Jyväskylän yliopisto
Tietotekniikan laitos

Luentomoniste 11

University of Jyväskylä
Department of Mathematical
Information Technology
Lecture Notes 11

OHJELMOINTI 2

uudistettu painos

Vesa Lappalainen ja Santtu Viitanen

Jyväskylä 2012

Jyväskylän yliopisto
Tietotekniikan laitos

Luentomoniste 11

University of Jyväskylä
Department of Mathematical Information Technology
Lecture Notes 11

OHJELMOINTI 2

uudistettu painos

Vesa Lappalainen ja Santtu Viitanen

Jyväskylä 2012

Raino A. E. Mäkinen
Jyväskylän yliopisto
Tietotekniikan laitos
PL 35 (Agora)
40014 Jyväskylän yliopisto
fax (014) 260 4980
http://www.mit.jyu.fi

Copyright © 2012 Vesa Lappalainen, Santtu Viitanen ja
Jyväskylän yliopisto
ISBN 978-951-39-4624-1 (uudistettu painos)
ISSN 1456-9787

Jyväskylän yliopistopaino
Jyväskylä 2012

Sisällys

Lukijalle ... 1
1. Johdanto ... 5
1.1 Ketterät menetelmät ... 6
1.2 Extreme Programming ... 6
1.2.1 Kurssilla sovellettavia XP:n käytäntöjä ... 6
1.3 Ohjelman suunnittelu ... 7
1.4 Työkalun valinta ... 7
1.5 Koodaus ... 8
1.6 Testaus ... 8
1.7 Hyväksymistestaus ja palaute ... 9
1.8 Käyttöönotto ... 9
1.9 Ylläpito ... 9
1.10 Yhteenveto, kielellä ei väliä ... 10
2. Kerhon jäsenrekisteri ... 11
2.1 Tehtävän tarkennus ... 11
2.2 Työkalun valinta, vaihtoehtoja ... 12
2.3 Tietorakenteet ja tiedostot ... 12
2.4 Käyttöohje ja käyttöliittymä ... 13
2.4.1 Ohjelman käynnistys ... 13
2.4.2 Hakeminen ... 15
2.4.3 Muokkaaminen ... 15
2.4.4 Lisää uusi jäsen ... 16
2.4.5 Poista jäsen ... 16
2.4.6 Tulosta ... 16
2.4.7 Lopeta ... 16
2.4.8 Apua ... 16
2.4.9 Tietoja ... 16
2.5 Hyväksymistestaus ... 16
2.5.1 Tyypillisiä vikoja ... 16
2.6 Tarvittavien algoritmien hahmottaminen ... 17
2.6.1 Ylemmän tason aliohjelmat ... 17
2.6.2 Alemman tason aliohjelmat ... 17
2.7 Koodaus ohjelmointikielelle ... 18
2.8 Varautuminen tulevaan, eli relaatiotietomalli ... 18
2.8.1 Kaikki samassa tietueessa ... 18
2.8.2 Erimalliset tietueet ... 18
2.8.3 Relaatiomalli tiedostoon ... 19
2.8.4 XML-muotoinen tiedosto ... 20
2.9 Graafiset käyttöliittymät ... 21
2.9.1 Komponentit ... 21
2.9.2 Omat komponentit ... 23
2.9.3 Graafisten käyttöliittymien suunnittelutyökalut ... 23
2.9.4 Tapahtumat ... 25
3. Algoritmin suunnittelu ... 27
3.1 Algoritmi ... 27
3.2 Lajittelu yleisesti ... 28
3.2.1 Nimien ja numeroiden vertaus ... 28

3.2.2 Algoritmin sanallinen versio on kuvaavampi! 29
3.2.3 Numeroiden järjestäminen 29
3.2.4 Kuplalajittelu 30
3.2.5 Lajittelu avaimen mukaan 31
3.2.6 Algoritmin parantaminen 31
3.3 Algoritmin tarkentaminen 31
3.3.1 Pienimmän etsiminen 31
3.3.2 Paikalleen sijoittaminen 32
3.4 Haku järjestetystä joukosta 32
3.4.1 Suora haku 32
3.4.2 Puolitushaku 32
3.5 Yhteenveto 33
4. Algoritmeissa tarvittavia rakenteita 35
4.1 Ehtolauseet 35
4.2 Valintalauseet 36
4.3 Silmukat 37
4.4 Muuttuja-käsite 37
4.4.1 Yksinkertaiset muuttujat 37
4.4.2 Pöytätesti 38
4.4.3 Yksiulotteiset taulukot, alkiot rivissä 39
4.4.4 Osoittimet 41
4.4.5 Moniulotteiset taulukot 42
4.4.6 Sekarakenteet 44
4.5 Osoittimista ja indekseistä 45
4.6 Aliohjelmat avuksi 46
4.7 Vaihtoehtojen tutkiminen totuustaulun avulla 46
4.7.1 Kaikkien vaihtoehtojen kirjaaminen 46
4.7.2 Vaihtoehtojen lukumäärä 47
4.7.3 Useita vaihtoehtoja samalla muuttujalla 48
4.7.4 Loogiset operaatiot 49
4.8 Muistele tätä 50
5. Esimerkkejä eri kielistä 51
5.1 Esimerkkiohjelmat 51
5.1.1 C 51
5.1.2 C++ 52
5.1.3 Java 52
5.1.4 C# 52
5.1.5 Pascal 52
5.1.6 Fortran 53
5.1.7 ADA 53
5.1.8 BASIC 53
5.1.9 APL 53
5.1.10 Modula–2 53
5.1.11 Lisp 53
5.1.12 FORTH 54
5.1.13 Assembler 54
5.2 Käytettävän kielen valinta 54
6. Java –kielen alkeita 57
6.1 C#:sta Javaan 57
6.2 Hello World! Java ja C#–kielillä 58

6.3 Tekstitiedostosta toimivaksi konekieliseksi versioksi ... 59
6.3.1 Kirjoittaminen ... 59
6.3.2 Kääntäminen ... 59
6.3.3 Linkittäminen ... 60
6.3.4 Ohjelman ajaminen ... 60
6.3.5 Varoitus ... 62
6.3.6 Integroitu ympäristö ... 62
6.4 Ohjelman yksityiskohtainen tarkastelu ... 62
6.4.1 Kommentointi ... 62
6.4.2 Miten kommentoida ... 63
6.4.3 JavaDoc ... 64
6.4.4 Valmiin kommenttilohkon lukeminen ... 64
6.4.5 Tarvittavien luokkien esittely ... 64
6.4.6 Luokan esittely ... 65
6.4.7 Pääohjelman esittely ... 65
6.4.8 Lausesulut ... 67
6.4.9 Tulostuslause ... 67
6.4.10 Lauseen loppumerkki ; ... 67
6.4.11 Isot ja pienet kirjaimet ... 67
6.4.12 White spaces, tyhjä ... 67
6.4.13 Vakiomerkkijonot ... 68
6.4.14 Vakiolukuarvot ... 69
7. Java–kielen muuttujista ja aliohjelmista ... 71
7.1 Mittakaavaohjelman suunnittelu ... 71
7.2 Muuttujat ... 72
7.2.1 Matkan laskeminen ... 73
7.2.2 Muuttujan nimeäminen ... 75
7.2.3 Muuttujalle sijoittaminen = ... 76
7.2.4 Muuttujan esittely ja alkuarvon sijoittaminen ... 76
7.3 Muuttujan arvon lukeminen päätteeltä ... 77
7.3.1 Lukeminen merkkijonoon ... 77
7.3.2 Lukuarvon selvittäminen merkkijonosta ... 78
7.3.3 Apumetodit ... 79
7.3.4 Apuluokat ... 80
7.3.5 Luokan testaaminen ... 80
7.3.6 Luokan käyttäminen ... 81
7.4 Viitteet ... 81
7.4.1 Miksi viitteet? ... 81
7.4.2 Lokaalit muuttujat ... 82
7.4.3 Dynaaminen muisti ... 83
7.4.4 Viitteiden vertaaminen ... 83
7.4.5 Viitteeseen sijoittaminen ... 84
7.4.6 null-viite ... 84
7.5 Aliohjelmat (metodit, funktiot) ... 85
7.5.1 Parametriton aliohjelma ... 85
7.5.2 Funktiot ja parametrit ... 86
7.5.3 Parametrin nimi kutsussa ja esittelyssä ... 88
7.5.4 Nimessään arvon palauttavat funktiot ... 89
7.5.5 Ketjutettu kutsu ... 90
7.5.6 Yksinkertaisen aliohjelman kutsuminen ... 91

7.5.7 Aliohjelmat tulostavat harvoin 92
7.5.8 Jäsenten tulostaminen 93
7.6 Parametrinvälitys 95
7.6.1 Useita parametreja 95
7.6.2 Parametrin paikka ratkaisee, ei nimi 96
7.6.3 Metodin nimen kuormittaminen 97
7.6.4 Muuttujien lokaalisuus 97
7.6.5 Parametrinvälitysmekanismi 98
7.6.6 Aliohjelmien kirjoittaminen 100
7.6.7 Luokkamuuttujat ja suhde lokaaleihin muuttujiin 100
7.7 Testipääohjelmat 103
7.8 Yksikkötestaus 104
7.9 JUnit 104
7.10 ComTest 105
7.10.1 ComTestin makrokieli 105
7.10.2 ComTestin edistyneemmät ominaisuudet 106
7.11 Mittakaavaohjelma graafisena 108
8. Kohti olio-ohjelmointia 111
8.1 Miksi olioita tarvitaan 112
8.2 Hynttyyt yhteen, eli muututaan olioksi 113
8.2.1 Terminologiaa 113
8.2.2 Ensimmäinen olio-esimerkki 113
8.2.3 toString() 115
8.2.4 Luokka (*class*) ja olio (*object*) 115
8.2.5 Suojaustasot ja kapselointi 116
8.2.6 Muodostajat (*constructor*) 117
8.2.7 Oletusmuodostaja (*default constructor*) 117
8.2.8 Sisäinen tilan valvonta 118
8.2.9 Metodien kuormittaminen (lisämäärittely, *overloading*) 121
8.2.10 `this`-osoitin 121
8.3 Perintä 123
8.3.1 Luokan ominaisuuksien laajentaminen 123
8.3.2 Alkuperäisen luokan muuttaminen 123
8.3.3 Saantimetodit 124
8.3.4 Koostaminen 124
8.3.5 Perintä, inheritance 126
8.3.6 Polymorfismi, eli monimuotoisuus 128
8.3.7 Myöhäinen sidonta 128
8.4 Kapselointi 129
8.4.1 Rajapinta ja sisäinen esitys 129
8.5 Rajapinta ja monimuotoisuus 131
8.6 `Object`-luokan metodien korvaaminen 134
8.7 Mistä hyviä luokkia 136
8.8 Valmiita luokkia 137
8.8.1 Merkkijonoluokat 137
9. Java–kielen ohjausrakenteista ja operaattoreista 141
9.1 if–lause 142
9.1.1 Ehdolla suoritettava yksi lause 142
9.1.2 Ehdolla suoritettava useita lauseita 142
9.2 Loogiset lausekkeet 143

9.2.1 Vertailuoperaattorit 144
9.2.2 Sijoitus palauttaa arvon! 144
9.3 Loogisten lausekkeiden yhdistäminen 145
9.3.1 Loogiset operaattorit &&, || ja ! 145
9.3.2 Loogisen lausekkeen suoritusjärjestys 146
9.3.3 Loogiset operaattorit & ja | 147
9.4 Bittitason operaattorit 147
9.5 `if – else` –rakenne 148
9.5.1 Sisäkkäiset `if`–lauseet 149
9.5.2 Useat peräkkäiset ehdot 153
9.6 `do-while` –silmukka 154
9.7 `while` –silmukka 155
9.8 `for` –silmukka, tavallisin muoto 156
9.9 `for-each` silmukka 156
9.10 Java–kielen lauseista 157
9.10.1 Sijoitusoperaattori = ... 157
9.10.2 Sijoitus- ja kasvatusoperaattori += 157
9.10.3 Lisäysoperaattori ++ 158
9.11 for –silmukka, yleinen muoto 159
9.12 `break` ja `continue` 160
9.12.1 `break` 160
9.12.2 `continue` 161
9.13 `switch` –valintalause 162
9.13.1 | ei toimi `switch` –lauseessa! 164
9.14 Ikuinen silmukka 164
9.14.1 Yhteenveto silmukoista 165
10. Oliosuunnittelu 167
10.1 Olio on luokan esiintymä 167
10.2 Tavoitteet 167
10.3 Luokat 167
10.4 CRC–kortit 167
10.4.1 Jäsen-luokka (`Jasen`) 168
10.4.2 Kerho-luokka, yksinkertainen (`Kerho`) 168
10.4.3 Käyttöliittymä-luokka (`Naytto`) 168
10.4.4 Luokkajaon tarkastelua 169
10.5 Harrastukset mukaan kortteihin 169
10.5.1 "Oliomalli" 169
10.5.2 Relaatiomalli kortteihin 169
10.5.3 Harrastus-luokka (`Harrastus`) 170
10.5.4 Kerho-luokka (`Kerho`) 170
10.5.5 Jäsenet-luokka (`Jasenet`) 170
10.5.6 Harrastukset-luokka (`Harrastukset`) 170
11. Jäsenrekisterin runko 171
11.1 Runko ilman kommentteja 171
11.2 Valittava tietorakenne 171
11.2.1 Taulukko 172
11.2.2 Linkitetty lista 172
11.2.3 Sekarakenne 173
11.3 Harrastukset mukaan 174

11.3.1 Tiedot jäsenen yhteyteen 174
11.3.2 Relaatiomalli tietorakenteeseen 175
12. Java–kielen taulukoista 177
12.1 Yksiulotteiset taulukot 177
12.1.1 Taulukon määrittely 177
12.1.2 Taulukon alkioihin viittaaminen indeksillä 178
12.1.3 Taulukon läpikäyminen `for` ja `for-each`–silmukoilla 178
12.1.4 Taulukon alustaminen 178
12.2 Merkkijonot 179
12.2.1 Merkkityyppi 179
12.2.2 String 179
12.2.3 StringBuilder ja StringBuffrer 179
12.3 Moniulotteiset taulukot Javassa 179
12.3.1 Kiinteä esittely 179
12.3.2 Yksiulotteisen taulukon käyttäminen moniulotteisena 180
12.3.3 Taulukko taulukoista 180
12.3.4 Taulukko viitteistä 181
12.4 Komentorivin parametrit (`argv`) 182
13. Dynaaminen muistinkäyttö 185
13.1 Muistin käyttö 185
13.2 Dynaamisen muistin käyttö Javassa 186
13.2.1 `new` 186
13.2.2 Olion tuhoaminen 186
13.2.3 Taulukon luominen `new []` 187
13.3 Dynaamiset taulukot 187
13.3.1 Poikkeukset 188
13.3.2 Poikkeukset ja *ComTest* 189
13.4 Geneerinen taulukko 190
13.5 Javan *Collections Framework* 191
13.5.1 ArrayList ja geneerisyys 192
13.5.2 Iteraattori 193
13.5.3 Geneerisyys 193
13.5.4 Algoritmit 193
13.6 Tietovirta parametrina 197
14. Funktio-olio 199
14.1 Numeerinen integrointi 199
14.1.1 Sinifunktion integrointi 199
14.2 Numeerinen integrointi ja rajapinta 201
14.3 Rajapinnan käyttäminen 202
15. Tiedostot 203
15.1 Tiedostojen käsittely 203
15.1.1 Lukeminen 204
15.2 Tiedostojen käsittely Javan tietovirroilla 204
15.2.1 Tiedoston avaaminen muodostajassa 205
15.2.2 Tiedostosta lukeminen. 206
15.2.3 Tiedoston lopun testaaminen 206
15.2.4 Tiedostoon kirjoittaminen 206
15.2.5 Tiedoston sulkeminen `close` 207
15.3 Tiedoston yhdellä rivillä monta kenttää 208
15.3.1 Ongelma 208

15.4 Merkkijonon paloittelu 208
15.4.1 `parse` 209
15.4.2 `erota` 209
15.4.3 Esimerkki `erota`-funktion käytöstä 210
15.4.4 Erota funktion toiminta vaihe vaiheelta 210
15.4.5 Luvun erottaminen 211
15.5 Lukeminen ja paloittelu 212
15.5.1 Olio joka lukee itsensä 213
15.6 Esimerkki tiedoston lukemisesta 214
15.7 Kerhon talletukset 217
16. Kerho-ohjelman rakenne 219
16.1 Jäsen ja kentät 219
16.1.1 Algoritmi näytön ja jäsenen keskustelulle 219
16.2 Näytön ja jäsenen välinen rajapinta 219
16.3 Kerhon rakenne 220
16.3.1 KerhoSwingin ja käyttöliittymän yhteistoiminta 221
16.4 Jäsenien selaaminen 221
16.5 Jäsenkenttien lisääminen 222
16.6 Oikeellisuustarkistukset 226
16.6.1 Säännölliset lausekkeet 226
16.6.2 Säännöllisten lausekkeiden käyttäminen 227
16.7 Kentät graafiseen käyttöliittymään 228
16.8 Etsiminen 230
16.9 Lajittelu avaimen kentän mukaan 230
16.10 Tulostaminen 231
Hakemisto 233

Tehtävät:

Tehtävä 2.1 Ketkä harrastavat? 20
Tehtävä 2.2 Mikä on tilaa säästävin tallennusmuoto 21
Tehtävä 3.1 Kävelyohjeet 28
Tehtävä 3.2 Muita lajittelualgoritmeja 30
Tehtävä 3.3 Algoritmin kompleksisuus 30
Tehtävä 3.4 Lajittelujärjestys 30
Tehtävä 3.5 Kuplalajittelu 30
Tehtävä 3.6 Loppuminen erikoistapauksessa 31
Tehtävä 3.7 QuickSortin kompleksisuus 31
Tehtävä 3.8 Lisäys oikealle paikalleen vaiko lisäys loppuun ja lajittelu? 31
Tehtävä 3.9 Puolitushaku 33
Tehtävä 3.10 Puolitushaun kompleksisuus 33
Tehtävä 3.11 Kumin paikkaus 33
Tehtävä 3.12 Sunnuntai–ilta 33
Tehtävä 3.13 Onkiminen 33
Tehtävä 3.14 Järjestyksen kääntäminen päinvastaiseksi 33
Tehtävä 4.1 Ajanlisäys 36
Tehtävä 4.2 Postimaksu 36
Tehtävä 4.3 Korvaaminen ehtolauseilla 36

Tehtävä 4.4 Uiminen ... 37
Tehtävä 4.5 Ynnää luvut 1–100 ... 37
Tehtävä 4.6 Vuokaavio ... 38
Tehtävä 4.7 Algoritmin parantaminen ... 39
Tehtävä 4.8 Pöytätesti ... 39
Tehtävä 4.9 Lajittelun testaus ... 40
Tehtävä 4.10 Korttien poisto ... 40
Tehtävä 4.11 Korttien poisto osoittimia käyttäen ... 41
Tehtävä 4.12 Kaksiulotteisen taulukon indeksit ... 42
Tehtävä 4.13 Sijoitus 3–ulotteiseen taulukkoon ... 43
Tehtävä 4.14 3–ulotteinen taulukko 1–ulotteiseksi ... 43
Tehtävä 4.15 Kolmiulotteinen taulukko ... 44
Tehtävä 4.16 Neliulotteinen taulukko ... 44
Tehtävä 4.17 Sanojen muuttaminen ... 45
Tehtävä 4.18 Lihapullan paistaminen ... 46
Tehtävä 4.19 Kombinaatioiden lukumäärä ... 47
Tehtävä 4.20 BAL=kyllä? ... 48
Tehtävä 4.21 Kuka valehtelee? ... 48
Tehtävä 4.22 de Morganin kaava ... 49
Tehtävä 4.23 Osittelulaki ... 49
Tehtävä 4.24 Ehtojen sieventäminen ... 50
Tehtävä 4.25 Merkkijonot ... 50
Tehtävä 4.26 Päivämäärät ... 50
Tehtävä 6.1 Nimi ja osoite ... 59
Tehtävä 6.2 Terve maailma! ... 69
Tehtävä 6.3 Nimi ja osoite vakioksi ... 69
Tehtävä 6.4 Tetraedri ... 70
Tehtävä 7.1 Vakion korvaaminen ... 75
Tehtävä 7.2 Avainsanat ... 75
Tehtävä 7.3 Muuttujan nimeäminen ... 75
Tehtävä 7.4 Muuttujien esittely ... 76
Tehtävä 7.5 Oletuksen tulostaminen ... 79
Tehtävä 7.6 Muiden tyyppien lukeminen ... 81
Tehtävä 7.7 Mittakaavan kysyminen ... 81
Tehtävä 7.8 Funktio ja osoitin ... 90
Tehtävä 7.9 String vs. StringBuffer ... 90
Tehtävä 7.10 Math-luokka ... 91
Tehtävä 7.11 Funktiot ... 91
Tehtävä 7.12 Ympyrän ala ja pallon tilavuus ... 91
Tehtävä 7.13 Pääohjelma yhtenä funktiokutsuna ... 91
Tehtävä 7.14 Päämenuun kerhon nimi ... 96
Tehtävä 7.15 Toisen asteen yhtälön juuri ... 96
Tehtävä 7.16 Toisen asteen polynomi, `root_1` ... 96
Tehtävä 7.17 root_1 testaus ... 96
Tehtävä 7.18 Toisiaan kutsuvat aliohjelmat ... 97
Tehtävä 7.19 Eri nimet ... 98
Tehtävä 7.20 Muotoilu? ... 100
Tehtävä 7.21 Tiedon lukeminen ... 100
Tehtävä 7.22 Muuttujien näkyvyys ... 102
Tehtävä 8.1 Tulostus ... 113

Tehtävä 8.2 Negatiivinen minuuttiasetus ... 120
Tehtävä 8.3 Päivämääräluokka ... 120
Tehtävä 8.4 Päivämääräluokan toteutus ... 121
Tehtävä 8.5 Mitäs me tehtiin kun ei ollut kuormitusta? ... 121
Tehtävä 8.6 Lisäys yhdellä ... 121
Tehtävä 8.7 Vain tuntien asettaminen ... 121
Tehtävä 8.8 Luokan muuttaminen ... 123
Tehtävä 8.9 Sekuntien tulostus aina tai oletuksena ... 123
Tehtävä 8.10 Miksi vielä yksi lisaa-kutsu? ... 129
Tehtävä 8.11 Ei turhaa lisaa-kutsua ... 129
Tehtävä 8.12 Saantimetodi sekunneille ... 129
Tehtävä 8.13 Saantimetodien käyttäminen ... 129
Tehtävä 8.14 `minuutteina()` ... 131
Tehtävä 8.15 `equals` `toString` avulla ... 136
Tehtävä 8.16 `equals` `AikaSek`-luokkaan ... 136
Tehtävä 8.17 `AikaSek` perimällä. ... 136
Tehtävä 8.18 Vertailu ... 136
Tehtävä 8.19 Ensimmäinen melkein järkevä olio ... 138
Tehtävä 9.1 vaihda ... 143
Tehtävä 9.2 abs ... 143
Tehtävä 9.3 jarjesta2 ... 143
Tehtävä 9.4 maksimi ja minimi ... 143
Tehtävä 9.5 Loogiset/bittitason operaattorit ... 148
Tehtävä 9.6 Luku parilliseksi ... 148
Tehtävä 9.7 += ... 157
Tehtävä 9.8 1+2+..+i ... 159
Tehtävä 9.9 Tarvitaanko sisäkkäisiä silmukoita? ... 161
Tehtävä 9.10 continuen korvaaminen ... 161
Tehtävä 9.11 Eri silmukoiden vertailu ... 162
Tehtävä 9.12 switch –> if ... 164
Tehtävä 9.13 Päävalinta ... 164
Tehtävä 9.14 lääni, versio 2 ... 164
Tehtävä 11.1 Lisäys ... 174
Tehtävä 11.2 Etsiminen ... 174
Tehtävä 11.3 Poisto ... 174
Tehtävä 11.4 Lajittelu ... 174
Tehtävä 11.5 Lisää harrastus ... 175
Tehtävä 11.6 Mitä harrastaa ... 175
Tehtävä 11.7 Kuka harrastaa ... 175
Tehtävä 11.8 Poista harrastus ... 175
Tehtävä 11.9 Jäsenen poistaminen ... 176
Tehtävä 11.10 "Roskaharrastukset" ... 176
Tehtävä 11.11 Monta saman lajin harrastajaa ... 176
Tehtävä 12.1 Taulukon alkioiden nollaus ... 179
Tehtävä 12.2 Matriisit ... 180
Tehtävä 12.3 Matriisi 1–ulotteisena ... 180
Tehtävä 12.4 Transpoosi ... 182
Tehtävä 12.5 Palindromi ... 183
Tehtävä 15.1 Tiedoston lukujen summa ... 205
Tehtävä 15.2 Kommentit näytölle ... 208

Tehtävä 15.3 Ohjelman "sekoaminen" ... 208
Tehtävä 15.4 Tietorakenne ... 216
Tehtävä 15.5 Perintä ... 217
Tehtävä 15.6 Tunnistenumero ... 217
Tehtävä 15.7 Graafinen mittakaava ... 217

Kuvat:

Kuva 4.1 Ehtolauseet ... 36
Kuva 4.2 `switch`–valintalause ... 36
Kuva 4.3 `do`–silmukka ja `do-while`–silmukka ... 37
Kuva 6.1 Ohjelman kääntäminen ja linkittäminen ... 61
Kuva 6.2 Java-ohjelman kääntäminen ja linkittäminen ... 61
Kuva 7.1 Olioviitteet ... 83
Kuva 7.2 Kaksi viitettä samaan olioon ... 84
Kuva 8.1 Suojaustasot ... 116
Kuva 8.2 Aika perinnällä ... 128
Kuva 8.3 Musta laatikko ... 129
Kuva 11.1 Taulukko C++ -kielessä ... 172
Kuva 11.2 Javan taulukko ... 172
Kuva 11.3 Linkitetty lista ... 173
Kuva 11.4 Tietorakenne kun kerho tallettaa jäsenet ... 174
Kuva 11.5 Harrastukset linkitettynä listana ... 175
Kuva 11.6 Harrastukset relaation avulla ... 176
Kuva 13.1 Javan tärkeimmät tietorakennerajapinnat ... 192

Malliohjelmat:

nimet.dat - ensimmäinen ehdotus tiedostoksi ... 13
nimet.dat - sarakkeet linjaan ... 13
nimet.dat – ei näin ... 17
nimet.dat - harrasteet samalle riville ... 18
nimet.dat - harrasteet omalle riville ... 19
nimet.dat - relaatiokannan päätaulu ... 19
harrastukset.dat - harrasteet relaation avulla ... 19
kelmit.xml – kerho XML-muodossa ... 20
alkeet.hello.Hello.java - ensimmäinen Java ohjelma ... 58
C#-versio 58
alkeet.hello.Hello2.java - malliohjelma ... 62
alkeet.hello.Hello3.java - tervehdys parametrina ... 66
C#-versio tervehdyksestä parametrina ... 66
alkeet.hello.Hello7.java - tervehdys vakioksi ... 69
alkeet.kuutio.Kuutio.java - monikulmion tiedot vakioksi ... 69
C#-versio Kuutio-ohjelmasta ... 70
muuttujat.matka.MatkaScan.java - mittakaavamuunnos 1:200000 kartalta ... 74
muuttujat.matka.Matka.java - mittakaavamuunnos 1:200000 kartalta ... 74
muuttujat.syotto.Syotto.java – kokonaisluvun lukeminen päätteeltä ... 80
muuttujat.jono.Jonotesti.java - merkkijonoviitteet ... 82
muuttujat.matka.Matka_a1.java - ohjeet aliohjelmaksi ... 85
muuttujat.matka.Matka_a3.java - erilaisia funktioita /** ... 86
muuttujat.matka.Matka_a4.java - erilaisia tapoja kutsua funktiota ... 88

muuttujat.funktio.FunJaOlio.java - sivuvaikutuksellinen funktio 89
muuttujat.tulostus.Tulostustesti.java - tulostus näytölle ja tiedostoon 93
HT3 KerhoGUI.java – tulostamisikkunan avaaminen 93
HT3 KerhoSwing.java – viedään tulostusikkunalle 94
muuttujat.funktio.Parampaikka.java - parametrin paikka kutsussa ratkaisee 96
muuttujat.nakyvyys.Lokaali.java - lokaalien muuttujien näkyvyys 98
muuttujat.funktio.Aikalisa.java - yritys lisätä arvoja 99
muuttujat.funktio.Alisotku.java - parametrin välitystä 100
muuttujat.funktio.Alisotk2.java - parametrin välitystä 102
muuttujat.testaus.Alkuluku.java 103
muuttujat.testaus.Esiintymat.java 106
muuttujat.graafinen.Mittakaava.java 109
oliot.muut.Aikalis4.java - useita aika "muuttujia" 112
oliot.aika.olio.Aika.java - kunnon olioksi 113
oliot.aika.olio.Aikatesti.java - testiluokka Aika-luokalle 116
oliot.aika.olio.Aika.java - muodostaja alustamaan tiedot 117
oliot.aika.muodostaja.Aika.java - lisätään oletusmuodostaja 118
oliot.aika.valvonta.Aika.java - sisäinen tilan valvonta asetuksessa 120
oliot.aika.metodi.Aika.java - aliohjelma vastaan metodi 121
oliot.aika.koostaminen.AikaSek.java - laajentaminen koostamalla 124
oliot.aika.perinta.AikaSek.java - laajentaminen perimällä 126
oliot.aika.sisesitys.Aika.java - sisäinen toteutus minuutteina 129
oliot.aika.perinta.AikaSek.java - esimerkki polymorfisesta taulukosta 131
oliot.aika.koostaminen.AikaSek.java - kömpelö esimerkki polymorfisesta taulukosta 131
oliot.aika.rajapinta.AikaRajapinta.java - malli kaikkien Aika-luokkien rajapinnasta 132
oliot.aika.rajapinta.Aika.java - luokka joka toteuttaa rajapinnan 132
oliot.aika.rajapinta.AikaSek.java - luokka joka toteuttaa rajapinnan => polymorfismi 133
oliot.aika.object.Aika.java - luokka joka toteuttaa Object 134
oliot.henkilo.Henkilo.java - 1. järkevä olio 138
oliot.henkilo.Opiskelija.java - 1. järkevä olio 139
ohjausrak.Ifsij2.java - esimerkki tahallisesta sijoituksesta ehdossa 144
ohjausrak.polynomi.v1.Polynomi2.java - esimerkki 2. asteen yhtälön ratkaisemiseta 149
ohjausrak.polynomi.v2.Polynomi2.java - karsittu versio 2. asteen yhtälöstä 151
ohjausrak.polynomi.v3.Polynomi2.java - normaalit tapaukset ensin ratkaisussa 152
ohjausrak.polynomi.v4.Polynomi2.java - else -osat pois 152
ohjausrak.Postimaksu.java - esimerkki samanarvoisista ehtolauseista 153
alkeet.alkuluku.Alkuluku.java - testataan onko luku alkuluku 154
ohjausrak.Dowhile.java - lukujen lukeminen kunnes halutulla välillä 155
ohjausrak.alkuluku2.Alkuluku2.java - alkulukutesti while-silmukalla 155
ohjausrak.vali.Valinsum.java - esimerkki for-silmukasta 156
ohjausrak.Plusplus.java - esimerkki ei-yksikäsitteisestä ++ operaattorin käytöstä 158
ohjausrak.Valinsum.java - useita alustuslauseita for-silmukassa 159
ohjausrak.Valinsum.java - C:mäinen silmukka 159
ohjausrak.Break.java - silmukan katkaisu keskeltä 160
ohjausrak.Break.java - ulomman silmukan katkaisu keskeltä 160
ohjausrak.Continue.java - silmukan lopun ohittaminen 161
ohjausrak.SwingAanestys.java – esimerkki switch-lauseesta 162
ohjausrak.Switch.java - switch, jossa break tahallaan jätetty pois 163
taulukot.Mat2.c - matriisi parametrina riviosoittimen avulla 181
taulukot.Mat3.java - matriisi osoitintaulukon avulla 182

taulukot.Argv.java - komentorivin parametrit 182
dynaaminen.Taulukko.java -esimerkki dynaamisesta taulukosta 187
dynaaminen.TaulukkoGen.java -esimerkki dynaamisesta taulukosta 190
dynaaminen.ArrayListMalliGen.java –ArrayList-luokka 192
dynaaminen. AlgoritmiMalliGen.java – Collections-luokka 194
funktio.integroi.Integroi.java – Sinifunktion numeerinen integrointi 200
funktio.integroi.Integroi2.java – funktio-oliot, rajapinta 201
funktio.integroi.Integroi2.java – funktio-oliot, funktio 201
funktio.integroi.Integroi2.java – funktio-oliot, integrointi 201
funktio.integroi.Integroi2.java – Toisen asteen polynomi 202
tiedosto.TiedKaScanner.java - Lukujen lukeminen tiedostosta 204
tiedosto.Kertotaulu.java - Tiedostoon tulostaminen 206
tuotteet.dat - esimerkkitiedosto 208
tiedosto.ErotaEsim.java - esimerkki erota-funktion käytöstä 210
tiedosto.LueTuote.java - esimerkki tiedoston lukemisesta 212
tiedosto.LueRek.java - esimerkki oliosta joka käsittelee tiedostoa 213
tiedosto.LueTRek.java - esimerkki tiedoston lukemisesta 214
KerhoSwing.java – Harjoitustyö vaihe 7 – get ja set-metodit 221
Jasen.java – Harjoitustyö vaihe 7 222
Kentta.java – Harjoitustyö vaihe 7 223
PerusKentta.java – Harjoitustyö vaihe 7 223
JonoKentta.java – Harjoitustyö vaihe 7 225
Tarkistaja.java – Harjoitustyö vaihe 7 226
PuhelinKentta.java – Harjoitustyö vaihe 7 227
KerhoSwing.java – Harjoitustyö vaihe 7 – luoNaytto()-metodi 228
KerhoSwing.java – Harjoitustyö vaihe 7 – Jäsenen muuttaminen 230

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. Hyvä kotimainen vaihtoehto on esimerkiksi: **Jorma Kyppö, Mika Vesterholm:** *Java-ohjelmointi*, 2008, Talentum Oyj. Myös ohjelmointiympäristön mukana olevasta *OnLine*–avustuksesta (*Help*) saa tarvittavaa lisätietoa.

Monisteen esimerkkiohjelmat löytyvät elektronisessa muodossa:

Mikroluokka:	hakemisto:	n:\kurssit\ohj2\moniste\esim
WWW:	URL:	https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/

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, ovat 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 no-

peammin. 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

Monisteen vuoden 2012 versioon on lisätty graafisen käyttöliittymän tekeminen Swing-kehyksellä sekä korjattua lukuisia pieniä yksityiskohtia. Kiitoksia Santtu Viitaselle tehdystä työstä.

Palokassa 25.12.2011

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, 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

Kannattaa huomata että listalla varsinaisesti tietokoneella tehtävä työ on listan viimeisissä kohdissa. Pitkään ohjelmistojen suunnittelu ja toteutus seurasivatkin orjallisesti vaihe vaiheelta yllä olevan kaltaista listaa. Tämä vesiputousmalliksi kutsuttu toteutustapa oli ennen ohjelmistokehityksen kulmakivi, jolla toteutettiin käytännössä kaikki ohjelmointiprojektit. Maailmalla ilmestyi kuitenkin tutkimuksia joiden mukaan suurin osa

ohjelmistoprojekteista itse asiassa epäonnistui, mikä tietysti on hälyttävää millä tahansa alalla.

1.1 Ketterät menetelmät

Nykyään perinteisen vesiputousmallin rinnalle on noussut niin sanottu ketterä ohjelmistokehitys (*Agile Software Development*). Se on joukko menetelmiä joiden leimaavin piirre on ohjelmien kehittäminen pienissä pätkissä, joissa jokaisessa toteutetaan kaikki ohjelmistokehityksen vaiheet. On jopa mahdollista että yksi ihminen työstää useaa vaihetta kerralla! Ketterillä menetelmillä on oma julistus, *Agile Manifesto*, jonka arvoja ne pyrkivät noudattamaan.

- **Yksilöt ja vuorovaikutus** yli prosessien ja työkalujen
- **Toimiva ohjelma** yli kokonaisvaltaisen dokumentaation
- **Asiakasyhteistyö** yli sopimusneuvottelujen
- **Muutoksiin vastaaminen** yli suunnitelman seuraamisen

Tavoitteita tulkittaessa täytyy kuitenkin muistaa, että vaikka menetelmät pitävätkin lihavoituja asioita arvokkaampana, niin ne eivät tee silti muista merkityksettömiä.

Vaikka tavoitteet ovatkin yhteneväiset, niin menetelmien väliset erot ovat usein suuria. Jotkut saattavat painottuvat projektinhallintaan, kun taas joku tarjoaa käytännön ohjeita ohjelmoijan työskentelytapoihin.

1.2 Extreme Programming

Tämän kurssilla opetuksessa ja varsinkin harjoitustyön toteuttamisessa pyritään mahdollisuuksien mukaan soveltamaan ja lainaamaan paljon niin sanotulta *Extreme Programming* (*XP*) menetelmältä. Tietenkin menetelmä on kehitetty työelämän tarpeisiin, eikä sen soveltaminen sellaisenaan opetuskäyttöön ole mahdollista.

1.2.1 Kurssilla sovellettavia XP:n käytäntöjä

- Iteraatiot
- Aiemmista kokemuksista oppiminen
- Testilähtöinen ohjelmointi
- Pariohjelmointi
- Uudelleenrakentaminen
- Yhteisomistajuus
- Jatkuva integrointi
- Et tule tarvitsemaan sitä (yksinkertainen rakenne)
- Hallinnon (opettajat ja ohjaajat) taustatuki
- Tasainen työtahti
- Julkaisujen suunnittelu
- Hyväksyntätestaus
- Lyhyin väliajoin tuotettavat julkaisut (pienet julkaisut)

1.3 Ohjelman suunnittelu

Aluksi kurssi keskittyy ohjelmoinnin perusteiden, kuten algoritmien ja oman ohjelman suunnitteluun. 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.

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 kyse 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.

1.4 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.

1.5 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.

1.6 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. Tavan heikkous piilee kuitenkin siinä, että mikäli haluamme muuttaa nyt alkuperäistä koodia, meidän on mahdoton tietää vaikuttaako muutos johonkin toiseen ohjelmiston osa-alueeseen joka käyttää koodia hyväkseen.

Nykyisin ratkaisuksi on kehitetty testausta automatisoivia työkaluja, kuten Javan käyttämä *JUnit*. Yksikkötestauksen idea on kirjoittaa jokaisen ohjelmiston osaan testikoodi, mikä voidaan ajaa keskitetysti vaikka koko ohjelmistolle kerralla.

Eräs yksikkötestausta hyödyntävä tekniikka on testivetoinen kehitys (*TDD, test-driven development*). Sen tarkoituksena on kirjoittaa koodi testattavaksi ja testit ennen varsinaisen ohjelmakoodin kirjoittamista. Tämän ehkä aluksi nurinkuriselta tuntuvalla ajatuksella on kuitenkin useita hyötyjä. Kyse ei ole niinkään testaustyökalusta, vaan ohjelman suunnittelusta, josta syntyykin sivutuotteena valmiit testitapaukset.

Tällä kurssilla testaamiseen voi käyttää myös Jyväskylän yliopistossa kehitettyä *ComTest* työkalua. Työkalu helpottaa *JUnit* testien tekemistä ja sen avulla pystyy samalla luomaan myös kattavan *JavaDoc* dokumentaation.

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

Ennen virheiden löytämiseksi testiohjelmiin lisättiin tulostuslauseita. Nykyisin tehokkaat debuggerit helpottavat testausta huomattavasti: ohjelman toimintaa voidaan seurata askel kerrallaan ja epäilyttävien muuttujien arvoja voidaan tarkistaa kesken suorituksen. On myös mahdollista laittaa ohjelma pysähtymään jonkin muuttujan saadessa virheellisen arvon.

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

1.7 Hyväksymistestaus ja palaute

Ketterien menetelmien tärkeimpiä osa-alueita on jatkuva vuorovaikutus asiakkaan kanssa. Aluksi tällä kurssilla asiakkaana toimii oppilas itse suunnitellessaan harjoitustyöohjelman toiminnot ja käyttötarkoituksen. Harjoitustyötä tehdään pienissä vaiheissa, eli *iteraatioissa*, joiden tarkoitus on pitää ohjelma jatkuvasti toimivana kokonaisuutena, mihin on helppo lisätä uusia ominaisuuksia yksi kerrallaan. Vaiheen päätettyä tehdään hyväksymistestaus, jossa työ esitellään asiakkaalle (ohjaajalle), jolta saa vinkkejä ja palautetta ohjelman toiminnan parantamiseksi.

Jokaisessa vaiheessa toteutetaan jokin ohjelman osa-alue tai parannellaan vanhaa. Tämä työ sisältää suunnittelun, testauksen, koodauksen ja dokumentoinnin.

1.8 Käyttöönotto

Ketteriä menetelmiä käyttämällä ohjelman käyttöönottovaiheessa sen pitäisi olla testattu ja valmis. Tietysti julkaisuversioonkin pääsee lähes aina livahtamaan joitakin *bugeja*, mutta niiltä ei taitavinkaan ohjelmoija voi välttyä. Asiakas on lisäksi pidetty mukana koko prosessin ajan, eikä ikäviä yllätyksiä – joissa ohjelma ei olekaan toiminnallisuudeltaan sitä mitä on odotettu – pääse syntymään.

1.9 Ylläpito

Jos kuitenkin ohjelmasta paljastuu virheitä tai puuttuvia toimintoja. Virheet pitää korjata ja puuttuvat toiminnot mahdollisesti lisätä, jolloin ollaan jälleen ohjelmansuunnittelun alkuvaiheessa. Hyvin suunniteltuun ohjelmaan saattaa olla helppo lisätä uusia toimintoja ja vastaavasti huonosti suunnitellussa saattavat jopa tietorakenteet mennä uusiksi. Tosin tätäkään ei pidä pelätä, sillä yksinkertaisesti aina ei ole mahdollista ottaa etukäteen kaikkea huomioon.

Myös ohjelman alkuperäiset kirjoittajat ovat saattaneet häipyä ja mikäli kehitysprosessiin ei ole kiinnitetty tarpeellista huomiota, niin joku onneton kesätyöntekijä joutuu ensitöikseen paikkaamaan toisten huonosti dokumentoimaa sotkua.

1.10 Yhteenveto, kielellä ei väliä

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, C#, 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. TEX, 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
- hetu
- 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, vaihtoehtoja

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 joi-

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

nimet.dat - ensimmäinen ehdotus tiedostoksi

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
;sukunimi etunimi|hetu|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|||
```

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

nimet.dat - sarakkeet linjaan

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
;sukunimi etunimi   |hetu       |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, mutta sillä ei tietenkään nykyään ole mitään merkitystä. Lisäksi yhden henkilön tiedot eivät mahdu kerralla näyttöön. Onneksi kuitenkin lähes kaikki 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 ja käyttöliittymä

Jatkosuunnittelu on ehkä helpointa tehdä suunnittelemalla ohjelman toimintaa käyttöohjeen tai käyttöliittymän tavoin.

Vaikka nykyaikaisilla ohjelmointiympäristöillä käyttöliittymän piirtäminen on tottuneelle käyttäjälle todella nopeaa, niin ensikertalaisen on kuitenkin helpompaa nopeampaa toteuttaa alustava suunnittelu perinteisesti esimerkiksi kynällä ja paperilla.

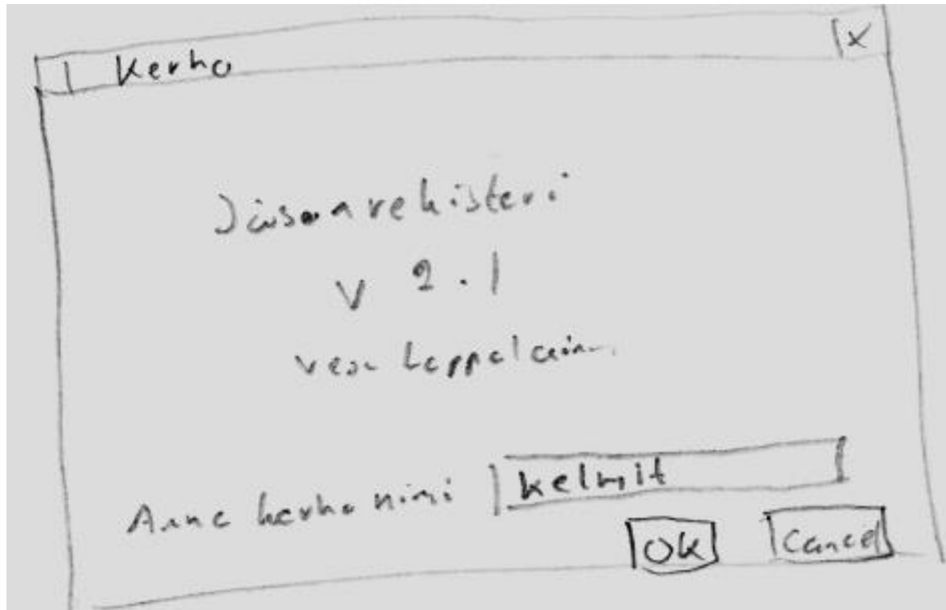
Suunnittelussa toimitaan käyttäjän ja helppokäyttöisyyden (= myös nopea käyttö, ei aina välttämättä hiiri) ehdoilla. On myös huomioitava ohjelmoitava alusta ja siinä vakiintuneet tavat toteuttaa toimintoja.

2.4.1 Ohjelman käynnistys

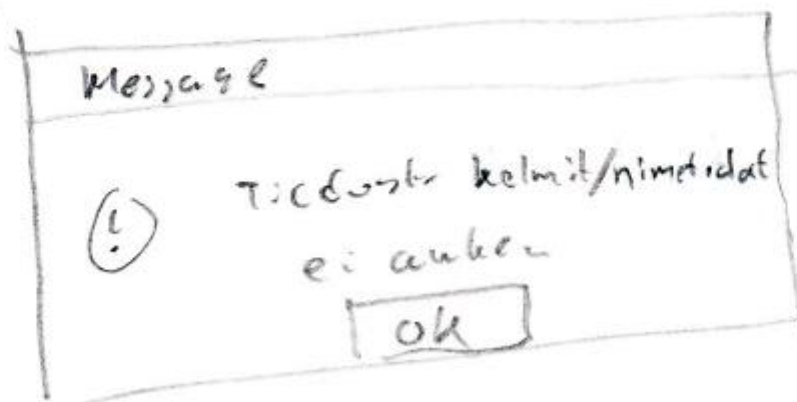
Ohjelma käynnistetään klikkaamalla kerho.jar-ikonia tai antamalla komentoriviltä komento

```
java –jar kerho.jar
```

Kun ohjelma käynnistyy, tulostuu näyttöön

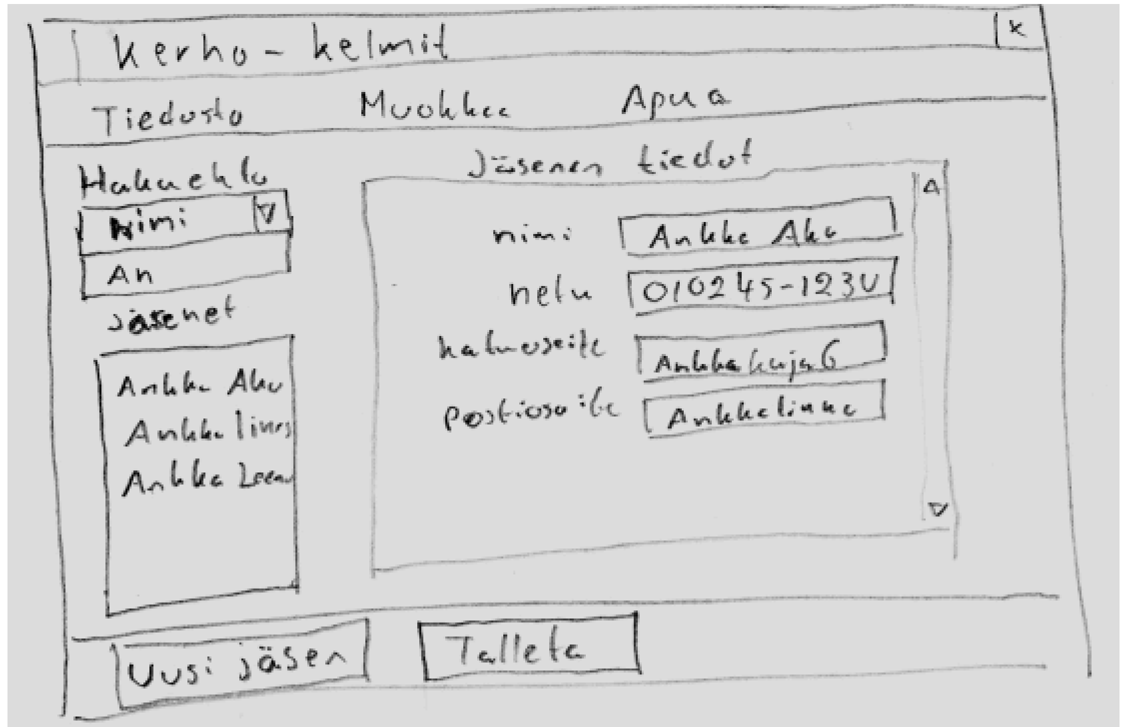


Kerhon tiedot on tallennettu vaikkapa tiedostoon nimet.dat (hakemistoon kelmit). Näin voimme ylläpitää samalla ohjelmalla useiden eri kerhojen tietoja. Mitäpä jos tiedostoa ei ole? Tällöin voi syynä olla kirjoitusvirhe tai se, ettei rekisteriä ole vielä edes aloitettu. Miten ohjelman tulee tällöin menetellä? Mikäli käyttäjä antaa tiedoston nimen, jollaista ei tunneta, tulostuu näyttöön:



Tällöin käyttäjä voi aloittaa syöttämään uusia jäseniä tai jos kirjoitti nimen väärin, hän voi ottaa menusta Avaa-valinnan ja antaa uuden nimen

Edellä on edetty siihen saakka, kunnes ohjelmassa on päädytty pääikkunaan.



Pääikkunassa on seuraava menurakenne.

```
Tiedosto         Muokkaa                 Apua
========         =======                 =====
Talleta          Lisää uusi jäsen        Apua
Avaa...          Poista jäsen...         Tietoja...
Tulosta...
Lopeta
```

Seuraavaksi voimme lähteä tarkastelemaan eri alakohtien toimintaa.

2.4.2 Hakeminen

Pääikkunan vasemmassa reunassa näkyy Hakuehto. Tästä voi valita minkä kentän mukaan etsitään. Tämän jälkeen tekstikenttään voi syöttää hakuehdon ja listaan tulee vain ne jäsenet joille haku toteutuu. Hakutermi saa löytyä valitusta kentästä mistä kohti vaan. Esimerkiksi jos kirjoitetaan hakuehtoon *s*, niin haetaan kaikki jäsenet joiden nimessä on s jossakin kohti.

Löytyneet jäsenet lajitellaan valitun hakukentän perusteella.

2.4.3 Muokkaaminen

Valittua jäsentä voidaan muokata menemällä tietoihin oikeaan kohtaan ja kirjoittamalla uusi arvo. Jos tietoon syötetään jotakin mikä ei kelpaa toimitaan seuraavasti.

```
Hetussa syötetty muodossa: 010243G1234

Tulee ilmoitus:

Väärä erotinmerkki
```

Samalla virheellinen syöttökenttä menee punaiseksi.

2.4.4 Lisää uusi jäsen

Luo uuden tyhjän jäsenen.

2.4.5 Poista jäsen

Poistaa listasta valitun jäsenen. Varmistaa ennen poistoa.

```
Poistetaanko jäsen Ankka Iines?

Ok        Cancel
```

2.4.6 Tulosta

Tulostaa hakuehdon täyttävät jäsenet erilliseen ikkunaan halutussa muodossa. Tässä "esikatselussa" voi vielä muuttaa tietoja ja sitten tulostaa paperille.

2.4.7 Lopeta

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.

2.4.8 Apua

Näyttää selaimessa ohjelman käyttöohjeen

2.4.9 Tietoja

Näyttää ohjelmasta tietoja vähän samaan tapaan kuin aloitusikkunassakin.

2.5 Hyväksymistestaus

Kun vaihe on valmis ja ohjelma täyttää sille asetetut vaatimukset se käydään läpi yhdessä asiakkaan ja tiimin kanssa, eli tässä tapauksessa jollakin kurssin ohjaajista. Usein tässä vaiheessa keksiikin työhön jotain parannuksia, mitä ei itse ole tullut ajatelleeksi. Viat yleensä korjataan seuraavaan vaiheeseen mennessä, mutta mikäli työ on jäänyt huomattavan keskeneräiseksi, niin kannattaa näyttää koko vaihe uudestaan.

Ensimmäisessä vaiheessa tutkitaan siis kirjoitettua käyttöohjetta, tietorakennetta ja piirrettyjä kuvia, jotka muodostavat alustavan suunnitelman ohjelman toiminnallisuudesta.

2.5.1 Tyypillisiä vikoja

Alussa tyypillisimmät viat liittyvät liian minimalistiseen dokumentaatioon. Dokumentaatiossa on hyvä pyrkiä täsmällisyyteen, sillä luotua tietoa hyödynnetään jatkuvasti projektin edetessä. Kaikki yleisimmät virheet johtuvat siitä, että yritetään oikaista asioissa, jotka vievät muutenkin vain murto-osan vaiheeseen käytetystä ajasta.

Tyypillistä on että mallitiedostot näyttävät jokseenkin tältä:

```
nimet.dat – ei näin
```

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
;sukunimi etunimi   |hetu       |katuosoite  |postinumero|postiosoite|kotipuhelin|työpuhelin|
Joku1          |00000-5555 |Ankkakuja 6 |12345      |ANKKALINNA |12-12324   |          |
Joku2          |…
…
```

Tiedoston sisällöstä saa nyt jonkinlaisen idean, mutta selkeyden vuoksi mallidataa tarvitsee useita rivejä ja sen tulisi koostua ”oikeista” arvoista. Dokumentaatiota kirjoittaessa pieni ajan säästäminen kostautuu useasti projektin edetessä. Kun sisällön tekee nyt kunnolla, niin samaa dataa voi käyttää hyödyksi viidennessä vaiheessa tietorakennekuvaa piirrettäessä, sekä mallitiedostona ohjelmaa luotaessa.

Lisäksi kannattaa miettiä ohjelman käyttötarkoitusta. Olisi rasittavaa jos Kerho aina varmistaisi saako käyttäjän lisätä, koska virheellisesti luodun henkilön tietoja voi kuitenkin jälkikäteen muokata. Käyttäjälle näytettävät varmistusdialogit sopivat peruuttamattomien muutoksien yhteyteen, mutta väärässä paikassa käytettynä ne hidastavat käyttöä täysin turhaan.

Ennen harjoitustyön näyttämistä kannattaa aina käydä tarkistamassa kurssin wikistä malliharjoitustyö ja tyypilliset harjoitustyön viat, jolloin selviää turhalta korjaamiselta.

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. pitkän merkkijonon pilkkominen osamerkkijonoihin annetun merkin kohdalta
2. loppuvälilyöntien poistaminen merkkijonosta
3. isojen ja pienien kirjainten muuttaminen merkkijonossa esimerkiksi:

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

4. hetun oikeellisuuden tarkistus

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

2.7 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*). Puuttuvien toimintojen kohdalla ohjelma voidaan laittaa 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.

2.8 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?

2.8.1 Kaikki samassa tietueessa

Eräs mahdollisuus olisi lisätä kunkin rivin loppuun jollakin erotinmerkillä harrastukset:

nimet.dat - harrasteet samalle riville

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
; sukunimi etunimi |hetu       |…|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.

2.8.2 Erimalliset tietueet

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

nimet.dat - harrasteet omalle riville

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
; sukunimi etunimi |hetu       |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.

2.8.3 Relaatiomalli tiedostoon

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 nimet.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.

nimet.dat - relaatiokannan päätaulu

```
Kelmien kerho ry
; Kenttien järjestys tiedostossa on seuraava:
;id|sukunimi etunimi   |hetu       |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 (hakemistoon kelmit), jossa tunnusnumerolla ilmaistaan kuka harrastaa mitäkin harrastusta.

harrastukset.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 nimet.dat. Sitten etsittäisiin ja tulostettaisiin kaikki rivit joissa tunnus on 2 tiedostosta harrastukset.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 2.1 Ketkä harrastavat?

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

2.8.4 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 2.2 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".

2.9 Graafiset käyttöliittymät

Luultavammin nopein tapa käyttöliittymäsuunnittelijalle tai ohjelmoijalle on suunnitella graafinen käyttöliittymä kynän sijaan suoraan jollakin nykyaikaisella graafisella ohjelmointiympäristöllä. Tähän osaan sitten lisätään heti tai jälkeenpäin itse toiminnallisuus. Tällaisia työkaluja on esimerkiksi *Eclipse (vaatii WindowBuilder pluginin), NetBeans, Visual Studio, C++Builder, Delphi* ja myös muiden ohjelmointikielten resurssityökalut.

Ohjelma suunnitellaan nimenomaan "piirtämällä" käyttäjälle näkyvä käyttöliittymän osa. Tässä tilanteessa on mahdollista pitää jopa asiakas mukana, jolloin ohjelman vaatimukset ja suunnitelmat selkeytyvät molempiin suuntiin. Ohjelmasta saattaa puuttua jotain tärkeitä ominaisuuksia, se saattaa olla liian vaikea käyttää, eikä ole myöskään tavatonta että alustavasti ohjelmaan on suunniteltu jopa tarpeettomia osia. Tilanteessa kiteytyy pitkälti se miksi johdanto-osuudessa esitellyt ketterien menetelmien arvot ovat käytännössä niin toimivia.

2.9.1 Komponentit

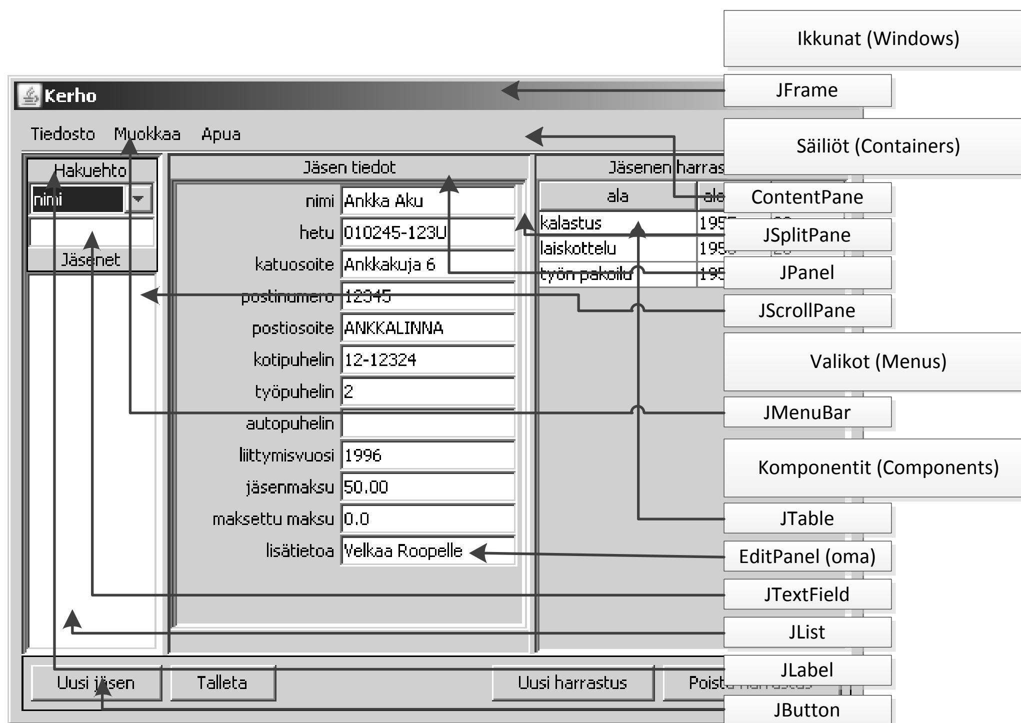
Tyypillisesti käyttöliittymät koostuvat ns. komponenteista. Ohjelmointikielelle tehdyt valmiit käyttöliittymäkirjastot sisältävät joukon valmiita komponentteja, kuten ikkunoita, paneeleita, tai vaikkapa nappeja. Yleensä komponentit rakentuvat muutamasta erilaisesta tyypistä, joskin käytettyjen kirjastojen välillä saattaa olla jonkinlaisia eroja.

Ikkunat (*Windows*) ovat korkean tason komponentteja, jotka sisältävät muita komponentteja. Ohjelmassa on yleensä yksi pääikkuna, jolla tosin voi olla lapsia, eli toisia ikkunoita. Tyypillisesti tyhjä ikkuna sisältää yläpalkin, jossa on ikoni, tekstiä, ikkunan kokoon vaikuttavat pikanäppäimet ja rasti sulkemista varten. Ikkunan "tyhjä" osio koostuu säiliöstä.

Säiliöt (*Containers*) ovat ikkunoita matalamman tason komponentteja, joiden tehtävä on helpottaa muiden komponenttien ryhmittelyä erilaisten sijoitteluiden (*layout*) avulla. Monimutkaiset rakenteet saattavat vaatia useiden sisäkkäisten ja rinnakkaisten säiliöiden käyttöä.

Valikot (*Menus*) ovat tapa jäsennellä kontrolleja. Sijoitellaan usein ohjelman yläreunaan tai esimerkiksi hiiren oikean painikkeen taakse.

Kontrollit (*Controls*), kuten esimerkiksi napit, tekstikentät, muokattavat tekstikentät, edistymispalkit ja valintalaatikot, ovat tyypillisesti matalimman tason komponentteja, jotka toteuttavat jotakin täsmällistä toiminnallisuutta.



Kuva 3.1 Kerhon käyttöliittymässä käytettyjä Swing-komponentteja

Komponenttien sijoittelussa ja niiden toiminnassa kannattaa matkia paljon muita ohjelmia. Toki valikkopalkki on mahdollista laittaa vaikka ikkunan alareunaan, mutta samalla varmasti kasvattaa käyttäjän kynnystä oppia ohjelman sujuva käyttö. Kannattaa myös miettiä ohjelman käyttötarkoitusta ja alustaa. Hiiri ole välttämättä ainoa tapa käyttää graafistakaan ohjelmaa, joten usein tarvittavaan ohjelmaan on hyvä olla jonkinlaisia käyttöä nopeuttavia pikanäppäimiä. Toisaalta helppokäyttöisimmänkään tietokoneohjelman käytettävyys tuskin siirtyy sellaisenaan kännykälle.

2.9.2 Omat komponentit

On kuitenkin selvää, ettei valmis kirjasto voi tarjota suoraan kaikkea tarpeellista. Tähän on käytännössä kolme erilaista lähestymistapaa. Ensimmäinen - yleensä tarpeettoman työläs - ratkaisu on luoda tarvittavan toiminnallisuuden tarjoavat osa itse. Tällaisen komponentin pitää täyttää tietty määrä sille asetettuja vaatimuksia, jonka jälkeen se on käytettävissä käyttöliittymässä. Käytännöllisempää kuin tyhjästä aloittaminen usein onkin ylikirjoittaa ja laajentaa haluttu toiminnallisuus jo valmiista komponentista.

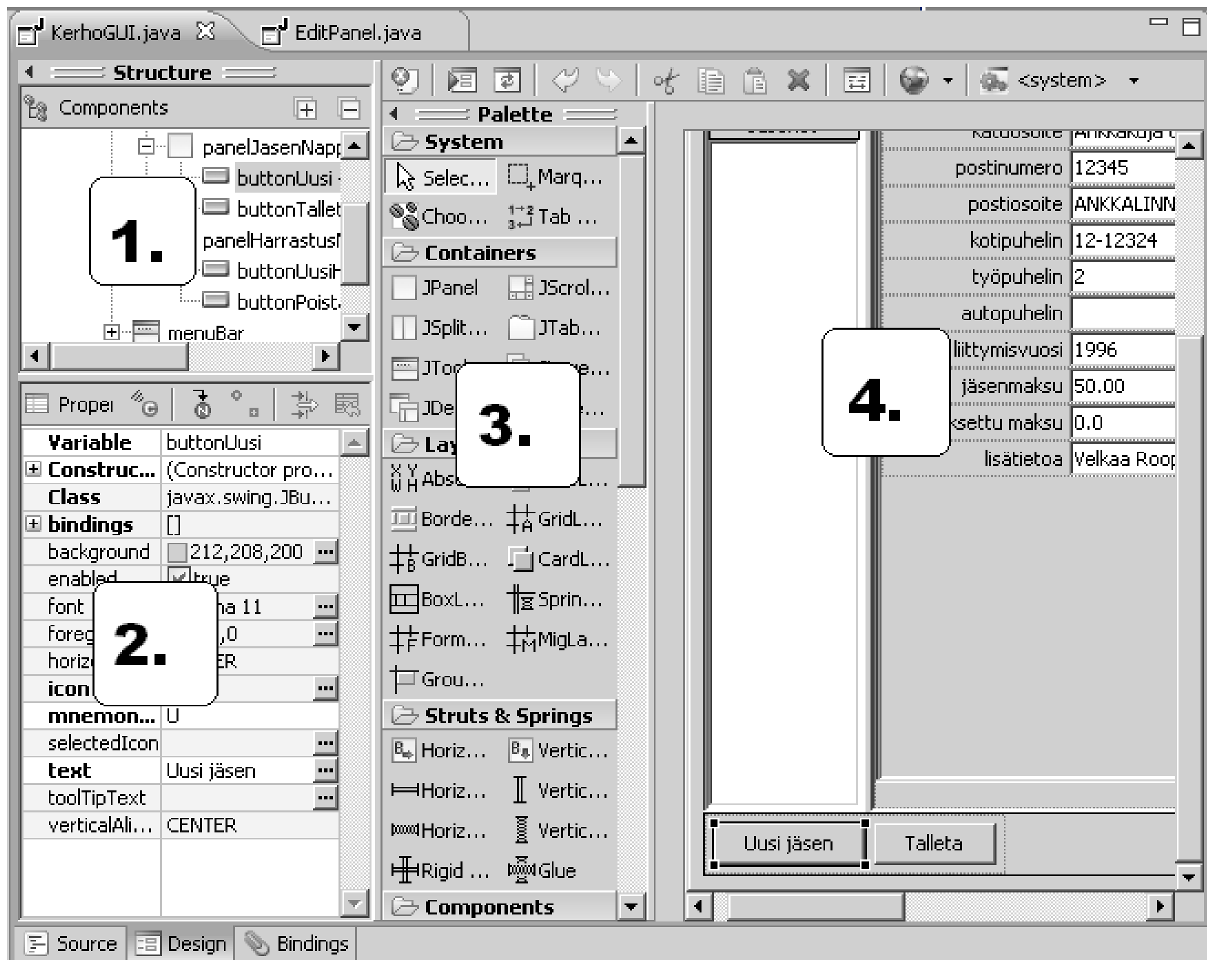
Ohjelmoinnissa hyvä nyrkkisääntö on, että samaa koodia ei kannata kirjoittaa kahdesti, vaan silloin se tulee refaktoroida esimerkiksi uuteen funktioon. Sama periaate toimii käyttöliittymien kohdalla. Kerho-ohjelmasta huomaamme, että jäsenten tiedot syötetään kenttiin, joissa vasemmalla puolella on tekstiä ja oikealla syötekenttä. Tällöinhän olisi kätevää, jos voisimme yhdistää nämä kaksi komponenttia yhdeksi kokonaisuudeksi. Tätä ratkaisua kutsutaan *koostamiseksi*. Käyttäessämme Javan Swing kirjastoa, voimme luoda esimerkiksi EditPanel –komponentin, jolla on vaihdettava tekstikenttä (`JLabel`) ja kirjoituskenttä (`JTextField`).

Kuva 3.2 Koostettu komponentti EditPanel

Kuva 3.3 EditPanel -komponentteja allekkain

2.9.3 Graafisten käyttöliittymien suunnittelutyökalut

Käytettävästä ympäristöstä ja ohjelmointikielestä riippumatta käyttöliittymien suunnittelutyökalut muistuttavat hyvin paljon toisiaan. Esimerkkinä on käytetty *Eclipseen* asennettua *WindowBuilder Pro* –laajennusta Javan Swing-ympäristössä. Toiminnallisuuteen ei kuitenkaan syvennytä kuin pinnallisesti, koska työkalujen kehittyminen on nopeaa ja on mahdollista, että seuraavan version kohdalla tämäkin moniste on jo vanhentunut.



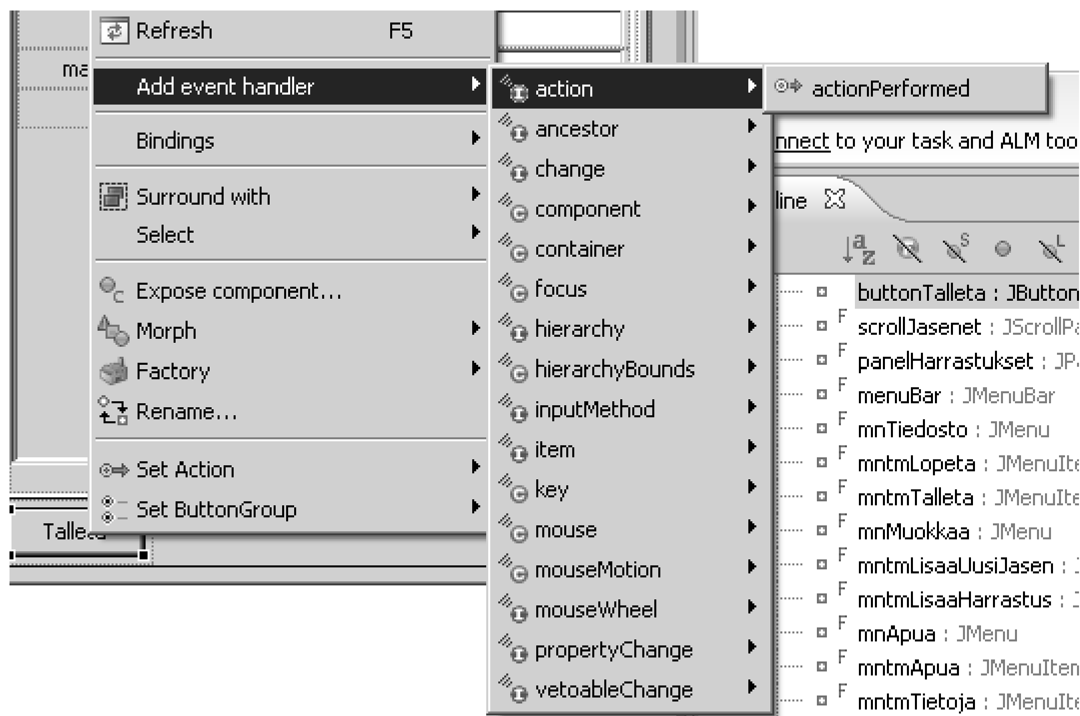
Kuva 3.4 Eclipse ja WindowBuilder Pro –plugin

1. Komponentit ja niiden sisäkkäinen rakenne (*Structure*). Kannattaa myös huomata että kaikki esikatselussa näkyvä, pääikkunaa myöten, on oma komponenttinsa.
2. Ominaisuudet (*Properties*). Komponentin voi aktivoidaa joko rakenne- tai esikatseluikkunasta, jolloin sen ominaisuuksia voi muuttaa. Erilaisilla komponenteilla on toisistaan eroavat ominaisuudet. Tällaisia ominaisuuksia ovat esimerkiksi napissa lukeva teksti tai sen koko.
3. Käytettävissä olevat työkalut ja komponentit (*Palette*). Kokoelma valmiita komponentteja, joita pystyy ottamaan käyttöön vetämällä halutun joko rakenne- tai esikatseluikkunaan
4. Esikatselu (*Preview*).

Ohjelmointikieli ja siinä käytetyt kirjastot kyllä tuovat itse ohjelmakoodiin suuriakin eroja. Nykyään monet kirjastot käyttävät hyväkseen XML-tiedostoja, joihin tallennetaan ulkoasun ja komponenttien ominaisuudet samaan tapaan kuin html-tiedostoihin. Kurssilla käytetty Javan Swing-kirjasto ei kuitenkaan tätä mahdollisuutta ainakaan vielä tarjoa. XML:n käytöllä saavutetaan ainakin teoriassa parempi siirrettävyys eri järjestelmien ja laitteiden välillä. Eri tekniikoiden välillä ei kuitenkaan ole mitään yhtenäistä standardia, mutta XML-pohjaiset toteutukset on kuitenkin helpompi tulkita.

2.9.4 Tapahtumat

Käyttöliittymien toiminnallisuus toteutetaan tapahtumien (*event*) avulla. Hiiren oikea näppäin halutun komponentin päällä avaa valikon josta voi tutkia erilaisista käyttäjän toimista aktivoituvia tapahtumia. Oikean toiminnallisuuden kanssa pitää hieman miettiä. Miten painikkeen (*Button*) painaminen tapahtuu? Nopeasti voisi ajatella että kun hiiren vie painikkeen päälle ja painaa (`MousePressed`), niin on intuitiivista että seuraa tapahtuma. Näinhän ei kuitenkaan ole, vaan yleensä tapahtuma seuraa vasta sitten kun hiiri on painettu alas ja päästetty ylös (`MouseClicked`). Oikea tapahtuma painikkeelle on kuitenkin `ActionPerformed`, joka huomio myös näppäimistön avulla tehdyt valinnat.



Kuva 3.5 WindowBuilder Pro ja valikko tapahtumien hallintaan

Tapahtuman luominen lisää seuraavat rivit koodiin. Ohjelma lisää painikkeelle uuden tapahtumakuuntelijan. Ohjelmakoodin toiminnan ymmärtäminen vaatii kuitenkin suhteellisen edistynyttä olio-ohjelmoinnin tietämystä, joten ei kannata säikähtää vaikka tekninen toteutus ei täysin aukeaisikaan.

```
buttonTallenna.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        //tähän voi kirjoittaa omaa ohjelmakoodia
        tallenna();
    }
});
```

Lohkoon jossa kutsumme `tallenna()`-metodia olisimme tietysti voineet kirjoittaa halutun toiminnallisuuden suoraankin. Metodikutsu on kuitenkin parempi tapa, koska

koodista tulee näin paremmin jäsenneltyä ja tulkittavaa. On myös mahdollista että tarvitsemme samaa toiminnallisuutta useampaan kertaan. Tällöin voisimme laittaa esimerkiksi *CTRL+S* pikanäppäinyhdistelmän osoittamaan samaan metodiin.

Tässä vaiheessa ohjelmamme ei osaa kuitenkaan vielä tallentaa mitään, joten metodin tehtävä on vain kertoa se erillisessä `dialog`-ikkunassa.

```
private void tallenna() {
    JOptionPane.showMessageDialog(null, "ei osata vielä tallentaa");
}
```

3. 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.

3.1 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 3.1 Kävelyohjeet

Yritä kirjoittaa ohjeet siitä miten kävellään.

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

3.2 Lajittelu yleisesti

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

3.2.1 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.

3.2.2 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!

3.2.3 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 3.2 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 3.3 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 3.4 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!

3.2.4 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 3.5 Kuplalajittelu

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

3.2.5 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.

3.2.6 Algoritmin parantaminen

Kaikki edelliset algoritmit ovat kompleksisuudeltaan normaalitapauksessa samanlaisia.

Tehtävä 3.6 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 3.7 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 3.8 Lisäys oikealle paikalleen vaiko lisäys loppuun ja lajittelu?

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

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

3.3 Algoritmin tarkentaminen

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

3.3.1 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
```

3.3.2 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.
```

3.4 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?

3.4.1 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*)!

3.4.2 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 3.9 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 3.10 Puolitushaun kompleksisuus

Mikä on puolitushaun kompleksisuus?

3.5 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 3.11 Kumin paikkaus

Kirjoita algoritmi polkupyörän kumin paikkaamiseksi.

Tehtävä 3.12 Sunnuntai–ilta

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

Tehtävä 3.13 Onkiminen

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

Tehtävä 3.14 Järjestyksen kääntäminen päinvastaiseksi

Kirjoita algoritmi pöydälle levitetyn 13 kortin kääntämiseksi päinvastaiseen järjestykseen.

4. 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.

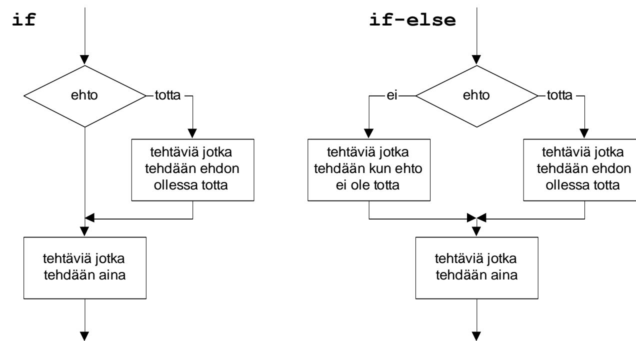
4.1 Ehtolauseet

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 4.1 Ehtolauseet

Tehtävä 4.1 Ajanlisäys

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

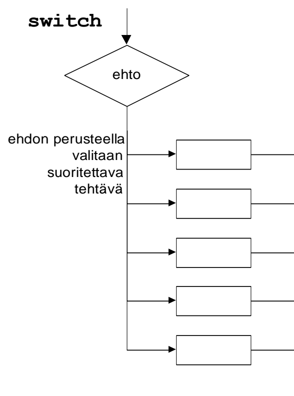
Tehtävä 4.2 Postimaksu

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

4.2 Valintalauseet

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

```
Yritys myy verkkokaupasta erilaisia tietoteknisiä tuotteita. Jokaisella
tuoteryhmällä on vastaava henkilö, jonka sähköpostiin halutaan ohjata oikeat
tukipyynnöt. Käyttäjä valitsee tuotekategorian alasvetovalikosta, jonka
perusteella pyyntö lähtetään oikealle henkilölle.

  Tietokoneet  ->   Matti
  Puhelimet    ->   Marko
  Kamerat      ->   Terttu
  Audio,Muut   ->   Niko
```



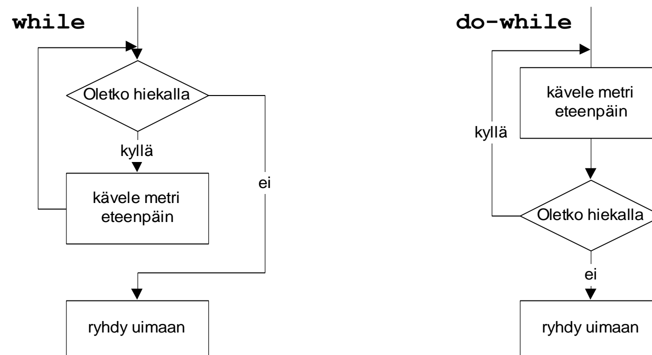
Kuva 4.2 `switch`–valintalause

Tehtävä 4.3 Korvaaminen ehtolauseilla

Mieti kuinka valintalauseen logiikka korvattaisiin ehtolauseiden avulla.

4.3 Silmukat

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



Kuva 4.3 `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 4.4 Uiminen

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

Tehtävä 4.5 Ynnää luvut 1–100

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

4.4 Muuttuja-käsite

Algoritmeissa tarvitaan usein muuttujia.

```
kellonaika
rahan määrä
```

4.4.1 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 4.6 Vuokaavio

Piirrä jaollisuuden testausalgoritmista vuokaavio.

4.4.2 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 4.7 Algoritmin parantaminen

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

Tehtävä 4.8 Pöytätesti

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

4.4.3 Yksiulotteiset taulukot, alkiot rivissä

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:

```
80. 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 4.9 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 4.10 Korttien poisto

Kirjoita algoritmi kuvakorttien poistamiseksi taulukosta käyttäen indeksejä.

4.4.4 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.
```

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

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 4.11 Korttien poisto osoittimia käyttäen

Kirjoita algoritmi kuvakorttien poistamiseksi taulukosta käyttäen osoittimia.
Pöytätestaa algoritmi.

4.4.5 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):

	0	1	2	3	4	5	6
0	♠ 7				♠ A		
1		♣ K		♥ 5			
2			♦ A				
3	♥ 7	♠ 2	♦ 2	♠ 9	♥ 6	♥ 3	♦ 7
4			♥ 2				♥ J

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

```
peli[3][1] = ♠2
```

Tehtävä 4.12 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`).

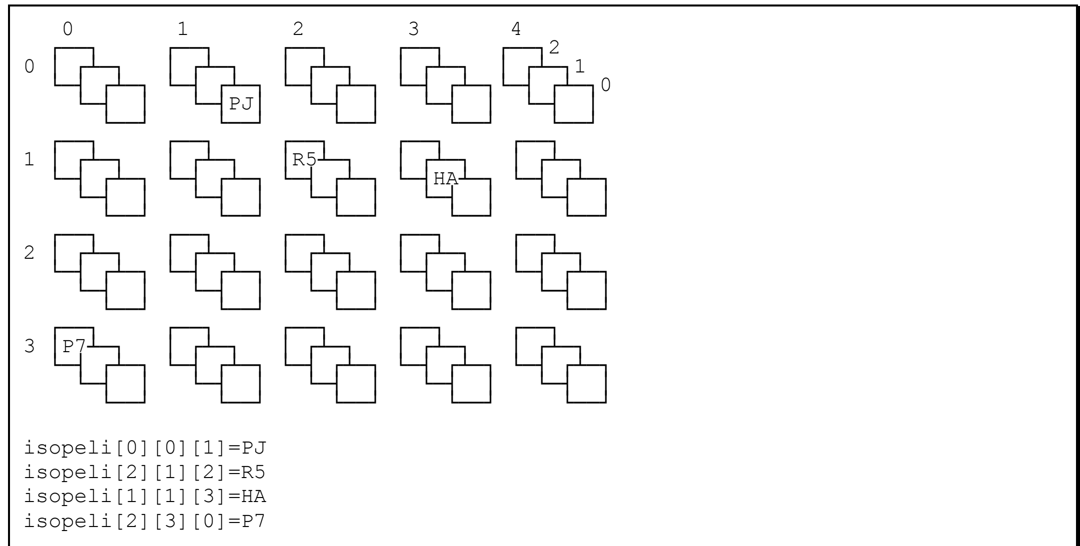
Tarvittaessa moniulotteiset taulukot voidaan muodostaa yksiulotteisenkin taulukon avulla. Esimerkin taulukko voitaisiin muodostaa yksiulotteisesta taulukosta siten, että yksiulotteisen taulukon 7 ensimmäistä alkiota kuvaisivat matriisin 0:ta riviä, 7 seuraavaa matriisin ensimmäistä riviä jne.

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:



Tehtävä 4.13 Sijoitus 3–ulotteiseen taulukkoon

Esitä 5 muuta sijoitusta taulukkoon.

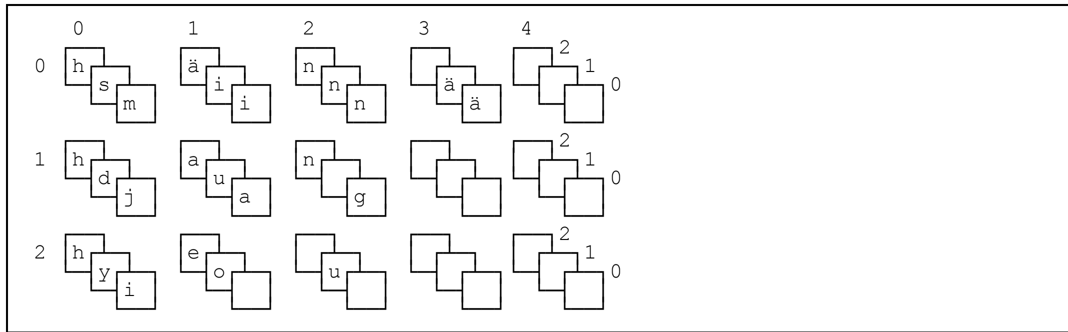
Tehtävä 4.14 3–ulotteinen taulukko 1–ulotteiseksi

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 4.15 Kolmiulotteinen taulukko

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

Tehtävä 4.16 Neliulotteinen 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.

4.4.6 Sekarakenteet

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

```
0         1         2         3
0123456789012345678901234567890123
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 4.17 Sanojen muuttaminen

Mitä ongelmia edellä olisi, mikäli yhdenkin sanan pituutta kasvatettaisiin?

Voitaisiinko edellä käyttää samoja sanoja uudestaan ja jos niin miten?

4.5 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:              C++:lla:
                       sivu[1] = &Batman;
Batman               Javalla:
Gotham City            Sivu[1] = Batman;
999
```

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.

4.6 Aliohjelmat avuksi

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 4.18 Lihapullan paistaminen

Täydennä edellinen paistamisalgoritmi. Onko parametreja tarpeeksi?

4.7 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:

4.7.1 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 A	B	2 C	3 D	V	4 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.

4.7.2 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 4.19 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?

4.7.3 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 4.20 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 4.21 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?

4.7.4 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 4.22 de Morganin kaava

Todista oikeaksi myös toinen *de Morganin* kaava:

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

Tehtävä 4.23 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 4.24 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))

4.8 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 4.25 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 4.26 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?

5. 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.

5.1 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:

5.1.1 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]
```

5.1.2 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
```

5.1.3 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
```

5.1.4 C#

```
// C# -kieli
public class Olen
{
   public static void Main(string[] args)
   {
      System.Console.WriteLine("Terve! Olen C#-kielella kirjoitettu ohjelma.");
   }
}
-------------------------------------------------------------------
- Kirjoita HelloWorld.cs jollakin editorilla
- käännä: csc HelloWorld.cs
- aja: HelloWord
```

5.1.5 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]
```

5.1.6 Fortran

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

5.1.7 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;
```

5.1.8 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]
```

5.1.9 APL

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

5.1.10 Modula–2

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

5.1.11 Lisp

```
; Common Lisp
(print "Terve! Olen Common Lisp-kielellä kirjoitettu ohjelma.")
-------------------------------------------------------------------
- Kirjoita rivi jossakin Common Lisp-tulkissa, kuten CLISP
```

5.1.12 FORTH

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

5.1.13 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.

5.2 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.

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.

C# on Microsoftin vuonna 2000 julkaisema oliopohjainen ohjelmointikieli. Syntaksiltaan se on muistuttaa paljon *C*:tä, mutta sen kehityksessä on otettu paljon inspiraatiota myös *Javasta*, *Delphistä* ja *Visual Basicista*. *Javan* ja *C#*:n samankaltaisuus on ollut jonkinlainen kiistelyn aihe jopa kielten kehittäjien kesken. Kielten viimeisimmät päivitykset ovat kuitenkin tuoneet kehittyneimpiin ominaisuuksiin huomattavia eroja. Hyvänä esimerkkinä *C#*:n 3.0:ssa esitelty *Language Integrated Query (LINQ)* laajennus toi kieleen monia funktionaalisen ohjelmoinnin ominaisuuksia. *Javan* tavoin myös *C#* ohjelmat ajetaan virtuaalikoneen päällä. Käytettävä virtuaalikone perustuu Microsoftin *Common Language Infastructure* spesifikaatioon. Suosituin *CLI* toteutus on Microsof-

tin oma virtuaalikone, mutta myös muutama vapaan lähdekoodin vaihtoehto kuten *Mono* on saatavilla.

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.

6. 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:

https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/alkeet/

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*.

6.1 C#:sta Javaan

Mikäli olet käynyt jo Ohjelmointi 1 kurssin C#:lla, niin ei turhaan kannata säikähtää kielen vaihtumista. Java ja C# ovat ominaisuuksiltaan hyvin samankaltaisia kieliä. Molemmat ovat staattisesti tyypitettyjä olio-ohjelmointikieliä, molempien suoritus tapahtuu virtuaalikoneessa ja lisäksi ne ovat hyvin samankaltaisia jopa syntaksiltaan.

Erot käsitellään tässä monisteessa sitä mukaan kun niitä tulee vastaan. Merkittävimmät niistä koskevat kuitenkin vasta kielten uudempia ja kehittyneempiä ominaisuuksia, eikä

niitä ole edes Ohjelmointi 1 kurssin puitteissa tullut vastaan. Internetissä on monia eroista kielten eroa käsitteleviä artikkeleita, mutta yksi hyvä paikka on Ville Salosen kandinaatintutkielma:

```
https://svn.cc.jyu.fi/srv/svn/ohj2/ville/Ville_Salonen_-_Javasta_Csharppiin_-_Kandidaatintutkielma.pdf
```

Ville on myös pohtinut esseessään hieman edistyneempää asiaa ja syitä eroavaisuuksiin. Näkökulma on enemmänkin "konepellin alta", joten seuraava linkki voi olla hyvää luettavaa tämän kurssin jälkeen.

```
https://svn.cc.jyu.fi/srv/svn/ohj2/ville/Javasta_Csharpiin_pintaa_syvemmalta.pdf
```

Tarpeen vaatiessa monisteen muutamaan ensimmäiseen lukuun Java-kielisten esimerkkien rinnalle on kirjoitettu myös C# versiot. Kurssin kannalta C#:n osaaminen ei kuitenkaan ole olennaista.

6.2 Hello World! Java ja C#–kielillä

alkeet.hello.Hello.java - ensimmäinen Java ohjelma

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

C#-versio

```csharp
// C# -kieli
class Hello
{
   public static void Main(string[] args)
   {
      System.Console.WriteLine("Hello world!");
   }
}
```

Ohjelmat ovat selvästikin hyvin samankaltaiset. Ensimmäiseksi kannattaa kiinnittää huomio syntaksiin. Javan (yleisesti käytetty) tapa aloittaa lohko `{` -merkillä ilman rivinvaihtoa eroaa hieman C#:n vastaavasta. Molemmat ohjelmat kääntyisivät kummallakin tavalla, mutta koodin luettavuus helpottuu, kun käyttää virallista tyyliä. Lisäksi kehitysympäristöjen tarjoamien koodin formatointityökalujen käyttö on helpompaa jos oletusasetuksia ei tarvitse erikseen muuttaa. Vähän tärkeämpi koodin muotoiluun liittyvä asia on, että funktiot on tapana kirjoittaa *Java*ssa pienellä, kun *C#*:n käytäntö on nimetä ne isolla. Jälleen kuitenkin kumpikin tyyli menee molempien kääntäjistä läpi.

On myös helppo huomata että käytettyjen kirjastojen ja funktioiden nimet ovat muutenkin erilaiset. Silti usein tulee vastaan tilanteita, joissa käytettyjen kirjastojen nimet ovat, jolleivät peräti samoja, niin ainakin hyvin samankaltaisia. Tähän liittyy kuitenkin se ansa, että samallakin tavalla nimettyjen funktioiden toiminta saattaa erota ratkaisevasti toisistaan! Esimerkiksi C#:issa usein viedään parametrina alku ja osavälin pituus kun Javassa viedään parametrina alku ja loppu-indeksi, joka ei enää tule mukaan. Jos aloi-

tetaan esimerkiksi jonon alusta, niin silloin samat parametrin arvot toimivat molemmilla, mutta muussa tapauksessa ei:

```
         //   012345
String jono = "abcde";
C#:   jono.Substring(0,3)  => abc
      jono.Substring(1,3)  => bcd
Java: jono.substring(0,3)  => abc
      jono.substring(1,3)  => bc
```

Javassa jälkimmäinen lause tuottaa *bc*, mutta *C#*:ssa *bcd*. Syy selviää metodien dokumentaatiota tutkimalla. Ensimmäinen esimerkki tulostaa merkkijonon joka alkaa paikasta (kirjaimesta) kaksi ja loppuu kolmanteen paikkaan, kun taas toinen ottaa toisesta paikasta lähtien kolme merkkiä.

Java

a	b	c	d	e
0	1	2	3	4

C#

a	b	c	d	e
0	1	2	3	4

Kannattaa siis aluksi varmistaa dokumentaatiosta mitä onkaan tekemässä. Ohjelmointikielien samankaltaisuuksista huolimatta jokaisella kielellä on omat tapansa toteuttaa asioita. Se minkä vuoksi Javan `substring` metodin viimeinen parametri `3` ei sisällytäkään vastaavaa merkkiään palautettavaan merkkijonoon, johtuu Javan käytännöstä jättää merkki pois (*exclude*), joka on tyypillistä toiminnallisuutta useille Java-funktioille. Valittu tapa säästää käytännössä usein turhia `-1` laskuja.

Tehtävä 6.1 Nimi ja osoite

Kirjoita Java–ohjelma joka tulostaa:

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

6.3 Tekstitiedostosta toimivaksi konekieliseksi versioksi

6.3.1 Kirjoittaminen

Ohjelmakoodi kirjoitetaan millä tahansa tekstieditorilla tekstitiedostoon vaikkapa nimelle `Hello.java`. Yleensä tiedoston tarkentimella annetaan sille tyyppi, jonka avulla käyttöjärjestelmä ja ohjelmat saavat lisätietoa minkälainen tiedosto on kyseessä.

6.3.2 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.

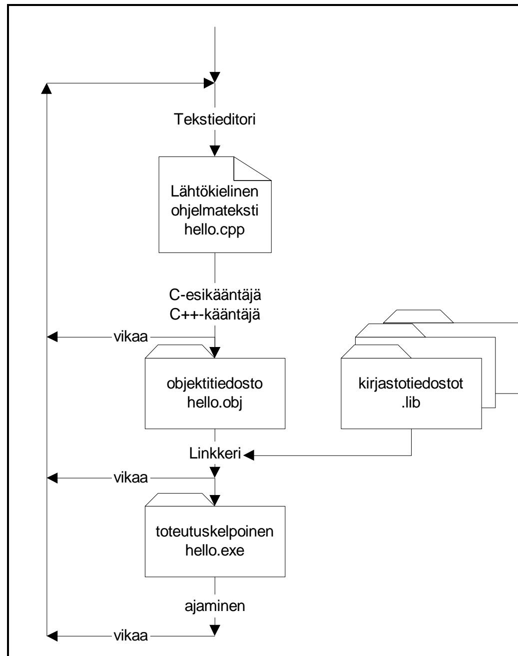
6.3.3 Linkittäminen

Linkittäjällä (kielestä riippumaton ohjelma) liitetään kirjastorutiinit käännettyyn objektitiedostoon. Linkittäjä korvaa tyhjät kutsut varsinaisilla kirjastorutiinien osoitteilla, kunhan se 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*).

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



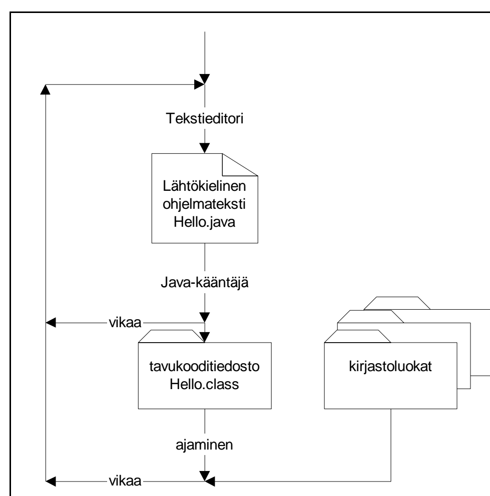
Kuva 6.1 Ohjelman kääntäminen ja linkittäminen

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 6.2 Java-ohjelman kääntäminen ja linkittäminen

6.3.5 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 – käytännössä se tietysti säilytetään mm. ylläpidon takia. Siis me kirjoitamme tekstiä, joka ehkä (toivottavasti) muistuttaa Java–kielen syntaksin mukaista ohjelmaa. Vasta käännös ja linkkaus tekevät todella toimivan ohjelman.

6.3.6 Integroitu ympäristö

On olemassa ohjelmankehitysympäristöjä, joissa editori, kääntäjä ja linkkeri (sekä mahdollisesti debuggeri, virheenjäljitin) on yhdistetty käyttäjän kannalta yhdeksi toimivaksi kokonaisuudeksi. Esimerkkeinä *Eclipse, 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*, *JavaDoc*-yhteistoiminta Java-kehittimissä), jotka perinteisesti ovat olleet *CASE*-suunnitteluohjelmien aluetta. Lisäksi myös koodin generointi kaavioista onnistuu rajoitetusti.

6.4 Ohjelman yksityiskohtainen tarkastelu

Seuraavaksi tutkimme ohjelmaa lause kerrallaan:

alkeet.hello.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!");
  }
}
```

6.4.1 Kommentointi

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

tai

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

Ohjelman alussa on kommentoitu mitä ohjelman 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.

6.4.2 Miten kommentoida

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 puolivälistä
```

- 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) parametreja, 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.

6.4.3 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.

Dokumentaatiokommenttien käyttö helpottaa dokumentaation hallitsemista. Muuttaessaan funktion toiminnallisuutta ohjelmoijan on helppo muuttaa myös dokumentaatiota, koska se on saatavilla samasta paikasta. Lisäksi kehittyneet ohjelmointiympäristöt osaavat lukea ja näyttää oikein muodostetun dokumentaation automaattisesti koodia kirjoittaessa.

Kommentointi kannattaa käytännössä tehdä yksittäisten metodien tarkkuudella. Myös tämän monisteen esimerkeistä on pyritty tekemään *JavaDoc*in mukaisia

Lisäinformaatiota dokumentaatioon annetaan *tagien* muodossa @-merkillä ja sen jälkeen tulevalla avainsanalla. Esimerkistä löytyvät koodin tekijän ja version ilmoittavat `@author` ja `@version` –tagien käyttäminen on hyödyllistä vaikkapa tiimeissä tapahtuvissa ohjelmointiprojekteissa. Kaksi muuta tärkeää merkintää ovat aliohjelmatasolla käytettävät `@param` ja `@return`, joilla kuvaillaan halutut parametrit ja palautettava arvo.

Katso lisää ohjeita *JavaDoc*in ja *tagien* käytöstä osoitteessa

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

6.4.4 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.

6.4.5 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.

C# osaajille tiedoksi, että *using* sanalla on vastaava toiminnallisuus on *C#*:ssa kuin *import*illa. *C#* ei kuitenkaan tue jokerimerkin käyttöä, joten kaikki kirjastot määritellään siinä täsmällisesti.

6.4.6 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 `}`.

6.4.7 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. Huom! Javassa vain luokkien nimet aloitetaan isolla kirjaimella.

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. Luok-

kametodi 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 parametreina. Käynnistys parametreja voi olla nolla tai useita.

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

`)` Metodin parametrilistan (argumenttilistan) loppusulku.

Ohjelma

alkeet.hello.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]);
  }
}
```

C#-versio tervehdyksestä parametrina

```
class Hello3
    {
        public static void Main(string[] args)
        {
            for (int i = 0; i < args.Length; i++)
            {
                Console.WriteLine("Parametri " + i + ": " + args[i]);
            }
            Console.ReadKey();
        }
    }
```

Tulostaisi seuraavalla tavalla:

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

6.4.8 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.

6.4.9 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]);
```

6.4.10 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.

6.4.11 Isot ja pienet kirjaimet

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

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

6.4.12 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.

6.4.13 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.

```
alkeet.hello.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 6.2 Terve maailma!

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

Tehtävä 6.3 Nimi ja osoite vakioksi

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

6.4.14 Vakiolukuarvot

Vakiomäärittelyä voitaisiin käyttää esimerkiksi kokonaislukuvakioiden määrittelemiseen:

```
alkeet.kuutio.Kuutio.java - monikulmion tiedot vakioksi
```

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

C#-versio Kuutio-ohjelmasta.

```
/**
 * <summary> Ohjelma tulostaa tietoja kuutiosta </summary>
 */
class Kuutio
{
   public const string TAHOKAS = "Kuutiossa";
   public const int KARKIA = 8;
   public const int SIVUTASOJA = 6;
   public const int SARMIA = 12;

   public static void Main(string[] args)
   {
      System.Console.Write(TAHOKAS + " on " + KARKIA + " kärkeä,");
      System.Console.Write(" " + SIVUTASOJA + " sivutasoa ja");
      System.Console.WriteLine(" " + SARMIA + " särmää.");
   }
}
```

Tehtävä 6.4 Tetraedri

Muuta edellistä ohjelmaa siten, että tulostetaan samat asiat tetraedristä.

7. 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
- yksikkötestit

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:

```
https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/muuttujat/
```

7.1 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.

7.2 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
```

Primitiivimuuttujiin liittyvät myös C#:n ja Javan suurimmat erot. Ensimmäiseksi ainoastaan positiivisille luvuille tarkoitettua *unsigned* tietotyyppiä ei Javasta löydy, vaan primitiivityypit voivat olla aina sekä positiivisia, että negatiivisia. Lisäksi C# on eräässä mielessä Javaa puhtaampi oliokieli, koska ohjelmoijalle päin kaikki sen tietotyypit käyttäytyvät kuin oliot (periytyvät `object`-kantaluokasta). Käytännössä C#:ssa kuitenkin primitiivit muunnetaan olioiksi vasta kun se on tarpeellista (*boxing*), tai vastaavasti arvo voidaan muuttaa takaisin primitiivityypiksi (*unboxing*). Tämä siksi että malli mahdollistaa myös tehokkaan tiedonkäsittelyn ja laskennan. Olioiden käsittely on huomattavasti raskaampaa kuin primitiivimuuttujien.

Katso lisää Javan tietotyypeistä linkistä:

http://download.oracle.com/javase/books/tutorial/java/nutsandbolts/datatypes.html

7.2.1 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:

muuttujat.matka.MatkaScan.java - mittakaavamuunnos 1:200000 kartalta

```
package muuttujat.matka;
import java.util.*;
/**
 * Ohjelmalla lasketaan mittakaavamuunnoksia 1:200000 kartalta
 * @author Vesa Lappalainen
 * @version 1.1 / 25.01.2007
 */
class MatkaScan {
  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>");

    Scanner in = new Scanner(System.in);
    String s = in.nextLine();
    if ( s.equals("") ) { System.out.println("Kiitti"); 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.

Syötteen lukeminen onnistui aika kivuttomasti `java.util`-paketista löytyvän `Scanner`-luokan avulla (luokista lisää myöhemmissä luvuissa). Kuten ohjelmoinnissa yleensäkin, niin saman asian voi toteuttaa kuitenkin monella tavalla. Java oli esimerkiksi vuosia ilman helppokäyttöistä tähän työhön soveltuvaa työkalua, jolloin lukemisen toteuttaminen oli huomattavasti monimutkaisempaa.

muuttujat.matka.Matka.java - mittakaavamuunnos 1:200000 kartalta

```
…
    // 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);
…
}
```

Tehtävä 7.1 Vakion korvaaminen

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

7.2.2 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://www.oracle.com/technetwork/java/codeconv-138413.html
```

Tehtävä 7.2 Avainsanat

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

Tehtävä 7.3 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;
```

7.2.3 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 7.4 Muuttujien esittely

Esittele edellisissä sijoitus –esimerkeissä tarvittavat muuttujat.

7.2.4 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 ennen kuin sille on annettu alkuarvo.

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

7.3 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) >
```

7.3.1 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`.

7.3.2 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.

7.3.3 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 kysyInt(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 = kysyInt("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 7.5 Oletuksen tulostaminen

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

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

7.3.4 Apuluokat

Seuraava kysymys sitten onkin että mihin tuo apumetodi `kysyInt` 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 kirjoitetaan? Yleiskäyttöisyyden nimissä tuo luokka kannattaa kirjoittaa omaan tiedostoonsa.

Kirjoitammekin koodin vaikkapa tiedostoon Syotto.java:

muuttujat.syotto.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 kysyInt(String kysymys, int oletus)
  {
...
  }

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

7.3.5 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.

7.3.6 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 `kysyInt` löytyy.

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

Tehtävä 7.6 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 7.7 Mittakaavan kysyminen

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

7.4 Viitteet

7.4.1 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.

muuttujat.jono.Jonotesti.java - merkkijonoviitteet

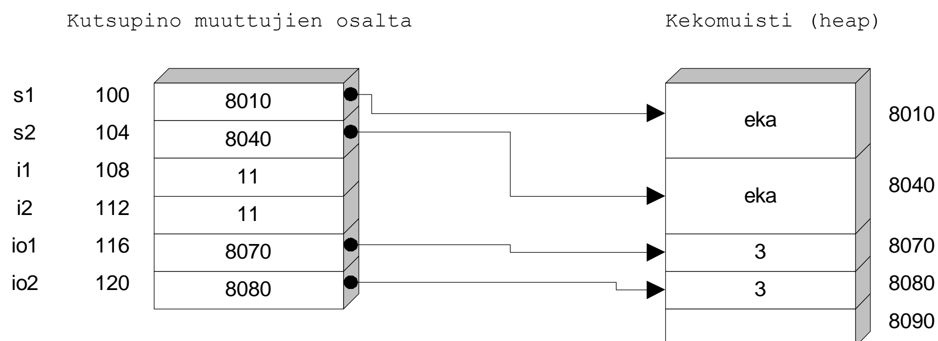
```
/**
 * Tutkitaan olioviitteiden käyttäytymistä
 * @author Vesa Lappalainen
 * @version 1.0, 08.01.20003
 */
class Jonotesti {

  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.

7.4.2 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 7.1 Olioviitteet

7.4.3 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.

7.4.4 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 )
```

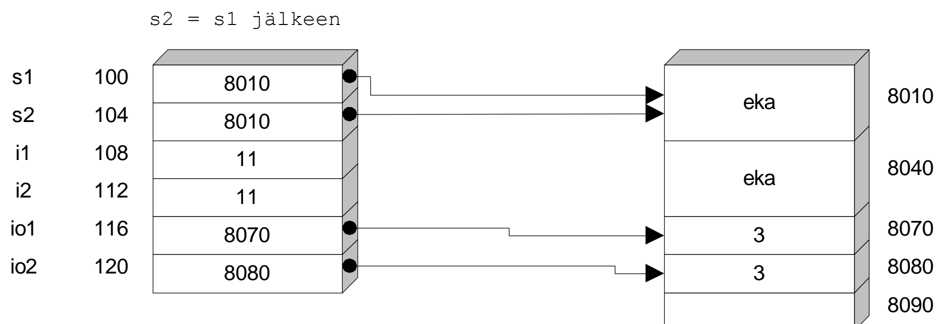
mikä ei ole totta. Javan primitiivityypit sen sijaan sijoittuvat suoraan arvoina pinomuistiin (tai myöhemmin olioiden attribuuttien tapauksessa oliolle varattuun muistialueeseen). Siksi vertailu

```
( 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.

7.4.5 Viitteeseen sijoittaminen

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



Kuva 7.2 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).

7.4.6 null-viite

Viitemuuttujan arvo voi olla myös `null`. Tämä tarkoittaa sitä, 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
```

7.5 Aliohjelmat (metodit, funktiot)

Eräs ohjelmoinnin tärkeimmistä rakenteista on aliohjelma. C–kielessä kaikkia erityyppisiä 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 se, 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.

7.5.1 Parametriton aliohjelma

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

muuttujat.matka.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.kysyInt("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.

7.5.2 Funktiot ja parametrit

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

```
    ohjeet();
    matka_mm = Syotto.kysyInt("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):

muuttujat.matka.Matka_a3.java - erilaisia funktioita /**

```
 * Ohjelmalla lasketaan mittakaavamuunnoksia 1:200000 kartalta
 * @author Vesa Lappalainen
 * @version 1.0 / 05.01.2003
 */
public class Matka_a3 {
  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.");
  }

  /**
   * Muuttaa mm mittakaavan mukaisesti kilometreiksi
```

```
  * @param matka_mm muutettavat millit
  * @return mittakavan mukaiset kilometrit
  */
 private static double mittakaava_muunnos(int matka_mm)
 {
   return matka_mm*MITTAKAAVA/MM_KM;
 }

 /**
  * 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) {
   int    matka_mm;
   double matka_km;

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

}
```

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.kysyInt` 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!

7.5.3 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:

muuttujat.matka.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).

7.5.4 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:

muuttujat.funktio.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 7.8 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 7.9 String vs. StringBuffer

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

7.5.5 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.kysyInt("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 kumpi 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 parametrina 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 7.10 Math-luokka

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

Tehtävä 7.11 Funktiot

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

Tehtävä 7.12 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 7.13 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?).

7.5.6 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.

7.5.7 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.

Jos halutaan että aliohjelma kuitenkin tulostaa, niin useimmiten sille kannattaa siinä tapauksessa viedä parametrina tietovirta johon tulostetaan. Samoin tulostavien aliohjelmien nimessä kannattaa tavalla tai toisella ilmaista että aliohjelma aikoo tulostaa. Palaamme tähän esimerkin kanssa seuraavissa luvuissa. Alla kuitenkin pikainen esimerkki:

muuttujat.tulostus.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.
  }
}
```

7.5.8 Jäsenten tulostaminen

Tietovirtojen ansiosta samoja aliohjelmia voidaan käyttää myös järjestelmässä, jossa varsinaista konsolitulostusta ei voi tehdä. Tällaisia ovat mm. graafiset käyttöliittymät.

Myös kerho-ohjelman on tarkoitus tulostaa jäsenien tietoja. Kun ongelmaa miettii tarkemmin, niin ei ole juuri mitään mieltä tehdä tulostusikkunaa pelkästään yhteen tarkoitukseen. Kannattaa siis luoda tietovirtojen avulla yleiskäyttöinen ikkuna.

HT3 KerhoGUI.java – tulostamisikkunan avaaminen

```
/**
 * Tulostaa kerhon tiedot TulostusDialogiin
 */
protected void tulosta() {
    TulostusDialog dialog = new TulostusDialog();
    kerhoswing.tulostaValitut(dialog.getTextArea());
    dialog.setVisible(true);
}
```

Ensimmäisellä rivillä luodaan uusi ikkuna, mutta vasta viimeisellä asetetaan se näkyväksi. Keskimmäinen rivi kutsutaan `tulostaValitut()`-metodia, jolle viedään parametrina juuri luodun ikkunan tekstikenttä.

Seuraavassa luodaan oma tietovirta `TextAreaOutputStream.java`, jolle viedään edelleen parametrina tulostusikkunan tekstikenttä. Nyt viimeisellä rivillä tulostaa kenttään, samalla tavoin kun tulostaisimme esimerkiksi `System.out` tietovirtaan.

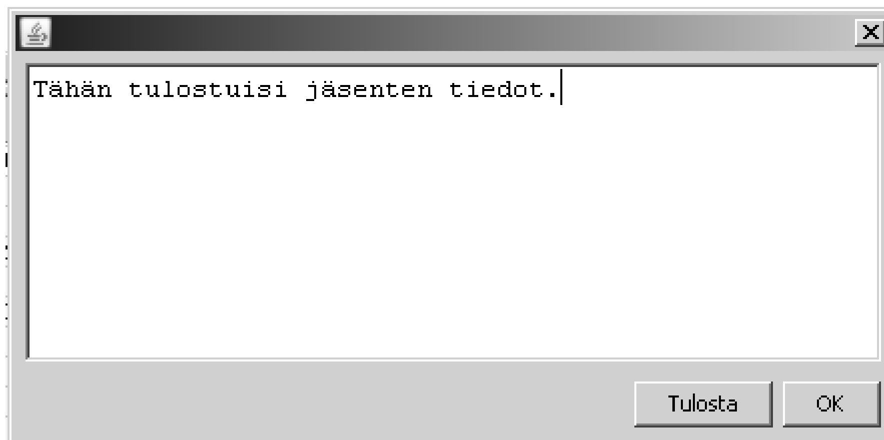
```
HT3 KerhoSwing.java – viedään tulostusikkunalle
```

```
/**
 * Tulostaa listassa olevat jäsenet tekstialueeseen
 * @param text alue johon tulostetaan
 */
public void tulostaValitut(JTextArea text) {
    PrintStream os = TextAreaOutputStream.getTextPrintStream(text);
    os.println("Tähän tulostuisi jäsenten tiedot");
}
```

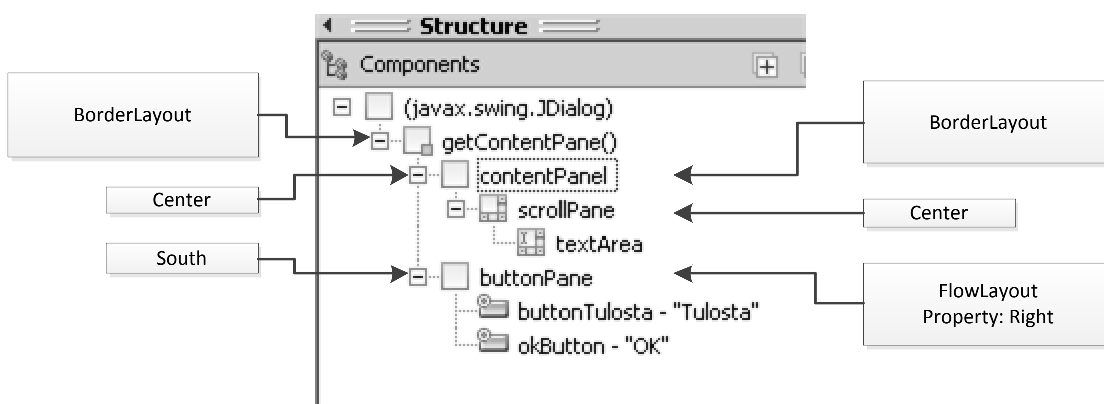
Toisaalta viimeisin vaihe on ehkä hieman turha. Nykyaikaisessa käyttöliittymäohjelmoinnissa tietovirrat eivät useinkaan ole käytännöllisin tapa siirtää tekstiä. Nyt kuitenkin Kerhon vanha versio on toiminut komentorivin päällä, joten siellä löytyy valmiit metodit tietovirtojen käyttöön. Olisi myös ollut mahdollista käyttää tekstikenttää suoraan.

```
    dialog.getTextArea.setText("Tähän tulostuisi jäsenten tiedot");
```

Itse tulostusikkunaan riittää yksinkertainen rakenne yhdellä tekstikentällä ja muutamalla napilla. Tekstikenttä kannattaa jättää käyttäjän editoitavaksi.



Kuva 7.3 Tulostusikkuna



Kuva 7.4 Tulostusikkunan asemointi Swing-kirjastolla

Nyt rakennettuna on yleiskäyttöinen tulostusikkuna. Jäljellä on vielä painikkeiden toiminnallisuuden ohjelmointi. *Ok*-napin painaminen sulkee ikkunan.

```
okButton.addActionListener(new ActionListener() {
   @Override
      public void actionPerformed(ActionEvent arg0) {
         dispose();
      }
});
```

Itse tulostamiseen `Swing`-kirjasto tarjoaa helposti käytettävät työkalut.

```
buttonTulosta.addActionListener(new ActionListener() {
   @Override
      public void actionPerformed(ActionEvent arg0) {
         try {
             getTextArea().print();
             } catch (PrinterException e) {
             // e.printStackTrace();
         }
      }
});
```

7.6 Parametrinvälitys

7.6.1 Useita parametreja

Suuressa osassa edellisissä esimerkeissämme meillä on ollut vain 0 tai yksi parametria välitettävänä aliohjelmaan. Käytännössä usein tarvitsemme useampia parametreja. 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 parametrina. 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 7.14 Päämenuun kerhon nimi

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

Tehtävä 7.15 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 7.16 Toisen asteen polynomi, `root_1`

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

Tehtävä 7.17 root_1 testaus

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

7.6.2 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:

muuttujat.funktio.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*).

7.6.3 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 7.18 Toisiaan kutsuvat aliohjelmat

Kirjoita yhden parametrin `mittakaava_muunnos` siten, että se kutsuu kahden parametrin `mittakaava_muunnos`ta.

7.6.4 Muuttujien lokaalisuus

Kukin aliohjelma muodostaa oman kokonaisuutensa. Edellä olleissa esimerkeissä aliohjelmat eivät tiedä ulkomaailmasta mitään muuta, kuin sen, mitä niille tuodaan parametreina 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:

muuttujat.nakyvyys.Lokaali.java - lokaalien muuttujien näkyvyys

```java
/**
 * 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 7.19 Eri nimet

Korjaa edellinen ohjelma siten, että kullakin erityyppisellä muuttujalla on eri nimi.

7.6.5 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 parametreina tunnit ja minuutit sekä niihin lisättävä minuuttimäärä. Jos ensimmäinen yritys olisi seuraava:

```
muuttujat.funktio.Aikalisa.java - yritys lisätä arvoja
```

```java
/**
 * 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 7.20 Muotoilu?

Kokeilepa laittaa ajaksi esim. 12:05. Mitä tulostuu? Miten vian voisi korjata?

Tehtävä 7.21 Tiedon lukeminen

Kirjoita aliohjelma `lue_kello`, joka kysyy ja lukee arvon kellonajalle, syöttö muodossa 12:15.

7.6.6 Aliohjelmien kirjoittaminen

Uuden aliohjelmien kirjoittaminen kannattaa aina aloittaa aliohjelmakutsun kirjoittamisesta vähintään testiohjelmaan. Näin voidaan suunnitella mitä parametreja ja missä järjestyksessä aliohjelmalle viedään. Näinhän teimme mittakaava–ohjelmassakin.

7.6.7 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:

muuttujat.funktio.Alisotku.java - parametrin välitystä

```java
/**
 * 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 */
/* 12 */ 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 muuttuja `c` (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
lause	a	* b	c	d	* a	b	d	* a	* b	c	SB N1	SB 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
08 b = d-1;						31							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 7.22 Muuttujien näkyvyys

Pöytätestaa seuraava ohjelma:

muuttujat.funktio.Alisotk2.java - parametrin välitystä

```java
/**
 * 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 */ }
}
```

7.7 Testipääohjelmat

Tähän mennessä esimerkit on testattu vain testipääohjelmilla. Tapa on toimiva kun toteutettavan ohjelmiston koko on pieni ja kaikki sen moduulit voi käydä suhteellisen pienellä työllä läpi.

Monisteessa aiemmin esiteltiin Alkulukuohjelma.

muuttujat.testaus.Alkuluku.java

```
    /**
     * 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 luvun jolla jaollinen tai 1 jos alkuluku
     */
     public static int onkoAlkuluku(int luku) {
        int jakaja = 2;
        int kasvatus = 1;
        if ( luku == 2 ) return 1;                  // 0

        do {
            int jakojaannos = luku % jakaja;
            if (jakojaannos == 0) return jakaja; // 1
            jakaja += kasvatus; // 2
            kasvatus = 2; // 3

        } while (jakaja < luku / 2);

        return 1;
    }
```

Testipääohjelmalla testaaminen metodilla, joka kertoo jos tulos ei ollutkaan odotettu.

```
    /**
     * Testipääohjelma onkoAlkuluku aliohjelman testaukseen
     * @param luku
     * @param odotettuVastaus true = alkuluku, false != alkuluku
     */
    public static int testiOnkoAlkuluku(int luku, int pieninJakaja) {
        int tulos = onkoAlkuluku(luku);
        if (pieninJakaja != tulos)
        {
            System.out.println("Luvun "+luku+" pienin jakaja on "+pieninJakaja+
                " mutta oli "+tulos);
            return 1;
```

```
        }
        return 0;
    }

    public static void main(String[] args) {

        testiOnkoAlkuluku(25, 5);
        testiOnkoAlkuluku(2, 1);
        testiOnkoAlkuluku(4, 2);
        testiOnkoAlkuluku(123, 3);
        testiOnkoAlkuluku(7, 1);

    }
```

Ohjelmiston koon kasvaminen asettaa kuitenkin uusia haasteita testaukselle.

- Mitä tehdä tilanteessa, jossa tehdään muutos funktioon, jota jo useat ohjelman osat käyttävät? Testipääohjelmilla joudutaan käymään käsin läpi jokainen moduuli, jotta voimme varmistua siitä ettei mikään mennyt rikki.
- Ohjelmakoodin sekaan kirjoitetut testit jäävät rasittamaan ohjelman tuotantoversiota.
- Koodin luettavuus heikkenee, kun testausrutiinit paisuttavat yksinkertaistakin varsin suureksi.

7.8 Yksikkötestaus

Ratkaisuksi edellä esiteltyihin ongelmiin on kehitetty menetelmä nimeltä yksikkötestaus (*Unit Testing*). Testipääohjelmien käyttö on jo periaatteessa eräänlaista yksikkötestausta, mutta nykyään termillä viitataan lähinnä ympäristöihin ja työkaluihin joilla testausta voidaan helpottaa ja automatisoida mahdollisimman pitkälle. Varsinaisten testitapausten (*test case*) lisäksi voidaan rakentaa myös koko projektin laajuisia testisarjoja (*test suite*), jotka voi ajaa vaikkapa automaattisesti yön aikana.

Nykyisin suosittu ohjelmointitekniikka on testivetoinen kehitys (*Test Driven Development, TDD*). Sen idea on kehittää koodi valmiiksi testattavaksi tehdä testit etukäteen, jonka jälkeen vasta aloitetaan itse ohjelmointi. Suunnitteluvaiheessa tapahtuva testaus selkeyttää ohjelman rakennetta ja toimintaa, jolloin se on helppo toteuttaa suoriutumaan halutuista selkeistä vaatimuksista. Kyse ei siis ole niinkään testauksesta, vaan ohjelman suunnittelusta, jonka sivutuotteena syntyvät kaikki testitapaukset.

Alkuun testien tekeminen saattaa vaikuttaa ylimääräiseltä työltä. Käytännössä kokeneinkin ohjelmoija joutuu kuitenkin joka tapauksessa kokeilemaan ohjelmansa toiminnallisuuden tavalla tai toisella. Kattavien testitapauksen kirjoittaminen etukäteen minimoi tällaiset turhat häiriöt ja mahdollistaa keskittymisen itse asiaan. Yksikkötestaukseen käytettävät kirjastot sisältävät myös valmiita funktioita tavallisimpiin testausrutiineihin, kuten esimerkin merkkijonojen vertailuun.

7.9 JUnit

Javassa yksikkötestaus tapahtuu yleensä *JUnit* –kirjastolla, jota kehitysympäristöt kuten *Eclipse* ja *NetBeans* tukevat suoraan. Testit on mahdollista ohjelmoida hieman testipääohjelmien tapaan suoraan testattavan asian yhteyteen, mutta tavallisesti testit kuitenkin erotetaan erillisiin paketteihin. *JUnit*-testit erotetaan muusta koodista `@test` –tagilla. Edellisen ohjelman testaus lyhentyy muotoon.

```
@Test
public void testOnkoAlkuluku19() {
  assertEquals("mahdollista virheenjäljitystä helpoittava viesti", 5,
               onkoAlkuluku(25));
  assertEquals(1, onkoAlkuluku(2));
  assertEquals(2, onkoAlkuluku(4));
  assertEquals(3, onkoAlkuluku(123));
  assertEquals(1, onkoAlkuluku(7));
}
```

Isommissa projekteissa *JUnit* testit toteutetaan vähintäänkin erilliseen pakettiin ja omiin luokkiinsa, jolloin niiden ylläpitäminen ja organisoiminen on helpompaa. Molemmat tavat ovat kuitenkin täysin oikein ja esimerkiksi kurssin demojen testaaminen onnistuukin varmasti helpoiten luokan yhteyteen kirjoittamalla. Isompia kuin yhden tai muutaman luokan ohjelmia testattaessa kannattaa kuitenkin eristää testaus itse ohjelmasta.

7.10 ComTest

Kurssilla on myös mahdollista käyttää yksikkötestaukseen *ComTest*-työkalua. *ComTest* on *Eclipse* -liitännäinen, joka generoi oman makrokielensä pohjalta täysin validia *Java/JUnit* –koodia. Makrokieli kirjoitetaan suoraan testattavan metodin tai luokan kommentteihin. Hieman samanlainen menetelmä on käytössä mm. *Python*-ohjelmointikielen *doctest* -kirjastossa.

ComTest pyrkii tekemään testien kirjoittamisen tekeminen vähemmän työlääksi ja helpommin ylläpidettäväksi. Ajatuksena on että kun testejä ei piiloteta eri tiedostoon, niin ohjelmoijalla on yksi askel vähemmän päästä niihin käsiksi. Lisäksi kun tämän lähestymistavan yhdistää yksinkertaiseen makrokieleen, niin samalla saadaan myös käyttöesimerkit dokumentaatioon. Yhdellä kertaa ohjelmoija voi siis periaatteessa suunnitella, testata ja dokumentoida koodinsa!

7.10.1 ComTestin makrokieli

Nyt `onkoAlkuluku`-aliohjelman testaus muuttuu muotoon

```
 * @example
 * <pre name="test">
 *   onkoAlkuluku(25)  === 5;
 *   onkoAlkuluku(2)   === 1;
 *   onkoAlkuluku(4)   === 2;
 *   onkoAlkuluku(123) === 3;
 *   onkoAlkuluku(7)   === 1;
 * </pre>
```

Testi alkaa kommenttilohkoon kirjoitetulla @example tagilla. Käytetyt `<pre>` tagit ovat normaalia html-koodia *JavaDoc* dokumentaatiota varten. Niiden sisälle voi kirjoittaa *ComTest* makrokieltä ja normaalia Javaa. Generoidusta *JUnit* testistä poistuvat tietysti rivin aloittavat kommentit, joten jos sellaiselle on tarvetta, on kommentoitava ”tuplasti”.

Valmiin testirungon voi tehdä kirjoittamalla kommenttilohkoon *comt* ja painamalla näppäinyhdistelmää *ctrl+välilyönti.*

7.10.2 ComTestin edistyneemmät ominaisuudet

Seuraava hieman monimutkaisempi ohjelma käyttää hyväkseen silmukoita ja taulukoita, joten jos ne ovat hyvässä muistissa Ohjelmointi 1 kurssilta, niin seuraava on hyvää luettavaa. Muussa tapauksessa kannattaa palata esimerkkiin myöhemmin.

Ohjelma sisältää algoritmin joka palauttaa indeksin ensimmäisen taulukon lukuun, jolla on eniten esiintymiä jälkimmäisessä taulukossa. Mikäli esiintymiä kahdella luvulla on sama määrä, niin palautetaan se joka esiintyi jonossa ensimmäisenä.

muuttujat.testaus.Esiintymat.java

```
/**
 * Ohjelma testauksen esittelyä varten
 * @author Santtu Viitanen
 * @version 1.0, 4.7.2011
 */
public class Esiintymat {

    /**
     * Palauttaa indeksin t1 taulun arvoon, jolla on eniten esiintymiä t2
     * taulussa. Mikäli kahdella luvulla on saman verran esiintymiä palautetaan
     * jonossa ensimmäinen arvo. Jos t1 on tyhjä palautetaan -1
     *
     * @param t1 taulukko, jonka indeksi palautetaan
     * @param t2 taulukko, johon verrataan
     * @return indeksi t1 taulun arvoon.
     */
    public static int enitenEsiintymia(int[] t1, int[] t2) {

        int paras = -1; // eniten esiintymiä omaavan luvun indeksi
        int parhaanEsiintymat = -1; //esiintymien määrät

        for (int i = 0; i < t1.length; i++) {
            int ehdokkaanEsiintymat = 0;

            for (int j = 0; j < t2.length; j++)
                if (t1[i] == t2[j])
                    ehdokkaanEsiintymat++;

            if (ehdokkaanEsiintymat > parhaanEsiintymat) {
                paras = i;
                parhaanEsiintymat = ehdokkaanEsiintymat;
            }
        }
        return paras;
    }
}
```

Testipääohjelman käyttö on muuttunut jo huomattavan työlääksi

```
    /**
     * @param args ei käytössä
     */
    public static void main(String[] args) {
        int[] t1 = { 1, 1, 3 };
        int[] t2 = { 1, 2 };
        int[] t3 = { 1, 3, 3, 4 };
        int[] t4 = {};
        enitenEsiintymiaTesti(t1, t2, 0);
        enitenEsiintymiaTesti(t1, t3, 2);
        enitenEsiintymiaTesti(t1, t4, 0);
        enitenEsiintymiaTesti(t4, t1, -1);
        enitenEsiintymiaTesti(t4, t4, -1);
    }

    /**
     * Testataan enitenEsiintymia-aliohjelmaa
     * @see #enitenEsiintymia(int[], int[]);
     * @param i oletettu tulos
```

```
     */
    private static int eniteniEsiintymiaTesti(int[] t1, int[] t2, int odotettu) {
        int tulos = enitenEsiintymia(t1, t2);
        if (tulos == odotettu)
            return 0;
        System.out.println("Tulokseksi tuli: " + tulos + " vaikka piti tulla: "
                + odotettu);
        return 1;
    }
```

ComTestilla funktion testaaminen muuttuu muotoon

```
    /**
     * Palauttaa indeksin t1 taulun arvoon, jolla on eniten esiintymiä t2
     * taulussa. Mikäli kahdella luvulla on saman verran esiintymiä palautetaan
     * jonossa ensimmäinen arvo. Jos t1 on tyhjä palautetaan -1
     *
     * @param t1 taulukko, jonka indeksi palautetaan
     * @param t2 taulukko, johon verrataan
     * @return indeksi t1 taulun arvoon.
     * @example
     * <pre name="test">
     * int[] t1 = {1,1,3};
     * int[] t2 = {1,2};
     * int[] t3 = {1,3,3,4};
     * int[] t4 = {};
     * enitenEsiintymia(t1,t2) === 0;
     * enitenEsiintymia(t1,t3) === 2;
     * enitenEsiintymia(t1,t4) === 0;
     * enitenEsiintymia(t4,t1) === -1;
     * enitenEsiintymia(t4,t4) === -1;
     * </pre>
     */
    public static int enitenEsiintymia(int[] t1, int[] t2) {
…
```

Testit sisältävät tyypillisesti paljon toistuvia rakenteita. Tämän vuoksi *ComTest* mahdollistaa myös taulukkomuotoisen testaamisen. Esimerkkimme oli varsin yksinkertainen, mutta monimutkaisemmissa toteutuksissa testejä voi olla kymmeniä, jolloin taulukkomuotoinen testaus vähentää kirjoittamista ja sillä saa aikaan havainnollisemman dokumentaation.

```
     * @example
     * <pre name="test">
     * int[] t1 = {1,1,3};
     * int[] t2 = {1,2};
     * int[] t3 = {1,3,3,4};
     * int[] t4 = {};
     * enitenEsiintymia($tparam1,$param2) === $tulos
     *  $param1 | $param2 | $tulos
     *  ----------------------
     *   t1     | t2     | 0
     *   t1     | t3     | 2
     *   t1     | t4     | 0
     *   t4     | t1     | -1
     *   t4     | t4     | 1
     * </pre>
```

Reaalilukujen vertaamiseen käytetään aaltoviivoja

```
 * @example
 * <pre name="test">
 * #TOLERANCE=0.01    // Määrää vertailun tarkkuuden
 * hypotenuusa(0,0) ~~~ 0.0;
 * hypotenuusa(0,1) ~~~ 1.0;
 * hypotenuusa(1,1) ~~~ 1.41;
 * hypotenuusa(1,2) ~~~ 2.24;
 * hypotenuusa(5,6) ~~~ 7.81;
 * </pre>
```

Muita käytettäviä makroja ovat esimerkiksi

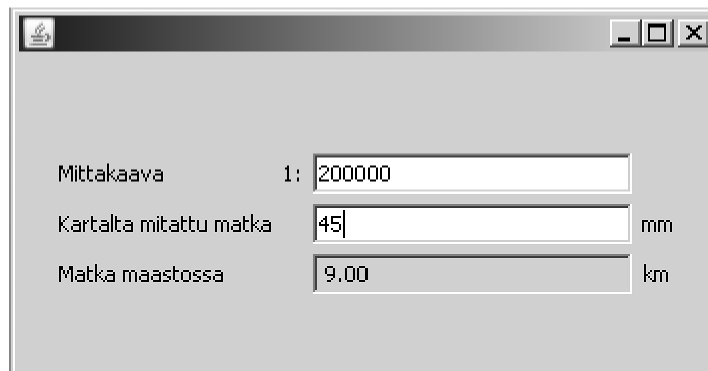
- `#CLASSIMPORT` – Lisää luokkia import lauseeseen
- `#STATICIMPORT` – kääntäjä löytää staattisen metodin ilman viittausta luokkaan (oletuksena päällä)

Tarkemmat ohjeet *ComTestin* asennukseen ja käyttöön löytyvät osoitteesta

```
https://trac.cc.jyu.fi/projects/comtest
```

7.11 Mittakaavaohjelma graafisena

Seuraavaksi on aika sitoa opittuja asioita yhteen. Nyt osataan jo toteuttaa yksinkertaisia graafisia ohjelmia joilla on toiminnallisuutta. Otetaan pohjaksi tässä luvussa esitelty komentoriviltä toimiva `Matka.java` ja toteutetaan se graafisena versiona (`Swing`).



Kuva 7.3 Mittakaavaohjelman käyttöliittymä

Uuden `JFrame` ikkunan voi luoda esimerkiksi *Eclipse*llä. Graafisessa editorissa kuvaa vastaavan käyttöliittymän saa suurin piirtein seuraavilla toimenpiteillä

- *Properties* ikkunasta pääikkunalle layout `GridBagLayout`
- `Label` ja `TextField` komponentit oikeisiin paikkoihinsa ja niille kuvaava nimeäminen (*Variable*). Esimerkissä kentille on käytetty `textMittakaava`, `textKartalta` ja `textMaastossa`.
- Labelien tekstit kuvaa vastaaviksi ja `textMittakaava` kentälle oletusteksti 200000

- Käyttöliittymä on tarkoitus toimia ilman erillistä *Laske*-nappia, joten graafisesta näkymästä asetetaan `textMittakaava` ja `textKartalla` kentille tapahtumaksi `KeyReleased`.

  Koska kahdella tekstikentällä on sama toiminnallisuus, niin laitetaan molemmista kutsu samaan `laske()`-metodiin.

muuttujat.graafinen.Mittakaava.java

```
/**
 *
 * @author Vesa Lappalainen @version 1.0, 27.1.2011
 * @author Santtu Viitanen @version 1.1, 03.08.2011
 */
public class Mittakaava extends JFrame {

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


    /**
     * Create the frame.
     */
    public Mittakaava() {
        addWindowListener(new WindowAdapter() {
 ...
        textKartalta.addKeyListener(new KeyAdapter() {
            @Override
            public void keyReleased(KeyEvent arg0) {
                laske();
            }
        });
        textMittakaava.addKeyListener(new KeyAdapter() {
            @Override
            public void keyReleased(KeyEvent arg0) {
                laske();
            }
        });
//Metodin loppuosa on suurimmaksi osaksi graafisessa
editorissa generoitua koodia
...
   }
```

Tarvitsemme ohjelmassa vakiot oletusmittakaavalle, sekä kertoimelle jolla teemme muutokset.

```
    public static final double MITTAKAAVA = 200000.0;
    public static final double MM_KM = 1000.0 * 1000.0;
```

Mittakaavamuunnoksia olemme laskeneet jo aiemminkin. Tähän ei liity mitään graafista. Muunnetaan kartalta millimetreinä mitattu matka kilometreiksi maastossa.

```
    public static double mittakaavamuunnos(double matka_mm, double mittakaava) {
        return mittakaava*matka_mm / MM_KM;
    }
```

Luvun hakemiseen tekstikentästä on myös hyvä olla yleiskäyttöinen metodi. Malliohjelmassa on käytetty `Ali.jar`-kirjastoa, josta löytyy valmis funktio reaalilukujen parsimiseen.

```
    public static double haeLuku(JTextField text,double oletus) {
        double luku = Mjonot.erotaDouble(text.getText(), oletus);
        return luku;
    }
```

Jäljellä onkin enää tekemiemme aliohjelmien ja graafisen käyttöliittymän komponenttien yhdistäminen toimivaksi kokonaisuudeksi.

```
    public static void laitaTulos(JTextField text, double luku) {
        String tulos = String.format("%5.2f",luku);
        tulos = tulos.replace(',', '.');
        text.setText(tulos);
    }

    private void laske() {
        double mittakaava = haeLuku(textMittakaava,MITTAKAAVA);
        double matka_mm = haeLuku(textKartalta, 0);
        double matka_km = mittakaavamuunnos(matka_mm,mittakaava);
        laitaTulos(textMaastossa,matka_km);
    }
}
```

Jäljellä onkin enää tekemiemme aliohjelmien ja graafisen käyttöliittymän komponenttien yhdistäminen toimivaksi kokonaisuudeksi.

Tehtävä 7.23 Lisää toiminnallisuutta ohjelmaan

Lisää ohjelmaan seuraava toiminnallisuus `selectAll()`-metodin avulla.
Enter `textMittakaava` ja `textKartalla` kentissä valitsee kyseisen kartan tekstit.
Enter `textMaastossa` kentässä valitsee kaikkien kenttien tekstit.

8. 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:

```
https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/oliot/
```

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.

8.1 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?

```
oliot.muut.Aikalis4.java - useita aika "muuttujia"
```

```java
... 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 8.1 Tulostus

Mitä ohjelma Aikalis4.java tulostaa?

8.2 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.

8.2.1 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 vastaava muuttuja - luokan ilmentymä - on sitten se kuuluisa **olio** (*object*)

8.2.2 Ensimmäinen olio-esimerkki

Muutetaanpa Aikalisa kunnon luokaksi ja olioksi:

oliot.aika.olio.Aika.java - kunnon olioksi

```
package oliot.aika.olio;
/**
 * Ensimmäinen kunnon olioesimerkki
 * @author  Vesa Lappalainen @version 1.0, 01.02.2003
 * @author Santtu Viitanen @version 1.1, 7.7.2011
 * @example
 * <pre name="testi">
 *  Aika a1 = new Aika(12,15);
 *  a1.lisaa(55);    a1.toString() === "13:10";
 * </pre>
 */
public class Aika {

 private int h=0, m=0;

  /**
   * Alustaa ajan
   * @param h tunnit
   * @param m minuutit
   * @example
   * <pre name="test">
   * new Aika(12,15).toString() === "12:15";
   * </pre>
   */
  public Aika(int h,int m) { // Muodostaja
    this.h = h;
    this.m = m;
  }

  /**
   * @return aika merkkijonona muodossa 12:05
   * @example
   * <pre name="test">
```

```
   * new Aika(12,15).toString() === "12:15";
   * new Aika(13,05).toString() === "13:05";
   * new Aika(7,0).toString()   === "07:00";
   * </pre>
   */
  public String toString() {
      return String.format("%02d:%02d",h, m);
  }

  /**
   * Lisää aikaan valitun minuuttimäärän
   * @param lisaMin lisättävä minuuttimäärä
   * @example
   * <pre name="test">
   *  Aika a1 = new Aika(13,16);
   *  a1.lisaa(27);    a1.toString() === "13:43";
   *  a1.lisaa(39);    a1.toString() === "14:22";
   * </pre>
   */
  public void lisaa(int lisaMin)  {
    int yht_min = h * 60 + m + lisaMin;
    h = yht_min / 60;
    m = yht_min % 60;
  }
}
```

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 tässä vaiheessa ehdi perehtyä kovinkaan syvällisesti.

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 lisaMin)  {
    int yht_min = h * 60 + m + lisaMin;
    h = yht_min / 60;
    m = yht_min % 60;
  }
...
```

Metodia kutsutaan ilmoittamalla olion nimi ja metodi, jota kutsutaan

```
//lisätään aikaan 55min
a1.lisaa(55);

//Merkkijonoon kello tallennettavaksi kutsutaan oliolta sen merkkijonoesitys
String kello = a1.toString();
```

Tällekin on keksitty oma nimi: välitetään oliolle viesti "lisaa" (*message passing).* Tässä kuitenkin jatkossa voi vielä lipsahtaa ja vahingossa sanomme kuitenkin, että kutsutaan metodia `lisaa`, vaikka ehkä pitäisi puhua viestin välittämisestä.

Suurin hyöty esimerkin toteuttamisesta olio-ohjelmoinnilla on sen tapa millä se hoitaa tulostuksen.

8.2.3 toString()

Metodilla `toString` voidaan määritellä miltä olio näyttää merkkijonona. Olion voi tulostaa tietovirtaan, jolloin se kutsuu automaattisesti `toString` metodia. Näin luokkaa ei sidota käytettäväksi missään tietyssä ympäristössä.

Huomattavaa on siis että `toString` palauttaa vain merkkijonon. On vasta käyttöliittymävaiheen ohjelmointia tietää mitä sillä halutaan tehdä.

```
...
  public String toString() {
      return String.format("%02d:%02d",h, m);
  }
...
```

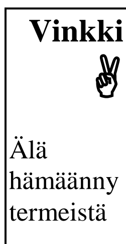
Komentoriviohjelma voisi käyttää Aikaa esimerkiksi seuraavasti tulostamalla sen (`System.out`) tietovirtaan.

```
Aika aika = new Aika(13,37);
System.out.println("Kello on "+aika);
```

Ohjelma ei myöskään kaadu, vaikka `toString()`–metodia ei olisikaan erikseen määritelty. Tarkalleen ottaen tällöin tietovirtaan tulostuu olion hajautusarvo heksadesimaalijärjestelmän lukuna.

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
```

8.2.4 Luokka (*class*) ja olio (*object*)

Luokka on tavallaan "piparkakkumuotti" kaikille samankaltaisille "olioille". Luokalla ei sinänsä tee mitään (jos siinä ei 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
```

8.2.5 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 8.1 Suojaustasot

Kirjoitamme testiluokan havainnollistamaan tätä toiminnallisuutta:

oliot.aika.olio.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);    System.out.println(a1);
    a2.lisaa(27);    System.out.println(a1);
    a3.lisaa(39);    System.out.println(a1);
  }
}
```

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 alkavatkin olio-ohjelmoinnin hienoudet! Aloittelijasta saattaa tuntua että mitä turhaa tehdään asioista monimutkaisempaa kuin 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?

8.2.6 Muodostajat (*constructor*)

*Java*n (kuten myös C# ja C++) olioilla on yksi erityinen metodi: **muodostaja** (konstruktori, rakentaja, *constructor*), jota kutsutaan olion 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.

oliot.aika.olio.Aika.java - muodostaja 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);
```

8.2.7 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 jokin muu muodostaja, ei oletusmuodostaja enää tulekaan automaattisesti.

Meidän pitäisi päättää nyt paljonko kellomme on, jos sitä ei erikseen ilmoiteta. Olkoon kello vaikka 00:00, eli keskiyö. Esittelemme oletusmuodostajan

oliot.aika.muodostaja.Aika.java - lisätään oletusmuodostaja

```
/** … */
public class Aika {

 private int h=0, m=0;

  /**
   * Alustaa ajan muotoon 00:00
   * @example
   * <pre name="test">
   * new Aika().toString() === "00:00";
   * </pre>
   */
  public Aika() { // Oletusmuodostaja
      h = 0; m = 0;
    }


  /** … */
  public Aika(int h,int m) { // Muodostaja
    this.h = h;
    this.m = m;
  }


  /** … */
  public String toString() {
      return String.format("%02d:%02d",h, m);
  }

  /** … */
  public void lisaa(int lisaMin)  {
    int yht_min = h * 60 + m + lisaMin;
    h = yht_min / 60;
    m = yht_min % 60;
  }
}
```

Tässä tapauksessa oletusmuodostajaksi olisi kelvannut myös tyhjä lohko. Miksi?

```
  public Aika() {  }
```

8.2.8 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);
```

Koska `aseta`-metodi hoitaa sisäisen tilan valvonnan, sitä kannattaa hyödyntää myös muodostajissa. Näin olion tilaa muutetaan vain muutamassa metodissa, jotka voidaan ohjelmoida niin huolella, ettei mitään yllätyksiä pääse koskaan tapahtumaan.

```
    public Aika() { aseta(0, 0);}
    public Aika(int h) { aseta(h, 0);}
    public Aika(int h, int m) { aseta(h, m); }
```

Nyt pitää kuitenkin päättää mitä tarkoittaa laiton asetus! Mikäli barbaarimaisesti sovimme että minuuteissa yli 59 arvot ovat aina 59 ja alle 0:n arvot ovat aina 0, voisi `aseta`-metodi olla kirjoitettu seuraavasti:

```
  public final void aseta(int ih,int im) {
    h = ih; m = im;
    if ( m > 59 ) m = 59;
    if ( m <  0 ) m =  0;
  }
```

Elegantimpi ratkaisu ongelmaan on siirtää ylimääräiset minuutit tunteihin ja ylimääräiset tunnit vuorokausiin. Vielä emme kuitenkaan ole selvillä vesillä, koska kokonaislukumuuttujiin voi tietysti syöttää myös negatiivisia lukuja. Ratkaisuna otamme puuttuvat minuutit tunneista ja tunnit vuorokausista.

```
   /**
    * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
    * @param h asetettava tuntimäärä
    * @param m asetettava minuuttimäärä
    * @return montako vuorokautta jäi yli
    * <pre name="test">
    *  Aika a1 = new Aika();
    *  a1.aseta(12,15); a1.toString() === "12:15";
    *  a1.aseta(15,45); a1.toString() === "15:45";
    *  a1.aseta(-49,-125); a1.toString() === "20:55";
    * </pre>
    */
   public final int aseta(int h, int m) {

        h += m / 60; // liiat minuutit tunteihin
        m %= 60; // minuutit väille 0-59
        int vrk = h / 24; // liiat tunnit vuorokausiin
        h %= 24; // tunnit välille 0-23

        if (m<0) { m += 60; h--; }  //negatiiviset arvot
        if (h<0) { h += 24; vrk--; }

        this.h = h; // asetetaan lasketut arvot attribuutteeihin
        this.m = m;
        return vrk; // motako vuorokautta jäi yli
    }
```

Algoritmi on jo huomattavan monimutkainen, joten kirjoittamalla kattavat testit etukäteen säästyy paljolta vaivalta.

oliot.aika.valvonta.Aika.java - sisäinen tilan valvonta asetuksessa

```
/**
 * Sisäinen tilanvalvonta Aikaan.
 * @author  Vesa Lappalainen @version 1.0, 01.02.2003
 * @author Santtu Viitanen @version 1.1, 7.7.2011
 * @example
 * <pre name="test">
 *  Aika a1 = new Aika(12,15);
 *  a1.lisaa(55);  a1.toString() === "13:10";
 *  a1.aseta(12,15); a1.toString() === "12:15";
 * </pre>
 */
public class Aika {

    private int h = 0, m = 0;

    /** … */
    public Aika() { // Oletusmuodostaja
        aseta(0, 0);
    }

    /** … */
    public Aika(int h) {
        aseta(h, 0);
    }

    /** … */
    public Aika(int h, int m) { // Muodostaja
        aseta(h, m);
    }

   /** … */
    public final int aseta(int h, int m) {

        h += m / 60; // liiat minuutit tunteihin
        m %= 60; // minuutit väille 0-59
        int vrk = h / 24; // liiat tunnit vuorokausiin
        h %= 24; // tunnit välille 0-23

        if (m<0) { m += 60; h--; }  //negatiiviset arvot
        if (h<0) { h += 24; vrk--; }

        this.h = h; // asetetaan lasketut arvot attribuutteeihin
        this.m = m;
        return vrk; // motako vuorokautta jäi yli
    }

    /** … */
    public void lisaa(int lisaMin) {
        aseta(h, m + lisaMin);
    }
    /** … */
    public String toString() { … }
}
```

Tarkkaavainen lukija luultavammin ihmettelee tässä vaiheessa `final`- määrettä `aseta`-metodin edellä. Koodi kääntyy myös ilman, mutta sen käyttö on suositeltavaa kutsuttaessa metodia suoraan muodostajalta. Tässä vaiheessa syytä on työlästä selittää, mutta siihen palataan perintää koskevassa luvussa.

Tehtävä 8.2 Negatiivinen minuuttiasetus

Mitä ohjelma Aika.java tulostaisi? Miksi ohjelma toimisi halutulla tavalla?

Tehtävä 8.3 Päivämääräluokka

Mieti miten päivämäärää kuvaava luokka kuuluisi toteuttaa ja mitä tulisi ottaa huomioon.

Tehtävä 8.4 Päivämääräluokan toteutus

Esittele luokka, jolla kuvataan päivämäärä. Kirjoita ainakin sopiva muodostaja ja metodi `toString`, joka palauttaa päivämäärän merkkijonona.

8.2.9 Metodien kuormittaminen (lisämäärittely, *overloading*)

Edellisessä esimerkissä oli kolme samannimistä metodia `Aika`. Kussakin oli eri määrä parametreja. Tätä sanotaan metodin kuormittamiseksi, eli mahdollisuudeksi määritellä lisää merkityksiä (eli kuormaa, eng. *overloading*) metodin nimelle. Varsinainen kutsuttava metodi tunnistetaan nimen ja parametrilistassa olevien lausekkeiden avulla. Metodin nimi koostuukin tavallaan nimen ja parametrilistan yhdisteestä. Siten esimerkiksi

```
String m = "kissa";
String s;

s = m.substring(1);             // s === "issa"
s = m.substring(1,3);           // s === "is"
```

kumpikin `substring`-kutsu kutsuu eri metodia. Metodien kuormitus onkin varsin mukava lisä ohjelmointiin, se ei kuitenkaan ole varsinaisia olio-ohjelmoinnin piirteitä.

Kuormitetussa metodeissa ero on oltava parametreissa, pelkkä ero metodin paluuarvossa ei riitä erottelemaan mitä metodia tarkoitetaan. Huomattakoon että kuormitus ei ole riippuvainen pelkästään parametrien määrästä, vaan myös tyypillä on merkitystä.

Tehtävä 8.5 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 8.6 Lisäys yhdellä

Tee vielä uusi `lisaa` metodi, jota voidaan kutsua `a1.lisaa();` jolloin metodi lisää aikaa yhdellä minuutilla.

Tehtävä 8.7 Vain tuntien asettaminen

Kirjoita vielä yksi `lisaa`-metodi, jolla voidaan asettaa pelkät tunnit.

8.2.10 `this`-osoitin

Jos verrataan aliohjelmaa

oliot.aika.metodi.Aika.java - aliohjelma vastaan metodi

```
/**
 * …
 * <pre name="test">
 *  Aika a1 = new Aika();
 *  a1.aseta(a1, 12,15); a1.toString() === "12:15";
 *  a1.aseta(a1, 15,45); a1.toString() === "15:45";
 *  a1.aseta(a1, -49,-125); a1.toString() === "20:55";
 * </pre>
 */
 public static int aseta(Aika aika, int h, int m) {
```

```
        h += m / 60; // liiat minuutit tunteihin
        m %= 60; // minuutit väille 0-59
        int vrk = h / 24; // liiat tunnit vuorokausiin
        h %= 24; // tunnit välille 0-23

        if (m<0) { m += 60; h--; }  //negatiiviset arvot
        if (h<0) { h += 24; vrk--; }

        aika.h = h; // asetetaan lasketut arvot attribuutteeihin
        aika.m = m;
        return vrk; // motako vuorokautta jäi yli
    }
...
```

ja metodia

```
 /**
  * …
  * <pre name="test">
  *  Aika a1 = new Aika();
  *  a1.aseta(12,15); a1.toString() === "12:15";
  *  a1.aseta(15,45); a1.toString() === "15:45";
  *  a1.aseta(-49,-125); a1.toString() === "20:55";
  * </pre>
  */
  public final int aseta(int h, int m) {

        h += m / 60; // liiat minuutit tunteihin
        m %= 60; // minuutit väille 0-59
        int vrk = h / 24; // liiat tunnit vuorokausiin
        h %= 24; // tunnit välille 0-23

        if (m<0) { m += 60; h--; }  //negatiiviset arvot
        if (h<0) { h += 24; vrk--; }

        this.h = h; // asetetaan lasketut arvot attribuutteeihin
        this.m = m;
        return vrk; // motako vuorokautta jäi yli
    } }
...
```

niin helposti näyttää, että ensin mainitussa funktiossa on enemmän parametreja. Tosiasiassa kummassakin niitä on täsmälleen sama määrä. Nimittäin jokaisen ei-staattisen metodin ensimmäisenä näkymättömänä parametrina tulee aina itse luokan osoite, `this`. Voitaisiinkin kuvitella, että metodi on toteutettu:

```
"public final int aseta(Aika this, int h, int m)  {" // Näin EI SAA KIRJOITTAA!!!
{
...
"a1.aseta(a1,13,37)";
```

Oikeasti `this` -*viitettä ei saa esitellä*, vaan se on ilman muuta mukana parametreissa sekä esittelyssä että kutsussa. Mutta voimme todellakin kirjoittaa:

```
public final int aseta(int h, int m)   {
{
  …
    this.h = h; // asetetaan lasketut arvot attribuutteeihin
    this.m = m;
}
```

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, jos näin oltaisiin haluttu tehdä edellisessä esimerkissä oltaisiin voitu kirjoittaa

```
public final int aseta(int hours, int minutes)  {
{
  …
    h = hours; // asetetaan lasketut arvot attribuutteeihin
    m = minutes;
}
```

8.3 Perintä

8.3.1 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.

8.3.2 Alkuperäisen luokan muuttaminen

Läheskään aina ei voi täysin välttää sitä, ettei 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 aina mahdollista, vaan silloin täytyy turvautua muihin (parempiin) tapoihin.

Tehtävä 8.8 Luokan muuttaminen

Muuta ohjelmaa Aika.java siten, että ajassa on mukana myös sekunnit. Kuitenkin niin, että alkuperäinen testiohjelma säilyy sellaisenaan toimivana. Voit lisätä testiohjelmaan uusia rivejä sekuntien testaamiseksi.

Tehtävä 8.9 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.

8.3.3 Saantimetodit

Tulee tietysti tilanteita, joissa luokan ulkopuolinen haluaa päästä käsiksi sisäisiin tietoihin, kuten esimerkkiluokkiemme aikoihin. Onneksi tähän asti esimerkkiluokkien rakenne on tehty sen verran älykkäästi, ettei alemman tason komponentteihin ole sotkettu käyttöliittymän toimintoja.

Aika olisi mahdollista saada selville parsimalla `toString` metodeista halutut palat, mutta tämä on tietysti hyvin epäkäytännöllistä ja aiheuttaa ylimääräistä työtä. Oikea tapa on kirjoittaa saantimetodi kullekin attribuutille, joka perustellusti voidaan katsoa tarpeelliseksi julkaista jollekin ulkopuoliselle.

Esimerkeistä koostaminen ja perintä tulevat käyttämään paketista `oliot.aika.metodi` löytyvää `Aika`-luokkaa. Molempien tapojen kannalta on käytännöllistä, mikäli ne pystyvät lukemaan alkuperäisen luokan attribuutteja. Lisätään siis alkuperäiseen luokkaan seuraavat minuutit tai tunnit palauttavat metodit.

```
…
  public int getH() { return h; }
  public int getM() { return m; }
…
```

Nyt voitaisiin esimerkiksi kutsua:

```
System.out.println("Tunnit = " + a1.getH());
```

Näkyvyysmääreet `public` tai `protected` antaisivat toki perivälle luokalle oikeuden muuttaa attribuutteja. Mikä etuja saavutetaan siis saantimetodeilla 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.

8.3.4 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 kannattaa aina vakavasti harkita.

Nyt voimme kirjoittaa uuden luokan, joka koostetaan luokasta `Aika` ja sekunneista:

oliot.aika.koostaminen.AikaSek.java - laajentaminen koostamalla

```
import oliot.aika.metodi.Aika;

/**
 * Luokan laajentaminen koostamalla
 * …
 */
public class AikaSek {

    private Aika hm = new Aika();
    private int s = 0;

    /**
     * Alustaa ajan sekuntien tarkkuudella
     * @param h tunnit
```

```
     * @param m minuutit
     * @param s sekunnit
     * @example
     * <pre name="test">
     * new AikaSek(11,12,13).toString() === "11:12:13";
     * </pre>
     */
    public AikaSek(int h, int m, int s) { // Muodostaja
        aseta(h, m, s);
    }

    /**
     * Alustaa ajan minuuttien tarkkuudella
     * @param h tunnit
     * @param m minuutit
     */
    public AikaSek(int h, int m) { aseta(h, m, 0); }

    /**
     * alustaa ajan tuntien tarkkuudella
     * @param h tunnit
     */
    public AikaSek(int h) { aseta(h, 0, 0); }
    public AikaSek() { }

    /**
     * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
     * @param h asetettavat tunnit
     * @param m asetettavat minuutit
     * @param s asetettavat sekunnit
     * @return yli- tai alijääneet vuorokaudet
     * @example
     * <pre name="test">
     * AikaSek aika = new AikaSek(1,2,3);
     * aika.toString() === "01:02:03";
     * aika.aseta(-1,32,42) === -1;
     * aika.toString() === "23:32:42";
     * </pre>
     */
    public final int aseta(int h, int m, int s) {
        m += s / 60; // liiat sekunnit minuutteihin
        s %= 60; // sekunnit välille -59 - 59
        if (s < 0) {
            s += 60;
            m--;
        }
        int vrk = hm.aseta(h, m);
        this.s = s;
        return vrk;
    }

    public final int aseta(int h, int m) { return aseta(h, m, 0); }
    public final int aseta(int h) { return aseta(h, 0); }

    /**
     * Lisää aikaa sekuntien tarkkuudella
     * …
     */
    public void lisaa(int lisaMin, int lisaSek) {
        aseta(hm.getH(), hm.getM() + lisaMin, this.s + lisaSek);
    }
    /**
     * Lisää aikaa minuuttien tarkkuudella
     * …
     */
    public void lisaa(int lisaMin) {
        lisaa(lisaMin, 0);
    }

    /** … */
    public String toString() {
        return hm.toString() + String.format(":%02d", s);
    }
}
```

Luokassa on niin vähän ominaisuuksia, että uudessa luokassamme olemme joutuneet 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.

8.3.5 Perintä, inheritance

Viimeisenä vaihtoehtona tarkastelemme perintää (*inheritance*). Valinta koostamisen ja perinnän välillä on vaikea. Aina edes olioasiantuntijat eivät osaa sanoa yleispätevästi kumpi on parempi. Nyrkkisääntönä voisi pitää seuraavaa *is-a* -sääntöä:

```
Jos voi sanoa että LuokkaA on LuokkaB (is-a), niin peritään. Tällöin voi puhua
isä- ja lapsiluokasta.
Jos sanotaan että luokassa A on (has-a) toinen luokka B, niin koostetaan. Nyt
puhutaan kooste- ja osaluokasta.
```

Esimerkiksi ei voi sanoa että ”auto on moottori”, vaan ”autossa on moottori”. Tällöin autoa ei siis peritä moottorista, vaan auto koostetaan osista, joista yksi on moottori.

Kokeillaanpa ajan kanssa: "*luokka jossa on aika sekunteina*" on "*aika-luokka*". Kuulostaa hyvältä. Siis perimään:

oliot.aika.perinta.AikaSek.java - laajentaminen perimällä

```
import oliot.aika.metodi.Aika;

/**
 * Luokan laajentaminen perimällä
 * @author  Vesa Lappalainen @version 1.0, 01.02.2003
 * @author Santtu Viitanen @version 1.1, 7.7.2011
 * @example
 * <pre name="test">
 * AikaSek a1 = new AikaSek(14,55,45);
 * a1.toString() === "14:55:45";
 * a1.lisaa(3,30); a1.toString() === "14:59:15";
 * AikaSek a2 = new AikaSek(); a2.toString() === "00:00:00";
 * AikaSek a3 = new AikaSek(12); a3.toString() === "12:00:00";
 * AikaSek a4 = new AikaSek(12,15); a4.toString() === "12:15:00";
 * </pre>
 */
public class AikaSek extends Aika {

    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
     * @return yli- tai alijääneet vuorokaudet
     * <pre name="test">
     * AikaSek aika = new AikaSek(1,2,3);
     * aika.toString() === "01:02:03";
     * aika.aseta(-1,32,42) === -1;
     * aika.toString() === "23:32:42";
     * </pre>
     */
    public final int aseta(int h, int m, int s) {
        m += s / 60; // liiat sekunnit minuutteihin
        s %= 60; // sekunnit välille -59 - 59
        if (s < 0) {
            s += 60;
            m--;
        }
        int vrk = aseta(h, m);
        this.s = s;
        return vrk;
    }
```

```
    public AikaSek() { }

    /** … */
    public AikaSek(int h, int m, int s) { aseta(h, m, s); }

    /**…  */
    public AikaSek(int h, int m) { aseta(h, m, 0); }

    /** … */
    public AikaSek(int h) { aseta(h, 0, 0); }

    /** … */
    public String toString() {
        return super.toString() + String.format(":%02d", s);
    }

    /** … */
    public void lisaa(int lisaMin, int lisaSek) {
        aseta(this.getH(), this.getM() + lisaMin, this.s + lisaSek);
    }
}
```

Tässä tapauksessa kirjoittamisen vaiva oli melkein sama kuin koostamisessakin. Niitä `aseta`, `lisaa` ja `toString` metodeja, jotka löytyivät jo kantaluokasta `Aika`, ei tarvinnut kirjoittaa. Itse asiassa myös `Aika` on perinyt `toString` metodinsa `Object` luokasta, josta on kaikkien Javan luokkien kantaluokka. `Object`-luokka sisältää jo valmiina muutamia olio-ohjelmoinnin kannalta tärkeitä yleiskäyttöisiä metodeja.

Muodostajasta pitää kirjoittaa kaikki eri versiot, sillä muodostaja ei valitettavasti 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:

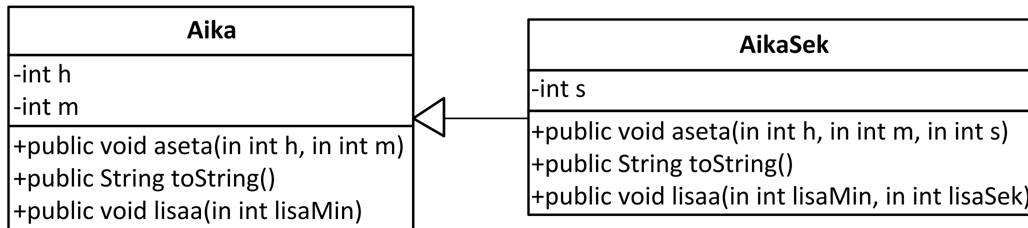
```
public class AikaSek extends Aika {
```

Jos täytyy viitata yliluokan metodeihin, joille on kirjoitettu aliluokassa oma määrittely, käytetään yliluokan viitettä `super`

```
    super.lisaa(lisaMin+s/60);
```

Yliluokan viitettä ei tarvita, mikäli samannimistä metodia ei ole aliluokassa.

Perintää on helppo kuvata UML-luokkakaavioilla. Nuolen suuntaa käytetään tarkoittamaan että *peritään jostakin*.



Kuva 8.2 Aika perinnällä

8.3.6 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 muuten vaikea saavuttaa perinteisellä ohjelmoinnilla: polymorfismi (*polymorphism*) eli monimuotoisuus.

Lisätäänpä vielä *ComTest*-ohjelman loppuun:

```
* AikaSek a1 = new AikaSek(14,55,45);
* ...
* //ESIMERKKI POLYMORFISMISTA
* #import oliot.aika.metodi.Aika;
* Aika aika = new Aika(11,12); aika.toString() === "11:12";
* aika = a1; aika.toString() === "14:59:15";
* aika.lisaa(20); aika.toString() === "15:19:15";
```

Mistä tässä oli kyse? Viite `aika` on monimuotoinen, eli sama osoitin voi osoittaa useaan erityyppiseen luokkaan. Tämä on mahdollista, jos luokat ovat samasta perimähierarkiasta kuten tässä tapauksessa ja viite on tyypiltään näiden yhteisen kantaluokan olion viite.

8.3.7 Myöhäinen sidonta

Miksi edellä jälkimmäisessä `aika.toString()` kutsussa kutsuttiin luokan `AikaSek` tulosta metodia eikä `Aika` -luokan metodia `toString`.

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*). 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.

Aliluokaan voidaan kirjoittaa uusi versio yliluokan vastaavasta metodista. Tästä käytetään termiä uudelleenmäärittäminen (korvaaminen, syrjäyttäminen, *overriding*). Usein korvatussa metodissa kutsutaan myös yliluokan alkuperäistä metodia.

Tehtävä 8.10 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 8.11 Ei turhaa lisaa-kutsua

Mitä tarvitsee muuttaa jotta viimeinen `lisaa`-kutsu saadaan pois?

8.4 Kapselointi

Termi kapselointi liittyy kiinteästi olio-ohjelmointiin. Sen voi ymmärtää toteutuksena joka kokoaa toimintoja yhdeksi kokonaisuudeksi, kuten luokaksi tai paketiksi. Muodostunutta kokonaisuutta voidaan hallita helpommin. Kapseloimalla voidaan piilottaa rakenteen sisäisiä ominaisuuksia, joihin ei haluta pääsyä ulkopuolelta. Sisäistä rakennetta voidaan myös parantaa muuttamatta ohjelman toimintaa ulospäin.

Tehtävä 8.12 Saantimetodi sekunneille

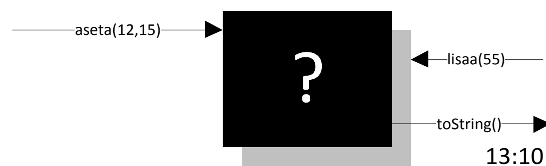
Täydennä AikaSek.java:hen em. saantimetodit ja lisäksi `getS()` aliluokkaan `AikaSek`.

Tehtävä 8.13 Saantimetodien käyttäminen

Muuta vielä edellisessä tehtävässä jokainen mahdollinen viittaus luokan sisälläkin saantimetodeja käyttäväksi suoran attribuuttiviittauksen sijasta.

8.4.1 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 8.3 Musta laatikko

Tämä luokan sisustan piilottaminen antaa meille mahdollisuuden toteuttaa luokka oleellisesti eri tavalla. Voimme esimerkiksi toteuttaa ajan minuutteina vuorokauden alusta laskien:

oliot.aika.sisesitys.Aika.java - sisäinen toteutus minuutteina

```
/**
 * Vaihdettu sisäinen esitystapa
 * …
 */
public class Aika {

  private int yhtMin;
```

```
  /**
   * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
   * Ei osaa negatiivisia aikoja
   * @param h asetettavat tunnit
   * @param m asetettavat minuutit
   * @return yli- tai alijääneet vuorokaudet
   * <pre name="test">
   *  Aika a1 = new Aika();
   *  a1.aseta(12,15); a1.toString() === "12:15";
   *  a1.aseta(15,45); a1.toString() === "15:45";
   * </pre>
   */
  public final int aseta(int h,int m) {
    yhtMin = 60*h + m;
    return yhtMin / 60 / 24; //vrk
  }

  public Aika() { aseta(0,0); }
  public Aika(int h) { aseta(h,0); }

  /**
   * Alustaa ajan
   * @param h tunnit
   * @param m minuutit
   */
  public Aika(int h,int m) {
    aseta(h,m);
  }

  /** … */
  public String toString() {
      return String.format("%02d:%02d",getH(), getM());
  }

  /**
   * Lisää aikaan halutun minuuttimäärän
   * @param lisaMin lisättävä minuuttimäärä
   * @example
   * <pre name="test">
   * Aika aika = new Aika(11,12);
   * aika.lisaa(66); aika.toString() === "12:18";
   * </pre>
   */
  public void lisaa(int lisaMin)  {
      aseta(0,yhtMin+lisaMin);
  }

  /** … */
  public static void lisaa(Aika aika,int lisaMin) { aika.lisaa(lisaMin);  }


  /**
   * @return aika tunteina
   * @example
   * <pre name="test">
   * new Aika(11,12).getH() === 11;
   * </pre>
   */
  public int getH() { return yhtMin / 60; }

  /**
   * @return aika minuutteina
   * <pre name="test">
   * new Aika(11,12).getM() === 12;
   * </pre>
   */
  public int getM() { return yhtMin % 60; }
}
```

Tehtävä 8.14 `minuutteina()`

Lisää luokkiin `oliot.metodit.Aika` ja `oliot.perinta.AikaSek` sisäisiin toteutustapoihin saantimetodi `getMinuutteina`, joka palauttaa kellonajan vuorokauden alusta minuutteina laskettuna.

8.5 Rajapinta ja monimuotoisuus

Yksi perinnän tärkeimmistä ominaisuuksista on mahdollisuus monimuotoisuuteen, polymorfismiin. Esimerkiksi

oliot.aika.perinta.AikaSek.java - esimerkki polymorfisesta taulukosta

```
Aika a1 = new Aika();
Aika a2 = new Aika(13);
Aika a3 = new Aika(14,175);
AikaSek a4 = new AikaSek(14,55,45);

Aika aika = a1;  aika = a4;
aika = new AikaSek(1,95,70);

Aika ajat[] = new Aika[5];
ajat[0] = a1; ajat[1] = a2; ajat[2] = a3; ajat[3] = a4;
ajat[4] = new AikaSek(23,59,59);

for (int i=0; i < ajat.length; i++ ) {
  System.out.print(ajat[i]); System.out.print(" +" + i + " => ");
  ajat[i].lisaa(i); System.out.println(ajat[i]);
}
```

Taulukko `ajat` koostuu viitteistä `Aika`-luokan olioihin. Myös `AikaSek` toteuttaa saman rajapinnan, koska se on perittyä samasta luokasta. Siksi taulukkoon voi laittaa mitä tahansa `Aika`-luokan jälkeläisluokankin olioita.

Esimerkissä `AikaSek` luokka koostettiin sekunneista ja luokan `Aika` oliosta. Nyt valitettavasti vain polymorfismi ei toimi, eli `AikaSek` ja `Aika` 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:

oliot.aika.koostaminen.AikaSek.java - kömpelö esimerkki polymorfisesta taulukosta

```
Aika a1 = new Aika();
Aika a2 = new Aika(13);
Aika a3 = new Aika(14,175);
AikaSek a4 = new AikaSek(14,55,45);

Object ajat[] = new Object[5];
ajat[0] = a1; ajat[1] = a2; ajat[2] = a3; ajat[3] = a4;
ajat[4] = new AikaSek(23,59,59);

for (int i=0; i < ajat.length; i++ ) {
  if ( ajat[i] instanceof Aika ) {
    Aika aika = (Aika)ajat[i];  // pakotettu tyypin muunnos
    System.out.print(aika + " +" + i + " => ");
    aika.lisaa(i); System.out.println(aika);
  }
  if ( ajat[i] instanceof AikaSek ) {
    AikaSek aika = (AikaSek)ajat[i];  // pakotettu tyypin muunnos
    System.out.print(aika + " +" + i + " => ");
    aika.lisaa(i); System.out.println(aika);
  }
}
```

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:

oliot.aika.rajapinta.AikaRajapinta.java - malli kaikkien Aika-luokkien rajapinnasta

```
public interface AikaRajapinta {

  /**
   * Asettaa uuden ajan ja pitää huolen että aika on aina oikeaa muotoa.
   * @param h asetettavat tunnit
   * @param m asetettavat minuutit
   */
  public final int aseta(int h,int m);

  /**
   * Palauttaa ajan merkkijonona muodossa 15:05
   */
  public String toString();

  /**
   * Lisää aikaan valitun minuuttimäärän
   * @param lisa_min lisättävä minuuttimäärä
   */
  void lisaa(int lisa_min);

  int getH();
  int getM();
}
```

Seuraavaksi kaikki luokat, jotka halutaan kuuluvan "samaan kategoriaan", ilmoitetaan toteuttavan tämän rajapinnan:

oliot.aika.rajapinta.Aika.java - luokka joka toteuttaa rajapinnan

```
public class Aika implements AikaRajapinta {

  private int h, m;

  public final int aseta(int h,int m) {
      h += m / 60; // liiat minuutit tunteihin
      m %= 60; // minuutit väille -59 - 59
      int vrk = h / 24; // liiat tunnit vuorokausiin
      h %= 24; // tunnit välille 0-23

      if (m<0) { m += 60; h--; }  //negatiiviset arvot
      if (h<0) { h += 24; vrk--; }

      this.h = h; // asetetaan lasketut arvot attribuutteeihin
      this.m = m;
      return vrk; // motako vuorokautta jäi yli tai ali
  }

  public Aika() { this.h = 0; this.m = 0; }

  public Aika(int h) {
    aseta(h,0);
  }

  public Aika(int h,int m) { // Muodostaja
    aseta(h,m);
  }

  public String toString() {
      return String.format("%02d:%02d",getH(), getM());
  }
```

```
  public void lisaa(int lisaMin)  {
    aseta(h,m+lisaMin);
  }

  public static void lisaa(Aika 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; }

}
```

Valitettavasti rajapinnan toteuttavaa luokkaa ei voi pakottaa tekemään uutta `toString` metodia, koska `Object` luokan metodina sen toteuttaa valmiiksi jo jokainen luokka. Vaikka kääntäjä ei ilmoittaisikaan virheestä, niin tässä tapauksessa `toString` on hyödyllistä lisätä rajapinnan määrittelyihin, jotta ohjelmoija näkee halutun toiminnallisuuden.

Sitten esim. koosteluokka ilmoitetaan toteuttamaan myös sama rajapinta:

oliot.aika.rajapinta.AikaSek.java - luokka joka toteuttaa rajapinnan => polymorfismi

```
public class AikaSek implements AikaRajapinta {

  private Aika hm = new Aika();
  private int s;

  public final int aseta(int h,int m, int s) {
     m += s / 60; //liiat sekunnit minuutteihin
     s %= 60;     //sekunnit välille -59 - 59
     if (s<0) { s += 60; m--; }
     int vrk = hm.aseta(h,m);
     this.s = s;
     return vrk;
  }

  public final int aseta(int h,int m) { return aseta(h,m,0); }
  public final int aseta(int h)       { return aseta(h,0);   }

  public AikaSek()              { aseta(0,0,0); }

  public AikaSek(int h,int m, int s) { // Muodostaja
    aseta(h,m,s);
  }

  public AikaSek(int h,int m) { aseta(h,m,0); }
  public AikaSek(int h)       { aseta(h,0,0); }

  /**
   * @return aika merkkijonona muodossa 12:05:03
   */
  @Override
  public String toString() {
    return hm.toString()+String.format(":%02d", s);
  }

  public void lisaa(int lisaMin,int lisaSek)  {
      aseta(hm.getH(), hm.getM()+lisaMin, this.s+lisaSek);
  }

  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) {
    Aika a1 = new Aika();
    Aika a2 = new Aika(13);
    Aika a3 = new Aika(14,175);
    AikaSek a4 = new AikaSek(14,55,45);

    // 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 AikaSek(23,59,59);

    for (int i=0; i < ajat.length; i++ ) {
      System.out.print(ajat[i]+ " +" + i + " => ");
      ajat[i].lisaa(i); System.out.println(ajat[i]);
    }
  }
}
```

Näin voimme jälleen tehdä taulukon, johon voimme laittaa kaikkia `AikaRajapinta`-määrittelyn toteuttavien luokkien olioita.

8.6 `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 voitaisiin 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
```

oliot.aika.object.Aika.java - luokka joka toteuttaa Object

```
import oliot.aika.rajapinta.AikaRajapinta;

/**
 * Luokka toteuttamaan sovitun julkisen rajapinnan ja Object-
 * luokan metodeja
 * …
 */
public class Aika implements AikaRajapinta {

    private int h, m;

    /** … */
    public Aika() { aseta(0, 0); }

    /** … */
    public Aika(int h, int m) { aseta(h, m); }
```

```
    /** … */
    public final int aseta(int h, int m) { …}

    /** …  */
    public void lisaa(int lisaMin) { … }

    public int getH() { return h;  }
    public int getM() { return m; }

    /** … */
    public String toString() {
        return String.format("%02d:%02d", getH(), getM());
    }

    /**
     * @example
     * <pre name="test">
     * Aika a1 = new Aika(13,37);
     * Aika a2 = new Aika(13,37);
     * a1 === a2
     * </pre>
     */
    public boolean equals(Object o) {
        if (!(o instanceof AikaRajapinta))
            return false;
        AikaRajapinta a = (AikaRajapinta) o;
        return getH() == a.getH() && getM() == a.getM();
    }

    /**
     * @example
     * <pre name="test">
     * Aika aika = new Aika(13,37);
     * Aika aika2 = (Aika)aika.clone();
     * aika2.toString() === "13:37";
     * </pre>
     */
    public Object clone() {
        return new Aika(getH(), getM());
    }

    /**
     * Aika sekunteina vuorokauden alusta
     * @example
     * <pre name="test">
     * new Aika(13,37).hashCode() === 49020;
     * </pre>
     */
    public int hashCode() { return 3600 * getH() + 60 * getM(); }
}
```

Metodi `toString` onkin jo entuudestaan tuttu

```
public String toString() {
    return String.format("%02d:%02d",getH(), getM());
}
```

Kun 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 Aika(getH(),getM());
  }
```

Tehtävä 8.15 `equals` `toString` avulla

Toteuta `equals`-metodi `toString`-metodin avulla. Arvioi ratkaisun tehokkuutta.

Tehtävä 8.16 `equals` `AikaSek`-luokkaan

`equals`-metodiin tulee ongelmia toteutettaessa `AikaSek`-luokkaa. Mieti mitä.

Tehtävä 8.17 `AikaSek` perimällä.

Esimerkissä `AikaSek` on toteutettu sekunnit sisältävä aikaluokka koostamalla. Kokeile miten nyt onnistuu perintä `Aika`-luokasta ja mitä metodeja pitää korvata.

Tehtävä 8.18 Vertailu

Tutki dokumenteista rajapintaa `Comparable` ja muuta luokat `Aika` ja `AikaSek` toteuttamaan tuo rajapinta.

8.7 Mistä hyviä luokkia

Alun perin kirjoittamamme luokka `Aika` kokikin varsin kovia tarkemmassa tarkastelussa. Näistä muutoksista osa oli vielä aivan perusasioita; läheskään kaikkea emme vieläkään ole ottaneet huomioon (lisäyksessä tapahtuvan ylivuodon luovuttaminen päivämäärälle, jne.). Miten sitten on monimutkaisempien luokkien kanssa? Niin kauan pärjää, kun luokat ovat 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 yli-

luokka, 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. *Microsoftin .NET* tai *Delphi,* 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", siirtyvät ammattilaiset yhä spesifimmille aloille.

Monien kielten 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.

8.8 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.

8.8.1 Merkkijonoluokat

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 äärettömästi 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++*:ssa 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 `StringBuilder`. 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 8.19 Ensimmäinen melkein järkevä olio

Täydennä seuraavat luokat niin että testit menevät läpi

oliot.henkilo.Henkilo.java - 1. järkevä olio

```
/**
 * Henkilöluokka
 * Täydennä luokka.
 * @author  Vesa Lappalainen
 * @version 1.0, 05.02.2003
 * Henkilo kalle = new Henkilo("Kalle",35,1.75);
 * kalle.toString() === "Kalle, pituus 1.75";
 * kalle.kasvata(2.3);
 * kalle.toString() === "Kalle, pituus 4.05";
 */
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) {

  }

}
```

oliot.henkilo.Opiskelija.java - 1. järkevä olio

```
/**
 * Opiskelija, joka on peritty henkilöstä
 * Täydennä luokka.
 * @author  Vesa Lappalainen @version 1.0, 05.02.2003
 * @author Santtu Viitanen @version 1.1, 9.7.2011
 * @example
 * <pre name="test">
 * Opiskelija ville = new Opiskelija("Ville",21,1.80,9.9);
 * ville.toString() === "Ville, pituus 1.8, keskiarvo 9.9";
 * </pre>
 */
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 "";
  }

}
```

9. 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`, `for` ja `for-each`
- 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:

```
https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/ohjausrak/
```

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.

9.1 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.

9.1.1 Ehdolla suoritettava yksi lause

Olkoon meillä aliohjelma nimeltään `tulosta`, joka parametrina viedyn luvun:

```
if ( a > b ) tulosta(a);
```

9.1.2 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*):

```
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 9.1 vaihda

Esitä pöytätestin avulla miksei vaihtaminen onnistu pelkästään lauseilla:
a = b; b = a;

Tehtävä 9.2 abs

Kirjoita funktio
`int itseisarvo(int i),`
joka palauttaa `i`:n itseisarvon (negat. muutet. posit.).

Tehtävä 9.3 jarjesta2

Kirjoita aliohjelma
`void tulosta2(int a, int b),`
joka tulostaa luvut suuruusjärjestyksessä .

Tehtävä 9.4 maksimi ja minimi

Kirjoita funktio
`int maksimi(int a, int b),`
joka palauttaa suuremman kahdesta luvusta.

Kirjoita vastaava funktio `minimi`.

9.2 Loogiset lausekkeet

Java–kielessä vain boolean-arvoiset lausekkeet käsitellään loogisina lausekkeina. Arvo `false` on epätosi ja `true` on tosi.

```
a = 4;
if ( a == 4 ) ...
boolean samat;
samat = ( a == 4 );
if ( samat ) ...
```

9.2.1 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!");
```

9.2.2 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 syntynyt 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:

ohjausrak.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
}
```

9.3 Loogisten lausekkeiden yhdistäminen

Loogisia lauseita voidaan yhdistää loogisten operaatioiden avulla. Lisäksi esimerkiksi C:ssä lauseita voidaan yhdistää myös normaaleilla operaatioilla (+,−,*,/), mutta tämä ei ole oikein hyvien tapojen mukaista.

9.3.1 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 ) ) ...
```

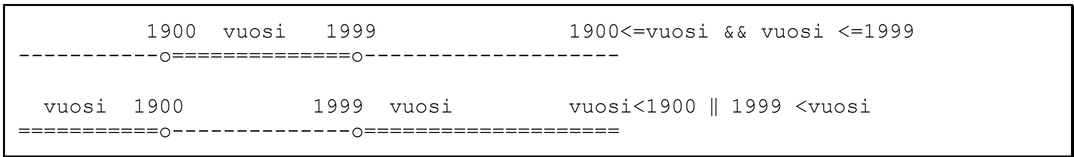
Huomattakoon edellä miten väliä korostettiin kirjoittamalla välin päätepisteet lauseen laidoille.

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:



```
          1900  vuosi   1999              1900<=vuosi && vuosi <=1999
-----------o=============o--------------------

  vuosi  1900          1999  vuosi         vuosi<1900 || 1999 <vuosi
===========o-------------o====================
```

9.3.2 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.

Kekseliäämpi ohjelmoija kirjoittaisi testin kuitenkin luettavampaan muotoon

```
If ( ”kissa”.equals(jono) ) tulosta(”On kissa”);
```

Nyt kysytään varmasti olemassa olevalta `String` luokan instanssilta onko sen sisältö sama kuin jonon, jolloin **null**-viittausta ei tarvitse tarkastaa.

9.3.3 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
```

Aikaisempaa esimerkkiä mukaillen:

```
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") )
        System.out.println("On kissa");
```

9.4 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 9.5 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 9.6 Luku parilliseksi

Kirjoita funktio `parilliseksi`, joka palauttaa parametrinaan olevan kokonaisluvun pienemmäksi parilliseksi luvuksi "katkaistuna". Eli esim. 3 –> 2. 5 –> 4. 4 –> 4.

9.5 `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!");
    }
```

9.5.1 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:

ohjausrak.polynomi.v1.Polynomi2.java - esimerkki 2. asteen yhtälön ratkaisemiseta

```
/**
```

```
 * Luokka toisen asteen polynomille ja sen nollakohdille
 * @author Vesa Lappalainen @version 1.0, 16.02.2003
 * @author Santtu Viitanen @version 1.1 12.7.2011
 * @example
 * <pre name="test">
 * Polynimi p;
 * p = new Polynomi2(1,2,1);
 * p.juuret() === "x1 = -1.0 => P(x1) = 0.0  ja  x2 = -1.0 => P(x2) = 0.0"
 * p = new Polynomi2(2,1,0);
 * p.juuret() === "x1 = -0.5 => P(x1) = 0.0  ja  x2 = 0.0 => P(x2) = 0.0"
 * p = new Polynomi2(1,-2,1);
 * p.juuret() === "x1 = 1.0 => P(x1) = 0.0  ja  x2 = 1.0 => P(x2) = 0.0"
 * p = new Polynomi2(2,-1,0);
 * p.juuret() === "x1 = 0.0 => P(x1) = 0.0  ja  x2 = 0.5 => P(x2) = 0.0"
 * new Polynomi2(2,1,1).juuret() === "Ei yhtään reaalijuuria"
 * p = new Polynomi2(2,0,0);
 * p.juuret() === "x1 = -0.0 => P(x1) = 0.0  ja  x2 = 0.0 => P(x2) = 0.0"
 * p = new Polynomi2(0,2,1);
 * p.juuret() === "x1 = -0.5 => P(x1) = 0.0  ja  x2 = -0.5 => P(x2) = 0.0"
 * new Polynomi2(0,0,1).juuret() === "Ei yhtään reaalijuuria"
 * </pre>
 */
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 = ratkaise2AsteenYhtalo();
  }

  private int ratkaise2AsteenYhtalo() {
    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; }
  public String juuret() {
      if ( getReaalijuuria() <= 0 ) return "Ei yhtään reaalijuuria";
      return
```

```
      "x1 = " + getX1() + " => P(x1) = " + f(getX1()) + "  ja  "+
      "x2 = " + getX2() + " => P(x2) = " + f(getX2());
      }
}
```

Edellinen metodi `ratkaise2AsteenYhtalo` 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:

ohjausrak.polynomi.v2.Polynomi2.java - karsittu versio 2. asteen yhtälöstä

```
  private int ratkaise2AsteenYhtalo() {
    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:

ohjausrak.polynomi.v3.Polynomi2.java - normaalit tapaukset ensin ratkaisussa

```
  private int ratkaise2AsteenYhtalo() {
    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:

ohjausrak.polynomi.v4.Polynomi2.java - else -osat pois

```
  private int ratkaise2AsteenYhtalo() {
    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 );
```

9.5.2 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:

ohjausrak.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
```

On myös helppo huomata että aliohjelma `postimaksu` toimisi täysin samalla tavalla, vaikka jokaisen `if`-lauseen `else`-osan jättäisi kokonaan pois. Usein liian vaikeilla ehtorakenteilla monimutkaistetaan turhaan koodia.

Tehtävä 9.7 elset pois

Aliohjelma postimaksu oli mahdollista kirjoittaa ilman `else`-lauseita. Miksi?

Tehtävä 9.8 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 9.9 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".

9.6 `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`):

alkeet.alkuluku.Alkuluku.java - testataan onko luku alkuluku

```
/**
 * Ohjelmalla testataan onkoAlkuluku-aliohjelmaa
 * @author Vesa Lappalainen
 * @version 20.1.2011
 *
 */
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 luvun jolla jaollinen tai 1 jos alkuluku
     * @example
     * <pre name="test">
     *   onkoAlkuluku(25)  === 5;
     *   onkoAlkuluku(2)   === 1;
     *   onkoAlkuluku(4)   === 2;
     *   onkoAlkuluku(123) === 3;
     *   onkoAlkuluku(7)   === 1;
     * </pre>
     */
    public static int onkoAlkuluku(int luku) {
        int jakaja = 2;
        int kasvatus = 1;
        if ( luku == 2 ) return 1;                  // 0

        do {
            int jakojaannos = luku % jakaja;
            if (jakojaannos == 0) return jakaja; // 1
            jakaja += kasvatus; // 2
            kasvatus = 2; // 3

        } while (jakaja < luku / 2);
```

```
        return 1;
    }
}
```

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:

ohjausrak.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);

  }
}
```

9.7 `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.

ohjausrak.alkuluku2.Alkuluku2.java - alkulukutesti while-silmukalla

```
  * <pre name="test">
  * pienin_jakaja(25) === 5;
  * pienin_jakaja(123) === 3;
  * pienin_jakaja(7) === 1;
  * </pre>
  */
```

```java
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;
}
```

9.8 `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:

ohjausrak.vali.Valinsum.java - esimerkki for-silmukasta

```java
/**
 * Lasketaan yhteen luvut 1..ylaraja
 * @param ylaraja summan yläraja
 * @return summa
 * @example
 * <pre name="test">
 * valinSumma($param1) === $tulos
 * $param1 | $tulos
 * -----------------
 *   0     |  0
 *   1     |  1
 *   2     |  3
 *   3     |  6
 * </pre>
 */
public static int valinSumma(int ylaraja) {
  int i,summa=0;
  for (i=1; i<=ylaraja; i++)
    summa += i;
  return summa;
}
```

Tehtävä 9.10 valinSumma

Muuta `valinSumma` –aliohjelmaa siten, että myös alaraja viedään parametrina. Kirjoita pääohjelma, jolla toiminta voidaan testata.

Käytännössä tällaisia silmukoita ei saa tehdä, koska ongelman ratkaisuun on valmis kaava. Millainen?

9.9 `for-each` silmukka

`for`-silmukan monikäyttöisyydestä huolimatta se on hieman monimutkainen, kun tarkoituksena on vain käydä tietorakenne läpi alkio kerrallaan. Helpompaa onkin usein käyttää for-each-silmukkaa, jonka käyttö kuvataan tarkemmin luvussa **Java-kielen taulukoista**.

9.10 Java–kielen lauseista

9.10.1 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 yhtä aikaa 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;
```

9.10.2 Sijoitus- ja kasvatusoperaattori +=

`valinSumma` aliohjelmassa meillä esiintyi myös kaksi uutta sijoitusoperaattoria, jotka ovat lyhenteitä tavallisille sijoituksille:

lyhenne	tavallinen sijoitus
summa += i; i++	summa = summa + 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 9.7 +=

Mitä ovat muuttujien arvot seuraavien sijoitusten jälkeen:

```
int a=10,b=3,c=5;
a %= b;
b *= a+c;
b >>= 2;
```

9.10.3 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:

ohjausrak.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);
```

9.11 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:

ohjausrak.Valinsum.java - useita alustuslauseita for-silmukassa

```
  /**
   …
   * <pre name="test">
   * valinSumma2($param1) === $tulos
   * $param1 | $tulos
   * -----------------
   *   0     |  0
   *   1     |  1
   *   2     |  3
   *   3     |  6
   * </pre>
   */
  public static int valinSumma2(int ylaraja) {
    int i,summa;
    for (summa=0, i=1; i<=ylaraja; i++)
      summa += i;
    return summa;
  }
```

Erittäin C:mäinen tapa tehdä yhteenlasku olisi:

ohjausrak.Valinsum.java - C:mäinen silmukka

```
  public static int valinSumma3(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 9.8 1+2+..+i

Miksi `valinSumma3` laskee yhteen luvut 1..i?

9.12 break ja continue

9.12.1 break

Joskus kesken silmukan tulee vastaan tilanne, jossa silmukan suoritus haluttaisiin keskeyttää. Tällöin voidaan käyttää C–kielen break–lausetta, joka katkaisee **sisimmän** silmukan suorituksen.

ohjausrak.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:

ohjausrak.Break.java - ulomman silmukan katkaisu keskeltä

```
  private static void breakTesti3() {
    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 voitaisiin 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 9.9 Tarvitaanko sisäkkäisiä silmukoita?

Tarvitaanko aliohjelmassa `breakTesti3` todella sisäkkäisiä silmukoita? Esitä ratkaisu jossa on vain yksi silmukka.

9.12.2 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:

ohjausrak.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 9.10 continuen korvaaminen

Kirjoita käänteislukujen tulostusohjelma ilman `continue`–lausetta.

Tehtävä 9.11 Eri silmukoiden vertailu

Kirjoita lukujen alaraja-yläraja summausfunktio käyttäen

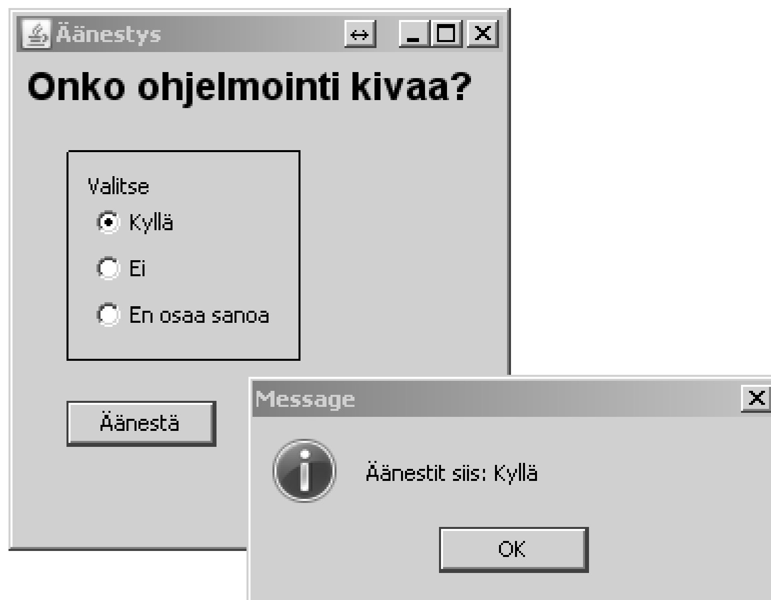
a) `while` -lausetta
b) `do-while` -lausetta
c) `goto` –lausetta

Muista, että alaraja saattaa olla suurempi kuin yläraja, eli summa väliltä `[3,0]` on `0`!

9.13 `switch` –valintalause

`Switch` -valintalause on selkeä tapa esimerkiksi käyttöliittymäohjelmoinnissa toteuttaa käyttäjän tekemien monivalintojen logiikka, mutta toki se soveltuu moneen muuhunkin asiaan. Valintalausetta voi käyttää primitiivityyppien `byte`, `short`, `char` ja `int` kanssa, enumeroiduilla tyypeillä, sekä Java SE 7:n jälkeen myös merkkijonoilla.

Käyttöliittymästä käyttäjä pystyy valitsemaan vaihtoehdon, minkä perusteella pystymme tulostamaan ikkunan, jossa näkyy valittu vaihtoehto. Käyttöliittymää tehdessä Swing ympäristössä `JRadioButton` komponentit saa kytkettyä toisiinsa `ButtonGroup`in avulla.



Kuva 9.1 Äänestysohjelman käyttöliittymä

ohjausrak.SwingAanestys.java - esimerkki switch-lauseesta

```
/**
 * Pieni esimerkki äänestys-ohjelmasta switch-lauseen demonstroimiseksi. *
 * @author Vesa Lappalainen
 * @version 6.2.2011
 */
public class SwingAanestys extends JFrame {

...

    private final JLabel lblValitse = new JLabel("Valitse");
    private final JRadioButton rb0 = new JRadioButton("Kyll\u00E4");
    private final JRadioButton rb1 = new JRadioButton("Ei");
    private final JRadioButton rb2 = new JRadioButton("En osaa sanoa");
    private final JButton buttonAanesta = new JButton("\u00C4\u00E4nest\u00E4");
```

```java
    private final ButtonGroup groupAanestys = new ButtonGroup();

...

    public SwingAanestys() {

...

        groupAanestys.add(rb0);
        rb0.setHorizontalAlignment(SwingConstants.TRAILING);
        rb0.setSelected(true);
        rb0.setMnemonic('K');

        panelValinta.add(rb0);
        groupAanestys.add(rb1);
        rb1.setMnemonic('E');

        panelValinta.add(rb1);
        groupAanestys.add(rb2);
        rb2.setMnemonic('O');

        panelValinta.add(rb2);
        panelAanestys.add(verticalStrut);
        buttonAanesta.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                aanesta();
            }
        });

        panelAanestys.add(buttonAanesta);
        getRootPane().setDefaultButton(buttonAanesta);
    }

    /// Omat aliohjelmat

    private void aanesta() {
        ButtonModel b = groupAanestys.getSelection();
        char nappain = (char)b.getMnemonic();
        String kohde = "";
        switch (nappain) {
            case 'K': kohde = "Kyllä"; break;
            case 'E': kohde = "Ei"; break;
            case 'O': kohde = "En osaa sanoa"; break;
        }
        JOptionPane.showMessageDialog(null,"Äänestit siis: " + kohde);
    }
```

`switch` –lauseessa `case` osien lopuksi `break` on yleensä välttämätön. `break` estää suorittamasta seuraavia rivejä.

Joskus harvoin `break`in puuttumista voidaan käyttää hyväksi, mutta tällöin vaaditaan taitavaa `break`-käskyn käyttöä.

ohjausrak.Switch.java - switch, jossa break tahallaan jätetty pois

```java
  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 lauseen logiikka pitää silloin olla huolellisesti mietitty.

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 9.12 switch –> if

Kirjoita `Switch.java` ohjelmanpätkä käyttäen `if`–rakenteita muuttamatta itse suoritettavia lauseita.

Tehtävä 9.13 Päävalinta

Kirjoita `paavalinta` käyttäen vain `if` ja `else` rakenteita.

Tehtävä 9.14 lääni, versio 2

Kirjoita `laani`–aliohjelma käyttäen `Switch`–rakennetta.

9.13.1 | 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:
```

9.14 Ikuinen silmukka

Usein silmukat lipsahtavat tahattomasti 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 );
```

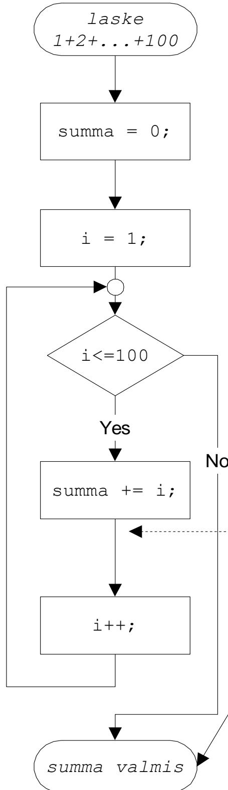
Näissä kahdessa ensimmäisessä korostuu silmukan ikuisuus. Viimeinen ei ole hyvä vaihtoehto.

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.

9.14.1 Yhteenveto silmukoista

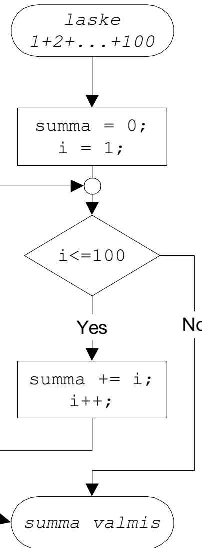
for

```
summa = 0;
for (i=1; i<=100; i++) {
  summa += i;
}
```



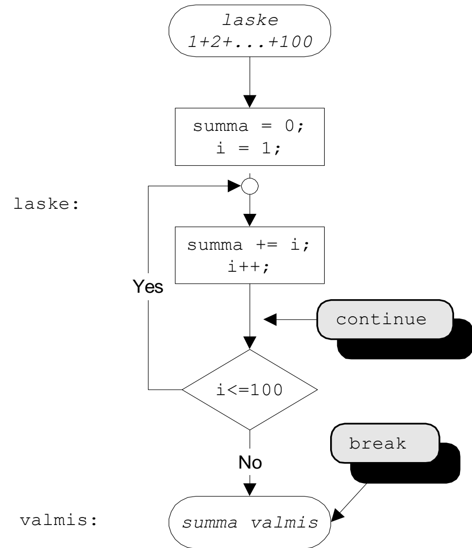
while

```
summa = 0;
i = 1;
while ( i <= 100 ) {
  summa += i;
  i++;
}
```



do-while

```
summa = 0;
i = 1;
do {
  summa += i;
  i++;
} while ( i <= 100 );
```




```
summa = 0;
i = 1;
laske:
  if ( i > 100 ) goto valmis
  summa += i;
  i++;
  goto laske;
valmis:
```

```
summa = 0;
i = 1;
laske:
  summa += i;
  i++;
if ( i <= 100 ) goto laske;
```

```
/* 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
```

```
summa = 0;
i = 1;
laske: if ( i <= 100 ) {
  summa += i;
  i++;
  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

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");
    }
```

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)

10. 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?

10.1 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*).

10.2 Tavoitteet

Asetamme ensin ohjelman toteutukselle ulkoisen toiminnan lisäksi tiettyjä lisätavoitteita:

- käyttöliittymä (tekstipohjainen vai ikkunoitu) on voitava muuttaa kohtuullisella ohjelmoinnilla
- ohjelma on voitava pienillä muutoksilla muuttaa muuksikin kuin jäsenrekisteriksi (puhelinluettelo, levyrekisteri)
- jäseneen voitava helposti lisätä kenttiä

10.3 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.

10.4 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 eivät 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. Kortti annetaan aina yhdelle ryhmän jäsenelle. Kortin saaja voi tarkistaa, saako hän korttia vastaavasta luokasta tarvitsemansa tiedot kääntämättä korttia. Jollei saa, luokkia on vielä helppo muuttaa kun ohjelmaa ei ole kirjoitettu.

10.4.1 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

10.4.2 Kerho-luokka, yksinkertainen (`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`

10.4.3 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`

10.4.4 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 (`KerhonNaytto`) ja jäsenen näyttö (`JasenenNaytto`).

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.

10.5 Harrastukset mukaan kortteihin

Entä sitten kun haluamme lisätä kerholaisille harrastuksia. Ainakin tarvitaan `Harrastus`–luokka lisää. Kuka huolehtii harrastuksista?

Jos valinta tehdään valitun tiedostomuodon mukaan, niin mahdollisuuksia on:

10.5.1 "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.

10.5.2 Relaatiomalli kortteihin

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.

10.5.3 Harrastus-luokka (`Harrastus`)

Vertaa toimintaa `Jasen`–luokan toimintoihin.

10.5.4 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`

10.5.5 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`

10.5.6 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.

11. 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

11.1 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.

11.2 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:

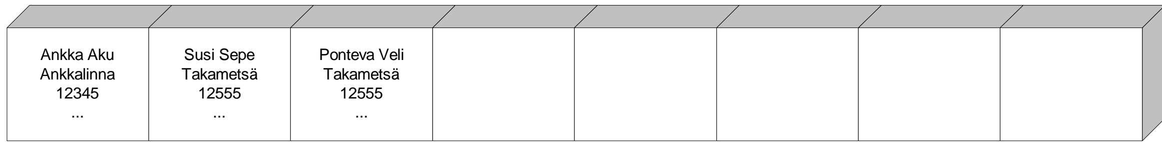
```
lue tiedosto muistiin
```

```
käsittele aineistoa muistissa
```

```
talleta aineisto takaisin levylle
```

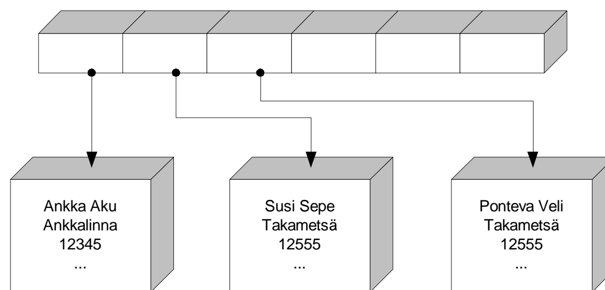
11.2.1 Taulukko

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 11.1 Taulukko C++ -kielessä

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:



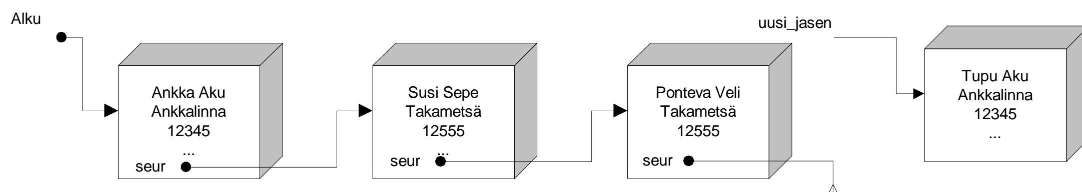
Kuva 11.2 Javan taulukko

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).

11.2.2 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 11.3 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.

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.

11.2.3 Sekarakenne

Valitsemme tähän esimerkkitoteutukseen tietorakenteeksi sekarakenteen:

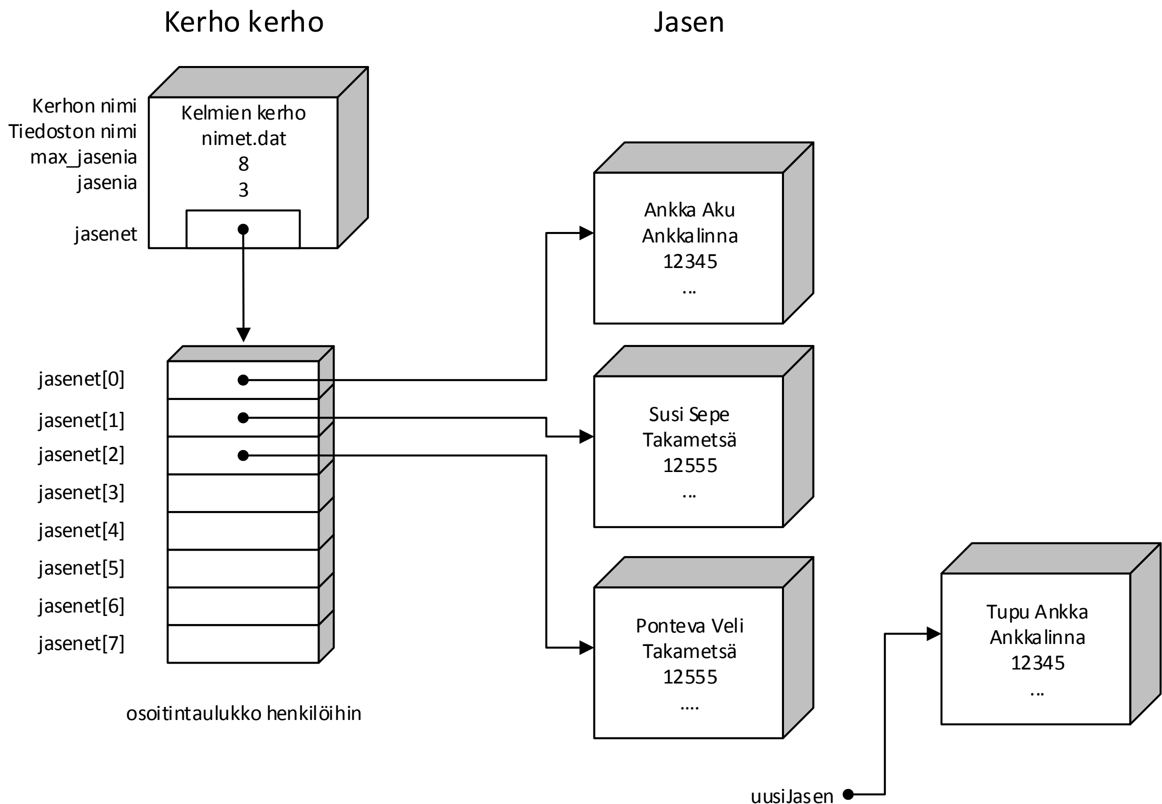
Siis perusrakenteena meillä on `Kerho`–tyyppi, joka pitää sisällään kerhon perustiedot. Kerhosta on osoitin taulukkoon, jossa on osoittimet varsinaisiin henkilöiden tietoihin (`Jasen`).

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 11.4 Tietorakenne kun kerho tallettaa jäsenet

Tehtävä 11.1 Lisäys

Kirjoita algoritmi henkilön lisäämiseksi rakenteeseen.

Tehtävä 11.2 Etsiminen

Kirjoita algoritmi tietyn henkilön etsimiseksi (vaikkapa nimellä).

Tehtävä 11.3 Poisto

Kirjoita algoritmi löydetyn henkilön (miten löytö kannattaa säilyttää?) poistamiseksi rakenteesta.

Tehtävä 11.4 Lajittelu

Kirjoita algoritmi rakenteen lajittelemiseksi aakkosjärjestykseen. Mitä lajittelussa kannattaa vaihdella?

11.3 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.

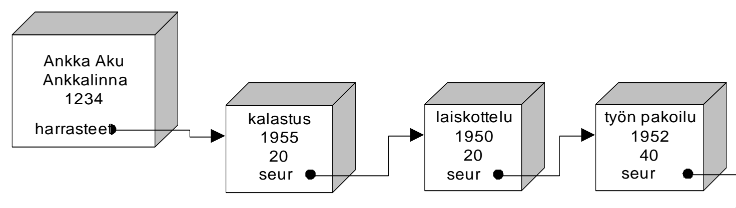
11.3.1 Tiedot jäsenen yhteyteen

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 11.5 Harrastukset linkitettynä listana

11.3.2 Relaatiomalli tietorakenteeseen

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 11.5 Lisää harrastus

Kirjoita algoritmi uuden harrastuksen lisäämiseksi. (Ks. alla oleva kuva)

Tehtävä 11.6 Mitä harrastaa

Kirjoita algoritmi, joka vastaa kysymykseen "Mitä henkilö N harrastaa?"

Tehtävä 11.7 Kuka harrastaa

Kirjoita algoritmi, joka vastaa kysymykseen: "Ketkä harrastavat harrastusta H?".

Tehtävä 11.8 Poista harrastus

Kirjoita algoritmi harrastuksen poistamiseksi.

Tehtävä 11.9 Jäsenen poistaminen

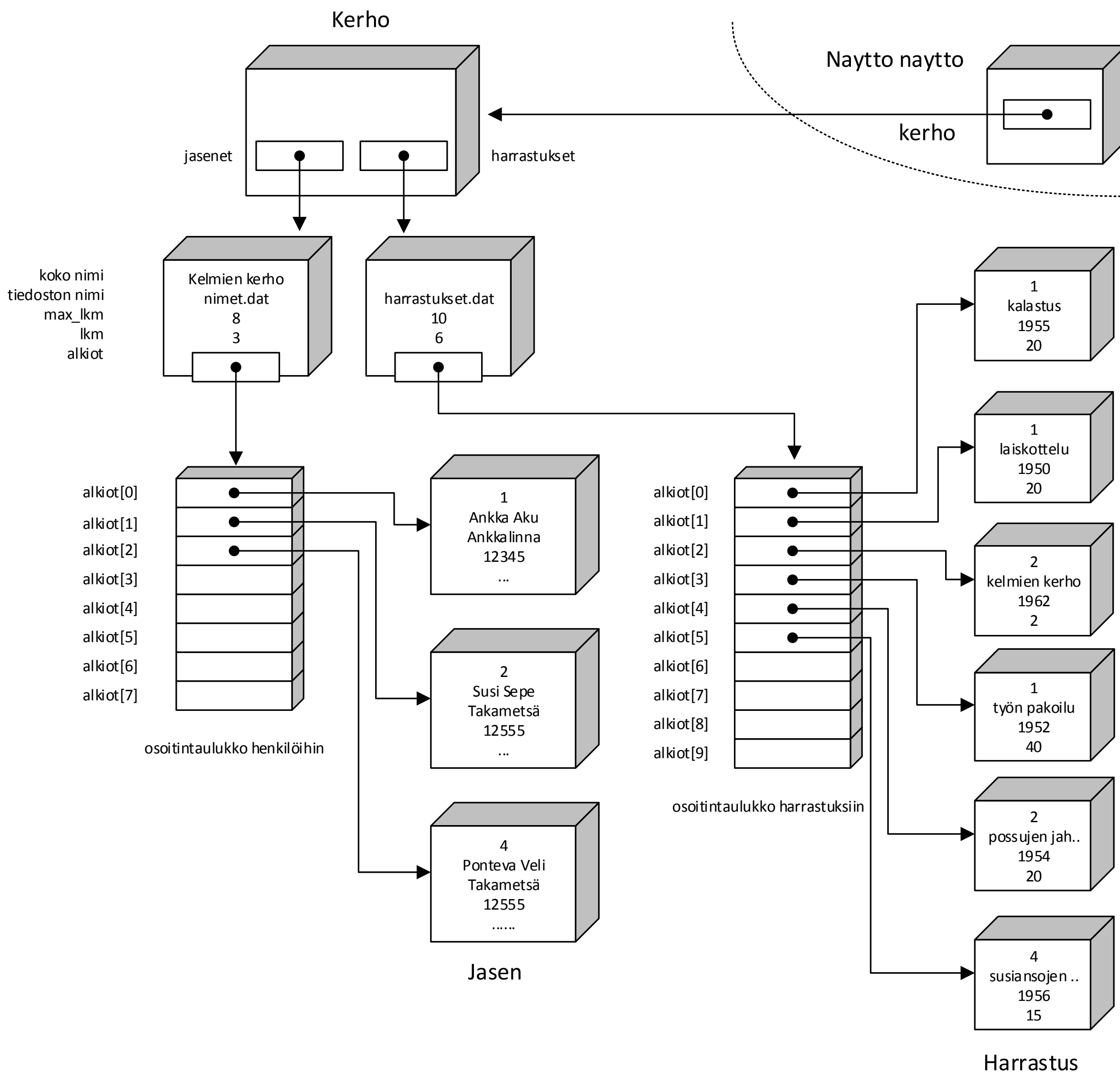
Kirjoita algoritmi, joka poistaa jäsenen jonkin nimi on N.

Tehtävä 11.10 "Roskaharrastukset"

Kirjoita algoritmi, joka poistaa "roskaharrastukset", eli ne harrastukset, joille ei löydy "omistajaa". Tällaiseen tilanteeseen hyvässä ohjelmassa ei tietenkään koskaan päädytä.

Tehtävä 11.11 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 11.6 Harrastukset relaation avulla

12. 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:

https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/taulukot/

12.1 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.

12.1.1 Taulukon määrittely

Taulukko määritellään kertomalla taulukon alkioiden tyyppi ja luomalla sitten varsinainen taulukko:

```
int kPituudet[];         // saadaan vasta viite jolla voidaan viitata taulukkoon
kPituudet = new int[12];  // luodaan taulukko;
```

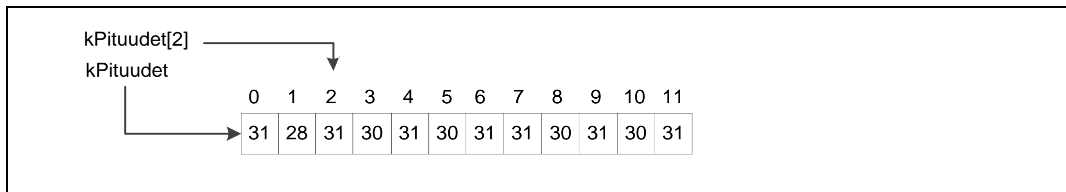
Tällöin taulukon 1. alkion indeksi on 0 ja 12. alkion indeksi on 11.

Määrittelyllä muuttujasta `kPituudet` tulee osoitin kokonaislukuun; taulukon alkuun.

12.1.2 Taulukon alkioihin viittaaminen indeksillä

Taulukon alkioon voidaan viitata alkion indeksin avulla

```
kPituudet[0]=31; /* tammikuu */
kPituudet[1]=28; /* helmikuu */
```



Taulukon rajojen ylityksestä seuraa `IndexOutOfBoundsException`-poikkeus

```
kPituudet[24]=31;
```

eli 2 paikkaa eteenpäin taulukon alusta lukien.

Huomautus!

12.1.3 Taulukon läpikäyminen `for` ja `for-each`–silmukoilla

Käydään taulukko `kPituudet` läpi ja lasketaan niiden summa `for`-silmukalla

```
int summa = 0;
for (int i=0; i<kPituudet.length; i++)
  summa += kPituudet[i];
```

On tärkeää huomata että taulukoiden käsittelyssä indeksi liikkuu välillä `[0,YLÄRAJA[`.

Monikäyttöisyydestään huolimatta tavallinen `for`-lause on hieman kömpelö tapa tietorakenteiden läpikäymiseen alkio kerrallaan. Siksi Javaan on lisätty myös `for-each`-silmukka , jolla edellinen koodi näyttäisi tältä.

```
int summa = 0;
for (int paivia : kPituudet)
  summa += paivia;
```

12.1.4 Taulukon alustaminen

Taulukko voidaan alustaa (vain) esittelyn yhteydessä:

```
                /*  1. 2. 3. 4. 5. 6. 7. 8. 9.10.11.12 */
int kPituudet[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Tehtävä 12.1 Taulukon alkioiden nollaus

Kirjoita funktio–aliohjelma `taulukonNollaus`, ensin normaalilla `for`-silmukalla. Miksi nollaus ei onnistu `for-each`-silmukalla?

Mikäli vaihdat taulukon ja silmukan käyttämään `Integer`-olioita niin tyhjentäminen ei onnistu vieläkään.

```
Integer[] kPituudet = {1,3,5,2};
```

Miksi näin ja minkä tyyppisten olioiden sisältöä voi muokata `for-each`-rakenteella?

12.2 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 `StringBuilder` (tai `StringBuffer`).

12.2.1 Merkkityyppi

Yksittäinen merkki on Java–kielessä tyyppiä `char`:

```
char rek1Merkki;
rek1Merkki = 'X';
```

Merkkimuuttujiin voidaan vallan hyvin sijoittaa myös merkin koodiarvo

```
char m;
m = 65;
if ( m == 'A' ) ...
```

Lukuarvo tarkoittaa merkin (UNICODE–) koodia.

12.2.2 String

Javan `String`-luokka tarjoaa muuttumattoman merkkijonon (*immutable*). Merkkijonon "sisältö" voidaan vaihtaa vain luomalla uusi merkkijono.

12.2.3 StringBuilder ja StringBuffrer

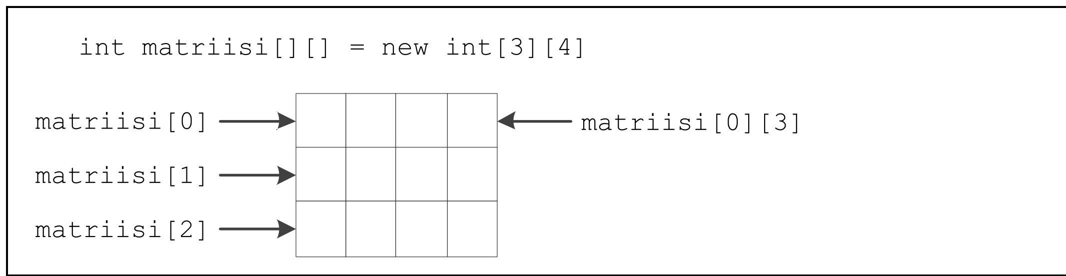
Jos halutaan merkkijono, jonka sisältöä voidaan muuttaa (*mutable*), pitää käyttää `StringBuilder`- tai `StringBuffer`-luokkaa. Nämä luokat ovat ulospäin identtiset, mutta vanhempi `StringBuffer` on hitaampi, toimien toisaalta paremmin rinnakkaisessa käsittelyssä. Nykyisin `StringBuilder` on suositeltavampi.

12.3 Moniulotteiset taulukot Javassa

Moniulotteiset taulukot ovat Javassa vain yksiulotteisia taulukoita taulukoista.

12.3.1 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 12.2 Matriisit

Kirjoita seuraavat aliohjelmat, jotka saavat parametrinaan 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).

12.3.2 Yksiulotteisen taulukon käyttäminen moniulotteisena

Toisaalta moniulotteinenkin taulukko voidaan toteuttaa 1–ulotteisena. Tästä muunnoksestahan puhuttiin jo monisteen alkuosassa. On makuasia kumpi 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 12.3 Matriisi 1–ulotteisena

Kirjoita aliohjelma `teeYksikko`, jolle tuodaan parametrina neliömatriisin rivien lukumäärä ja 1–ulotteisen taulukon viite, ja joka alustaa tämän neliömatriisin yksikkömatriisiksi.

12.3.3 Taulukko taulukoista

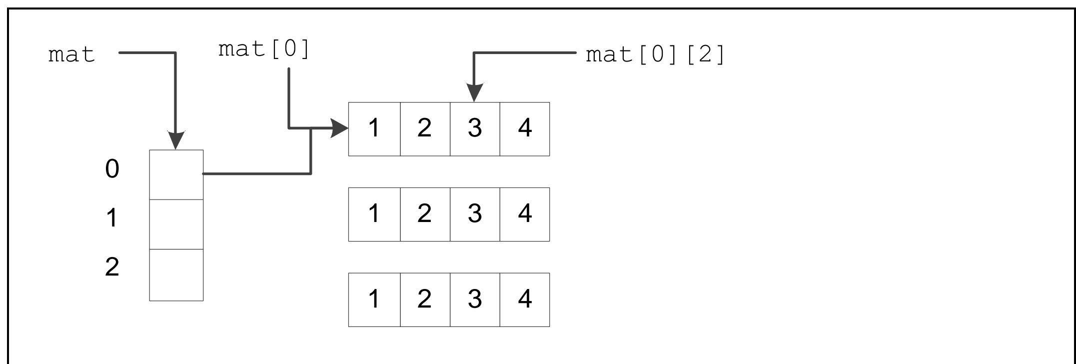
Javassahan moniulotteinen taulukko on siis taulukko taulukoista. C#:issa on myös aito kaksiulotteinen taulukko, toki myös taulukko taulukoista.

taulukot.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);
  }
}
```

12.3.4 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:



```
taulukot.Mat3.java - matriisi osoitintaulukon avulla
```

```java
/**
 * Matriisi kasattuna irrallisista riveistä
 * @author Vesa Lappalainen
 * @version 1.0, 04.03.2003
 */
public class Mat3 {

  public static double alkioidenSumma(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 = alkioidenSumma(mat,2,3);
    s2 = alkioidenSumma(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 12.4 Transpoosi

Kirjoita taulukko–osoittimia käyttäen aliohjelma, joka saa parametrinaan 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.

12.4 Komentorivin parametrit (`argv`)

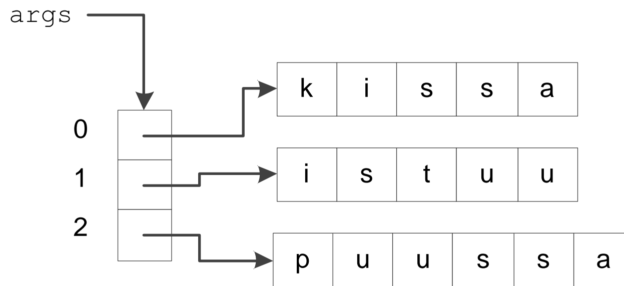
Esimerkiksi Java–kielinen pääohjelma saa käyttöjärjestelmältä tällaisen taulukon kutsussa olleista argumenteista:

```
taulukot.Argv.java - komentorivin parametrit
```

```java
/**
 * 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 ajetaan komentoriviltä on tulostus seuraavan näköinen.

```
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 12.5 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>_
```

13. 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;`

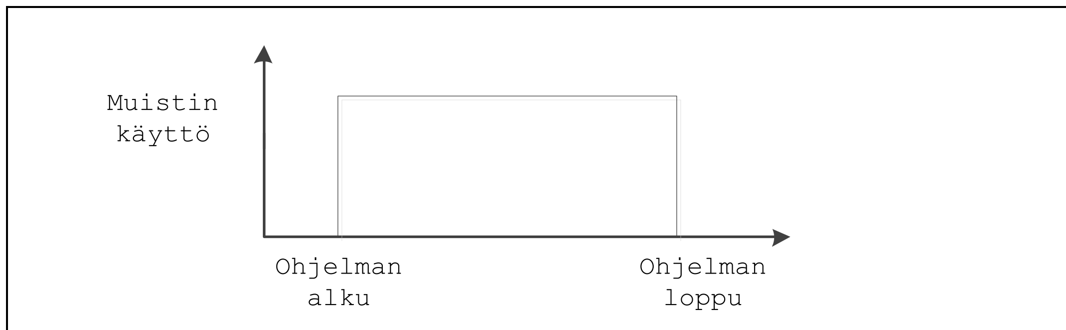
Luvun esimerkkikoodit:

https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/dynaaminen/

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).

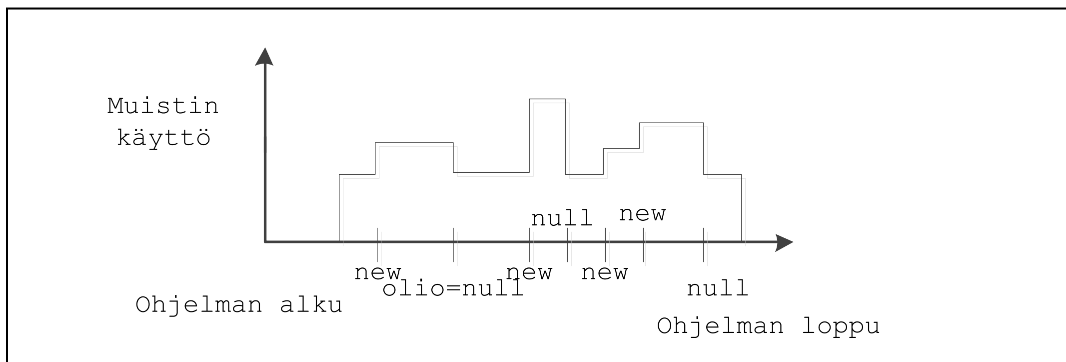
13.1 Muistin käyttö

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.

13.2 Dynaamisen muistin käyttö Javassa

13.2.1 `new`

Javassa kaikki oliot luodaan dynaamisesta muistista (keosta).

13.2.2 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.

13.2.3 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 null-viitettä.
```

13.3 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:

dynaaminen.Taulukko.java -esimerkki dynaamisesta taulukosta

```
/**
 * Esimerkki dynaamisesta taulukosta
 * @author Vesa Lappalainen @version 1.0, 02.03.2002
 * @author Santtu Viitanen @version 1.1, 21.07.2011
 * @example
 * <pre name="test">
 *  #THROWS Taulukko.TaulukkoTaysiException
 *  Taulukko luvut = new Taulukko();
 *  luvut.lisaa(0); luvut.lisaa(2); luvut.lisaa(99);
 *  luvut.toString() === "0,2,99,";
 *  luvut.set(1,4); luvut.toString() === "0,4,99,";
 *  int luku = luvut.get(2);
 *  luvut.get(2) === 99;
 *  luvut.set(21, 4); #THROWS IndexOutOfBoundsException
 *  luvut.lisaa(0); luvut.lisaa(0); //taulukko täyteen
 *  luvut.lisaa(0);  #THROWS Taulukko.TaulukkoTaysiException
 * </pre>
 */
public class Taulukko {
  public static class TaulukkoTaysiException extends Exception {
    TaulukkoTaysiException(String viesti) { super(viesti); }
  }

  private int alkiot[];
  private int lkm;

  public Taulukko() {
    alkiot = new int[5];
  }

  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];
  }
}
```

13.3.1 Poikkeukset

Ohjelmoidessa joutuu usein varautumaan jo ennalta mahdollisiin virhetilanteisiin. Ohjelmistoa kirjoitetaan kuitenkin monella tasolla, eikä poikkeustilanteiden käsittely usein edes kuulu matalan tason komponenteille. Javassa hyvä apu ongelmaan on poikkeustenhallinta (*exception handling*).

Ideana on että virheen sattuessa mistä kohtaa tahansa ohjelmakoodia voi heittää poikkeuksen. Poikkeuksen heittäminen kerrotaan metodin esittelyn yhteydessä `throws` apusanalla, jotta sen toiminnallisuutta käyttävä ohjelman osa tietää varautua siihen.

```
  public void lisaa(int i) throws TaulukkoTaysiException {
    if ( lkm >= alkiot.length ) throw new TaulukkoTaysiException("Tila loppu");
    alkiot[lkm++] = i;
  }
```

Javassa on kahdenlaisia poikkeuksia. Tarkistettuja poikkeuksia, jotka periytyvät `Exception`-luokasta ja ajonaikaisia, jotka periytyvät `Error` tai `RuntimeException`-luokista. Erona näillä kahdella tyypillä on, ettei ajonaikaisia poikkeuksia (kuten `NullPointer`) ole välttämätöntä ottaa kiinni. Käytännössä ohjelmoidessa käytetään lähinnä tarkistettuja poikkeuksia.

Luokka `TaulukkoTaysiException` on tarkistettu poikkeus. `Exception` luokan konstruktori ottaa parametrina virheviestin, jonka ohjelmoija itse määrittää. Vaikka tässä tapauksessa luokan nimi on jo varsin kuvaava, niin viestin on silti hyvä antaa lisätietoa ongelman luonteesta. Saattaa jopa olla että valmiiseenkin ohjelmaan pääsee bugi, jolloin viesti kulkeutuu aina loppukäyttäjälle asti.

```
  public class TaulukkoTaysiException extends Exception {
    TaulukkoTaysiException(String viesti) { super(viesti); }
  }
```

Mikäli käytettävä metodi on ilmoittanut voivansa heittää poikkeuksen, sen tulemiseen pitää valmistautua (`try`), tai jos poikkeuksien hoitaminen ei kuulu vielä alkuperäistä metodiakaan käyttävälle ohjelman osalle, niin poikkeus voidaan heittää myös uudestaan eteenpäin. Kun poikkeuksen tulemiseen on valmistauduttu, niin se tulee ottaa myös kiinni (`catch`), jolloin päätetään mitä toimenpiteitä kuuluu suorittaa poikkeuksen sattuessa. Esimerkin tapauksessa tulostetaan konsoliin virheviesti ”*tila loppui”*. Olisi myös mahdollista tehdä tarkistuksen yhteyteen myös `finally`-lohko, jonka sisältö suoritetaan joka tapauksessa, eli sekä ilman poikkeusta tai sen kanssa.

```
    try {
      luvut.lisaa(0); luvut.lisaa(2);
      luvut.lisaa(99);
    } catch ( TaulukkoTaysiException e ) {
      System.out.println("Virhe: " + e.getMessage());
    }
    // finally { }
```

13.3.2 Poikkeukset ja *ComTest*

Oliot ja poikkeukset tuovat testaamiseen aluksi pienet erikoisuutensa, mutta oikein generoitu *JUnit*-testi syntyy kuitenkin suhteellisen vaivattomasti. Testissä jokaista potentiaalista poikkeusta ei kannata ottaa kiinni, vaan `#THROWS`-makrolla määritellään myös generoitu testimetodi heittämään poikkeuksen eteenpäin. Generoitu testi ei myöskään sijaitse samassa luokassa kuin `TaulukkoTaysiException`, joka puolestaan sijaitsee `Taulukko`-luokan sisällä, joten testit kaatuvat, mikäli polku ole oikeassa muodossa.

```
/**
…
 * @example
 * <pre name="test">
 *  #THROWS Taulukko.TaulukkoTaysiException
 *  Taulukko luvut = new Taulukko();
 *  luvut.lisaa(0); luvut.lisaa(2); luvut.lisaa(99);
 *  luvut.toString() === "0,2,99,";
 *  luvut.set(1,4); luvut.toString() === "0,4,99,";
 *  int luku = luvut.get(2);
 *  luvut.get(2) === 99;
…
```

Mikäli testi ei jostain syystä toimi ja joku riveistä heittääkin odottamatta poikkeuksen, niin tällöin testiympäristö ottaa sen automaattisesti vastaan, eikä testi mene läpi, mikä onkin tietysti haluttu lopputulos.

Toisaalta on tiedettävä, että myös ohjelmoidut poikkeukset toimivat. Käytetään jälleen samaa `#THROWS`-makroa halutun lauseen jälkeen

```
…

 *  luvut.set(21, 4); #THROWS IndexOutOfBoundsException
 *  luvut.lisaa(0); luvut.lisaa(0); //taulukko täyteen
 *  luvut.lisaa(0);  #THROWS Taulukko.TaulukkoTaysiException
 * </pre>
 */
```

jolloin *JUnit* tiedostoon generoituu jotain tämänkaltaista.

```
    try {
    luvut.set(21, 4);
    fail("Taulukko: 14 Did not throw IndexOutOfBoundsException");
    } catch(IndexOutOfBoundsException _e_){ _e_.getMessage(); }
    luvut.lisaa(0); luvut.lisaa(0);
    try {
    luvut.lisaa(0);
    fail("Taulukko: 16 Did not throw Taulukko.TaulukkoTaysiException");
    } catch(Taulukko.TaulukkoTaysiException _e_){ _e_.getMessage();
```

13.4 Geneerinen taulukko

Edellisen `Taulukko.java` esimerkin ongelmana on, ettei sillä pysty tallentamaan kuin `int` muotoisia arvoja. Oman taulukkoluokan luominen jokaiselle erilaiselle tyypille on tietysti työlästä, joten tarvitaan parempi ratkaisu. Javaan lisättiin 1.5 versiossa tuki geneerisyydelle, jonka avulla oliolle voi sitä luodessaan viestittää millaista dataa halutaan tallentaa.

Edelliseen Taulukko.java ohjelmaan ei tarvitse lisätä kuin vapaasti nimettävä tyyppiparametri, tässä tapauksessa `TYPE`, jolla korvataan vanhat taulukon talletusmuotoihin liittyvät `int` arvot.

dynaaminen.TaulukkoGen.java -esimerkki dynaamisesta taulukosta

```
/**
 * Esimerkki dynaamisesta taulukosta Java 1.5:n geneerisyyttä
 * ja "autoboxingia" käyttäen.
 * @author Vesa Lappalainen @version 1.0, 02.03.2002
 * @version 1.1, 01.03.2005
 * @author Santtu Viitanen @version 1.2, 21.07.2011
 * @param <TYPE> Tyyppi jota talletetaan
 * @example
 * <pre name="test">
 *  #THROWS TaulukkoGen.TaulukkoTaysiException
 *  TaulukkoGen<Integer> luvut = new TaulukkoGen<Integer>();
 *  Integer oma = new Integer(7);
 *  luvut.lisaa(0); luvut.lisaa(2);
 *  luvut.lisaa(99);  // Tekee oikeasti luvut.lisaa(new Integer(99));
 *  luvut.lisaa(oma);
 *  luvut.toString() === "0,2,99,7,";
 *  luvut.set(1,4);
 *  luvut.set(3,88);
 *  oma = luvut.get(3);
 *  luvut.toString() === " 0,4,99,88,";
 *  int luku = luvut.get(2);  // oik: luku = (Integer)(luvut.get(2)).intValue();
 *  luku === 99;
 *  luvut.lisaa(4);
 *  luvut.lisaa(4); #THROWS TaulukkoGen.TaulukkoTaysiException
 *
 *  TaulukkoGen<String> sanat = new TaulukkoGen<String>();
 *  sanat.lisaa("kissa");
 *  sanat.lisaa("koira");
 *  sanat.toString() === " kissa koira";
 * </pre>
 */
public class TaulukkoGen<TYPE> {
  public static class TaulukkoTaysiException extends Error {… }

  private TYPE alkiot[];
  private int lkm;

  public TaulukkoGen() {
    this(5);
  }

  public TaulukkoGen(int koko) {
    alkiot = (TYPE [])new Object[koko];
  }

  public void lisaa(TYPE i) throws TaulukkoTaysiException { … }

  public String toString() { … }

  public void set(int i, TYPE alkio) throws IndexOutOfBoundsException { … }

  public TYPE get(int i) throws IndexOutOfBoundsException { … }
}
```

Hieman geneerisyyden kaltainen toiminnallisuus saavutettiin ennen 1.5 päivitystä hieman työläästi `Object`-luokan taulukolla ja eksplisiittisillä tyyppimuunnoksilla (*casting*). Tämä on kuitenkin huomattavan vaivalloinen tapa ohjelmoida.

```
Object mjono = new String("df");
System.out.println(((String)mjono).length());
```

Konepellin alla Javan geneerisyys perustuu pitkälti samaan ideaan, mutta nyt tyyppimuunnokset tehdäänkin tietorakenteen sisällä, kuten edellisessä esimerkissä

```
public TaulukkoGen(int koko) {
  alkiot = (TYPE [])new Object[koko];
  // alkiot = new TYPE[koko];
}
```

Geneerisyyden kääntöpuolena sen sisäinen toteutus antaa tallentaa ainoastaan oliomuotoista dataa. Ohjelmoijalle päin kyllä näyttää, että `Integer`-taulukolle on mahdollista antaa `int`-arvo, mutta todellisuudessa sille viedäänkin aina uusi `Integer`-olio. Vastaavasti kun tietorakenteesta haetaan `int`-muuttujaan tallennettava arvo, niin sijoituksen tulos saadaan `intValue()`-metodista. Raa'assa numeronmurskauksessa joudutaan nyt suorittamaan työläitä *autoboxing* (`int` → `Integer`, `Integer` → `int`) operaatioita. Paljon laskentatehoa vaativaa toiminnallisuutta varten on siis hyvä tietää millaista tietorakennetta kannattaa käyttää.

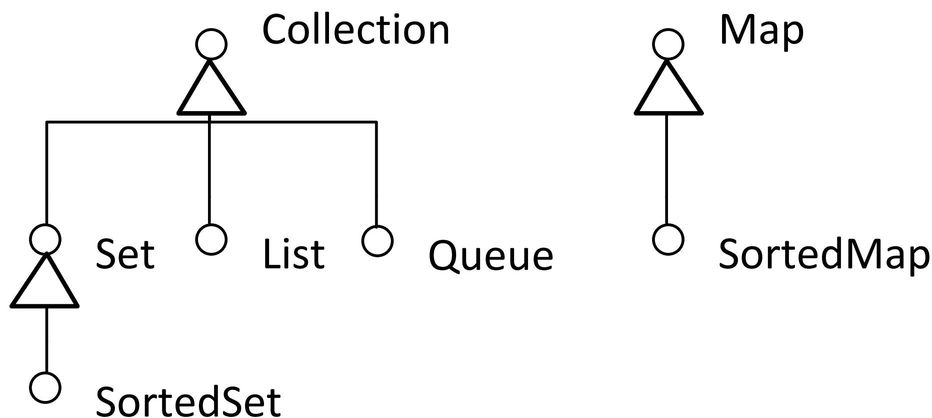
13.5 Javan *Collections Framework*

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.

Tietorakenteiden arkkitehtuuri on varsin monimutkainen kokoelma erilaisia rajapintoja, toteutuksia ja algoritmeja. TaulukkoGen.javan kaltaista toiminnallisuutta ei yleensä kannata luoda itse, vaan käyttää valmista ratkaisua, jolle on jo olemassa joukko valmiiksi optimoituja algoritmeja. Tärkeintä onkin tietää, minkälaista tietorakennetta ongelmaan kannattaa soveltaa.

Taulukkopohjaiset rakenteet, kuten `ArrayList` ja `Vector` soveltuvat hyvin tapauksiin, jossa luetaan alkioita indeksien avulla ja lisäys/poisto-operaatiot tehdään taulukon loppuun. Toisaalta jos haluttuun toiminnallisuuteen kuuluu useita alkioiden poistoja ja lisäyksiä satunnaiseen kohtaa tietorakennetta olisi `LinkedList` parempi vaihtoehto.



Kuva 13.1 Javan tärkeimmät tietorakennerajapinnat

13.5.1 ArrayList ja geneerisyys

Seuraavassa on Taulukko.javaa vastaava geneerinen esimerkki toteutettu `ArrayList`-rakenteella:

dynaaminen.ArrayListMalliGen.java –ArrayList-luokka

```
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Collection;

import fi.jyu.mit.ohj2.Tiedosto;

/**
 * Esimerkki Javan ArrayListin käytöstä Java 1.5:n geneerisyyden
 * ja "autoboxin" avulla.
 * @author Vesa Lappalainen @version 1.0, 02.03.2002
 * @version 1.1, 01.03.2005
 * @author Santtu Viitanen @version 1.2, 22.07.2011
 */

public class ArrayListMalliGen {

    public static void tulosta(OutputStream os, Collection<Integer> luvut) {
        PrintStream out = Tiedosto.getPrintStream(os);
        for (int luku : luvut) {
            out.print(luku + " ");
        }
        out.println();
    }

    public static void tulostaIter(OutputStream os, Collection<Integer> luvut) {
        PrintStream out = Tiedosto.getPrintStream(os);
        for (Iterator<Integer> it = luvut.iterator(); it.hasNext(); ) {
            int luku = it.next();
            out.print(luku + " ");
        }
        out.println();
    }

    public static void main(String[] args) {
        ArrayList<Integer> luvut = new ArrayList<Integer>(7);
        try {
            luvut.add(0);
            luvut.add(2);
            luvut.add(99);
        } catch (Exception 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);
        tulosta(System.out, luvut);
        try {
            luvut.set(21, 4);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Virhe: " + e.getMessage());
        }
    }
}
```

13.5.2 Iteraattori

Esimerkissä taulukon tulostus on tehty malliksi myös *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.

Oikeasti pinnan alla `for-each` on vain syntaktista karkkia (*syntactic sugar*) iteraattoritoteutukselle.

`ArrayList`in ja `Vector`in tapauksissa tietorakenne voitaisiin käydä läpi myös taulukkomaisesti,

```
  for (i=0; i<luvut.size();i++)
```

```
    out.print(luvut.get(i));
```

mutta tällöin tietorakenteen vaihtaminen esimerkiksi linkitetyksi listaksi vaatisi muutoksia `tulosta`-aliohjelmaan. Eli aina kun mahdollista, kannattaa välttää käyttämästä sitä tietoa, mikä tietorakenne on käytössä.

13.5.3 Geneerisyys

Alkuperäisissä Javan tietorakenteissa ei voitu taata mitenkään mitä tietorakenteeseen säilöttiin, vaan sille olisi voinut antaa sekaisin mitä tahansa olioita. Tällainen toiminnallisuus on useimmiten paitsi hyödytöntä, myös tietorakenteen käyttämistä rajoittavaa. Tähän tuli onneksi avuksi geneerisyys

Parametrilla `Collection<Integer>` taataan, että metodille tuodaan `Collection`-rajapinnan toteuttava tietorakenne, johon säilötty `Integer`-olioita. Nyt vältymme explisiittisten tyyppimuunnosten tekemiseltä, joka puolestaan antaa meille mahdollisuuden hyödyntää `for-each` rakennetta.

13.5.4 Algoritmit

Kun tietorakenteelta oletetaan tietty rajapinta, voidaan sille suorittaa sopiva algoritmi, esimerkiksi lajittelu, tietämättä tietorakenteen yksityiskohtia:

dynaaminen. AlgoritmiMalliGen.java – Collections-luokka

```
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 AlgoritmiMalliGen {

  /**
   * Luokka joka vertailee kahta kokonaislukuoliota ja
   * palauttaa niiden järjestyksen niin, että lajittelu menee
   * laskevaan järjestykseen.
   */
  public static class LaskevaInt implements Comparator<Integer> {
    public int compare(Integer o1, Integer o2) {
      return o2 - o1;
    }
  }

```

```
  public static void tulosta(OutputStream os,  Collection <Integer> luvut) {
    PrintStream out = Tiedosto.getPrintStream(os);
    for (Integer i : luvut ) {
      int luku = i;
      out.print(luku + " ");
    }
    out.println();
  }


  public static void main(String[] args) {
    ArrayList <Integer> luvut = new ArrayList <Integer> ();
    try {
      luvut.add(0);  luvut.add(2);
      luvut.add(99); luvut.add(7);
      luvut.add(22); luvut.add(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 = Collections.max(luvut);
    System.out.println("Suurin = " + suurin);          // Suurin = 99
    int pienin = Collections.min(luvut);
    System.out.println("Pienin = " + pienin);          // Pienin = 0
    pienin = Collections.max(luvut,new LaskevaInt());
    System.out.println("Pienin = " + pienin);          // Pienin = 0

    List<Integer> luvut2 = new LinkedList<Integer>();
    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

  }
}
```

Lisää Javan *Collections Framework*ista ja sen käyttämisestä

http://download.oracle.com/javase/tutorial/collections/intro/index.html

13.6 Tietovirta parametrina

Metodi `tulosta` on esitelty

```
public static void tulosta(OutputStream os,  Collection luvut) {
```

Näin voidaan tulostusvaiheessa valita mille laitteelle tulostetaan.

14. Funktio-olio

Mitä tässä luvussa käsitellään?

- Olioiden ja rajapintojen kertaus
- funktion vieminen parametrina
- Integraalin laskeminen numeerisesti

Luvun esimerkkikoodit:

```
https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/taulukot/
```

Tämän luvun tarkoitus on johdatella lukija ymmärtämään tarve välittää myös toimintoja (funktioita) parametrina. Itse asiassa nykyaikaiset graafiset ohjelmointikehykset perustuvat nimenomaan tähän ideaan. On tehty esimerkiksi valmis painike (*button*), mutta painikkeen tekijä ei tietenkään voi tietää mitä painikkeella pitäisi missäkin ohjelmassa tehdä. Siksi painikkeelle viedään tiedoksi tietyn rajapinnan toteuttava toiminto ja kun painiketta painetaan, kutsutaan tätä toimintoa ja näin painikkeesta on saatu yleiskäyttöinen. Tätä sanotaan tapahtumalähtöiseksi ohjelmoinniksi (*event driven*)

C#:issa voidaan tehdä samalla tavalla kuin nyt esitetään, mutta vastaava onnistuu myös helpommin delegaattien avulla.

Toinen tässä monisteessa käytettävä sovelluskohde on oikeellisuustarkistus. Jäsenen kentät ovat muuten keskenään hyvin samankaltaisia, mutta kullekin voi olla erilainen tapa tarkistaa milloin käyttäjän syöte on oikein.

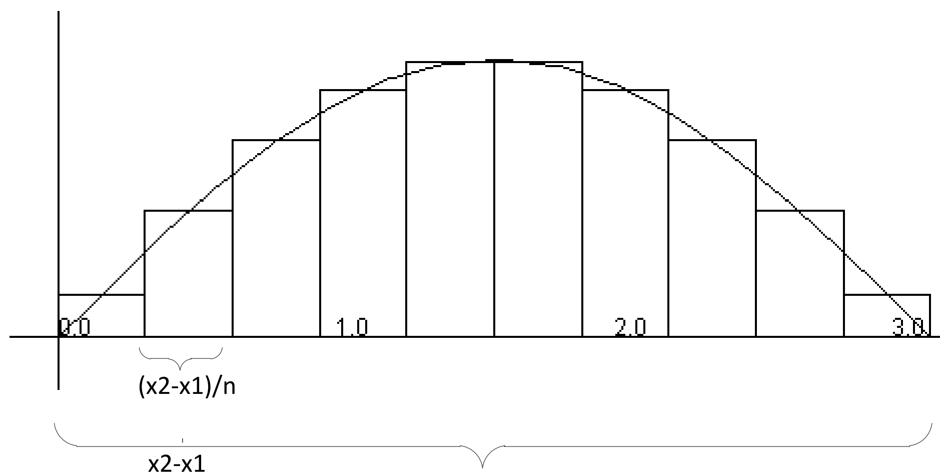
14.1 Numeerinen integrointi

Suoraan valmiin graafisen kehyksen toimintojen selittäminen on kohtuullisen iso operaatio. Otetaan siksi pienempi esimerkki, missä itse valittu toiminto (funktio) nousee selkeästi suuren roolin itse ongelman kannalta. Aloitetaan integroimalla yksi tietty funktio ja huomataan sitten, että suuri osa koodista olisi funktiosta riippumatonta. Sitten esimerkissä toteutetaan numeerinen integrointi myös muutamille muille funktiotyypeille.

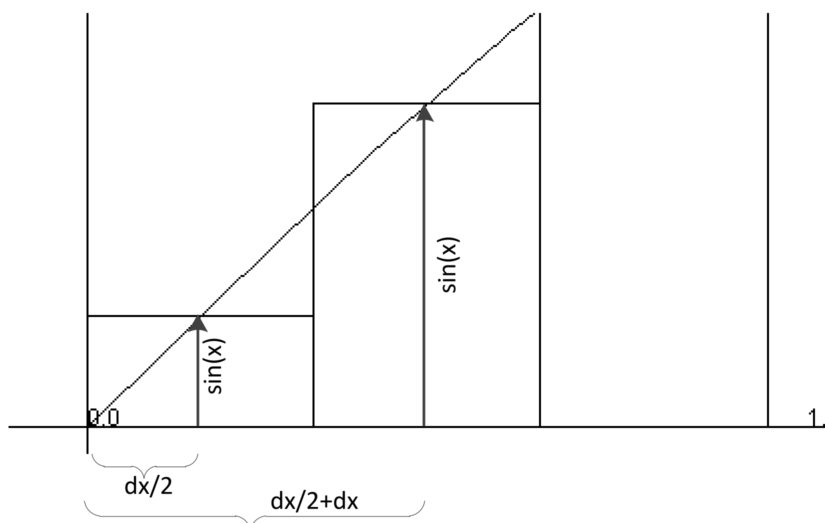
Luvun matematiikkaa ei kannata pelästyä, sillä kyseessä on varsin yksinkertainen toimenpide. Kyseessä on vain suorakulmioiden pinta-alojen yhteenlasku. Numeerinen integrointi onnistuu, kun jaetaan funktion alapuoli pylväisiin, niin että yksittäisen pylvään korkeus on yhtä suuri kuin funktion arvo pisteessä x. Lopulta kaikkien pylväiden yhteenlaskettu pinta-ala lasketaan. Mitä enemmän pylväitä välille laitetaan, niin sitä tarkempi tulos saadaan, mutta vastaavasti joudutaan tekemään enemmän laskutoimituksia.

14.1.1 Sinifunktion integrointi

Tutustutaan ensin ongelmaan tavallisen sinifunktion avulla, mutta yleistetään myöhemmin ohjelma toimimaan myös muilla funktioilla.



Kuva 14.1 Sinifunktion numeerinen integrointi



Kuva 14.2 Sinifunktion numeerinen integrointi

Muuttuja `summa` sisältää kaikkien laatikoiden yhteenlasketut pinta-alat. Laatikoiden alojen lasku onnistuu, kunhan tiedetään yksittäisen laatikon leveys, jota koodissa merkitään `dx` ja korkeus, joka puolestaan on funktion arvo pisteessä x.

funktio.integroi.Integroi.java – Sinifunktion numeerinen integrointi

```
import static java.lang.Math.*;

/**
 * Ohjelmalla integroidaan numeerisesti funktio sin(x)
 * @author Vesa Lappalainen
 * @version 1.0, 25.03.2003
 */
public class Integroi {

    /**
     * Integroidaan sin(x) välillä x1-x2
     * @param x1 alkuarvo
     * @param x2 loppuarvo
     * @param tiheys monellako askeleella
     * @return likiarvo integraalille
     */
```

```
    public static double integroi(double x1, double x2, int tiheys) {
        double summa = 0;
        double dx = (x2-x1)/tiheys;
        for ( double x=dx/2; x < x2; x += dx)
            summa += sin(x) * dx;
        return summa;
    }

    public static void main(String[] args) {
        double ala = integroi(0,PI,10000000);
        System.out.printf("%17.15f%n",ala);
    }
}
```

14.2 Numeerinen integrointi ja rajapinta

Käyttömahdollisuudet pelkkiä sinifunktioiden pinta-aloja laskevalle metodille ovat tietysti hyvin rajalliset. Laajennetaan metodia yleiskäyttöiseksi rajapintojen ja perinnän avulla.

Rajapinnan avulla määritellään metodi, jolle tuodaan parametrina reaaliluku ja joka palauttaa reaaliluvun.

funktio.integroi.Integroi2.java – funktio-oliot, rajapinta

```
    /**
     * Rajapinta kaikille funktiolle R->R
     */
    public interface FunktioRR {
        /**
         * @param x piste jossa lasketaan
         * @return funktion arvo pisteessä
         */
        public double f(double x);
    }
```

Irrotetaan nyt sinifunktion toteutus `FunktioRR`-rajapinnan (funktio joka kuvaa reaaliluvun toiseksi reaaliluvuksi) toteuttavaan luokkaan. Kysyttäessä `Sin`-luokan instanssilta funktion arvoa on pisteessä x, välitetään pyyntö ja palautetaan arvo `Math`-kirjaston `sin`-metodilta.

funktio.integroi.Integroi2.java – funktio-oliot, funktio

```
    static class SinFun implements FunktioRR {
        public double f(double x) { return Math.sin(x); }
    }
```

Nyt `integroi`-metodiin tuodaan uutena parametrina `FunktioRR`-rajapinnan toteuttava olio(viite) `f`. Itse integraalin laskeminen tapahtuu yhä samalla tapaa, eli ainoa muutos on vaihtaa `sin`–funktion viitteet oliolta `f` arvon kysyviksi lauseiksi.

funktio.integroi.Integroi2.java – funktio-oliot, integrointi

```
    /**
     * Integroidaan sin(x) välillä x1-x2
     * @param f integroitva funktio-olio
     * @param x1 alkuarvo
     * @param x2 loppuarvo
     * @param tiheys monellako askeleella
     * @return likiarvo integraalille
     */
    public static double integroi(FunktioRR f,double x1, double x2, int tiheys) {
```

```
        double summa = 0;
        double dx = (x2-x1)/tiheys;
        for ( double x=dx/2; x < x2; x += dx)
            summa += f.f(x) * dx;
        return summa;
    }

    public static void main(String[] args) {
        SinFun sin = new SinFun();
        double ala = integroi(sin,0,PI,10000);
        System.out.printf("%17.15f%n",ala);
    }
```

14.3 Rajapinnan käyttäminen

Tehtyä rajapintaa voidaan nyt hyödyntää myös muilla funktiotyypeillä. Tehdään luokka toisen asteen polynomiyhtälöiden ($ax^2 + bx + c = 0$) integroimista varten.

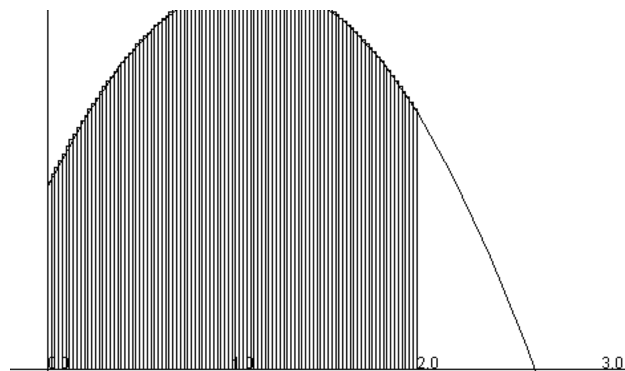
funktio.integroi.Integroi2.java – Toisen asteen polynomi

```
    public static class P2 implements FunktioRR {
        private double a,b,c;

        public P2() {
            a = 1;
        }

        public P2(double a, double b, double c) {
            this.a = a;
            this.b = b;
            this.c = c;
        }

        public double f(double x) { return a*x*x + b*x + c; }
    }
```

Toisin kuin `SinFun`-luokasta, tulee `P2`-luokasta hieman monimutkaisempi. Tarvitaan mm. konstruktori, jotta parametrina voidaan viedä polynomille kertoimet.

Luokkaa voidaan käyttää esimerkiksi seuraavasti.

```
        P2 p2 = new P2(-0.9,2,1);        // -0.9^2 +2x +1
        double ala = integroi(p2,0,2,100);
        System.out.printf("%17.15f%n",ala);
```



Kuva 14.3 Polynomin numeerinen integrointi

15. 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:

https://svn.cc.jyu.fi/srv/svn/ohj2/moniste/esimerkit/src/tiedosto/

Pyrimme seuraavaksi lisäämään kerho–ohjelmaamme tiedostosta lukemisen ja tiedostoon tallettamisen. Tätä varten tutustumme ensin lukemiseen mahdollisesti liittyviin ongelmiin.

15.1 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.

15.1.1 Lukeminen

Aikanaan C-ohjelman takia muutettiin hieman alkuperäistä suunnitelmaa jäsenrekisteritiedoston sisällöstä:

```
Kelmien kerho ry
100
; Kenttien järjestys tiedostossa on seuraava:
id| nimi         |hetu       |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  |           ...
```

Silloin lisättiin rivi, jossa kerrotaan tiedoston maksimikoko. Tätä tarvittiin jäsenlistan luomisessa jotta listasta saataisiin heti mahdollisimman oikean kokoinen. Vaikka tällä tiedolla ei ole enää mitään merkitystä, on se esimerkeissä mukana, jotta tiedostot olisivat myös vanhoilla C-ohjelmilla luettavissa. Omiin uusiin ohjelmiin tätä muutosta ei kannattane tehdä.

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.

15.2 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.TiedKaScanner.java - Lukujen lukeminen tiedostosta

```
package tiedosto;
import java.util.*;
import java.io.*;
```

```java
import fi.jyu.mit.ohj2.Mjonot;
/**
 * Lukujen lukeminen tiedostosta Scanner-luokkaa käyttäen
 * @author Vesa Lappalainen
 * @version 1.0, 05.03.2007
 */
public class TiedKaScanner {

  /** @param args ei käytössä */
  public static void main(String[] args)  {

    Scanner fi;

    try {  // Avataan tiedosto lukemista varten
      // fi = new Scanner(new File("luvut.dat"));
      // Jotta UTF8/ISO-8859 toimii:
      fi = new Scanner(new FileInputStream(new File("luvut.dat")));
    } catch (FileNotFoundException ex) {
      System.out.println("Tiedosto ei aukea!");
      return;
    }

    double summa=0;
    int n=0;

    try {
      String s; double luku;
      while ( fi.hasNext() ) {
        s = fi.next();
        try {
          luku = Double.parseDouble(s);
        } catch (NumberFormatException ex) {
          continue;
        }
        summa += luku;
        n++;
      }
    } finally {
      fi.close();
    }

    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 15.1 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.

15.2.1 Tiedoston avaaminen muodostajassa

Tiedosto voidaan siis avata heti kun tiedostoa vastaava tietovirta luodaan:

```java
fi = new ScannerF("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.

15.2.2 Tiedostosta lukeminen.

Tiedostosta lukeminen on jälleen analogista päätesyötön kanssa:

```
s = fi.next();
```

Mikäli tiedosto on loppu, saa `s` `null`-arvon.

15.2.3 Tiedoston lopun testaaminen

Helpoin ratkaisu on perustaa lukusilmukka siihen, että kysytään `hasNext`-metodilla sisältääkö tiedosto vielä uutta riviä.

```
while ( fi.hasNext() ) {
  ... käsittele jonoa s
}
```

15.2.4 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 parametrina 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).

15.2.5 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
      }
    }
```

Tiedosto kannattaa sulkea heti kun sen käyttö on loppu.

Tehtävä 15.2 Kommentit näytölle

Kirjoita ohjelma, joka kysyy tiedoston nimen ja tämän jälkeen tulostaa tiedostosta rivien `/*******` ja `------*/` välisen osan näytölle.

15.3 Tiedoston yhdellä rivillä monta kenttää

Jäsenrekisterissä on tiedoston yhdellä rivillä useita kenttiä. Kentät saattavat olla myös eri tyyppisiä. Miten lukeminen hoidetaan varmimmin?

15.3.1 Ongelma

Olkoon meillä vaikkapa seuraavanlainen 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 15.3 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>
```

15.4 Merkkijonon paloittelu

Tutkitaanpa ongelmaa tarkemmin. Tiedostosta on siis luettu rivi, joka on muotoa

```
| |V|o|l|v|o| | | | |1|2|3|0|0| | | |1|
```

Jos saisimme erotettua tästä 3 merkkijonoa:

```
pala1              pala2               pala3
| |V|o|l|v|o| |    | | |1|2|3|0|0| |   | |1|
```

voisimme kustakin palasesta erikseen ottaa haluamme tiedot. Esimerkiksi 1. palasesta saadaan tuotteen nimi, kun siitä poistetaan turhat välilyönnit `trim` -metodilla. Lukujen käsittely ei kuitenkaan ole aivan näin yksinkertaista.

15.4.1 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
```

15.4.2 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;
  }

}
```

15.4.3 Esimerkki `erota`-funktion käytöstä

Kirjoitetaan lyhyt esimerkki, jolla demonstroidaan funktion käyttöä:

tiedosto.ErotaEsim.java - esimerkki erota-funktion käytöstä

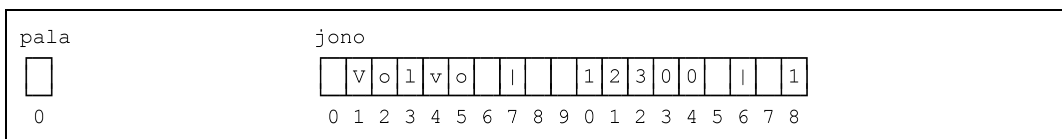
```java
import fi.jyu.mit.ohj2.Mjonot;

/**
 * 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:

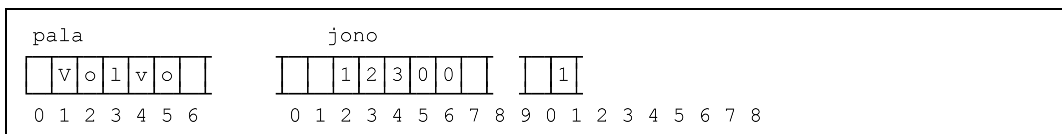
```
0: pala = ''          jono = ' Volvo |  12300 | 1'
1: pala = ' Volvo '   jono = '  12300 | 1'
2: pala = '  12300 '  jono = ' 1'
3: pala = ' 1'        jono = ''
4: pala = ''          jono = ''
```

15.4.4 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:

```
pala                    jono
[ | |1|2|3|0|0| ]       [ |1]
0 1 2 3 4 5 6           0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
```

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:

```
pala                    jono
[ |1]                   [ ]
0 1 2 3 4 5 6           0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
```

Vastaava toistuu neljännessä kutsussa, eli koko jono sitten kopioidaan paluuarvoksi ja tilanne on neljännen kutsun jälkeen:

```
pala                    jono
[ ]                     [ ]
0 1 2 3 4 5 6           0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
```

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);
}
```

15.4.5 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());
}
```

15.5 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
-------------------------------------------
```

15.5.1 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

```java
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");
    }
  }
}
```

15.6 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 15.4 Tietorakenne

Piirrä kuva `Tuotteet` –luokan tietorakenteesta.

Tehtävä 15.5 Perintä

Miten voisit perinnän avulla saada tiedoston LueRek.java luokasta `Tuote` tiedoston LueTRek.java vastaavan luokan (tietysti eri nimelle, esim. `RekTuote`). Mitä muutoksia olisi hyvä tehdä alkuperäisessä `Tuote`–luokassa.

Tehtävä 15.6 Tunnistenumero

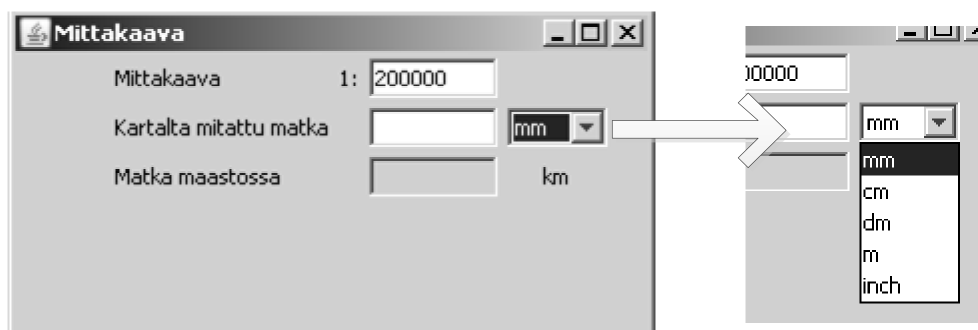
Lisää LueTRek.java–ohjelmaan tunnistenumeron käsittely mahdollista tulevaa relaatiokäyttöä varten.

Tehtävä 15.7 Graafinen mittakaava

Kirjoita mittakaavaohjelma, jossa on vakiotaulukko

yks	mm
mm	1.0
cm	10.0
dm	100.0
m	1000.0
inch	25.4

Alasvetovalikosta voi valita millä yksiköllä matka on mitattu kartalta. Muunnos tehdään aina kilometreiksi kahden desimaalin tarkkuudella. Mieti miten saat toimimaan ilman erillistä *Laske* –nappia, eli suorittamaan muunnokset automaattisesti aina kun jotain arvoa muutetaan.



Kuva 17.1 Esimerkkikuva mittakaavaohjelmasta

Tehtävä 15.8 Graafinen mittakaava ja luku tiedostosta

Muuta ohjelmaa siten, että yksiköiden muunnostaulukko luetaan ohjelman aluksi tiedostosta `muunnos.dat`.

15.7 Kerhon talletukset

Kerho-ohjelmassa talletusvastuut kannattaa jakaa niin, että `Kerho`-luokka määrää `Jasenet` ja `Harrastukset`-luokat tekemään omat lukemiset ja talletukset. Sitten esim. `Jasenet`-luokka avaa tiedoston kirjoittamista varten ja sitten pyytää jokaiselta jäseneltä erikseen merkkijonona tiedostoon talletettavan muodon. Vastaavasti tiedon lukemisessa `Jasenet`-luokka avaa tiedoston kirjoittamista varten ja sitten lukee tiedostosta rivi kerrallaan. luo uuden jäsenen ja antaa rivin jäsenelle käsiteltäväksi ja jäsenen kenttiin laitettavaksi (parse).

16. Kerho-ohjelman rakenne

Mitä tässä luvussa käsitellään?

- Jäsenen ja käyttöliittymän välinen keskustelu
- Kenttien lisääminen käyttöliittymään
- Tietorakenteeseen lukeminen
- Oikeellisuustarkastukset
- Kentistä hakeminen
- Säännölliset lausekkeet

16.1 Jäsen ja kentät

Jäsenen käyttöliittymää voitaisiin lähteä tekemään niin, että erikseen rakennetaan tiedot siitä, että jäsenessä on nimi, henkilötunnus jne. Hyvin nopeasti huomataan että tämä tie johtaa toistuvaan koodiin ja vaikeaan ylläpitoon. Seuraavaksi mietitäänkin, miten voitaisiin yleistää jäsenen tiedot niin, että käyttöliittymän ei tarvitsekaan tietää jäsenen tietojen yksityiskohtia.

16.1.1 Algoritmi näytön ja jäsenen keskustelulle

Lisäämällä byrokratiaa näytön ja jäsenen välille, voidaan näyttö pitää tietämättömänä siitä, mitä kenttiä jäsenessä todella on. Minkälaista byrokratiaa? Esimerkiksi "keskustelu" näytön ja jäsenen välillä voisi olla seuraavanlainen komentoriviliittymää varten:

```
1.  Näyttö:  montako kenttää sinulla on jäsen?
2.  Jäsen:   13 kenttää
3.  Näyttö:  no annappa minulle 1. kenttä merkkijonona!
4.  Jäsen:   Ankka Aku
5.  Näyttö:  Milläs kysymyksellä tämä kenttä kysytään?
6.  Jäsen:   Jäsenen nimi
7.  Näyttö kysyy käytäjältä Jäsenen nimi (Ankka Aku) >  => jono
8.  Näyttö tutkii vastattiinko q tms. erikoismerkki, jos niin pois
9.  Näyttö:  Sijoitapa jäsen tämä jono 1. kentäksi.
10. Näyttö jatkaa kohdasta 3 mutta kentälle 2 kunnes kaikki 13 kenttää käsitelty
```

Vastaavasti graafisessa liittymässä idea voisi olla seuraava

```
1. Näyttö:  montako kenttää sinulla on jäsen?
2. Jäsen:   13 kenttää
3. Näyttö silmukassa 1..13:
3.1  Näyttö: Anna kenttään i tarvittava otsikkoteksti
3.2  Jäsen:  palauttaa merkkijonon (esim. nimi tai hetu)
3.3  Näyttö: luo lomakkeelle syöttökentän jonka vieressä ko merkkijono
3.4  Näyttö: anna kentän i sisältö merkkijonona
3.5  Jäsen:  palauttaa merkkijonon (esim. Ankka Aku tai 030451-111A)
3.6  Näyttö: laittaa merkkijonon syöttökentän oletusarvoksi
4. Näyttö: jää odottamaan kun käyttäjä täyttää lomaketta
4.1 Näyttö: kun käyttäjä muuttanut jotakin syöttöruutua (i), voi kysyä
            jäseneltä että onko i:een kenttään oikea arvo laittaa käyttäjän
            antaman jono (oikeellisuustarkistus), jos ei valitetaan
5. Näyttö: kun käyttäjä kuittaa lomakkeen, annetaan syötetyt arvot yksi
   kerrallaan jäsenelle
```

16.2 Näytön ja jäsenen välinen rajapinta

Edellisestä algoritmista huomaamme, että jäseneen kannattaa tehdä rajapinta, jossa on esimerkiksi metodit:

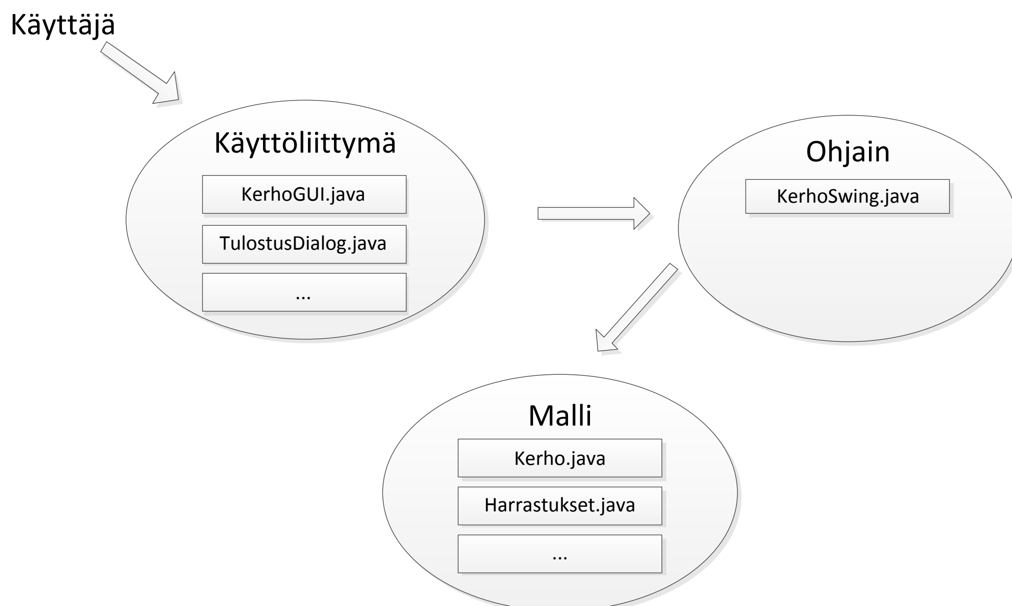
```
int getKenttia()         - monta kenttää on jäsenellä
int ekaKentta()          - 1. mielekäs kenttä kysyttäväksi (mm. id:tä ei kysytä)
String getKysymys(int k) - antaa k:n kentän kysymiseksi tarvittavan tekstin
String anna(int k)       - palauttaa k:n kentän sisällön merkkijononona
String aseta(int k, String jono) - asettaa k:n kentän arvoksi jonon ja palauttaa
                           null jos jono on hyvä, muuten virheilmoituksen
                           tekstinä joka voidaan tulostaa käyttäjälle
```

Käytännön toteutus noille metodeille voi perustua yksinkertaisimmillaan ihan `switch`-lauseisiin. Esimerkkikoodissa `Harrastus`-luokka on toteutettu näin. Ylläpidettävämmin tehtynä toteutus voi perustua taulukoihin polymorfisista kenttätyypeistä. Tästä on esimerkki `Jasen`-luokan toteutuksessa. Esimerkistä tosin vielä puuttuu yhteinen rajapinta, jonka `Jasen` ja `Harrastus` voisivat toteuttaa. Nyt ihannetilanteessa päästään siihen, että jos jäsenen kenttien lukumäärää pitää muuttaa, tapahtuu tämä muutos vain `Jasen`-luokassa eikä mihinkään muualle (edes käyttöliittymään) tarvitse koskea koodissa.

16.3 Kerhon rakenne

Kerhon arkkitehtuuri voidaan jakaa karkeasti kolmeen osaan. Alimmalla tasolla toimivat käyttäjälle näkymättömät asiat, kuten tietorakenteet ja tiedon tallennus. Käyttäjä itse pääsee vaikuttamaan ainoastaan ylimpään kerrokseen, jossa on ohjelmoituna näkyvä käyttöliittymä ja otetaan esimerkiksi kiinni käyttäjän aiheuttamia tapahtumia. Näiden kahden kerroksen välissä toimii kontrolleri, joka sisältää käyttöliittymän tarvitsemaa toiminnallisuutta ja toimii välittäjänä kahden kerroksen välillä.

Ohjelmoinnin yhteydessä törmää usein ohjelmistoarkkitehtuurin käsitteeseen. Kerhollakin on paljon yhteistä MVC (*Model-View-Controller*)-arkkitehtuurista kehittyneen MVP (*Model-View-Presenter*)-mallin kanssa. Arkkitehtuurien tarkoituksena on jakaa ohjelma helposti hallittaviin osa-alueisiin, mutta usein niitä ei ole edes mielekästä noudattaa mitenkään kirjaimellisesti.



Kuva 15.1 Kerhon arkkitehtuuri

16.3.1 KerhoSwingin ja käyttöliittymän yhteistoiminta

Kerhossa on pyritty erottamaan suurin osa käyttöliittymän toiminnallisuuteen liittyvästä koodista erilliseen `KerhoSwing` apuluokkaan.

Vastaan tulee tietysti tilanne, että `KerhoSwing` sisältää metodin, jolla halutaan päivittää jotain tiettyä käyttöliittymän komponenttia. Esimerkiksi kuvassa jäsenet sisältävä `JList`. Ensin `KerhoSwing` luokassa esitellään halutun tyyppinen komponentti, jonka jälkeen sille luodaan `get`- ja `set`-metodit. Mikäli käytössä on *Eclipse*, niin metodien luominen onnistuu myös automaattisesti, kun painaa koodissa hiiren oikealla napilla komponentin nimen päällä → Source → Generate Getters and Setters.

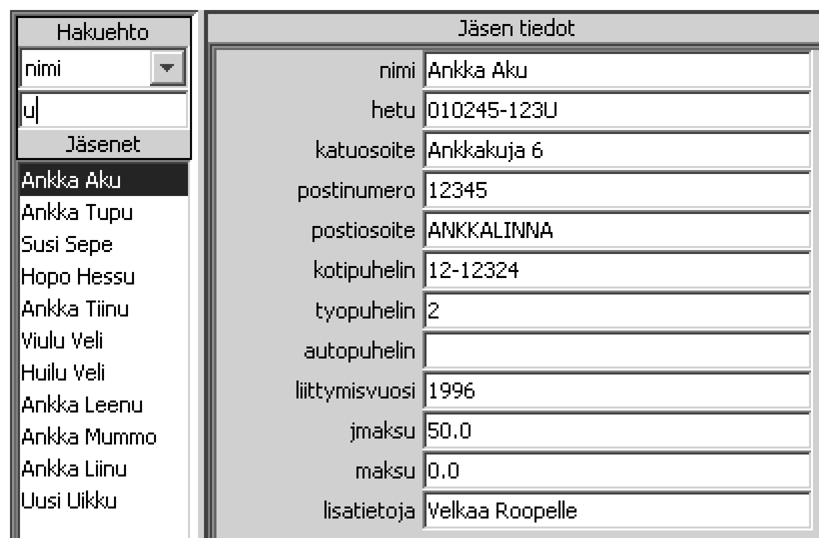
KerhoSwing.java – Harjoitustyö vaihe 7 – get ja set-metodit

```
    private JList listJasenet;
...
    public JList getListJasenet() {
        return listJasenet;
    }

    public void setListJasenet(JList listJasenet) {
        this.listJasenet = listJasenet;
    }
```

Nyt käyttöliittymäluokan `KerhoSwing`-komponentin ominaisuuksista löytyy juuri luotu listJasenet-kenttä, jonka voi nyt kytkeä haluttuun käyttöliittymän omaan `JList`-komponenttiin.

16.4 Jäsenien selaaminen

Jäsenien käsittelyä helpottamaan tehdään ohjelman vasempaan laitaan jäsenlistaus. Käyttäjiä voidaan myös suodattaa halutun hakuehdon perusteella.



Kuva 15.2 Jäsenlistaus ja suodatus

16.5 Jäsenkenttien lisääminen

Tehtyyn `EditPanel`-komponenttiin lisättiin suunnitteluvaiheessa kenttien otsikot käsin. Nyt kuitenkin otsikot ovat jo `Jasen`-olion tiedossa ja samaa ohjelmakoodia joudutaan kirjoittamaan useaan paikkaan. Kannattavampaa on tehdä toteutus, jossa kentät haetaan suoraan oliolta. Tällöin helpottuu paitsi muutosten tekeminen ohjelmaan, myös ohjelman siirto toisiin järjestelmiin. Samalta syystä kannattaa olion tehtäväksi antaa myös oikeellisuustarkistukset.

Jäsen tiedot	
nimi	Ankka Aku
hetu	010245-123U
katuosoite	Ankkakuja 6
postinumero	12345
postiosoite	ANKKALINNA
kotipuhelin	12-12324
työpuhelin	2
autopuhelin	
liittymisvuosi	1996
jäsenmaksu	50.00
maksettumaksu	0.00
lisätietoja	Velkaa Roopelle

Kuva 15.3 Jäsenen kentät

Täydelliset esimerkit osoitteessa

```
http://users.jyu.fi/~vesal/ohj2/vaihe7/
```

`Jasen`-luokasta löytyy kentät taulukko, johon voidaan tallentaa `Kentta`-rajapinnan toteuttavia olioita.

Jasen.java – Harjoitustyö vaihe 7

```
…
private Kentta kentat[] = {  // valitettavasti ei voi olla final vaikka pitäisi,
clone estää tämän :-(
                new IntKentta("id"),
                new JonoKentta("nimi"),
                new HetuKentta("hetu",new HetuTarkistus()),
                new JonoKentta("katuosoite"),
                new JonoKentta("postinumero",
                        new SisaltaaTarkistaja(SisaltaaTarkistaja.NUMEROT)),
                new JonoKentta("postiosoite"),
                new PuhelinKentta("kotipuhelin"),
                new PuhelinKentta("työpuhelin"),
                new PuhelinKentta("autopuhelin"),
                new IntKentta("liittymisvuosi"),
                new RahaKentta("jäsenmaksu"),
                new RahaKentta("maksettumaksu"),
                new JonoKentta("lisätietoja")
    };
…
```

Rajapinnalla taataan, että käyttöliittymästä saa selville ja pystyy asettamaan oliolle kenttään liittyviä tietoja.

```
Kentta.java - Harjoitustyö vaihe 7
```

```java
public interface Kentta extends Cloneable {

    /**
     * kentän arvo merkkijonona.
     * @return kenttä merkkkijonona
     */
    String toString();

    /**
     * Palauttaa kentään liittyvän kysymyksen.
     * @return kenttän liittyvä kysymys.
     */
    String getKysymys();

    /**
     * Asettaa kentän sisällön ottamalla tiedot
     * merkkijonosta.
     * @param jono jono josta tiedot otetaan.
     * @return null jos sisältö on hyvä, muuten merkkijonona virhetieto
     */
    String aseta(String jono);


    /**
     * Palauttaa kentän tiedot veratiltavana merkkijonona
     * @return vertailtava merkkijono kentästä
     */
    String getAvain();

    Kentta clone() throws CloneNotSupportedException ;

}
```

Mitä oikeastaan halutaan tallentaa? Osa kentistä on selkeitä merkkijonoja ilman täsmällisesti määriteltyjä muotoja (nimi, osoite, postiosoite…). Toisaalta löytyy myös kokonaislukuja, desimaalilukuja, puhelinnumeroita ja henkilötunnus. Jokaista luokkaa ei tietenkään kannata kirjoittaa alusta asti toteuttamaan rajapintaa, vaan ensin luodaan kaikille yhteinen kantaluokka. Nyt kentillä ei kuitenkaan ole mitään selkeää vanhempaa, joka toteuttaisi kaikki rajapinnan metodit, vaikka osa niistä onkin jo toteutettavissa.

Java tarjoaa rajapinnan ja luokan välimaastoon rakenteen nimeltä `abstract class`. Se on luokka josta ei voi suoraan luoda instanssia, mutta johon on mahdollista ohjelmoida periville luokille valmiiksi osa toiminnallisuutta ja metodeita. Perivä luokka myös laajentaa (`extends`) abstraktin luokan, eikä toteuta (`implements`). Suurimpana erona on että luokka voi toteuttaa useita rajapintoja, mutta ei periä. Tästä syystä abstraktia luokkaa ei tule pitää vaihtoehtona rajapinnalle.

http://download.oracle.com/javase/tutorial/java/IandI/abstract.html

```
PerusKentta.java - Harjoitustyö vaihe 7
```

```java
/**
 * Peruskenttä joka implementoi kysymyksen käsittelyn
 * ja tarkistajan käsittelyn.
 *
 * @author Vesa Lappalainen
 * @version 1.0, 22.02.2003
 * @version 1.3, 02.04.2003
 */
public abstract class PerusKentta implements Kentta { // NOPMD
    private final String kysymys;

    /**
     * Yleisen tarkistajan viite
```

```
     */
    protected Tarkistaja tarkistaja = null;

    /**
     * Alustetaan kenttä kysymyksen tiedoilla.
     * @param kysymys joka esitetään kenttää kysyttäessä.
     */
    public PerusKentta(String kysymys)  { this.kysymys = kysymys; }

    /**
     * Alustetaan kysymyksellä ja tarkistajalla.
     * @param kysymys joka esitetään kenttää kysyttäessä.
     * @param tarkistaja tarkistajaluokka joka tarkistaa syötän oikeellisuuden
     */
    public PerusKentta(String kysymys,Tarkistaja tarkistaja) {
        this.kysymys = kysymys;
        this.tarkistaja = tarkistaja;
    }

    /**
     * @return kentän arvo merkkijonona
     * @see kanta.Kentta#toString()
     */
    @Override
    public abstract String toString();

    /**
     * @return Kenttää vastaava kysymys
     * @see kanta.Kentta#getKysymys()
     */
    @Override
    public String getKysymys() {
        return kysymys;
    }

    /**
     * @param jono josta otetaan kentän arvo
     * @see kanta.Kentta#aseta(java.lang.String)
     */
    @Override
    public abstract String aseta(String jono);

    /**
     * Palauttaa kentän tiedot veratiltavana merkkijonona
     * @return vertailtava merkkijono kentästä
     */
    @Override
    public String getAvain() {
        return toString().toUpperCase();
    }


    /**
     * @return syväkopio oliosta
     */
    @Override
    public Kentta clone() throws CloneNotSupportedException {
        return (Kentta)super.clone();
    }


}
```

Luokkaan jätettiin vielä abstraktit metodit `toString` ja `aseta`, joten periviin luokkiin tulee kirjoittaa vähintään ne.

Tässä esiteltiin myös viite yleiselle `Tarkistaja`-rajapinnan toteuttavalle oliolle. Kerho käyttää tarkistajia kenttiin syötetyn tiedon validointiin, mutta tavallinen `PerusKentta` ei tietenkään voi vielä edes hyödyntää tällaista toiminnallisuutta, vaan mielekäs tapa siihen on jonkun perivän kentän `aseta`-metodissa.

Nyt voidaan lähteä toteuttamaan kenttien olioita. Helpoin tapaus on vapaamuotoinen merkkijono, johon käyttäjä saa tallentaa mitä merkkejä vain, mutta johon on kuitenkin mahdollista liittää erillinen `Tarkistaja` syötteen muodon validointia varten.

JonoKentta.java – Harjoitustyö vaihe 7

```
/**
 * Kenttä tavallisia merkkijonoja varten.
 * @author Vesa Lappalainen
 * @version 31.3.2008
 *
 */
public class JonoKentta extends PerusKentta {
    private String jono = "";

    /**
     * Alustetaan kenttä kysymyksen tiedoilla.
     * @param kysymys joka esitetään kenttää kysyttäessä.
     * @example
     * <pre name="test">
     *    JonoKentta jono = new JonoKentta("nimi");
     *    jono.getKysymys() === "nimi";
     *    jono.toString() === "";
     *    jono.aseta("Aku");
     *    jono.toString() === "Aku";
     * </pre>
     */
    public JonoKentta(String kysymys) { super(kysymys); }


    /**
     * Alustetaan kysymyksellä ja tarkistajalla.
     * @param kysymys joka esitetään kenttää kysyttäessä.
     * @param tarkistaja tarkistajaluokka joka tarkistaa syötön oikeellisuuden
     */
    public JonoKentta(String kysymys,Tarkistaja tarkistaja) {
        super(kysymys,tarkistaja);
    }

    /**
     * @return Palauetaan kentän sisältö
     * @see kanta.PerusKentta#toString()
     */
    @Override
    public String toString() { return jono; }

    /**
     * @param s merkkijono joka asetetaan kentän arvoksi
     * @see kanta.PerusKentta#aseta(java.lang.String)
     */
    @Override
    public String aseta(String s) {
        if ( tarkistaja == null ) {
            this.jono = s;
            return null;
        }

        String virhe = tarkistaja.tarkista(s);
        if ( virhe == null ) {
            this.jono = s;
            return null;
        }
        return virhe;
    }
}
```

Ohjelman rakenne menee jo jokseenkin monimutkaiseksi. Kuitenkin tilanne, jossa toiminnallisuus on sotkettu käyttöliittymään ja luokilla ei ole selkeitä vastuualueita johtaa

ennen pitkää ylläpitokelvottomaan koodiin. Tässä luokilla on selkeät vastuualueet, joten myös bugien paikallistaminen on helpompaa.

16.6 Oikeellisuustarkistukset

Oikeellisuustarkistukset voi hoitaa monellakin tapaa. Usein helpointa on käyttää säännöllisiä lausekkeita, mutta esimerkiksi Kehossakin tarvittavan henkilötunnukset tarkistaminen ei niillä suoraan onnistu. Kerhossa on käytetty myös `Tarkistaja` rajapinnan toteuttavia luokkia, joihin voi helpommin ohjelmoida monimutkaisempaakin toiminnallisuutta. Syötteiden validointi on varsin suoraviivaista työtä, jossa käytännössä tarvitaan vain käyttäjän antama merkkijono ja tulos. Määritellään yksinkertainen, mutta täsmällinen toiminnallisuus sen toteuttaville luokille.

Tarkistaja.java – Harjoitustyö vaihe 7

```java
/**
 * Rajapinta yleiselle tarkistajalle.
 * Tarkistajan tehtävä on tutkia onko annettu
 * merkkijono kelvollinen kentän sisällöksi ja jos on.
 * palautetaan null.
 * Virhetapauksessa palautetaan virhettä mahdollisimman
 * hyvin kuvaava merkkijono.
 * @author Vesa Lappalainen @version 31.3.2008
 */
public interface Tarkistaja {
    /**
     * Tutkitaan käykö annettu merkkijono kentän sisällöksi.
     * @param jono tutkittava merkkijono
     * @return null jos jono oikein, muuten virheilmoitusta vastaava merkkijono
     */
    String tarkista(String jono);
}
```

Tehtävä 15.1 Henkilötunnuksen tarkistaminen

Henkilötunnuksen muoto ja laskeminen noudattavat tarkkoja määrittelyjä. Tutustu aiheeseen esimerkiksi Wikipediasta ja katso Kerho-ohjelmassa `Tarkista`-rajapinnan toteuttava ratkaisu `HetuKentta.java.`

16.6.1 Säännölliset lausekkeet

Tehokas työkalu syötteiden tarkastamiseen on säännölliset lausekkeet (*regular expressions, regex*). Tietojenkäsittelytieteessä niitä on hyödynnetty jo yli 50 vuotta, eli tässäpä opeteltavaksi tekniikka, joka tuskin on heti vanhentumassa. Säännöllisen lausekkeen idea on määritellä lauseke, jota merkkijono joka vastaa tai ei vastaa.

Useimmissa suurissa ohjelmointikielissä on jo valmiina kirjastot säännöllisten lausekkeiden hyödyntämiseen. Eri toteutusten välillä on kuitenkin pieniä eroja, joten tarvittaessa kannattaa konsultoida manuaalia.

http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

Javassa yksinkertaisimmillaan säännöllinen lauseke ab vastaa merkkijonoa ab, mutta ei kuitenkaan merkkijonoa abc. Taulukossa lauseketta vastaavat merkkijonot ovat korostettuna.

regexp	merkkijono	Huomautus
ab	**ab** abc	Perustapaus

Koska säännölliset lausekkeet ovat oma merkkijonokielensä, niin ne sisältävät myös operaattoreita, joista tässä yleisimmät. Tavallisesti operaattoria verrataan edelliseen merkkiin.

regexp	merkkijono	Huomautus
ab*	**a** **ab** **abbb** aab	b esiintyy nollasta äärettömän monen kertaan
ab*	abc	Tähti ei kuitenkaan toimi jokerimerkkinä
ab+	a **ab** **abb**	+ toimii kuten tähti, mutta vaatii vähintään yhden esiintymän, eli täysin vastaava toiminnallisuuden saavuttaa myös lauseella abb*
ab?	**a** **ab** abb	b kerran tai ei kertaakaan
a|b	**a** **b** ab	a tai b
a|b*	ab **bbb**	a tai b*
[ab]	**a** **b** ab	a tai b
[ab]*	**b** **abaaab**	Operaattoria sovelletaan kaikkiin kirjaimiin sulkujen sisälllä
a.	**ab** abc	Piste korvaa minkä tahansa kirjaimen

Sulkujen avulla voi luoda lauseen sisälle pienempiä lausekkeita, joita käsitellään omina kokonaisuuksinaan ja joihin sovelletaan operaattoreita sellaisenaan.

regexp	merkkijono	Huomautus
(ab)+c	a ab ac **abc** abbc **abababc**	ab ja vähintään kerran ja c

Jos tarkastetaan varattua merkkiä (operaattorit), tai ”-merkkiä, niin käytetään niiden edessä kenoviivaa \, sekä kenoviivaa tarkastaessa toista kenoviivaa.

16.6.2 Säännöllisten lausekkeiden käyttäminen

Javan merkkijonoluokka sisältää useita metodeita, jotka hyödyntävät suoraan säännöllisiä lausekkeita. Näin tehdään myös Kerhon puhelinnumeroita käsittelevässä luokassa.

PuhelinKentta.java - Harjoitustyö vaihe 7

```
…
    @Override
    public String aseta(String jono) {
```

```
        if ( !jono.matches("[0-9\\-\\+ ]*")) return "Sallitaan vain merkit 0-9 -
+ ";
        return super.aseta(jono);
    }
}
```

Metodi tarkastaa, ettei käyttäjän syöttämästä jonosta löydy muita merkkejä, tai muuten palautetaan virheilmoitus. Mikäli merkkijonosta `"[0-9\\-\\+ ]*"` riisutaan kaikki Java, niin saadaan pelkistetympi muoto `[0-9\-\+ ]*`, eli merkkijonossa saa olla mitä tahansa `*` sulkeiden `[]` sisällä olevia merkkejä `0-9`, -, +, tai välilyönti. Viivalla ilmaistaan väliä käytetyssä merkistössä, koska merkit 0, 1…9 sijaitsevat peräkkäin, niin voidaan hyväksyä väli, missä sijaitsevat kaikki numerot. Erilaisille merkkiryhmille on myös useita vaihtoehtoisia ilmauksia, kuten kokonaisluille löytyvä `\d`.

Tehtävä 15.2 Merkistövälit

[0-9] katsoi että kirjoitettu merkki asettui tietylle välille käytetyssä merkistössä. Voiko samaa tekniikkaa soveltaa aakkosille?

Tehtävä 15.3 Säännöllisten lausekkeiden käyttäminen

Avaa näitä tehtäviä varten myös Javan säännöllisten lausekkeiden API

```
http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html
```

- Kirjoita säännöllinen lauseke, joka täsmää lukuun, joka on kirjoitettu maksimissaan kahden desimaalin tarkkuudella. Erottimeksi kelpaa sekä pilkku, että piste.
- Mitä ongelmia päivämäärän tarkastamiseen liittyy? Miten päivämäärän kysyminen käyttäjältä kannattaa toteuttaa?

16.7 Kentät graafiseen käyttöliittymään

Kuten jo aiemmin todettiin, niin jäsenkentät kannattaa hakea ohjelman suoritusvaiheessa suoraan jäseneltä.

KerhoSwing.java - Harjoitustyö vaihe 7 - luoNaytto()-metodi

```
    /**
     * Luo panelJasen:een jäsenen kenttiä vastaavat labelit ja edit-kentät
     */
    private void luoNaytto() {
        int n = apujasen.getKenttia() - apujasen.ekaKentta();
        editsj = new EditPanel[n];
        panelJasen.removeAll();

        for (int i = 0, k = apujasen.ekaKentta(); k < apujasen.getKenttia(); k++,
i++) {
            EditPanel edit = new EditPanel(); // NOPMD
            edit.setCaption(apujasen.getKysymys(k));
            editsj[i] = edit;
            edit.setName("ej" + k);
            edit.getEdit().setName("tj" + k);
            panelJasen.add(edit);
            edit.addKeyListener(new KeyAdapter() { // NOPMD
                @Override
                public void keyReleased(KeyEvent e) {
                    kasitteleMuutosJaseneen((JTextField)e.getComponent());
                }
            });

        }
    }
```

Ensin ohjelman pitää tietää montako `EditPanel`-komponenttia tullaan tarvitsemaan. Tässä tapauksessa lukumäärä saadaan `apujasen`-oliolta.

```
        int n = apujasen.getKenttia() - apujasen.ekaKentta();
        editsj = new EditPanel[n];
```

Tiedot sisältävässä paneelissa saattaa olla suunnitteluvaiheessa tehtyjä mallidataa sisältäviä komponentteja. Ne on hyvä jättää havainnollistuksen vuoksi paikalleen suunnitelmaan, mutta ohjelman ajon ajaksi ne pitää poistaa. Tämä onnistuu käskemälle paneelille.

```
panelJasen.removeAll();
```

Silmukassa laitetaan kenttien kysymykset ja nimet `EditPanel`-komponentteihin, sekä asetetaan ne näkyviksi. Lisäksi jokaiselle kentälle on luotava vielä oma tapahtumakuuntelija.

```
            edit.addKeyListener(new KeyAdapter() { // NOPMD
                @Override
                public void keyReleased(KeyEvent e) {
                    kasitteleMuutosJaseneen((JTextField)e.getComponent());
                }
            });
```

Osa suunniteltua toiminnallisuutta oli tarkastaa, että onko syötetty tieto oikeassa muodossa ja antaa tarvittaessa käyttäjälle virheilmoitus.

Kuva 15.4 Kentät ja virheilmoitukset

Kun käyttöliittymästä tehdään muutos kenttään, niin tullaan `kasitteleMuutosJaseneen`-metodiin. Aluksi tarkastetaan onko jäsentä jo editoitu, jos ei, niin siitä luodaan muokattava kopio `editJasen`-olioon.

Seuraavaksi haetaan parametrina tuotuun `TextField`-kenttään kirjoitettu teksti ja selvitetään mikä kentistä on kyseessä. Nyt `Jasen`-olioon voidaan yrittää asettaa muuttunut data. Mikäli asetus onnistuu, niin palautuu tyhjä viite. Toisaalta jos palautui virheilmoituksen sisältävä merkkijono, niin se voidaan suoraan asettaa näkymään punaiselle taustalle käyttöliittymään.

KerhoSwing.java - Harjoitustyö vaihe 7 - Jäsenen muuttaminen

```
    private void kasitteleMuutosJaseneen(JTextField edit) {
        if (jasen == null) {
            jasen = new Jasen();
            jasen.rekisteroi();
            setEditJasen(null);
        }
        if ( editJasen == null )
            try {
                     setEditJasen(jasen.clone());
                } catch (CloneNotSupportedException e) { //virhettä ei tule }
        String s = edit.getText();
        int k = Integer.parseInt(edit.getName().substring(2));
        String virhe = editJasen.aseta(k, s);
        setVirhe(virhe);
        if (virhe == null) {
            edit.setToolTipText("");
            edit.setBackground(normaaliVari);
        } else {
            edit.setToolTipText(virhe);
            edit.setBackground(virheVari);
        }
    }
```

16.8 Etsiminen

Kun käyttöliittymä tarvitsee tiettyä osajoukkoa jäsenistä, kannattaa Jasenet-luokan palauttaa tietorakenne, johon on kerätty viitteet hakuehdon täyttävistä jäsenistä:

```
public Collection<Jasen> etsi(String hakuehto, int k) {
    List<Jasen> loytyneet = new ArrayList<Jasen>();
    for (Jasen jasen : this) {
        if (WildChars.onkoSamat(jasen.anna(k), hakuehto)) loytyneet.add(jasen);
    }
    Collections.sort(loytyneet, new Jasen.Vertailija(k));
    return loytyneet;
}
```

16.9 Lajittelu avaimen kentän mukaan

Lajittelussa ongelmaksi tulee se, että eri kenttiä vertaillaan eri tavoilla. Esimerkiksi suomen henkilötunnus merkkijonona

```
010245-123U
121237-121V
```

ei anna oikeata ikäjärjestystä jos tuo laitetaan aakkosjärjestyksen mukaiseen järjestykseen. Siksi tehdään `Jasen`-luokkaan `Vertailija`-luokka, jossa on toteutus metodille `compare`, jonka tehtävä on verrata kahta alkiota keskenään ja palauttaa negatiivi-

nen, nolla tai positiivinen luku sen mukaan, miten verrattavat alkiot suhtautuvat toisiinsa (kuvitteellisesti vähennyslaskun `j1-j2` tulos):

```
public static class Vertailija implements Comparator<Jasen> {
    private final int kenttanro;

    public Vertailija(int k) {
        this.kenttanro = k;
    }

    @Override
    public int compare(Jasen j1, Jasen j2) {
        String s1 = j1.getAvain(kenttanro);
        String s2 = j2.getAvain(kenttanro);
        return s1.compareTo(s2);
    }
}
```

Vastaavasti jäsenen metodi `getAvain` palauttaa vastaavan kentän avain-arvon:

```
return kentat[k].getAvain();
```

Kukin kenttä voi nyt toteuttaa tuon metodin `getAvain` haluamallaan tavalla. Esimerkiksi hetun tapauksessa metodi voisi järjestää hetun osat uuteen järjestykseen (kuten ruotsalaisilla on valmiiksi), joka toimii myös aakkosjärjestyksessä:

```
371212-121V
450102-123U
```

Tosin tarkemmin tehtynä myös välimerkin vuosisata pitää ottaa oikein huomioon. Numeerisen kentän tapauksessa merkkijonona on sama ongelma.

```
123
22
5
```

ei toimi aakkosjärjestyksessä, mutta kun numerot muutetaan sopivaan merkkijonomuotoon, esimerkiksi

```
0000005
0000022
0000123
```

niin merkkijonoksi muutettu aakkostaminen toimii.

Hieman ehkä parempikin toiminnallisuus saataisiin, jos vertailuvastuu olisi siirretty suoraan kunkin kentän vastuulle.

16.10 Tulostaminen

Paperille tulostaminen kannattaa nykyisin hoitaa joko niin, että tuotetaan esim. XHTML-koodia, jota taas vaikka mikä ohjelma (yleensä selaimet) osaa tulostaa. Muita mahdollisuuksia on tuottaa jonkun tekstinkäsittelyohjelman muotoa, esim. LaTeX-koodia, josta tuotetaan siisti PDF.

Microsoft-pohjaisissa järjestelmissä tulostamisvastuu voidaan antaa myös Word-ohjelmalle ja muissa järjestelmissä vastaavalle tekstinkäsittelyohjelmalle. Esimerkiksi Word on erittäin hyvin ohjelmoitavissa kommunikoimaan toisten ohjelmien kanssa. Ajojonoilla ja makroilla myös LibreOffice (entinen OpenOffice) saadaan taipumaan yhteistyöhön ilman että käyttäjän tarvitsee tietää käyttävänsä ko. ohjelmaa.

Hakemisto

--, 158

!

!, 145
  looginen not, 49
!=, **144**

%

%, 154
%=, 157

&

&, 147
&&, 145
  looginen and, 49
&, bittitason AND, 147
&=, 157

*

*/, **63**
*=, 157

.

.C, 60
.EXE, 60
.java, 60
.LIB, 60
.OBJ, 60

/

/*, **63**
/**, 64
/=, 157

;

;, **67**

^

^
  looginen xor, 49
^,bittitason XOR, 147
^=, 157

{

{, 54
{ }, **67**, 98

|

|, 78, 147
|, bittitason OR, 147
||, 78, 145
  looginen or, 49
|=, 157

~

~,bittitason NOT, 147

+

++, 158
+=, **157**

<

<, **144**
<<,rullaus, 147
<<=, 157
<=, **144**

=

=, **76**, 144, 157
-=, 157
= =, 76
==, **144**

>

>, **144**
>=, **144**
>>,rullaus, 147
>>=, 157

0

0, 144

1

1, 144
10-sormijärjestelmä, 8
1–ulotteinen taulukko, **39**

2

2–asteen polynomi, 96
2-asteen yhtälö, 149
2–asteen yhtälö, 96
2-ulotteinen taulukko, 181
2–ulotteinen taulukko, **42**

3

3–ulotteinen taulukko, **43**
3x3 matriisi, 180

4

4–ulotteinen taulukko, **44**

8

8086, 54

A

aakkosjärjestys, **28**
abs, 90
abstract class, 221
acos, 90
ADA, 10, 53
aggregation, 124
Agile, 6
Agile Manifesto, 6
aikainen sidonta, 128
Aikalisa.java
  olioalk, 112
algoritmi, 8, **27**
algoritmin kompleksisuus, **30**
algoritmin parantaminen, 31
algoritmin tarkentaminen, **31**
aliohjelma, 28, **33**, **46**, 50, **85**
alkion poisto, 41
alkuluku, **37**
alustus, taulukko, 178
AND, 49, 145
AND, bittitaso, 147
ANSI–C, 55
APL, 53
app, 204
append, 204
argc, **182**
args, 66
argumentit. ks.argv
argv, **182**
ArrayList, 191
asin, 90
assembler, 35, 54
atan, 90
atan2, 90
attribuutti, 113
autoboxing, 191
automaattiset muuttujat, 186
avaaminen, tiedosto, 204
avain, 229

B

BASIC, 10, 53
BEGIN, 54
binääritiedosto, 201
bittitason operaattorit, 147
block, 142
boolean expression shortcut, 146
Boolen algebra, **49**
Borland Pascal, 54
Borland-C++, 62
BOTTOM-UP, 8, 18
break, **160**, 164
bubble sort, **30**
BufferedReader, 77
byte, **72**

C

C, 10, 51
C#, 57
C++, **52**, 55
CAD, 7
case, **162**
CASE, 62
catch, 78, 79
ceil, 90
char, 98, **179**
child class, 127
clone, 134
close, **205**
ComTest, 9
ComTest \b, 105
constructor, 117
continue, 79, **161**
cos, 90
CRC-kortti, 168

D

de Morganin kaava, **49**
declaration
  esittely, 76
declare, 75
default constructor, 118
definition
  alustus, 76
Delphi, 54
direktiivi, 67
do, **154**
double, **72**, 98
do-while, **154**
do–while, 37
dynaaminen muuttuja, **185**
dynaaminen taulukko, **187**

E

Eclipse, 21
ehto, 142
ehtojen sieventäminen, 50
ehtolause, 142
ehtolauseet, **35**
else, **148**
encapsulation, 117
equals, 134
erota, funktio, **207**
erotaDouble, funktio, 207
erotaInt, funktio, 207
erotinmerkki, 12
esilisäys, 158
etsi pienin, 31
etsiminen, 173, 228
etsimisalgoritmi, 40
evaluointi, 146
event, 25
event driven, 197
exp, 90
extands, 127
Extensible Markup Language, 20
Extreme Programming, 6

F

false, 49
FALSE, 144
FileNotFoundException, 204
final, 68
finally, 205
find, 209
float, **72**
floor, 90
for, **156**, **159**
for-each, 193
FORTH, 54
Fortran, 10, 53
Fortran 77, 55
funktio, **85**
funktio-olio, 197

G

garbage collection, 116
garbage-collection, 84
gc, 84, 116
geneerisyys, 190, 193
globaali muuttuja, 100
GT, 71

H

haku, 32
haku järjestetystä joukosta, **32**
hashCode, 134
heap, 83
Hello7.java
  java-alk, 68
HTML, 64
hybridikieli, 55

I

IEEEremainder, 90
if, **142**
if, peräkkäiset, 153
if, sisäkkäiset, 149
if-else, **148**
ikuinen silmukka, **164**
ilmentymä, 167
immutable, 138
import, 65
indeksi, **39**, 178
IndexOutOfBoundsException, 178
inheritance, 126
InputStreamReader, 77
insertion sort, 29
instance
  esiintymä, 167
int, 65, **72**, 98
INTEGER, 72
interface, 129
IOException, 78
is-a-sääntö, 126
isot ja pienet kirjaimet, **67**
isäluokka, 127
iteraattori, 193

J

jakojäännös, 37
jatkaminen, tiedosto, 204
Java Virtual Machine, 61
java, komento, 61
java.lang, 64
javac, 60
JavaDoc, 64
Java-virtuaalikone, 61
Jbuilder, 62
jono, 39
JUnit, 9
JUnit \b, 104
JVM, 61
jälkilisäys, 158
järjestäminen, 29

K

k_pituudet, 178
kaksiulotteinen taulukko, 43
kapselointi, 113, 117
kasamuisti, 83
keko, 186
kekomuisti, 83
kerho, 11
KerhoSwing, 219
ketterät menetelmät, 6
keyword, 75
kirjoitin, 205
kokonaisluku, 72
kolmiulotteinen taulukko, 43
kombinaatiot, 47
komentorivin parametrit, **182**
kommentti, **63**
kompleksisuus, **30**
konstruktori, 117
koodi, 179
koordinaatisto, 42
koostaminen, 124
koottu lause, 142
korttipakka, 39
korvaaminen, 129
kotelointi, 117
kuormittaa, 97, 121
kuplalajittelu, **30**
kävely, 28
kääntäminen, **59**

L

ladontaohjelma, 10
lajittelu, **28**, 39, 173, 228
lajittelu avaimen mukaan, **31**
lapsiluokka, 127
laskeva, 30
late binding, 128
lausekieli, 8, 29, 33
lauseryhmä, 63
lausesulut, **67**
lineaarinen lista, 12, 172
LinkedList, 191
linkitetty lista, **172**
linkittäminen, **60**
Lisp, 53
lista, 39, 172, 173

lisämäärittely, 121
lisäys, 173
lisäysoperaattori, --, 158
lisäysoperaattori, ++, 158
litereal, 75
log, 90
lohko, 142
lokaali muuttuja, 82, **97**
lokaalit muuttujat, 186
long, **72**
looginen lause, 143
loogiset operaatiot, 49
loppuvälilyönti, 13
lukeminen, 202
luokan esiintymä, 167
luokkametodi, 85
luokkamuuttuja, 100

M

main, **65**, 66
Math, 76, 90
matriisi, **42**, 180
max, 90
menu, **14**
merkki, 179
merkkijono, 50, **179**
message passing, 114
method, 113
metodi, 113
Microsoft, 62
min, 90
mittakaava, 71, 73
Mjonot, 207
Mjonot.java, 207
Modula-2, 10
Modula–2, 53
modulaarinen testaus, 80
modulitestaus, 65
monimuotoisuus, 128, 131
moniulotteinen taulukko, 42
moniulotteinen taulukko 1-ulotteisena, 180
moniulotteinen taulukko 1–ulotteisena, **42**
moniulotteiset taulukot, 179
muistinsiivous, 116
muodostaja, 117
mutable, 138
muuttuja, **72**
  dynaaminen. ks. dynaaminen muuttuja
muuttujan esittely, 72
muuttujat, 37
MVC, 218
MVP, 218
myöhäinen sidonta, 128

N

NetBeans, 21, 62
new, 83
nimet.dat, 13
NOT, 49, 145, 147
nouseva, 30
null, 84
NumberFormatException, 79
näkyvyysalue, 98

O

object, 113
Object-luokka, 134
ohjelman ajaminen, **60**
oikeellisuustarkistus, 224
oletusarvo, 17
oletusmuodostajaksi, 118
olio, 113, 167
olion tuhoaminen, 186
omat komponentit, 23
OR, 49, 145
OR, bittitaso, 147
osaongelma, 33
osoitin, **41**
osoitintaulukko, 181
out, 67
overload, 97
overloading, 121
overriding, 129

P

paista, 46
palanen, 17
parametri, **46**, 86
parametri, komentorivi, 182
parametri, useita, 95
parent class, 127
parse, 207
parseInt, 79
Pascal, 10, 52, 72
perintä, 114, 126
peräkkäishaku, **32**
perään kirjoittaminen, tiedosto, 204
PI, 76
pienin, 31
pino, 39
poikkeukset, 189
poikkeus, 78
poista, 50
poisto, 41, 173
polymorfismi, 114, 128, 131
polymorphism, 128
pow, 90
println, **67**
proto, **7**
public, 65
puhelinluettelo, 32
puolipiste, 67
puolitushaku, **32**
puu, 12
päivämäärät, 50
pätkiminen, 17
pääohjelma, 87
päätöstaulu, 46
pöytätesti, **38**, 87

Q

QuickSort, 31

R

rajapinta, 129, 200
rakentaja, 117
random, 90
read-only, 124
refaktorointi, 23
references, 81
regular expression, 224
rekisteröinti, 36
relaatiomalli, 175
relaatiotietokantamalli, 19
repeat, 37
return, 79, **89**, 152
rint, 90
rivilista, 180
rivinvaihto, 13, 67
rivitalo, 42
rivityyppi, 42
roskien keruu, 116
roskienkeruu, 84
round, 90
rullaus, 147
ruutupaperi, 33

S

saantimetodi, 124
sarakelista, 180
scope, 98
sekarakenne, 44, 173
selection sort, 30
shift, 147
short, **72**
sidontajärjestys, 146
sijoitus, 76
sijoitus on lauseke, 144
sijoitusoperaattori, %=, 157
sijoitusoperaattori, &=, 157
sijoitusoperaattori, *=, 157
sijoitusoperaattori, /=, 157
sijoitusoperaattori, ^=, 157
sijoitusoperaattori, |=, 157
sijoitusoperaattori, +=, 157
sijoitusoperaattori, <<=, 157
sijoitusoperaattori, =, 157
sijoitusoperaattori, -=, 157
sijoitusoperaattori, >>=, 157
silmukat, **37**
sin, 90
sormi, 41
sovelluskehitin, 7
sqrt, 90
sscanf, 207
standardikirjasto, 31, 85
static, 65
stdin, 201
stdout, 201
string
    npos, 209
String, **66**
StringBuffer, 179
StringBuilder, 179
subclass, 127
substring, 59
sulkeminen, 205
suora haku, **32**

suoritusjärjestys, 146
super, 127
switch, **162**
switch, break, 162
switch, case, 162
switch, default, 162
syntaksivirhe, 67
syrjäyttäminen, 129
System, 67
säännöllinen lauseke, 224

T

tabulointi, 67
tai, operaattori, 78
tallennus, 16
tan, 90
tapahtuma, 25
tapahtumalähtöinen ohjelmointi, 197
tapahtumaohjattu järjestelmä, 55
taulukko, 171, 177
taulukko, alustaminen, 178
taulukko, alustus, 180
taulukko, dynaaminen, **187**
taulukko, moniulotteinen, 179
taulukko, moniulotteinen 1-ulotteisena, 180
taulukko, osoittimista, 181
taulukko, taulukoista, 180
taulukkolaskenta, 7, 10, 12
taulukot, **39**
TDD, 104
tekstieditori, 13
tekstinkäsittely, 7, 10
tekstitiedosto, **12**, 59, 201
test case, 104
Test Driven Development, 104
TeX, 10
this, 122
tiedosto, 12, 201
tiedosto, avaaminen, 204
tiedosto, binääri, 201
tiedosto, jatkaminen, 204
tiedosto, lukeminen, 202
tiedosto, rivillä monta kenttää, 206
tiedosto, sulkeminen, 205
tiedosto, teksti, 201
tiedoston jatkaminen, 204
tietokanta, 7, 10
tietokantaohjelmisto, 12
tietorakenne, 171
tietovirta parametrina, 195
toDegrees, 90
TOP-DOWN, 8, 18
toRadians, 90
toString, 134
totuustaulu, **46**, 48
true, 49, 79
TRUE, 144
try, 79, 205
tulostaminen, 229
tuotteet.dat
tiedosto, 206
Turbo Pascal, 54
tyhjä, 67

U

uiminen, 37
Unicode, 75
UNICODE, 179
UNIT, 54
unit testing, 104
UNIX, 55
using, 65
uudelleenmääritys, 129

V,W

vaihda, 142
vaihtoehtojen lukumäärä, 47
vakio, 73, **75**
vakioarvo, 69
valikko, 14
valintalause, **36**
valmisohjelma, 7
VAR, 72
VBA, 97
Vector, 191, 192
vektori, 33, **39**
vertailu, **28**
vertailuoperaattori, **144**
vesiputousmalli, 5
while, 79, **155**
white space, **67**
viesti, 114
viitemuuttuja, 81
WindowBuilder, 21
virtuaalikone, 61
Visual, 21
Visual Basic, 54
Visual Studio, 62
void, 66, 89
vuokaavio, 37
välilyönti, 67
välitön ali/yliluokka, 127

X

x y, 42
XML, 21
XOR, 49
XOR, bititaso, 147
XP, 6

Y

yksikkötestaus, 104
yksikkötestausta, 65
ylläpito, 9

Jyväskylän yliopisto
Tietotekniikan laitos

Luentomoniste 11

1. MÄKINEN, RAINO A. E., Numeeriset menetelmät. 1999 (107 s.)

2. LAPPALAINEN, VESA ja RISTO LAHDELMA, Olio-ohjelmointi ja C++. 1999 (107 s.)

3. LAPPALAINEN, VESA, Windows-ohjelmointi C-kielellä. 1999 (150 s.)

4. ORPONEN, PEKKA, Tietorakenteet ja algoritmit 2. 2.p., 2000 (50 s.)

5. LAPPALAINEN, VESA, Ohjelmointi++. 1999 (315 s.)

6. MÄNNIKKÖ, TIMO, Johdatus ohjelmointiin. 2000 (155 s.)

7. KOIKKALAINEN, PASI ja PEKKA ORPONEN, Tietotekniikan perusteet. 2001 (150 s.)

8. ARNĂUTU, VIOREL, Numerical methods for variational problems. 2001 (100 s.)

9. KRAVCHUK, ALEXANDER, Mathematical modelling of the biomedical tomography: The 12th Jyväskylä Summer School. 2003 (83 s.)

10. MIETTINEN, KAISA, Epälineaarinen optimointi. 2003 (146 s.)

11. LAPPALAINEN, VESA & VIITANEN SANTTU, Ohjelmointi 2. 2012 (234 s.)

12. KAIJANAHO, ANTTI-JUHANI & KÄRKKÄINEN TOMMI, Formaalit menetelmät. 2005 (171 s.)

13. HOPPE, RONALD H. W., Numerical solution of optimization problems with PDE constraints: Lecture notes of a course given in the 14th Jyväskylä Summer School, August 9-27, 2004. 2006 (65 s.)

14. JYRKI JOUTSENSALO, TIMO HÄMÄLÄINEN & ALEXANDER SAYENKO, QoS Supported Networks, Scheduling, and Pricing; Theory and Applications (214 s.)

15. MARTTI HYVÖNEN, VESA LAPPALAINEN, Ohjelmointi 1. 2009. (132 s.)

16. ANTTI-JUHANI KAIJANAHO, Ohjelmointikielten periaatteet. 2010. (153 s.)

17. MARTTI HYVÖNEN, VESA LAPPALAINEN, ANTTI-JUSSI LAKANEN, Ohjelmointi 1 C#. 2012, 2. painos. (160 s.)

ISBN 978-951-39-4624-1 (uudistettu painos)
ISSN 1456-9787