna irrallisista riveistä
 * @author Vesa Lappalainen
 * @version 1.0, 04.03.2003
 */
public class Mat3 {

  public static double alkioiden_summa(double mat[][],int riveja, int sarakkeita)
{
    int riv = Math.min(riveja,mat.length);
    double summa = 0;
    for (int i=0; i<riv; i++) {
      int sar = Math.min(sarakkeita,mat[i].length);
      for (int j=0; j<sar; j++)
        summa += mat[i][j];
    }
    return summa;
  }

  public static void main(String[] args) {
    double s1,s2;
    double r0[] = {1,2,3,4}, r1[] = {5,6,7,8}, r2[] = {9,0,1,2};
    double mat[][] = {r0,r1,r2};
    s1 = alkioiden_summa(mat,2,3);
    s2 = alkioiden_summa(mat,3,4);
    System.out.println("Summat on " + s1 + " ja " + s2);

  }
}
```

Javan menettelyssä on vielä se etu, ettei kaikkien rivien välttämättä tarvitsisi edes olla yhtä pitkiä. Harvassa matriisissa osa osoittimista voisi olla jopa `null`–osoittimia, mikäli rivillä ei ole alkioita (aliohjelman pitäisi tietysti tarkistaa tämä). Oikeasti rivit usein vielä luotaisiin dynaamisesti ajonaikana tarvittavan pituisina.

Tehtävä 13.4 Transpoosi

Kirjoita taulukko–osoittimia käyttäen aliohjelma, joka saa parametrinään kaksi matriisia ja niiden dimensiot. Aliohjelma tarkistaa voiko toiseen matriisiin tehdä toisen transpoosin (vaihtaa rivit ja sarakkeet keskenään) ja tekee transpoosin jos pystyy. Onnistuminen palautetaan aliohjelman nimessä.

13.4 Komentorivin parametrit (`argv`)

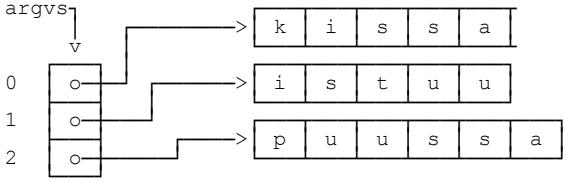
Esimerkiksi Java–kielinen pääohjelma saa käyttöjärjestelmältä tällaisen taulukon kutsussa olleista argumenteista:

java-taul\Argv.java - komentorivin parametrit

```
/**
 * Ohjelma tulostaa komentorivin parametrit
 * @author Vesa Lappalainen
 * @version 1.0, 04.03.2003
 */
public class Argv {
  public static void main(String[] args) {
    System.out.println("Argumenttejä on " + args.length + " kappaletta:");
    for (int i=0; i<args.length; i++)
      System.out.println(i + ": " + args[i]);
  }
}
```

Kun ohjelma ajettaisiin komentoriviltä saattaisi tulostus olla seuraavan näköinen (*MS–DOS* -koneessa):

```
C:\kurssit\moniste\esim\java-taul>java Argv kissa istuu puussa[RET]
Argumentteja on 3 kappaletta:
0: kissa
1: istuu
2: puussa
C:\kurssit\moniste\esim\java-taul>_
```



Tehtävä 13.5 Palindromi

Kirjoita Java-ohjelma `Pali`, jota kutsutaan komentoriviltä seuraavasti:

```
C:\OMAT\OHJELMOI\VESA>java Pali kissa[RET]
kissa EI ole palindromi!
C:\OMAT\OHJELMOI\VESA>java Pali saippuakauppias[RET]
saippuakauppias ON palindromi!
C:\OMAT\OHJELMOI\VESA>_
```

14. Parametrin välityksestä ja osoittimista, kertaus

.

Mitä tässä luvussa käsitellään?

- kerrataan aliohjelmien kutsumista
- kerrataan aliohjelmien suunnittelua
- kerrataan viitteitä parametreinä
- *HUOM! Luku on vielä pahasti kesken*

14.1 Miksi aliohjelmia käytetään

Aliohjelmia käytettiin mm. seuraavista syistä:

- kyseessä on luonteeltaan oma looginen kokonaisuutensa
- joku toinen (tai itse aikaisemmin) on kirjoittanut halutun tehtävän suorittavan aliohjelman

Mikäli aliohjelma kirjoitetaan itse, tulee ongelmaksi tietysti se, mitä parametrejä aliohjelmaan esitellään.

Mikäli aliohjelman on kirjoittanut joku toinen, on ongelmana se miten aliohjelmaa kutsutaan.

15. Kommentointi ja jakaminen osiin

.

Mitä tässä luvussa käsitellään?

- ohjelman kommentointi
- ohjelman jakaminen osiin
- .jar-tiedostot *(ei vielä kirjoitettu)*
- *HUOM! Luku on vielä pahasti kesken*

15.1 Kommentointi

"*Löysin vuosi sitten kirjoittamani kolmen rivin aliohjelman. Siinä ei ollut yhtään kommenttia. Miten se hoiti tehtävänsä? Mietin asiaa kaksi päivää ja sitten ymmärsin miksi tehtävä oli triviaali eikä tarvinnutkaan kommenttia.*" (Vapaa käännös verkossa olleesta kirjeestä / *vl*).

Kommentoimme jatkossa aliohjelmat huolellisesti. Seuraavana on kuvattu eräs tapa. Tietenkin on miljoona muutakin tapaa, mutta ainakin seuraavan tavan ajatus kannattaa pitää mielessä.

15.1.1 Valmiin kommenttilohkon lukeminen

Mikäli käytetty editori ei tue automaattisesti kommentointia, kannattaa kirjoittaa aina tarvittavat kommentit vaikka tiedostoihin a.t (alku) ja m.t (metodi) ja opetella käyttämään editorin "lisää tiedosto" toimintoa.

15.1.2 Parametrilistan kommentointi

Javassa käytetään *JavaDocin* mukaista kommentointia:

```
  /** Palauttaa merkkijonon jossa on n kappaletta välilyöntejä
   * @param n montako välilyöntiä tarvitaan
   * @return jono, jossa on n kpl välilyöntejä
   */
  public static String tyhja(int n) {
...
```

15.1.3 Koodin kommentointi

Itse ohjelmakoodi kommentoidaan seuraavasti:

- selviä kielen rakenteita ei saa kommentoida. Ei siis

```
i=5; // sijoitetaan i on 5                /* TURHA! */
```

- kuitenkin mikäli lauseella on selvä merkitys algoritmin kannalta, kommentoidaan tämä

```
i=5; // aloitetaan puolestavälistä
```

- ryhmitellään lauseet tyhjien rivien avulla loogisiksi kokonaisuuksiksi. Tällaisen kokonaisuuden alkuun voidaan laittaa kommenttirivi, joka kuvaa kaikkien seuraavien lauseiden merkitystä.

- mikäli tekee mieli kommentoida lauseryhmä, kannattaa miettiä voitaisiinko koko ryhmä kirjoittaa aliohjelmaksi. Aliohjelman nimi sitten kuvaisi toimintaa niin hyvin, ettei kommenttia enää tarvittaisikaan. Kuitenkin jos näin suunnitellulle aliohjelmalle tulee iso kasa (liki 10) parametrejä, täytyy asiaa ajatella uudestaan.
- muuttujien nimet valitaan kuvaaviksi. Kuitenkin mitä lokaalimpi muuttujan käyttö, sitä lyhyemmäksi nimi voidaan jättää. `i` ja `j` sopivat aivan hyvin silmukkamuuttujien nimiksi ja `p` yms. osoittimen nimeksi (lokaalisti).
- globaaleja muuttujia vältetään 'kaikin keinoin'
- olioiden ansiosta globaalit muuttujat voidaan yleensä välttää kokonaan!
- vakiotyyliset (alustetaan esittelyn yhteydessä eikä ole tarkoitus ikinä muuttaa) globaalit muuttujat on sallittu sellaisenaan ja niiden nimet kannattaa ehkä kirjoittaa isolla.
- funktioiden paluuarvolle valitaan tietty tyyli, joka pyritään säilyttämään koko ohjelman ajan. Esimerkiksi `true` = onnistui ja `false` epäonnistui.

15.2 Omat aliohjelmakirjastot

Aiemmin rakensimme joukon merkkijonojen käsittelyssä tarvittavia apuohjelmia. Nämä aliohjelmat voitaisiin kopioida suoraan myös kerhorekisteriimme. Käytännössä näin ei kuitenkaan kannata tehdä, sillä valmiiksi testatut aliohjelmat olisivat mukana vain turhaan lisäämässä käännösaikaa.

Tämän takia aliohjelmat kirjoitetaan omaksi tiedostokseen, vaikkapa nimelle `Mjonot.java`.

Kirjastossa on mm. `wildmat`–aliohjelma merkkijonojen samaistamiseksi, kun jonossa saa esiintyä jokerimerkkejä * ja ? (vrt. MS–DOS).

Tehtävä 15.1 wildmat (opettavainen)

Pöytätestaa `wildmat`–aliohjelma syötöllä `Kissa` ja `*ss*`. Ohje: Kirjoita normaalin pöytätestin mukainen taulukko. Koska funktio kutsuu itseään, kannattaa jokaista uutta itseään kutsumista varten kirjoittaa oma uusi sarake uuden kutsukerran muuttujista. Osoittimet `s` ja `m` kannattaa säilyttää muodossa, jossa ne näyttävät loppumerkkijonon. Esimerkiksi:

```
aluksi  s = "Kissa";  *s='K'
s++  -> s = "issa"    *s='i'
s++  -> s = "ssa"     *s='s'
```

15.2.1 Kirjaston testaus

Kirjastojen testaamista varten kannattaa kirjoittaa tiedostoon `main`-metodi, jossa kutsutaan testiarvoilla kirjaston rutiineja.

15.3 Kerho-ohjelman jako osiin

Kirjoittamisen ja ylläpidon kannalta voi olla helpompi jakaa ohjelma muutamaan loogiseen osaan. Nyt on valmiiksi kirjoitettuna jäsenrekisterin runko–osa. Tällä rungolla voidaan testata tietorakenteiden toimivuus: Rungossa meillä on käytössä alkeellinen näytölle tulostava aliohjelma.

Suurimmaksi osaksi kannattaa kirjoittaa niin, että kukin luokka muodostaan yhden tiedoston.

On turha toivo, että keksisimme kaikki määritykset ja aliohjelmat kerralla. Tehtävää täytyy hahmotella palanen kerrallaan. Kun jokin homma tuntuu venyvän liian pitkäksi tai monimutkaiseksi, määrittelemme tehtävän useampaan alatoimintoon ja toteutamme nämä toiminnot sitten aliohjelmina/luokkina. Aliohjelmien/metodien parametrit saattavat vielä myöhemmin muuttua, kun huomataan saman tehtävän käyvän sekä tähän että tuohon tehtävään. Esimerkiksi etsiminen ja selailu käy samalla myös korjailuun ja poistoon. Ainoana erona on, että korjaus– ja poistonäppäimet eivät ole pelkässä etsimisessä sallittuja.

16. Dynaaminen muistinkäyttö

Aina ei aavista kokoa
suuruuttapa suuren suurta
dynaamisuus siis avuksi
mielehen muuttuva kokokin.

Varaa muistia tarvittaissa
uutta uikuta muuttujille
viitteen päähän pantavaksi
sieltä sitten saatavaksi.

Listoja ja taulukoita
vinkeitä vipeltimiä
algoritmejä arvokkaita
valmiinakin tarjonnassa.

Mitä tässä luvussa käsitellään?

- Dynaaminen muistinhallinta
- Dynaamiset taulukot
- Muistinsiivous
- Algoritmit

Syntaksi:

```
Dyn.olion.luonti      muuttuja = new Luokka(parametrit)
"Hävittäminen"        muuttuja = null;
```

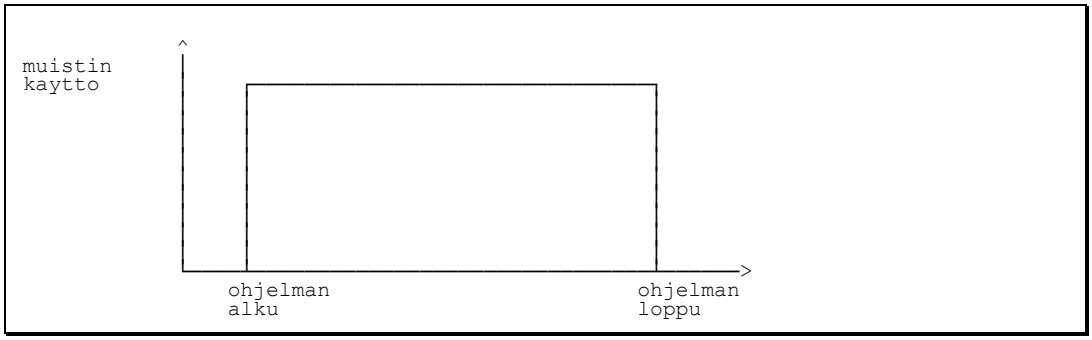
Luvun esimerkkikoodit:

http://www.mit.jyu.fi/~vesal/kurssit/ohj2/moniste/esim/dyna/

Olemme oppineet varaamaan muuttujia esittelyn yhteydessä. Usein olioita voidaan luoda (= varata muistitilaa) myös ohjelman ajon aikana. Tämä on tarpeellista erityisesti silloin, kun ohjelman kirjoittamisen aikana ei tiedetä muuttujien määrää tai ehkei jopa edes kokoa (taulukot).

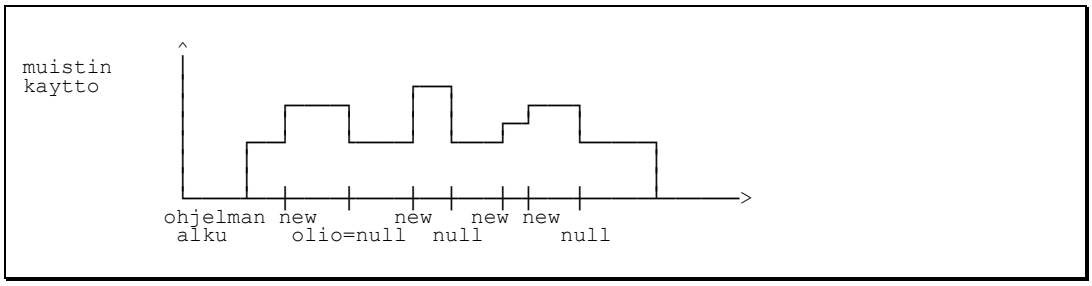
16.1 Muistin käyttö

Karkeasti ottaen tavallisen ohjelman muistinkäyttö näyttäisi ajan funktiona seuraavalta:



Edellinen kuva on hieman yksinkertaistettu, koska "oikeasti" aliohjelmien lokaalit muuttujat (automaattiset muuttujat) syntyvät aliohjelmaan tultaessa ja häviävät aliohjelmasta poistuttaessa. Näin ollen käytetyn muistin yläraja vaihtelee sen mukaan mitä aliohjelmia on kesken suorituksen.

Dynaamisia muuttujia voidaan tarvittaessa luoda ja kun muistitilaa ei enää tarvita, voidaan vastaavat muuttujat vapauttaa:



Näin muistin maksimimäärä saattaa pysyä huomattavasti pienempänä kuin ilman dynaamisia muuttujia. Idea on siis siinä, että muistia varataan aina vain sen verran, kuin sillä hetkellä tarvitaan. Kun muistia ei enää tarvita, vapautetaan muisti.

Ajonaikana luotaviin muuttujiin tarvitaan osoitteet. Nämä osoitteet pitää sitten tallettaa johonkin. Talletus voitaisiin tehdä esimerkiksi taulukkoon tai sitten alkioista pitää muodostaa linkitetty lista.

16.2 Dynaamisen muistin käyttö Javassa

16.2.1 `new`

Javassa kaikki oliot luodaan dynaamisesta muistista (keosta).

16.2.2 Olion tuhoaminen

Kun oliota ei enää tarvita, se häviää aikanaan, kun kaikki siihen osoittavat viitteet asetetaan `null`-arvoon tai poistetaan viitteen kokonaan (poistutan metodista jolloin lokaalit viitteet katoavat) . Tällöin olio muuttuu roskaksi ja muistisiivous aikanaan tuhoaa kaikki oliot, joihin ei ole viitteitä.

16.2.3 Taulukon luominen `new []`

Jos `new`–operaattorilla on luotu taulukollinen olioita niin varsinaiset oliot pitää luoda erikseen.

```
monta = new Luokka[20];
...vielä ei ole varsinaisia alkiota, vain 20 viitettä.
```

16.3 Dynaamiset taulukot

Kerhon jäsenrekisterissä käytettiin osoitintaulukkoa dynaamisesti. Vastaavan rakenteen tarve tulee usein ohjelmoinnissa. Tällöin tulee aina vastaan ongelma: montako alkiota taulukossa on nyt? Jäsenrekisterissä tämä oli ratkaistu `Jasenet`–luokassa tekemällä sinne kentät, joista nämä rajat selviävät.

Javassa edällä mainittu dynaaminen taulukko voidaan toteuttaa käyttäjän kannalta todella joustavaksi:

dyna\Taulukko.java -esimerkki dynaamisesta taulukosta

```
/**
 * Esimerkki dynaamisesta taulukosta
 * @author Vesa Lappalainen
 * @version 1.0, 02.03.2002
 */

public class Taulukko {
  public class TaulukkoTaysiException extends Exception {
    TaulukkoTaysiException(String viesti) { super(viesti); }
  }

  private int alkiot[];
  private int lkm;

  public Taulukko() {
    alkiot = new int[10];
  }

  public Taulukko(int koko) {
    alkiot = new int[koko];
  }

  public void lisaa(int i) throws TaulukkoTaysiException {
    if ( lkm >= alkiot.length ) throw new TaulukkoTaysiException("Tila loppu");
    alkiot[lkm++] = i;
  }

  public String toString() {
    StringBuffer s = new StringBuffer("");
    for (int i=0; i<lkm; i++)
      s.append(" " + alkiot[i]);
    return s.toString();
  }

  public void set(int i, int luku) throws IndexOutOfBoundsException {
    if ( ( i < 0 ) || ( lkm <= i ) )
       throw new IndexOutOfBoundsException("i = " + i);
    alkiot[i] = luku;
  }

  public int get(int i) throws IndexOutOfBoundsException {
    if ( ( i < 0 ) || ( lkm <= i ) )
      throw new IndexOutOfBoundsException("i = " + i);
    return alkiot[i];
  }

  public static void main(String[] args) {
    Taulukko luvut = new Taulukko();
    try {
      luvut.lisaa(0); luvut.lisaa(2);
      luvut.lisaa(99);
    } catch ( TaulukkoTaysiException e ) {
      System.out.println("Virhe: " + e.getMessage());
    }
```

```
    System.out.println(luvut);
    luvut.set(1,4);
    System.out.println(luvut);
    int luku = luvut.get(2);
    System.out.println("Paikassa 2 on " + luku);
    luvut.set(21,4);
  }
}
```

16.4 Javan tietorakenneluokat ja algoritmeja

Koska erilaisten dynaamisten tietorakenteiden (vrt. Taulukko.java) käyttö on erittäin yleistä, on Javaan standardiin lisätty joukko tietorakenteita. Jotta nämä tietorakenteet pystyisivät tallentamaan erilaisia tyyppejä on niistä tehty sellaisia, että ne tallentavat Javan kaikkien luokkien kantaluokan `Object`-luokan viitteitä.

Meidänkin esimerkissämme `Jasenet` ja `Harrastukset` eroavat toisistaan vain hyvin vähän. Ero on itse asiassa muutaman `Jasen` –sanan muuttuminen `Harrastus` –sanaksi. Jos olisimme olleet tarpeeksi ”ovelia”, olisimme voineet tehdä vain yhden geneerisen tietorakenteen, joista olisi sitten luotu kaksi erilaista esiintymää.

16.4.1 `vector`-luokka

Seuraavassa on Taulukko.javaa vastaava esimerkki toteutettu `Vector`-luokan avulla.

dyna\VectorMalli.java – vector-luokka

```
import java.util.Vector;
import java.util.Iterator;
import java.io.*;
import fi.jyu.mit.ohj2.*;

/**
 * Esimerkki Javan vektorin käytöstä
 * @author Vesa Lappalainen
 * @version 1.0, 02.03.2002
 */
public class VectorMalli {

  public static void tulosta(OutputStream os,  Vector luvut) {
    PrintStream out = Tiedosto.getPrintStream(os);
    for (Iterator i = luvut.iterator(); i.hasNext(); ) {
      int luku = ((Integer)i.next()).intValue();
      out.print(luku + " ");
    }
    out.println();
  }
```

```
  public static void main(String[] args) {
    Vector luvut = new Vector(7);
    try {
      luvut.add(new Integer(0)); luvut.add(new Integer(2));
      luvut.add(new Integer(99));
    } catch ( Exception e ) {
      System.out.println("Virhe: " + e.getMessage());
    }
    System.out.println(luvut);
    luvut.set(1,new Integer(4));
    System.out.println(luvut);
    int luku = ((Integer)luvut.get(2)).intValue();
    System.out.println("Paikassa 2 on " + luku);
    tulosta(System.out,luvut);
    luvut.set(21,new Integer(4));
  }
}
```

16.4.2 Iteraattori

Esimerkissä taulukon tulostus on tehty *iteraattorin* avulla. *Iteraattorin* ideana on tarjota tietty, erittäin suppea joukko operaatiota, joita siihen voidaan kohdistaa. Näin samalla rajapinnalla varustettu *iteraattori* voidaan toteuttaa hyvin erilaisille tietorakenteille esimerkiksi taulukoille ja linkitetyille listoille. *Iteraattorille* esitettyjä suomennoksia ovat esimerkiksi selain ja vipellin.

Vektorin tapauksessa tietorakenne voitaisiin käydä läpi myös taulukkomaisesti,

```
  for (i=0; i<luvut.size();i++)
    out.print(((Integer)(luvut.get(i)).intValue());
```

mutta tällöin tietorakenteen vaihtaminen esimerkiksi linkitetyksi listaksi vaatisi muutoksia `tulosta`-aliohjelmaan. Eli aina kun mahdollista, kannattaa välttää käyttämästä sitä tietoa, mikä tietorakenne on käytössä.

16.4.3 ListArray

Edellinen esimerkki voitaisiin toteuttaa myös `ListArray`-rakenteella:

dyna\ArrayListMalli.java – ListArray-luokka

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Collection;
import java.io.*;
import fi.jyu.mit.ohj2.*;

/**
 * Esimerkki Javan ArrayListin käytöstä
 * @author Vesa Lappalainen
 * @version 1.0, 02.03.2002
 */

public class ArrayListMalli {

  public static void tulosta(OutputStream os,  Collection luvut) {
    PrintStream out = Tiedosto.getPrintStream(os);
    for (Iterator i = luvut.iterator(); i.hasNext(); ) {
      int luku = ((Integer)i.next()).intValue();
      out.print(luku + " ");
    }
    out.println();
  }
```

```
  public static void main(String[] args) {
    ArrayList luvut = new ArrayList(7);
    try {
      luvut.add(new Integer(0)); luvut.add(new Integer(2));
      luvut.add(new Integer(99));
    } catch ( Exception e ) {
      System.out.println("Virhe: " + e.getMessage());
    }
    System.out.println(luvut);
    luvut.set(1,new Integer(4));
    System.out.println(luvut);
    int luku = ((Integer)luvut.get(2)).intValue();
    System.out.println("Paikassa 2 on " + luku);
    tulosta(System.out,luvut);
    luvut.set(21,new Integer(4));
  }
}
```

16.4.4 Algoritmit

Kun tietorakenteelta oletetaan tietty rajapinta, voidaan sille suorittaa sopiva algoritmi, esimerkiksi lajittelu, tietämättä tietorakenteen yksityiskohtia:

```
import java.util.ArrayList;
import java.util.Vector;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.io.*;
import fi.jyu.mit.ohj2.*;

/**
 * Esimerkki Javan algoritmien käytöstä
 * @author Vesa Lappalainen
 * @version 1.0, 05.03.2002
 */

public class AlgoritmiMalli {

  /**
   * Luokka joka vertailee kahta kokonaislukuoliota ja
   * palauttaa niiden järjestyksen niin, että lajittelu menee
   * laskevaan järjestykseen.
   */
  public static class LaskevaInt implements Comparator {
    public int compare(Object o1, Object o2) {
      return ((Integer)o2).intValue() - ((Integer)o1).intValue();
    }
  }

  public static void tulosta(OutputStream os,  Collection luvut) {
    PrintStream out = Tiedosto.getPrintStream(os);
    for (Iterator i = luvut.iterator(); i.hasNext(); ) {
      int luku = ((Integer)i.next()).intValue();
      out.print(luku + " ");
    }
    out.println();
  }
```

```
  public static void main(String[] args) {
    ArrayList luvut = new ArrayList();
    try {
      luvut.add(new Integer(0));  luvut.add(new Integer(2));
      luvut.add(new Integer(99)); luvut.add(new Integer(7));
      luvut.add(new Integer(22)); luvut.add(new Integer(71));
    } catch ( Exception e ) {
      System.out.println("Virhe: " + e.getMessage());
    }
    System.out.println(luvut);                          // [0, 2, 99, 7, 22, 71]

    Collections.sort(luvut);
    tulosta(System.out,luvut);                          // 0 2 7 22 71 99
    Collections.sort(luvut,new LaskevaInt());
    tulosta(System.out,luvut);                          // 99 71 22 7 2 0
    Collections.shuffle(luvut);
    tulosta(System.out,luvut);                          // 99 2 7 71 0 22
    Collections.sort(luvut,Collections.reverseOrder());
    tulosta(System.out,luvut);                          // 99 71 22 7 2 0
    Collections.reverse(luvut);
    tulosta(System.out,luvut);                          // 0 2 7 22 71 99

    int suurin = ((Integer)Collections.max(luvut)).intValue();
    System.out.println("Suurin = " + suurin);           // Suurin = 99
    int pienin = ((Integer)Collections.min(luvut)).intValue();
    System.out.println("Pienin = " + pienin);           // Pienin = 0
    pienin = ((Integer)Collections.max(luvut,new LaskevaInt())).intValue();
    System.out.println("Pienin = " + pienin);           // Pienin = 0

    List luvut2 = new LinkedList();
    luvut2.addAll(0,luvut);
    tulosta(System.out,luvut2);                         // 0 2 7 22 71 99
    luvut2 = luvut.subList(2,5);
    tulosta(System.out,luvut2);                         // 7 22 71

  }
}
```

16.5 Tietovirta parametrinä

Metodi `tulosta` on esitelty

```
  public static void tulosta(OutputStream os,  Collection luvut) {
```

Näin voidaan tulostusvaiheessa valita mille laitteelle tulostetaan.

17. Tiedostot

Tällää tiedostoon rivitkin
virtaan viskaile tavuset
sinne sullo saatavaksi
muitten metsästettäväksi.

Rivit riivi tiedostosta
ime virrasta tavuset
sieltä sieppaa saataville
levyltä lue lukuset.

Pistele rivit paloiksi
kunnolla katko kummajaiset
sanat sieltä sommittele
numerotkin napsi niistä.

Mitä tässä luvussa käsitellään?

- Tiedostojen käsittely Javan tietovirroilla
- Tiedostot joissa rivillä monta kenttää

Syntaksi:

```
Tied. avaaminen luku: f = new BufferedReader(new FileReader(nimi));
                kirj. f = new PrintWriter(new FileWriter(nimi))
                jatk. f = new PrintWriter(new FileWriter(nimi,true))
Stream          kirj: f = new PrintStream(new FileOutputStream(nimi));
                jatk: f = new PrintStream(new FileOutputStream(nimi),true);
Lukeminen       s = f.readLine();
Kirjoittaminen  f.println(s);
Sulkeminen      f.close();  // aina finally lohkossa!
```

Luvun esimerkkikoodit:

http://www.mit.jyu.fi/~vesal/kurssit/ohj2/moniste/esim/tiedosto/

Pyrimme seuraavaksi lisäämään kerho–ohjelmaamme tiedostosta lukemisen ja tiedostoon tallettamisen. Tätä varten tutustumme ensin lukemiseen mahdollisesti liittyviin ongelmiin.

17.1 Tiedostojen käsittely

Tiedostojen käsittely ei eroa päätesyötöstä ja tulostuksesta, siis tiedostojen käyttöä ei ole mitenkään syytä vierastaa! Itse asiassa päätesyöttö ja tulostus ovat vain `stdin` ja `stdout` –nimisten tiedostojen käsittelyä.

Tiedostoja on kahta päätyyppiä: tekstitiedostot ja binääritiedostot. Tekstitiedostojen etu on siinä, että ne käyttäytyvät täsmälleen samoin kuin päätesyöttökin. Binääritiedoston etu on taas siinä, että talletus saattaa viedä vähemmän tilaa (erityisesti numeerista muotoa olevat tyypit) ja suorasaannin käyttö on järkevämpää.

Keskitymme aluksi tekstitiedostoihin, koska niitä voidaan tarvittaessa editoida tavallisella editorilla. Näin ohjelman testaaminen helpottuu, kun tiedosto voidaan rakentaa valmiiksi ennen kuin ohjelmassa edes on päätesyöttöä lukevaa osaa.

17.1.1 Lukeminen

Muutamme hieman alkuperäistä suunnitelmaamme jäsenrekisteritiedoston sisällöstä:

```
Kelmien kerho ry
100
; Kenttien järjestys tiedostossa on seuraava:
id| nimi         |sotu       |katuosoite  |postinumero|postiosoite|kotipuhelin...
1|Ankka Aku      |010245-123U|Ankkakuja 6 |12345      |ANKKALINNA |12-12324   ...
2|Susi Sepe      |020347-123T|            |12555      |Perämetsä  |           ...
3|Ponteva Veli   |030455-3333|            |12555      |Perämetsä  |           ...
```

Olemme lisänneet rivin, jossa kerrotaan tiedoston maksimikoko. Tätähän tarvittiin jäsenlistan luomisessa. Nyt kokoa voidaan tarvittaessa muuttaa tiedostosta tekstieditorilla tarvitsematta tietää ohjelmoinnista mitään.

Tiedoston sisällössä on kuitenkin pieni ongelma: siinä on sekaisin sekä puhtaita merkkijonoja, numeroita että tietuetyyppisiä rivejä. Vaikka kielessä onkin työkalut sekä numeeristen tietojen lukemiseksi tiedostosta, että merkkijonojen lukemiseen, nämä työkalut eivät välttämättä toimi yksiin. Siksi usein kannattaa käyttää lukemiseen vain yhtä työkalua, joka useimmiten on kokonaisen tiedoston rivin lukeminen.

17.2 Tiedostojen käsittely Javan tietovirroilla

Javan `IO`-systeemi on varsin monimutkainen. Erilaisia tietovirtoja on yli 60 kappaletta. Alimman tason virta-luokat ovat abstrakteja luokkia määräten vain virtojen rajapinnan. Ylemmällä tasolla hoidetaan fyysistä lukemista ja kirjoittamista. Fyysinen lukeminen ja kirjoittaminen voi tarkoittaa levyn käyttöä, verkon käyttöä tai muiden `IO`-porttien käyttöä. Seuraavaksi ylemmällä tasolla tarjotaan yksinkertaisempaa rajapintaa esimerkiksi rivien käsittelyyn. Siksi virtoja käytettäessä niitä pitää kerrostaa.

Kun perustoimet on saatu tehtyä, on tiedostojen käsittely Javassa esimerkiksi `System.in` ja `System.out` –tietovirtoja vastaavien tietovirtojen käsittelyä.

Olkoon meillä tiedosto nimeltä `luvut.dat`:

```
13.4
23.6
kissa
1.9
<EOF>     <-  ei aina välttämättä mikään merkki
```

Kirjoitetaan esimerkkitiedoston luvut lukeva ohjelma *Java*n tietovirroilla. Tarkoitus on hylätä ne rivit, joilla ei ole pelkästään reaalilukua:

tiedosto\Tied_ka.java - Lukujen lukeminen tiedostosta

```
import java.io.*;
import fi.jyu.mit.ohj2.Mjonot;
/**
 * Lukujen lukeminen tiedostosta
 * @author Vesa Lappalainen
 * @version 1.0, 07.03.2003
 */
public class Tied_ka {

  public static void main(String[] args)  {

    BufferedReader fi;

    try {  // Avataan tiedosto lukemista varten
      fi = new BufferedReader(new FileReader("luvut.dat"));
    } catch (FileNotFoundException ex) {
      System.out.println("Tiedosto ei aukea!");
      return;
    }

    double summa=0;
    int n=0;

    try {
      String s; double luku;
      while ( ( s = fi.readLine() ) != null ) {
        try {
          luku = Double.parseDouble(s);
        } catch (NumberFormatException ex) {
          continue;
        }
        summa += luku;
        n++;
      }
    } catch (IOException ex) {
      System.out.println("Virhe tiedostoa luettaessa!");
    } finally {  // Aina ehdottomasti finally:ssa resurssien vapautus
      try {
        fi.close();  // tiedoston sulkeminen heti kun sitä ei enää tarvita
      } catch (IOException ex) {
        System.out.println("Tiedostoa ei saa suljettua!");
      }
    }

    double ka = 0;
    if ( n > 0 ) ka = summa/n;
    System.out.println("Lukuja oli " + n + " kappaletta.");
    System.out.println("Niiden summa oli " + Mjonot.fmt(summa,4,2));
    System.out.println("ja keskiarvo oli " + Mjonot.fmt(ka,4,2));

  }
}
```

Tehtävä 17.1 Tiedoston lukujen summa

1. Muuta tiedoston Tied_ka.java –ohjelmaa siten, että väärän rivin kohdalla tulostetaan väärä rivi ja lopetetaan koko ohjelma.
2. Muuta edelleen ohjelmaa siten, että väärät rivit tulostetaan näyttöön:

```
Tiedostossa oli seuraavat laittomat rivit:
kissa
Lukuja oli...
```

   Ilmoitusta ei tietenkään tule, mikäli tiedostossa ei ole laittomia merkkejä. Tyhjää riviä ei tulkita vääräksi riviksi.

17.2.1 Tiedoston avaaminen muodostajassa

Tiedosto voidaan siis avata heti kun tiedostoa vastaava tietovirta luodaan:

```
new FileReader("luvut.dat")
```

Parametri `"luvut.dat"` on tiedoston nimi levyllä. Nimi voi sisältää myös hakemistopolun, mutta tätä kannattaa välttää, koska hakemistot eivät välttämättä ole samanlaisia kaikkien käyttäjien koneissa. Jos hakemistopolkuja käyttää, niin erottimena kannattaa käyttää `/`-merkkiä. Samoin kannattaa olla tarkkana isojen ja pienien kirjainten kanssa, sillä useissa käyttöjärjestelmissä kirjainten koolla on väliä.

Lukemista varten avattaessa tiedoston täytyy olla olemassa tai avaus epäonnistuu. Avauksen epäonnistumisesta heitetään `FileNotFoundException`-poikkeus.

17.2.2 Tiedostosta lukeminen.

Tiedostosta lukeminen on jälleen analogista päätesyötön kanssa:

```
s = fi.readLine();
```

Mikäli tiedosto on loppu, saa `s` `null`-arvon.

17.2.3 Tiedoston lopun testaaminen

Helpoin ratkaisu on perustaa lukusilmukka siihen, että yritetään lukea kokonainen tiedoston rivi ja jos tämä epäonnistuu, on tiedostokin todennäköisesti loppu.

```
while ( ( s = fi.readLine() ) != null ) {
  ... käsittele jonoa s
}
```

17.2.4 Tiedostoon kirjoittaminen

Vastaavasti kirjoittamista varten avattuun tiedostoon kirjoitettaisiin

```
PrintStream fo;
...
fo = new PrintStream(new FileOutputStream("taulu.txt"));
// avataan tiedosto kirjoittamista varten
// avauksessa vanha tiedosto tuhoutuu
```

Mikäli avattaessa tiedostoa kirjoittamista varten, ei haluta tuhota vanhaa sisältöä, vaan kirjoittaa vanhan perään, käytetään avauksessa toista parametria, jolla kerrotaan halutaanko kirjoittaa edellisen tiedoston perään (*append*):

```
fo = new PrintStream(new FileOutputStream("taulu.txt",true));
// avataan perään kirjoittamista varten
```

Tiedoston jatkaminen on erittäin kätevä esimerkiksi virhelogitiedostoja kirjoitettaessa.

tiedosto\Kertotaulu.java - Tiedostoon tulostaminen

```
import java.io.*;
/**
 * Ohjelmalla tulostetaan kertotaulu tiedostoon.  Jos tiedosto on
 * olemassa, jatketaan vanhan tiedoston perään.
 * @author Vesa Lappalainen
 * @version 1.0, 21.02.2003
 */
public class Kertotaulu {

  public static void main(String[] args)  {
    PrintStream fo = null;
    try {
      fo = new PrintStream(new FileOutputStream("taulu.txt",true));
    }  catch (FileNotFoundException ex) {
      System.out.println("Tiedosto ei aukea"); return;
    }

    int kerroin = 5;

    try {
      for (int i=0; i<10; i++)
        fo.println( i + "*" + kerroin + " = " + i*kerroin);
    } finally {
      fo.close();
    }
  }
}
```

Edellä voisi käyttää `PrintStream` virran sijasta `PrintWriter`-luokkaa, joka olisi yhteensopivampi `Reader`-luokan kanssa:

```
PrintWriter fo;
...
  fo = new PrintWriter(new FileWriter(nimi,true))
```

Kuitenkin `PrintStream` on taas yhteensopiva `System.out`:in kanssa, joten joissakin tapauksissa tämä puolustaa `PrintStream`-luokan käyttämistä.

Useimmiten kannattaa kaikki näyttöön tulostavat aliohjelmat/metodit kirjoittaa sellaiseksi, että niille viedään parametrinä se tietovirta, johon tulostetaan. Näin samalla aliohjelmalla voidaan helposti tulostaa sitten näyttöön tai tiedostoon tai jopa kirjoittimelle (joka on vain yksi tietovirta muiden joukossa, esim. *Windowsissa* PRN-niminen tiedosto).

17.2.5 `Tiedoston sulkeminen close`

Avattu tiedosto on aina lukemisen tai kirjoittamisen lopuksi syytä sulkea. Tiedoston käsittely on usein puskuroitua, eli esimerkiksi kirjoittaminen tapahtuu ensin apupuskuriin, josta se kirjoittuu fyysisesti levylle vain puskurin täyttyessä tai tiedoston sulkemisen yhteydessä. Käyttöjärjestelmä päivittää tiedoston koon levylle usein vasta sulkemisen jälkeen. Sulkemattoman tiedoston koko saattaa näyttää 0 tavua.

Javassa tiedoston sulkeminen pitää aina varmistaa `try-finally`-lohkolla:

```
... avaa tiedosto
try {
  ... käsittele tiedostoa
} finally {  // Aina ehdottomasti finally:ssa resurssien vapautus
  try {
    fi.close();  // tiedoston sulkeminen heti kun sitä ei enää tarvita
  } catch (IOException ex) {
    ... toimenpiteet jos tiedostoa ei saada suljettua
  }
}
```

Tiedosto kannattaa sulkea heti kun sen käyttö on loppu.

Tehtävä 17.2 Kommentit näytölle

Kirjoita ohjelma, joka kysyy tiedoston nimen ja tämän jälkeen tulostaa tiedostosta rivien **/********* ja------*/välisen osan näytölle.

17.3 Tiedoston yhdellä rivillä monta kenttää

Jäsenrekisterissä on tiedoston yhdellä rivillä useita kenttiä. Kentät saattavat olla myös eri tyyppisiä. Miten lukeminen hoidetaan varmimmin?

17.3.1 Ongelma

Olkoon meillä vaikkapa seuraavanlainen tiedosto:

tiedosto\tuotteet.dat - esimerkkitiedosto

```
Volvo |  12300 | 1
Audi  |  55700 | 2
Saab  |   1500 | 4
Volvo | 123400 | 1<EOF>
```

Tiedostoa voitaisiin periaatteessa niin että luetaan ensin yksi merkkijono, sitten tolppa, sitten reaaliluku, tolppa ja lopuksi kokonaisluku.

Ratkaisussa on kuitenkin seuraavia huonoja puolia:

- mikäli tiedoston loppu ei olekaan viimeisen rivin lopussa, tulee ”ylimääräisen” rivin käsittelystä ongelmia
- mikäli jokin rivi on väärää muotoa, menee ohjelma varsin sekaisin

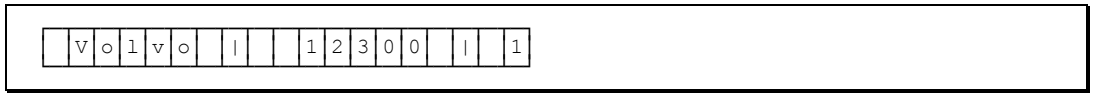
Tehtävä 17.3 Ohjelman "sekoaminen"

Jos esimerkin hahmotellussa ratkaisussa olisi silmukka, joka tulostaa tiedot kunkin lukemisen jälkeen, niin mitä tulostuisi seuraavasta tiedostosta:

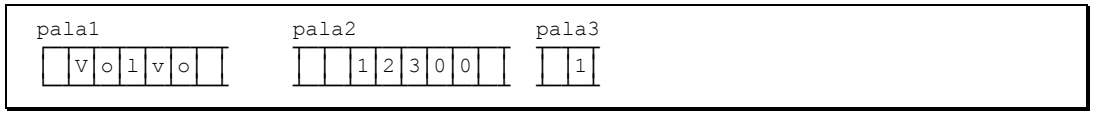
```
    Volvo |  12300 | 1
    Audi     55700 | 2
    Saab  |   1500 | 4
    Volvo | 123400 | 1
<EOF>
```

17.4 Merkkijonon paloittelu

Tutkitaanpa ongelmaa tarkemmin. Tiedostosta on siis luettu rivi, joka on muotoa



Jos saisimme erotettua tästä 3 merkkijonoa:



voisimme kustakin palasesta erikseen ottaa haluamme tiedot. Esimerkiksi 1. palasesta saadaan tuotteen nimi, kun siitä poistetaan turhat välilyönnit. Hinta saataisiin 2. palasesta kutsulla

```
sscanf(pala2.c_str(),"%lf",&hinta);
```

17.4.1 `parse`

Merkkijono pitää varsin usein muuttaa reaaliluvuksi tai kokonaisluvuksi. Java tarjoaa luokissa `Integer` ja `Double` mahdollisuuden muuttaa merkkijono vastaavaksi lukutyypiksi:

```
double d = Double.parseDouble(jono);
int i = Integer.parseInt(jono);
```

Edellän mainitut metodit heittävät poikkeuksen jos jono sisältää mitä tahansa muuta kuin pelkkiä lukuun kuuluvia merkkejä.

Siksi kirjoitammekin luokkaan Mjonot kaksi funktiota `erotaDouble` ja `erotaInt`:

```
public static double erotaDouble(String jono, double oletus) ...
public static double erotaInt(String jono, int oletus) ...
```

Jos funktio ei löydä merkkijonosta lukua, se palauttaa oletuksen. Nämä funktiot toimivat oikein myös seuraavien jonojen kanssa:

```
12.34 e   => 12.34
14 kpl    => 14
```

17.4.2 `erota`

Tehdään myös yleiskäyttöinen funktio `erota`, jonka tehtävä on ottaa merkkijonon alkuosa valittuun merkkiin saakka, poistaa valittu merkki ja palauttaa sitten funktion tuloksena tämä alkuosa. Itse merkkijonoon jää jäljelle ensimmäisen merkin jälkeinen osa. Funktio on kirjoitettu tiedostoon Mjonot.java:

```
  public static String erota(StringBuffer jono, char merkki,
                             boolean etsitakaperin) {
    if ( jono == null ) return "";
    int p;
    if ( !etsitakaperin ) p = jono.indexOf(""+merkki);
    else p = jono.lastIndexOf(""+merkki);
    String alku;
    if ( p < 0 ) {
      alku = jono.toString();
      jono.delete(0,jono.length());
      return alku;
    }
    alku = jono.substring(0,p);
    jono.delete(0,p+1);
    return alku;
  }

}
```

17.4.3 Esimerkki `erota`-funktion käytöstä

Kirjoitetaan lyhyt esimerkki, jolla demonstroidaan funktion käyttöä:

tiedosto\ErotaEsim.java - esimerkki erota-funktion käytöstä

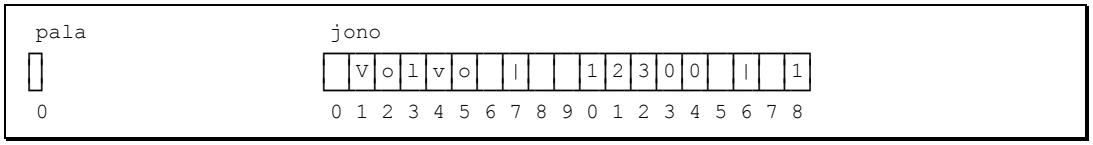
```
import fi.jyu.mit.ohj2.Mjonot;

/**
 * Ohjelmalla demonstroidaan erota-funktion toimintaa
 * @author Vesa Lappalainen
 * @version 1.0, 21.02.2003
 */
public class ErotaEsim {

  private static void tulosta(int n,String pala, StringBuffer jono)
  {
    int valeja = 10-pala.length();
    System.out.println(n + ": pala = '" + Mjonot.fmt(pala + "'",-10) +
                       " jono = '" + jono + "'");
  }

  public static void main(String[] args)  {
    StringBuffer jono = new StringBuffer(" Volvo |  12300 | 1");
    String pala="";                  tulosta(0,pala,jono);
    pala = Mjonot.erota(jono,'|');   tulosta(1,pala,jono);
    pala = Mjonot.erota(jono,'|');   tulosta(2,pala,jono);
    pala = Mjonot.erota(jono,'|');   tulosta(3,pala,jono);
    pala = Mjonot.erota(jono,'|');   tulosta(4,pala,jono);

  }
}
```

Ohjelma tulostaa:

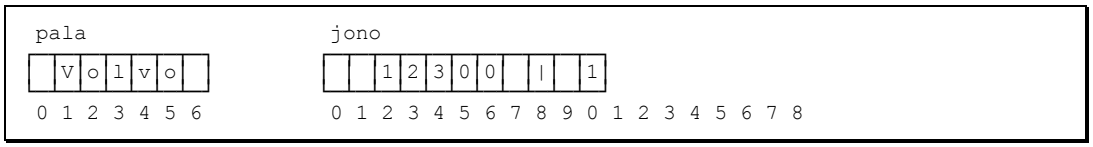
```
0: pala = ''          jono = ' Volvo |  12300 | 1'
1: pala = ' Volvo '   jono = '  12300 | 1'
2: pala = '  12300 '  jono = ' 1'
3: pala = ' 1'        jono = ''
4: pala = ''          jono = ''
```

17.4.4 Erota funktion toiminta vaihe vaiheelta

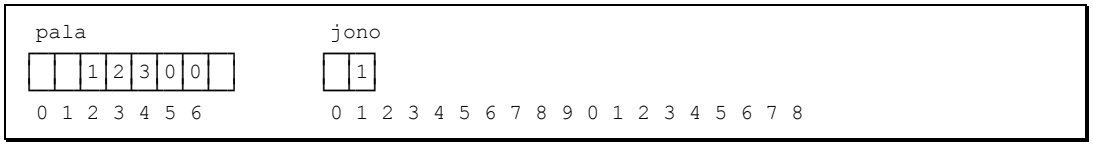
Ennen ensimmäistä kutsua tilanne on seuraava:



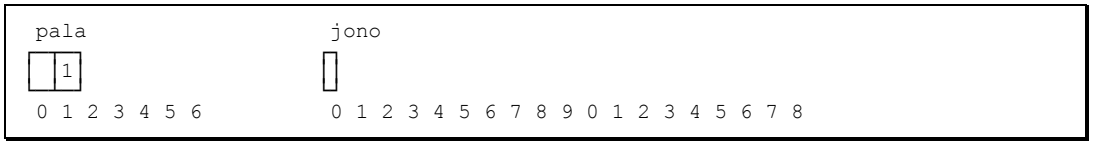
Ensimmäisessä kutsussa `erota`-funktio löytää etsittävän | -merkin paikasta 7. Merkit 0-6 kopioidaan funktion paluuarvoksi ja sitten jonosta tuhotaan merkit 0-7. Funktion paluuarvo sijoitetaan muuttujaan `pala`:



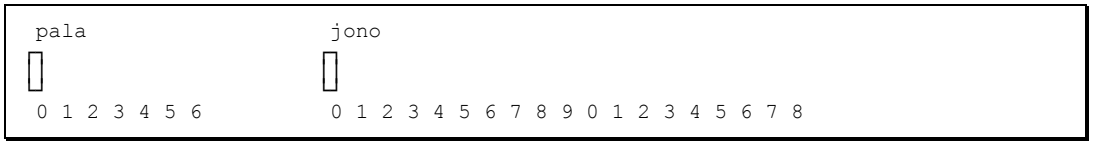
Seuraavalla kutsulla (kerta 2) |-merkki löytyy jonosta paikasta 8. Nyt merkit jonon merkit 0-7 kopioidaan funktion paluuarvoon ja merkki 8 tuhotaan. Kutsun jälkeen tilanne on:



Kolmannessa kutsussa merkkiä | ei enää löydy jonosta. Tämä ilmenee siitä, että `find`-metodi palauttaa arvon `string::npos` (*no position*), eli ei paikkaa. Näin koko jono kopioidaan funktion paluuarvoksi ja kutsun jälkeen tilanne on:



Vastaava toistuu neljännessä kutsussa, eli koko jono sitten kopioidaan paluuarvoksi ja tilanne on neljännen kutsun jälkeen:



Tämän jälkeen tilanne pysyy samana vaikka `erota`-funktiota kutsuttaisiin kuinka monta kertaa tahansa. Tästä saadaan se etu, että `erota`-funktiota voidaan turvallisesti kutsua kuinka monta kertaa tahansa, vaikkei jonosta enää palasia saataisikaan. Jos kutsua tehdään silmukassa, voidaan silmukan lopetusehdoksi kirjoittaa

```
while ( !jono.equals("") ) {
  pala = erota(jono,'|');
  System.out.println(pala);
}
```

17.4.5 Luvun erottaminen

Usein samassa jonossa on sekaisin lukuja ja merkkijonoja. Jotta käsittely saataisiin symmetrisemmäksi eri tyyppien välillä, niin luokkaan **`Mjonot`** on kirjoitettu myös polymorfiset funktiot:

```
StringBuffer jono = new StringBuffer("   Volvo   145  | 2000 e  | 3 kpl ")
String s = "";
double d = 0.0;
int kpl = 0;
s   = erota(jono,'|',s);      // "Volvo 145"
d   = erota(jono,'|',d);      // 2000
kpl = erota(jono,'|',kpl);    // 3
```

Idea on siinä, että jos myöhemmin huomataan vaikka että `kpl` pitäisi olla tyypiltään `double` eikä `int`, riittää vain muuttujan `kpl` tyypin vaihtaminen.

Jos halutaan käsitellä tilanteet, joissa joku kenttä onkin virheellistä muotoa, on edellisestä myös poikkeuksen heittävät muodot:

```
try {
  s   = erotaEx(jono,'|',s);    // "Volvo 145"
  d   = erotaEx(jono,'|',d);    // 2000
  kpl = erotaEx(jono,'|',kpl);    // 3
} catch  ( NumberFormatException ex ) {
  System.out.println(ex.getMessage());
}
```

17.5 Lukeminen ja paloittelu

Nyt voimme toteuttaa "tuotetiedoston" lukevan ohjelman *Java*n tietovirroilla ja funktioiden `erotaEx` avulla:

tiedosto\LueTuote.java - esimerkki tiedoston lukemisesta

```
import fi.jyu.mit.ohj2.*;
import java.io.*;

/**
 * Ohjelmalla luetaan tuotetiedosto ja tulostetaan tuotteet
 * @author Vesa Lappalainen
 * @version 1.0, 21.02.2003
 */
public class LueTuote {

  public static boolean tulosta_tuotteet() {
    String srivi,pala;
    String nimike; double hinta; int kpl;

    BufferedReader fi = Tiedosto.avaa_lukemista_varten("tuotteet.dat");
    if ( fi == null ) return false;

    System.out.println(); System.out.println(); System.out.println();
    System.out.println("-----------------------------------------");
```

```
    try {
      while ( ( srivi = fi.readLine() ) != null ) {
        StringBuffer rivi = new StringBuffer(srivi);
        try {
          nimike = Mjonot.erotaEx(rivi,'|',"");
          hinta  = Mjonot.erotaEx(rivi,'|',0.0);
          kpl    = Mjonot.erotaEx(rivi,'|',0);
        } catch (NumberFormatException ex) {
          System.out.println("Virhe: " + ex.getMessage());
          continue;
        }
        System.out.println(Mjonot.fmt(nimike,-20) +" " + Mjonot.fmt(hinta,7,0) +
                           Mjonot.fmt(kpl,4));
      }
    } catch (IOException ex) {
      System.out.println("Vikaa tiedostoa luettaessa");
    } finally {
      try {
        fi.close();
      } catch (IOException ex) {
        System.out.println("Ongelmia tiedoston sulkemisessa");
      }
    }

    System.out.println("-----------------------------------------");
    System.out.println(); System.out.println(); System.out.println();

    return true;
  }

  public static void main(String[] args)  {
    if ( !tulosta_tuotteet() ) System.out.println("Tuotteita ei saada luetuksi");
  }
}
```

Ohjelma tulostaa:

```
-----------------------------------------
Volvo                 12300    1
Audi                  55700    2
Saab                   1500    4
Volvo                123400    1
-----------------------------------------
```

17.5.1 Olio joka lukee itsensä

Muutetaan vielä tuotteiden lukua oliomaisemmaksi, eli annetaan tuotteelle kuuluvat tehtävät kokonaan `Tuote`–luokan vastuulle, samalla lisätään `Tuotteet`–luokka.

tiedosto\LueRek.java - esimerkki oliosta joka käsittelee tiedostoa

```
import java.io.*;
import fi.jyu.mit.ohj2.*;
/**
 * Esimerkki oliosta joka käsittelee tiedostoa
 * @author Vesa Lappalainen
 * @version 1.0, 09.03.2003
 */
public class LueRek {

  static public class Tuote {
    private String nimike = "";
    private double hinta = 0.0;
    private int kpl = 0;

    public Tuote() {}
    public Tuote(String rivi) { parse(rivi); }
```

```
    public void parse(String s) throws NumberFormatException {
      StringBuffer sb = new StringBuffer(s);
      nimike = Mjonot.erotaEx(sb,'|',nimike);
      hinta  = Mjonot.erotaEx(sb,'|',hinta);
      kpl    = Mjonot.erotaEx(sb,'|',kpl);
    }

    public String toPrintString() {
      return Mjonot.fmt(nimike,-20) + " " + Mjonot.fmt(hinta,7,0) +
             Mjonot.fmt(kpl,4);
    }

  }

  static public class Tuotteet {
    private String nimi = "";

    public Tuotteet(String nimi) { this.nimi = nimi; }

    public boolean tulosta(OutputStream os) {
      PrintStream out = Tiedosto.getPrintStream(os);
      BufferedReader fi = Tiedosto.avaa_lukemista_varten("tuotteet.dat");
      if ( fi == null ) return false;

      out.println(); out.println(); out.println();
      out.println("-----------------------------------------");

      try {
        String rivi;  Tuote tuote;
        while ( ( rivi = fi.readLine() ) != null ) {
          try {
            tuote = new Tuote(rivi);
          } catch (NumberFormatException ex) {
            System.err.println("Virhe: " + ex.getMessage());
            continue;
          }
          out.println(tuote.toPrintString());
        }
      } catch (IOException ex) {
        System.err.println("Vikaa tiedostoa luettaessa");
      } finally {
        try {
          fi.close();
        } catch (IOException ex) {
          System.err.println("Ongelmia tiedoston sulkemisessa");
        }
      }

      out.println("-----------------------------------------");
      out.println(); System.out.println(); System.out.println();

      return true;
    }

  }

  public static void main(String[] args)  {
    Tuotteet tuotteet = new Tuotteet("tuotteet.dat");
    if ( !tuotteet.tulosta(System.out) ) {
      System.err.println("Tuotteita ei saada luetuksi");
    }
  }
}
```

17.6 Esimerkki tiedoston lukemisesta

Seuraavaksi kirjoitamme ohjelman, jossa tulee esiin varsin yleinen ongelma: tietueen etsiminen joukosta. Kirjoitamme edellisiä esimerkkejä vastaavan ohjelman, jossa tavallisen tulostuksen sijasta tulostetaan kunkin tuoteluokan yhteistilanne.

tiedosto\LueTRek.java - esimerkki tiedoston lukemisesta

```
import java.io.*;
import fi.jyu.mit.ohj2.*;
/**
 * Ohjelma lukee tiedostoa tuotteet.dat, joka on muotoa:
 * <pre>
 *   Volvo |  12300 | 1
 *   Audi  |  55700 | 2
 *   Saab  |   1500 | 4
 *   Volvo | 123400 | 1
 * </pre>
 * Ohjelma tulostaa kuhunkin tuoteluokkaan kuuluvien tuotteiden
 * yhteishinnat ja kappalemäärät sekä koko varaston yhteishinnan
 * ja kappalemäärän.  Eli em. tiedostosta tulostetaan:
 * <pre>
 * ------------------------------------------
 * Volvo                135700    2
 * Audi                 111400    2
 * Saab                   6000    4
 * ------------------------------------------
 * Yhteensä             253100    8
 * ------------------------------------------
 *</pre>
 * @author Vesa Lappalainen
 * @version 1.0, 09.03.2003
 */
public class LueTRek {

  /*****************************************************************************/
  /**
   * Luokka yhden tuotteen tiedoille.
   */
  static public class Tuote {
    private String nimike = "";
    private double hinta = 0.0;
    private int kpl = 0;

    public Tuote() {}
    public Tuote(String rivi) {
      try {
        parse(rivi);
      } catch (NumberFormatException ex) {
      }
    }

    public void parse(String s) throws NumberFormatException {
      StringBuffer sb = new StringBuffer(s);
      nimike = Mjonot.erotaEx(sb,'|',nimike);
      hinta  = Mjonot.erotaEx(sb,'|',hinta);
      kpl    = Mjonot.erotaEx(sb,'|',kpl);
    }

    public String toPrintString() {
      return Mjonot.fmt(nimike,-20) + " " + Mjonot.fmt(hinta,7,0) +
             Mjonot.fmt(kpl,4);
    }

    public void ynnaa(Tuote tuote) {
      hinta += tuote.hinta * tuote.kpl;
      kpl   += tuote.kpl;
    }

    public String getNimike() { return nimike; }
    public void setNimike(String nimike) { this.nimike = nimike; }

  }
```

```
/*****************************************************************************/
/**
 * Luokka joka säilyttää kunkin ero tuotteen yhteissumman ja lukumäär'n
 * sekä kaikkien tuotteiden yhteissumman ja lukumäärän
 */
static public class Tuotteet {
  private String nimi = "";
  private int lkm;
  private Tuote alkiot[];
  private Tuote yhteensa = new Tuote("Yhteensä");

  public Tuotteet(String nimi) {
    this.nimi = nimi;
    alkiot = new Tuote[10];
  }

  public int etsi(String tnimi) {
    for (int i=0; i<lkm; i++)
      if ( alkiot[i].getNimike().equalsIgnoreCase(tnimi) ) return i;
    return -1;
  }

  public int lisaa(String tnimi) {
    if ( alkiot.length <= lkm ) return -1;
    alkiot[lkm] = new Tuote(tnimi);
    return lkm++;
  }

  public boolean ynnaa(Tuote tuote) {
    if ( tuote.getNimike().equals("") ) return false;
    int i = etsi(tuote.getNimike());
    if ( i < 0 ) i = lisaa(tuote.getNimike());
    if ( i < 0 ) return false;

    alkiot[i].ynnaa(tuote);
    yhteensa.ynnaa(tuote);
    return true;
  }


  public boolean lue() {
    BufferedReader fi = Tiedosto.avaa_lukemista_varten("tuotteet.dat");
    if ( fi == null ) return false;

    try {
      String rivi;  Tuote tuote = new Tuote();
      while ( ( rivi = fi.readLine() ) != null ) {
        try {
          tuote.parse(rivi);
          ynnaa(tuote);
        } catch (NumberFormatException ex) {
          System.err.println("Rivillä jotakin pielessä " + rivi + "  " +
                             ex.getMessage());
          continue;
        }
      }
    } catch (IOException ex) {
      System.err.println("Vikaa tiedostoa luettaessa");
    } finally {
      try {
        fi.close();
      } catch (IOException ex) {
        System.err.println("Ongelmia tiedoston sulkemisessa");
      }
    }

    return true;
  }
```

```
    public void tulosta(OutputStream os) {
      PrintStream out = Tiedosto.getPrintStream(os);

      out.println(); out.println(); out.println();
      out.println("-----------------------------------------");

      for (int i=0; i<lkm; i++)
        out.println(alkiot[i].toPrintString());

      out.println("-----------------------------------------");
      out.println(yhteensa.toPrintString());
      out.println("-----------------------------------------");
      out.println(); System.out.println(); System.out.println();
    }

  }

  /*****************************************************************************/
  public static void main(String[] args)  {
    Tuotteet varasto = new Tuotteet("tuotteet.dat");
    if ( !varasto.lue() ) {
      System.err.println("Tuotteita ei saada luetuksi");
      return;
    }
    varasto.tulosta(System.out);
  }
}
```

Tehtävä 17.4 Tietorakenne

Piirrä kuva `Tuotteet` –luokan tietorakenteesta.

Tehtävä 17.5 Perintä

Miten voisit perinnän avulla saada tiedoston luerek.cpp luokasta `Tuote` tiedoston LueTRek.java vastaavan luokan (tietysti eri nimelle, esim. `RekTuote`). Mitä muutoksia olisi hyvä tehdä alkuperäisessä `Tuote`–luokassa.

Tehtävä 17.6 Tunnistenumero

Lisää LueTRek.java–ohjelmaan tunnistenumeron käsittely mahdollista tulevaa relaatiokäyttöä varten.

Tehtävä 17.7 Mittakaava

Kirjoita mittakaavaohjelma, jossa on vakiotaulukko

yks	mm
mm	1.0
cm	10.0
dm	100.0
m	1000.0
inch	25.4

ja jonka toiminta näyttäisi seuraavalta:

```
...
Mittakaava ja matka>1:10000 10 cm[RET]
Matka maastossa on 1.00 km.
Mittakaava ja matka>1:200000 20[RET]
Matka maastossa on 4.00 km.
Mittakaava ja matka>loppu[RET]
Kiitos!
```

Muuta ohjelmaa siten, että yksiköiden muunnostaulukko luetaan ohjelman aluksi tiedostosta `muunnos.dat`.

Muuta ohjelmaa vielä siten, että mikäli mittakaava jätetään antamatta, käytetään edellisellä kerralla annettua mittakaavaa ja ensimmäinen luku onkin matka.

```
...
Mittakaava ja matka>1:10000 10 cm[RET]
Matka maastossa on 1.00 km.
Mittakaava ja matka>0.20 dm[RET]
Matka maastossa on 0.20 km.
Mittakaava ja matka>loppu[RET]
Kiitos!
```

Hakemisto