Jyväskylän yliopisto
Tietotekniikan laitos

Luentomoniste 15

University of Jyväskylä
Department of Mathematical Information Technology

Lecture Notes 15

OHJELMOINTI 1

2. uudistettu painos

Martti Hyvönen ja Vesa Lappalainen

Jyväskylä 2010

Jyväskylän yliopisto *University of Jyväskylä*
Tietotekniikan laitos *Department of Mathematical Information Technology*

Luentomoniste 15 *Lecture Notes 15*

"The city's central computer told you? R2D2, you know better than to trust a strange computer!" -C3PO

OHJELMOINTI 1

- jotta tietokoneisiin voitaisiin luottaa myös tulevaisuudessa

Martti Hyvönen ja Vesa Lappalainen

Versio 1.5 07.09.2010

Jyväskylä 2010

Sisällys

Esipuhe........1
1. Mitä ohjelmointi on?........2
2. Ensimmäinen Java-ohjelma........3
2.1 Ohjelman kirjoittaminen........3
2.2 Ohjelman kääntäminen ja ajaminen........3
2.3 Ohjelman rakenne........4
2.3.1 Virhetyypit........5
2.3.2 Tyhjät merkit (White spaces)........5
2.4 Kommentointi........6
2.4.1 Dokumentointi........6
3. Algoritmit........8
3.1 Mikä on algoritmi?........8
3.2 Tarkentaminen........8
3.3 Yleistäminen........9
3.4 Peräkkäisyys........9
4. Yksinkertainen graafinen Java-ohjelma........10
4.1 Mikä on kirjasto?........10
4.2 Esimerkki piirtämisestä Jyväskylän yliopiston Graphics-kirjastolla........10
4.2.1 Ohjelman suoritus........11
4.2.2 Ohjelman oleellisemmat kohdat........11
5. Lähdekoodista prosessorille........14
5.1 Kääntäminen........14
5.2 Suorittaminen........14
6. Aliohjelmat........15
6.1 Aliohjelman kutsuminen........16
6.2 Aliohjelman kirjoittaminen........17
6.3 Aliohjelmien dokumentointi........19
6.4 Aliohjelmat, metodit ja funktiot........21
7. Muuttujat........22
7.1 Muuttujan määrittely........22
7.1.1 Javan alkeistietotyypit........22
7.1.2 Muuttujan nimeäminen........23
7.1.3 Javan varatut sanat........24
7.2 Arvon asettaminen muuttujaan........24
7.3 Muuttujien näkyvyys........25
7.4 Vakiot........25
7.5 Aritmeettiset lausekkeet........25
7.5.1 Javan aritmeettiset operaatiot........25
7.5.2 Aritmeettisten operaatioiden suoritusjärjestys........27
7.5.3 Huomautuksia........27
7.6 Esimerkki: Painoindeksi........28
8. Oliotietotyypit........29
8.1 Mitä oliot ovat?........29
8.2 Olion luominen........29
8.3 Oliotietotyyppien ja alkeistietotyyppien ero........30
8.4 Metodin kutsuminen........32
8.5 Olion tuhoamisen hoitaa roskienkeruu........32
8.6 Olioluokkien dokumentaatio........33
8.6.1 Constructor Summary........33
8.6.2 Method Summary........34
8.6.3 Huomautus: Luokkien dokumentaation googlettaminen........34
8.7 Tyyppimuunnokset........34
9. Aliohjelman paluuarvo........36
10. Eclipse........38
10.1 Asennus........38

10.2 Käyttö....38
10.2.1 Ensimmäinen käyttökerta....38
10.2.2 Ohjelman kirjoittaminen....39
10.2.3 Ohjelman kääntäminen ja ajaminen....39
10.2.4 Debuggaus....39
10.2.5 Paketit....40
10.2.6 Jar-tiedostojen käyttäminen....40
10.3 Hyödyllisiä ominaisuuksia....40
10.3.1 Syntaksivirheiden etsintä....40
10.3.2 Quick Fix....40
10.3.3 Kooditäydennys (content assist)....41
10.3.4 Koodimallit (Templates)....41
11. ComTest....42
11.1 ComTest:n käyttö....42
11.2 Liukulukujen testaaminen....43
12. Merkkijonot....45
12.1 String....45
12.1.1 Hyödyllisiä metodeja....45
12.2 Muokattavat merkkijonot: esimerkkinä StringBuilder....46
12.2.1 Muita hyödyllisiä metodeja....47
12.2.2 StringBuffer....47
12.3 Merkkijonojen tulostaminen....47
12.3.1 Huomautus: Aritmeettinen+ vs. merkkijonoja yhdistelevä+....47
12.3.2 Vinkki: Näppärä tyyppimuunnos String-tyypiksi....48
12.3.3 Metodi: Reaalilukujen muotoilu String.format-metodilla....48
12.3.4 Metodi: Muotoilujen tulostaminen System.out.printf-metodilla....48
13. Ehtolauseet (Valintalauseet)....50
13.1 Mihin ehtolauseita tarvitaan?....50
13.2 if-rakenne: ”Jos aurinko paistaa, mene ulos.”....50
13.3 Vertailuoperaattorit....51
13.3.1 Huomautus: Sijoitusoperaattori (=) ja vertailuoperaattori (==)....51
13.3.2 Vertailuoperaattoreiden käyttö....51
13.4 if-else -rakenne....52
13.4.1 Esimerkki: Pariton vai parillinen....53
13.5 Loogiset operaatiot....54
13.5.1 De Morganin lait....54
13.5.2 Osittelulaki....55
13.6 else if -rakenne....55
13.6.1 Esimerkki: Tenttiarvosanan laskeminen....57
13.7 switch-rakenne....58
13.7.1 Esimerkki: Arvosana kirjalliseksi....58
13.8 Esimerkki: Olioiden ja alkeistietotyyppien erot....60
14. Taulukot....63
14.1 Taulukon luominen....63
14.2 Taulukon alkioon viittaaminen....64
14.3 Esimerkki: Arvosana kirjalliseksi....64
14.4 Moniulotteiset taulukot....65
14.5 Taulukon kopioiminen....67
14.6 Taulukot parametreina....67
15. Toistorakenteet (silmukat)....68
15.1 Idea....68
15.2 while-silmukka....68
15.2.1 Esimerkki: Taulukon tulostaminen....70
15.3 do-while -silmukka....71
15.3.1 Esimerkki: Tikkataulu....73
15.4 for-silmukka....74
15.4.1 Esimerkki: keskiarvo-aliohjelma....75
15.4.2 Esimerkki: Taulukon kääntäminen käänteiseen järjestykseen....76

15.4.3 Esimerkki: Arvosanan laskeminen taulukoilla....77
15.5 For-each -silmukka....80
15.5.1 Esimerkki: Sisäkkäiset silmukat....80
15.6 Silmukan suorituksen kontrollointi break- ja continue-lauseilla....81
15.6.1 break....81
15.6.2 continue....81
15.7 Ohjelmointikielistä puuttuva silmukkarakenne....82
15.8 Yhteenveto....82
16. Merkkijonojen pilkkominen....84
16.1 StringTokenizer....84
16.1.1 Esimerkki: Merkkijonon pilkkominen StringTokenizerilla....84
16.2 split....85
17. Järjestäminen....87
18. Konsoliohjelmien tekeminen....88
18.1 Tietovirrat....88
18.1.1 Standardivirrat....88
18.2 Käyttäjän syötteen lukeminen....88
18.2.1 Esimerkki: Yksinkertainen käyttöliittymä switch-case -rakenteen avulla....89
18.3 Käyttäjän syötteen lukeminen Ali.jar kirjastoa käyttämällä....90
18.4 Parametrien antaminen ohjelmaa käynnistettäessä (args-taulukko)....91
19. Rekursio....93
19.1 Sierpinskin kolmio....96
19.2 Nopeampi Sierpinskin Kolmio....100
20. Dynaamiset tietorakenteet....103
20.1 Rajapinnat....103
20.2 ArrayList....104
20.2.1 Tietorakenteen määrittäminen....104
20.2.2 Peruskäyttö....105
20.2.3 Lukujen tallentaminen tietorakenteeseen, autoboxing....105
21. Esimerkki: Hirsipuupeli....106
21.1 Simppeli versio....106
21.2 EasyWindow-luokasta Window-luokkaan....111
21.2.1 Esimerkki: Hirsipuun piirto....111
21.3 Hirsipuun piirtäminen pelissä....114
22. Tiedostot....117
22.1 Tiedostot Ali.jar -kirjaston avulla....117
22.2 Sanojen lukeminen tiedostosta hirsipuupelissä....117
22.2.1 Luokka: Random....118
22.2.2 Arpomisaliohjelma hirsipuupeliin....118
23. Poikkeukset....120
23.1 try-catch....120
23.2 finally-lohko....121
23.3 Yleistä....121
24. Lukujen esitys tietokoneessa....122
24.1 Lukujärjestelmät....122
24.2 Paikkajärjestelmät....122
24.3 Binääriluvut....122
24.3.1 Binääriluku 10-järjestelmän luvuksi....123
24.3.2 10-järjestelmän luku binääriluvuksi....123
24.4 Negatiiviset binääriluvut....124
24.4.1 Suora tulkinta....124
24.4.2 1-komplementti....125
24.4.3 2-komplementti....125
24.4.4 2-komplementin yhteenlasku....125
24.5 Lukujärjestelmien suhde toisiinsa....126
24.6 Liukuluku (floating-point)....128
24.6.1 Liukuluvun binääriesityksen muuttaminen 10-järjestelmään....128
24.6.2 10-järjestelmän luku liukuluvun binääriesitykseksi....129

24.6.3 Huomio: doublen lukualue....130
24.6.4 Liukulukujen tarkkuus....130
24.6.5 Intelin prosessorikaan ei ole aina osannut laskea liukulukuja oikein....131
25. ASCII-koodi....132
26. Syntaksin kuvaaminen....134
26.1 BNF....134
26.2 Laajennettu BNF (EBNF)....135
27. Jälkisanat....136
Liite: Hirsipuu olioilla tehtynä....137
Liite: Sanasto....141
Liite: Yleisimmät virheilmoitukset ja niiden syyt....142
27.1 ArrayIndexOutOfBoundsException....142
27.2 Unresolved compilation problem....142
27.3 NullPointerException....142
27.4 NoSuchElementException....142

Esipuhe

Tämä moniste on nimenomaan luentomoniste kurssille Ohjelmointi 1. Luentomoniste tarkoittaa sitä, että sen ei ole tarkoituskaan korvata kunnon kirjaa, vaan esittää asiat samassa järjestyksessä ja samassa valossa kuin ne esitetään luennolla. Jotta moniste ei paisuisi kohtuuttomasti, ei asioita käsitellä missään nimessä täydellisesti. Siksi tarvitaan tueksi jokin kunnon ohjelmointia käsittelevä kirja. Useimmat saatavilla olevat kirjat keskittyvät hyvin paljon tiettyyn ohjelmointikieleen. Erityisesti aloittelijoille tarkoitetut. Osin tämä on luonnollista, koska ihminenkin kommunikoidakseen toisen kanssa tarvitsee jonkin yhteisen kielen. Ja siksi ohjelmoinnin aloittaminen ilman, että ensin opetellaan jonkun kielen perusteet, on aika haastavaa.

Kirjoissa jäsentelyn selkeyden takia käsitellään yleensä yksi aihe perusteellisesti alusta loppuun. Aloittaessaan puhumaan lapsi ei kuitenkaan ole kykeneväinen omaksumaan kaikkea tietyn lauserakenteen kieliopista. Vastaavasti ohjelmoinnin alkeita kahlattaessa vastaanottokyky ei vielä riitä kaikkien kikkojen käsittämiseen. Monisteessa ja luennolla asioiden käsittelyjärjestys on sellainen, että asioista annetaan ensin esimerkkejä tai johdatellaan niiden tarpeeseen ja sitten jonkin verran selitetään mistä oli kyse. Siksi monisteesta saa yhden näkemyksen mukaisen pintaraapaisun asioille ja kirjoista ja nettilähteistä asiaa on syvennettävä.

Tässä monisteessa käytetään esimerkkikielenä Java-kieltä. Kuitenkin nimenomaan esimerkkinä, koska monisteen rakenne ja esimerkit voisivat olla aivan samanlaisia mille tahansa muullekin ohjelmointikielelle. Tärkeintä on nimenomaan ohjelmoinnin ajattelutavan oppiminen. Kielen vaihtaminen toiseen samansukuiseen kieleen on enemmänkin kuin savon murteen vaihtaminen Turun murteeseen, kuin suomen kielen vaihtamista ruotsinkieleen. Eli jos yhdellä kielellä on oppinut ohjelmoimaan, kykenee kyllä jo lukemaan toisella kielellä kirjoitettuja ohjelmia pienen harjoittelun jälkeen. Toisella kielellä kirjoittaminen on hieman haastavampaa, mutta samat rakenteet sielläkin toistuvat. Ohjelmointikielet tulevat ja menevät, tätäkin vastaavaa kurssia on pidetty Jyväskylän yliopistossa seuraavilla kielillä: Fortran, Pascal, C, ja C++. Joissakin yliopistoissa aloituskielenä on Python.

Ohjelmointia on täysin mahdotonta oppia pelkästään kirjoja lukemalla. Siksi kurssi sisältää luentojen ohella myös viikoittaisten harjoitustehtävien (demojen) tekemistä, ohjattua pääteharjoittelua tietokoneluokassa sekä harjoitustyön tekemisen. Näistä lisätietoa, samoin kuin kurssilla käytettävien työkalujen hankkimisesta ja asentamisesta löytyy kurssin kotisivuilta:

```
http://users.jyu.fi/~vesal/kurssit/ohjelmointi1/2009/
```

sekä kurssin Wiki-sivuilta:

```
https://trac.cc.jyu.fi/projects/ohj1/wiki
```

Moniste perustuu monelta osin Timo Männikön vuoden 2000 kurssille kokoamaan monisteeseen, joka taas pohjautuu monen kirjoittajan monisteisiin aina -80 -luvulta alkaen. Monisteen rakenne ja esimerkit noudattelevat Vesa Lappalaisen syksyllä 2008 pitämän kurssin runkoa. Monisteen kirjoittamistyön on tehnyt kesällä 2009 harjoittelija Martti Hyvönen. Monistetta oikolukemassa ja vinkkejä antamassa on ollut lukuisa määrä henkilöitä, joista tietysti erityiskiitos Jonne Itkoselle.

Jyväskylässä 28.8.2009

Vesa Lappalainen, Martti Hyvönen

1. Mitä ohjelmointi on?

Ohjelmointi on yksinkertaisimmillaan toimintaohjeiden antamista ennalta määrätyn toimenpiteen suorittamista varten. Ohjelmoinnin kaltaista toimintaa esiintyy jokaisen ihmisen arkielämässä lähes päivittäin. Algoritmista esimerkkinä voisi olla se, että annamme jollekulle puhelimessa ajo-ohjeet, joiden avulla hänen tulee päästä perille ennestään vieraaseen paikkaan. Tällöin luomme sarjan ohjeita ja komentoja, jotka ohjaavat toimenpiteen suoritusta. Alkeellista ohjelmointia on tavallaan myös mikroaaltouunin käyttäminen, sillä tällöin uunille annetaan selkeät ohjeet siitä, kuinka kauan ja kuinka suurella teholla sen tulee toimia.

Kaikissa edellisissä esimerkeissä oli siis kyse selkeiden yksikäsitteisten ohjeiden antamisesta. Kuitenkin esimerkit käsittelivät hyvinkin erilaisia viestintätilanteita. Ihmisten välinen kommunikaatio, mikroaaltouunin kytkimien kiertäminen tai nappien painaminen, samoin kuin videon ajastimen säätö laserkynällä ovat ohjelmoinnin kannalta toisiinsa rinnastettavissa, mutta ne tapahtuvat eri työvälineitä käyttäen. Ohjelmoinnissa työvälineiden valinta riippuu asetetun tehtävän ratkaisuun käytettävissä olevista välineistä. Ihmisten välinen kommunikaatio voi tapahtua puhumalla, kirjoittamalla tai näiden yhdistelmänä. Samoin ohjelmoinnissa voidaan usein valita erilaisia toteutustapoja tehtävän luonteesta riippuen.

Ohjelmoinnissa on olemassa eri tasoja riippuen siitä, minkälaista työvälinettä tehtävän ratkaisuun käytetään. Pitkälle kehitetyt korkean tason työvälineet mahdollistavat työskentelyn käsitteillä ja ilmaisuilla, jotka parhaimmillaan muistuttavat luonnollisen kielen käyttämiä käsitteitä ja ilmaisuja, kun taas matalan tason työvälineillä työskennellään hyvin yksinkertaisilla ja alkeellisilla käsitteillä ja ilmaisuilla.

Eräänä esimerkkinä ohjelmoinnista voidaan pitää sokerikakun valmistukseen kirjoitettua ohjetta:

```
Sokerikakku

  6       munaa
  1,5 dl  sokeria
  1,5 dl  jauhoja
  1,5 tl  leivinjauhetta

1.  Vatkaa sokeri ja munat vaahdoksi.
2.  Sekoita jauhot ja leivinjauhe.
3.  Sekoita muna-sokerivaahto ja jauhoseos.
4.  Paista 45 min 175°C lämpötilassa.
```

Valmistusohje on ilmiselvästi kirjoitettu ihmistä varten, vieläpä sellaista ihmistä, joka tietää leipomisesta melko paljon. Jos sama ohje kirjoitettaisiin ihmiselle, joka ei eläessään ole leiponut mitään, ei edellä esitetty ohje olisi alkuunkaan riittävä, vaan siinä täytyisi huomioida useita leipomiseen liittyviä niksejä: uunin ennakkoon lämmittäminen, vaahdon vatkauksen salat, yms.

Koneelle kirjoitettavat ohjeet poikkeavat merkittävästi ihmisille kirjoitetuista ohjeista. Kone ei osaa automaattisesti kysyä neuvoa törmätessään uuteen ja ennalta arvaamattomaan tilanteeseen. Se toimii täsmälleen niiden ohjeiden mukaan, jotka sille on annettu, olivatpa ne vallitsevassa tilanteessa mielekkäitä tai eivät. Kone toistaa saamiaan toimintaohjeita uskollisesti sortumatta ihmisille tyypilliseen luovuuteen. Näin ollen tämän päivän ohjelmointikielillä koneelle tarkoitetut ohjeet on esitettävä hyvin tarkoin määritellyssä muodossa ja niissä on pyrittävä ottamaan huomioon kaikki mahdollisesti esille tulevat tilanteet. [MÄN]

2. Ensimmäinen Java-ohjelma

2.1 Ohjelman kirjoittaminen

Java-ohjelmia voi kirjoittaa millä tahansa tekstieditorilla. Tekstieditoreja on kymmeniä, ellei satoja, joten yhden nimeäminen on vaikeaa. Osa on kuitenkin suunniteltu varta vasten ohjelmointia ajatellen. Tällaiset tekstieditorit osaavat muotoilla ohjelmoijan kirjoittamaa *koodia* automaattisesti, joka selkeyttää koodin ymmärtämistä. Ohjelmoijien suosimia ovat mm. ConText, Vim ja Emacs, mutta monet muutkin ovat varmasti hyviä. Monisteen alun esimerkkien kirjoittamiseen soveltuu hyvin mikä tahansa tekstieditori.

koodi, lähdekoodi = ohjelmoijan tuottama tiedosto, josta varsinainen ohjelma muutetaan tietokoneen ymmärtämäksi konekieleksi

Kirjoitetaan tekstieditorilla alla olevan mukainen Java-ohjelma ja tallennetaan se vaikka nimellä HelloWorld.java. Tiedostopäätteen on oltava juuri tuo .java, muuten ohjelman *kääntäminen* ei onnistu. Kannattaa olla tarkkana, sillä jotkut tekstieditorit yrittävät oletuksena tallentaa kaikki tiedostot muodossa .txt ja tällöin tiedoston nimi voi helposti tulla muotoon HelloWorld.java.txt.

```
public class HelloWorld {

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

Tämän ohjelman pitäisi tulostaa näytölle teksti Hello World! Voidaksemme kokeilla ohjelmaa käytännössä, täytyy se ensiksi kääntää tietokoneen ymmärtämään muotoon.

kääntäminen = Ohjelman kääntämisellä tarkoitetaan kirjoitetun lähdekoodin muuntamista suoritettavaksi ohjelmaksi.

Esimerkkejä muilla ohjelmointikielillä kirjoitetusta HelloWorld -ohjelmasta löydät vaikkapa:
http://www2.latech.edu/~acm/HelloWorld.html

2.2 Ohjelman kääntäminen ja ajaminen

Jotta ohjelman kääntäminen ja suorittaminen onnistuu, täytyy koneelle olla asennettuna joku Java-sovelluskehitin. Aluksi riittää Sunin sivuilta löytyvä ilmainen JDK (Java Developer Kit). JDK tulee seuraavien Java alustojen (**Java Platform**) mukana:

- Java SE: Java Standard Edition Platform
- Java EE: Java Enterprise Edition Platform
- Java ME: Java Micro Edition Platform

Tämän kurssin tarpeisiin soveltunee parhaiten ylin vaihtoehto. Enterprise Edition on suunnattu vaativampaan ohjelmistokehitykseen ja Micro Edition mobiilisovellusten kehitykseen.

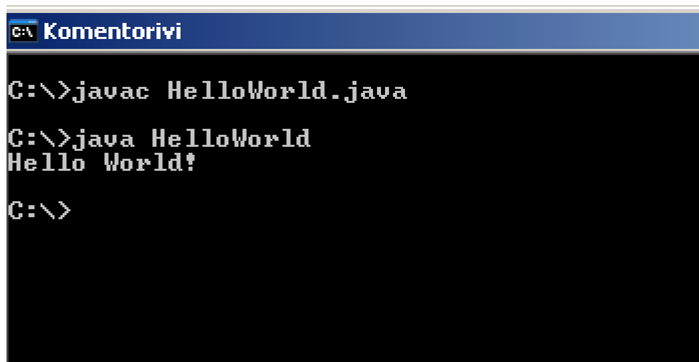
Kun JDK on asennettuna, käynnistetään komentorivi (**Command Prompt**) ja mennään siihen hakemistoon mihin HelloWorld.java tiedosto on tallennettu. Ohjelma käännetään nyt komennolla:

```
javac HelloWorld.java
```

Komento ”javac” tulee sanoista Java Compiler (compiler = kääntäjä). Kääntämisen jälkeen hakemistoon ilmestyy HelloWorld.class-niminen tiedosto, joka voidaan ajaa komennolla:

```
java HelloWorld
```

Ohjelman tulisi nyt tulostaa näyttöön teksti Hello World!, kuten alla olevassa kuvassa.



Kuva 1: Ohjelman kääntäminen ja ajaminen Windowsin komentorivillä.

Huomaa, että käännettäessä kirjoitetaan koko tiedoston nimi .java päätteineen, kun taas ajettaessa käytetään pelkkää luokan nimeä ilman .class-päätettä.

Jos saat virheilmoituksen ” 'javac' is not recognized as an internal or external command, operable program or batch file.”, niin komentoa ”javac” ei silloin löydy hakupolusta. Lisääminen onnistuu komennolla:

```
set PATH=%PATH%;hakemistopolku_josta_asennettu_jdk_löytyy\bin
```

Omalla koneellani tämä on esimerkiksi:

```
set PATH=%PATH%;C:\Program Files\Java\jdk1.6.0_14\bin
```

Komennon voi lisätä pysyvästi hakupolkuun Windows Vistassa seuraavasti:

Control Panel → System and Maintenance → System → Advanced system settings → Environment variables...[VES]

2.3 Ohjelman rakenne

Ensimmäinen kirjoittamamme ohjelma HelloWorld.java on oikeastaan yksinkertaisin mahdollinen Java-ohjelma.

```
public class HelloWorld {
```

Yllä olevalla ohjelman ensimmäisellä rivillä määritellään *luokka* (**class**) jonka nimi on `HelloWorld`. Javassa luokkien nimet alkavat aina isolla kirjaimella. Luokan edessä oleva `public`-määre tarkoittaa, että luokka on julkinen, kuten luokat useimmiten ovat. Jokainen Java-ohjelma on kirjoitettava luokan sisään, joten jokaisessa Java-ohjelmassa on tällöin vähintään yksi luokka. Luokan, jonka sisään Java-ohjelma kirjoitetaan, on oltava samanniminen kuin tiedoston nimi. Jos tiedoston nimi on HelloWorld.java on luokan nimen siis oltava `HelloWorld` kuten meidän esimerkissämme. Tässä vaiheessa ei kuitenkaan vielä kannata liikaa vaivata päätänsä sillä, mikä luokka oikeastaan on, se selviää tarkemmin myöhemmin. Nyt riittää ajatella luokkaa ”kotina” *aliohjelmille*. Aliohjelmista puhutaan kohta lisää.

Ensimmäisen rivin perässä on oikealle auki oleva aaltosulku. Useissa ohjelmointikielissä yhteen liittyvät asiat ryhmitellään tai kootaan aaltosulkeiden sisälle. Oikealle auki olevaa aaltosulkua sanotaan aloittavaksi aaltosuluksi ja tässä tapauksessa se kertoo kääntäjälle, että tästä alkaa `HelloWorld`-luokkaan liittyvät asiat. Jokaista aloittavaa aaltosulkua vastaan täytyy olla

vasemmalle auki oleva lopettava aaltosulku. `HelloWorld`-luokan lopettava aaltosulku on rivillä seitsemän, joka on samalla ohjelman viimeinen rivi. Aaltosulkeiden rajoittamaa aluetta kutsutaan lohkoksi (**block**).

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

Rivillä kolme määritellään uusi aliohjelma nimeltä `main`. Nimensä ansiosta se on tämän luokan pääohjelma. Pääohjelma täytyy Javassa aina määritellä yllä olevassa muodossa. Samoin kuin luokan, niin myös pääohjelman sisältö kirjoitetaan aaltosulkeiden sisään. Javassa ohjelmoijan kirjoittaman koodin suorittaminen alkaa aina käynnistettävän luokan pääohjelmasta. Toki sisäisesti ehtii tapahtua paljon asioita jo ennen tätä.

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

Rivillä viisi tulostetaan näytölle Hello World!. Javassa tämä tapahtuu pyytämällä Javan standardikirjaston `System`-luokan `out`-oliota tulostamaan `println`-*metodilla* (**method**). Kirjastoista, olioista ja metodeista puhutaan lisää kohdassa 4.1 Mikä on kirjasto? ja luvussa 8 Oliotietotyypit. Tulostettava merkkijono kirjoitetaan sulkeiden sisälle lainausmerkkeihin. Tämä rivi on myös tämän ohjelman ainoa *lause* (**statement**). Lauseiden voidaan ajatella olevan yksittäisiä toimenpiteitä, joista ohjelma koostuu. Jokainen lause päättyy Javassa puolipisteeseen. Koska lauseen loppuminen ilmoitetaan puolipisteellä, ei Javan *syntaksissa* (**syntax**) "tyhjillä merkeillä" (**white space**), kuten rivinvaihdoilla ja välilyönneillä ole merkitystä ohjelman toiminnan kannalta. Ohjelmakoodin luettavuuden kannalta niillä on kuitenkin suuri merkitys. Tästä lisää hieman myöhemmin. Puolipisteen unohtaminen on yksi yleisimmistä ohjelmointivirheistä ja tarkemmin sanottuna *syntaksivirheistä.*

syntaksi = tietyn ohjelmointikielen (esim. Javan) kielioppisäännöstö

2.3.1 Virhetyypit

Ohjelmointivirheet voidaan jakaa karkeasti syntaksivirheisiin ja loogisiin virheisiin.

Syntaksivirhe estää ohjelman kääntymisen vaikka merkitys eli *semantiikka* olisikin oikein. Siksi ne huomataankin aina viimeistään ohjelmaa käännettäessä. Syntaksivirhe voi olla esimerkiksi joku kirjoitusvirhe tai puolipisteen unohtaminen lauseen lopusta.

Loogisissa virheissä semantiikka, eli merkitys, on väärin. Ne on vaikeampi huomata, sillä ohjelma kääntyy semanttisista virheistä huolimatta. Ohjelma voi jopa näyttää toimivan täysin oikein. Jos looginen virhe ei löydy *testauksessakaan* (**testing**), voivat seuraukset ohjelmistosta riippuen olla tuhoisia. Tässä yksi esimerkki loogisesta virheestä:

http://www.youtube.com/watch?v=2eQpUgHkBcg

2.3.2 Tyhjät merkit (White spaces)

Esimerkkinämme ollut HelloWorld-ohjelma voitaisiin, ilman että sen toiminta muuttuisi, vaihtoehtoisesti kirjoittaa myös muodossa:

```
public class HelloWorld
                          {


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


}
```

tai muodossa:

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

Vaikka molemmat yllä olevista esimerkeistä ovat syntaksiltaan oikein, eli ne noudattavat Javan kielioppisääntöjä, ovat ne huomattavasti epäselvempiä lukea kuin alkuperäinen ohjelmamme. Javassa onkin sovittu ns. koodauskäytänteet (**code conventions**), jotka määrittelevät miten ohjelmakoodia tulisi kirjoittaa. Kun kaikki kirjoittavat samalla tavalla, on muiden koodin lukeminen helpompaa. Tämän monisteen esimerkit on pyritty kirjoittamaan näiden käytänteiden mukaisesti. Javan koodauskäytänteet löytyvät osoitteesta: http://java.sun.com/docs/codeconv/.

Merkkijonoja käsiteltäessä välilyönneillä, tabulaattoreilla ja rivinvaihdoilla on kuitenkin merkitystä. Vertaa alla olevia tulostuksia.

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

Yllä oleva rivi tulostaa: Hello World!, kun taas alla oleva rivi tulostaa: H e l l o   W o r l d !

```
System.out.println("H e l l o    W o r l d !");
```

2.4 Kommentointi

"Good programmers use their brains, but good guidelines save us having to think out every case." *-Francis Glassborow*

Lähdekoodia on usein vaikea ymmärtää pelkkää ohjelmointikieltä lukemalla. Tämän takia koodin sekaan voi ja pitää lisätä selosteita eli kommentteja. Kommentit ovat sekä tulevia ohjelman lukijoita/ylläpitäjiä varten että myös koodin kirjoittajaa itseään varten. Monet asiat voivat kirjoitettaessa tuntua ilmeisiltä, mutta jo viikon päästä saakin pähkäillä, että miksihän tuonkin tuohon kirjoitin.

Kääntäjä jättää kommentit huomioimatta, joten ne eivät vaikuta ohjelman toimintaan. Javassa on kolmenlaisia kommentteja.[DEI][KOS]

```
// Yhden rivin kommentti
```

Yhden rivin kommentti alkaa kahdella vinoviivalla (`//`). Sen vaikutus kestää koko rivin loppuun.

```
/* Tämä
   kommentti
   on usean
   rivin
   pituinen */
```

vinoviivalla ja asteriskilla alkava (`/*`) kommentti kestää niin kauan kunnes vastaan tulee asteriski ja vinoviiva (`*/`).

2.4.1 Dokumentointi

Kolmas kommenttityyppi on *dokumentaatiokommentti*. Dokumentaatiokommenteissa on tietty syntaksi ja tätä noudattamalla voidaan dokumentaatiokommentit muuttaa HTML-sivuksi javadoc-työkalua käyttämällä. Dokumentaatiokommentteja sanotaankin usein myös Javadoc-kommenteiksi.

Dokumentaatiokommentti olisi syytä kirjoittaa ennen jokaista luokkaa, pääohjelmaa, aliohjelmaa ja metodia (aliohjelmista ja metodeista puhutaan myöhemmin). Lisäksi jokainen Java-tiedosto alkaa aina dokumentaatiokommentilla, josta selviää tiedoston tarkoitus, tekijä ja versio.

Dokumentaatiokommentti alkaa aina vinoviivalla ja kahdella asteriskilla (`/**`). Jokainen seuraava dokumentaatiokommenttirivi aloitetaan asteriskilla (`*`). Dokumentaatiokommentti lopetetaan kuten tavallinen monen rivin kommentti asteriskilla ja vinoviivalla (`*/`).

```
/**
 *  Tämä
 *  on
 *  Dokumentaatiokommentti
 */
```

Dokumentaatiokommenteissa javadoc-työkalulle annetaan lisätietoja javadoc-tagien avulla. Javadoc-tagit alkavat '@'-merkillä, jonka perään tulee tagin nimi. Javadoc-tageja ovat esimerkiksi `@author`, jolla ilmoitetaan ohjelman oikeuksien haltija. Tiedot kaikista tageista löytyy Javan dokumentaatiosta:

http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/javadoc.html#javadoctags.

Voisimme kirjoittaa nyt Javadoc-kommentin HelloWorld-ohjelman alkuun seuraavasti:

```
/**
 * Esimerkkiohjelma, joka tulostaa tekstin "Hello World!"
 *
 * @author Martti Hyvönen
 */
public class HelloWorld {

  /**
   * Pääohjelmassa tulostetaan Hello World!
   * @param args ei käytössä
   */
  public static void main(String args[]) { // suoritus alkaa siis tästä
    System.out.println("Hello World!"); // tämä lause tulostaa
  }
}
```

Dokumentaatiokommentin alussa kerrotaan dokumentoitavan kohteen tarkoitus. Ensimmäisen virkkeen pitäisi vielä olla lyhyt ja ytimekäs kuvaus tästä tarkoituksesta, sillä jossain dokumentaation tiivistelmissä näytetään vain tuo ensimmäinen virke. [DOC]

Dokumentointi on keskeisin osa ohjelmistotyötä. Dokumentointi helpottaa tulevien käyttäjien ja ylläpitäjien tehtävää. Tämä on erittäin tärkeää sillä 40-60% ylläpitäjien ajasta kuluu muokattavan ohjelman ymmärtämiseen. [KOSK][KOS]

3. Algoritmit

"First, solve the problem. Then, write the code." - John Johnson

3.1 Mikä on algoritmi?

Pyrittäessä kirjoittamaan koneelle kelpaavia ohjeita joudutaan suoritettavana oleva toimenpide kirjaamaan sarjana yksinkertaisia toimenpiteitä. Toimenpidesarjan tulee olla yksikäsitteinen, eli sen tulee joka tilanteessa tarjota yksi ja vain yksi tapa toimia, eikä siinä saa esiintyä ristiriitaisuuksia. Yksikäsitteistä kuvausta tehtävän ratkaisuun tarvittavista toimenpiteistä kutsutaan algoritmiksi.

Ohjelman kirjoittaminen voidaan aloittaa hahmottelemalla tarvittavat algoritmit eli kirjaamalla lista niistä toimenpiteistä, joita tehtävän suoritukseen tarvitaan:

```
Kahvinkeitto:

1.  Täytä pannu vedellä.
2.  Keitä vesi.
3.  Lisää kahvijauhot.
4.  Anna tasaantua.
5.  Tarjoile kahvi.
```

Algoritmi on yleisesti ottaen mahdollisimman pitkälle tarkennettu toimenpidesarja, jossa askel askeleelta esitetään yksikäsitteisessä muodossa ne toimenpiteet, joita asetetun ongelman ratkaisuun tarvitaan.

3.2 Tarkentaminen

Kun tarkastellaan lähes mitä tahansa tehtävänantoa, huomataan, että tehtävän suoritus koostuu selkeästi toisistaan eroavista osatehtävistä. Se, miten yksittäinen osatehtävä ratkaistaan, ei vaikuta muiden osatehtävien suorittamiseen. Vain sillä, että kukin osasuoritus tehdään, on merkitystä. Kahvinkeitossa jokainen osatehtävä voidaan jakaa edelleen osasiin:

```
Kahvinkeitto:

1.  Täytä pannu vedellä:
  1.1.  Pistä pannu hanan alle.
  1.2.  Avaa hana.
  1.3.  Anna veden valua, kunnes vettä on riittävästi.
2.  Keitä vesi:
  2.1.  Aseta pannu hellalle.
  2.2.  Kytke virta keittolevyyn.
  2.3.  Anna lämmetä, kunnes vesi kiehuu.
3.  Lisää kahvijauhot:
  3.1.  Mittaa kahvijauhot.
  3.2.  Sekoita kahvijauhot kiehuvaan veteen.
4.  Anna tasaantua:
  4.1.  Odota, kunnes suurin osa kahvijauhoista on vajonnut
        pannun pohjalle.
5.  Tarjoile kahvi:
  5.1.  Tämä sitten onkin jo oma tarinansa...
```

Edellä esitetyn kahvinkeitto-ongelman ratkaisu esitettiin jakamalla ratkaisu viiteen osavaiheeseen. Ratkaisun algoritmi sisältää viisi toteutettavaa lausetta. Kun näitä viittä lausetta tarkastellaan lähemmin, osoittautuu, että niistä kukin on edelleen jaettavissa osavaiheisiin, eli ratkaisun pääalgoritmi voidaan jakaa edelleen alialgoritmeiksi, joissa askel askeleelta esitetään, kuinka kukin osatehtävä ratkaistaan.

Algoritmien kirjoittaminen osoittautuu hierarkkiseksi prosessiksi, jossa aluksi tehtävä jaetaan osatehtäviin, joita edelleen tarkennetaan, kunnes kukin osatehtävä on niin yksinkertainen, ettei sen suorittamisessa enää ole mitään moniselitteistä.

3.3 Yleistäminen

Eräs tärkeä algoritmien kirjoittamisen vaihe on yleistäminen. Tällöin valmiiksi tehdystä algoritmista pyritään paikantamaan kaikki alunperin annetusta tehtävästä riippuvat tekijät, ja pohditaan voitaisiinko ne kenties kokonaan poistaa tai korvata joillakin yleisemmillä tekijöillä.

Tarkastele edellä esitettyä algoritmia kahvin keittämiseksi ja luo vastaava algoritmi teen keittämiseksi. Vertaile algoritmeja: mitä samaa ja mitä eroa niissä on? Onko mahdollista luoda algoritmi, joka yksiselitteisesti selviäisi sekä kahvin että teen keitosta? Onko mahdollista luoda algoritmi, joka saman tien selviytyisi maitokaakosta ja rommitotista?

3.4 Peräkkäisyys

Kuten luvussa 1 olevassa reseptissä ja muissakin ihmisille kirjoitetuissa ohjeissa, niin myös tietokoneelle esitetyt ohjeet luetaan ylhäältä alaspäin ellei muuta ilmoiteta. Esimerkiksi ohjeen lumiukon piirtämisestä voisi esittää yksinkertaistettuna alla olevalla tavalla.

```
Piirrä säteeltään 20cm kokoinen ympyrä koordinaatiston pisteeseen (20,80)
Piirrä säteeltään 15cm kokoinen ympyrä edellisen ympyrän päälle
Piirrä säteeltään 10cm kokoinen ympyrä edellisen ympyrän päälle
```

Yllä oleva koodi ei ole vielä mitään ohjelmointikieltä, mutta se sisältää jo ajatuksen siitä kuinka lumiukko voitaisiin tietokoneella piirtää. Piirrämme lumiukon Java-ohjelmointikielellä seuraavassa luvussa.

4. Yksinkertainen graafinen Java-ohjelma

Seuraavassa esimerkissä käytetään Jyväskylän Yliopiston piirtämistä helpottavaa grafiikkakirjastoa. Kirjaston voit ladata koneelle osoitteesta https://trac.cc.jyu.fi/projects/ohj1/wiki/graphics, josta löytyy myös ohjeet kirjaston asennukseen ja käyttöön.

4.1 Mikä on kirjasto?

Java-ohjelmat koostuvat luokista. Luokat taas sisältävät metodeja (ja aliohjelmia), jotka suorittavat tehtäviä ja mahdollisesti palauttavat arvoja suoritettuaan näitä tehtäviä. Metodi voisi esimerkiksi laskea kahden luvun summan ja palauttaa tuloksen tai piirtää ohjelmoijan haluaman kokoisen ympyrän. Samaan asiaan liittyviä metodeja kootaan luokkaan ja luokkia kootaan edelleen kirjastoiksi. Idea kirjastoissa on, ettei kannata tehdä uudelleen sitä minkä joku on jo tehnyt. Toisin sanoen, pyörää ei kannata keksiä uudelleen. Java-ohjelmoijan kannalta oleellisin kirjasto on Javan luokkakirjasto – API (**Application programming interface**). APIn *dokumentaatioon* (**documentation**) kannattaa tutustua sillä sieltä löytyy monia todella hyödyllisiä metodeja. Dokumentaatio löytyy Sunin sivuilta osoitteesta http://java.sun.com/reference/api/. Valitse oman Java-versiosi mukainen dokumentaatio.[DEI][KOS]

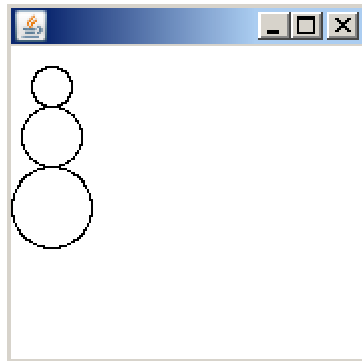
dokumentaatio = Sisältää tiedot kaikista kirjaston luokista ja niiden metodeista (ja aliohjelmista). Löytyy useimmiten ainakin web-muodossa.

4.2 Esimerkki piirtämisestä Jyväskylän yliopiston Graphics-kirjastolla

Piirretään lumiukko käyttämällä Graphics-kirjastoa.

```
// Otetaan graphics-kirjaston EasyWindow-luokka käyttöön
import fi.jyu.mit.graphics.EasyWindow;

/**
 * Luokka, jossa harjoitellaan piirtelyä EasyWindow-luokkaa käyttämällä
 * @author vesal
 * @version 14.9.2008
 */
public class Lumiukko {

   /**
    * Pääohjelmassa piirretään yksi lumiukko
    * @param args ei käytössä
    */
   public static void main(String[] args) {
      EasyWindow window = new EasyWindow();

      window.addCircle(20,80-2*15-20-10,10);
      window.addCircle(20,80-15-20,15);
      window.addCircle(20,80,20);

      window.showWindow();
   }
}
```

Ajettaessa ohjelman tulisi piirtää yksinkertainen lumiukko ikkunan vasempaan yläreunaan, kuten alla olevassa kuvassa.



Kuva 2: Lumiukko EasyWindow-luokalla piirrettynä

4.2.1 Ohjelman suoritus

Ohjelman suoritus aloitetaan aina pääohjelmasta ja sitten edetään rivi riviltä ylhäältä alaspäin ohjelman loppuun niin kauan kuin lauseita riittää. Ohjelmassa voi olla myös rakenteita, joissa toistetaan tiettyjä rivejä useampaan kertaan vain muuttamalla jotain arvoa tai arvoja. Pääohjelmassa voi olla myös aliohjelmakutsuja jolloin hypätään pääohjelmasta suorittamaan aliohjelmaa ja palataan sitten takaisin pääohjelman suoritukseen. Aliohjelmista puhutaan enemmän luvussa 6. Aliohjelmat

4.2.2 Ohjelman oleellisemmat kohdat

Tarkastellaan ohjelman oleellisempia kohtia.

```
import fi.jyu.mit.graphics.EasyWindow;
```

Aluksi meidän täytyy kertoa kääntäjälle, että haluamme ottaa käyttöön `graphics`-kirjaston `EasyWindow`-luokan. Nyt tämän luokan metodit ovat käytettävissämme. Kahdella seuraavalla rivillä joilla on tekstiä luodaan `Lumiukko`-luokka ja aloitetaan pääohjelma kuten ensimmäisessä HelloWorld-esimerkissämme.

```
EasyWindow window = new EasyWindow();
```

Tällä rivillä luodaan `EasyWindow`-ikkuna, joka tallennetaan `window`-nimiseen muuttujaan. Tähän ikkunaan voimme myöhemmin piirtää sitten lumiukon. Tällä rivillä käytetään myös ensimmäistä kertaa Graphics-kirjastoa. Tarkemmin sanottuna luomme uuden `EasyWindow`-olion eli `EasyWindow`-luokan ilmentymän. Olioista puhutaan lisää luvussa 8 Oliotietotyypit.

```
    window.addCircle(20,80-2*15-20-10,10);
    window.addCircle(20,80-15-20,15);
    window.addCircle(20,80,20);
```

Kolme seuraavaa lausetta luovat lumiukon tekemällä kolme ympyrää. Ympyrä luodaan `EasyWindow`-luokan `addCircle`-metodilla. Ympyrä luodaan nyt aikaisemmin luomaamme `window`-nimiseen ikkunaan. Koska `addCircle` on `EasyWindow`-luokan metodi, täytyy meidän ensiksi kirjoittaa luomamme `EasyWindow`-ikkunan nimi. Sen perään tulee piste, jonka jälkeen kirjoitetaan haluamamme metodi eli tässä tapauksessa `addCircle`-metodi. Jotta saisimme haluamamme kokoisen ympyrän haluamaamme paikkaan, täytyy meidän vielä antaa `addCircle`-metodille tiedot ympyrän koosta ja paikasta. Metodeille annettavia tietoja sanotaan *parametreiksi* (**parameter**). Käyttämällämme `addCircle`-metodilla on kolme parametria. Ympyrän keskipisteen x-koordinaatti, ympyrän keskipisteen y-koordinaatti ja ympyrän säde. Parametrit kirjoitetaan metodin nimen perään sulkeisiin ja ne erotetaan toisistaan pilkuilla.

```
metodinNimi(parametri1, parametri2,..., parametriX);
```

Vertaa yllä olevaa yleistä muotoa `addCircle`-metodiin alla.

```
addCircle(ympyrän x-koordinaatti, ympyrän y-koordinaatti, ympyrän säde);
```

Saadaksemme ympyrät piirrettyä oikeille paikolle, täytyy meidän laskea koordinaattien paikat. Oletuksena ikkunan vasen yläkulma on koordinaatiston piste (0,0) x:n arvot kasvavat oikealle ja y:n arvot alaspäin.

Esimerkissä koordinaattien laskemiseen on käytetty Javan aritmeettisia operaatioita. Voisimme tietenkin laskea koordinaattien pisteet myös itse, mutta miksi tehdä niin jos tietokone voi laskea pisteet puolestamme? Laskutoimituksia tehdään Javan aritmeettisilla operaatioilla. Perusoperaatiot ovat summa (+), vähennys (-), kerto (*), jako (-) ja jakojäännös (%). Aritmeettisista operaatioista puhutaan lisää muuttujien yhteydessä kohdassa 7.5 Aritmeettiset lausekkeet.

Mieti, mikä käyttämistämme `addCircle`-metodeista piirtää minkäkin ympyrän?

Lumiukon alimmainen ympyrä piirretään pisteeseen (20,80) ja sen säde on 20. Se piirretään siis lauseella:

```
window.addCircle(20,80,20);
```

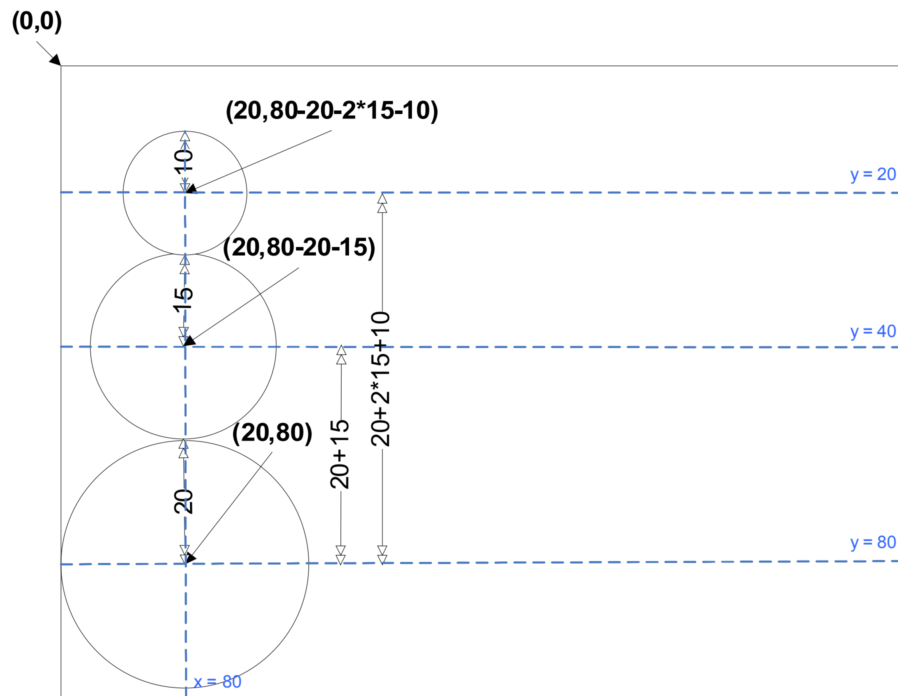
Keskimmäinen ympyrä tulee alimman ympyrän päälle, eli ympyrät sivuavat toisiaan. Keskimmäisen ympyrän keskipiste tulee siis kohtaan (20,80- alimman ympyrän säde - keskimmäisen ympyrän säde) . Kun haluamme, että keskimmäisen ympyrän säde on 15, niin silloin keskimmäisen ympyrän keskipiste tulee kohtaan (20,80-20-15) ja se piirretään lauseella:

```
window.addCircle(20,80-15-20,15);
```

Ylin ympyrä sivuaa sitten taas keskimmäistä ympyrää, joten sen keskipiste tulee kohtaan (20,80- alimman ympyrän säde - keskimmäisen ympyrän halkaisija - ylimmän ympyrän säde). Kun haluamme ylimmän ympyrän säteen olevan 10, niin sen keskipiste tulee pisteeseen (20,80-20-2*15-10) ja se piirretään lauseella:

```
window.addCircle(20,80-2*15-20-10,10);
```

Seuraava kuva selventänee ympyröiden keskipisteiden laskentaa.



Kuva 3: Lumiukon pallojen keskipisteiden laskeminen

Kaikki tiedot luokista, luokkien metodeista sekä siitä mitä parametreja metodeille tulee antaa löydät käyttämäsi kirjaston dokumentaatiosta. Käyttämämme Graphics-luokan dokumentaatio löytyy osoitteesta: http://users.jyu.fi/~vesal/kurssit/ohj1/graphics/.

Etsi `Graphics`-luokan dokumentaatiosta `EasyWindow`-luokka. Mitä tietoa löydät `addCircle`-metodista? Mitä muita metodeita luokassa on?

Luomamme ”window”-niminen ikkuna ei oletuksena ole näkyvillä ennen kuin kerromme sille, että se saa näkyä. Ikkuna pistetään näkymään metodilla `showWindow()`. Ikkunamme saadaan siis näkymään seuraavalla lauseella:

```
window.showWindow();
```

Ikkunan voisi laittaa näkymään jo heti sen luomisen jälkeen, mutta tällöin prosessorin nopeudesta riippuen näkisimme kun ympyrät piirretään, jota emme välttämättä halua.

5. Lähdekoodista prosessorille

5.1 Kääntäminen

Tarkastellaan nyt tarkemmin sitä kuinka Java-lähdekoodi muuttuu lopulta prosessorin ymmärtämään muotoon. Aluksi ohjelmoija luo jollain editorilla ohjelman lähdekoodin, joka tallennetaan aina muodossa ”.java”. Lähdekoodi käännetään Javan kääntäjällä (**Java compiler**) *tavukoodiksi*. Tavukoodi-tiedoston pääte on ”.class”. Ennen kääntämistä kääntäjä kuitenkin tarkastaa, että koodi on syntaksiltaan oikein. [VES][KOS]

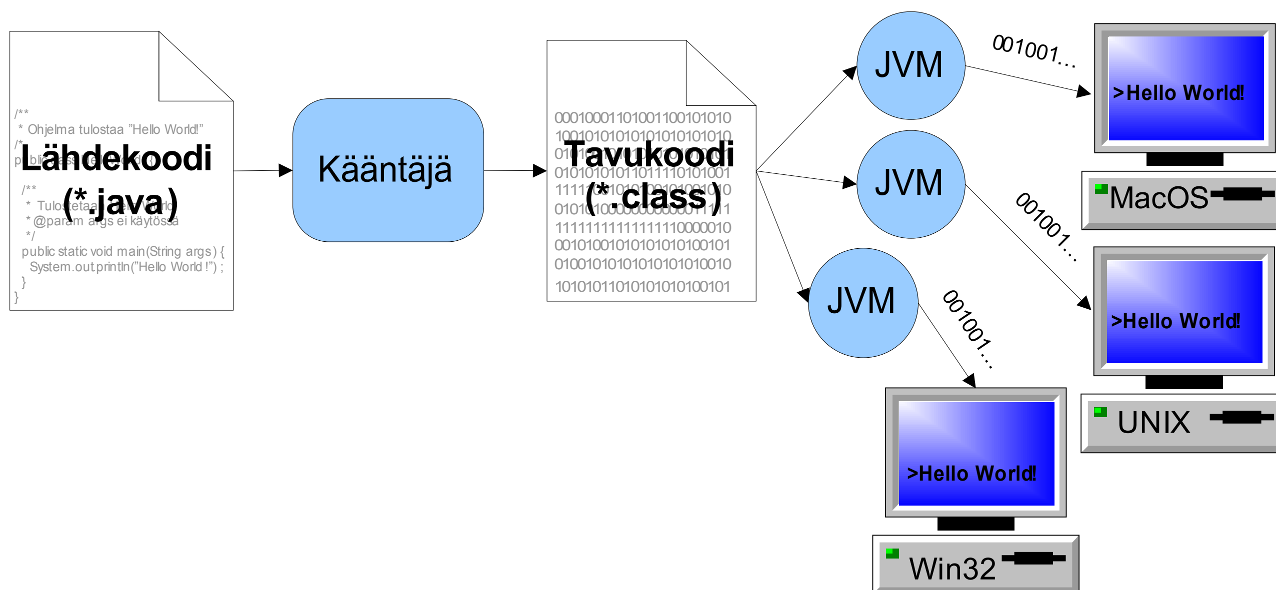
Kääntäminen tehtiin Windowsissa komentorivillä (**Command Prompt**) käyttämällä komentoa ”javac tiedostonNimi.java”.

5.2 Suorittaminen

Tavukoodi on käyttöjärjestelmäriippumatonta koodia, jonka suorittamiseen tarvitaan Java-virtuaalikonetta (**Java Virtual Machine**). Java-virtuaalikone on oikeaa tietokonetta matkiva ohjelma, joka tulkkaa tavukoodia ja suorittaa sitä sitten kohdekoneen prosessorilla. Tässä on merkittävä ero perinteisiin käännettäviin kieliin (esimerkiksi C ja C++), joissa käännös on tehtävä erikseen jokaiselle eri laitealustalle.

Jokaisella käyttöjärjestelmällä (Windows, Linux, Macintosh, mobiililaitteet, ym.) on oltava oma Java-virtuaalikone, koska jokainen käyttöjärjestelmä toimii eri lailla. Tämän ansiosta Java-ohjelmat kuitenkin toimivat kaikissa järjestelmissä joille on saatavilla Java-virtuaalikone. [VES][KOS]

Ohjelman suorittaminen tehtiin Windowsin komentokehotteella komennolla ”java OhjelmanNimi”. Kääntäjä ja Java-virtuaalikone tulevat Java-sovelluskehittimien mukana (esim. JDK).



Kuva 4: Lähdekoodista toimivaksi ohjelmaksi. JVM = Java Virtual Machine eli Java-virtuaalikone.

6. Aliohjelmat

“Copy and paste is a design error.” - David Parnas

Pääohjelman lisäksi ohjelma voi sisältää muitakin aliohjelmia. Aliohjelmaa *kutsutaan* pääohjelmasta, metodista tai toisesta aliohjelmasta suorittamaan tiettyä tehtävää. Aliohjelmat voivat saada parametreja ja palauttaa arvon, kuten metoditkin. Pohditaan seuraavaksi mihin aliohjelmia tarvitaan.

Jos tehtävänämme olisi piirtää useampi lumiukko, niin tämänhetkisellä tietämyksellämme tekisimme todennäköisesti jonkin alla olevan kaltaisen ratkaisun.

```
import fi.jyu.mit.graphics.EasyWindow;

/**
 * Harjoitellaan piirtämistä EasyWindow-luokan avulla.
 * @author vesal
 * @version 14.9.2008
 */
public class Lumiukot {

   /**
    * Piirretään kaksi lumiukkoa.
    * @param args ei käytössä
    */
    public static void main(String[] args) {
        EasyWindow window = new EasyWindow();

        //Ensimmäinen lumiukko
        window.addCircle(20,80-2*15-20-10,10);
        window.addCircle(20,80-15-20,15);
        window.addCircle(20,80,20);

        //Toinen lumiukko
        window.addCircle(70,90-2*15-20-10,10);
        window.addCircle(70,90-15-20,15);
        window.addCircle(70,90,20);

        window.showWindow();
    }

}
```

Huomataan, että ensimmäisen ja toisen lumiukon piirtäminen tapahtuu lähes samanlaisilla koodinpätkillä. Vain `addCircle`-metodin saamat parametrit muuttuvat hieman.

Toisaalta voisimme kirjoittaa koodin myös niin, että lumiukon alimman pallon keskipiste tallennetaan muuttujiin `x` ja `y`. Näiden pisteiden avulla voimme sitten laskea muiden pallojen paikat.

```
  double x, y;

  x=20; y=80;
  window.addCircle(x,y-2*15-20-10,10);
  window.addCircle(x,y-15-20,15);
  window.addCircle(x,y,20);

  x=70; y=90;
  window.addCircle(x,y-2*15-20-10,10);
  window.addCircle(x,y-15-20,15);
  window.addCircle(x,y,20);
```

Tarkastellaan nyt muutoksia hieman tarkemmin.

```
double x, y;
```

Yllä olevalla rivillä esitellään kaksi *liukuluku*tyyppistä *muuttujaa*. Liukuluku on eräs tapa esittää reaalilukuja tietokoneissa. Javassa jokaisella muuttujalla on oltava tyyppi ja eräs liukulukutyyppi Javassa on `double`. Muuttujista ja niiden tyypeistä puhutaan lisää luvussa 7 Muuttujat.

liukuluku (**floating point**) = Liukuluku on tietokoneissa käytettävä esitysmuoto reaaliluvuille. Tarkempaa tietoa liukuluvusta löytyy monisteen lopusta luvusta "Reaalilukujen esitys tietokoneessa".

```
x=20; y=80;
```

Yllä olevalla rivillä on kaksi lausetta. Ensimmäisellä asetetaan muuttujaan `x` arvo 20 ja toisella muuttujaan `y` arvo 80. Nyt voimme käyttää lumiukon pallojen laskentaan näitä muuttujia.

```
x=70; y=90;
```

Vastaavasti yllä olevalla rivillä asetetaan nyt muuttujiin uudet arvot, joita käytetään seuraavan lumiukon pallojen paikkojen laskemiseen.

Näiden muutosten jälkeen molempien lumiukkojen varsinainen piirtäminen tapahtuu nyt täysin samalla koodinpätkällä:

```
window.addCircle(x,y-2*15-20-10,10);
window.addCircle(x,y-15-20,15);
window.addCircle(x,y,20);
```

Uusien lumiukkojen piirtäminen olisi nyt jonkin verran helpompaa, sillä meidän ei tarvitse kuin ilmoittaa ennen piirtämistä uuden lumiukon paikka ja varsinaisen lumiukkojen piirtäminen onnistuisi kopioimilla ja liittämällä koodia (**copy-paste**). Kuitenkin aina kun kopioi ja liittää koodia pitäisi pysähtyä miettimään, että onko tässä mitään järkeä?

Kahden lumiukon tapauksessa tämä vielä onnistuu ilman, että koodin määrä kasvaa kohtuuttomasti, mutta entä jos meidän pitäisi piirtää 10 tai 100 lumiukkoa? Kuinka monta riviä ohjelmaan tulisi silloin? Kun lähes samanlainen koodinpätkä tulee useampaan kuin yhteen paikkaan, on useimmiten syytä muodostaa siitä oma aliohjelma. Koodin monistaminen moneen paikkaan lisäisi vain koodirivien määrää, tekisi ohjelman ymmärtämisestä vaikeampaa ja vaikeuttaisi testaamista. Lisäksi jos monistetussa koodissa olisi vikaa, jouduttaisiin korjaukset tekemään myös useampaan paikkaan. Hyvän ohjelman yksi mitta (kriteeri) onkin, että jos jotain pitää muuttaa, niin muutokset kohdistuvat vain yhteen paikkaan.

6.1 Aliohjelman kutsuminen

Haluamme siis aliohjelman, joka piirtää meille lumiukon tiettyyn ikkunan pisteeseen. Kuten metodeille, myös aliohjelmalle viedään parametrien avulla sen tarvitsemaa tietoa. Parametreina tulisi viedä vain minimaalisimmat tiedot, joilla aliohjelman tehtävä saadaan suoritettua.

Sovitaan, että aliohjelmamme piirtää aina samankokoisen lumiukon haluamaamme `EasyWindow`-ikkunan pisteeseen. Mitkä ovat minimaalisimmat aliohjelman tarvitsemat tiedot, joiden avulla lumiukko saadaan piirrettyä?

Aliohjelma tarvitsee ainakin tiedon mihin pisteeseen lumiukko piirretään. Viedään parametrina lumiukon alimman pallon keskipiste. Muiden pallojen paikat voidaan laskea tämän pisteen avulla. Lisäksi meidän täytyy viedä parametrina ikkuna, johon lumiukko piirretään. Jos ikkuna luotaisiin vasta aliohjelman sisällä, piirtäisi aliohjelma aina jokaisen lumiukon omaan ikkunaansa. Nämä parametrit riittävätkin jo lumiukon piirtämiseen. Aliohjelma tarvitsee siis kolme parametria: ikkuna johon lumiukko piirretään, lumiukon alimman pallon x-koordinaatti ja lumiukon alimman pallon y-koordinaatti.

Kun aliohjelmaa käytetään ohjelmassa sanotaan, että aliohjelmaa kutsutaan. Kutsu tapahtuu kirjoittamalla aliohjelman nimi ja antamalla sille parametrit. Aliohjelmakutsun erottaa metodikutsusta vain se, että metodia kutsuttaessa täytyy ensiksi kirjoittaa sen olion nimi, jonka

metodia kutsutaan. Esimerkiksi `addCircle`-metodia kutsuttaessa piti ensiksi kirjoittaa sen ikkunan nimi, johon ympyrä haluttiin piirtää.

Päätetään, että aliohjelman nimi on `lumiukko`. Päätetään lisäksi, että aliohjelman ensimmäinen parametri on ikkuna johon lumiukko piirretään, toinen parametri lumiukon alimman pallon keskipisteen x-koordinaatti ja kolmas parametri lumiukon alimman pallon keskipisteen y-koordinaatti. Tällöin `window`-ikkunaan voitaisiin piirtää lumiukko, jonka alimman pallon keskipiste on (20,80), seuraavalla kutsulla:

```
lumiukko(window,20,80);
```

Kutsussa voisi myös ensiksi mainita sen luokan nimen mistä aliohjelma löytyy. Tällä tavalla aliohjelmaa voisi kutsua myös muista luokista, koska määrittelimme sen julkiseksi (**public**).

```
Lumiukot.lumiukko(window,20,80);
```

Vaikka tämä muoto muistuttaa jo melko paljon metodin kutsua on ero kuitenkin selvä. Metodia kutsuttaessa toimenpide tehdään aina tietylle oliolle, kuten `window.addCircle(20,80,20)` lisää ympyrän juuri siihen ikkunaan, johon `window`-olio viittaa. Ikkunoita voisi olla myös muita erinimisiä. Alla olevassa aliohjelmakutsussa kuitenkin käytetään vain luokasta `Lumiukot` löytyvää `lumiukko`-aliohjelmaa.

Jos olisimme toteuttaneet jo varsinaisen aliohjelman piirtäisi seuraava pääohjelma meille kaksi lumiukkoa:

```
    /*
     * Pääohjelmassa kokeillaan piirtelyä.
     *
     * @param arg ei käytössä
     */
    public static void main(String[] args) {
         EasyWindow window = new EasyWindow();
         lumiukko(window,20,80);
         lumiukko(window,70,90);
         window.showWindow();
    }
```

Seuraavaksi meidän täytyy toteuttaa itse aliohjelma, jotta kutsut alkavat toimimaan.

6.2 Aliohjelman kirjoittaminen

Varsinaista aliohjelman toiminnallisuuden kirjoittamista sanotaan aliohjelman määrittelyksi tai esittelyksi (**declaration**). Kirjoitetaan nyt määrittely aliohjelmalle, jonka kutsun jo teimme. Monesti on viisasta suunnitella aliohjelmakutsu ensiksi, kirjoittaa se paikalleen ja toteuttaa varsinainen aliohjelman kirjoittaminen vasta myöhemmin. Monet työkalut osaavatkin luoda meille valmiiksi rungon aliohjelman toteutusta varten, jos olemme kirjoittaneet sille ensiksi kutsun. Katso kohta 10.3.2 Quick Fix .

```
public static void lumiukko(EasyWindow window, double x, double y) {
   window.addCircle(x,y-2*15-20-10,10);
   window.addCircle(x,y-15-20,15);
   window.addCircle(x,y,20);
}
```

Aliohjelman toteutuksen ensimmäistä riviä sanotaan aliohjelman *otsikoksi* (**header**) tai esittelyriviksi. Otsikon alussa määritellään aliohjelman *näkyvyys* julkiseksi (**public**). Kun näkyvyys on julkinen, niin aliohjelmaa voidaan kutsua eli käyttää myös muissa luokissa. Aliohjelma määritellään myös staattiseksi (**static**). Kaikki aliohjelmat ovat staattisia - jos emme määrittelisi aliohjelmaa staattiseksi, olisi se oikeastaan metodi, eli olion toiminto. Olioista ja metodeista puhutaan lisää luvussa 8 Oliotietotyypit. Aliohjelmalle on annettu myös määritys `void`, joka

tarkoittaa sitä, että aliohjelma ei palauta mitään arvoa. Aliohjelma voisi nimittäin myös lopettaessaan palauttaa jonkun arvon, jota tarvitsisimme ohjelmassamme. Tällaisista aliohjelmista puhutaan luvussa 9 Aliohjelman paluuarvo. `void`-määrityksen jälkeen aliohjelmalle on annettu nimeksi ”lumiukko”.

Aliohjelman nimen jälkeen ilmoitetaan sulkeiden sisässä aliohjelman parametrit. Jokaista parametria ennen on ilmoitettava myös parametrin *tietotyyppi*. Ensimmäinen parametri oli ikkuna johon lumiukko piirretään. Sen nimi on ”`window`” ja tietotyyppi `EasyWindow`. Seuraavat parametrit olivat alimman lumiukon x- ja y-koordinaatit. Molempien tietotyyppi on "`double`" ja nimet kuvaavasti ”`x`” ja ”`y`”. Tietotyyppi `double` tarkoittaa liukulukua. Muista tietotyypeistä voit lukea kohdasta 7.1.1 Javan alkeistietotyypit ja luvusta 8 Oliotietotyypit.

> Aliohjelman parametrien nimien ei tarvitse olla samoja kuin kutsussa. Niiden nimet kannattaa kuitenkin olla mahdollisimman kuvaavia.

Varsinaista aliohjelman toiminnallisuutta kirjoittaessa käytämme nyt parametreille antamiamme nimiä. Alimman ympyrän keskipisteen koordinaatit saamme nyt suoraan parametreista `x` ja `y`, mutta muiden ympyröiden keskipisteet meidän täytyy laskea alimman ympyrän koordinaateista. Tämä tapahtuu täysin samalla tavalla kuin edellisessä esimerkissä. Itse asiassa, jos vertaa aliohjelman lohkon sisältöä edellisen esimerkin koodiin, on se täysin sama.

Javassa on tapana aloittaa aliohjelmien ja metodien nimet pienellä kirjaimella ja nimessä esiintyvä jokainen uusi sana alkamaan isolla kirjaimella (kirjoitustavasta käytetään termiä **lower camel case**). Jos haluaisimme antaa aliohjelman nimeksi "piirrä komea lumiukko", kirjoitettaisiin se siis muodossa `piirraKomeaLumiukko`.

> Luokkien nimet alkavat isolla kirjaimella, aliohjelmien, muuttujien yms. nimet pienellä.

Tarkastellaan seuraavaksi mitä aliohjelmakutsussa tapahtuu.

```
lumiukko(window,20,80);
```

Yllä olevalla kutsulla sijoitetaan aliohjelman `window`-nimiseen muuttujaan, pääohjelman `window`-niminen muuttuja. Kutsussa olevan muuttujan nimi ei tarvitse olla sama kuin aliohjelman otsikossa määritelty muuttujan nimi. Lisäksi aliohjelman muuttujaan `x` sijoitetaan arvo 20 (liukulukuun voi sijoittaa kokonaislukuarvon) ja aliohjelman muuttujaan `y` arvo 80. Aliohjelmakutsun suorituksessa lasketaan siis ensiksi jokaisen kutsussa olevan *lausekkeen* arvo ja sitten lasketut arvot sijoitetaan kutsussa olevassa järjestyksessä aliohjelman vastinparametreille. Siksi vastinparametrien pitää olla sijoitusyhteensopivia kutsun lausekkeiden kanssa. Esimerkin kutsussa lausekkeet ovat yksinkertaisimpia mahdollisia: muuttujan nimi, kokonaislukuarvo 20 ja kokonaislukuarvo 80. Ne voisivat kuitenkin olla kuinka monimutkaisia lausekkeita tahansa, esimerkiksi:

```
lumiukko(window,22.7+sin(2.4),80.1-Math.PI);
```

> Lause (**statement**) ja lauseke (**expression**) ovat eri asia. Lauseke on arvojen, aritmeettisten operaatioiden ja aliohjelmien (tai metodien yhdistelmä), joka evaluoituu tietyksi arvoksi. Lauseke on siis lauseen osa. Seuraava kuva selventää eroa.

Kuva 5: Lauseen ja lausekkeen ero

Koska määrittelimme koordinaattien parametrien tyypiksi `double`, voisimme yhtä hyvin antaa parametreiksi desimaalilukuja. Täytyy vain muistaa, että Javassa desimaaliluvuissa käytetään pistettä erottamaan kokonaisosa desimaaliosasta.

Kokonaisuudessaan ohjelma näyttää nyt seuraavalta:

```
import fi.jyu.mit.graphics.EasyWindow;

/**
 * Piirretään lumiukkoja
 * @author vesal
 * @version 14.9.2008
 */
public class Lumiukot {

   public static void lumiukko(EasyWindow window, double x, double y) {
      window.addCircle(x,y-2*15-20-10,10);
      window.addCircle(x,y-15-20,15);
      window.addCircle(x,y,20);
   }

   /*
    * Pääohjelmassa kokeillaan piirtelyä.
    *
    * @param args ei käytössä
    */
   public static void main(String[] args) {
      EasyWindow window = new EasyWindow();
      lumiukko(window,20,80); // 1. kutsu
      lumiukko(window,70,90); // 2. kutsu
      window.showWindow();
   }

}
```

Kutsuttaessa aliohjelmaa hyppää ohjelman suoritus välittömästi parametrien sijoitusten jälkeen kutsuttavan aliohjelman ensimmäiselle riville ja alkaa suorittamaan aliohjelmaa kutsussa määritellyillä parametreilla. Kun päästään aliohjelman koodin loppuun palataan jatkamaan kutsun jälkeisestä seuraavasta lausekkeesta. Esimerkissämme kun ensimmäinen lumiukko on piirretty, palataan tavallaan 1. kutsun puolipisteeseen ja sitten pääohjelma jatkuu kutsumalla toista lumiukon piirtämistä.

Jos nyt haluaisimme piirtää lisää lumiukkoja, lisäisi jokainen uusi lumiukko koodia vain yhden rivin.

Aliohjelmien käyttö selkeyttää ohjelmaa ja aliohjelmia kannattaa kirjoittaa vaikka niitä kutsuttaisiin vain yhden kerran. Hyvää aliohjelmaa voidaan kutsua muustakin käyttöyhteydestä.

6.3 Aliohjelmien dokumentointi

Jokaisen aliohjelman tulisi sisältää dokumentaatiokommentti. Aliohjelman dokumentaatiokommentin tulee sisältää ainakin: lyhyt kuvaus aliohjelman toiminnasta, selitys kaikista parametreista sekä selitys mahdollisesta paluuarvosta. Dokumentaatiokommentin ensimmäisen virkkeen täytyy olla lyhyt ja selkeä kuvaus aliohjelman toiminnasta, sillä dokumentaatiossa kohdassa Method Summary näytetään vain tuo ensimmäinen virke. Jokainen parametri selitetään oman `@param`-tagin perään ja paluuarvo `@return`-tagin perään. Tehdään Javadoc-kommentti `lumiukko`-aliohjelmalle.

```
/**
  * Piirtää ikkunaan lumiukon haluamaamme paikkaan.
  *
  * @param window ikkuna johon lumiukko piirretään
  * @param x lumiukon alimman ympyrän x-koordinaatti
  * @param y lumiukon alimman ympyrän y-koordinaatti
```

```
 */
public static void lumiukko(EasyWindow window, double x, double y) {
       window.addCircle(x,y-2*15-20-10,10);
       window.addCircle(x,y-15-20,15);
       window.addCircle(x,y,20);
}
```

Javadoc-työkalun tuottama HTML-sivu tästä luokasta näyttäisi nyt seuraavalta:

Class Lumiukko

```
java.lang.Object
  └ Lumiukko
```

```
public class Lumiukko
extends java.lang.Object
```

Constructor Summary
Lumiukko()

Method Summary	
static void	lumiukko(fi.jyu.mit.graphics.EasyWindow window, double x, double y) Piirtää ikkunaan lumiukon haluamaamme paikkaan.
static void	main(java.lang.String[] args) Pääohjelmassa kokeillaan piirtelyä

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Kuva 6: Osa `Lumiukko`-luokan dokumentaatiosta

Dokumentaatiossa näkyy kaikki luokan aliohjelmat ja metodit. Huomaa, että koska Javassa yleisesti myös aliohjelmia kutsutaan metodeiksi, löytyvät ne siksi kaikki Method Summary-osiosta.

Jokaisesta aliohjelmasta ja metodista löytyy lisäksi tarkemmat tiedot Method Detail -kohdasta. Aliohjelman `lumiukko` Method Detail -osio näyttäisi seuraavalta:

Method Detail

lumiukko

```
public static void lumiukko(fi.jyu.mit.graphics.EasyWindow window,
                            double x,
                            double y)
```

Piirtää ikkunaan lumiukon haluamaamme paikkaan.

Parameters:
`window` - ikkuna johon lumiukko piirretään
`x` - lumiukon alimman ympyrän x-koordinaatti
`y` - lumiukon alimman ympyrän y-koordinaatti

Kuva 7: Lumiukko-luokan Method Detail-osa

6.4 Aliohjelmat, metodit ja funktiot

Kuten ehkä huomasit, aliohjelmilla ja metodeilla on paljon yhteistä. Monissa Java kirjoissa nimitetään myös aliohjelmia metodeiksi. Tällöin aliohjelmat erotetaan olioiden metodeista nimittämällä niitä staattisiksi metodeiksi. Tässä monisteessa metodeista puhutaan kuitenkin vain silloin, kun tarkoitetaan olioiden toimintoja. Esimerkiksi lumiukon piirrossa piirsimme ympyrän käyttämällä `addCircle`-metodia. Se on `EasyWindow` luokan ilmentymän, eli olion toiminto. `addCircle`-metodia ei voi kutsua ilman `EasyWindow`-oliota, joka tietää, mihin ympyrä piirretään.

Aliohjelmista puhutaan tällä kurssilla, koska sitä termiä käytetään monissa muissa ohjelmointikielissä. Tämä kurssi onkin ensisijaisesti ohjelmoinnin kurssi, jossa käytetään Java-kieltä. Päätavoitteena on siis oppia ohjelmoimaan ja työkaluna meillä sen opettelussa on Java-kieli.

Aliohjelmamme `lumiukko` ei palauttanut mitään arvoa. Aliohjelmaa (tai metodia) joka palauttaa jonkun arvon voidaan kutsua myös tarkemmin *funktioksi* (**function**).

Aliohjelmia ja metodeja nimitetään eri tavoin eri kielissä. Esimerkiksi C++-kielessä sekä aliohjelmia että metodeja sanotaan funktioiksi. Metodeita nimitetään C++-kielessä tarkemmin vielä jäsenfunktioiksi.

7. Muuttujat

Muuttajat (**variable**) toimivat ohjelmassa tietovarastoina erilaisille asioille. Niihin voidaan tallentaa laskennan välituloksia, tietoa ohjelman käyttäjästä ja paljon muuta. Ilman muuttujia järkevä tiedon käsittely olisi oikeastaan mahdotonta. Oikeastihan muuttujien arvot tallennetaan keskusmuistiin tai rekistereihin. Muuttujan nimi onkin ohjelmointikielten helpotus, sillä näin ohjelmoijan ei tarvitse tietää tarvitsemansa tiedon keskusmuisti- tai rekisteriosoitetta, vaan riittää muistaa itse nimeämänsä muuttujan nimi.[VES]

7.1 Muuttujan määrittely

Javassa jokaisella muuttujalla täytyy olla tietotyyppi, joka ilmoitetaan määrittelyn yhteydessä. Tietotyyppi kertoo minkälaista tietoa muuttujaan tullaan tallentamaan. Muuttuja määritellään (**declare**) kirjoittamalla ensiksi tietotyyppi ja sen perään muuttujan nimi. Muuttujan nimet aloitetaan Javassa pienellä kirjaimella, jonka jälkeen jokainen uusi sana alkaa aina isolla kirjaimella:

```
MuuttujanTietotyyppi muuttujanNimi;
```

Esimerkiksi muuttuja, johon tullaan tallentamaan henkilön ikä voitaisiin määritellä alla olevalla tavalla.

```
int henkilonIka;
```

Samantyyppisiä muuttujia voidaan määritellä kerralla useampia erottamalla muuttujien nimet pilkulla.

```
double paino, pituus;
```

Määrittely onnistuu kuitenkin myös erikseen:

```
double paino;
double pituus;
```

7.1.1 Javan alkeistietotyypit

Javan tietotyypit voidaan jakaa alkeistietotyyppeihin (**primitive types**) ja oliotietotyyppeihin. Oliotyyppejä käsitellään myöhemmin luvussa 8 Oliotietotyypit. Oliotietotyyppeihin kuuluu muun muassa merkkijonojen tallennukseen tarkoitettu `String`-olio.

Eri tietotyypit vievät eri määrän tilaa. Nykyajan koneissa on niin paljon muistia, että ainakin Ohjelmointi 1-kurssilla kannattaa valita tietotyyppi johon varmasti mahtuu haluamamme tieto.

Javan kaikki alkeistietotyypit on alla olevassa taulukossa.

Java-merkki	**Koko**	**Selitys**	**Arvoalue**
`boolean`		kaksiarvoinen tietotyyppi	true tai false
`byte`	8 bittiä	yksi tavu	-128 - 127
`char`	16 bittiä	yksi merkki	kaikki merkit
`short`	16 bittiä	pieni kokonaisluku	-32768 - 32767
`int`	32 bittiä	keskikokoinen kokonaisluku	-2147483648 - 2147483647
`long`	64 bittiä	iso kokonaisluku	-2^{63} - 2^{63}-1
`float`	32 bittiä	liukuluku	noin 7 desimaalin tarkkuus
`double`	64 bittiä	tarkka liukuluku	noin 15 desimaalin tarkkuus

Tässä monisteessa suositellaan aina käytettävän `double`-tietotyyppiä desimaalilukujen talletukseen, vaikka monessa paikassa `float`-tietotyyppiä käytetäänkin. Tämä johtuu siitä, että liukuluvut (joina

desimaaliluvut tietokoneessa käsitellään) ovat harvoin tarkkoja arvoja tietokoneessa. Itse asiassa ne ovat tarkkoja vain kun ne esittävät jotakin kahden potenssin kombinaatiota, kuten esimerkiksi 2.0, 7.0, 0.5 tai 0.375. Useimmiten liukuluvut ovat pelkkiä approksimaatioita oikeasta reaaliluvusta. Valitettavasti esimerkiksi lukua 0.1 ei pystytä tietokoneessa esittämään biteillä tarkasti. Tällöin laskujen määrän kasvaessa lukujen epätarkkuus vain lisääntyy. Tämän takia onkin turvallisempaa käyttää aina `double`-tietotyyppiä, koska se suuremman bittimääränsä takia pystyy tallentamaan enemmän merkitseviä desimaaleja. Reaalilukujen esityksestä tietokoneessa puhutaan lisää kohdassa 24.6 Liukuluku (floating-point) [VES][KOS]

7.1.2 Muuttujan nimeäminen

Muuttujan nimen täytyy olla siihen talletettavaa tietoa kuvaava. Yleensä pelkkä yksi kirjain on huono nimi muuttajalle, sillä se harvoin kuvaa kovin hyvin muuttujaa. Kuvaava muuttujan nimi selkeyttää koodia ja vähentää kommentoimisen tarvetta. Jotkut luulevat, että lyhyet muuttujien nimet ovat parempia, sillä se nopeuttaa koodin kirjoittamista. Nykyaikaisia kehitysympäristöjä käytettäessä tämä on kuitenkin virheluulo, sillä editorit osaavat ennustaa määriteltyjen muuttujien nimiä, joten niitä ei tarvitse kirjoittaa kokonaan kuin ensimmäisen kerran.

Yksikirjaimisia muuttujien nimiäkin voi perustellusti käyttää, jos niillä on esimerkiksi jo matematiikasta tai fysiikasta ennestään tuttu merkitys. Näin koordinaatteja kuvaamaan nimet `x` ja `y` ovat hyviä. Fysikaalisessa ohjelmassa `s` voi hyvin kuvata matkaa.

Kun muuttujien nimiä muutetaan jälkeenpäin paremmin merkitystään kuvaavaksi puhutaan, että koodia *refaktoroidaan*.

> refaktorointi = prosessi, jossa tietokoneohjelman lähdekoodia muutetaan siten, että sen toiminnallisuus säilyy, mutta sisäinen rakenne paranee. Refaktrointia on esimerkiksi muuttujien, aliohjelmien, metodien jne. uudelleennimeäminen paremmin merkitystään kuvaavaksi (**rename**) ja pitkien metodien (ja aliohjelmien) pilkkominen pienempiin osiin (**extract method**). Kattava listaus erilaisista refaktoroinneista löytyy seuraavasta linkistä: http://www.refactoring.com/catalog/index.html

Muuttujan nimi ei saa Javassa koskaan alkaa numerolla!

Javan ohjelmointistandardien mukaan muuttujan nimi alkaa pienellä kirjaimella ja jos muuttujan nimi koostuu useammasta sanasta aloitetaan uusi sana aina isolla kirjaimella kuten alla.

```
int polkupyoranRenkaanKoko;
```

Javassa muuttujan nimi voi periaatteessa sisältää myös ääkkösiä. Ääkkösten käyttämisestä voi kuitenkin koitua ongelmia, kun siirrytään *koodistosta* toiseen. Tässä monisteessa ääkkösten käyttämistä ei siten suositella.

> koodisto = Koodisto määrittelee jokaiselle *merkistön* merkille yksikäsitteisen koodinumeron. Merkin numeerinen esitys on usein välttämätön tietokoneissa. Merkistö määrittelee joukon merkkejä ja niille nimen, numeron ja jonkinnäköisen muodon kuvauksen. Merkistöllä ja koodistolla tarkoitetaan usein samaa asiaa, kuitenkin esimerkiksi Unicode-merkistö sisältää useita eri koodaus tapoja (UTF-8,UTF-16,UTF-32). Koodisto on siis se merkistön osa, joka määrittelee merkille numeerisen koodiarvon. Koodistoissa syntyy ongelmia yleensä silloin, kun siirrytään jostain skandimerkkejä (ä,ö,å..) sisältävästä koodistosta seitsemän bittiseen ASCII-koodistoon, joka ei tue skandeja. ASCII-koodistosta puhutaan lisää luvussa 25 ASCII-koodi

Muuttujan nimi ei saa myöskään olla mikään Javan varatuista sanoista eli sanoista joilla on Javassa joku muu merkitys.

7.1.3 Javan varatut sanat

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

7.2 Arvon asettaminen muuttujaan

Muuttujaan asetetaan arvo sijoitusoperaattorilla (**assignment operator**) "=". Lauseita joilla asetetaan muuttujille arvoja sanotaan sijoituslauseiksi. (**assignment statement**).

```
x = 20.0;
henkilonIka = 23;
paino = 80.5;
pituus = 183.5;
```

Muuttuja täytyy olla määritelty ennen kuin siihen voi asettaa arvoa ja muuttujaan voi asettaa vaan määrittelyssä annetun tietotyypin mukaisia arvoja tai sen kanssa sijoitusyhteensopivia arvoja. Esimerkiksi liukulukutyyppeihin (`float` ja `double`) voi sijoittaa myös kokonaislukutyyppisiä arvoja. Esimerkiksi alla oleva onnistuisi:

```
double liukuluku;
int kokonaisluku = 4;
liukuluku = kokonaisluku;
```

Sen sijaan toisinpäin tämä ei onnistu. Alla oleva koodi ei kääntyisi:

```
//TÄMÄ KOODI EI KÄÄNNY!
int kokonaisluku;
double liukuluku = 4.0;
kokonaisluku = liukuluku;
```

Muuttujaan voi asettaa arvon myös jo määrittelyn yhteydessä.

```
boolean onkoKalastaja = true;
char merkki = 't';
int kalojenLkm = 0;
double luku1 = 0, luku2 = 0, luku3 = 0;
```

Huomio, että `char`-tyyppiseen muuttujaan sijoitetaan arvo laittamalla merkki `'`-merkkien väliin. Näin sen erottaa myöhemmin käsiteltävästä `String`-tyyppiseen muuttujaan sijoittamista, jossa sijoitettava merkkijono laitetaan `"`-merkkien väliin. Sijoituslause voi sisältää myös monimutkaisiakin lausekkeita, esimerkiksi aritmeettisia operaatioita:

```
double numeroidenKeskiarvo = (2+4+1+5+3+2) / 6.0;
```

Sijoituslause voi sisältää myös muuttujia:

```
double huoneenPituus = 540.0;
double huoneenLeveys = huoneenPituus;
double huoneenAla = huoneenPituus*huoneenLeveys;
```

Eli sijoitettava voi olla mikä tahansa lauseke, joka tuottaa muuttujalle kelpaavan arvon.

Javassa täytyy aina asettaa joku arvo muuttujaan ennen sen käyttämistä. Kääntäjä ei käännä koodia, jossa käytetään muuttujaa jolle ei ole asetettu arvoa. Alla oleva ohjelma ei siis kääntyisi.

```
//TÄMÄ OHJELMA EI KÄÄNNY
public class Esimerkki
  public static void main(String[] args) {
    int ika;
    System.out.println(ika);
  }
}
```

7.3 Muuttujien näkyvyys

Muuttajaa voi käyttää (lukea ja asettaa arvoja) vain siinä lohkossa, missä se on määritelty. Muuttujan määrittelyn täytyy aina olla ennen (koodissa ylempänä), kun sitä ensimmäisen kerran käytetään. Jos muuttuja on käytettävissä sanotaan, että muuttuja *näkyy*. Aliohjelman sisällä määritelty muuttuja ei siis näy muissa aliohjelmissa.

Muuttuja voidaan määritellä myös niin, että se näkyy kaikkialla. Tällöin muuttujaa sanotaan *globaaliksi muuttujaksi* (**global variable**). Globaaleja muuttujia tulee välttää sillä ne lisäävät virheiden riskiä varsinkin laajemmissa ohjelmissa. Lisäksi niiden käyttäminen pystytään useimmiten välttämään pysähtymällä hetkeksi miettimään.

7.4 Vakiot

One man's constant is another man's variable. -Alan Perlis

Muuttujien lisäksi ohjelmointikielissä voidaan määritellä vakioita (**constant**). Vakioiden arvoa ei voi muuttaa määrittelyn jälkeen. Javassa vakio määritellään muuten kuten muuttuja, mutta muuttujan tyypin eteen kirjoitetaan lisämääre `final`.

```
final int KUUKAUSIEN_LKM = 12;
```

Javan ohjelmointistandardien mukaan vakioiden nimet kirjoitetaan isoilla kirjaimilla ja sanat erotetaan toisistaan alaviivoilla. Näin ne erottaa muuttujien nimistä.

Muuttujista poiketen vakioita voidaan turvallisesti määritellä niin, että ne näkyvät kaikkialla. Silloin niitä kutsutaan *globaaleiksi vakioiksi* (**global constant**). Tämä on turvallista, koska vakioiden arvoa ei pystytä muuttamaan määrittelyn jälkeen. Tällöin meille ei voi tulla sitä ongelmaa, että joku aliohjelmamme (tai metodimme) muuttaisi vahingossa tietoa, jota joku toinen aliohjelma (tai metodi) tarvitsisi.

7.5 Aritmeettiset lausekkeet

Usein meidän täytyy tallentaa muuttujiin erilaisten laskutoimitusten tuloksia. Javassa laskutoimituksia voidaan tehdä aritmeettisilla operaatioilla (**arithmetic operation)**, joista mainittiin jo kun teimme lumiukkoesimerkkiä. Ohjelmassa olevia aritmeettisia laskutoimituksia sanotaan aritmeettisiksi lausekkeiksi (**arithmetic expression**).

7.5.1 Javan aritmeettiset operaatiot

Javassa laskutoimituksia suoritetaan aritmeettisilla operaatiolla, joista + ja – tulivatkin esille aikaisemmissa esimerkeissä. Muita operaattoreita ovat:

Operaattori	Toiminto	Esimerkki
`+`	Summa	`System.out.println(1+2); // tulostaa 3`
`-`	Vähennys	`System.out.println(1-2); // tulostaa -1`
`*`	Kerto	`System.out.println(2*3); // tulostaa 6`
`/`	Jako	`System.out.println(6 / 2); // tulostaa 3`
`%`	Jakojäännös	`System.out.println(18 % 7); // tulostaa 4`

`++`	Lisäysoperaattori. Lisää muuttujan arvoa yhdellä.	`int luku = 0;`
		`System.println(luku++); //tulostaa 0`
		`System.println(luku++); //tulostaa 1`
		`System.println(luku); //tulostaa 2`
		`System.println(++luku); //tulostaa 3`

`--`	Vähennysoperaattori. Vähentää muuttujan arvoa yhdellä.	`int luku = 5;`
		`System.out.println(luku--); //tulostaa 5`
		`System.out.println(luku--);  //tulostaa 4`
		`System.out.println(luku);  //tulostaa 3`
		`System.out.println(--luku);  //tulostaa 2`
		`System.out.println(luku);  //tulostaa 2`

`+=`	Lisäysoperaatio.	`int luku = 0;`
		`luku += 2; //luku muuttujan arvo on 2`
		`luku += 3; //luku muuttujan arvo on 5`
		`luku += -1; //luku muuttujan arvo on 4`

`-=`	Vähennysoperaatio	`int luku = 0;`
		`luku -= 2; // luku muuttujan arvo on -2`
		`luku -= 1 // luku muuttujan arvo on -3`

`*=`	Kertolaskuoperaatio	`int luku = 1;`
		`luku *=  3; // luku-muuttujan arvo on 3`
		`luku *= 2; //luku-muuttujan arvo on 6`

`/=`	Jakolaskuoperaatio	`double luku = 27;`
		`luku /= 3; //luku-muuttujan arvo on 9`
		`luku /= 2.0; //luku-muuttujan arvo on 4.5`

`%=`	Jakojäännösoperaatio	`int luku = 9;`
		`luku %= 5; //luku-muuttujan arvo on 4`
		`luku %=2; //luku-muuttujan arvo on 0`

Lisäysoperaattoria (++) ja vähennysoperaattoria (--) voidaan käyttää, ennen tai jälkeen muuttujan. Käytettäessä ennen muuttujaa, arvoa muutetaan ensin ja mahdollinen toiminto esimerkiksi tulostus, tehdään vasta sen jälkeen. Jos operaattori sen sijaan on muuttujan perässä, toiminto tehdään ensiksi ja arvoa muutetaan vasta sen jälkeen.

7.5.2 Aritmeettisten operaatioiden suoritusjärjestys

Javassa aritmeettisten operaatioiden suoritus on vastaava kuin matematiikan laskujärjestys. Kerto- ja jakolaskut suoritetaan ennen yhteen- ja vähennyslaskua. Lisäksi sulkeiden sisällä olevat lausekkeet suoritetaan ensin.

```
System.out.println( 5+3*4-2 );  //tulostaa 15
System.out.println( (5+3) * (4-2) );  //tulostaa 16
```

7.5.3 Huomautuksia

Kun yritetään tallentaa kokonaislukujen jakolaskun tulosta liukulukutyyppiseen (`float` tai `double`) muuttujaan, voi tulos tallettua kokonaislukuna, jos jakaja ja jaettava ovat molemmat ilmoitettu ilman desimaaliosaa.

```
double laskunTulos = 5 / 2;
System.out.println(laskunTulos); //tulostaa 2.0
```

Jos kuitenkin vähintään yksi jakolaskun luvuista on desimaalimuodossa, niin laskun tulos tallentuu muuttujaan oikein.

```
double laskunTulos = 5 / 2.0;
System.out.println(laskunTulos); //tulostaa 2.5
```

Liukuluvuilla laskettaessa kannattaa pitää desimaalimuodossa myös luvut, joilla ei ole desimaaliosaa, eli ilmoittaa esimerkiksi luku 5 muodossa 5.0.

Kokonaisluvuilla laskettaessa kannattaa huomioida seuraava:

```
int laskunTulos = 5 / 4;
System.out.println(laskunTulos); //tulostaa 1

laskunTulos = 8 / 3;
System.out.println(laskunTulos); //tulostaa 2, vaikka 8 / 3 = 2.66
```

Kokonaisluvuilla laskettaessa lukuja ei siis pyöristetä lähimpään kokonaislukuun, vaan aina alaspäin. Desimaaliosa menee Javan jakolaskuissa ikään kuin "hukkaan". Jos lukuun halutaan matemaattinen pyöristys, voidaan käyttää esimerkiksi `Math`-kirjaston `round`-metodia.

```
    long laskunTulos = Math.round(8.0/3.0);
    System.out.println(laskunTulos); //tulostaa 3
```

7.6 Esimerkki: Painoindeksi

Tehdään ohjelma joka laskee painoindeksin. Painoindeksi lasketaan jakamalla paino (kg) pituuden (m) neliöllä, eli kaavalla pituus / paino^2. Javalla painoindeksi saadaan siis laskettua seuraavasti.

```
/**
 * Ohjelma joka laskee painoindeksin pituuden (m) ja
 * painon (kg) perusteella.
 * @author Martti Hyvönen
 * @version 19.8.2009
 */
public class Painoindeksi {

   /**
    * @param args ei käytössä
    */
   public static void main(String[] args) {
      double pituus = 1.83;
      double paino = 80.0;
      double painoIndeksi = paino / (pituus*pituus);
      System.out.println(painoIndeksi);
   }
}
```

Pituuden korotus toiseen potenssiin voitaisiin myös laskea käyttämällä `Math`-kirjaston `x`-metodia.

8. Oliotietotyypit

Javan alkeistietotyypit antavat melko rajoittuneet puitteet ohjelmointiin. Niillä pystytään tallentamaan ainoastaan numeroita ja yksittäisiä kirjaimia. Vähänkin monimutkaisimmissa ohjelmissa tarvitaan monimutkaisempia rakenteita tiedon tallennukseen. Javassa ja muissa oliokielissä tällaisen rakenteen tarjoavat oliot. Javassa jo merkkijonokin toteutetaan oliona.

8.1 Mitä oliot ovat?

Olio (**object**) on tietorakenne, jolla pyritään ohjelmoinnissa kuvaamaan mahdollisimman tarkasti reaalimaailman ilmiöitä. Luokkapohjaisissa kielissä (kuten Java ja C++ sekä C#) olion rakenteen ja käyttäytymisen määrittelee luokka, joka kuvaa olion attribuutit ja metodit. Attribuutit ovat olion ominaisuuksia ja metodit olion toimintoja. Olion sanotaan olevan luokan ilmentymä. Yhdestä luokasta voi siis luoda useita olioita, joilla on samat metodit ja attribuutit. Attribuutit voivat sen sijaan saada samasta luokasta luoduilla olioilla eri arvoja. Olioita voi joko tehdä itse tai käyttää jostain kirjastosta löytyviä valmiita olioita. Tarkastellaan olioiden käsitteitä esimerkin avulla.

Luokan ja olion suhdetta voisi kuvata seuraavalla esimerkillä. Kaikki luentosalissa olijat ovat ihmisiä. Heillä on tietyt samat ominaisuudet, jotka ovat kaikilla ihmisillä. Kuitenkin jokainen salissa olija on erilainen ihmisen ilmentymä, eli jokaisella oliolla on oma identiteetti. Eri ihmisillä voi olla erilainen tukka ja eriväriset silmät. Lisäksi ihmiset voivat olla eri pituisia, painoisia jne. Luentosalissa olevat identtiset kaksosetkin olisivat eri ilmentymiä ihmisestä. Jos Ihminen olisi luokka, niin kaikki luentosalissa olijat olisivat `Ihminen`-luokan ilmentymiä eli `Ihminen`-olioita. Tukka, silmät, pituus ja paino olisivat sitten olion ominaisuuksia eli attribuutteja. Ihmisellä voisi olla lisäksi joitain toimintoja eli metodeja kuten `syo()`, `meneToihin()`, `opiskele()` jne. Tarkastellaan seuraavaksi hieman todellisempaa esimerkkiä olioista.

Oletetaan, että suunnittelisimme yritykselle palkanmaksujärjestelmää. Siihen tarvittaisiin muun muassa `Tyontekija`-luokka. `Tyontekija`-luokalla täytyisi olla ainakin seuraavat attribuutit: nimi, tehtävä, osasto, palkka. Luokalla täytyisi olla myös ainakin seuraavat metodit: `maksaPalkka`, `muutaTehtävä`, `muutaOsasto`, `muutaPalkka`. Jokainen työntekijä olisi nyt omanlaisensa `Tyontekija`-luokan ilmentymä eli olio.

Omien olioluokkien tekeminen ei kuulu vielä Ohjelmointi 1-kurssin asioihin, mutta käyttäminen kyllä. Tarkastellaan seuraavaksi kuinka oliota käytetään.

8.2 Olion luominen

Olioviite määritellään kirjoittamalla ensiksi sen luokan nimi, josta olio luodaan. Seuraavaksi kirjoitetaan olion nimi. Nimen jälkeen tulee yhtäsuuruusmerkki, jonka jälkeen oliota luotaessa kirjoitetaan sana `new` ilmoittamaan, että luodaan uusi olio. Seuraavaksi kirjoitetaan luokan nimi uudelleen, jonka perään kirjoitetaan sulkuihin mahdolliset olion luontiin liittyvät parametrit. Parametrit riippuvat siitä kuinka luokan *konstruktori* (**constructor**) on toteutettu. Konstruktori (eli muodostaja) on metodi, joka suoritetaan aina kun uusi olio luodaan. Valmiita luokkia käyttääkseen ei tarvitse kuitenkaan tietää konstruktorin toteutuksesta, vaan tarvittavat parametrit selviävät aina luokan dokumentaatiosta. Yleisessä muodossa uusi olio luodaan alla olevalla tavalla.

```
Luokka olionNimi = new Luokka(parametri1, parametri2,..., parametriX);
```

Jos olio ei vaadi luomisen yhteydessä parametreja, kirjoitetaan silloin tyhjä sulkupari.

Uusi `Tyontekija`-olio voitaisiin luoda esimerkiksi seuraavasti. Parametrit riippuisivat nyt siitä kuinka olemme toteuttaneet `Tyontekija`-luokan *konstruktorin*. Tässä tapauksessa annamme nyt parametrina oliolle kaikki attribuutit.

```
Tyontekija akuAnkka = new Tyontekija("Aku Ankka", "Johtaja", "Osasto3" , 3000);
```

Monisteen alussa loimme lumiukkoja piirrettäessä `EasyWindow`-luokan olion seuraavasti.

```
EasyWindow window = new EasyWindow();
```

Javan ehkä yleisin olio on merkkijono eli `String`-luokan olio. Se voidaan luoda seuraavasti.

```
String nimi = new String("Hessu");
```

`String`-luokan olio voidaan poikkeuksellisesti luoda myös alkeistietotyyppisten muuttujien määrittelyä muistuttavalla tavalla. Alla oleva oleva lause on vastaava kuin edellisessä kohdassa, mutta lyhyempi kirjoittaa.

```
String nimi = "Hessu";
```

Itse asiassa oliomuuttuja on Javassa ainoastaan *viite* varsinaiseen olioon. Siksi niitä kutsutaankin usein myös *viitemuuttujiksi.* Viitemuuttajat eroavat oleellisesti alkeistietotyyppisistä muuttujista.

8.3 Oliotietotyyppien ja alkeistietotyyppien ero

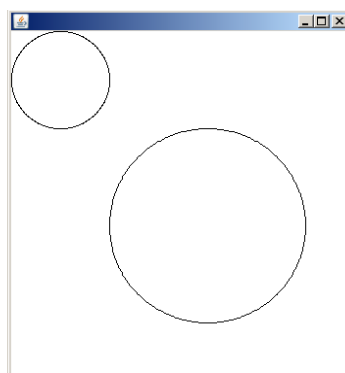
Viitemuuttujat eroavat alkeistietotyypeistä siinä, että ne ovat *viitteitä* tiettyyn olioon. Samaan olioon voi viitata useampi muuttuja. Vertaa alla olevia koodinpätkiä.

```
int luku1 = 10;
int luku2 = luku1;
luku1 = 0;
System.out.println(luku2); //tulostaa 10
```

Yllä oleva tulostaa ”10” niin kuin pitääkin. Muuttujan `luku2` arvo ei siis muutu, vaikka asetamme kolmannella rivillä muuttujaan `luku1` arvon 0. Tämä johtuu siitä, että toisella rivillä asetamme muuttujaan `luku2` muuttujan `luku1` arvon, emmekä viitettä muuttujaan `luku1`. Oliotietotyyppisten muuttujien kanssa asia on toinen. Vertaa yllä olevaa esimerkkiä seuraavaan:

```
EasyWindow ikkuna1 = new EasyWindow();
ikkuna1.addCircle(50, 50, 50);


EasyWindow ikkuna2 = ikkuna1;
ikkuna2.addCircle(200, 200, 100);

ikkuna1.showWindow();
```

Yllä oleva koodi piirtää seuraavan kuvan:

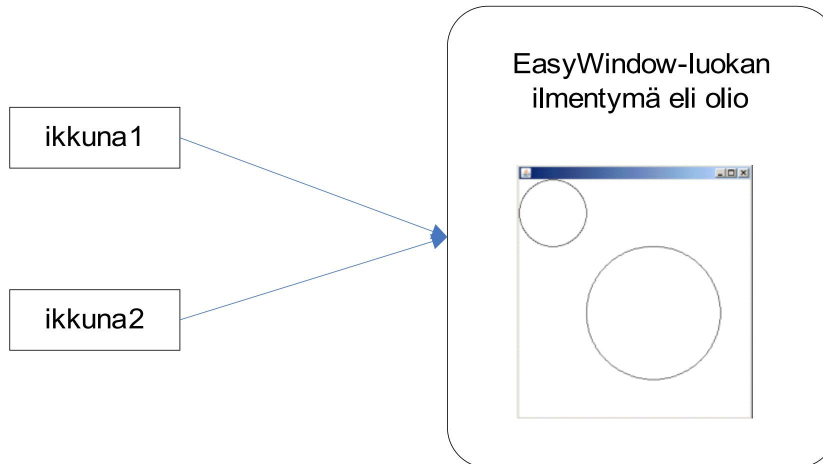


Kuva 8: Molemmat ympyrät piirrettiin samaan ikkunaan, koska molemmat muuttujat viittaavat samaan ikkunaan.

Nopeasti voisi olettaa, että `ikkuna1`-oliossa näkyisi nyt vain yksi ympyrä. Näin ei kuitenkaan ole, vaan molemmat ympyrät piirretään samaan ikkunaan kuten kuvassa. Tämä johtuu siitä, että muuttujat `ikkuna1` ja `ikkuna2` ovat olioviitteitä, jotka viittaavat (ts. osoittavat) samaan olioon.

```
EasyWindow ikkuna2 = ikkuna1;
```

Toisin sanoen yllä olevalla rivillä ei luoda uutta `EasyWindow`-oliota, vaan ainoastaan uusi olioviite, joka viittaa nyt samaan olioon kuin `ikkuna1`.



Kuva 9: Sekä ikkuna1 että ikkuna2 viittaavat samaan olioon.

oliomuuttuja = viite todelliseen olioon. Samaan olioon voi olla useitakin viitteitä.

Viitteitä käsitellään tarkemmin luvussa 13.8 Esimerkki: Olioiden ja alkeistietotyyppien erot.

8.4 Metodin kutsuminen

Jokaisella tietystä luokasta luodulla oliolla on käytössä kaikki tämän luokan julkiset metodit. Metodikutsussa käsketään oliota tekemään jotain. Voisimme esimerkiksi sanoa `window`-oliolle, että piirrä sisääsi ympyrä tai käskeä `Tyontekija`-oliota muuttamaan palkkaansa.

Olion metodeita kutsutaan kirjoittamalla ensiksi olion nimi, piste ja kutsuttavan metodin nimi. Metodin mahdolliset parametrit laitetaan sulkeiden sisään ja erotetaan toisistaan pilkulla. Jos metodi ei vaadi parametreja, täytyy sulut silti kirjoittaa, niiden sisälle ei vaan tule mitään. Yleisessä muodossa metodikutsu on seuraava:

```
olionNimi.metodinNimi(parametri1,parametri2,...parametriX);
```

Voisimme nyt esimerkiksi korottaa `akuAnkka`-olion palkkaa alla olevalla tavalla.

```
akuAnkka.muutaPalkka(3500);
```

Tai piirtää `window`-olioon ympyrän kuten lumiukkoesimerkissä.

```
window.addCircle(20,80,20);
```

`String`-luokasta löytyy esimerkiksi `charAt`-metodi, joka palauttaa yhden kirjaimen haluamastamme kohdasta merkkijonoa. Parametrina `charAt`-metodi tarvitsee indeksin, eli tiedon siitä monennenko merkin haluamme saada tietää. Indeksointi alkaa nollasta. Ensimmäisen kirjaimen merkkijonosta saisi esimerkiksi tietää seuraavasti.

```
String nimi = "Aku Ankka";
char ensimmainenKirjain = nimi.charAt(0);
```

8.5 Olion tuhoamisen hoitaa roskienkeruu

Kun olioon ei enää viittaa yhtään muuttujaa, täytyy olion käyttämät muistipaikat vapauttaa muuhun käyttöön. Tästä huolehtii Javan automaattinen roskienkeruu (**garbage collection**). Kun olioon ei ole enää viitteitä, se merkataan ja aina tietyin väliajoin roskienkerääjä (**garbage collector**) vapauttaa merkattujen olioiden muistipaikat.

Kaikissa ohjelmointikielissä näin ei ole (esim. C++), vaan muistin vapauttamisesta tulee huolehtia itse. Näissä kielissä tämä tehdään metodilla, jota sanotaan destruktoriksi (**destructor** = hajottaja). Destruktori suoritetaan aina kun olio tuhotaan. Vertaa konstruktoriin, joka suoritettiin kun olio luodaan. Javassa siis ei ole destruktoria.

8.6 Olioluokkien dokumentaatio

Luokan dokumentaatio sisältää tiedot luokasta, luokan konstruktoreista ja metodeista. Yleensä dokumentaatiossa on myös muutama esimerkki. Tutustutaan nyt tarkemmin `String`-luokan dokumentaatioon. `String`-luokan dokumentaatio löytyy sivulta:

http://java.sun.com/javase/6/docs/api/java/lang/String.html

Alussa on yleistä tietoa luokasta ja muutama esimerkki. Oleellisimpia ovat kuitenkin kohdat Constructor Summary ja Method Summary.

8.6.1 Constructor Summary

Tämä kohta sisältää tiedot kaikista luokan konstruktoreista. Konstruktoreita voi olla useita, kunhan niiden parametrit eroavat toisistaan. Jokaisella konstruktorilla on taulukossa oma rivi. Rivillä on kerrottu minkä tyyppisiä ja montako parametria konstrukori ottaa vastaan ja lyhyesti mitä se tekee. Kaikista konstruktoreista on lisätietoa samalla sivulla alempana kohdassa Constructor Detail. Rivillä ensimmäinen `String`-sana toimii hyperlinkkinä joka vie siihen kohtaa missä tästä konstruktorista kerrotaan enemmän.

Tässä vaiheessa voi olla vielä hankalaa ymmärtää kaikkien konstruktorien merkitystä, sillä ne sisältävät tietotyyppejä joita emme ole vielä käsitelleet. Esimerkiksi tietotyypin perässä olevat hakasulkeet (esim. `int[]`) tarkoittavat että kyseessä on *taulukko*. Taulukoita käsitellään myöhemmin luvussa 14 Taulukot.

Eräs `String`-luokan konstruktori on yleisessä muodossa seuraava:

```
String(String original)
```

Se saa siis parametrikseen merkkijonon. Sitä voitaisiin käyttää esimerkiksi seuraavasti:

```
String elokuva = new String(”Casablanca”);
```

Vastaavasti merkkijono voitaisiin kuitenkin alustaa myös muilla `String`-luokan konstruktoreilla, joita on aika lista.

Jos taas tutkimme `EasyWindow`-luokan dokumentaatiota (löytyvät osoitteesta: http://users.jyu.fi/~vesal/kurssit/ohj1/graphics/), löydämme siitä kaksi eri konstruktoria. Toinen ei saa mitään parametreja, jota olemmekin jo käyttäneet. Se luo oletuksena tietyn oletuskokoisen ikkunan. Oletuskoko ei selviä dokumentaatiosta. Voisimme kuitenkin määrittää ikkunan koon

myös itse, käyttämällä alempaa konstruktoria. Ensimmäinen parametri ilmoittaa ikkunan leveyden ja toinen korkeuden.

Constructor Summary
`EasyWindow()` Luo uuden ikkunan
`EasyWindow(int width, int height)` Luo uuden ikkunan

Kuva 10: Tiedot luokan konstruktoreista löytää Constructor Summary -kohdasta.

Luodaan malliksi esimerkiksi 100x100 kokoinen ikkuna.

```
EasyWindow pikkuIkkuna = new EasyWindow(100,100);
pikkuIkkuna.showWindow();
```

8.6.2 Method Summary

Tämä kohta sisältää tiedot kaikista luokan metodeista. Jokaisella metodilla on taulukossa oma rivi. Ensimmäinen sarake kertoo minkä tyyppisen arvon metodi palauttaa. Esimerkiksi `String`-luokassa on ensimmäisellä rivillä käyttämämme `charAt`-metodi, joka siis palauttaa `char`-tyyppisen arvon.

Toisesta sarakkeesta selviää metodin nimi. Mitä parametreja metodi tarvitsee, sekä lyhyt kuvaus mitä metodi tekee. Kuten konstruktoreista niin myös metodeista löytyy lisätietoa samalta sivulta kohdasta Method Detail. Jokaisen metodin nimi toimii hyperlinkkinä, joka vie siihen kohtaa missä metodista kerrotaan lisää.

8.6.3 Huomautus: Luokkien dokumentaation googlettaminen

Luokan uusimman dokumentaation löytää Googlella hakusanalla: "Java 6 LuokanNimi". Numerolla 6 viitataan Javan uusimpaan versioon. Pelkällä "Java LuokanNimi" hakusanalla tuntuu löytyvän aina vanhaa dokumentaatiota. Esimerkiksi `String`-luokan dokumentaatio löytyy hakusanalla: "Java 6 String".

8.7 Tyyppimuunnokset

Javassa ei tietyn tyyppiseen muuttujaan voi tallentaa kuin yhtä tyyppiä. Tämän takia meidän täytyy joskus muuttaa esimerkiksi `String`-tyyppinen muuttuja `int`-tyyppiseksi tai `double`-tyyppinen muuttaja `int`-tyyppiseksi ja niin edelleen. Kun muuttujan tyyppi vaihdetaan toiseksi, kutsutaan sitä tyyppimuunnokseksi.

Kaikilla olioilla tulisi olla `toString`-metodi, jolla olio voidaan muuttaa merkkijonoksi. Merkkijonon muuttaminen alkeistietotyypiksi onnistuu sen sijaan jokaiselle alkeistietotyypille tehdystä luokasta löytyvällä metodilla. Alkeistietotyypithän eivät ole olioita, joten niillä ei ole metodeita. Jokaista alkeistietotyyppiä varten on kuitenkin tehty Javaan niin sanotut *kääreluokat*, joista löytyy alkeistietotyyppien käsittelyyn hyödyllisiä metodeita. Alkeistietotyyppejä vastaavat kääreluokat löytyy seuraavasta taulukosta:

Alkeistietotyyppi	**Kääreluokka**
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Merkkijonon (String) muuttaminen int-tyypiksi onnistuu Javan Integer-luokan parseInt-funktiolla seuraavasti:

```
String luku1 = "24";
int luku2 = Integer.parseInt(luku1);
```

Tarkasti sanottuna parseInt-funktio luo parametrina saamansa merkkijonon perusteella uuden int-tyyppisen tiedon, joka talletetaan muuttujaan luku2.

Jos luvun parsiminen (jäsentäminen, muuttaminen) ei onnistu saadaan "java.lang.NumberFormatException"-poikkeus. Doublen parsiminen onnistuu vastaavasti, mutta Double-luokasta löytyvällä parseDouble-funktiolla.

```
String luku3 = "2.45";
double luku4 = Double.parseDouble(luku3);
```

Alkeistietotyypin muuttaminen merkkijonoksi onnistuu taas luokista löytyvällä toString-funktiolla seuraavasti:

```
// int merkkijonoksi
String intMerkkijonona = Integer.toString(24);

// double merkkijonoksi
String doubleMerkkijonona = Double.toString(2.45);
```

Kääreluokkien avulla voidaan alkeistietotyyppejä *kääriä* (**wrap**) olioksi. Tällöin alkeistietotyyppejä voidaan käsitellä kuten olioita. Käärimisestä on hyötyä esimerkiksi silloin kun haluamme tallentaa alkeistietotyyppejä dynaamiseen (muuttuvaan) taulukkoon. Olioiksi käärimistä ei kuitenkaan kannata tehdä turhaan, koska olion vie aina enemmän muistia kuin alkeistietotyyppi. Alkeistietotyyppien käärimisestä puhutaan lisää kohdissa 20.2 ArrayList ja 20.2.3 Lukujen tallentaminen tietorakenteeseen, autoboxing.

9. Aliohjelman paluuarvo

Aliohjelmat-luvussa tekemämme `lumiukko`-aliohjelma ei palauttanut mitään arvoa. Usein on kuitenkin hyödyllistä, että lopettaessaan aliohjelma palauttaa jotain tietoa ohjelman suorituksesta. Mitä hyötyä olisi esimerkiksi aliohjelmasta, joka laskee kahden luvun keskiarvon, jos emme koskaan saisi tietää mikä niiden lukujen keskiarvo on? Voisimmehan me tietenkin tulostaa luvun keskiarvon suoraan aliohjelmassa, mutta usein on järkevämpää palauttaa tulos paluuarvona. Tällöin aliohjelmaa voidaan käyttää myös tilanteessa, jossa keskiarvoa ei haluta tulostaa, vaan sitä tarvitaan johonkin muuhun laskentaan. Paluuarvon palauttaminen tapahtuu `return`-lauseella. Aliohjelman suoritus loppuu aina `return`-lauseeseen.

Toteutetaan nyt aliohjelma, joka laskee kahden kokonaisluvun keskiarvon ja palauttaa tuloksen paluuarvona.

```
public static double keskiarvo(int a, int b) {
   double ka;
   ka = (a+b)/2.0; // Huom 2.0 auttaa, että tulos on reaaliluku
   return ka;
}
```

Ensimmäisellä rivillä määritellään jälleen julkinen ja staattinen aliohjelma. Lumiukko-esimerkissä `static`-sanan jälkeen luki `void`, joka tarkoitti, että aliohjelma ei palauttanut mitään arvoa. Koska nyt haluamme, että aliohjelma palauttaa parametrina saamiensa kokonaislukujen keskiarvon, niin meidän täytyy kirjoittaa paluuarvon tyyppi `void`-sanan tilalle `static`-sanan jälkeen. Koska kahden kokonaisluvun keskiarvo voi olla myös desimaaliluku, niin paluuarvon tyyppi on `double`. Sulkujen sisällä ilmoitetaan jälleen parametrit. Nyt parametreina on kaksi kokonaislukua `a` ja `b`. Toisella rivillä määritellään reaalilukumuuttuja `ka`. Kolmannella rivillä lasketaan parametrien `a` ja `b` summa ja jaetaan se kahdella muuttujaan `ka`. Neljännellä rivillä palautetaan `ka`-muuttujan arvo.

Aliohjelmaa voitaisiin nyt käyttää pääohjelmassa esimerkiksi alla olevalla tavalla.

```
double karvo;
karvo = keskiarvo(3,4);
System.out.println("Keskiarvo = " + karvo);
```

Tai lyhyemmin kirjoitettuna:

```
System.out.println("Keskiarvo = " + keskiarvo(3,4));
```

Koska `keskiarvo`-aliohjelma palauttaa aina `double`-tyyppisen liukuluvun, voidaan kutsua käyttää kuten mitä tahansa `double`-tyyppistä arvoa. Se voidaan esimerkiksi tulostaa tai tallentaa muuttujaan.

Itse asiassa koko `keskiarvo`-aliohjelman voisi kirjoittaa lyhyemmin muodossa:

```
public static double keskiarvo(int a, int b) {
    double ka = (a+b)/2.0;
    return ka;
}
```

Yksinkertaisimmillaan `keskiarvo`-aliohjelman voisi kirjoittaa jopa alla olevalla tavalla.

```
public static double keskiarvo(int a, int b) {
    return (a+b)/2.0;
}
```

Kaikki yllä olevat tavat ovat oikein, eikä voi sanoa mikä tapa on paras. Joskus "välivaiheiden" kirjoittaminen selkeyttää koodia, mutta `keskiarvo`-aliohjelman tapauksessa mielestäni viimeisin tapa on selkein ja lyhin.

Aliohjelmassa voi olla myös useita `return`-lauseita. Tästä esimerkki kohdassa: 13.4.1 Esimerkki: Pariton vai parillinen.

Aliohjelma ei kuitenkaan voi palauttaa kerralla suoranaisesti useita arvoja. Toki voidaan palauttaa esimerkiksi taulukko, jossa sitten on monia arvoja. Toinen keino olisi tehdä olio, joka sisältäisi useita arvoja ja palautettaisiin.

Metodeita ja aliohjelmia, joilla on parametri tai parametreja ja paluuarvo sanotaan joskus myös funktioiksi. Nimitys ei ole hullumpi, jos vertaa `keskiarvo`-aliohjelmaa vaikkapa matematiikan funktioon f(x,y): (x + y)/2. Funktiolla ei lisäksi saisi olla sivuvaikutuksia, kuten esimerkiksi tulostamista tai globaalien muuttujien muuttamista.

Mitä eroa on tämän:

```
double tulos = keskiarvo(5,2); // funktio keskiarvo laskisi kahden luvun keskiarvon
System.out.println(tulos); //tulostaa 3.5
System.out.println(tulos); //tulostaa 3.5
```

ja tämän:

```
System.out.println( keskiarvo(5,2) ); //tämäkin tulostaa 3.5
System.out.println( keskiarvo(5,2) ); //tämäkin tulostaa 3.5
```

koodin suorituksessa?

Ensimmäisessä lukujen 5 ja 2 keskiarvo lasketaan vain kertaalleen, jonka jälkeen tulos tallennetaan muuttujaan. Tulostuksessa käytetään sitten tallessa olevaa laskun tulosta.

Jälkimmäisessä versiossa lukujen 5 ja 2 keskiarvo lasketaan tulostuksen yhteydessä. Keskiarvo lasketaan siis kahteen kertaan. Vaikka alemmassa tavassa säästetään yksi koodirivi, kulutetaan siinä turhaan tietokoneen resursseja laskemalla sama lasku kahteen kertaan. Tässä tapauksessa tällä ei ole juurikaan merkitystä, sillä `keskarvo`-aliohjelman suoritus ei juurikaan rasita nykyaikaisia tietokoneita. Kannattaa kuitenkin opetella tapa, ettei ohjelmassa tehtäisi mitään turhia suorituksia.

Muuttujat-luvun lopussa tehtiin ohjelma, joka laski painoindeksin. Tee ohjelmasta uusi versio, jossa painoindeksin laskeminen tehdään aliohjelmassa. Aliohjelma saa siis parametreina pituuden ja painon ja palauttaa painoindeksin.

10. Eclipse

Vaikka ohjelmointi on kivaa pelkällä editorillakin, ohjelmien koon kasvaessa alkaa kaipaamaan työvälineiltä hieman enemmän ominaisuuksia. Peruseditoreja enemmän ominaisuuksia tarjoavat sovelluskehittimet eli IDE:t (**Integrated Development Environment**). Javalle tehtyjä ilmaisia sovelluskehittimiä ovat muun muassa NetBeans ja Eclipse. Tässä monisteessa tutustumme tarkemmin Eclipseen.

Kaikki ohjeet on testattu toimiviksi Eclipse Classic 3.5.0 versiolla Windows-ympäristössä. Jotkut näppäinkomennot voivat vaihdella eri käyttöjärjestelmien versiolla.

10.1 Asennus

Monisteen kirjoitushetkellä Eclipsen uusin versio on 3.6, jolle on annettu myös nimi Galileo. Eclipse on saatavilla Windows, Linux ja Macintosh käyttöjärjestelmiin ja sen voi ladata osoitteesta http://www.eclipse.org/downloads/ . Eclipsestä on eri tarkoituksiin eri versioita. Javan koodaaminen onnistuu ainakin seuraavilla versioilla:

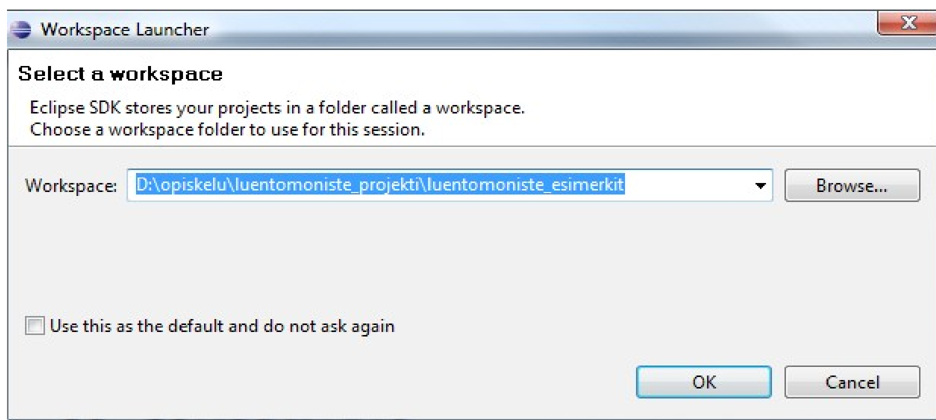
- Eclipse IDE for Java EE Developers (206 MB)
- Eclipse Classic 3.6.0 (170 MB)
- Eclipse IDE for Java Developers (99 MB)

Windowsiin Eclipse asennetaan lataamalla pakattu tiedosto, joka puretaan haluttuun paikkaan.

10.2 Käyttö

10.2.1 Ensimmäinen käyttökerta

Eclipse kysyy aina käynnistettäessä *workspacen* hakemistoa. Workspace eli työtila on Eclipsen hakemistorakenteen "ylin solmu". Workspaceen voi sitten luoda useita projekteja ja projekteihin edelleen luokkia



Kuva 11: Eclipsen workspacen eli työtilan valinta.

Kun workspace on valittu, tulee seuraavaksi ensimmäisellä käyttökerralla näytettävä tervetuloa-sivu, josta pääsee klikkaamaan itsensä *tutoriaaleihin* ym. Varsinaiseen työtilaan pääsee joko sulkemalla tervetuloa-sivun ruksilla, tai klikkaamalla oikealla olevaa nuolta.

tutoriaali = opas

Nyt edessä pitäisi olla Eclipsen perusnäkymä. Eclipse voi vaikuttaa aluksi melko monimutkaiselta,

mutta sen peruskäytön oppii todella nopeasti. Eclipsen tiedostorakenteessa jokaisen tiedoston on kuuluttava johonkin projektiin. Luodaan nyt ensimmäinen projekti. Huomaa, että monet asiat Eclipsessä voi tehdä ainakin kahdesta eri paikasta. Jotta monisteen paksuus ei kasva järjettömyyksiin, niin kerrotaan tässä aina vain yksi tapa ja mahdollisesti lisäksi näppäinkomento, jos sellainen on. Projektin luominen onnistuu valitsemalla ylävalikosta File → New → Java Project. Asetukset pitäisi olla oletuksena niin kuin pitää, joten riittää, kun antaa uudelle projektille nimen ja klikkaa alhaalta Finish. Nyt Project Exploreriin pitäisi ilmestyä nimeämäsi projekti.

10.2.2 Ohjelman kirjoittaminen

Koska jokainen Java-ohjelma kirjoitetaan luokan sisään, niin ohjelman tekeminen aloitetaan Eclipsessä luomalla luokka. Valitaan ylävalikosta: File → New → Class. Nyt edessä pitäisi olla ikkuna uuden luokan luomiseksi. Luokalle pitää antaa ainakin nimi. Lisäksi kannattaa usein klikata kohtaan ”Which method stubs would like to create?” ruksi kohtaan ”public static void main(String args[])”, jolloin Eclipse luo luokkaan automaattisesti pääohjelman.

Nyt meillä on edessä uusi luokan raakile ja editori, jolla ohjelma voidaan kirjoittaa.

10.2.3 Ohjelman kääntäminen ja ajaminen

Kun ohjelma on kirjoitettu, sen ajaminen onnistuu ylhäältä Run-napista (vihreä ympyrä jossa valkoinen kolmio). Nappia painamalla Eclipse kääntää ohjelman automaattisesti ja suorittaa heti sen jälkeen. Nopeinta ohjelman suorittaminen on kuitenkin painamalla **Ctrl-F11**. Jos haluamme lopettaa ohjelman suorituksen jostain syystä kesken, onnistuu se konsolisovelluksissa alhaalta konsolivälilehden päällä sijaitsevaa Terminate-nappia (punainen neliö) painamalla.

10.2.4 Debuggaus

Varsinkin monimutkaisemmista ohjelmista loogisen virheen löytäminen on välillä vaikeaa. Tähän on apuna Eclipsen debuggaus-toiminto. Siinä ohjelman suoritusta voi seurata rivi riviltä, samoin kuin muuttujien arvojen muuttumista. Tämä auttaa huomattavasti virheen tai epätoivotun toiminnan syyn selvittämisessä. Vanha tapa tehdä samaa asiaa on lisätä ohjelmaan tulostuslauseita, mutta sitten nämä ohjelman muutokset jäävät helposti ohjelmaan ja saattavat toisinaan myös muuttaa ohjelman toimintaa.

Termi "debug" johtaa yhden legendan mukaan aikaan, jolloin tietokoneohjelmissa ongelmia aiheuttivat releiden väliin lämmittelemään päässeet luteet. Ohjelmien korjaaminen oli siis kirjaimellisesti hyönteisten (**bugs**) poistoa. Katso lisätietoja vaikka Wikipediasta:

http://en.wikipedia.org/wiki/Software_bug#Etymology.

Eclipsessä debuggaus aloitetaan asettamalla ensin johonkin kohtaan koodia keskeytyskohta (**breakpoint).** Keskeytyskohta on kohta, johon haluamme testauksen aikana ohjelman suorituksen väliaikaisesti pysähtyvän. Ohjelman pysähdyttyä voidaan sitä sitten alkaa suorittamaan rivi riviltä. Keskeytyskohta tulee siis asettaa koodiin ennen oletettua virhekohtaa. Jos haluamme debugata koko ohjelman, asetamme vain keskeytyskohdan ohjelman alkuun.

Kun keskeytyskohta on asetettu, klikataan ylhäältä Debug-nappia (vihreä hyönteinen) tai painetaan vain **F11**. Nyt Eclipse kysyy halutaanko avata Debug-näkymä, vastataan kyllä.

Ohjelman suoritus on nyt pysähtynyt siihen kohtaan mihin asetimme keskeytyskohdan. Debuggausnäkymässä oikealla näkyy kaikki tällä hetkellä näkyvillä olevat muuttujat ja niiden arvot. Keskellä näkyy ohjelman koodi, jossa on vihreällä se rivi missä kohtaa ohjelmaa ollaan suorittamassa. Alhaalla näkyy konsoli, johon siis tulee kaikki konsoliin tehdyt tulostukset.

Ohjelman suoritukseen rivi riviltä on nyt kaksi eri komentoa: Step Into (**F5**) ja Step Over (**F6**). Napit toimivat muuten samalla tavalla, mutta jos kyseessä on aliohjelmakutsu, niin Step Into -komennolla mennään aliohjelman sisälle, kun Step Over -komento suorittaa rivin kuin se olisi yksi lause. Kaikki tällä hetkellä näkyvyysalueella olevat muuttujat ja niiden arvot nähdään oikealla olevalla Variables-välilehdellä.

Kun emme enää halua suorittaa ohjelmaa rivi riviltä, voimme joko suorittaa ohjelman loppuun Resume (**F8**)-napilla tai keskeyttää ohjelman suorituksen Terminate (**Ctr+F2**)-napilla.

10.2.5 Paketit

Eclipse-projekteihin voi luoda myös paketteja (**package**). Paketit voidaan ymmärtää eräänlaisiksi kansioiksi, eli niihin voi siis laittaa kooditiedostoja. Paketit ovat välttämättömiä silloin kun haluamme käyttää toisesta itse tekemästämme tiedostosta löytyviä aliohjelmia ja olioita. Tällöin tiedostoon, jonka olioita ja aliohjelmia haluamme käyttää, voidaan viitata paketin nimen avulla.

Jos kooditiedostomme on jonkun paketin sisällä, täytyy se ilmoittaa kooditiedostossa pakettiesittelyllä (**package declaration**). Tämä tehdään ennen kaikkia muita lauseita ja esittelyitä. Esimerkiksi jos tiedosto HelloWorld.java sijaitsee paketin esimerkit sisällä, kirjoitetaan HelloWorld.java-tiedoston alkuun lause:

```
package esimerkit;
```

Eclipse osaa pitää pakettiesittelyt kunnossa automaattisesti. Siirrettäessä tiedosto paketista toiseen, osaa Eclipse muuttaa pakettiesittelyn koodin alussa.

10.2.6 Jar-tiedostojen käyttäminen

Oman tai jonkun muun tekemän .jar- tiedoston (esimerkiksi kurssilla käytettävä Graphics.jar) voi lisätä Eclipseen seuraavasti. Valitse ylävalikosta Project → Properties → Java Build Path. Tämän jälkeen paina oikealta Add External Jars ja hae tiedostoistasi tallentamasi .jar-päätteinen tiedosto.

10.3 Hyödyllisiä ominaisuuksia

10.3.1 Syntaksivirheiden etsintä

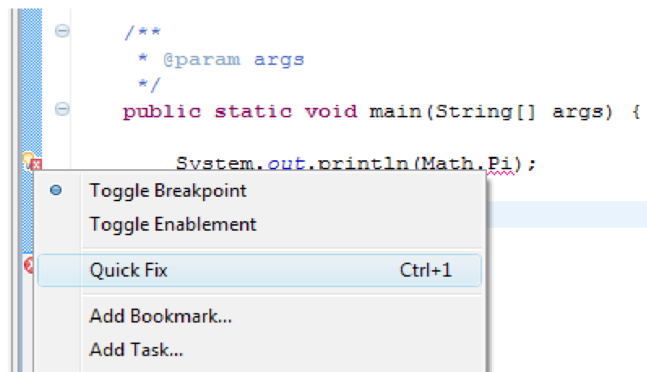
Eclipse huomaa osan syntaksivirheistä, joten osa virheistä voidaan korjata jo ennen kääntämistä. Kun Eclipse löytää virheen, ilmestyy virheellisen rivin kohdalle vasemmalle marginaaleihin punainen rasti. Lisäksi jos virhe paikallistuu alleviivaa Eclipse virheellisen koodinpätkän punaisella aaltoviivalla. Viemällä hiiri punaisen rastin päälle, Eclipse kertoo tarkemmin mikä kyseisessä kohdassa on vikana. Huomaa, että Eclipse ei välttämättä paikallista virhettä täysin oikein. Usein virhe voi olla myös edellisellä tai seuraavalla rivillä. Virheestä ilmoittaa myös marginaaleissa oleva lamppu, jonka päällä on punainen rasti. Tällöin voidaan käyttää QuickFix-toimintoa.

10.3.2 Quick Fix

Eclipse osaa itse korjata osan virheistä Quick Fix-toiminnolla. Kun Quick Fix-toimintoa voi käyttää, ilmoittaa Eclipse virheestä marginaalissa lampulla, jonka päällä on punainen rasti. Nyt hiiren oikeaa nappia painamalla löytyy Quick Fix-toiminto. Toimintoa käyttämällä ohjelma ehdottaa ratkaisuja ohjelman korjaamiseksi ja valinnan jälkeen myös tekee korjaukset automaattisesti. Nopeampi tapa käyttää Quick Fix:ä on viedä kursori riville, jolla virhe on, ja painaa **Ctrl+1**.

Jos kirjoitamme aliohjelma-kutsun ennen aliohjelman määrittelyä, osaa Quick Fix luoda luoda

meille rungon aliohjelman määrittelyä varten.



Kuva 12: Quick Fix-toiminto auttaa huomattavasti korjaamaan virheitä koodissa.

10.3.3 Kooditäydennys (content assist)

Jos ei ole konekirjoituksen Mika Häkkinen, voi koodia kirjoittaessa Eclipsessä huijata painamalla **Ctrl+Space**. Tällöin Eclipse koittaa arvata mitä haluat kirjoittaa. Jos mahdollisia vaihtoehtoja on monta, näyttää Eclipse vaihtoehdot listana.

10.3.4 Koodimallit (Templates)

Eclipsessä on olemassa valmiita koodimalleja, jotka saa nopeasti ilmestymään Eclipsen editoriin kirjoittamalla koodimallin nimi ja painamalla **Ctrl+Space**. Esimerkiksi kirjoittamalla Eclipsen editoriin "syso" ja painamalla **Ctrl+Space** ilmestyy editoriin automaattisesti teksti `System.out.println("").`

Pääohjelman saa vastaavasti editoriin kirjoittamalla ”main” ja painamalla **Ctrl+Space**.

Koodimalleja voi tehdä myös itse kohdasta Window → Preferences → Java → Editor → Templates ja klikkaamalla New.

Kurssin Wikistä löytyy lisää vinkkejä ja neuvoja Eclipsen käyttöön:

https://trac.cc.jyu.fi/projects/ohj1/wiki/Ohj1Eclipse

11. ComTest

“Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.” – Edsger W. Dijkstra

Javassa on olemassa JUnit-niminen yksikkötestausrajapinta (**unit testing framework**). Tämä mahdollistaa ns. yksikkötestien kirjoittamisen. Jo melko yksinkertaisten ohjelmien testaaminen konsoliin tulostelemalla veisi paljon aikaa. Tulostukset pitäisi tehdä uudestaan jokaisen muutoksen jälkeen, sillä emme voisi mitenkään tietää, että ennen muutosta tekemämme testit toimisivat vielä muutoksen jälkeen. Yksikkötestauksen idea on, että jokaiselle aliohjelmalla ja metodille kirjoitetaan oma testinsä erilliseen tiedostoon, jotka voidaan sitten kaikki ajaa kerralla. Näin voimme suorittaa kerran kirjoitetut testit jokaisen pienenkin muutoksen jälkeen todella helposti.

Ongelmana on, että JUnit-testien kirjoittaminen on melko työlästä. Tähän on apuna Jyväskylän Yliopiston ComTest-projekti. ComTest:n ideana on, että testit voidaan kirjoittaa yksinkertaisella syntaksilla aliohjelmien ja metodien dokumentaatiokommentteihin, joista sitten luodaan varsinaiset JUnit-testit. Samalla kirjoitetut testit toimivat dokumentaatiossa esimerkkinä aliohjelman tai metodin toiminnasta. ComTest:n asennusohjeet löytyy sivulta:

https://trac.cc.jyu.fi/projects/comtest#a5.Asennusjak%C3%A4ytt%C3%B6

Koska ComTest on vielä kehitysvaiheessa, löytyy sivuilta myös ajankohtaisimmat tiedot ComTest:n käytöstä.

11.1 ComTest:n käyttö

Ensinnäkin kaikkien tiedostojen, joille halutaan luoda JUnit-testi ComTest:ä käyttämällä, täytyy olla jonkun paketin sisällä. Paketin täytyy lisäksi olla joku muu kuin oletuspaketti (**default package**), jonka sisällä tiedostot Eclipsessä ovat, elleivät ole minkään muun paketin sisällä.

Kun kaikki tarvittavat toimenpiteet on tehty ja kirjoittaa dokumentaatiokommenttiin ”comt” ja painaa **Ctrl+Space**, pitäisi Eclipsen luoda valmiiksi kohta johon testit kirjoitetaan. Tällöin dokumentaatiokommentteihin pitäisi ilmestyä seuraavat rivit:

```
* @example
* <pre name="test">
* </pre>
```

Testit kirjoitetaan noiden pre-tagien sisälle. Aloittavan pre-tagin name-attribuutin täytyy olla "test". Tagia ennen pitää antaa javadoc-työkalua varten @example-tagi. Nyt testit näkyvät esimerkkeinä dokumentaatiossa.

Aliohjelmat ja metodit testataan yksinkertaisesti antamalla niille parametreja ja kirjoittamalla mitä niiden odotetaan palauttavan. ComTest-testeissä käytetään erityistä vertailuoperaattoria, jossa on kolme yhtä suuri kuin -merkkiä (===). Tämä tarkoittaa, että arvon pitää olla sekä samaa tyyppiä, että sama. Kirjoitetaan esimerkiksi `keskiarvo`-aliohjelmalle testit.

```
/**
 * Laskee parametrina saamiensa lukujen keskiarvon.
 * @example
 * <pre name="test">
 * keskiarvo(0,0)    === 0.0;
 * keskiarvo(0,5)    === 2.5;
 * keskiarvo(-1,1)   === 0.0;
 * keskiarvo(10,20)  === 15.0;
 * keskiarvo(-5,-10) === -7.5;
 * </pre>
 * @param a ensimmäinen luku
 * @param b toinen luku
 * @return lukujen keskiarvo
```

```
 */
public static double keskiarvo(int a, int b) {
   return (a+b)/2.0;
}
```

Tarkastellaan testejä nyt hieman tarkemmin.

```
* keskiarvo(0,0)    === 0.0;
```

Yllä olevalla rivillä testataan, että jos `keskiarvo`-aliohjelma saa parametrikseen arvot 0 ja 0, niin myös sen palauttavan arvon tulisi olla 0.

```
keskiarvo(0,5) === 2.5;
```

Seuraavaksi testataan, että jos parametreistä ensimmäinen on luku 0 ja toinen luku 5, niin näiden keskiarvohan on tietenkin 2.5 eli myös aliohjelman tulee palauttaa luku 2.5 ja niin edelleen.

Aliohjelmassa `keskiarvo` on syytä tehdä testit sekä positiivisille että negatiivisille parametreille. Lisäksi täytyy tehdä testi, jossa toinen parametreista on negatiivinen ja toinen positiivinen.

Varsinaisen JUnit-testin voi nyt luoda ja ajaa painamalla hiiren oikeaa nappia ja valitsemalla ComTest → Generate, Run Junit. Jos JUnit-välilehti näyttää vihreää palkkia, testit menivät oikein. Punaisen palkin tapauksessa testit menivät joko väärin, tai sitten JUnit-tiedostossa on virheitä.

Myös testit täytyy testata. Voihan olla, että kirjoittamissamme testeissä on myös virheitä. Tämä onkin syytä testata kirjoittamalla testeihin virhe tarkoituksella. Tällöin JUnit-välilehdellä pitäisi tietenkin näkyä punainen palkki. Jos näin ei ole, on joku testeistä väärin. Hyvien testien kirjoittaminen on myös oma taitonsa. Kaikkia mahdollisia tilanteitahan ei millään voi testata, joten joudumme valitsemaan, mille parametreille testit tehdään. Täytyisi ainakin testata todennäköiset virhepaikat. Näitä ovat yleensä ainakin kaikenlaiset "ääritilanteet".

Esimerkkinä olevassa `keskiarvo`-aliohjelmassa tällaisia ääritilanteita ei oikeastaan ole, joten testiarvot on valittu melko sattumanvaraisesti. On kuitenkin syytä testata, että `keskiarvo`-aliohjelma toimii sekä positiivisilla että negatiivisilla arvoilla. Lisäksi kannattaa vielä testata niin, että toinen parametri on positiivinen ja toinen negatiivinen.

11.2 Liukulukujen testaaminen

Liukulukuja (`double` ja `float`) testataan ComTest:n vertailu operaattorilla, jossa on kolme aaltoviivaa (~~~). Tämä johtuu siitä, että kaikkia reaalilukuja ei pystytä esittämään tietokoneella tarkasti, joten täytyy sallia pieni virhemarginaali. Tehdään `keskiarvo`-aliohjelmasta versio, joka osaa laskea kahden `double`-tyyppisen luvun keskiarvon ja kirjoitetaan sille samalla dokumentaatiokommentit ja ComTest-testit. Aliohjelmalla voi olla sama nimi kuin kokonaislukuja laskevalla `keskiarvo`-aliohjelmalla, koska se saa erityyppiset parametrit. Kun samannimisille aliohjelmille kirjoitetaan eri parametreilla useita määrittelyjä sanotaan, että aliohjelmaa *kuormitetaan* (**overload**).

```
/**
 * Aliohjelma laskee parametrinaan saamiensa kahden
 * double-tyyppisen luvun keskiarvon.
 *
 * @example
 * <pre name="test">
 * keskiarvo(0.0,0.0)   ~~~  0.0;
 * keskiarvo(1.2,0.0)   ~~~  0.6;
 * keskiarvo(0.8,0.2)   ~~~  0.5;
 * keskiarvo(-0.1,0.1)  ~~~  0.0;
 * keskiarvo(-1.5,-2.5) ~~~ -2.0;
 * </pre>
 *
```

```
 * @param a toinen summattava luku
 * @param b toinen summattava luku
 * @return lukujen summan
 */
public static double keskiarvo(double a, double b) {
   return (a+b)/2.0;
}
```

Huomaa, että liukulukuja testattaessa täytyy parametrit antaa desimaaliosan kanssa. Esimerkiksi yllä olevassa esimerkissä ensimmäinen testi EI voi olla `keskiarvo(0,0) ~~~ 0.0`. Tällöin ajettaisiin aliohjelma `keskiarvo(int x, int y)`, jos sellainen olisi olemassa.

Testauksen tarkkuutta voidaan säätää `#TOLERANCE` -kommentilla. Kommentti asetetaan pre-tagien sisään ennen testejä.

```
#TOLERANCE="0.0001";
```

Esimerkiksi yllä oleva kommentti tarkoittaisi, että jos lukujen erotuksen itseisarvo on pienempi kuin 0.0001, tulkitaan ne vielä samaksi luvuksi.

12. Merkkijonot

Tutustutaan tarkemmin Javan merkkijonoihin. Merkkijonot voidaan jakaa muuttumattomiin ja muokattaviin. Javan muuttumaton merkkijono on tyypiltään String, johon olemmekin jo hieman tutustuneet olioiden yhteydessä. Muuttumatonta merkkijonoa ei voi muuttaa luomisen jälkeen. Muokattavan merkkijonon käsittely on sen sijaan monipuolisempaa. Kuitenkin hyvin usein pärjäämme pelkällä muuttumattomalla `String`-tyyppisellä merkkijonolla. Tutustutaan seuraavaksi molempiin hieman tarkemmin.

12.1 String

Merkkijono on kokoelma peräkkäisiä merkkejä. Tarkalleen ottaen merkkijono toteutetaan Javassa sisäisesti taulukkona, joka sisältää merkkejä (`char`). Taulukoista on tässä monisteessa oma lukunsa myöhemmin.

Olioiden yhteydessä tutustuimme jo hieman `String`-tyyppiin. Sen voi siis alustaa kahdella tavalla:

```
String henkilo1 = new String("Ville Virtanen");
String henkilo2 = "Kalle Korhonen";
```

Jälkimmäinen tapa muistuttaa enemmän alkeistietotyyppien alustamista, mutta merkkijonot ovat Javassa siis aina olioita.

12.1.1 Hyödyllisiä metodeja

`String`-luokassa on paljon hyödyllisiä metodeja, joista käsitellään nyt muutama. Kaikki metodit näet Javan dokumentaatiosta.

- `equals(Object anObject)` Palauttaa tosi jos kaksi oliota ovat sisällöltään samat.

```
if (etunimi.equals("Aku")) System.out.println("Löytyi!");
```

- `equalsIgnoreCase(String anotherString)` Palauttaa tosi jos kaksi merkkijonoa on sisällöltään samoja jos kirjainten koko unohdetaan.

```
if (etunimi.equalsIgnoreCase("aku")) System.out.println("Löytyi!");
```

- `length()` Palauttaa merkkijonon pituuden.

```
String henkilo = "Ville";
System.out.println(henkilo.length()); //tulostaa 5
```

- `charAt(int index)` Palauttaa merkkijonosta tietyn merkin. Parametrina saa indeksin eli merkkijonon kohdan, josta merkki palautetaan. Merkkijonojen indeksointi alkaa Javassa nollasta, eli ensimmäinen merkki on indeksissä 0. Viimeisen merkin indeksin saa, kun vähentää merkkijonon pituudesta yhden.

```
String henkilo = "Ville";
char ekaMerkki = henkilo.charAt(0);
char vikaMerkki = henkilo.charAt(henkilo.length() - 1);
```

- `substring(int beginIndex)` Palauttaa osan merkkijonosta alkaen parametrinaan saamastaan indeksistä.

```
String henkilo = "Ville Virtanen";
String sukunimi = henkilo.substring(6);
```

- `substring(int beginIndex, int endIndex)` Palauttaa osan merkkijonosta

parametrinaan saamiensa indeksien välistä. Ensimmäinen parametri on palautettavan merkkijonon ensimmäisen merkin indeksi ja toinen parametri palautettavan merkkijonon viimeistä merkkiä seuraava indeksi.

```
String henkilo = "Ville Virtanen";
String etunimi = henkilo.substring(0,5); //etunimi: "Ville"
```

- `toLowerCase()` Palauttaa merkkijonon niin, että kaikki kirjaimet on muutettu pieniksi kirjaimiksi.

```
String henkilo = "Ville Virtanen";
System.out.println(henkilo.toLowerCase()); //tulostaa "ville virtanen"
```

- `toUpperCase()` Palauttaa merkkijonon niin, että kaikki kirjaimet on muutettu isoiksi kirjaimiksi.

```
String henkilo = "Ville Virtanen";
System.out.println(henkilo.toUpperCase()); //tulostaa "VILLE VIRTANEN"
```

- `replace(char oldChar, char newChar)` Korvaa merkkijonon kaikki tietyt merkit toisilla merkeillä. Ensimmäisenä parametrina korvattava merkki ja toisena korvaaja.

```
String sana = new String ("katti");
kissa = sana.replace('t', 's');
System.out.println(sana);  //tulostaa "kassi"
```

- `compareTo(String anotherString)` Vertaa merkkijonon aakkosjärjestystä toiseen merkkijonoon. Palauttaa arvon 0 jos merkkijonot ovat samat, nollaa pienemmän arvon jos kyseinen merkkijono on aakkosjärjestykseltään ennen kuin parametrina annettu ja nollaa suuremman arvon jos kyseinen merkkijono on parametrina annetun jälkeen.

```
String henkilo1 = "Korhonen";
String henkilo2 = "Virtanen";
henkilo1.compareTo(henkilo2);//Palauttaisi jonkun nollaa pienemmän luvun
```

Metodia `compareTo` käytetään, kun pitää verrata kumpi jono on ennen toista aakkosissa. Metodista on myös versio, joka vertaa järjestystä välittämättä kirjasinkoosta: `compareToIgnoreCase`.

12.2 Muokattavat merkkijonot: esimerkkinä StringBuilder

Niin sanottujen muuttumattomien merkkijonojen (String) lisäksi Javassa on muuttuvia merkkijonoja. Muuttuvien merkkijonojen idea on, että voimme lisätä ja poistaa siitä merkkejä luomisen jälkeen. `String`-tyyppisen merkkijonon muuttaminen ei onnistu sen luomisen jälkeen, vaan jos haluaisimme muuttaa sitä, täytyisi meidän luoda kokonaan uusi merkkijono.

Eräs muuttuva merkkijonoluokka Javassa on `StringBuilder`. Merkkijonon perään lisääminen onnistuu `append`-metodilla. `append`-metodilla voi lisätä merkkijonon perään muun muassa kaikkia Javan alkeistietotyyppejä sekä `String`-olioita. Myös kaikkien Javasta valmiina löytyvien olioiden lisääminen onnistuu `append`-metodilla, sillä ne sisältävät `toString`-metodin, jolla oliot voidaan muuttaa merkkijonoksi. Alla oleva koodinpätkä esittelee `append`-metodia.

```
double a = 3.5;
int b = 6;
double c = 9.5;

StringBuilder yhtalo = new StringBuilder();
yhtalo.append("f(x): ");     //yhtalo: "f(x): "
yhtalo.append(a);            //yhtalo: "f(x): 3.5"
yhtalo.append(" + ");        //yhtalo: "f(x): 3.5 + "
yhtalo.append(b);            //yhtalo: "f(x): 3.5 + 6"
yhtalo.append('x');          //yhtalo: "f(x): 3.5 + 6x"
```

```
yhtalo.append(" = ");          //yhtalo: "f(x): 3.5 + 6x = "
yhtalo.append(c);              //yhtalo: "f(x): 3.5 + 6x = 9.5"
```

Tiettyyn paikkaan pystyy merkkejä ja merkkijonoja lisäämään `insert`-metodilla. `insert`-metodi saa parametrikseen indeksin eli kohdan johon merkki tai merkit lisätään, sekä lisättävän merkin tai merkit. Indeksointi alkaa nollasta. Merkkijonon ensimmäinen merkki on siis indeksissä 0. `insert`-metodilla voi lisätä myös kaikkia samoja tietotyyppejä kuin `append`-metodillakin. Voisimme esimerkiksi lisätä edelliseen esimerkkiin luvun 3.5 perään vielä muuttujan x.

```
yhtalo.insert(9,'x');        //yhtalo: "f(x): 3.5x + 6x = 9.5"
```

Huomaa, että `insert`-metodi ei korvaa indeksissä 9 olevaa merkkiä, vaan lisää merkkijonoon kokonaan uuden merkin. Merkkijonon pituus kasvaa siis yhdellä. Korvaamiseen on olemassa oma `replace`-metodinsa.

12.2.1 Muita hyödyllisiä metodeja

- `charAt(int index)` Palauttaa merkin (`char`) tietystä merkkijonon kohdasta.
- `delete(int start, int end)` Poistaa merkit parametrinaan saamaltaan väliltä.
- `deleteCharAt(int index)` Poistaa merkin tietystä merkkijonon kohdasta.
- `length()` Palauttaa merkkijonon pituuden.

12.2.2 StringBuffer

`StringBuffer` on `StringBuilderia` vastaava muokattava merkkijono. Itse asiassa se sisältää kaikki samat metodit kuin `StringBuilder`. Erot löytyvätkin niiden sisäisestä toteutuksesta. `StringBuffer` on ollut mukana Javan versiosta JDK 1.0 asti, kun taas `StringBuilder` tuli vasta versioon 1.5. Kaikissa uusissa ohjelmissa kannattaa käyttää `StringBuilderia`, sillä se on nopeampi. Kuitenkin vanhoissa esimerkeissä ja ohjelmissa voi esiintyä myös `StringBuffer`, joten on hyvä tietää myös mikä se on.

12.3 Merkkijonojen tulostaminen

Tulostuksessa voidaan merkkijonoja yhdistellä "+"-merkillä kuten alla:

```
String etunimi = "Ville";
String sukunimi = "Virtanen";
System.out.println(etunimi + " " + sukunimi); //tulostaisi: Ville Virtanen
System.out.println(sukunimi + ", " + etunimi); //tulostaisi: Virtanen, Ville
```

"+"-merkillä on Javassa siis kaksi merkitystä. Sillä voidaan yhdistellä merkkijonoja tai se voi toimia aritmeettisena operaattorina (eli voidaan laskea numeeristen arvojen summia).

12.3.1 Huomautus: Aritmeettinen+ vs. merkkijonoja yhdistelevä+

Merkkijonoihin voidaan ”+”-merkkiä käyttämällä yhdistellä myös numeeristen muuttujien arvoja. Tällöin ero siinä, että toimiiko ”+”-merkki aritmeettisena operaattorina vai merkkijonoja yhdistelevänä operaattorina on todella pieni. Tutki alla olevaa esimerkkiä:

```
public class PlusMerkki {

   public static void main(String args[]) {
      int luku1 = 2;
      int luku2 = 5;

      //tässä "+"-merkki toimii aritmeettisena operaattorina
      System.out.println(luku1 + luku2); //tulostaa 7

      //tässä "+"-merkki toimii merkkijonoja yhdistelevänä operaattorina
```

```
        System.out.println(luku1 + "" + luku2);  //tulostaa 25

        //Tässä ensimmäinen "+"-merkki toimii aritmeettisena operaattorina
        //ja toinen "+"-merkki merkkijonoja yhdistelevänä operaattorina
        System.out.println(luku1 + luku2 + "" + luku1); //tulostaa 72
    }
}
```

Merkkijonojen yhdistäminen luo aina uuden olion ja siksi sitä on käytettävä harkiten, silmukoissa jopa kokonaan `StringBuilder`illä ja `append`-metodilla korvaten.

12.3.2 Vinkki: Näppärä tyyppimuunnos String-tyypiksi

Itse asiassa lisäämällä muuttujaan ”+”-merkillä merkkijono, tekee Java automaattisesti tyyppimuunnoksen ja muuttaa muuttajasta ja siihen lisätystä merkkijonosta `String`-tyyppisen. Tämän takia voidaan alkeistietotyyppiset muuttujat muuttaa näppärästi `String`-tyyppisiksi lisäämällä muuttujan eteen tyhjä merkkijono.

```
int luku = 23;
boolean totuusarvo = false;

String merkkijono1 = "" + luku;
String merkkijono2 =  "" + totuusarvo;
```

Ilman tuota tyhjän merkkijonon lisäämistä tämä ei onnistuisi, sillä `String`-tyyppiseen muuttujaan ei tietenkään voi tallentaa `int`- tai `boolean`-tyyppistä muuttujaa.

Tämä ei kuitenkaan mahdollista reaaliluvun muuttamista `String`-tyypiksi tietyllä tarkkuudella. Tähän on apuna `String`-luokan `format`-metodi.

12.3.3 Metodi: Reaalilukujen muotoilu String.format-metodilla

`String`-luokan `format`-metodi tarjoaa monipuoliset muotoilumahdollisuudet useille tietotyypeille, mutta katsotaan tässä kuinka sillä voi muotoilla reaalilukuja. `Math`-luokasta saa luvun pii 15 desimaalin tarkkuudella kirjoittamalla `Math.PI`. Huomaa, että `PI` ei ole metodi, joten perään ei tule sulkuja. PI on `Math`-luokan julkinen staattinen vakio (**public static final**). Jos haluaisimme muuttaa piin `String`-tyypiksi vain kahden desimaalin tarkkuudella, onnistuisi se seuraavasti:

```
String pii = String.format("%5.2f",Math.PI); //pii = " 3,14"
```

Tässä `format`-metodi saa kaksi parametria. Ensimmäistä parametria sanotaan muotoilu merkkijonoksi (**format string**). Toisena parametrina on sitten muotoiltava arvo.

`Format`-merkkijonon prosenttimerkki kertoo, että tästä alkaa varsinaiset muotoilutiedot. Siihen asti olevat merkit tallennetaan sellaisenaan. Luku 5 tarkoittaa, että merkkijonosta tehdään viiden merkin pituinen. Jos merkkijonosta ei muuten tulisi viiden merkin pituista, lisättäisiin alkuun tyhjiä merkkejä. Tämän avulla saa tulostettua nätisti eri mittaisia desimaalilukuja allekkain. Katso alempaa esimerkkiä. Pisteen jälkeen tuleva luku 2 ilmoittaa monenko desimaalin tarkkuudella luku tallennetaan. Viimeisenä oleva ”f”-merkki ilmoittaa, että lukua käsitellään desimaalilukuna.

12.3.4 Metodi: Muotoilujen tulostaminen System.out.printf-metodilla

Muotoilujen tulostaminen voidaan tehdä käyttämällä `System.out.printf`-metodia. Sen parametrit ja toiminta ovat täysin samat kuin `String.format`-metodilla, mutta nyt muotoiltu merkkijono ainoastaan tulostuu. Edellinen esimerkki voitaisiin siis tulostaa nyt seuraavasti:

```
System.out.printf("%5.2f",Math.PI);
```

Eri pituisia desimaalilukuja on helppo tulostaa allekkain käyttämällä muotoiltua tulostusta.

Tulostetaan seuraavaksi muutamia pörssikursseja.

```
double tieto = 10.1;
double nokia = 9.36;
double google = 429.17;

System.out.println("Pörssikurssit 18.7.2009");
System.out.printf("%-12s %10.2f € \n", "Tieto Oyj:",tieto);
System.out.printf("%-12s %10.2f € \n", "Nokia Oyj:",nokia);
System.out.printf("%-12s %10.2f $ \n", "Google Inc.:",google);
```

Yllä oleva koodi tulostaisi:

```
Pörssikurssit 18.7.2009
Tieto Oyj:        10,10 €
Nokia Oyj:         9,36 €
Google Inc.:     429,17 $
```

Tässä esimerkissä muotoillaan kahta eri asiaa. Yrityksen nimeä ja yrityksen pörssikurssia. Muotoiltavia asioita voi määritellä siis useita. Metodin parametrien määrä vain lisääntyy jokaista muotoiltavaa asiaa kohden yhdellä.

Format-merkkijonon '-'-merkki on niin sanottu lippu (**flag**). Tämä kyseinen lippu tarkoittaa, että arvo sisennetään poikkeuksellisesti vasemmalle. Lippuja on myös muita ja niillä voidaan antaa muotoilulle lisämäärityksiä. Koska ensimmäinen muotoiltava tietotyyppi on nyt `String`, ilmoitetaan tämä leveysmäärityksen jälkeen s-kirjaimella. Viimeisenä olevat kenoviiva ja n-kirjain (`\n`) tarkoittavat, että tulostetaan rivinvaihto. Tämä ei liity varsinaisesti muotoiltuun tulostukseen, vaan tulostettaessa merkkijono "\n" tulostetaan aina rivinvaihto.

Tarkat tiedot format-jonon syntaksista löytyvät Javan dokumentaatiosta

http://java.sun.com/javase/6/docs/api/java/util/Formatter.html#syntax.

13. Ehtolauseet (Valintalauseet)

"Älä turhaan käytä iffiä, useimmiten pärjäät ilmankin" - Vesa Lappalainen

13.1 Mihin ehtolauseita tarvitaan?

Tehtävä: Suunnittele aliohjelma, joka saa parametrina kokonaisluvun. Aliohjelman tulee palauttaa `true` (tosi), jos luku on parillinen ja `false` (epätosi), jos luku on pariton.

Tämänhetkisellä tietämyksellä yllä olevan kaltainen aliohjelma olisi lähes mahdoton toteuttaa. Pystyisimme kyllä selvittämään onko luku parillinen, mutta meillä ei ole keinoa muuttaa paluuarvoa sen mukaan, onko luku parillinen vai ei. Kun ohjelmassa haluamme tehdä eri asioita riippuen esimerkiksi käyttäjän syötteistä tai aliohjelmien parametreista, tarvitsemme ehtolauseita.

Tavallinen ehtolause sisältää aina sanan ”jos”, ehdon ja toimenpiteet mitä tehdään jos ehto on tosi. Arkielämän ehtolause voitaisiin ilmaista vaikka seuraavasti: ”Jos aurinko paistaa, mene ulos.” Hieman monimutkaisempi ehtolause voisi sisältää myös ohjeen, mitä tehdään jos ehto ei pädekään: ”Jos aurinko paistaa, mene ulos, muuten koodaa sisällä”. Molemmille rakenteille löytyy Javasta vastineet. Tutustutaan ensiksi ensimmäiseen eli `if`-rakenteeseen.

13.2 if-rakenne: ”Jos aurinko paistaa, mene ulos.”

Yleisessä muodossa Javan `if`-rakenne on alla olevan kaltainen:

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

Esimerkki ehtolause: ”Jos aurinko paistaa, mene ulos” voidaan nyt esittää Javan syntaksin mukaan seuraavasti:

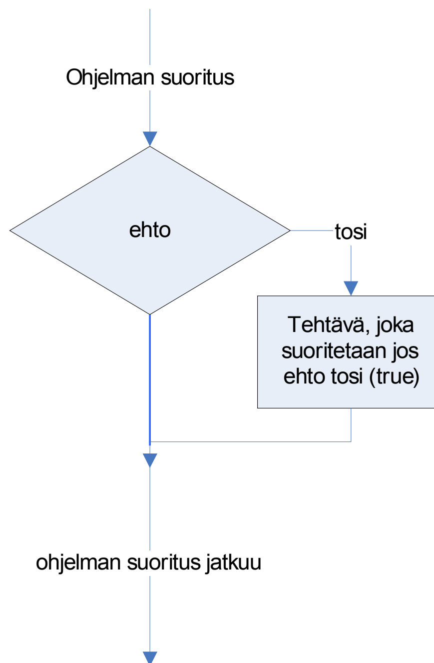
```
if ( aurinkoPaistaa ) meneUlos();
```

Jos ehdon ollessa totta täytyy suorittaa useampia lauseita, tulee ehdon jälkeen muodostaa oma lohko.

```
if ( ehto ) {
  lausi1;
  lause2;
  .
  .
  lause n;
}
```

Ehto on aina joku *looginen lauseke*. Looginen lauseke voi saada vain kaksi arvoa: tosi (`true`) tai epätosi (`false`). Jos looginen lauseke saa arvon ”tosi”, perässä oleva lause tai lauseet suoritetaan, muuten ei tehdä mitään ja jatketaan ohjelman suoritusta. Looginen lauseke voi sisältää muun muassa lukuarvoja, joiden suuruuksia voidaan vertailla vertailuoperaattoreilla.

Vuokaaviolla `if`-rakennetta voisi kuvata seuraavasti:



Kuva 13: if-rakenne vuokaaviona

vuokaavio = Kaavio, jolla mallinnetaan *algoritmia* tai prosessia.

13.3 Vertailuoperaattorit

Vertailuoperaattoreilla voidaan vertailla aritmeettisia arvoja.

Operaattori	Nimi	Toiminta
==	yhtä suuri kuin	Palauttaa tosi, jos vertailtavat arvot yhtä suuret.
!=	eri suuri kuin	Palauttaa tosi, jos vertailtavat arvot erisuuret.
>	suurempi kuin	Palauttaa tosi, jos vasemmalla puolella oleva luku on suurempi.
>=	suurempi tai yhtä suuri kuin	Palauttaa tosi, jos vasemmalla puolella oleva luku on suurempi tai yhtä suuri
<	pienempi kuin	Palauttaa tosi, jos vasemmalla puolella oleva luku on pienempi.
<=	pienempi tai yhtä suuri kuin	Palauttaa tosi, jos vasemmalla puolella oleva luku on pienempi tai yhtä suuri.

13.3.1 Huomautus: Sijoitusoperaattori (=) ja vertailuoperaattori (==)

Varo, ettet käytä sijoitusoperaattoria (=) vertailuun. Tämä on yksi yleisimmistä ohjelmointivirheistä. Vertailuun aina kaksi yhtä suuri kuin -merkkiä ja sijoitukseen yksi.

13.3.2 Vertailuoperaattoreiden käyttö

```
if ( luku < 0 ) System.out.println("Luku on negatiivinen");
```

Yllä oleva lauseke tulostaa "Luku on negatiivinen", jos muuttuja `luku` on pienempi kuin nolla. Ehtona on siis looginen lauseke `luku < 0`, joka saa arvon "tosi", aina kun muuttuja `luku` on nollaa pienempi. Kun ehto saa arvon "tosi", perässä oleva lause tai lohko suoritetaan.

13.4 if-else -rakenne

`if-else` –rakenne sisältää myös kohdan mitä tehdään jos ehto ei olekaan tosi.

```
Jos aurinko paistaa mene ulos, muuten koodaa sisällä.
```

Yllä oleva lause sisältää ohjelmoinnin `if-else` –rakenteen idean. Siinä on ehto ja ohje mitä tehdään jos ehto on tosi sekä ohje mitä tehdään jos ehto on epätosi. Lauseen voisi kirjoittaa myös:

```
jos ( aurinko paistaa ) mene ulos
muuten koodaa sisällä
```

Yllä oleva muoto on jo useimpien ohjelmointikielten syntaksin mukainen. Siinä ehto on erotettu sulkeiden sisään, ja perässä on ohje, mitä tehdään jos ehto on tosi. Toisella rivillä sen sijaan on ohje mitä tehdään jos ehto on epätosi. Javan syntaksin mukaiseksi ohjelma saadaan, kun ohjelmointikieleen kuuluvat sanat muutetaan englanniksi.

```
if ( aurinko paistaa ) mene ulos;
else koodaa sisällä;
```

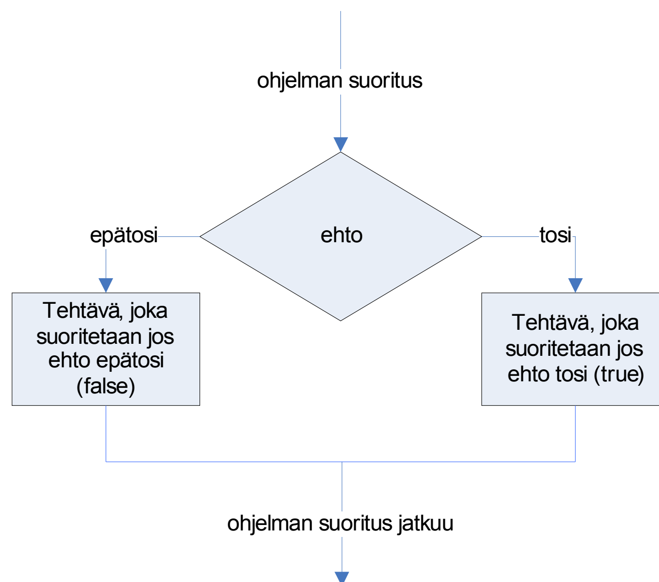
`if-else` -rakenteen yleinen muoto:

```
if ( ehto ) lause1;
else lause2;
```

Kuten pelkässä `if`-rakenteessa myös `if-else` –rakenteessa lauseiden tilalla voi olla myös lohko.

```
if ( ehto ) {
  lause1;
  lause2;
  lause3;
} else {
  lause4;
  lause5;
}
```

`if-else` -rakennetta voisi sen sijaan kuvata seuraavalla vuokaaviolla:



Kuva 14: If-else -rakenne vuokaaviona

13.4.1 Esimerkki: Pariton vai parillinen

Tehdään aliohjelma joka palauttaa `true` jos luku on parillinen ja `false` jos luku on pariton.

```
public static boolean onkoLukuParillinen(int luku) {
   if ( (luku % 2) == 0 ) return true;
   else return false;
}
```

Aliohjelma saa parametrina kokonaisluvun ja palauttaa siis `true`, jos kokonaisluku oli parillinen ja `false`, jos kokonaisluku oli pariton. Toisella rivillä otetaan muuttujan `luku` ja luvun 2 jakojäännös. Jos jakojäännös on `0`, niin silloin luku on parillinen, eli palautetaan `true`. Jos jakojäännös ei mennyt tasan, niin silloin luvun on pakko olla pariton eli palautetaan `false`.

Itse asiassa, koska aliohjelman suoritus päättyy `return`-lauseeseen, voitaisiin `else`-sana jättää kokonaan pois. Aliohjelman `else`-lauseeseenhan mennään ohjelmassa nyt vain siinä tapauksessa, että `if`-ehto ei ollut tosi. Voisimmekin kirjoittaa aliohjelman hieman lyhyemmin seuraavasti:

```
public static boolean onkoLukuParillinen(int luku) {
   if ( (luku % 2) == 0 ) return true;
   return false;  // Huom! Ei tarvita else
}
```

Usein `if`-lauseita käytetään aivan liikaa. Tämänkin esimerkin voisi yhtä hyvin kirjoittaa vieläkin lyhyemmin (ei aina selkeämmin kaikkien mielestä) seuraavasti:

```
public static boolean onkoLukuParillinen(int luku) {
   return ( (luku % 2) == 0 );
}
```

Tämä johtuu siitä, että lauseke `( (luku % 2) == 0 )`, on `true` jos luku on parillinen ja muuten `false`. Saman tien voimme siis palauttaa suoraan tuon lausekkeen arvon ja aliohjelma toimii täysin samanlailla.

13.5 Loogiset operaatiot

Loogisia lausekkeita voidaan myös yhdistellä *loogisilla operaattoreilla*.

Java-koodi	Operaattori	Toiminta
!	looginen ei	Tosi, jos lauseke epätosi.
&	looginen ja	Tosi, jos molemmat lausekkeet tosia.
&&	looginen ehdollinen ja	Tosi, jos molemmat lausekkeet tosia. Eroaa edellisestä siinä, että jos lausekkeen totuusarvo on jo saatu selville, niin loppua ei enää tarkisteta. Toisin sanoen jos ensimmäinen lauseke oli jo epätosi, niin toista lauseketta ei enää suoriteta.
|	looginen tai	Tosi, jos toinen lausekkeista on tosi.
||	looginen ehdollinen tai	Tosi, jos toinen lausekkeista on tosi. Vastaavasti jos lausekkeen arvo selviää jo aikaisemmin, niin loppua ei enää tarkisteta. Toisin sanoen, jos ensimmäinen lauseke saa arvon tosi, niin koko lauseke saa arvon tosi ja jälkimmäistä ei tarvitse enää tarkastaa.

Ei-operaattori kääntää loogisen lausekkeen päinvastaiseksi.

```
if ( !(luku <= 0) ) System.out.println("Luku on suurempi kuin nolla");
```

Ei-operaattori siis palauttaa vastakkaisen `boolean`-arvon. Todesta tulee epätosi ja epätodesta tosi. Jos yllä olevassa lauseessa `luku`-muuttuja saisi arvon 5, niin ehto `luku <= 0` saisi arvon `false`. Kuitenkin ei-operaattori saa arvon `true`, kun lausekkeen arvo on `false`, joten koko ehto onkin `true` ja perässä oleva tulostuslause tulostuisi. Lause olisi siis sama kuin alla oleva:

```
if ( 0 < luku ) System.out.println("Luku on suurempi kuin nolla");
```

Ja-operaatiossa molempien lausekkeiden pitää olla tosia, että koko ehto olisi tosi.

```
If ( (1 <= luku) && ( luku <= 99 ) ) System.out.println("Luku on välillä 1-99");
```

Yllä oleva ehto toteutuu, jos luku välillä 1-99. Vastaava asia voitaisiin hoitaa myös sisäkkäisillä ehtolauseilla seuraavasti

```
if ( 1 <= luku )
  if ( luku <= 99 ) System.out.println("Luku on välillä 1-99");
```

Tällaisia sisäkkäisiä ehtolauseita pitäisi kuitenkin välttää, sillä ne lisäävät virhealttiutta ja vaikeuttavat testaamista.

Epäyhtälöiden lukemista voi helpottaa, mikäli ne kirjoitetaan niin, käytetään aina nuolta vasemmalle merkkiä. Tällöin epäyhtälön operandit ovat samassa järjestyksessä kuin ihmiset mieltävät lukujen suuruusjärjestyksen.

13.5.1 De Morganin lait

Huomaa, että joukko-opista ja logiikasta tutut De Morganin lait pätevät myös loogisissa operaatioissa. Olkoon p ja q `boolean`-tyyppisiä muuttujia. Tällöin:

```
!(p || q) on sama asia kuin !p && !q
!(p && q) on sama asia kuin !p || !q
```

Lakeja voisi testata alla olevalla koodinpätkällä vaihtelemalla muuttujien p ja q arvoja. Riippumatta muuttujien p ja q arvoista tulostusten pitäisi aina olla `true`.

```
public class DeMorgansLaws {

       /**
```

```
         * Testiohjelma DeMorganin laeille
         */
        public static void main(String[] args) {
                boolean p = true;
                boolean q = true;
                System.out.println(!( p || q ) == ( !p && !q ) );
                System.out.println(!( p && q ) == ( !p || !q ) );
        }
}
```

De Morganin lakia käyttämällä voidaan lausekkeita joskus saada sievemmiksi. Tällaisinaan lauseet tuntuvat turhilta, mutta jos `p` ja `q` ovat esimerkiksi epäyhtälöitä:

```
if ( !( a < 5  && b < 3 ) ) ...
if ( !(a < 5) || ! (b < 3 ) ) ...
if ( a >= 5 || b >= 3 ) ...
```

niin **not** -operaattorin siirto voikin olla mielekästä. Toinen tällainen laki on osittelulaki.

Tee ComTest-testi, joka todistaa, että De Morganin -laki pätee. Eli testaa De Morganin -laki kaikilla mahdollisilla muuttujien p ja q arvojen kombinaatioilla.

13.5.2 Osittelulaki

Osittelulaki sanoo, että:

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

Samaistamalla `*  <=>  &&` ja `+  <=>  ||` todetaan loogisille operaatioillekin osittelulaki:

```
p && (q || r) = (p && q) || (p && r)
```

Päinvastoin kuin normaalissa logiikassa, loogisille operaatioille osittelulaista on myös toinen versio:

```
p || (q && r) = (p || q) && (p || r)
```

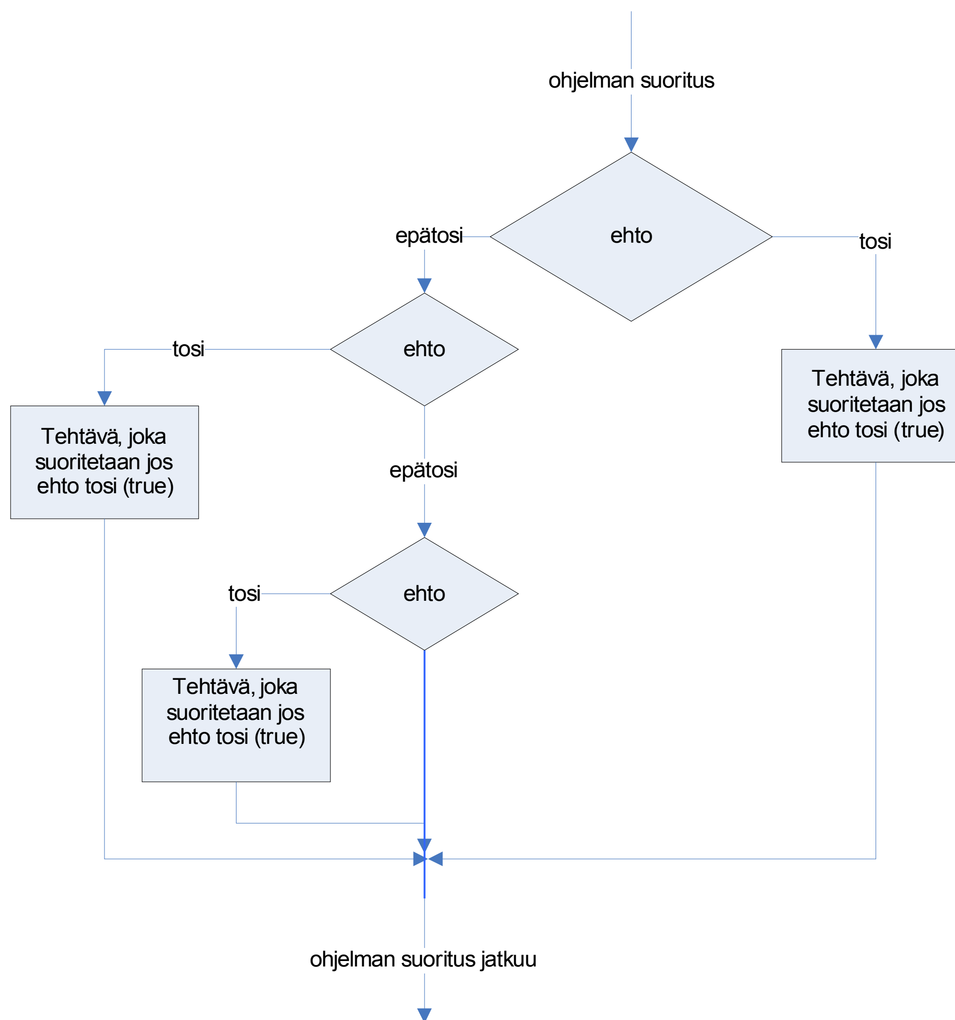
13.6 else if -rakenne

Jos muuttujalle täytyy tehdä monia toisensa poissulkevia tarkistuksia, voidaan käyttää erityistä `else if`-rakennetta. Siinä on kaksi tai useampia ehtolauseita ja seuraavaan ehtoon mennään vain, jos mikään aikaisemmista ehdoista ei ollut tosi. Rakenne on yleisessä muodossa seuraava.

```
if ( ehto1 ) lause1;
else if ( ehto2 ) lause2;
else if ( ehto3 ) lause3;
else lause4;
```

Alimpana olevaan `else`-osaan mennään nyt vain siinä tapauksessa, että mikään yllä olevista ehdoista ei ollut tosi. Tämä rakenne esitellään usein omana rakenteenaan vaikka oikeastaan tässä on vain useita peräkkäisiä `if-else` -rakenteita, joiden sisennys on vain hieman poikkeava.

Seuraava vuokaavio kuvaisi rakennetta, jossa on yksi `if`-lause ja sen jälkeen kaksi `else if`-lausetta.



Kuva 15: else-if -rakenne vuokaaviona

13.6.1 Esimerkki: Tenttiarvosanan laskeminen

Tehdään laitoksen henkilökunnalle aliohjelma, joka laskee opiskelijan tenttiarvosanan. Parametrinaan aliohjelma saa tentin maksimipistemäärän, läpipääsyrajan sekä opiskelijan pisteet. Aliohjelma palauttaa arvosanan 0-5 niin, että arvosanan 1 saa läpipääsyrajalla ja muut arvosanat skaalataan mahdollisimman tasaisesti.

```
public class LaskeTenttiArvosana {

  /**
   * Laskee tenttiarvosanan.
   *
   * @param maksimipisteet Tentin maksimipisteet
   * @param lapipaasyraja Tentin läpipääsyraja
   * @param tenttipisteet Opiskelijan saamat tenttipisteet
   *
   * @return tenttiarvosana välillä 0-5.
   */
  public static int laskeArvosana(int maksimipisteet, int lapipaasyraja,
    int tenttipisteet) {

    //Lasketaan eri arvosanoille tasaiset piste välit
    int arvosanojenPisteErot = (maksimipisteet - lapipaasyraja) / 5;
    int arvosana = 0;

    if( lapipaasyraja + 4*arvosanojenPisteErot < tenttipisteet ) arvosana = 5;
    else if( lapipaasyraja + 3*arvosanojenPisteErot < tenttipisteet ) arvosana = 4;
    else if( lapipaasyraja + 2*arvosanojenPisteErot < tenttipisteet ) arvosana = 3;
    else if( lapipaasyraja + arvosanojenPisteErot < tenttipisteet ) arvosana = 2;
    else if( lapipaasyraja <= tenttipisteet) arvosana = 1;
    return arvosana;
```

```
  }


  public static void main(String[] args) {
    //Tehdään muutama testi tulostus
    System.out.println(laskeArvosana(100,50,75));
    System.out.println(laskeArvosana(24,12,12));
  }
}
```

Aliohjelmassa lasketaan aluksi eri arvosanojen välinen piste-ero, jota käytetään arvosanojen laskemiseen. Arvosanojen laskeminen aloitetaan ylhäältä alaspäin. Ehto voi sisältää myös aritmeettisia operaatioita. Lisäksi alustetaan muuttuja `arvosana`, johon talletetaan opiskelijan saama arvosana. Muuttujaan `arvosana` talletetaan 5, jos tenttipisteet ylittävät läpipääsyrajan johon lisätään arvosanojen välinen piste-ero kerrottuna neljällä. Jos opiskelijan pisteet eivät riittäneet arvosanaan 5, mennään seuraavaan `else-if` -rakenteeseen ja tarkastetaan riittävätkö pisteet arvosanaan 4. Näin jatketaan edelleen kunnes kaikki arvosanat on käyty läpi. Lopuksi palautetaan muuttujan `arvosana` arvo. Pääohjelmassa aliohjelmaa on testattu muutamalla testitulostuksella.

Tässäkin esimerkissä monet `if`-lauseet voitaisiin välttää *taulukoinnilla*. Tästä puhutaan luvussa 14 Taulukot.

Miten ohjelmaa pitäisi muuttaa, jos pisteiden tarkastus aloitettaisiin arvosanasta 0?

Lyhenisikö koodi ja tarvittaisiinko `else`-lauseita, jos lause `arvosana = 5;` korvattaisiin lauseella `return 5;` ?

13.7 switch-rakenne

`switch`-rakennetta voidaan käyttää silloin, kun meidän täytyy suorittaa valintaa yksittäisten kokonaislukujen tai merkkien (`char`) perusteella. Jokaista odotettua muuttujan arvoa kohtaan on `switch`-rakenteessa oma `case`-osa, johon kirjoitetaan toimenpiteet, jotka tehdään tässä tapauksessa. Yleinen muoto `switch`-rakenteelle on seuraava.

```
switch (valisin) //valitsin on useimmiten joku muuttuja
{
   case arvo1:
      lauseet;
      break;

   case arvo2:
      lauseet;
      break;

   case arvoX:
      lauseet;
      break;

   default:
      lauseet;
}
```

Jokaisessa `case`-kohdassa on lauseiden jälkeen oltava `break`-lause, jolla hypätään pois lohkosta, eli `switch`-lauseen ohi. Jos `break`-lausetta ei olisi, suoritettaisiin muidenkin `case`-osien lauseet. Jos valitsimen arvo ei ollut mikään `case`-arvoista, mennään `default`-osaan.

13.7.1 Esimerkki: Arvosana kirjalliseksi

Tehdään aliohjelma, joka saa parametrina tenttiarvosanan numerona (0-5) ja palauttaa kirjallisen arvosanan `String`-oliona.

```
/**
 * Palauttaa parametrina saamansa numeroarvosanan kirjallisena.
 * @param numero tenttiarvosana numerona
 * @return tenttiarvosana kirjallisena
 */
public static String kirjallinenArvosana(int numero) {
   String arvosana = "";
   switch(numero) {
      case 0:
         arvosana = "Hylätty";
         break;

      case 1:
         arvosana = "Välttävä";
         break;

      case 2:
         arvosana = "Tyydyttävä";
         break;

      case 3:
         arvosana = "Hyvä";
         break;

      case 4:
         arvosana = "Kiitettävä";
         break;

      case 5:
         arvosana = "Erinomainen";
         break;

      default:
         arvosana = "Virheellinen arvosana";
   }
   return arvosana;
}
```

Koska `return`-lause lopettaa metodin toiminnan, voitaisiin yllä olevaa aliohjelmaa lyhentää palauttamalla jokaisessa `case`-osassa suoraan kirjallinen arvosana. Tällöin `break`-lauseet voisi jättää pois.

```
public static String kirjallinenArvosana(int numero) {
   switch(numero) {
      case 0:
         return "Hylätty";

      case 1:
         return "Välttävä";

      case 2:
         return "Tyydyttävä";

      case 3:
         return "Hyvä";

      case 4:
         return "Kiitettävä";

      case 5:
         return "Erinomainen";

      default:
         return "Virheellinen arvosana";
   }
}
```

`break`-lauseen voi siis turvallisesti jättää pois `case`-osasta, jos `case`-osassa palautetaan joku arvo `return`-lauseella. Kun `return`-lause tulisi ennen `break`-lausetta, ei `break`-lausetta kuitenkaan koskaan suoritettaisi. Muulloin `break`-lauseen poisjättäminen johtaa siihen, että suoritusta jatketaan seuraavasta `case`-kohdasta. Joskus näin halutaankin, mutta aloittelijalle `break`in puuttuminen on lähes aina virhe.

Lähes aina `switch`-rakenteen voi korvata `if` ja `else-if` –rakenteilla, niinpä sitä on pidettävä

vain yhtenä `if`-lauseena. Myös `switch`-rakenteen voi usein välttää käyttämällä taulukoita.

13.8 Esimerkki: Olioiden ja alkeistietotyyppien erot

Tehdään ohjelma, jolla demonstroidaan olioiden ja alkeistietotyyppien eroja.

```
/**
 * Tutkitaan olioviitteiden käyttäytymistä
 * @author Vesa Lappalainen
 * @version 1.0, 08.01.2003
 */
class Jonotesti {

   /**
    * Testaillaan olioiden ja alkeismuuttujien eroja.
    * @param args
    */
   public static void main(String[] args) {
      String s1 = "eka";
      String s2 = new String("eka");

      System.out.println(s1 == s2);       // false
      System.out.println(s1.equals(s2));  // true

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

      System.out.println(i1 == i2);       // true

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

      System.out.println(io1 == io2);       // false
      System.out.println(io1.equals(io2));  // true
      System.out.println(io1.intValue()== io2.intValue()); // true

      s2 = s1;
      System.out.println(s1 == s2);         // true
   }
}
```

Tarkastellaan ohjelmaa hieman tarkemmin:

```
String s1 = "eka";
String s2 = new String("eka");
```

Yllä olevilla riveillä luodaan kaksi `String`-luokan ilmentymää. Toinen luodaan niin, että varmasti syntyy uusi olio.

```
System.out.println(s1 == s2); // false
```

Vertailu palauttaa `false`, koska siinä verrataan olioviitteitä, ei niitä olioiden arvoja, joihin olioviitteet viittaavat.

```
System.out.println(s1.equals(s2));  // true
```

Arvoja joihin muuttujat viittaavat, voidaan vertailla `equals`-metodilla kuten yllä.

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

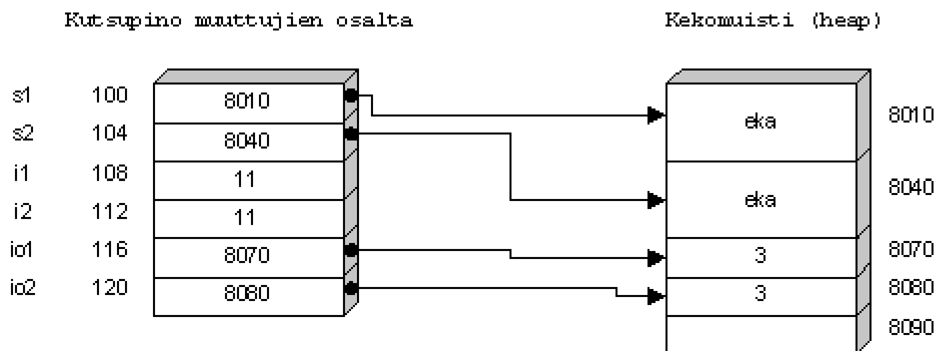
```
       ( i1 == i2 )
```

on totta.

```
Integer io1 = new Integer(3);
Integer io2 = new Integer(3);
System.out.println(io1 == io2);  // false
```

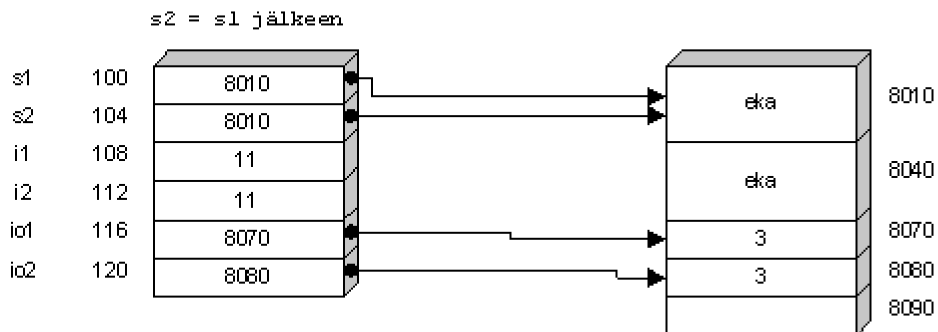
Vastaavasti kuten `String`-olioilla yllä oleva tulostus palauttaa `false`. Jälleen verrataan muuttujien arvoja, eikä arvoja joihin muuttujat viittaavat.

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*. Kutsupino on dynaaminen tietorakenne, johon tallennetaan tietoa aktiivisista aliohjelmista. Siitä käytetään usein myös pelkästään nimeä *pino*. Pinosta puhutaan lisää kurssilla ITKA203 Käyttöjärjestelmät. Tässä vaiheessa pino voisi hieman yksinkertaistettuna olla lokaalien muuttujien kohdalta suurin piirtein seuraavan näköinen:



Kuva 16: Olioviitteet

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



Kuva 17: 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. C++:n automaattiset oliot syntyvät esittelyn yhteydessä ja häviävät aina lohkon loppuessa, joten niiden häviämiseen voidaan liittää tehtäviä, jotka on aina tehtävä. [LAP]

Edellä muistipaikan 8040 olio muuttui roskaksi sijoituksessa `s2 = s1`. Olio voidaan muuttaa roskaksi myös sijoittamalla sen viitemuuttujaan `null`-viite. Tämän takia koodissa pitää usein testata onko olioviite `null` ennen kuin oliota käytetään, jos ei olla varmoja onko viitteen päässä oliota.

```
s2 = null;
...
if ( s2 != null ) System.out.println("s2:n pituus on " + s2.length() );
```

Ilman testiä esimerkissä tulisi `NullPointerException` -poikkeus. Tästä on kerrottu lisää kohdassa 27.3 NullPointerException.

14. Taulukot

Muuttujaan pystytään tallentamaan aina vain yksi arvo kerrallaan. Monesti ohjelmoinnissa kuitenkin tulee tilanteita, joissa meidän tulisi tallentaa useita samantyyppisiä yhteenkuuluvia arvoja. Jos meidän täytyisi tallentaa esimerkiksi kaikkien kuukausien pituudet, voisimme tietenkin tallentaa ne kuten alla:

```
int tammikuu  = 31;
int helmikuu  = 28;
int maaliskuu = 31;
int huhtikuu  = 30;
int toukokuu  = 31;
int kesakuu   = 30;
int heinakuu  = 31;
int elokuu    = 31;
int syyskuu   = 30;
int lokakuu   = 31;
int marraskuu = 30;
int joulukuu  = 31;
```

Kuukausien tapauksessa tämäkin tapa toimisi vielä jotenkin, mutta entäs jos meidän täytyisi tallentaa vaikka Ohjelmointi 1-kurssin opiskelijoiden nimet tai vuoden jokaisen päivän keskilämpötila?

Kun meidän täytyy käsitellä useita samaan asiaan liittyviä fyysisesti samankaltaisia arvoja, on usein syytä ottaa käyttöön *taulukko* (**array**). Taulukko on tietorakenne, johon voi tallentaa useita samantyyppisiä muuttujia. Taulukon koko täytyy määrittää etukäteen, eikä sitä voi myöhemmin muuttaa. Yksittäistä taulukon muuttujaa sanotaan *alkioksi* (**element**). Jokaisella alkiolla on taulukossa paikka, jota sanotaan *indeksiksi* (**index**). Taulukon indeksointi alkaa aina nollasta, eli esimerkiksi 12-alkioisen taulukon ensimmäisen alkion indeksi olisi 0 ja viimeisen 11.

14.1 Taulukon luominen

Javassa taulukon voi luoda sekä alkeistietotyypeille, että oliotietotyypeille, mutta yhteen taulukkoon voi tallentaa aina vain yhtä tietotyyppiä. Taulukon määritteleminen ja luominen tapahtuu yleisessä muodossa seuraavasti:

```
Tietotyyppi[] taulukonNimi;
taulukonNimi = new Tietotyyppi[taulukonKoko];
```

Ensiksi määritellään taulukon tietotyyppi, jonka jälkeen luodaan varsinainen taulukko. Tämän voisi tehdä myös samalla rivillä:

```
Tietotyyppi[] taulukonNimi = new Tietotyyppi[taulukonKoko];
```

Kuukausien päivien lukumäärille voisimme määritellä nyt taulukon seuraavasti:

```
int[] kuukausienPaivienLkm = new int[12];
```

Taulukkoon voi myös sijoittaa arvot määrittelyn yhteydessä. Tällöin sanotaan, että taulukko *alustetaan* (**initialize**). Tällöin varsinaista luontilausetta ei tarvita, sillä taulukon koko määräytyy sijoitettujen arvojen lukumäärän perusteella. Sijoitettavat arvot kirjoitetaan aaltosulkeiden sisään.

```
Tietotyyppi[] = {arvo1, arvo2,...arvoX};
```

Esimerkiksi kuukausien päivien lukumäärille voisimme määritellä ja alustaa taulukon seuraavasti:

```
int[] kuukausienPaivienLkm = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Taulukko voitaisiin nyt kuvata nyt seuraavasti:

indeksi:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
alkio:	31	28	31	30	31	30	31	31	30	31	30	31

Kuva 18: KuukausienPaivienLkm-taulukko

Huomaa, että jokaisella taulukon alkiolla on yksikäsitteinen indeksi. Indeksi tarvitaan, jotta taulukon alkiot voitaisiin myöhemmin "löytää" taulukosta. Jos taulukkoa ei alusteta määrittelyn yhteydessä, alustetaan alkiot automaattisesti oletusarvoihin taulukon luomisen yhteydessä. Tällöin numeeriset arvot alustetaan nollaksi, `boolean`-tyyppi saa arvon `false` ja oliotyypit (esim. String) arvon `null`. [MÄN][KOS]

14.2 Taulukon alkioon viittaaminen

Taulukon alkioihin pääsee käsiksi taulukon nimellä ja indeksillä. Ensiksi kirjoitetaan taulukon nimi, jonka jälkeen hakasulkeiden sisään halutun alkion indeksi. Yleisessä muodossa taulukon alkioihin viitataan seuraavasti:

```
taulukonNimi[indeksi];
```

Taulukkoon viittaamista voidaan käyttää nyt kuten mitä tahansa sen tyyppistä arvoa. Esimerkiksi voisimme tulostaa tammikuun pituuden `kuukausienPaivienLkm`-taulukosta.

```
System.out.println( kuukausienPaivienLkm[0] ); //tulostuu 31
```

Tai tallentaa tammikuun pituuden edelleen muuttujaan:

```
int tammikuu = kuukausienPaivienLkm[0];
```

Taulukkoon viittaava indeksi voi olla myös `int`-tyyppinen muuttuja (eli `long` EI kelpaa). `kuukausienPaivienLkm`-taulukkoon viittaaminen onnistuu yhtä hyvin seuraavasti:

```
int indeksi = 0;
System.out.println( kuukausienPaivienLkm[indeksi] );
```

Taulukon arvoja voi tietenkin myös muuttaa. Jos esimerkiksi olisi kyseessä karkausvuosi, voisimme muuttaa helmikuun pituudeksi 29. Helmikuuhan on taulukon indeksissä 1, sillä indeksointi alkoi nollasta.

```
kuukausienPaivienLkm[1] = 29;
```

Jos viittaamme taulukon alkioon, jota ei ole olemassa, saamme `ArrayIndexOutOfBounds`-poikkeuksen. Tällöin kääntäjä tulostaa seuraavan kaltaisen virheilmoituksen ja ohjelman suoritus päättyy.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
```

Myöhemmin opitaan kuinka poikkeuksista voidaan toipua ja ohjelman suoritusta jatkaa.

14.3 Esimerkki: Arvosana kirjalliseksi

Ehtolauseiden yhteydessä teimme `switch`-rakennetta käyttämällä aliohjelman, joka palautti parametrinaan saamaansa numeroarvosanaa vastaavan kirjallisen arvosanan. Tehdään nyt sama aliohjelma taulukkoa käyttämällä. Kirjalliset arvosanat voidaan nyt tallentaa `String`-tyyppiseen taulukkoon.

```
/**
 * Palauttaa parametrina saamansa numeroarvosanan kirjallisena.
 * @param numero tenttiarvosana numerona
 * @return tenttiarvosana kirjallisena
 */
public static String kirjallinenArvosana(int numero) {
   String[] arvosanat = {"Hylätty", "Välttävä", "Tyydyttävä",
                          "Hyvä", "Kiitettävä", "Erinomainen"};
   if ( numero < 0 || arvosanat.length <= numero ) return "Virheellinen syöte!";
   return arvosanat[numero];
}
```

Ensimmäiseksi aliohjelmassa määritellään ja alustetaan taulukko, jossa on kaikki kirjalliset arvosanat. Taulukko määritellään niin, että taulukon indeksissä 0 on arvosanaa 0 vastaava kirjallinen arvosana, taulukon indeksissä 1 on arvosanaa 1 vastaava kirjallinen arvosana ja niin edelleen. Tällä tavalla tietty taulukon indeksi vastaa suoraan vastaavaa kirjallista arvosanaa. Kirjallisten arvosanojen hakeminen on näin todella nopeaa.

Jos vertaamme tätä tapaa `switch`-rakenteella toteutettuun tapaan huomaamme, että koodin määrä väheni huomattavasti. Tämä tapa on lisäksi nopeampi, sillä jos esimerkiksi hakisimme arvosanalle 5 kirjallista arvosanaa, `switch`-rakenteessa tehtäisiin viisi vertailuoperaatiota. Taulukkoa käyttämällä vertailuoperaatioita ei tehdä yhtään, vaan ainoastaan yksi hakuoperaatio taulukosta.

14.4 Moniulotteiset taulukot

Taulukot voivat olla myös moniulotteisia. Javassa moniulotteiset taulukot ovat oikeastaan yksiulotteisia taulukoita, joiden alkioina on uusia taulukoita. Esimerkiksi kaksiulotteisin `String`-tyyppisen taulukon kurssin opiskelijoiden nimille voisi alustaa seuraavasti.

```
String[][] kurssinOpiskelijat = new String[256][2];
```

Taulukkoon voisi nyt asettaa kurssilaisten nimiä seuraavasti:

```
//ensimmäinen kurssilainen
kurssinOpiskelijat[0][0] = "Virtanen";
kurssinOpiskelijat[0][1] = "Ville";

//toinen kurssilainen
kurssinOpiskelijat[1][0] = "Korhonen";
kurssinOpiskelijat[1][1] = "Kalle";
```

Taulukkoa voisi kuvata kuten alla 2-ulotteisena taulukkona jossa on 256 riviä ja 2 saraketta.

Kuva 19: kurssinOpiskelijat-taulukko

Moniulotteiseen taulukkoon viittaaminen onnistuu vastaavasti kuin yksiulotteiseen. Ulottuvuuksien kasvaessa joudutaan vain antamaan enemmän indeksejä.

```
//tulostaa Ville Virtanen
```

```
System.out.println( kurssinOpiskelijat[0][1] + " " + kurssinOpiskelijat[0][0] );
```

Kun etunimi ja sukunimi on talletettu taulukkoon omille paikoilleen, mahdollistaa se tietojen joustavamman käsittelyn. Nyt opiskelijoiden nimet voidaan halutessa tulostaa muodossa: "etunimi sukunimi" tai muodossa, "sukunimi, etunimi" kuten alla:

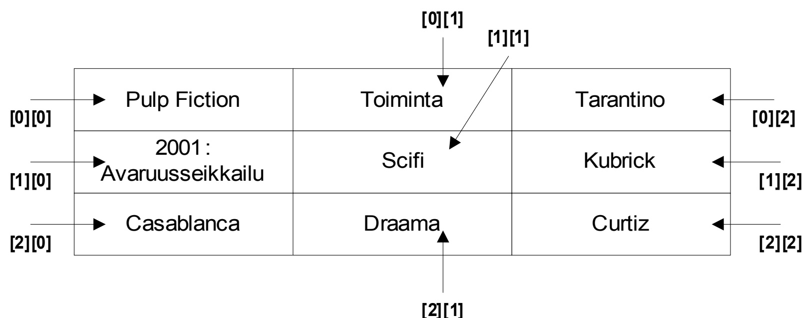
```
//tulostaa Virtanen, Ville
System.out.println( kurssinOpiskelijat[0][0] + ", " + kurssinOpiskelijat[0][1] );
```

Oikeasti esimerkki on kuitenkin huono. Järkevämpää olisi tehdä `Henkilo`-luokka, jossa olisi kentät etunimelle ja sukunimelle ja mahdollisille muille tiedoille. Tästä luokasta luotaisiin sitten jokaiselle opiskelijalle oma olio. Tällä kurssilla ei kuitenkaan tehdä vielä omia olioluokkia.

Moniulotteinen taulukko voidaan määriteltäessä alustaa kuten yksiulotteinenkin. Määritellään ja alustetaan seuraavaksi taulukko elokuville:

```
String[][] elokuvat = { {"Pulp Fiction", "Toiminta", "Tarantino"},
                        {"2001: Avaruusseikkailu", "Scifi", "Kubrick"},
                        {"Casablanca", "Draama", "Curtiz"} };
```

Yllä oleva määrittely luo 3*3 kokoisen taulukon:



Kuva 20: Taulukon elokuvat sisältö.

Alla oleva esimerkki hahmottaa taulukon alkioihin viittaamista.

```
System.out.println(elokuvat[0][0]);  //tulostaa "Pulp Fiction"
System.out.println("Tyyppi: " + elokuvat[0][1]); //tulostaa "Tyyppi:  Toiminta"
System.out.println("Ohjaaja: " + elokuvat[0][2]); //tulostaa "Ohjaaja: Tarantino"
```

Miten tulostat taulukosta Casablanca? Entä Kubrick?

14.5 Taulukon kopioiminen

Myös taulukot ovat olioita. Siispä taulukkomuuttujat ovat viitemuuttujia. Tämän takia taulukon kopioiminen EI onnistu alla olevalla tavalla kuten alkeistietotyypeillä:

```
taulukko1 = {1,2,3,4,5};
taulukko2 = taulukko1;

taulukko2[0] = 10;
System.out.println(taulukko1[0]) //tulostaa 10
```

Yllä olevassa esimerkissä sekä `taulukko1`, että `taulukko2` ovat olioviitteitä ja viittaavat nyt samaan taulukkoon.

Taulukon kopioiminen onnistuu muun muassa `clone`-metodilla.

```
int[] taulukko = {1,2,3,4,5};
```

```
//clone-metodi luo identtisen kopion taulukosta
int[] kopio_taulukosta = taulukko.clone();
```

Nyt meillä olisi identtinen kopio taulukosta, jonka muuttaminen ei siis vaikuttaisi alkuperäiseen taulukkoon.

14.6 Taulukot parametreina

Taulukoita voi käyttää ja ne ovat usein myös erittäin näppäriä aliohjelmien ja metodien parametreina. Esimerkiksi kuvioita piirrettäessä monilla olioilla on parametrina taulukko, jolla kerrotaan piirrettävän kuvion pisteet.

Esimerkkinä voidaan piirtää neliö käyttämällä Jyväskylän yliopiston Graphics-kirjastoa. Monikulmioita voidaan piirtää mm. `EasyWindow`-luokan `addPolygon`-metodilla. Metodille voi antaa monikulmion pisteet, joko kaksiulotteisena taulukkona tai kahtena yksiulotteisena taulukkona, joista toisessa on pisteiden x-arvot ja toisessa y-arvot. Käytetään tässä ensimmäistä tapaa. Lisää tietoa metodista löydät Graphics-kirjaston dokumentaatiosta:

http://users.jyu.fi/~vesal/kurssit/ohj1/graphics/

```
import fi.jyu.mit.graphics.EasyWindow;

public class Kuvioita {

   public static void main(String args[]) {
      EasyWindow window = new EasyWindow();
      double[][] pisteet = {{0.0,0.0},{50.0,0},{50.0,50.0},{0.0,50.0}};
      window.addPolygon(pisteet);
      window.showWindow();
   }
}
```

Seuraavassa luvussa tutustutaan taulukoiden käsittelyä huomattavasti helpottaviin rakenteisiin, silmukoihin.

15. Toistorakenteet (silmukat)

Ohjelmoinnissa tulee usein tilanteita, joissa samaa tai lähes samaa asiaa täytyy toistaa ohjelmassa useampia kertoja. Varsinkin taulukoiden käsittelyssä tällainen asia tulee usein eteen. Jos haluaisimme esimerkiksi tulostaa kaikki edellisessä luvussa tekemämme `kuukausienPaivienLkm`-taulukon luvut, onnistuisi se tietenkin seuraavasti:

```
System.out.println(kuukausienPaivienLkm[0]);
System.out.println(kuukausienPaivienLkm[1]);
System.out.println(kuukausienPaivienLkm[2]);
System.out.println(kuukausienPaivienLkm[3]);
System.out.println(kuukausienPaivienLkm[4]);
System.out.println(kuukausienPaivienLkm[5]);
System.out.println(kuukausienPaivienLkm[6]);
System.out.println(kuukausienPaivienLkm[7]);
System.out.println(kuukausienPaivienLkm[8]);
System.out.println(kuukausienPaivienLkm[9]);
System.out.println(kuukausienPaivienLkm[10]);
System.out.println(kuukausienPaivienLkm[11]);
```

Tuntuu kuitenkin tyhmältä toistaa lähes samanlaista koodia useaan kertaan. Tällöin on järkevämpää käyttää jotain toistorakennetta. Toistorakenteet soveltuvat erinomaisesti taulukoiden käsittelyyn, mutta niistä on myös moniin muihin tarkoituksiin. Toistorakenteista käytetään usein myös nimitystä *silmukat* (**loop**).

15.1 Idea

Ideana toistorakenteissa on, että toistamme tiettyä asiaa niin kauan kuin joku ehto on voimassa. Esimerkki ihmiselle suunnatusta toistorakenteesta aamupuuron syöntiin:

```
Syö aamupuuroa niin kauan, kun puuroa on lautasella.
```

Yllä olevassa esimerkissä on kaikki toistorakenteeseen vaadittavat elementit. Toimenpiteet mitä tehdään: "Syö aamupuuroa.", sekä ehto kuinka toistetaan: "niin kauan kuin puuroa on lautasella". Toinen esimerkki toistorakenteesta voisi olla seuraava:

```
Tulosta kuukausienPaivienLkm-taulukon kaikki luvut.
```

Myös yllä oleva lause sisältää toistorakenteen elementit, vaikka ne onkin hieman vaikeampi tunnistaa. Toimenpiteenä tulostetaan `kuukausienPaivienLkm`-taulukon lukuja ja ehdoksi voisi muotoilla: "kunnes kaikki luvut on tulostettu". Lauseen voisikin muuttaa muotoon:

```
Tulosta kuukausienPaivienLkm-taulukon lukuja, kunnes kaikki luvut on tulostettu.
```

Javassa on kolmen tyyppisiä toistorakenteita:

- `for`-silmukka
- `while`-silmukka
- `do-while`-silmukka

15.2 while-silmukka

`while`-silmukka on yleisessä muodossa seuraava:

```
while (ehto) lause;
```

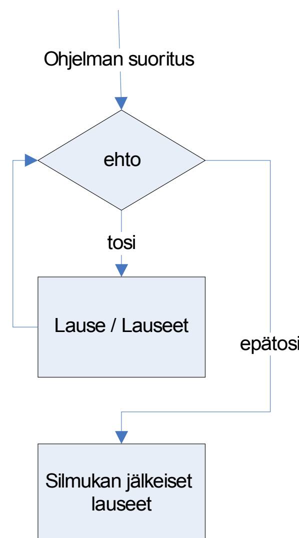
Kuten ehtolauseissa, täytyy ehdon taas olla joku lauseke, joka saa joko arvon `true` tai `false`. Usein ehdon jälkeen tulee kuitenkin yksittäisen lauseen sijaan lohko, sillä yhdellä lauseella ei useinkaan saa tehtyä muuta kuin *ikuisen silmukan* (**infinite loop, never ending loop**).

```
while (ehto) {
  lause1;
  lause2;
  lauseX;
}
```

ikuinen silmukka = Silmukka joka ei pääty koskaan. Ikuinen silmukka johtuu siitä, että silmukan ehto ei saa koskaan arvoa `false`. Useimmiten ikuinen silmukka on ohjelmointivirhe, mutta silmukka voidaan kyllä joskus asettaa aluksi jatkumaan ikuisesti, jolloin silmukasta kuitenkin poistutaan jollain ehdolla `break`-lauseen avulla. Tällöinhän silmukka ei oikeastaan ole ikuinen, vaikka tällaisesta silmukasta sitä nimitystä usein käytetäänkin. `break`-lauseesta puhutaan tässä luvussa myöhemmin kohdassa 15.6.1 break.

Silmukan lauseita toistetaan niin kauan, kuin ehto on voimassa, eli sen arvo on `true`. Ehto tarkastetaan aina ennen kuin siirrytään seuraavalle kierrokselle. Jos ehto saa siis heti alussa arvon `false`, ei lauseita suoriteta kertaakaan.

`while`-silmukan voisi esittää vuokaaviona seuraavasti:



Kuva 21: while-silmukka vuokaaviona

15.2.1 Esimerkki: Taulukon tulostaminen

Tehdään aliohjelma joka tulostaa `int`-tyyppisen yksiulotteisen taulukon sisällön.

```
public class Silmukat {

   /**
    * Tulostaa int-tyyppisen taulukon sisällön.
    *
    * @param taulukko tulostettava taulukko
    */
   public static void tulostaTaulukko(int[] taulukko) {
      int i = 0;
      while (i < taulukko.length){
         System.out.print(taulukko[i] + " ");
         i++;
      }
   }

   public static void main(String args[]) {
      int[] kuukausienPaivienLkm = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      tulostaTaulukko(kuukausienPaivienLkm);
   }
}
```

Tarkastellaan `tulostaTaulukko`-aliohjelman sisältöä hieman tarkemmin.

```
int i = 0;
```

Tässä luodaan uusi muuttaja, jolla kontrolloidaan mitä taulukon alkioita ollaan tulostamassa ja milloin taulukon kaikki alkiot on tulostettu. Muuttuja alustetaan arvoon `0`, sillä taulukon ensimmäinen alkio on aina indeksissä 0. Muuttajalle annetaan nimeksi "`i`". Useimmiten pelkät kirjaimet ovat huonoja muuttujan nimiä, koska ne kuvaavat muuttujaa huonosti. Silmukoissa kuitenkin muuttujan nimi "`i`" on vakiinnuttanut asemansa kontrolloimassa silmukoiden kierroksia, joten sitä voidaan hyvällä omalla tunnolla käyttää.

```
while (i < taulukko.length) {
```

Yllä olevalla aliohjelman toisella rivillä aloitetaan `while`-silmukka. Ehtona, että silmukkaa suoritetaan niin kauan kuin muuttujan `i` arvo on pienempi kuin taulukon pituus. Taulukon pituus saadaan aina selville kirjoittamalla nimen perään `.length`. Huomionarvoinen seikka on, että `length`-sanan perään ei tule sulkuja, sillä se ei ole metodi vaan attribuutti.

```
System.out.print(taulukko[i] + " ");
```

Ensimmäisessä silmukan lauseessa tulostetaan taulukon alkio indeksissä `i`. Perään tulostetaan välilyönti erottamaan eri alkiot toisistaan. `System.out.println`-metodin sijaan käytämme nyt toista `System.out`-luokan metodia. `System.out.print`-metodi ei tulosta perään rivinvaihtoa, joten sillä voidaan tulostaa taulukon alkiot peräkkäin.

```
i++;
```

Silmukan viimeinen lause kasvattaa muuttujan `i` arvoa yhdellä. Ilman tätä lausetta saisimme aikaan ikuisen silmukan, sillä indeksin arvo olisi koko ajan `0` ja silmukan ehto olisi aina tosi. Lisäksi metodi tulostaisi koko ajan taulukon ensimmäistä alkioita.

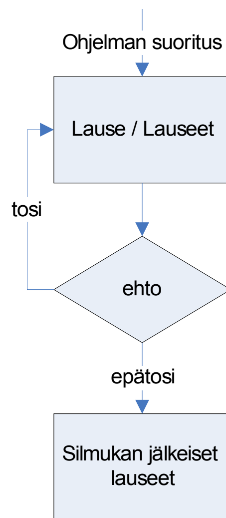
Taulukon tulostaminen olisi järkevämpi tehdä `for`-silmukalla. Tässä onkin tarkoituksena vain demonstroida, kuinka `while`-silmukka toimii, eikä olla erityisen hyvä esimerkki `while`-silmukan käytöstä. `while`-silmukkaa tulisikin käyttää silloin, kun meillä ei ole tarkkaa tietoa silmukan suorituskierrosten lukumäärästä. Myöhemmin löytyy vielä järkevämpää käyttöä `while`-silmukalle.

15.3 do-while -silmukka

`do-while` -silmukka eroaa `while`-silmukasta siinä, että `do-while` silmukassa ilmoitetaan ensiksi lauseet (mitä tehdään) ja vasta sen jälkeen ehto (kauanko tehdään). Tämän takia `do-while` -silmukka suoritetaankin joka kerta vähintään yhden kerran. Se soveltuukin parhaiten tilanteisiin, joissa joku asia on suoritettava vähintään yhden kerran riippumatta ehdosta. Yleisessä muodossa `do-while` -silmukka on seuraavanlainen:

```
do {
  lause1;
  lause2;
  lauseX;
} while(ehto);
```

Vuokaaviona `do-while` -silmukan voisi esittää seuraavasti:



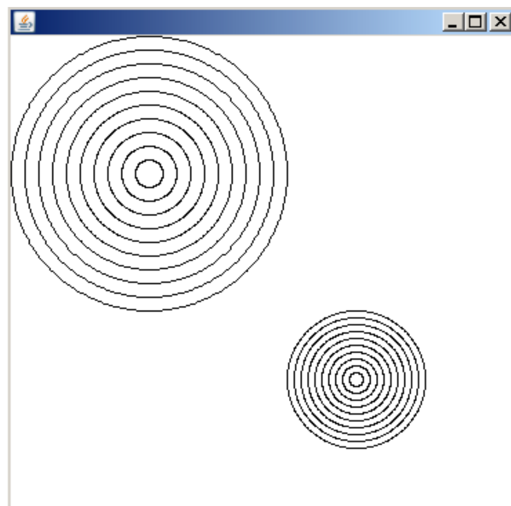
Kuva 22: do-while -silmukka vuokaaviona

15.3.1 Esimerkki: Tikkataulu

Tehdään aliohjelma, joka piirtää halutun kokoisen tikkataulun haluttuun paikkaan. Käytetään piirtämiseen JY:n Graphics-kirjaston `EasyWindow`-luokkaa.

```
import fi.jyu.mit.graphics.EasyWindow;

public class Tikkataulu {

  /**
   * Piirtää haluamamme kokoisen tikkataulun haluamaamme  paikkaan.
   * @param window ikkuna johon piirretään
   * @param x keskipisteen x-koordinaatti
   * @param y keskipisteen y-koordinaatti
   * @param sade tikkataulun säde
   */
  public static void piirraTikkataulu(EasyWindow window, int x, int y, int sade) {

    int rinkuloidenLeveys = sade/10;
    do {
      window.addCircle(x,y, sade);
      sade -= rinkuloidenLeveys;
    } while ( 0 < sade );
  }

  public static void main(String[] args) {
    EasyWindow window = new EasyWindow();
    piirraTikkataulu(window,100,100,100);
    piirraTikkataulu(window,300,300,50);
    window.showWindow();
  }
}
```

Ajettaessa koodin tulisi piirtää ikkunaan kaksi tikkataulua kuten seuraavassa kuvassa.



Kuva 23: Pari tikkataulua

Tutkitaan tarkemmin `piirraTikkataulu`-aliohjelmaa Aliohjelma ei palauta mitään, siis paluuarvon tyyppinä on `void`. Parametreina on `EasyWindow`-ikkuna johon kuviot piirretään, tikkataulun keskipisteen x- ja y-koordinaatit, sekä tikkataulun säde.

```
int rinkuloidenLeveys = sade/10;
```

Ensimmäisenä aliohjelmassa lasketaan kahden renkaan välinen etäisyys. Koska tikkataulussa on perinteisesti 10 rengasta, saadaan renkaiden leveys jakamalla tikkataulun säde luvulla 10. Mennään seuraavaksi `do-while` -silmukan `do`-osaan.

```
window.addCircle(x, y, sade);
```

Tikkataulu koostuu kymmenestä ympyrästä, niinpä ensimmäisenä `do`-osassa lisätään ikkunaan ympyrä. Ympyrän keskipiste pysyy jokaisella kierroksella samana, vain säde muuttuu. Ensimmäisenä piirretään uloin ympyrä. Niinpä ensimmäisen ympyrän säde on suoraan tikkataulun säde.

```
sade -= rinkuloidenLeveys;
```

Kierroksen lopuksi vähennetään `sade`-muuttujasta aliohjelman alussa laskettu renkaiden välinen etäisyys. Näin emme piirrä aina samankokoista ympyrää.

```
} while ( 0 < sade );
```

Lopuksi `do-while` -silmukassa määritellään ehto, kuinka kauan silmukan suorittamista jatketaan. Tässä tapauksessa ehto on, että niin kauan kuin säde on suurempi kuin 0. Koska tikkataulu koostuu aina kymmenestä ympyrästä, voitaisiin yhtä hyvin ennen silmukkaa määritellä joku laskurimuuttuja ja kirjoittaa ehdoksi, että ympyröitä piirretään niin kauan, kun laskuri on pienempi kuin 10 (`while(laskuri < 10)`). Tässä tapauksessa selvitään kuitenkin tyylikkäästi myös ilman laskuria.

15.4 for-silmukka

Kun silmukan suoritusten lukumäärä on ennalta tiedossa, on järkevintä käyttää `for`-silmukkaa. Esimerkiksi taulukoiden käsittelyyn `for`-silmukka on yleensä paras vaihtoehto. Syntaksiltaan `for`-silmukka eroaa selvästi edellisistä. Perinteinen `for`-silmukka on yleisessä muodossa seuraavanlainen:

```
for (muuttujien alustukset; ehto; silmukan lopussa tehtävät toimenpiteet){
   lauseet;
```

```
}
```

Silmukan `for`-rivi eli *kontrollilauseke* sisältää kolme operaatiota, jotka on erotettu toisistaan puolipisteellä.

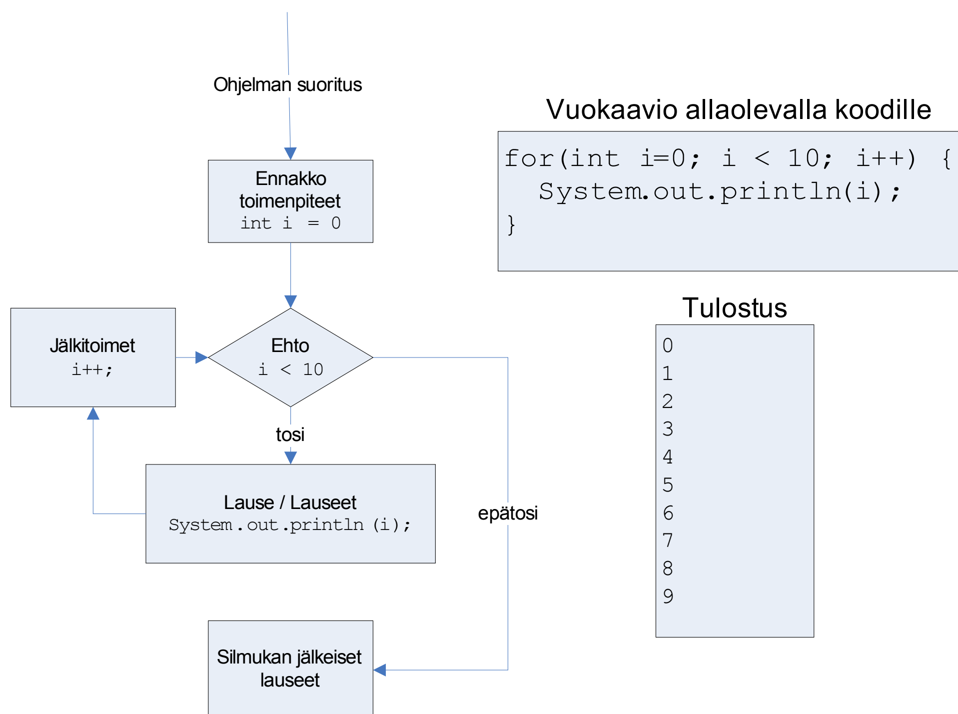
- Muuttujien alustukset: Useimmiten alustetaan vain yksi muuttuja, mutta myös useampien muuttujien alustaminen on mahdollista.
- Ehto: Kuten muissakin silmukoissa, lauseita toistetaan niin kauan kuin ehto on voimassa.
- Silmukan lopussa tehtävät toimenpiteet: Useimmiten muuttujan tai muuttujien arvoa kasvatetaan yhdellä, mutta myös suuremmalla määrällä kasvattaminen on mahdollista.

Yksinkertaisimmillaan `for`-silmukka on alla olevan kaltainen. Siinä tulostetaan 10 kertaa ”Hello World!”.

```
for (int i=1; i <= 10; i++) {
   System.out.println("Hello World!");
}
```

Kontrollilausekkeessa alustetaan aluksi muuttujan `i` arvoksi 1. Seuraavaksi ehtona on, että silmukan suoritusta jatketaan niin kauan kuin muuttujan `i` arvo on pienempää tai yhtä suurta kuin luku 10. Lopuksi ilmoitetaan, että muuttujan `i` arvoa kasvatetaan joka kierroksella yhdellä.

Vuokaaviona `for`-silmukan voisi kuvata alla olevalla tavalla.



Kuva 24: Vuokaavio for-silmukalle

15.4.1 Esimerkki: keskiarvo-aliohjelma

Muuttujien yhteydessä teimme aliohjelman, joka laski kahden luvun keskiarvon. Tällainen aliohjelma ei ole kovin hyödyllinen, sillä jos haluaisimme laskea kolmen tai neljän luvun keskiarvon, täytyisi meidän tehdä niillä omat aliohjelmat. Sen sijaan jos annamme luvut taulukossa, pärjäämme yhdellä aliohjelmalla. Tehdään siis nyt aliohjelma, joka laskee taulukossa olevien kokonaislukujen keskiarvon.

```
public class Silmukat {
   /**
    * Palauttaa parametrina saamansa int-taulukon
    * alkoiden keskiarvon.
    * @param luvut summattavat luvut
    * @return lukujen summa
    */
   public static double keskiarvo(int[] luvut) {
      double summa = 0;
      for (int i = 0; i < luvut.length; i++) {
         summa += luvut[i];
      }
      return summa/luvut.length;
   }
}
```

Ohjelmassa lasketaan ensiksi kaikkien taulukoiden lukujen summa muuttujaan `summa`. Koska taulukoiden indeksointi alkaa nollasta, täytyy myös laskurimuuttuja `i` asettaa aluksi arvoon `0`. Ehtona on, että silmukkaa suoritetaan niin kauan kuin muuttuja `i` on pienempi kuin taulukon pituus. Jos tuntuu, että ehdossa pitäisi olla yhtä suuri tai pienempi kuin -merkki (<=), niin pohdi seuraavaa. Jos taulukon koko olisi vaikka 7, niin tällöin viimeinen alkio olisi alkiossa `luvut[6]`, koska indeksointi alkaa nollasta. Tästä johtuen jos ehdossa olisi "<="-merkki, viitattaisiin viimeisenä taulukon alkioon `luvut[7]`, joka ei enää kuulu taulukon muistialueeseen. Tällöin ohjelma kaatuisi ja saisimme "ArrayIndexOutOfBoundsException"-poikkeuksen.

```
return summa/luvut.length;
```

Aliohjelman lopussa palautetaan lukujen summa jaettuna lukujen määrällä, eli taulukon pituudella.

15.4.2 Esimerkki: Taulukon kääntäminen käänteiseen järjestykseen

Kontrollirakenteen ensimmäisessä osassa voidaan siis alustaa myös useita muuttujia. Klassinen esimerkki tällaisesta tapauksesta on taulukon alkioiden kääntäminen päinvastaiseen järjestykseen.

Tehdään aliohjelma joka saa parametrina `int`-tyyppisen taulukon ja palauttaa taulukon käänteisessä järjestyksessä.

```
/**
 * Esitellään Javan silmukoita.
 *
 * @author martti
 * @version 17.8.2009
 */
public class Silmukat {

   /**
    * Aliohjelma kääntää kokonaisluku-taulukon alkiot päinvastaiseen
    * järjestykseen.
    *
    * @example
    * <pre name="test">
    * #import java.util.Arrays; //Täytyy importata, että testi toimii
    * int[] testiluvut1 = {1, 2, 3, 4, 5, 6};
    * int[] vertailuluvut1 = {6, 5, 4, 3, 2, 1};
    * int[] testiluvut2 = {1, 2, 3, 4, 5};
    * int[] vertailuluvut2 = {5, 4, 3, 2, 1};
    *
    * Arrays.equals(kaannaTaulukko(testiluvut1), vertailuluvut1) === true;
    * Arrays.equals(kaannaTaulukko(testiluvut2), vertailuluvut2) === true;
    *
    * </pre>
    *
    * @param taulukko käännettävä taulukko, johon myös tulos tulee
    * @return viite parametrina tuotuun taulukkoon
    */
   public static int[] kaannaTaulukko(int[] taulukko) {
      int temp = 0;
      for (int vasen = 0, oikea = taulukko.length-1; vasen < oikea; vasen++, oikea--) {
         temp = taulukko[vasen];
         taulukko[vasen] = taulukko[oikea];
```

```
            taulukko[oikea] = temp;
        }
        return taulukko;
    }
}
```

Ideana yllä olevassa aliohjelmassa on, että meillä on kaksi muuttujaa. Muuttujia voisi kuvata kuvainnollisesti osoittimiksi. Osoittimista toinen osoittaa aluksi taulukon alkuun ja toinen taulukon loppuun. Oikeasti osoittimet ovat `int`-tyyppisiä muuttujia, jotka saavat arvokseen taulukon indeksejä. Taulukon alkuun osoittavan muuttujan nimi on "`vasen`" ja taulukon loppuun osoittavan muuttujan nimi on "`oikea`". Vasenta osoitinta liikutetaan taulukon alusta loppuun päin ja oikeaa taulukon lopusta alkuun päin. Jokaisella kierroksella vaihdetaan niiden taulukon alkioiden paikat keskenään, joihin osoittimet osoittavat. Silmukan suoritus lopetetaan juuri ennen kuin osoittimet kohtaavat toisensa.

Tarkastellaan aliohjelmaa nyt hieman tarkemmin.

```
int temp = 0;
```

Ensimmäiseksi metodissa on alustettu `temp`-niminen muuttuja. Tätä tarvitaan, jotta taulukon alkioiden paikkojen vaihtaminen onnistuisi.

```
for(int vasen = 0, oikea = taulukko.length-1; vasen < oikea; vasen++, oikea--) {
```

Kontrollirakenteessa alustetaan ja päivitetään nyt kahta eri muuttujaa. Muuttujat erotetaan toisistaan pilkulla. Huomaa, että muuttujan tyyppi kirjoitetaan vain yhden kerran! Ehtona on, että suoritusta jatketaan niin kauan kuin muuttuja `vasen` on pienempää kuin muuttuja `oikea`. Lopuksi päivitetään vielä muuttujien arvoja. Eri muuttujien päivitykset erotetaan toisistaan jälleen pilkulla. Muuttujaa vasen kasvatetaan joka kierroksella yhdellä kun taas muuttujaa oikea sen sijaa vähennetään.

```
temp = taulukko[vasen];
```

Seuraavaksi laitetaan vasemman osoittimen osoittama alkio väliaikaiseen säilytykseen `temp`-muuttujaan.

```
taulukko[vasen] = taulukko[oikea];
```

Nyt voimme tallentaa oikean osoittimen osoittaman alkion vasemman osoittimen osoittaman alkion paikalle.

```
taulukko[oikea] = temp;
```

Yllä olevalla lauseella asetetaan vielä `temp`-muuttujaan talletettu arvo oikean osoittimen osoittamaan alkioon. Nyt vaihto on suoritettu onnistuneesti.

```
return taulukko;
```

Lopuksi `for`-silmukan jälkeen palautetaan vielä viite parametrina tuotuun taulukkoon, jonka pitäisi olla nyt käänteisessä järjestyksessä. Tässä funktiolla oli sivuvaikutus, eli se muutti parametrina vietyä taulukkoa. Jos haluttaisiin alkuperäisen taulukon säilyvän, pitäisi funktion alussa luoda uusi taulukko tulosta varten, sijoittaa arvot käänteisessä järjestyksessä ja lopuksi palauttaa viite uuteen taulukkoon.

Tee funktiosta `kaannaTaulukko` sivuvaikutukseton versio.

15.4.3 Esimerkki: Arvosanan laskeminen taulukoilla

Ehtolauseita käsiteltäessä tehtiin aliohjelma, joka laski tenttiarvosanan. Aliohjelma sai parametreina

tentin maksimipisteet, läpipääsyrajan ja opiskelijan tenttipisteet ja palautti opiskelijan arvosanan. Tehdään nyt vastaava ohjelma käyttämällä taulukoita.

```
public class Arvosana {

   /**
    * Laskee opiskelijan tenttiarvosanan asteikoilla 0-5.
    *
    * @param maksimipisteet tentin maksimipisteet
    * @param lapipaasyraja tentin läpipääsyraja
    * @param tenttipisteet opiskelijan tenttipisteet
    * @return opiskelijan tenttiarvosana
    */
   public static int laskeArvosana(int maksimipisteet,
                                   int lapipaasyraja, int tenttipisteet) {
      int[] arvosanaRajat = new int[6];
      int arvosanojenPisteErot = (maksimipisteet - lapipaasyraja) / 5;

      //Arvosanan 1 rajaksi tentin läpipääsyraja
      arvosanaRajat[1] = lapipaasyraja;

      //Asetetaan taulukkoon jokaisen arvosanan raja
      for (int i = 2; i <= 5; i++) {
         arvosanaRajat[i] = arvosanaRajat[i-1] + arvosanojenPisteErot;
      }

      //Katsotaan mihin arvosanaan tenttipisteet riittävät
      for (int i = 5; 1 <= i; i--) {
         if ( arvosanaRajat[i] <= tenttipisteet ) return i;
      }
      return 0;
   }


   public static void main(String[] args) {
      System.out.println(laskeArvosana(24, 12, 19)); //tulostaa 4
      System.out.println(laskeArvosana(24, 12, 11)); //tulostaa 0
   }
}
```

Aliohjelman idea on, että jokaisen arvosanan raja tallennetaan taulukkoon. Kun taulukkoa sitten käydään läpi lopusta alkuun päin, voidaan kokeilla mihin arvosanaan opiskelijan pisteet riittävät.

```
int[] arvosanaRajat = new int[6];
```

Aliohjelman alussa alustetaan tenttiarvosanojen pisterajoille kuuden alkion kokoinen taulukko. Taulukko alustetaan kuuden kokoiseksi, jotta voisimme tallentaa jokaisen arvosanan pisterajan vastaavan taulukon indeksin kohdalle. Arvosanan 1 pisteraja on taulukon indeksissä 1 ja arvosanan 2 indeksissä 2 jne. Näin taulukon ensimmäinen indeksi jää käyttämättä, mutta taulukkoon viittaaminen on selkeämpää.

```
int arvosanojenPisteErot = (maksimipisteet - lapipaasyraja) / 5;
```

Yllä oleva rivi laskee arvosanojen välisen piste-eron.

```
arvosanaRajat[1] = lapipaasyraja;
```

Tällä rivillä asetetaan arvosanan 1 rajaksi tentin läpipääsyraja.

```
 for (int i = 2; i <= 5; i++) {
      arvosanaRajat[i] = arvosanaRajat[i-1] + arvosanojenPisteErot;
 }
```

Yllä oleva silmukka laskee arvosanojen 2-5 pisterajat. Seuraava pisteraja saadaan lisäämällä edelliseen arvosanojen välinen piste-ero.

```
   for (int i = 5; 1 <= i; i--) {
      if ( arvosanaRajat[i] <= tenttipisteet ) return i;
   }
```

Tällä silmukalla sen sijaan katsotaan mihin arvosanaan opiskelijan tenttipisteet riittävät. Arvosanoja aletaan käydä läpi lopusta alkuun päin. Tämän takia muuttujan `i` arvo asetetaan aluksi arvoon 5 ja joka kierroksella sitä pienennetään yhdellä. Kun oikea arvosana on löytynyt, palautetaan tenttiarvosana (eli taulukon indeksi) välittömästi, ettei käydä taulukon alkioita turhaan läpi.

Pääohjelmassa ohjelmaa on testattu muutamilla testitulostuksilla.

Jos laskisimme useiden oppilaiden tenttiarvosanoja, niin aliohjelmamme laskisi myös `arvosanaRajat`-taulukon arvot jokaisella kerralla erikseen. Tämä on melko typerää tietokoneen resurssien tuhlausta. Meidän kannattaakin tehdä oma aliohjelma siitä osasta, joka laskee tenttiarvosanojen rajat. Tämä aliohjelma voisi palauttaa arvosanojen rajat suoraan taulukossa. Nyt voisimme muuttaa `laskeArvosana`-aliohjelmaa niin, että se saa parametrikseen arvosanojen rajat taulukossa ja opiskelijan tenttipisteet.

```
public class Arvosanat {

   /**
    * Laskee tenttiarvosanojen pisterajat taulukkoon.
    *
    * @param maksimiPisteet tentin maksimipisteet
    * @param lapiPaasyRaja tentin läpipääsyraja
    * @return arvosanojen pisterajat taulukossa
    */
   public static int[] laskeRajat(int maksimiPisteet, int lapiPaasyRaja) {
      int[] arvosanaRajat = new int[6];
      int arvosanojenPisteErot = (maksimiPisteet - lapiPaasyRaja) / 5;

      arvosanaRajat[1] = lapiPaasyRaja;

      //Asetetaan taulukkoon jokaisen arvosanan raja
      for (int i = 2; i <= 5; i++) {
         arvosanaRajat[i] = arvosanaRajat[i-1] + arvosanojenPisteErot;
      }
      return arvosanaRajat;
   }

   /**
    * Laskee opiskelijan tenttiarvosanan asteikoilla 0-5.
    *
    * @param arvosanaRajat arvosanojen rajat taulukossa.
    *  Arvosanan 1 raja taulukon indeksissä 1 jne.
    * @param tenttiPisteet
    * @return tenttiarvosana välillä 0-5
    */
   public static int laskeArvosana(int arvosanaRajat[],
                         int tenttiPisteet) {
      for (int i = 5; 1 <= i; i--) {
         if ( arvosanaRajat[i] <= tenttiPisteet ) return i;
      }
      return 0;
   }

   /**
    * Pääohjelmassa testataan aliohjelmia.
    * @param args ei käytössä
    */
   public static void main(String[] args) {
      pisterajat = laskeRajat(24,12);
      System.out.println(laskeArvosana(pisterajat,12)); //tulostaa 1
      System.out.println(laskeArvosana(pisterajat,20)); //tulostaa 5
      System.out.println(laskeArvosana(pisterajat,11)); //tulostaa 0
   }
}
```

Yllä olevassa esimerkissä lasketaan nyt arvosanarajat vain kertaalleen taulukkoon ja samaa taulukkoa käytetään nyt eri arvosanojen laskemiseen. Yhden aliohjelman kuuluisikin aina suorittaa vain yksi tehtävä tai toimenpide. Näin aliohjelman koko ei kasva mielettömyyksiin. Lisäksi mahdollisuus, että pystymme hyödyntämään aliohjelmaa joskus myöhemmin toisessa ohjelmassa lisääntyy.

15.5 For-each -silmukka

Taulukoita käsiteltäessä voidaan käyttää myös for-each -silmukkaa. Se on eräänlainen paranneltu versio `for`-silmukasta. Joskus sitä kutsutaan myös "uudeksi for -silmukaksi", sillä se on tullut Java-kieleen `for`-silmukan jälkeen. Nimensä mukaan se käy läpi kaikki taulukon alkiot. Se on syntaksiltaan selkeämpi silloin, kun haluamme tehdä jotain jokaiselle taulukon alkiolle. Sen syntaksi on yleisessä muodossa seuraava:

```
for (taulukonAlkionTyyppi alkio : taulukko) {
   lauseet;
}
```

Nyt `for`-silmukan kontrollilausekkeessa ilmoitetaan vain kaksi asiaa. Ensiksi annetaan tyyppi ja nimi muuttujalle, joka viittaa yksittäiseen taulukon alkioon. Tyypin täytyy olla sama kuin käsiteltävän taulukon alkiotyyppi, mutta nimen saa itse keksiä. Tälle muuttujalle tehdään ne toimenpiteet, mitä jokaiselle taulukon alkiolle halutaan tehdä. Toisena tietona for-each -silmukalle pitää antaa sen taulukon nimi, mitä halutaan käsitellä. Huomaa, että tiedot erotetaan for-each -silmukassa kaksoispisteellä. Tämä erottaa for-each –silmukan `for`-silmukasta. Esimerkiksi `kuukausienPaivienLkm`-taulukon alkioita voisi nyt tulostaa seuraavasti:

```
for (int kuukausi : kuukausienPaivienLkm) {
   System.out.print(kuukausi + " ");
}
```

Yllä oleva for-each -silmukka voitaisiin lukea seuraavasti: "For each kuukausi in kuukausienPaivienLkm...". Vapaasti suomennettuna: "Jokaiselle kuukaudelle kuukausienPaivienLkm-taulukossa...".

15.5.1 Esimerkki: Sisäkkäiset silmukat

Kaikkia silmukoita voi kirjoittaa myös toisten silmukoiden sisälle. Sisäkkäisiä silmukoita tarvitaan ainakin silloin, kun halutaan tehdä jotain moniulotteisille taulukoille. Luvussa 14.4 Moniulotteiset taulukot määrittelimme kaksiulotteisen taulukon elokuvien tallentamista varten. Tulostetaan nyt sen sisältö käyttämällä kahta for-each -silmukkaa.

```
String[][] elokuvat = { {"Pulp Fiction", "Toiminta", "Tarantino"},
                        {"2001: Avaruusseikkailu", "Scifi", "Kubrick",
                        {"Casablanca", "Draama", "Curtiz"} };

for (String[] elokuva : elokuvat) {
   for (String tieto : elokuva) {
      System.out.print(tieto + "|");
   }
   System.out.println();
}
```

Moniulotteinen taulukkohan oli vain yksiulotteinen taulukko, jonka alkiot olivat taulukoita. Esimerkissä oleva `elokuvat`-taulukon alkioina on siis kolme taulukkoa, jotka taas sisältävät tietoja elokuvista. Niinpä ulommassa `for`-silmukassa käydään läpi kaikki `elokuvat`-taulukon sisältämät taulukot. Siksi alkion tyypiksi täytyy muistaa laittaa `String[]`. Sisemmässä `for`-silmukassa sen sijaan käydään läpi aina kaikki yhden elokuvan tiedot. Käsiteltäväksi taulukoksi laitetaan nyt ulommassa taulukossa määritelty `elokuva`-niminen muuttuja. Tietyn elokuvan eri tiedot tai kentät on tässä päätetty erottaa "|"-merkillä. Sisemmän `for`-silmukan jälkeen tulostetaan vielä rivinvaihto `System.out.println`-metodilla. Näin eri elokuvat saadaan eri riveille.

Jos käyttäisimme jotain muuta silmukkarakennetta, täytyisi meidän ottaa huomioon, että ulomman taulukon indeksejä käydään läpi eri muuttujalla kuin sisempää taulukkoa. Yleensä kaksiulotteisissa taulukoissa otetaan toiseksi muuttujaksi `j`. Eclipse kuitenkin varottaisi meitä, jos samalla näkyvyysalueella olisi kaksi samannimistä muuttujaa.

15.6 Silmukan suorituksen kontrollointi break- ja continue-lauseilla

Silmukoiden normaalia toimintaa voidaan muuttaa `break`- ja `continue`-lauseilla. Niiden käyttäminen ei ole suositeltavaa, vaan silmukat pitäisi ensisijaisesti suunnitella niin, ettei niitä tarvittaisi.

15.6.1 break

`break`-lauseella hypätään välittömästi pois silmukasta ja ohjelman suoritus jatkuu silmukan jälkeen.

```
int laskuri = 0;
while (true) {
   if ( laskuri = 10 ) break;
   System.out.println("Hello world!");
   laskuri++;
}
```

Yllä olevassa ohjelmassa muodostetaan ikuinen silmukka asettamalla `while`-silmukan ehdoksi `true`. Tällöin ohjelman suoritus jatkuisi loputtomiin ilman `break`-lausetta. Nyt `break`-lause suoritetaan, kun laskuri saa arvon 10. Tämä rakennehan on täysin järjetön, sillä `if`-lauseen ehdon voisi asettaa käänteisenä `while`-lauseen ehdoksi ja ohjelma toimisi täysin samanlailla. Useimmiten `break`-lauseen käytön voikin välttää.

```
int laskuri = 0;
while (laskuri != 10) {
   System.out.println("Hello world!");
   laskuri++;
}
```

`break`-lauseen käyttö voi kuitenkin olla järkevää, jos kesken silmukan todetaan, että silmukan jatkaminen on syytä lopettaa. Ennen tämä tehtiin lisäämällä silmukan ehtoihin ylimääräisiä lippumuuttujia, mutta `break`-lauseen käyttäminen saattaa useissa tapauksissa olla selvempää.

15.6.2 continue

`continue`-lauseella hypätään silmukan alkuun ja silmukan suoritus jatkuu siitä normaalisti. Sillä voidaan siis ohittaa lohkon loppuosa.

```
for (int i = 0; i < 100; i++) {
   if ( i % 2 == 0 ) continue;
   System.out.println(i);
}
```

Yllä oleva ohjelmanpätkä siirtyy silmukan alkuun kun muuttujan `i` ja luvun 2 jakojäännös on 0. Muussa tapauksessa ohjelma tulostaa muuttujan `i` arvon. Toisin sanoen ohjelma tulostaa vain parittomat luvut. Myös `continue`-rakenne pystytään lähes aina välttämään. Yllä olevan ohjelmanpätkän voisi kirjoittaa vaikka seuraavasti:

```
for (int i = 0; i < 100; i++) {
   if ( i % 2 != 0 ) System.out.println(i);
}
```

Tai vielä yksinkertaisemmin seuraavasti:

```
for (int i = 1; i < 100; i += 2) {
   System.out.println(i);
}
```

Tyypillisesti `continue`-lausetta käytetään tilanteessa, jossa todetaan joidenkin arvojen olevan sellaisia, että tämä silmukan kierros on syytä lopettaa, mutta silmukan suoritusta täytyy vielä

jatkaa.

15.7 Ohjelmointikielistä puuttuva silmukkarakenne

Silloin tällöin ohjelmoinnissa tarvitsisimme rakennetta, jossa silmukan sisäosa on jaettu kahteen osaan. Ensimmäinen osa suoritetaan vaikka ehto ei enää olisikaan voimassa, mutta jälkimmäinen osa jätetään suorittamatta. Tällaista rakennetta ei Java-kielestä löydy valmiina. Tämän rakenteen voi kuitenkin tehdä itse, jolloin on perusteltua käyttää hallittua ikuista silmukkaa, joka lopetetaan `break`-lauseella. Rakenne voisi olla suunnilleen seuraavanalinen:

```
while ( true ) { //ikuinen silmukka
   Silmukan ensimmäinen osa //suoritetaan, vaikka ehto ei pädekkään
   if ( ehto ) break;
   Silmukan toinen osa //ei suoriteta enää, kun ehto ei ole voimassa
}
```

Jos silmukan ehdoksi asetetaan `true`, täytyy jossain kohtaa ohjelmassa olla `break`-lause, ettei silmukasta tulisi ikuista. Tällainen rakenne on näppärä juuri silloin, kun haluamme, että silmukan lopettamista tarkastellaan keskellä silmukkaa.

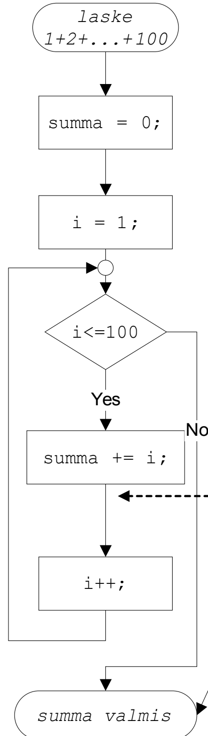
15.8 Yhteenveto

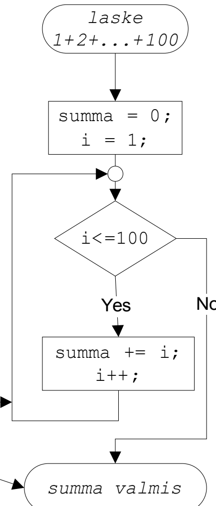
Silmukan valinta:

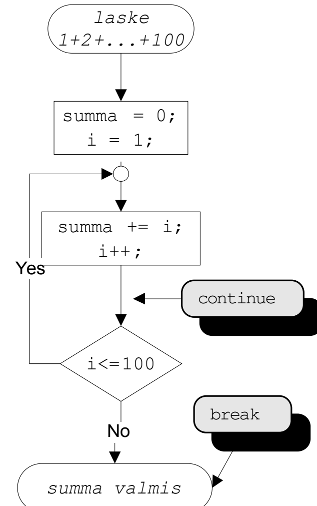
- `for`: Jos silmukan kierrosten määrä on ennalta tiedossa
- For-each: Jos haluamme tehdä jotain jonkun `Collection`-tietorakenteen tai taulukon kaikille alkioille. `Collection`-tietorakenteita on listattu luvussa 20 Dynaamiset tietorakenteet.
- `while`: Jos silmukan kierrosten määrä ei ole tiedossa, emmekä välttämättä halua suorittaa silmukkaa kertaakaan.
- `do-while`: Jos silmukan kierrosten määrä ei ole tiedossa, mutta haluamme suorittaa silmukan ainakin kerran.
- ”Ikuinen silmukka”, josta poistutaan `break`-lauseella: Jos haluamme, että poistuminen tapahtuu silmukan keskeltä.

Seuraava kuva kertaa vielä kaikki Javan valmiit silmukat:

Java-kielen silmukka-rakenteet








```
/* 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
```

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;
```

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)

Kuva 25: Javan silmukat

16. Merkkijonojen pilkkominen

16.1 StringTokenizer

Usein ohjelmoinnissa tulee tilanne, jossa haluamme pilkkoa merkkijonoa tietyn merkin kohdalta. Tämä onnistuu `StringTokenizer`-luokan oliolla. `StringTokenizer`-olio voidaan luoda muun muassa antamalla konstruktorille parametreina pilkottava merkkijono sekä merkkijonossa merkit joiden kohdalta pilkotaan. Voisimme esimerkiksi luoda `StringTokenizer`-olion pilkkomaan merkkijonoa `s` pilkkujen (,) kohdalta seuraavalla lauseella.

```
StringTokenizer st = new StringTokenizer(s,",");
```

Merkkejä joiden kohdalta pilkotaan voidaan antaa merkkijonossa vaikka kuinka monta. Jos halutaan pilkkoa merkkijonoa myös puolipisteen kohdalta, voidaan `StringTokenizer`-olio luoda seuraavasti.

```
StringTokenizer st = new StringTokenizer(s,";,");
```

Oliolta voi pyytää seuraavaa palaa `nextToken`-metodilla. Voisimme esimerkiksi tulostaa seuraavan palan lauseella:

```
System.out.println(st.nextToken());
```

Oliolta voi myös kysyä, että onko paloja vielä jäljellä. Tämä onnistuu `hasMoreTokens`-metodilla, joka palauttaa `true`, jos paloja on vielä jäljellä ja muuten `false`. Ennen kuin pyytää oliolta seuraavaa palaa, olisi syytä tarkastaa, että paloja on varmasti vielä jäljellä. Tämän voi tehdä vaikkapa `if`-lauseella seuraavasti.

```
if ( st.hasMoreTokens() ) System.out.println(st.nextToken());
```

16.1.1 Esimerkki: Merkkijonon pilkkominen StringTokenizerilla

Tehdään aliohjelma, joka tulostaa merkkijonon merkit allekkain niin, että merkkijono pilkotaan välilyönnin, puolipisteen ja pisteen kohdalta.

```
import java.util.StringTokenizer;

/**
 * Demonstroidaan merkkijonojen pilkkomista.
 * @author vesal
 * @version 13.10.2008
 */
public class Pilkkominen {
   /**
    * Tulostaa merkkijonon palat erotellen merkkijonon välilyönnin, puolipisteen ja
    * pilkun kohdalta.
    * @param s eroteltava merkkijono
    */
   public static void tulostaPalatTokenizer(String s) {
      StringTokenizer st = new StringTokenizer(s," ,;");
      System.out.println("Palasia tulee :" + st.countTokens());
      int n = 0;
      System.out.println("-------------------------------------------------");
      while ( st.hasMoreTokens() ) {
         String pala = st.nextToken();
         System.out.printf("%d: %s%n",n,pala);
         n++;
      }
   }

   /**
    * Testataan aliohjelmaa
    * @param args ei käytössä
    */
```

```
   public static void main(String[] args) {
      String s = "kissa,,,istuu,3,4,5,mato,kana;koira hirvi";
      tulostaPalatTokenizer(s);
   }
}
```

Tutkitaan esimerkkiä tarkemmin. Aliohjelma saa parametrinaan pilkottavan merkkijonon `s`.

```
StringTokenizer st = new StringTokenizer(s," ,;");
```

Yllä luodaan uusi `StringTokenizer`-olio, joka pilkkoo parametrimuuttujaa `s`, välilyönnin, puolipisteen, sekä pilkun kohdalta.

```
System.out.println("Palasia tulee :" + st.countTokens());
```

Yllä olevalla rivillä tulostetaan palasten lukumäärä käyttämällä `countTokens`-metodia.

```
while ( st.hasMoreTokens() ) {
```

Tässä aloitetaan `while`-silmukka. Ehtona on nyt, että silmukkaa suoritetaan niin kauan kun oliossa riittää palasia.

```
String pala = st.nextToken();
System.out.printf("%d: %s%n",n,pala);
n++;
```

Silmukan sisällä talletetaan seuraava palanen muuttujaan `pala` ja tulostetaan se muotoiltuna. Lopuksi päivitetään laskuria. Huomaa, että tässä ohjelmassa laskuria käytetään vain tulostuksessa. Silmukan ehdon kannalta sillä ei ole mitään merkitystä.

Ohjelman tulostus olisi nyt seuraava:

```
Palasia tulee :9
---------------------------------------------------
0: kissa
1: istuu
2: 3
3: 4
4: 5
5: mato
6: kana
7: koira
8: hirvi
```

16.2 split

Merkkijonoja voidaan pilkkoa myös `String`-olion `split`-metodilla. Metodi palauttaa palaset merkkijono-taulukossa. `split`-metodi pilkkoo merkkijonon parametrina annettavan *säännöllisen lausekkeen* (**regular expression**, **regex**) avulla. Säännöllinen lauseke tarjoaa monipuoliset ominaisuudet merkkijonon pilkkomiseen. Valitettavasti tämä tarkoittaa myös monimutkaista syntaksia. Säännöllisten lausekkeiden syntaksista voi lukea täältä:

http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html#sum

Merkkijono `s` saataisiin pilkottua välilyönnin, puolipisteen ja pilkun kohdalta nyt seuraavalla lauseella.

```
String[] palat = s.split("[ ,;]+");
```

Hakasulkeiden sisään kirjoitetut merkit ovat nyt vaihtoehtoja. Merkkijono katkaistaan, jos merkki on välilyönti, pilkku tai puolipiste. Perässä oleva ”+”-merkki tarkoittaa, että merkkejä saa esiintyä

peräkkäin myös useampia. Esimerkiksi merkkijono ”kissa,,,; koira” palauttaisi vain kaksialkioisen taulukon.

```
String s = "kissa,,; koira";
String[] palat = s.split("[ ,;]+"); //palat olisi nyt {"kissa", "koira"}
String[] palat2 = s.split("[ ,;]"); //palat2 olisi nyt {"kissa", "", "", "", "koira"}
```

Ilman ”+”-merkkiä peräkkäiset välilyönnit, pilkut tai puolipisteet aiheuttaisivat jokainen oman katkaisun.

Tee vastaava aliohjelma, joka tehtiin StringTokenizeria-käsiteltäessä käyttämällä split-metodia.

17. Järjestäminen

Kuinka järjestät satunnaisessa järjestyksessä olevan korttipakan kortit järjestykseen pienimmästä suurimpaan?

Yksi tutkituimmista ohjelmointiongelmista ja algoritmeista on järjestämisalgoritmi. Siis kuinka saamme esimerkiksi korttipakan kortit numerojärjestykseen. Tai ohjelmointiin soveltuvammin kuinka saamme järjestettyä taulukon luvut? Vaikka aluksi tuntuu, ettei erilaisia tapoja järjestämiseen ole kovin montaa, on niitä oikeasti kymmeniä ellei satoja.

Järjestämisalgoritmeja käsitellään enemmän muilla kursseilla (esim. ITKA201 Algoritmit 1 ja TIEP111 Ohjelmointi 2). Tässä vaiheessa meille riittää, että osaamme käyttää Javasta valmiina löytyvää järjestämisaliohjelmaa `sort`. Tämä on siitäkin syystä järkevää, että kielestä valmiina löytyvä järjestämisalgoritmi on lähes aina nopeampi, kuin itse tehty.

Taulukot voidaan järjestää käyttämällä `Arrays`-luokasta löytyvää `sort`-aliohjelmaa. Parametrina `sort`-aliohjelma saa järjestettävän taulukon. Aliohjelman tyyppi on `static void`, eli se ei palauta mitään, vaan ainoastaan järjestää taulukon.

```
int[] taulukko = {-4,5,-2,4,5,12,9};
Arrays.sort(taulukko);

//Tulostetaan alkiot, että nähdää onnistuiko järjestäminen.
for (int alkio : taulukko) {
   System.out.println(alkio);
}
```

Alkioiden pitäisi nyt tulostua numerojärjestyksessä. Taulukko voitaisiin myös järjestää vain osittain antamalla `sort`-aliohjelmalle lisäksi parametreina aloitus- ja lopetusindeksit.

```
int[] taulukko2 = {-4,5,-2,4,5,12,9};

//järjestetään nyt vain neljä ensimmäistä alkiota
Arrays.sort(taulukko2,0,3);

//Tulostetaan alkiot, että nähdää onnistuiko järjestäminen.
for (int alkio : taulukko2) {
   System.out.println(alkio);
}
```

Kaikkia alkeistietotyyppisiä taulukoita voidaan järjestää `sort`-aliohjelmalla. Lisäksi voidaan järjestää taulukoita, joiden alkioiden tietotyyppi *toteuttaa* (**implements**) `Comparable`-rajapinnan. Esimerkiksi `String`-luokka toteuttaa tuon rajapinnan. Rajapinnoista puhutaan lisää kohdassa 20.1 Rajapinnat.

18. Konsoliohjelmien tekeminen

Tähän asti olemme tehneet ohjelmia, jotka eivät kommunikoi käyttäjän kanssa mitenkään. Lähes kaikki ohjelmat kuitenkin ovat jotenkin vuorovaikutuksessa käyttäjänsä kanssa. Tietokoneohjelmia voidaankin jakaa sen mukaan kuinka ne kommunikoivat käyttäjän kanssa:

- Konsolisovellukset
- Graafiset työpöytäsovellukset
- Web-sovellukset
- Mobiilisovellukset
- jne...

Tämän kurssin puitteissa tutustutaan lähinnä vain konsolisovelluksiin. Muiden ohjelmien tekemistä oppii enemmän muilla kursseilla:

- TIEA212 Graafisten käyttöliittymien ohjelmointi
- TJTA270 www-sovellukset
- TIEP113 Ohjelmointi 2, JSP
- TIEP111 Ohjelmointi 2 (esimerkki mobiilisovelluksista)
- ym...

Javassa konsolisovellusten tekeminen onnistuu lukemalla käyttäjän syöttämää dataa ja tulostamalla tekstiä näytölle. Tekstiä olemmekin jo tulostaneet `System.out`-oliolla. Käyttäjän syöttämän datan lukeminen onnistuu sen sijaan `Scanner`-oliolla. `Scanner`-olio saa parametrikseen `System.in`-olion eli Javan standardisyöttövirran (**standard input stream**). Javassa on kaksi muutakin standardivirtaa.

18.1 Tietovirrat

18.1.1 Standardivirrat

- `System.out`: standarditulostusvirta (**standard output stream**)
- `System.err`: standardivirhetulostusvirta (**standard error output stream)**
- `System.in`: standardisyöttövirta (**standard input stream**)

Kaikilla standardivirroilla on oma tarkoituksensa. Tulostusvirtaa käytetään tekstin tulostukseen näytölle. Virhetulostusvirta on taas tarkoitettu virhetulostuksia varten. Syöttövirrasta taas voidaan lukea käyttäjän konsoliin syöttämää tekstiä.

Standardivirtojen lisäksi on olemassa muitakin virtoja, mutta niitä ei tällä kurssilla juurikaan käsitellä.

18.2 Käyttäjän syötteen lukeminen

Kirjoitetaan suoraan pääohjelmaan yksinkertainen ohjelma, joka pyytää käyttäjän kirjoittamaan jotain ja tulostaa sen.

```
import java.util.Scanner;

/**
 * Luokassa demonstroidaan syötteen lukua Scanner-oliolla.
 */
public class Konsolisovellukset {

   /**
    * Ohjelma kysyy käyttäjältä jotain ja tulostaa mitä
    * käyttäjä kirjoitti.
    * @param args
    */
   public static void main(String[] args) {
```

```
        System.out.print("Kirjoita jotain >");
        Scanner sc = new Scanner(System.in);
        String rivi = sc.nextLine();
        System.out.println("Kirjoitit: " + rivi);
    }
}
```

Aluksi tulostetaan kehotus käyttäjälle, että hänen tulisi kirjoittaa jotain. Konsolisovelluksissa käyttäjälle täytyy kertoa, koska hänen täytyy syöttää tekstiä, sillä muuten sitä ei välttämättä huomaa.

```
Scanner sc = new Scanner(System.in);
```

Tällä rivillä luodaan uusi `Scanner`-olio. `Scanner`-olio osaa muuttaa lähdedatansa tietyksi alkeistietotyypiksi tai merkkijonoksi. Se soveltuu erinomaisesti käyttäjän syöttämän datan käsittelyyn. `Scanner`-oliolle voi antaa parametriksi `System.in`-olion eli Javan standardin syöttövirran, kuten yllä on tehty.

```
String rivi = sc.nextLine();
```

Tällä rivillä määritellään uusi `String`-tyyppinen muuttuja, johon luetaan `Scanner`-oliosta seuraava kokonainen rivi. Tähän käytetään `Scanner`-olion `nextLine`-metodia, joka siis palauttaa rivin `String`-tyyppisenä. Muuttuja rivi saa arvokseen kaiken sen mitä käyttäjä syöttää ennen kuin painaa **enter**. Lähdetekstiä voidaan koittaa muuttaa myös tietyksi alkeistyypiksi muilla `Scanner`-luokan metodeilla. `Scanner`-luokan metodit löytyvät Javan dokumentaatiosta.

18.2.1 Esimerkki: Yksinkertainen käyttöliittymä switch-case -rakenteen avulla

Tehdään yksinkertainen käyttöliittymä käyttämällä `switch-case` -rakennetta.

```
public class Käyttöliittymä {

    /**
     * Yksinkertaisen konsolikäyttöliittymän malli
     * @param args ei käytössä
     */
    public static void main(String[] args) {
        char merkki;
        Scanner lukija = new Scanner(System.in);
        while ( true ) {
            System.out.println("Hyväksyttävät komennot: A,B,L = Lopeta");
            System.print(">");

            String syote = lukija.nextLine();
            if ( syote.length() == 0 ) continue; //jos painettiin vaan enter
            merkki = syote.charAt(0); //otetaan merkkijonon ensimmäinen merkki
            merkki = Character.toUpperCase(merkki); //muutetaan merkki isoksi
            switch(merkki) {
                case 'A':
                    System.out.println("Painoit A");
                    break;
                case 'B':
                    System.out.println("Painoit B");
                    break;
                case 'L':
                    System.out.println("Kiitos ohjelman käytöstä!");
                    return;
                default:
                    System.out.println("Virheellinen syöte");
            }
        }
    }
}
```

Pääohjelman alussa määritellään muuttuja `merkki`, sekä `Scanner`-olio jota käytetään käyttäjän syötön lukuun.

Seuraavaksi pääohjelmassa aloitetaan `do-while` -silmukka, joka toistaa ohjelmaa kunnes käyttäjä

syöttää L-kirjaimen. Aluksi `do-while` -silmukan sisällä tulostetaan ohjeet siitä, mitä komentoja on käytössä. Tulostusten jälkeen luetaan käyttäjän syöttämä rivi `Scanner`-olion `nextLine`-metodilla `syote`-nimiseen muuttujaan. Metodi `nextLine` lukee siis nimensä mukaan seuraavan rivin, eli tässä tapauksessa kaiken mitä käyttäjä syöttää ennen kuin painaa **enter**.

Seuraavaksi on syytä tarkistaa, ettei käyttäjä syöttänyt tyhjää merkkijonoa eli painanut pelkästään **enter**:iä. Tämä tarkastus voidaan tehdä `String`-olion `length`-metodilla. Merkkijonon pituus palautetaan `length`-metodilla. Jos merkkijonon pituus on nolla jatketaan silmukan suoritusta alusta `continue`-lauseella. Ilman tätä tarkistusta ohjelma kaatuisi käyttäjän painaessa **enter**, sillä seuraavalla rivillä viitattaisiin merkkijonon kirjaimeen, jota ei ole olemassa.

`String`-oliot sisältävät `charAt`-metodin. Sillä saadaan merkkijonosta käyttöön yksittäinen merkki. Parametrinaan se saa indeksin, eli monesko merkki merkkijonosta halutaan. Ensimmäinen merkki on indeksissä 0, kuten taulukoissa ensimmäinen alkio. Alla on tätä käyttämällä otettu `syote`-muuttujan ensimmäinen kirjain.

```
merkki = syöte.charAt(0);
```

Tällä tavalla saadaan syötteestä `char`-tyyppinen, joka kelpaa `switch-case` rakenteeseen. Voisi olla järkevämpää tulostaa virheilmoitus jos halutaan, että käyttäjä syöttää yksittäisen merkin, mutta syöttääkin merkkijonon. Jätetään tämän tekeminen kuitenkin harjoitustehtäväksi.

Muutetaan merkki vielä joka kerralla isoksi kirjaimeksi, niin ohjelmamme toimii huolimatta siitä syöttääkö käyttäjä isoja vai pieniä kirjaimia. Tämä on tehty alla:

```
merkki = Character.toUpperCase(merkki);
```

Koska `char` on alkeistietotyyppi, ei sillä voi olla metodeja kuten olioilla. Tämän takia emme voi tehdä seuraavaa kutsua: ~~`merkki.toUpperCase()`~~, joka onnistuisi kyllä `String`-tyyppisille muuttujille. On kuitenkin olemassa `Character`-kääreluokka, joka sisältää `toUpperCase`-metodin. Se saa parametrinaan merkin ja palauttaa sen isona kirjaimena.

Seuraavana on `switch-case`-rakenne. `case`-osat tulostavat nyt vain sen tiedon mitä merkkiä olemme painaneet, mutta näihin kohtiin olisi nyt helppo kirjoittaa jonkun oikean ohjelman toiminnallisuudet.

18.3 Käyttäjän syötteen lukeminen Ali.jar kirjastoa käyttämällä

Vaikka `Scanner`-luokka on oleellinen parannus Javan aiempaan syötteen käsittelyyn, ei senkään käyttöä voida suositella numeeristen arvojen lukemiseen, sillä mahdollisesta virheestä toipuminen vaatii varsin paljon ylimääräistä koodia.

Siksi käyttäjän syötteen lukemista on helpotettu Jyväskylän Yliopiston Ali.jar-kirjaston `Syotto`-luokalla. Ali.jar -kirjaston dokumentaation, käyttöohjeet ja itse tiedoston löytää täältä:

http://users.jyu.fi/~vesal/kurssit/ohj2/ali/ .

Esimerkiksi merkkijono voidaan lukea `Syotto`-luokan `kysy`-aliohjelmalla seuraavasti:

```
String syote = Syotto.kysy("Syötä joku merkkijono");
System.out.println("Syötit: " + syote);
```

Parametriksi `kysy`-aliohjelmalle voi antaa siis suoraan käyttäjältä kysyttävän kysymyksen. Yllä olevan koodin toiminta olisi seuraavanlainen:

```
Syötä joku merkkijono >Moi[ret]
```

```
Syötit: Moi
```

`kysy`-aliohjelmalla voidaan kysyä myös suoraan `int`- tai `double`-tyyppisiä arvoja. Tällöin toiseksi parametriksi kysymykselle pitää vain antaa oletusarvo. Oletusarvolla tarkoitetaan tässä sitä, että jos käyttäjä syöttää tyhjän merkkijonon, eli painaa ainoastaan **Enter**, niin tällöin metodi palauttaa oletusarvon. Esimerkiksi `int`-tyyppisen luvun kysyminen onnistuu seuraavasti:

```
int luku = Syotto.kysy("Syötä kokonaisluku", 0);
System.out.println("Syötit: " + luku);
```

Yllä olevan koodin toiminta olisi seuraava:

```
Syötä kokonaisluku (0) >4[ret]
Syötit: 4
```

Jos emme halua käyttää oletusarvoa, voimme kysyä luvun suoraan `kysyInt`-metodilla seuraavasti:

```
int luku2 = Syotto.kysyInt("Syötä kokonaisluku");
System.out.println("Syötit: " + luku2);
```

Koodin toiminta:

```
Syötä kokonaisluku >2[ret]
Syötit: 2
```

Vastaavasti onnistuisi `double`-tietotyyppien kyseleminen:

```
double luku3 = Syotto.kysy("Syötä reaaliluku",0.0);
double luku4 = Syotto.kysyDouble("Syötä toinen reaaliluku");

System.out.println("Syötit " + luku3 + " ja " + luku4);
```

Toiminta:

```
Syötä reaaliluku (0.0) >2.4[ret]
Syötä toinen reaaliluku >5.7[ret]
Syötit 2.4 ja 5.7
```

Kaikissa `Syotto`-luokan aliohjelmissa on se hienous, että meidän ei tarvitse huolehtia virheellisistä syötteistä. Lukuja kyselevät aliohjelmat nimittäin tarkastavat syöttääkö käyttäjä luvun ja jatkavat kyselyä niin kauan kunnes käyttäjä syöttää oikein. Jos käyttäjä syöttää tyhjän merkkijonon, aliohjelmat palauttavat oletuksena viedyn luvun arvon, edellisessä esimerkissä `0.0`.

`Syotto`-luokasta löytyy myös muita aliohjelmia. Tarkempaa tietoa muista aliohjelmista ja tässä käsitellyistä aliohjelmista löydät luokan Ali.jar-kirjaston dokumentaatiosta.

Muuta monisteen alkupuolella tehdystä painoindeksi-ohjelmasta sellainen, että pituus ja paino kysytään käyttäjältä.

18.4 Parametrien antaminen ohjelmaa käynnistettäessä (args-taulukko)

Konsoliohjelmalle voidaan antaa parametreja ohjelmaa käynnistettäessä. Parametrit kirjoitetaan komentoriville ohjelman nimen perään välilyönnin jälkeen ja erotetaan toisistaan välilyönneillä. Voisimme käynnistää esimerkiksi `Summa`-ohjelman komentoriviltä seuraavasti antamalla sille parametrina kaksi lukua:

```
java Summa 5 8
```

Ohjelmakoodissa parametreihin päästään käsiksi pääohjelman `args`-taulukon avulla. Parametrit tallentuvat siis String tyyppiseen `args`-nimiseen taulukkoon. Summa ohjelmamme voisi olla

esimeriksi seuraavanlainen:

```
public class Summa {

   /**
    * Summa ohjelma jolle voidaan antaa summattavat luvut parametrina.
    * @param args summattavat luvut
    */
   public static void main(String[] args) {
     if(args.length() == 0) return; //käyttäjä ei antanut parametreja
     double summa = 0;
     for(int i=0; i <  args.length; i++) {
       try {
         summa += Double.parseDouble(args[i]);
       } catch (NumberFormatException e) {
         System.out.println("Parametria ei saatu muutettua reaaliluvuksi.");
       }
     }
     System.out.println("Lukujen summa on: " + summa);
   }
}
```

Ohjelmassa käydään läpi kaikki ohjelman käynnistyksessä annetut parametrit ja koetetaan parsia niistä liukuluku ja lisätään ne `summa`-nimiseen muuttujaan. Käyttäjän antamien parametrien määrä saadaan selville `args`-taulukon parametreista kuten alla olevassa `for`-silmukan esittelyrivissä on tehty.

```
for(int i=0; i <  args.length; i++) {
```

For-silmukan sisällä oleva try-catch -rakenne on poikkeusten käsittelyä, eikä siitä kannata tässä vaiheessa hämääntyä. Try-catch -rakenteesta puhutaan lisää luvussa 23. Poikkeukset.

```
summa += Double.parseDouble(args[i]);
```

Yllä oleva rivi on ohjelman oleellisin rivi. Siinä `args`-taulukon käsittelyssä oleva parametri lisätään `summa`-nimiseen muuttujaan. `Double.parseDouble`-metodi yrittää parsia saamastaan parametrista liukuluvun.

19. Rekursio

“To iterate is human, to recurse divine.” -L. Peter Deutsch

Rekursiolla tarkoitetaan algoritmia joka tarvitsee itseään ratkaistakseen ongelman. Ohjelmoinnissa esimerkiksi aliohjelmaa, joka kutsuu itseään, sanotaan rekursiiviseksi. Rekursiolla voidaan ratkaista näppärästi ja pienemmällä määrällä koodia monia ongelmia, joiden ratkaiseminen olisi muuten (esim. silmukoilla) melko työlästä. Rakenteeltaan rekursiivinen algoritmi muistuttaa jotain seuraavaa:

```
public static void rekursio(parametrit) {
   if ( joku lopetusehto ) return;
   jotain toimenpiteitä;
   rekursio(uudet parametrit);  //itsensä kutsuminen
}
```

Oleellista on, että rekursiivisessa aliohjelmassa on joku lopetusehto. Muutoin aliohjelma kutsuu itseään loputtomasti. Toinen oleellinen seikka on, että aina seuraavan kutsun parametreja jotenkin muutetaan, muutoin rekursiolla ei saada mitään järkevää aikaiseksi.

Yksinkertainen esimerkki rekursioista voisi olla kertoman laskeminen. Muistutuksena viiden kertoma on siis tulo 5*4*3*2*1. Tämä ei välttämättä ole paras tapa laskea kertomaa, mutta havainnollistaa rekursiota hyvin.

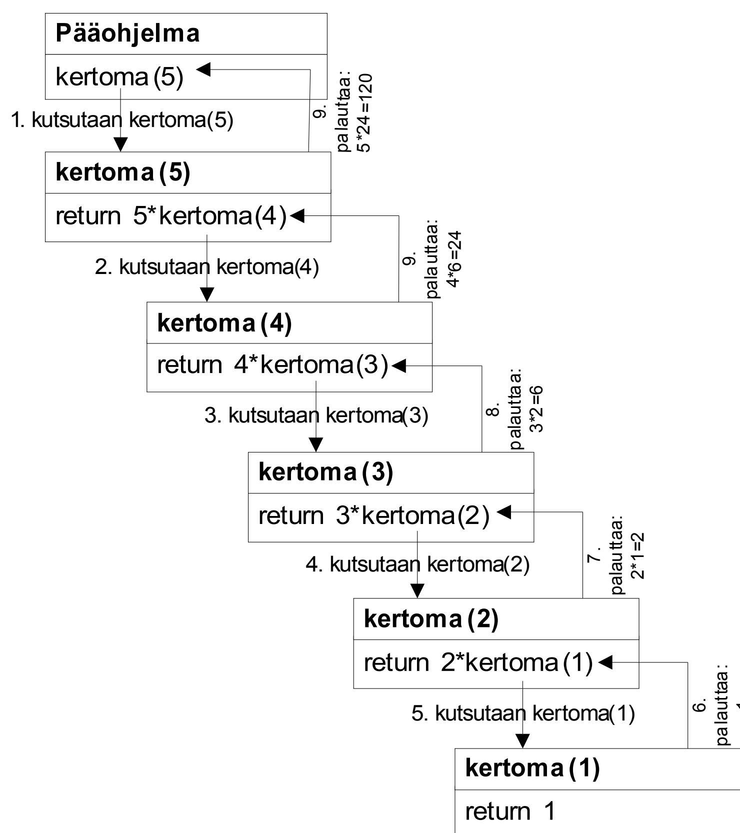
```
public class Kertoma {
   /**
    * Lasketaan luvun kertoma kaavasta
    * <pre>
    *  0! = 0
    *  1! = 1
    *  n! = n*(n-1)!;
    * </pre>
    * @param n minkä luvun kertoma lasketaan
    * @return n!
    */
   public static long kertoma(int n) {
      if ( n <= 1 ) return 1;
      return n*kertoma(n-1);
   }

   /**
    * @param args ei käytössä
    */
   public static void main(String[] args) {
      long k = kertoma(10);
      System.out.println(k);
   }
}
```

Aliohjelma `kertoma` saa parametrikseen luvun, jonka kertoma halutaan laskea. Tutustutaan aliohjelmaan tarkemmin.

```
if ( n <= 1 ) return 1;
```

Yllä oleva rivi on ikään kuin rekursion lopetusehto. Jos `n` on pienempää tai yhtä suurta kuin 1, niin palautetaan luku 1. Oleellista on, että lopetusehto on ennen uutta rekursiivista aliohjelmakutsua.

```
return n*kertoma(n-1);
```



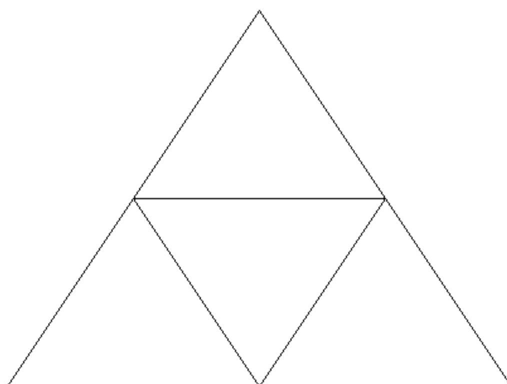
Kuva 26: Kertoman laskeminen rekursiivisesti. Vaiheet numeroitu.

Tällä rivillä tehdään nyt tuo rekursiivinen kutsu eli aliohjelma kutsuu itseään. Yllä oleva rivi onkin oikeastaan tuttu matematiikasta: n! = n*(n-1)! . Siinä palautetaan siis n kerrottuna n -1 kertomalla. Esimerkiksi luvun viisi kertoman laskemista yllä olevalla aliohjelmalla voisi havainnollistaa seuraavasti.

Tulosta voidaan lähteä kasaamaan lopusta alkuun päin. Nyt `kertoma(1)` palauttaa siis luvun 1 ja samalla lopettaa rekursiivisten kutsujen tekemisen. `kertoma(2)` taas palauttaa 2*`kertoma(1)` eli 2*1 eli luvun 2. Nyt taas `kertoma(3)` palauttaa 3*`kertoma(2)` eli 3*2 ja niin edelleen. Lopulta `kertoma(5)` palauttaa 5*`kertoma(4)` eli 5*24 = 120. Näin on saatu laskettua viiden kertoma rekursiivisesti.[LIA]

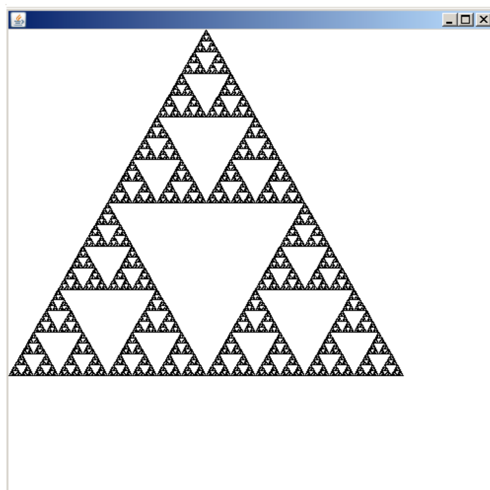
19.1 Sierpinskin kolmio

Sierpinskin kolmio on puolalaisen matemaatikko Wacław Sierpińskin vuonna 1915 esittelemä fraktaali. Se on tasasivuinen kolmio, jonka keskelle piirretään toinen tasasivuinen kolmio, niin että uuden kolmion kärjet ovat edellisen kolmion sivujen keskipisteet. Toisaalta piirtämisen voi ajatella niin, että jokaisen kolmion sisään piirretään kolme samankokoista tasasivuista kolmiota, joiden korkeus on puolet ulommasta kolmiosta. Lisäksi kaikkien kolmioiden kaksi kärkeä koskettavat molempien muiden kolmioiden kärkiä. Uudet kolmiot muodostuvat siis kolmion yläosaan, vasempaan alakulmaan ja oikeaan alakulmaan. Tilanne selviää paremmin kuvasta. Sierpinskin kolmion toinen vaihe on alla:



Kuva 27: Sierpinskin kolmion toisessa vaiheessa ensimmäisen kolmion sisään on piirretty kolme uutta kolmiota.

sekä lopputulos:



Kuva 28: Valmis Sierpinskin kolmio

Sierpinskin kolmion piirtäminen onnistuu loistavasti rekursiolla. Sierpinskin kolmiosta voi lukea lisää esim. Wikipediasta: http://en.wikipedia.org/wiki/Sierpinski_triangle.

Sierpinskin kolmio voitaisiin piirtää seuraavalla algoritmilla:

```
1. Piirrä tasasivuinen kolmio.
2. Piirrä kolmion sisään kolme uutta tasasivuista kolmiota,
   niin että niiden uusien kolmioiden korkeus ja leveys puolitetaan ja
   jokainen kolmio koskettaa kahta muuta kolmiota kärjillään.
3  Tee uusille kolmiolle kohta 2.
```

Algoritmia voitaisiin nyt tarkentaa *pseudokoodiksi*:

pseudokoodi = Ohjelmointikieltä muistuttavaa koodia, jonka tarkoitus on piilottaa eri ohjelmointikielten syntaksierot ja jättää jäljelle algoritmin perusrakenne. Algoritmia suunniteltaessa voi olla helpompaa hahmotella ongelmaa ensiksi pseudokielisenä, ennen kuin kirjoittaa varsinaisen ohjelman. Pseudokoodille ei ole mitään standardia, vaan jokainen voi kirjoittaa sitä omalla tavallaan. Järkevintä kuitenkin kirjoittaa niin, että mahdollisimman moni ymmärtäisi sitä.

```
piirraSierpinskinKolmio(korkeus, paikka) {
   piirraKolmio(korkeus,paikka)
```

```
    piirraSierpinskinKolmio(korkeus/2, ylaKolmionPaikka)
    piirraSierpinskinKolmio(korkeus/2, vasemmanAlaKolmionPaikka)
    piirraSierpinskinKolmio(korkeus/2, oikeanAlaKolmionPaikka)
}
```

Tämä muistuttaa jo paljon oikeaa koodia. Käytetään piirtämiseen tuttua `EasyWindow`-luokkaa. Piirtäminen onnistuu nyt seuraavalla koodilla.

```
package esimerkit;
import fi.jyu.mit.graphics.EasyWindow;

/**
 * @author vesal
 * @version 17.8.2009
 */
public class GraafinenRekursio {

	final static double PIENIN_KOLMIO = 1.00;
	/**
	 * Piirtää Sierpinskin kolmion. Parametreina kolmion vasemman
	 * alakulman x- ja y-koordinaatit, sekä kolmoin korkeus.
	 * @param window ikkuna johon piirretään
	 * @param x kolmion vasemman kulman x-koordinaatti
	 * @param y kolmion vasemman kulman y-koordinaatti
	 * @param h kolmion korkeus
	 */
	public static void sierpinskinKolmio(EasyWindow window, double x,
			double y, double h) {

		//s on kolmion sivun pituus
		//MAOL: tasasivuisen kolmion kaavat
		double s = (2*h / (Math.sqrt(3)));

		//varsinaisen kolmion piirto
		window.addLine(x, y, x+s/2, y-h); //vasen sivu
		window.addLine(x, y, x+s, y);  //pohja
		window.addLine(x+s, y, x+s/2, y-h); //oikea sivu

		if (h < PIENIN_KOLMIO) return;

		sierpinskinKolmio(window, x, y, h/2); //Vasen alakolmio
		sierpinskinKolmio(window, x+s/4, y-h/2, h/2); //Yläkolmio
		sierpinskinKolmio(window, x+s/2, y, h/2); //Oikea alakolmio
	}

	/**
	 * Piirretään rekursiivinen kuvio
	 * @param args ei käytössä
	 */
	public static void main(String[] args) {
		EasyWindow window = new EasyWindow();
		window.showWindow();
		sierpinskinKolmio(window, 0, 200, 200);
	}
}
```

Tarkastellaan ohjelmaa hieman tarkemmin.

```
final static double PIENIN_KOLMIO = 1.00;
```

Ennen varsinaista aliohjelmaa on määritelty globaalivakio, jolla kontrolloidaan kuinka kauan rekursiota jatketaan. Vakio `PIENIN_KOLMIO` näkyy siis kaikkialla luokassa `GraafinenRekursio`. `PIENIN_KOLMIO` on määritelty globaaliksi, ettei muuttujan alustus toistuisi järjettömän monta kertaa. Tässä ohjelmassa voidaan nimittäin suorittaa aliohjelma `sierpinskinKolmio` todella monta kertaa, riippuen vakion `PIENIN_KOLMIO` arvosta.

Aliohjelma `sierpinskinKolmio` saa neljä parametria: ikkunan johon kolmio piirretään, kolmion vasemman kärjen x- ja y-koordinaatit, sekä kolmion korkeuden. Nämä parametrit riittävät tasasivuisen kolmion piirtämiseen. Se, mistä pisteestä kolmion muut pisteet lasketaan, on tietysti makuasia. Parametrina voisi vasemman kärjen koordinaattien sijaan olla yhtä hyvin siis huipun

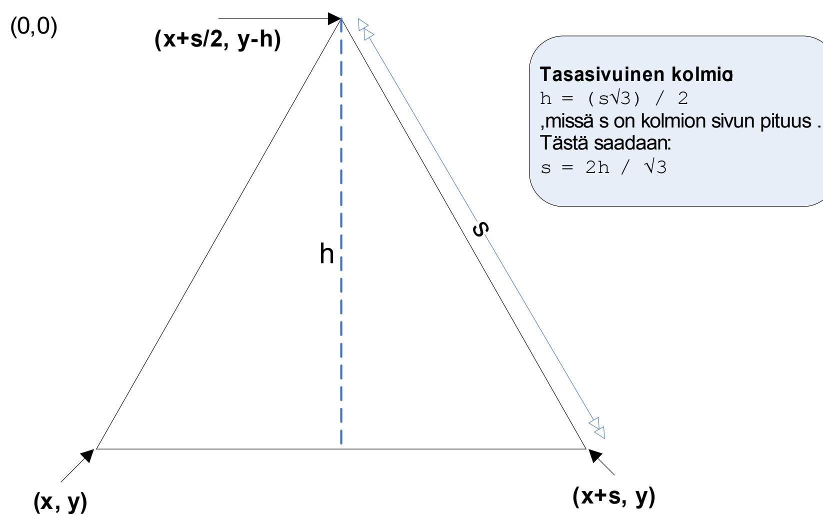
koordinaatit tai vaikka pohjan keskikohdan koordinaatit.

```
double s = (2*h / (Math.sqrt(3)));
```

Kolmion piirtäminen onnistuu helposti, kun laskemme muistiin muuttujaan `s` kolmion sivun pituuden kuten yllä.

```
window.addLine(x, y, x+s/2, y-h); //vasen sivu
window.addLine(x, y, x+s, y);  //pohja
window.addLine(x+s, y, x+s/2, y-h); //oikea sivu
```

Yllä olevat rivit piirtävät varsinaisen kolmion. Jokainen `sierpinskinKolmio`-aliohjelman suoritus piirtää siis ainoastaan yhden kolmion. Kolmion pisteiden laskemiseen käytetään nyt äsken laskettua kolmion sivun pituutta. Alla oleva kuva selkeyttää pisteiden laskentaa. Muista, että kolmio oli tasasivuinen, eli kaikki sivut ovat yhtä pitkät!



Kuva 29: Varsinaisen piirrettävän kolmion pisteiden laskeminen.

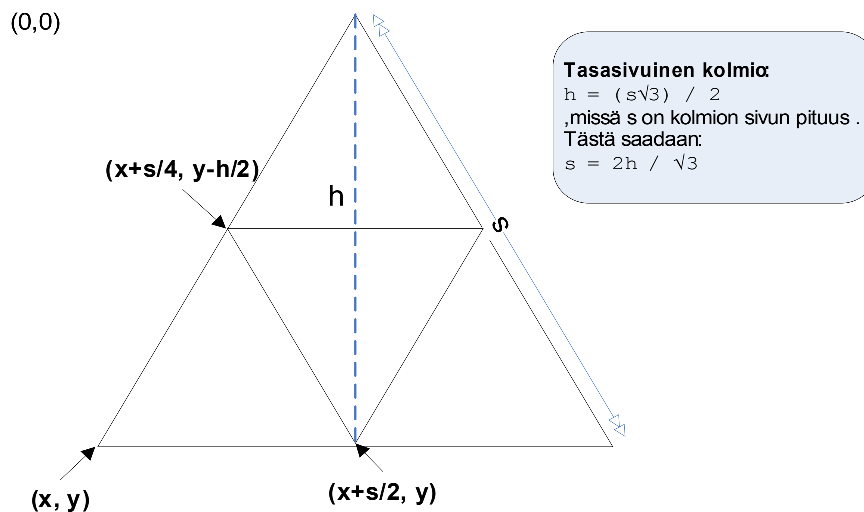
Kun kolmio on piirretty tarkastetaan jatketaanko rekursiivisia aliohjelmakutsuja. Tämä tehdään seuraavalla tarkastelulla:

```
if (h < PIENIN_KOLMIO) return;
```

Jos aliohjelman parametrimuuttuja `h` on siis pienempi kuin globaalivakio `PIENIN_KOLMIO`, voidaan rekursiiviset kutsut lopettaa. Koska aliohjelma oli tyyppiä `void`, palataan siitä pelkällä `return`-lauseella, ilman paluuarvoa.

```
sierpinskinKolmio(window, x, y, h/2); //Vasen alakolmio
sierpinskinKolmio(window, x+s/4, y-h/2, h/2); //Yläkolmio
sierpinskinKolmio(window, x+s/2, y, h/2); //Oikea alakolmio
```

Yllä olevilla kolmella rivillä suoritetaan sitten rekursiiviset aliohjelmakutsut, eli aliohjelma kutsuu itseään. Nyt halutaan siis piirtää äsken piirretyn kolmion sisään kolme uutta kolmiota. Tätä tehtävää varten kutsutaan `sierpinskinKolmio`-aliohjelmaa. Nyt meidän täytyy vain laskea aliohjelmalle uudet parametrit, jotta kolmiot saadaan piirrettyä oikeisiin kohtiin. Uudet kolmiot piirretään tietenkin samaan ikkunaan kuin edellisetkin, joten ensimmäinen parametri on kaikilla kutsuilla sama `window`. Kaikkien uusien kolmioiden korkeus on puolet edellisen kolmion korkeudesta. Lisäksi parametrina piti antaa vasemman kärjen koordinaatit, jonka avulla uusien kolmioiden piirtäminen jälleen tapahtuu. Nämä saadaan jälleen laskettua kolmion sivun pituuden avulla. Laskemista selventää paremmin alla oleva kuva:



Kuva 30: Uusien rekursiivisten kutsujen pisteiden laskeminen.

Pääohjelmassa luodaan aluksi uusi `EasyWindow`-olio. Tämän jälkeen kutsutaan sen `showWindow`-metodia ja lopuksi kutsutaan vielä `sierpinskinKolmio`-aliohjelmaa. Metodia `showWindow` voitaisiin kutsua myös `sierpinskinKolmio`-aliohjelman jälkeen, riippuen haluammeko nähdä kolmion piirtämisen vai emme. Koska prosessori piirtää kolmiota näytölle aika vauhdilla, voimme lisätä aliohjelmaan hieman viivettä, jolloin näemme kuinka kolmion piirtäminen tapahtuu. Viiveen lisääminen onnistuu lisäämällä johonkin kohtaan aliohjelmaa seuraavaa koodinpätkä:

```
//Viivettä piirtämiseen
try {
       Thread.sleep(100);
} catch (InterruptedException e) {
       e.printStackTrace();
}
```

`Thread`-luokan `sleep`-aliohjelma odottaa parametrina saamansa ajan, ennen kuin ohjelman suoritus jatkuu. Parametrina annettava aika on kokonaisluku millisekunteina.

Montako kertaa tässä esimerkissä lopulta suoritetaan aliohjelma `piirraSierpinskinKolmio`?

19.2 Nopeampi Sierpinskin Kolmio

Kokeile muutella `PIENIN_KOLMIO`-vakion arvoa. Kutsujen määrä kasvaa eksponentiaalisesti, sitä mukaan kun `PIENIN_KOLMIO`-vakion arvoa pienennetään. Ohjelman suoritus alkaakin hidastua huomattavasti, kun arvo alkaa mennä alle 1.0:n. Tämä aliohjelma ei olekaan kovin optimaalinen koska sisäisesti `addLine`-metodi luo joka kutsulla uuden olion viivaa varten ja laskemista varten vielä muutaman muun. Huomattavasti nopeamman version saisi tekemällä koko `sierpinskinKolmion`-piirrosta yhden olion.

Tämä toteutus on monisteessa liitteenä. Olioiden tekeminen ei kuulu tälle kurssille. Halutessasi voit kuitenkin vertailla toteutusten suoritusnopeutta.

Lisää siis äsken tehdyssä esimerkissä olleen `GraafinenRekursio`-luokan sisälle seuraava *aliluokka* (**subclass**).

```
    /**
     * Luokka Sierpinskin kolmion piirtämiseksi niin, että
     * se on vain yksi objekti.
     * @author vesal
     * @version 17.8.2009
     */
```

```
    public static class SierpinskinKolmio2 extends BasicShape implements Drawable {
        private double x;
        private double y;
        private double h;

        /**
         * Alustetaan Sierpipnskin kolmio
         * @param x  kärjen x-koordinaatti
         * @param y  kärjen y-koordinaatti
         * @param h  kolmion korkeus
         */
        public SierpinskinKolmio2(double x, double y, double h) {
           this.x = x;
           this.y = y;
           this.h = h;
        }


        private static class PiirraKolmio {
            Vector vr = new Vector();
            SPoint p1 = new SPoint(0,0);
            SPoint p2 = new SPoint(0,0);
            SPoint p3 = new SPoint(0,0);
            Graphics g;
            Matrix a;

            private PiirraKolmio(Graphics g, Matrix a) {
                this.g = g;
                this.a = a;
            }

            private void kolmio(double x, double y, double h ) {
                double s2 = h / (Math.sqrt(3));
                a.transform(vr.set(x,y)      ,p1);
                a.transform(vr.set(x-s2, y-h),p2);
                a.transform(vr.set(x+s2, y-h),p3);
                g.drawLine(p1.getX(), p1.getY(), p2.getX(), p2.getY());
                g.drawLine(p2.getX(), p2.getY(), p3.getX(), p3.getY());
                g.drawLine(p3.getX(), p3.getY(), p1.getX(), p1.getY());

                if (h < PIENIN_KOLMIO) return;
                kolmio(x-s2, y  , h/2); // Vasen alakolmio
                kolmio(x+s2, y  , h/2); // Oikea alakolmio
                kolmio(x   , y-h, h/2); // Yläkolmio
            }

        }

      /**
         * @param g
         * @param a
         */
        protected void drawShape(Graphics g, Matrix a) {
            PiirraKolmio kolmio = new PiirraKolmio(g, a);
            kolmio.kolmio(x,y,h);
        }

    }
```

Tämän jälkeen muuta pääohjelma alla olevaan muotoon:

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

       EasyWindow window1 = new EasyWindow();
       window1.showWindow();

       long alkuAikaOlio = System.currentTimeMillis();
       window1.add(new SierpinskinKolmio2(250, 450, 200));
       long loppuAikaOlio = System.currentTimeMillis();

       System.out.println("Yhden olion totetuksella kesti: " +
                   (loppuAikaOlio - alkuAikaOlio) + " millisekuntia");

       EasyWindow window2 = new EasyWindow();
       window2.showWindow();
       long alkuAika = System.currentTimeMillis();
```

```
        sierpinskinKolmio(window2, 0, 200, 200);
        long loppuAika = System.currentTimeMillis();

        System.out.println("Ensin tehdyllä totetuksella kesti taas: " +
                           (loppuAika - alkuAika) + " millisekuntia");

}
```

Ohjelma piirtää nyt kolmiot molemmilla tavoilla ja tulostaa kestot millisekunteina. Kestojen laskeminen on tehty `System.currentTimeMillis`-aliohjelmalla, joka palauttaa millisekunteina ajan, joka on kulunut keskiyöstä 1.1.1970. Vaikka tämä tuntuu melko erikoiselta tavalta ilmoittaa aika, pystyy tällä kuitenkin mittaamaan ajankulua Java-ohjelmissa.

Voit nyt testailla eri tapojen nopeuksia muuttamalla `PIENIN_KOLMIO`-vakion arvoa.

20. Dynaamiset tietorakenteet

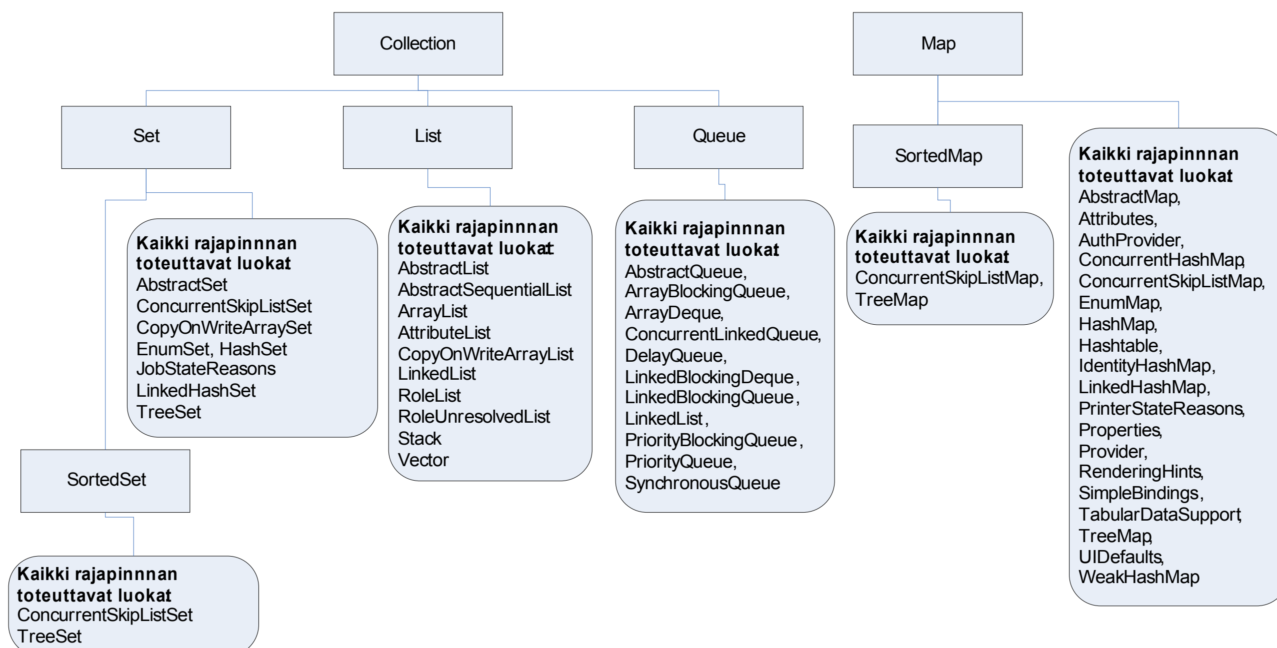
Taulukot tarjoavat meille vielä hyvin rajalliset puitteet ohjelmointiin. Mietitäänpä vaikka tilanne, jossa meidän tarvitsisi laskea käyttäjän syöttämiä lukuja yhteen Käyttäjä saisi syöttää niin monta lukua kuin haluaa ja lopuksi painaa **enter**, jolloin meidän täytyisi laskea ja tulostaa näytölle käyttäjän syöttämien lukujen summan. Minne talletamme käyttäjän syöttämät luvut? Taulukkoon? Minkä kokoisen taulukon luomme? 10 alkiota? 100? vai jopa 1000? Vaikka tekisimme kuinka ison taulukon, aina käyttäjä voi teoriassa syöttää enemmän lukuja ja luvut eivät mahdu taulukkoon. Toisaalta jos teemme 1000 kokoisen taulukon ja käyttäjä syöttääkin vain muutaman luvun, varaamme kohtuuttomasti koneen muistia. Tällaisia tilanteita varten Java-kieli on pullollaan dynaamisia tietorakenteita. Niiden koko kasvaa sitä mukaan kun alkioita lisätään. Dynaamisia tietorakenteita ovat muun muassa listat, puut, vektorit, pinot ym. Niiden käyttäminen ja rakenne eroaa huomattavasti toisistaan.

20.1 Rajapinnat

Javassa on olemassa *rajapintoja* (**interface**) helpottamaan ohjelmointia. Rajapinnassa määritellään tietyt metodit ja kaikkien luokkien, jotka toteuttavat (**implement**) tämän rajapinnan täytyy sisältää samat metodit. Rajapintojen hienous on se, että voimme käyttää samoja metodeja kaikkiin saman rajapinnan olioihin. Meillä voisi olla, vaikka rajapinta `Muodot.` Nyt voisimme tehdä luokat `Ympyra`, `Kolmio` ja `Suorakulmio`, jotka kaikki toteuttaisivat `Muodot`-rajapinnan. Voisimme nyt luoda esimerkiksi `Muodot`-tyyppisen taulukon, johon voisi nyt tallentaa kaikkia `Muodot`-rajapinnan toteutettavien luokkien olioita. Jos `Muodot`-rajapinnassa olisi määritelty metodi `varita()`, voisimme nyt värittää silmukassa kerralla taulukollisen ympyröitä, kolmioita ja suorakulmioita samalla metodilla.

Kaikki Javan tietorakenteet toteuttavat jonkun rajapinnan. Javassa tietorakenteet on jaettu kahteen rajapintaan: `Collection` ja `Map`. Lisäksi molemmat sisältävät *alirajapintoja* (**subinterface**). Alla olevassa kuvassa on kaikki Javan dynaamiset tietorakenteet. Rajapinnat on esitetty suorakulmioina ja ne toteuttavat luokat eli tietorakenteet on listattu pyöristetyissä suorakulmioissa. Valmiita tietorakenteita on siis Javassa aika kasa, joten ennen oman tietorakenteen tekemistä kannattaa tutustua niihin. Oman tietorakenteen tekeminen onkin jo sitten Ohjelmointi 2-kurssin asiaa.

Kuvasta puuttuu osa alirajapinnoista, koska niitäkin on aika järkyttävä määrä.



Kuva 31: Java 6:n tietorakenteet

20.2 ArrayList

Tutustutaan seuraavaksi yhteen Javan dynaamisista tietorakenteista, `ArrayList`-luokkaan. `ArrayList` muistuttaa jonkin verran taulukkoa.

`ArrayList`-olioon ja muihin dynaamisiin tietorakenteisiin voi tallentaa mitä olioita tahansa, mutta alkeistietotyyppien tallentaminen ei niihin onnistu. Alkeistietotyyppejä voidaan kyllä kääriä (**wrap**) olioihin kääreluokilla. Tätä kannattaa kuitenkin mahdollisuuksien mukaan välttää, koska muutamien kymmenien bittien kokoisten alkeistietotyyppien kääriminen olioksi kasvattaa muuttajan kokoa huomattavasti. Alkeistietotyyppejä vastaavat kääreluokat on kerrottu olioiden yhteydessä. Voit kerrata ne täältä: 9.7.Tyyppimuunnokset.

Kääreluokan muodostaminen onnistuu melko yksinkertaisesti antamalla luokan konstruktorille parametrina haluamamme arvo. Esimerkiksi `int`-tyyppisen luvun 4 kääriminen onnistuisi seuraavasti.

```
Integer intOlio = new Integer(4);
```

Tai vastaavasti muuttujaa käyttämällä:

```
int luku = 4;
Integer intOlio = new Integer(luku);
```

20.2.1 Tietorakenteen määrittäminen

Dynaamisen tietorakenteen määrittämisen syntaksi poikkeaa hieman tavallisen olion määrittelystä. Yleisessä muodossa määrittäminen menee seuraavasti:

```
TietorakenneLuokanNimi<TallettavienOlioidenLuokanNimi> rakenteenNimi =
         new TietorakenneLuokanNimi<TallettavienOlioidenLuokanNimi>();
```

Esimerkiksi elokuvia tallettava `ArrayList` voitaisiin määritellä seuraavasti:

```
ArrayList<String> elokuvat = new ArrayList<String>();
```

20.2.2 Peruskäyttö

Alkioiden lisääminen onnistuu `ArrayList`-olioon, ja itse asiassa kaikkiin `Collection`-rajapinnan toteuttavien luokkien olioihin, `add`-metodilla. `add`-metodi lisää alkion aina tietorakenteen perällä. Kun indeksointi alkaa jälleen nollasta, niin ensimmäinen lisätty alkio löytyy siis indeksistä 0, seuraava 1 jne. Elokuvia voitaisiin nyt lisätä seuraavasti:

```
elokuvat.add("Casablanca");
elokuvat.add("Star Wars");
```

Poistaminen onnistuu sen sijaan `remove`-metodilla. Parametriksi annetaan sen alkion indeksi, joka halutaan poistaa. "Casablanca"-merkkijonon poistaminen onnistuisi siis seuraavasti:

```
elokuvat.remove(0);
```

Koska rakenne on dynaaminen, muuttuu siis taulukon alkioiden järjestys lennosta. Nyt "Star Wars"-merkkijono löytyisi indeksistä 0.

Tietorakenteen koon tai oikeammin sanottuna tietorakenteen sisältämien alkioiden lukumäärän saa tietää `size`-metodilla.

```
System.out.println(elokuvat.size()); //tulostaisi nyt 1
elokuvat.add("Full Metal Jacket");
System.out.println(elokuvat.size()); //tulostaisi nyt 2
```

Tiettyyn alkioon pääsee käsiksi `get`-metodilla. Parametrina sille annetaan sen alkion indeksi, joka halutaan käsittelyyn. Metodi siis palauttaa tietyn olion tietyssä indeksissä. Ensimmäisen alkion voisi tulostaa esimerkiksi seuraavaksi:

```
System.out.println(elokuvat.get(0)); //tulostaisi "Star Wars"
```

Näillä metodeilla pärjää jo melko hyvin. Muista metodeista voi lukea luokan dokumentaatiosta.

20.2.3 Lukujen tallentaminen tietorakenteeseen, autoboxing

Valitettavasti Javan valmiisiin tietorakenteisiin ei voida tallentaa `int` ja `double` tyyppisiä muuttujia. Tähän avuksi tulee kääreluokat `Integer` ja `Double`. Kääreluokat löytyvät kohdasta 8.7 Tyyppimuunnokset. Javassa on versiota 1.5 alkaen ollut tekniikka nimeltä **autoboxing**. Tämä tarkoittaa esimerkiksi sitä, että jos `Integer`-tyyppiseen viitteeseen sijoitetaan kokonaisluku, "paketoi" kääntäjä tämän sijoituksen `Integer`-olion sisälle

```
 Integer luku;
 luku = 5;  // olisi sama kuin  luku = new Integer(5);
```

Kääntäen jos Integer-olio sijoitetaan int tyypin muuttujaan, kääntäjä "kaivaa paketista" luvun arvon

```
int i;
i = luku;  // kääntyy kuten   i = luku.intValue();
```

Tätä ominaisuutta käyttäen voidaan muodollisesti tallentaa myös perustyyppejä:

```
ArrayList<Integer> luvut = new ArrayList<Integer>();
luvut.add(3);          // sama kuin luvut.add(new Integer(3));
int i = luvut.get(0);  // sama kuin int i = luvut.get(0).intValue();
```

Autoboxing luo helposti ohjelmoijan huomaamatta uusia olioita ja siksi ominaisuuden kanssa on oltava varovainen jos suoritusteho ja muistinkäyttö ovat etusijalla.

21. Esimerkki: Hirsipuupeli

Tähän mennessä olemme oikeastaan oppineet ohjelmoinnin perusrakenteet. Kokeillaan seuraavaksi tehdä hieman laajempi esimerkki, jossa yhdistellään opittuja taitoja.

Hirsipuu on varmasti monille tuttu peli. Siinä yritetään saada selville sana, arvaamalla mitä kirjaimia se sisältää. Kirjaimia saa arvata väärin vain tietyn määrän. Lisäksi jokaisella väärällä arvauksella piirretään pala hirsipuuta ja hirressä roikkuvaa tikku-ukkoa. Jos sana ei selviä sallittujen väärien arvausten rajoissa, häviää pelin.

21.1 Simppeli versio

Tehdään aluksi yksinkertainen versio, jossa ei vielä piirretä hirsipuuta ja ukkoa, vaan väärien arvauksien lukumäärä vain kerrotaan käyttäjälle. Ohjelman tulisi toimia seuraavasti:

```
Hirsipuu-peli
=============


Sana: _ _ _ _ _
Anna kirjain >k[ret]
Annoit kirjaimen k

Sana: k _ _ _ _
Anna kirjain >ö[ret]
Annoit kirjaimen ö
Virheitä: 1/6
Vääriä kirjaimia: ö

Sana: k _ _ _ _
Anna kirjain >i[ret]
Annoit kirjaimen i

Sana: k i _ _ _
Anna kirjain >a[ret]
Annoit kirjaimen a

Sana: k i _ _ a
Anna kirjain >p[ret]
Annoit kirjaimen p
Virheitä: 2/6
Vääriä kirjaimia: ö p

Sana: k i _ _ a
Anna kirjain >s[ret]
Annoit kirjaimen s
Voitit!
Sana: kissa
```

Yksinkertainen algoritmi ohjelman toiminnasta voisi mennä jotenkin näin:

```
1. Tulosta sanan arvatut kirjaimet ja arvaamattomien kirjainten kohdalla alaviiva.
2. Kysy käyttäjältä seuraavaa kirjainarvausta.
  2.1 Jos kirjaimia löytyi päivitä arvattavaa sanaa
  2.2 Muuten vähennä arvauksien määrää
     2.2.1 Jos arvauksia ei ole enää jäljellä lopeta peli häviöön.
3. Jos kaikki kirjaimet on arvattu, lopeta peli voittoon.
4. Muuten jatka kohdasta 1.
```

Varsinainen ohjelman runko kannattaa toteuttaa ikuisella silmukalla, josta poistutaan sitten `break`-lauseilla. Ohjelmassa täytyy olla yksi dynaaminen merkkijono, jolla pidetään kirjaa mitä kirjaimia käyttäjä on arvannut ja mitä ei. Lisäksi oikeaa sanaa pidetään jatkuvasti muistissa muuttumattomassa merkkijonossa. Toimiva ohjelma voisi olla seuraavan kaltainen.

```
import fi.jyu.mit.ohj2.Syotto;

/**
 * Ohjelmalla pelataan Hirsipuu-peliä
 *
```

```
 * @author vesal
 */
public class Hirsipuu {

    /**
     * Tulostaa ohjelman logon
     */
    public static void tulostaLogo() {
        System.out.println("Hirsipuu-peli");
        System.out.println("=============");
        System.out.println();
    }


    /**
     * Luo sanasta jonon, jossa jokaisen merkin kohdalla on _
     * @param sana josta jono luodaan
     * @return sanan pituuden verran alleviivoja
     */
    public static StringBuilder luoTulosjono(String sana) {
        StringBuilder tulos =  new StringBuilder(sana);
        for (int i=0; i<tulos.length(); i++) {
            tulos.setCharAt(i,'_');
        }
        return tulos;
    }


    /**
     * Palauttaa jono niin että joka toinen paikka on tyhjä
     * Esim: k_ss_   => k _ s s _
     * @param jono
     * @return jono harvana
     * @example
     * <pre name="test">
     *   harvakseen("") === "";
     *   harvakseen("k") === "k";
     *   harvakseen("ki") === "k i";
     *   harvakseen("kissa") === "k i s s a";
     * </pre>
     */
    public static String harvakseen(String jono) {
        StringBuilder tulos = new StringBuilder();
        String vali = "";
        for (int i=0; i<jono.length(); i++) {
            tulos.append(vali + jono.charAt(i));
            vali = " ";
        }
        return tulos.toString();
    }


    /**
     * Palauttaa jonon niin että joka toinen paikka on tyhjä
     * Esim: k_ss_   => k _ s s _
     * @param jono
     * @return jono harvana
     * @example
     * <pre name="test">
     *   harvakseen(new StringBuilder("kissa")) === "k i s s a";
     * </pre>
     */
    public static String harvakseen(StringBuilder jono) {
        return harvakseen(jono.toString());
    }


    /**
     * Tutkii montako kertaa merkki löytyy sanasta.  Samalla
     * löytyneiden merkkien kohdalle tulos-jonoon merkitään ko. merkki.
     * Jos merkki jo on tulos-jonossa, se tulkitaan vääräksi.
     * @param sana   mistä merkkiä etsitään
     * @param merkki etsittävä kirjain
     * @param tulos  jono johon oikeat merkitään
     * @return oikeiden merkkien määrä
     * @example
     * <pre name="test">
     *   String sana = "kissa";
     *   StringBuilder tulos = luoTulosjono(sana);
     *   tutkiOikeat(sana,'z',tulos) === 0;
     *   tutkiOikeat(sana,'k',tulos) === 1;
```

```
     *   tulos.toString() === "k____";
     *   tutkiOikeat(sana,'k',tulos) === 0;
     *   tulos.toString() === "k____";
     *   tutkiOikeat(sana,'i',tulos) === 1;
     *   tulos.toString() === "ki___";
     * </pre>
     */
    public static int tutkiOikeat(String sana,char merkki, StringBuilder tulos) {
        int pituus = Math.min(sana.length(), tulos.length());
        int lkm = 0;
        for (int i=0; i<pituus; i++) {
            if ( merkki != sana.charAt(i) ) continue;
            if ( merkki == tulos.charAt(i)) continue;
            lkm++;
            tulos.setCharAt(i,merkki);
        }

        return lkm;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        final int MAXVAARIA = 6;
        String sana = "kissa"; // TODO  arvo tähän sana
        String vaaria = ""; // Sisältää ne väärät arvaukset
        int oikeita = 0;
        StringBuilder tulos = luoTulosjono(sana);

        tulostaLogo();

        while ( true ) {
            System.out.println();
            System.out.println("Sana: "+harvakseen(tulos));
            String syote = Syotto.kysy("Anna kirjain");
            //Tässä pitäisi tarkastaa, ettei käyttäjä paina pelkkää enteriä.
            char c = syote.charAt(0);
            System.out.println("Annoit kirjaimen " + c);
            int lkm = tutkiOikeat(sana,c,tulos);
            if ( lkm == 0 ) {
                vaaria += c;
                System.out.printf("Virheitä: %d/%d%n", vaaria.length(),MAXVAARIA);
                System.out.println("Vääriä kirjaimia: " + harvakseen(vaaria));
                if ( vaaria.length() >= MAXVAARIA) {
                    System.out.println("Hävisit!");
                    break;
                }
            }
            oikeita += lkm; // ei haittaa virheenkään
            if ( oikeita >= sana.length() ) {
                System.out.println("Voitit!");
                break;
            }

        }

        System.out.println("Sana: " + sana);
    }
}
```

Ohjelma näyttää aluksi pitkältä ja monimutkaiselta, mutta tutustutaan siihen paloittain. Oikeasti ohjelma ei edes ole syntynyt noin, vaan ensin on hahmoteltu pääohjelmaa, sitten toteutettu ja testattu aliohjelmia ja tätä kierrosta on jatkettu kunnes ohjelma on saanut yllä olevan muodon. Aloitetaan tutustuminen samasta paikasta, josta ohjelman suoritus alkaa myös oikeasti, eli pääohjelmasta.

```
final int MAXVAARIA = 6;
String sana = "kissa"; // TODO  arvo tähän sana
String vaaria = ""; // Sisältää ne väärät arvaukset
int oikeita = 0;
```

Pääohjelman alussa alustetaan joukko muuttujia. Vakio `MAXVAARIA` sisältää sallittujen väärien arvausten lukumäärän. Muuttuja `sana` on sana jota koitetaan arvata. Merkkijono `vaaria` on tarkoitettu väärien arvausten talletukseen. Kokonaislukumuuttujaan `oikeita` tallennetaan kuinka

monta oikeaa kirjainta käyttäjä on arvannut. Tällä muuttujalla kontrolloidaan ohjelman lopetusta siinä tapauksessa, että kaikki kirjaimet on arvattu.

```
StringBuilder tulos = luoTulosjono(sana);
```

Yllä olevalla rivillä luodaan `StringBuilder`-olio `tulos`. Aliohjelma `luoTulosjono` palauttaa `StringBuilder`-olion, jossa parametrina saatu sana on muutettu merkkijonoksi, joka sisältää saman verran pelkkiä alaviivoja. Muuttujalla `tulos` kontrolloidaan mitä sanoja käyttäjä on jo arvannut. Alaviivoja korvataan siis arvattavan sanan kirjaimilla sitä mukaan kun käyttäjä niitä arvailee.

```
tulostaLogo();
```

Yllä oleva aliohjelma tulostaa pelin logon.

Seuraavaksi mennään ikuisen silmukan sisälle, jossa varsinainen ohjelman toiminnallisuus tapahtuu.

```
System.out.println("Sana: "+harvakseen(tulos));
```

Yllä olevalla rivillä tulostetaan sana jota koitetaan arvata, mutta tietenkin niin, että vain jo arvatut kirjaimet näkyvät. Muuttuja tulos sisälsi siis arvattavan sanan muodossa, jossa arvaamattomat merkit ovat alaviivoja. Aliohjelma `harvakseen` palauttaa parametrina saamansa merkkijonon (kelpuuttaa `String` ja `StringBuilder` tyypit) muodossa, jossa merkkien välissä on yksi välilyönti. Sitä tarvitaan siis ainoastaan nätimpää muotoilua varten.

```
String syote = Syotto.kysy("Anna kirjain");

char c = syote.charAt(0);
```

Yllä olevilla kahdella rivillä kysytään käyttäjältä kirjainta ja tallennetaan syötetty merkki muuttujaan `c`. Ennen kuin mennään `charAt`-metodilla hakemaan syötetyn merkkijonon ensimmäistä kirjainta, täytyisi tarkastaa ettei käyttäjä painanut ainoastaan **Enteriä** (eli syöttänyt tyhjää merkkijonoa). Nyt ohjelma kaatuisi tässä tapauksessa, mutta tilan säästämiseksi tämä tarkastus on jätetty pois.

```
System.out.println("Annoit kirjaimen " + c);
```

Yllä oleva rivi ainoastaan tulostaa käyttäjälle mitä kirjainta hän on painanut.

```
int lkm = tutkiOikeat(sana,c,tulos);
```

Seuraavaksi tarkastetaan oliko sanassa käyttäjän arvaamia kirjaimia. Tämä tehdään `tutkiOikeat`-aliohjelmalla. Aliohjelma palauttaa sanasta löytyneiden kirjainten lukumäärän ja päivittää samalla `tulos`-muuttujaa. Tarkastellaan seuraavaksi `tutkiOikeat`-aliohjelmaa tarkemmin.

Aliohjelma saa parametrikseen oikean sanan, käyttäjän syöttämän kirjaimen sekä `StringBuilder`-tyyppisen muuttujan `tulos`, josta selviää siis pelin tämänhetkinen tilanne.

```
int lkm = 0;
```

Aluksi alustetaan löydettyjen kirjainten lukumäärä nollaksi. Seuraavaksi mennään `for`-silmukan sisään, jolla käydään siis kaikki oikean sanan kirjaimet läpi.

```
if ( merkki != sana.charAt(i) ) continue;
```

Jos käyttäjän syöttämä merkki oli eri kuin merkkijonon `sana` merkki indeksissä `i`, ei oikeaa merkkiä löytynyt tästä kohtaa sanaa ja voidaan siirtyä tarkastelemaan seuraavaa merkkiä.

`continue`-lauseella indeksiä siis kasvatetaan yhdellä ja silmukan suoritusta jatketaan alusta.

```
if ( merkki == tulos.charAt(i)) continue;
```

Silmukan suoritusta voidaan jatkaa alusta myös siinä tapauksessa, että kirjain on jo arvattu. Tämä tarkastetaan yllä olevalla rivillä. Muissa tapauksissa käyttäjä onnistui arvaamaan yhden kirjaimen sanasta, joten kasvatetaan arvattujen kirjainten lukumäärää yhdellä:

```
lkm++;
```

Lisäksi meidän täytyy muistaa päivittää `tulos`-jonoa. Asetetaan siihen nyt alaviivan paikalle arvattu merkki `setCharAt`-metodilla:

```
tulos.setCharAt(i,merkki);
```

Tässä `for`-rakenteessa käytettiin nyt hieman totutusta poikkeavaa rakennetta. `if`-lauseilla tarkastettiin, että EIKÖ löytynyt uutta kirjainta ja jatkettiin sitten silmukan suoritusta alusta. Käyttäjä arvasi oikean kirjaimen, jos molemmat `if`-lauseet olivat epätosia ja tällöin tehtiin tarvittavat toimenpiteet. Vastaavasti voitaisiin tehdä yksi `if`-lause, jolla tarkastetaan arvasiko käyttäjä vielä arvaamattoman oikean kirjaimen. Tarvittavat toimenpiteet tehdään sitten tämän `if`-lauseen sisällä:

```
for (int i=0; i<pituus; i++) {
   if ( merkki == sana.charAt(i) && merkki != tulos.charAt(i)) {
      lkm++;
      tulos.setCharAt(i,merkki);
   }
}
```

Yllä oleva rakenne toimisi täysin vastaavasti kuin esimerkissä oleva.

Aliohjelmassa `tutkiOikeat` täytyy muistaa vielä palauttaa löytyneiden kirjainten lukumäärä:

```
return lkm;
```

Seuraavaksi palataan takaisin pääohjelmaan.

```
if ( lkm == 0 ) {
```

Jos `tutkiOikeat`-aliohjelma palautti 0, ei sanasta löytynyt yhtään käyttäjän syöttämää kirjainta. Arvaus meni siis metsään, ja tarvittavat toimenpiteet tehdään yllä olevalla rivillä alkavan `if`-lauseen sisällä.

```
vaaria += c;
```

Lisätään käyttäjän arvaama merkki `vaaria`-merkkijonoon, jossa siis pidettiin tallessa käyttäjän syöttämiä merkkejä, joita sanasta ei löydy. Seuraavat kaksi riviä tulostavat käyttäjälle palautetta jäljellä olevien arvausten määrästä ja jo arvatuista merkeistä.

```
if ( vaaria.length() >= MAXVAARIA ) {
       System.out.println("Hävisit!");
       break;
}
```

Yllä olevassa koodinpätkässä tarkastetaan onko vääriä arvauksia jo enemmän kuin pelissä saa olla. Jos on, niin tulostetaan että peli päättyi häviöön ja poistutaan ikuisesta silmukasta `break`-lauseella, jolloin ohjelman suoritus myös loppuu.

```
oikeita += lkm;
```

Jos käyttäjä arvasi oikein sanasta löytyvän kirjaimen tai kirjaimia, jatkuu ohjelman suoritus yllä

olevalta riviltä. Siinä lisätään muuttujaan `oikeita` arvattavasta sanasta löytyneiden kirjainten lukumäärä. Tuo rivihän suoritetaan usein myös siinä tapauksessa, että kirjaimia ei löytynyt. Tämä ei kuitenkaan haittaa, koska silloin muuttujaan `oikeita` lisätään ainoastaan luku 0.

```
if ( oikeita >= sana.length() ) {
       System.out.println("Voitit!");
       break;
}
```

Yllä olevalla koodinpätkällä tarkastetaan onko kaikki kirjaimet jo arvattu. Tämä selviää vertaamalla muuttujaa `oikeita` muuttujan `sana` pituuteen. Jos muuttuja `oikeita` oli suurempi tai yhtä suuri kuin muuttujan `sana` pituus, sana on arvattu ja voidaan tehdä tarvittavat tulostukset ja poistua silmukasta ja samalla koko ohjelmasta `break`-lauseella.

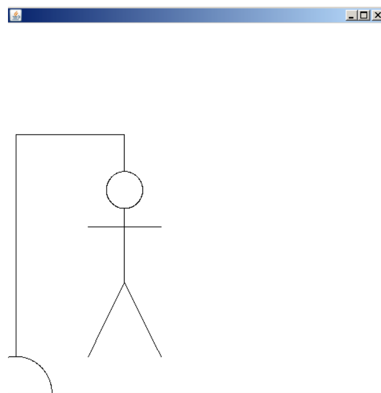
Nyt meillä on kasassa simppeli versio hirsipuu-pelistä. Seuraavaksi lisätään ohjelmaan hirren piirto.

21.2 EasyWindow-luokasta Window-luokkaan

Kurssin alkupuoliskolla käytettiin piirtämiseen `EasyWindow`-luokkaa, joka teki piirtämisestä hieman yksinkertaisempaa. Tavallisen `Window`-luokan ikkunan käyttäminen ei oikeastaan ole juurikaan vaikeampaa. Ideana on, että jokainen muoto tai kuvio on oma olionsa, joka sitten lisätään `Window`-luokan olioon eli ikkunaan. Kuvioita ei siis enää piirretä luokasta löytyvillä metodeilla (`addCircle` jne..), vaan jokainen kuvio on oma olionsa, joka lisätään ikkunaan samalla `add`-metodilla. `Window`-olioon voi lisätä kaikkia `Drawable`-rajapinnan toteuttavien luokkien olioita. Seuraavana joukko valmiita luokkia jotka toteuttavat `Drawable`-rajapinnan:

Axis	FunctionMapR2R
BasicDrawableCollection	FunctionMapRR
BasicShape	Line
CarSample.Car	Marker
Circle	Polygon
DrawableCollection	Polyline
FillPolygon	UkkoSample.Ukko

21.2.1 Esimerkki: Hirsipuun piirto



Kuva 32: Hirsipuun piirtäminen

Piirretään esimerkkinä `Window`-luokasta yllä olevan kuvan kaltainen hirsipuu, jota käytetään sitten varsinaisessa hirsipuu-pelissä.

```
import fi.jyu.mit.graphics.Circle;
import fi.jyu.mit.graphics.Line;
import fi.jyu.mit.graphics.Window;

public class HirsipuunPiirtoSimppeli {

	/**
	 * Ohjelma piirtää hirsipuun.
	 * @param args
	 */
	public static void main(String[] args) {
		Window ikkuna = new Window();
		ikkuna.scale(0,0,10,10);

		//Luodaan hirsipuun osat
		Circle hirsipuunJalka = new Circle(0,0,1);
		Line hirsipuunRunko = new Line(0,1,0,7);
		Line hirsipuunVarsi = new Line(0,7,3,7);
		Line hirsipuunKoysi = new Line(3,7,3,6);
		Circle paa = new Circle(3,5.5,0.5);
		Line vartalo = new Line(3,5,3,3);
		Line vasenJalka = new Line(3,3,2,1);
		Line oikeaJalka = new Line(3,3,4,1);
		Line kadet = new Line(2,4.5,4,4.5);

		//Lisätään osat ikkunaan
		ikkuna.add(hirsipuunJalka);
		ikkuna.add(hirsipuunRunko);
		ikkuna.add(hirsipuunVarsi);
		ikkuna.add(hirsipuunKoysi);
		ikkuna.add(paa);
		ikkuna.add(vartalo);
		ikkuna.add(vasenJalka);
		ikkuna.add(oikeaJalka);
		ikkuna.add(kadet);

		ikkuna.showWindow();

	}

}
```

Hirsipuu luodaan ympyröistä ja viivoista. Piirtämiseen on käytetty Jyväskylän Yliopiston Graphics-kirjastosta löytyviä `Line`- ja `Circle`-luokkia. Edellisten luokkien lisäksi myös `Window`-luokka täytyy muistaa lisätä ohjelmaan `import`-lauseilla (**import** = tuonti) koodin alussa.

Pääohjelman toisella rivillä käytetty `scale`-metodi skaalaa ikkunan haluamaksemme. Sen neljällä parametrilla ilmoitetaan ikkunan reuna-arvot. Ensimmäinen parametri tarkoittaa ikkunan vasemman reunan arvoa, toinen pohjan arvoa, kolmas oikean reunan arvoa ja neljäs parametri yläreunan arvoa. Yleisessä muodossa metodi on siis seuraava;

```
scale(double left, double bottom, double right, double top);
```

Toisin sanoen alla oleva lause asettaa siis ikkunan origoksi (0,0) vasemman alareunan ja oikeaksi yläreunaksi pisteen (10,10)

```
ikkuna.scale(0,0,10,10);
```

Omia skaalauksia tekemällä voidaan piirtämistä usein helpottaa.

Metodista on olemassa myös kolmeparametrinen versio, jolla ikkunaa voidaan skaalata venyttämällä sitä. Löydät sen kirjaston dokumentaatiosta.

Tämän esimerkin tavalla tehtynä yksinkertaisesta ohjelmasta tulee melko pitkä. Muoto-oliot voidaankin luoda myös lisäämisen yhteydessä, jolloin päästään huomattavasti lyhyempään lopputulokseen.

```
import fi.jyu.mit.graphics.Circle;
import fi.jyu.mit.graphics.Line;
```

```
import fi.jyu.mit.graphics.Window;

public class HirsipuunPiirtoSimppeli {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Window ikkuna = new Window();
		ikkuna.scale(0,0,10,10);
		ikkuna.add(new Circle(0,0,1));
		ikkuna.add(new Line(0,1,0,7));
		ikkuna.add(new Line(0,7,3,7));
		ikkuna.add(new Line(3,7,3,6));
		ikkuna.add(new Circle(3,5.5,0.5));
		ikkuna.add(new Line(3,5,3,3));
		ikkuna.add(new Line(3,3,2,1));
		ikkuna.add(new Line(3,3,4,1));
		ikkuna.add(new Line(2,4.5,4,4.5));
		ikkuna.showWindow();
	}
}
```

Jos tarvitsisimme myöhemmin viitettä tiettyyn olioon, saataisiin se talteen myös tässä esimerkissä, sillä `add`-metodi palauttaa viitteen lisättyyn olioon. Esimerkiksi ukon päähän saisimme viitteen seuraavasti:

```
Circle paa = ikkuna.add(new Circle(3,5.5,0.5));
```

Jos meidän täytyisi tehdä hirsipuun osille muutoksia, kannattaisi ne tallentaa samantien taulukkoon. Tämä voisi olla järkevää muutenkin, sillä hirsipuun osat muodostavat selvästi oman kokonaisuutensa. Ongelmana on, että taulukkoon pitäisi saada talletettua kahdentyyppisiä olioita. Nyt apuun tulevat jälleen rajapinnat. Koska sekä luokka `Circle` että luokka `Line` toteuttavat `Drawable`-rajapinnan, voimme luoda taulukon `Drawable`-tyyppiseksi. `Drawable`-tyyppiseen taulukkoon voimme nyt tallentaa `Circle`- ja `Line`-tyyppisiä olioita, sekä tietenkin myös kaikkia muita `Drawable`-rajapinnan toteuttavien luokkien olioita.

```
Drawable[] hirrenOsat = {
      new Circle(0,0,1),
      new Line(0,1,0,7),
      new Line(0,7,3,7),
      new Line(3,7,3,6),
      new Circle(3,5.5,0.5),
      new Line(3,5,3,3),
      new Line(3,3,2,1),
      new Line(3,3,4,1),
      new Line(2,4.5,4,4.5)
};
```

Toinen vaihtoehto olisi luoda taulukko `BasicShape`-tyyppiseksi. `BasicShape` on luokka jonka sekä `Circle`, että `Line`-luokka *perivät*.

periytyminen (**inheritance**) = Perimällä toisen luokan, saa perivä luokka käyttöönsä kaikki perityn luokan ominaisuudet (attribuutit) ja toiminnot (metodit). Perivää luokkaa sanotaan aliluokaksi ja perittyä luokkaa yliluokaksi. Aliluokalla voi olla lisäksi omia ominaisuuksia ja toimintoja, joita yliluokasta ei löydy. Yliopiston tietojärjestelmässä voisi olla esimerkiksi luokka `Henkilo`, jonka sitten luokat `Opettaja` ja `Oppilas` perivät. `Henkilo`-luokasta löytyisi nyt kaikkia ominaisuuksia ja toimintoja, joita kaikilla henkilöillä on. Siellä voisi olla esimerkiksi attribuutit nimi ja osoite. Kaikilla yliopiston henkilöillä ei kuitenkaan ole esimerkiksi työhuonetta. Työhuoneen numero löytyisikin `Opettaja`-luokasta. Periminen liittyy hyvin läheisesti rajapintoihin. C++-kielessä ei esimerkiksi ole rajapintoja, vaan samat asiat hoidetaan perinnällä. C++:sta löytyvä *moniperintä* ei ole mahdollista Javassa, jota korvaamaan on sitten kehitetty rajapinnat. Luokat voivat olla myös abstrakteja (**abstract class**), jolloin jokin ominaisuus jätetään toteuttamatta ja pakotetaan toteuttamaan se perinnän yhteydessä.

Vastaavasti kuten rajapinnoissa, voimme tallentaa taulukkoon kaikkia luokkia, jotka perivät taulukon tyyppinä olevan luokan. Hirsipuun osat voitaisiin siis yhtä hyvin tallentaa `BasicShape`-tyyppiseen taulukkoon.

```
BasicShape[] hirrenOsat = {
       new Circle(0,0,1),
       new Line(0,1,0,7),
       new Line(0,7,3,7),
       new Line(3,7,3,6),
       new Circle(3,5.5,0.5),
       new Line(3,5,3,3),
       new Line(3,3,2,1),
       new Line(3,3,4,1),
       new Line(2,4.5,4,4.5)
};
```

Huomaa kuitenkin, että `Drawable`-rajapinnasta löytyvät eri metodit kuin `BasicShape`-luokasta. Itse asiassa koska `BasicShape`-luokka toteuttaa `Drawable`-rajapinnan, saamme `BasicShape`-luokkaa käyttämällä käyttöön kaikki `Drawable`-rajapinnan metodit, sekä lisäksi `BasicShape`-luokan metodit.

21.3 Hirsipuun piirtäminen pelissä

Hirsipuun piirtäminen pelissä on lopulta melko simppeliä, kun meillä on hirren osat talletettuna nätisti taulukkoon. Meidän ei tarvitse kuin lisätä pääohjelman alkuun `Window`-ikkuna ja `hirrenOsat`-taulukko. Kun käyttäjä nyt syöttää merkin, jota ei ole arvattavassa sanassa, lisäämme vain `hirrenOsat`-taulukosta alkioita yksi kerrallaan ikkunaan. Pääohjelma muutettaisiin nyt siis alla olevaan muotoon, jossa sininen vahvennettu teksti on uutta koodia.

```
    /**
     * @param args
     */
    public static void main(String[] args) {
        final int MAXVAARIA = 6; //Ei voi olla yli 9
        String sana = "kissa"; // TODO  arvo tähän sana
        String vaaria = ""; // Sisältää ne väärät arvaukset
        int oikeita = 0;
        StringBuilder tulos = luoTulosjono(sana);


        //HIRREN PIIRTÄMISEEN LIITTYVÄT JUTUT
        Window ikkuna = new Window();
        ikkuna.scale(0,0,10,10);

        BasicShape[] hirrenOsat = {
            new Circle(0,0,1),
            new Line(0,1,0,7),
            new Line(0,7,3,7),
            new Line(3,7,3,6),
            new Circle(3,5.5,0.5),
            new Line(3,5,3,3),
            new Line(3,3,2,1),
            new Line(3,3,4,1),
            new Line(2,4.5,4,4.5)
         };

        int aluksiPiirretaan = Math.max(hirrenOsat.length - MAXVAARIA, 0);

        //Piirtää hirsipuuta valmiiksi aluksi
        for (int i=0; i < aluksiPiirretaan; i++) {
              ikkuna.add(hirrenOsat[i]);
        }

        tulostaLogo();

        while ( true ) {
            System.out.println();
            System.out.println("Sana: "+harvakseen(tulos));
            String syote = Syotto.kysy("Anna kirjain");
            //Tässä pitäisi tarkastaa, ettei käyttäjä paina pelkkää enteriä.
            char c = syote.charAt(0);
            System.out.println("Annoit kirjaimen " + c);
```

```
            int lkm = tutkiOikeat(sana,c,tulos);
            if ( lkm == 0 ) {
                vaaria += c;
                int seuraavaOsa = aluksiPiirretaan + vaaria.length() - 1;
                if(seuraavaOsa < hirrenOsat.length) ikkuna.add(hirrenOsat[seuraavaOsa]);
                System.out.printf("Virheitä: %d/%d%n", vaaria.length(),MAXVAARIA);
                System.out.println("Vääriä kirjaimia: " + harvakseen(vaaria));
                if ( vaaria.length() >= MAXVAARIA) {
                    System.out.println("Hävisit!");
                    break;
                }
            }
            oikeita += lkm; // ei haittaa virheenkään
            if ( oikeita >= sana.length() ) {
                System.out.println("Voitit!");
                break;
            }

        }

        System.out.println("Sana: " + sana);

    }
```

Lisäksi täytyy muistaa lisätä koodin alkuun tarvittavat `import`-lauseet:

```
import fi.jyu.mit.graphics.BasicShape;
import fi.jyu.mit.graphics.Circle;
import fi.jyu.mit.graphics.Drawable;
import fi.jyu.mit.graphics.Line;
import fi.jyu.mit.graphics.Window;
```

Tutkitaan seuraavaksi tarkemmin mitä uutta koodiin tuli.

```
Window ikkuna = new Window();
ikkuna.scale(0,0,10,10);

BasicShape[] hirrenOsat = {
       new Circle(0,0,1),
       new Line(0,1,0,7),
       new Line(0,7,3,7),
       new Line(3,7,3,6),
       new Circle(3,5.5,0.5),
       new Line(3,5,3,3),
       new Line(3,3,2,1),
       new Line(3,3,4,1),
       new Line(2,4.5,4,4.5)
};
```

Yllä olevat rivit ovat tuttuja jo aiemmista esimerkeistä. Luomme siis `Window`-tyyppisen ikkunan, joka skaalataan. Tämän jälkeen alustetaan `hirrenOsat`-taulukko, joka sisältää kaikki hirsipuun piirtoon tarvittavat palaset.

```
   int aluksiPiirretaan = Math.max(hirrenOsat.length - MAXVAARIA, 0);
```

Yllä olevalla rivillä lasketaan montako osaa hirsipuuta täytyy piirtää etukäteen, että hirsipuu tulisi valmiiksi. Hirsipuussa on nyt yhdeksän osaa, ja jos vääriä vastauksia sallitaan kuusi, täytyy aluksi piirtää kolme osaa valmiiksi. `Math.max`-metodia kutsumalla varmistetaan, ettei muuttuja `aluksiPiirretaan` saa koskaan negatiivista arvoa, joka kaataisi ohjelman myöhemmin. `Math.max`-metodi palauttaa siis parametreinaan saamistaan luvuista suuremman.

```
for (int i=0; i < aluksiPiirretaan; i++) {
       ikkuna.add(hirrenOsat[i]);
}
```

Yllä olevalla silmukalla lisätään ikkunaan niin monta osaa kun niitä aluksi piti piirtää.

```
int seuraavaOsa = aluksiPiirretaan + vaaria.length() - 1;
```

Yllä lasketaan seuraavan piirrettävän osan indeksi. Indeksi saadaan laskettua `vaaria`-taulukon

pituudesta. Sehän kertoo meille kuinka monta väärää kirjainta on jo syötetty. Täytyy vain muistaa ottaa huomioon jo alkuvalmisteluissa ikkunaan lisätyt hirren osat. Lisäksi, koska taulukoiden indeksointi alkaa nollasta, täytyy summasta muistaa vähentää luku 1.

```
if (seuraavaOsa < hirrenOsat.length) ikkuna.add(hirrenOsat[seuraavaOsa]);
```

Aluksi varmistetaan `if`-lauseella, ettei vaan viitata `hirrenOsat`-taulukon alkioon, jota ei ole olemassa. Tämä tapahtuisi, jos `MAX_VAARIA` muuttuja olisi suurempi kuin `hirrenOsat`-taulukon pituus. Jos `if`-lauseen ehto oli tosi, voidaan `ikkuna`-olioon turvallisesti lisätä seuraava hirsipuun osa.

Nyt meillä pitäisi olla hieno hirsipuu-ohjelma. Siinä on kuitenkin yksi ongelma. Jos muuttuja `MAXVAARIA` on suurempi kuin `hirrenOsat`-taulukon pituus, ohjelma toimii epätoivotulla tavalla. Hirsipuu tulee siis valmiiksi, jo ennen kuin kaikki väärät arvaukset on käytetty.

Muuta ohjelmaa niin, että se toimii toivotulla tavalla myös, jos MAXVAARIA muuttuja on asetettu suuremmaksi kuin mitä hirsipuussa on osia. Osien piirtäminen on siis aloitettava vasta siinä kohdassa, että hirsipuu tulee valmiiksi täsmälleen silloin kuin käyttäjä syöttää viimeisen väärän kirjaimen, eli juuri ennen pelin päättymistä häviöön.

Oikeasti hirsipuun piirtämisestä kannattaisi tehdä jo olio. Olioiden tekeminen ei kuitenkaan kuulu kurssin sisältöön, joten tässä esiteltiin tapa, jolla piirtämisen voi tehdä ilman oman olion tekemistä. Hirsipuupeli on olioilla tehtynä liitteenä monisteen lopussa. Olkoon se jonkin näköinen johdanto tai silta ohjelmointi 2-kurssille.

22. Tiedostot

Muuttujat toimivat tiedon talletuksessa niin kauan, kun ohjelma on käynnissä. Ohjelman suorituksen loputtua muuttujien muistipaikat luovutetaan kuitenkin muiden prosessien käyttöön. Tämän takia muuttujat eivät sovellu sellaisen tiedon talletukseen, jonka pitäisi säilyä kun ohjelma suljetaan. Pitkäaikaiseen tiedon talletukseen soveltuvat hyvin tiedostot ja tietokannat. Tiedostot ovat yksinkertaisempia ja ehkä helpompia käyttää, kun taas tietokannat tarjoavat paljon monipuolisempia ominaisuuksia. Tiedostoihin voidaan tallentaa myös esimerkiksi jotain ohjelman tarvitsemia alkuasetuksia. Hirsipuupelissämme arvataan nyt jatkuvasti kissa-sanaa. Tämä voi olla tylsää, jos haluaa pelata peliä useamman kerran. Järkevämpää olisi, että meillä on tiedostossa sanoja, joista ohjelma arpoo satunnaisen sanan arvattavaksi.

22.1 Tiedostot Ali.jar -kirjaston avulla

Käytetään tiedoston lukemiseen Ali.jar-kirjaston `Tiedosto.lueTiedosto`-aliohjelmaa. Se saa parametrina tiedoston nimen ja lukee sen nimisen tiedoston merkkijonotaulukkoon, jossa jokainen tiedoston rivi on oma alkionsa. Voisimme esimerkiksi tulostaa sanat.txt-nimisen tiedoston seuraavalla ohjelmalla:

```
import fi.jyu.mit.ohj2.Tiedosto;

public class TiedostotSimppeli {

   /**
    * Tulostetaan tiedoston "sanat.txt" sisältö.
    * @param args
    */
   public static void main(String[] args) {
      String[] sanat = Tiedosto.lueTiedosto("sanat.txt");

      for (String sana : sanat) {
         System.out.println(sana);
      }
   }
}
```

Eclipse hakee tiedostoja oletuksena projektin juuresta, kun Eclipsen käynnistää Windowsissa kuvakkeesta. Siksi Eclipse kannattaa käynnistää komentoriviltä tai vaihtaa ohjelman oletushakemistoa ennen ohjelman ajamista:

```
Run/Run Configurations...
valitaan ajettava ohjelma
Arguments -välilehti
Working directory ja ruksi kohtaan Other ja sitten haluttu hakemisto
```

22.2 Sanojen lukeminen tiedostosta hirsipuupelissä

Voisimme nyt muuttaa hirsipuupeliä niin, että se arpoo arvattavan sanan tiedostossa luetelluista sanoista. Sovitaan, että jokainen arvailtava sana on kirjoitettava tiedostoon omalle riville, niin sanojen erottelu sujuu helpommin. Tehdään nyt sanat.txt-niminen tiedosto, jonka sisältö on vaikka seuraava:

```
kissa
koira
kana
kukko
mato
```

Tiedoston lukeminen taulukkoon onnistuu täysin samalla tavalla kuin edellisessä esimerkissä. Enää täytyy hoitaa siis sanan arvonta. Tehdään sitä varten aliohjelma, joka saa parametrina merkkijono taulukon ja palauttaa siitä satunnaisen sanan. Satunnaislukujen arvonta onnistuu Javan `Random`-luokalla.

22.2.1 Luokka: Random

`Random`-luokka ei ole oletuksena käytettävissä, vaan meidän pitää tuoda se jälleen ohjelmaan seuraavalla `import`-lauseella:

```
import java.util.Random;
```

`Random`-luokasta löytyy metodeja joilla voimme arpoa erityyppisiä satunnaislukuja. Arpomista varten meidän täytyy luoda `Random`-olio, jotta metodeja voitaisiin kutsua. `Random`-oliolla on metodi `nextInt`, joka saa parametrikseen kokonaisluvun ja arpoo sitten satunnaisen luvun 0 ja parametrinaan saamansa luvun väliltä niin, että parametrina annettava luku ei enää kuulu arvottaviin lukuihin. Arvottava luku on siis aina puoliavoimella välillä [0,parametri[. Jos haluaisimme arpoa luvun suljetulta väliltä [0,10] täytyisi meidän siis lisätä parametriin luku 1. Huomaa, että kun käsitellään kokonaislukuja, suljettu väli [0,10] on sama asia kuin puoliavoin väli [0,11[. Alla oleva koodinpätkä arpoisi nyt siis luvun 0:n ja 10:n väliltä niin, että luvut 0 ja 10 kuuluvat arvottaviin lukuihin.

```
Random rand = new Random();
int satunnaisluku = rand.nextInt(11);
```

Jos haluaisimme arpoa luvun esimerkiksi suljetulta väliltä [50,99] joudumme käyttämään lisänä aritmeettisia operaatioita. Tämä onnistuu arpomalla luku väliltä [0,50[ja lisäämällä siihen luku 50 kuten alla.

```
Random rand = new Random();
int satunnaisluku = rand.nextInt(51) + 50;
```

22.2.2 Arpomisaliohjelma hirsipuupeliin

Näillä tiedoilla voisimme nyt tehdä tarvitsemamme aliohjelman.

```
/**
  * Funktiolla arvotaan yksi merkkijono taulukosta
  * @param jonot taulukko josta jono arvotaan
  * @return satunnainen jonot-taulukon rivi
  */
public static String arvo(String[] jonot) {
   Random rand = new Random();
   int n = rand.nextInt(jonot.length);
   return jonot[n];
}
```

Aliohjelma saa parametrinaan merkkijonotaulukon ja palauttaa siitä satunnaisen merkkijonon. Koska taulukon nimi on jonot, täytyy meidän arpoa luku väliltä [0,jonot.length-1]. Kokonaisluvuissa tämä on siis sama asia kun arpoa luku puoliavoimelta väliltä [0,jonot.length[. Lopuksi palautetaan merkkijonotaulukon alkio arvotusta kohdasta.

Nyt meidän täytyy enää käyttää aliohjelmaamme hirsipuupelin pääohjelmassa. Pääohjelma muuttuisi nyt seuraavaan muotoon, jossa uudet rivit ovat jälleen vahvennetulla sinisellä:

```
    /**
     * @param args
     */
    public static void main(String[] args) {
        final int MAXVAARIA = 6; //Ei voi olla yli 9
        String sana = "kissa";

        String[] sanat = Tiedosto.lueTiedosto("sanat.txt");
        if ( sanat != null ) sana = arvo(sanat);

        String vaaria = ""; // Sisältää ne väärät arvaukset
        int oikeita = 0;
        StringBuilder tulos = luoTulosjono(sana);
```

```
        //HIRREN PIIRTÄMISEEN LIITTYVÄT JUTUT
        Window ikkuna = new Window();
        ikkuna.scale(0,0,10,10);

        BasicShape[] hirrenOsat = {
            new Circle(0,0,1),
            new Line(0,1,0,7),
            new Line(0,7,3,7),
            new Line(3,7,3,6),
            new Circle(3,5.5,0.5),
            new Line(3,5,3,3),
            new Line(3,3,2,1),
            new Line(3,3,4,1),
            new Line(2,4.5,4,4.5)
         };

        int aluksiPiirretaan = Math.max(hirrenOsat.length - MAXVAARIA, 0);

        //Piirtää hirsipuuta valmiiksi aluksi
        for (int i=0; i < aluksiPiirretaan; i++) {
              ikkuna.add(hirrenOsat[i]);
        }

        tulostaLogo();

        while ( true ) {
            System.out.println();
            System.out.println("Sana: "+harvakseen(tulos));
            String syote = Syotto.kysy("Anna kirjain");
            //Tässä pitäisi tarkastaa, ettei käyttäjä paina pelkkää enteriä.
            char c = syote.charAt(0);
            System.out.println("Annoit kirjaimen " + c);
            int lkm = tutkiOikeat(sana,c,tulos);
            if ( lkm == 0 ) {
                vaaria += c;
                int seuraavaOsa = aluksiPiirretaan + vaaria.length() - 1;
                if(seuraavaOsa < hirrenOsat.length) ikkuna.add(hirrenOsat[seuraavaOsa]);
                System.out.printf("Virheitä: %d/%d%n", vaaria.length(),MAXVAARIA);
                System.out.println("Vääriä kirjaimia: " + harvakseen(vaaria));
                if ( vaaria.length() >= MAXVAARIA) {
                    System.out.println("Hävisit!");
                    break;
                }
            }
            oikeita += lkm; // ei haittaa virheenkään
            if ( oikeita >= sana.length() ) {
                System.out.println("Voitit!");
                break;
            }

        }

        System.out.println("Sana: " + sana);

    }
```

Ohjelmaan jätettiin vielä rivi, jossa muuttujaan `sana` alustetaan merkkijono "kissa". Tämä johtuu siitä, että haluamme ohjelman toimivan, vaikka tiedon lukeminen epäonnistuisi.

Sanojen lukeminen tiedostoista ei lisännyt siis koodia pääohjelmassa kuin kahdella rivillä. Tämä on huolellisesti suunniteltujen aliohjelmien ansiota. Onkin sanottu, että koko pääohjelman kuuluisi mahtua kerralla näytölle. Tätä vaatimusta oma pääohjelmamme ei vielä toteuta.

Voisiko pääohjelmaa vielä lyhentää muuttamalla yhteen liittyviä kokonaisuuksia aliohjelmiksi?

23. Poikkeukset

“If you don’t handle [exceptions], we shut your application down. That dramatically increases the reliability of the system.”
– Anders Hejlsberg

Poikkeus (**exception**) on ohjelman suorituksen aikana ilmenevä ongelma. Jos poikkeusta ei käsitellä, ohjelman suoritus yleensä kaatuu ja konsoliin tulostetaan jokin virheilmoitus. Tässä vaiheessa kurssia näin on varmasti käynyt jo monta kertaa. Poikkeus voi tapahtua jos esimerkiksi yritämme viitata taulukon alkioon, jota ei ole olemassa.

```
int[] taulukko = new int[5];
taulukko[5] = 5;
```

Esimerkiksi yllä oleva koodinpätkä aiheuttaisi `ArrayIndexOutOfBoundsException`-nimisen poikkeuksen. Näitä poikkeuksia tulee aluksi usein silloin, kun taulukoita käsitellään silmukoiden avulla ja silmukan lopetusehto on väärin. Poikkeuksia aiheuttavat myös esimerkiksi jonkun luvun jakaminen nollalla, sekä yritys muuttaa tekstiä sisältävä merkkijono joksikin numeeriseksi tietotyypiksi.

Poikkeuksia voidaan kuitenkin käsitellä hallitusti poikkeustenhallinnan (**exception handling**) avulla. Tällöin poikkeukseen varaudutaan ja ohjelman suoritusta voidaan jatkaa poikkeuksen sattuessa. Poikkeusten hallinta sisältää aina `try`- ja `catch`-lohkon. Lisäksi voidaan käyttää myös `finally`-lohkoa.

Javan poikkeukset ovat olioita. [VES][KOS][DEI]

23.1 try-catch

Ideana `try-catch` -rakenteessa on, että poikkeusalttiit lauseet sijoitetaan `try`-lohkon sisään. Tämän jälkeen `catch`-lohkossa kerrotaan mitä poikkeustilanteessa tehdään. Ennen `catch`-lohkoa täytyy kuitenkin kertoa mitä poikkeuksia yritetään ottaa kiinni (**catch**). Tämä ilmoitetaan sulkeissa `catch`-sanan jälkeen, ennen `catch`-lohkoa aloittavaa aaltosulkua. Yleisessä muodossa `try-catch` rakenne olisi seuraava:

```
try {
   //jotain lauseita mitä koitetaan suorittaa
} catch (PoikkeusLuokanNimi poikkeukselleAnnettavaNimi) {
   //jotain toimenpiteitä mitä tehdään kun poikkeus ilmenee
}
```

`catch`-lohkoon mennään vain siinä tapauksessa, että `try`-lohko aiheuttaa sen tietyn poikkeuksen, jota `catch`-osassa ilmoitetaan otettavan kiinni. Muissa tapauksissa `catch`-lohko ohitetaan. Jos `try`-lohkossa on useita lauseita, `catch`-lohkoon mennään heti ensimmäisen poikkeuksen sattuessa, eikä loppuja lauseita enää suoriteta. Otetaan esimerkiksi nollalla jakaminen. Nollalla jako aiheuttaisi `ArithmeticException`-poikkeuksen.

```
int n1=7,n2=0,n3=4;

try {
   System.out.printf("%d%n",10/n1);
   System.out.printf("%d%n",10/n2);
   System.out.printf("%d%n",10/n3);
} catch (ArithmeticException e) {
   System.out.println("Nollalla jako: " + e.getMessage());
}
```

Yllä olevassa esimerkissä keskimmäinen tulostus aiheuttaisi `ArithmeticException`-poikkeuksen ja tällöin siirryttäisiin välittömästi `catch`-lohkoon. Kolmesta `try`-lohkossa olevasta tulostusrivistä

tulostuisi siis vain ensimmäinen. Jos haluaisimme, että kaikki lauseet, jotka eivät heitä poikkeusta suoritettaisiin, täytyisi meidän tehdä jokaiselle lauseelle oma `try-catch` -rakenteensa. Tällöin saisimme aikaan melkoisen `try-catch` -viidakon. Useimmiten tällaisissa tilanteissa olisikin järkevää tehdä suoritettavasta toimenpiteestä aliohjelma, joka sisältäisi `try-catch` -rakenteen. Tällöin koodi siistiytyisi ja lyhenisi huomattavasti.

Esimerkissämme `catch`-lohkossa tulostetaan nyt virheilmoitus. Poikkeusolio on nimetty "`e`":ksi, joka on hyvin yleinen poikkeusolion viitemuuttujalle annettava nimi. Koska Javan poikkeukset olivat olioita, on niillä myös joukko metodeja. `catch`-lohkossa on kutsuttu `ArithmeticException`-luokan `getMessage`-metodia, joka palauttaa poikkeukselle määritellyn virheilmoituksen.

Voidaan määritellä myös useita `catch`-lohkoja, jolloin voimme ottaa kiinni monia erityyppisiä poikkeuksia.

```
try {
   //jotain lauseita mitä koitetaan suorittaa
} catch (PoikkeusTyyppiA e) {
   //jotain toimenpiteitä mitä tehdään kun poikkeus ilmenee
} catch (PoikkeusTyyppiB e) {
   //jotain toimenpiteitä mitä tehdään kun poikkeus ilmenee
} catch (PoikkeusTyyppiC e) {
   //jotain toimenpiteitä mitä tehdään kun poikkeus ilmenee
}
```

Jos poikkeustapauksessa tehtävät toimenpiteet eivät vaihtele riippuen poikkeuksen tyypistä, voimme ottaa kiinni yksinkertaisesti `Exception`-luokan olioita. Kaikki Javan poikkeusluokat perivät `Exception`-luokan, joten sitä käyttämällä saamme kiinni kaikki mahdolliset poikkeukset. Joskus voi olla järkevää laittaa viimeinen `catch`-lohko nappaamaan `Exception`-poikkeuksia, jolloin saamme kaikki loputkin mahdolliset poikkeukset kiinni. Monesti kuitenkin tiedämme hyvin tarkkaan, mitä poikkeuksia toimenpiteemme voivat aiheuttaa, joten tämä olisi turhaa. Ja jos emme tiedä mitään poikkeuksesta, emme sitä osaa käsitelläkään ja siksi `Exception`-luokan poikkeuksen kiinniottamisessa on oltava todella varovainen. [VES][KOS][DEI]

23.2 finally-lohko

`finally`-lohkon käyttäminen ei ole pakollista. Se tulee aina `catch`-lohkojen jälkeen. `finally`-lohko suoritetaan joka tapauksessa riippumatta siitä aiheuttiko `try`-lohko poikkeuksia. Monesti `finally`-lohko onkin hyödyllinen muun muassa käsiteltäessä tiedostoja, jolloin tiedosto on suljettava aina käsittelyn jälkeen poikkeuksista riippumatta. `finally`-lohkon sisältävä `try-catch` -rakenne olisi yleisessä muodossa seuraava:

```
try {
   //jotain lauseita mitä koitetaan suorittaa
} catch (PoikkeusLuokanNimi poikkeukselleAnnettavaNimi) {
   //jotain toimenpiteitä mitä tehdään kun poikkeus ilmenee
} finally {
   //joka tapauksessa suoritettavat lauseet
}
```

23.3 Yleistä

Poikkeukset ovat nimensä mukaan säännöstä poikkeavia tapahtumia. Niitä ei tulisikaan käyttää periaatteella: "En ole varma toimiiko tämä, joten laitan try-catch:n sisään." Poikkeukset ovat sitä varten, että hyvinkin suunnitellussa ja mietityssä koodissa voi joskus tapahtua jotain odottamatonta, johon varautuminen voi parhaimmillaan pitää lentokoneen kurssissa tai hätäkeskuspäivystyksen tietojärjestelmän pystyssä.

24. Lukujen esitys tietokoneessa

24.1 Lukujärjestelmät

Meille tutuin lukujärjestelmä on 10-järjestelmä. Siinä on 10 eri symbolia lukujen esittämiseen (0...9). Lukua 10 sanotaan 10-järjestelmän *kantaluvuksi*. Tietotekniikassa käytetään kuitenkin myös muita lukujärjestelmiä. Yleisimpiä ovat 2-järjestelmä (binäärijärjestelmä), 8-järjestelmä (oktaalijärjestelmä) ja 16-järjestelmä (heksajärjestelmä). Binäärijärjestelmässä luvut esitetään kahdella symbolilla (0 ja 1) ja oktaalijärjestelmässä vastaavasti kahdeksalla symbolilla (0...7). Samalla periaatteella heksajärjestelmässä käytetään 16 symbolia, mutta koska numerot loppuvat kesken, otetaan avuksi aakkoset. Symbolin 9 jälkeen tulee siis symboli A, jonka jälkeen B ja näin jatketaan edelleen F:n asti, joka vastaa siis 10-järjestelmän lukua 15. Heksajärjestelmä sisältää siis symbolit 0...9A...F.

Koska lukujärjestelmät sisältävät samoja symboleja, täytyy ne osata jotenkin erottaa toisistaan. Tämä tehdään usein alaindekseillä. Esimerkiksi binääriluku 11 voitaisiin kirjoittaa muodossa 11_2.Tällöin sen erottaa 10-järjestelmän luvusta 11, joka voitaisiin vastaavasti kirjoittaa muodossa 11_{10}. Koska alaindeksien kirjoittaminen koneella on hieman haastavaa, käytetään usein myös merkintää, jossa binääriluvun perään lisätään B-kirjain. Esimerkiksi 11B tarkoittaisi samaa kuin 11_2.

Kaikissa yllä mainituissa lukujärjestelmissä symbolin paikalla on oleellinen merkitys. Kun symboleja laitetaan peräkkäin, ei siis ole yhdentekevää millä paikalla luvussa tietty symboli on. [MÄN]

24.2 Paikkajärjestelmät

Käyttämämme lukujärjestelmät ovat paikkajärjestelmiä, eli jokaisen numeron paikka luvussa on merkitsevä. Jos numeroiden paikkaa luvussa vaihdetaan, muuttuu luvun arvokin. Luvun

$$n_3n_2n_1n_0$$

arvo on

$$n_3*k^3 + n_2*k^2 + n_1*k^1 + n_0*k^0$$

missä k on käytetyn järjestelmän kantaluku. Esimerkiksi 10-järjestelmässä:

$$2536 = 2*10^3 + 5*10^2 + 3*10^1 + 6*10^0 = 2*1000 + 5*100 + 3*10 + 6*1$$

Sanomme siis, että luvussa 2536 on 2 kappaletta tuhansia, 5 kappaletta satoja, 3 kappaletta kymmeniä ja 6 kappaletta ykkösiä.

Jos luvussa olevat symbolien paikat siis numeroidaan oikealta vasemmalle alkaen nollasta, saadaan luvun arvo selville summaamalla kussakin paikassa oleva arvo kerrottuna kantaluku potenssiin paikan numero. Tämä toimii myös desimaaliluvuille kun numeroidaan desimaalimerkin oikealla puolella olevat paikat -1, -2, -3 jne. Esimerkiksi

$$25.36 = 2*10^1 + 5*10^0 + 3*10^{-1} + 6*10^{-2} = 2*10 + 5*1 + 3*0.1 + 6*0.01$$

24.3 Binääriluvut

Binäärijärjestelmässä kantalukuna on 2 ja siten on käytössä kaksi symbolia: 0 ja 1. Binäärijärjestelmä on tietotekniikassa oleellisin järjestelmä, sillä lopulta laskenta suurimmassa osassa nykyprosessoreita tapahtuu binäärilukuina. Tarkemmin sanottuna binääriluvut esitetään

prosessorissa jännitteinä. Tietty jänniteväli vastaa arvoa 0 ja tietty jänniteväli arvoa 1.

24.3.1 Binääriluku 10-järjestelmän luvuksi

Esimerkiksi binääriluku 10110 voidaan muuttaa 10-järjestelmän luvuksi seuraavasti.

$$10110_2 = 1*2^4 + 0*2^3 + 1*2^2 + 1*21 + 0*2^0 = 0 + 2 + 4 + 0 + 16 = 22_{10}$$

Binäärimuodossa oleva desimaaliluku 101.1011 saadaan muutettua 10-järjestelmän luvuksi seuraavasti. Muuttaminen tehdään samalla periaatteella kun yllä. Nyt desimaaliosaan mentäessä potenssien vähentämistä edelleen jatketaan, jolloin potenssit muuttuvat negatiivisiksi:

$$101.1011_2 = 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} + 1*2^{-4} = 4 + 0 + 1 + 0.5 + 0 + 0.125 + 0.0625 = 5.6875_{10}$$

Binääriluku 101.1011 on siis 10-järjestelmän lukuna 5.6875.

24.3.2 10-järjestelmän luku binääriluvuksi

10-järjestelmän luku saadaan muutettua binääriluvuksi jakamalla sen kokonaisosaa toistuvasti kahdella ja merkkaamalla paperin syrjään 0, jos jako meni tasan ja muuten 1. Kun lukua ei voi enää jakaa, saa binääriluvun selville kun lukee jakojäännökset päinvastaisesta suunnasta, kuin mistä aloitimme laskemisen. Esimerkiksi luku 19_{10} voidaan muuttaa binääriluvuksi seuraavasti:

```
19/2 = 9, jakojäännös 1
 9/2 = 4, jakojäännös 1
 4/2 = 2, jakojäännös 0
 2/2 = 1, jakojäännös 0
 1/2 = 0, jakojäännös 1
```

Kun jakojäännökset luetaan nyt alhaalta ylöspäin, saamme binääriluvun 10011. Vastaavasti laskenta voitaisiin hahmotella kuten alla, josta jakojäännös selviää paremmin. Idea molemmissa on kuitenkin sama.

```
19 = 2*9+1
 9 = 2*4+1
 4 = 2*2+0
 2 = 2*1+0
 1 = 2*0+1
```

Muutetaan vielä luku 126_{10} binääriluvuksi.

```
126 = 2*63+0
 63 = 2*31+1
 31 = 2*15+1
 15 =  2*7+1
  7 =  2*3+1
  3 =  2*1+1
  1 =  2*0+1
```

Valmis binääriluku on siis 1111110

Desimaaliluvuissa täytyy kokonaisosa ja desimaaliosa muuttaa binääriluvuiksi erikseen. Kokonaisosa muutetaan binääriluvuksi kuten yllä. Desimaaliosa muutetaan kertomalla desimaaliosaa toistuvasti kahdella ja merkkaamalla paperin syrjään nyt 1, jos tulo oli suurempaa tai yhtä suurta kuin 1 ja 0 jos tulo jäi alle yhden. Muutetaan luku 0.8125_{10} binääriluvuksi.

```
0.8125 * 2 = 1.625
 0.625 * 2 = 1.25
  0.25 * 2 = 0.5
   0.5 * 2 = 1.0
```

Luku meni tasan, eli luku $0.8125_{10} = 0.1101_2$. Binääriluku voidaan siis lukea kuten alla olevassa

kuvassa.

```
0.8125 * 2 = 1.625
 0.625 * 2 = 1.25
  0.25 * 2 = 0.5
   0.5 * 2 = 1.0
```

Kuva 33: Luvun 0.8125 muuttaminen binääriluvuksi

Muutetaan vielä luku 0.675_{10} binääriluvuksi.

```
0.675 * 2 = 1.35
 0.35 * 2 = 0.7
  0.7 * 2 = 1.4
  0.4 * 2 = 0.8
  0.8 * 2 = 1.6
  0.6 * 2 = 1.2
  0.2 * 2 = 0.4
  0.4 * 2 = 0.8
```

Kun kerromme uudelleen samaa desimaaliosaa kahdella, voidaan laskeminen lopettaa. Tällöin kyseessä on päättymätön luku. Luvussa rupeaisi siis toistumaan jakso 11001100. Nyt luku luetaan samasta suunnasta, josta laskeminenkin aloitettiin. Enää meidän tarvitsee päättää millä tarkkuudella luku esitetään. Mitä enemmän bittejä käytämme, sitä tarkempi luvusta tulee.

$$0.675_{10} = 0.101011001100110011_2$$

Jaksoa voitaisiin siis jatkaa loputtomiin, mutta oleellista on, että lukua 0.675 ei pystytä esittämään tarkasti binääriluvuilla.

Koitetaan muuttaa luku 23.375_{10} binääriluvuksi. Ensiksi muutetaan kokonaisosa.

```
23 = 2*11+1
11 = 2 *5+1
 5 = 2 *2+1
 2 = 2* 1+0
 1 = 2* 0+1
```

Kokonaisosa on siis 10111_2. Muutetaan vielä desimaaliosa.

```
0.375 * 2 = 0.75
 0.75 * 2 = 1.5
  0.5 * 2 = 1.0
```

Eli $23.375_{10} = 10111.011_2$.

24.4 Negatiiviset binääriluvut

Negatiivinen luku voidaan esittää joko suorana, 1-komplementtina tai 2-komplementtina.

24.4.1 Suora tulkinta

Suorassa tulkinnassa varataan yksi bitti ilmoittamaan luvun etumerkkiä (+/-). Jos meillä on käytössä 4 bittiä, niin tällöin luku $+3_{10}$ = 0011 ja -3_{10} = 1011. Suoran esityksen mukana tulee ongelmia laskutoimituksia suoritettaessa; mm. luvulla nolla on tällöin kaksi esitystä, 0000 ja 1000, mikä ei ole toivottava ominaisuus.

24.4.2 1-komplementti

Jos luku on positiivinen, kirjoitetaan se normaalisti, ja jos luku on negatiivinen, niin käännetään kaikki bitit päinvastaisiksi. Esimerkiksi luku $+3_{10}$ = 0011 ja -3_{10} = 1100. Tässäkin systeemissä luvulla nolla on kaksi esitystä, 0000 ja 1111.

24.4.3 2-komplementti

Yleisimmin käytetty tapa ilmoittaa negatiiviset luvut on 2-komplementti. Tällöin positiivisesta luvusta otetaan ensin 1-komplementti, eli muutetaan nollat ykkösiksi ja ykköset nolliksi ("käännetään" kaikki bitit vastakkaisiksi), minkä jälkeen tulokseen lisätään 1. Tämän esitystavan etuna on se, että yhteenlasku toimii totuttuun tapaan myös negatiivisilla luvuilla. Vähennyslasku suoritetaan summaamalla luvun vastaluku:

```
2-3 = 2+(-3)
```

Muodostetaan luvusta 1 negatiivinen luku:

```
luku 1:            0001
käännetään bitit: 1110
lisätään 1:       1111
```

Luku -1 on siis kahden komplementtina 1111. Kokeillaan tehdä samaa luvulle 2.

```
luku 2:            0010
käännetään bitit: 1101
lisätään 1:       1110
```

Saatiin siis, että -2 on kahden komplementtina 1110. Kokeillaan vielä muuttaa 3 vastaavaksi negatiiviseksi luvuksi.

```
luku 3:            0011
käännetään bitit: 1100
lisätään 1:       1101
```

Saatiin, että -3 on kahden komplementtina 1101.

Voidaanko luvut muuttaa samalla menetelmällä takaisin positiivisiksi luvuiksi? Kokeile!

24.4.4 2-komplementin yhteenlasku

Jos vastauksen merkitsevin bitti (vasemman puoleisin) on 1, on vastaus negatiivinen ja 2-komplementtimuodossa. Tällöin vastauksen tulkitsemiseksi sille suoritetaan muunnos edellä esitetyllä tavalla (ensin käännetään bitit, sitten lisätään 1). Muunnoksen tuloksena saadaan luvun itseisarvo, itse luku on siis tällöin aina negatiivinen. Jos merkitsevin bitti on 0, on vastaus positiivinen, eikä mitään muunnosta tarvitse suorittaa.

Lasketaan esimerkiksi 2+1

```
 00
  0010
+ 0001
 -----
  0011
```

Merkitsevin bitti on 0, joten vastaus on $0011_2 = 3_{10}$. Lasketaan seuraavaksi 1-2.

```
 00
  0001
+ 1110
  ----
```

```
 1111
```

Merkitsevin bitti on nyt 1 eli luku on kahden komplementti. Kun käännetään bitit ja lisätään 1 saadaan luku 0001. Koska merkitsevin bitti oli 1 on luku siis negatiivinen, joten saatiin vastaukseksi -1.

Lasketaan vielä -2-3.

```
11
  1110
+ 1101
  ----
  1011
```

Luku on jälleen negatiivinen. Kun käännetään bitit ja lisätään 1, saadaan $0101_2 = 5_{10}$. Vastaus on siis -5_{10}.

Lopuksi vielä pari laskua joiden tulos ei mahdu 4:ään bittiin. Aluksi 6+7

```
011
  0110
+ 0111
  ----
  1101   => 0010 + 1  => -3 (siis negatiivinen luku kahden luvun yhteenlaksusta)
```

Vastaavasti -7-6

```
10
  1001
+ 1010
  ----
  0011  => +3 (positiivinen luku kahden negatiivisen yhteenlaskusta)
```

Kahdessa viimeisessä laskussa päädyttiin väärään tulokseen! Tämä on luonnollista, sillä tietenkään rajallisella bittimäärällä ei voida esittää rajaansa isompia lukuja. Meidän esimerkkimme 4-bitin lukualueella saadaan vain lukualue [-8,7]. Vertaa Javan alkeistietotyyppien lukualueisiin, jotka esiteltiin kohdassa 7.1.1 Javan alkeistietotyypit. 2-komplementin yksi lisäetu on se, että siinä mainitunkaltainen *ylivuoto* (**overflow**), eli lukualueen ylitys, on helppo todeta: viimeiseen bittiin (merkkibittiin) tuleva ja sieltä lähtevä muistinumero on erisuuri. Edellisissäkin esimerkeissä oikeaan tulokseen päätyneissä laskuissa ne olivat samat ja väärän tulokseen päätyneissä laskuissa eri suuret. *Alivuoto* (**underflow**) tulee vastaavasti liukuluvuilla silloin kun laskutoimituksen tulos tuottaa nollan, vaikka oikeassa maailmassa tulos ei vielä olisikaan nolla.

24.5 Lukujärjestelmien suhde toisiinsa

Koska binääriluvuista muodostuu usein hyvin pitkiä, ilmoitetaan ne usein ihmiselle helpommin luettavassa muodossa joko 8- tai 16-järjestelmän lukuina. Tutustutaan nyt jälkimmäiseen, eli heksajärjestelmään. Heksajärjestelmässä on käytössä merkit 0...9A...F, eli yhteensä 16 symbolia. Näin yhdellä symbolilla voidaan esittää jopa luku $15_{10} = 1111_2$. Heksalukuja A...F vastaavat 10-järjestelmän luvut näet alla olevasta taulukosta.

A_{16}	10_{10}
B_{16}	11_{10}
C_{16}	12_{10}
D_{16}	13_{10}
E_{16}	14_{10}
F_{16}	15_{10}

Yhdellä 16-järjestelmä symbolilla voidaan siis esittää 4-bittinen binääriluku. Binääriluku

voidaankin muuttaa heksajärjestelmän luvuksi järjestelemällä bitit oikealta alkaen neljän bitin ryhmiin ja käyttämällä kunkin 4-bitin yhdistelmän heksavastinetta. Muutetaan luku 11101101_2 heksajärjestelmään.

$$11101101_2 = 1110\ 1101_2$$
$$1110_2 = E_{16}$$
$$1101_2 = D_{16}$$
$$11101101_2 = 1110\ 1101_2 = ED_{16}$$

Vastaavasti voitaisiin muuttaa binääriluku 8-järjestelmän luvuksi, mutta nyt vain järjesteltäisiin bitit oikealta alkaen kolmen bitin ryhmiin.

Alla olevassa taulukossa on esitetty 10-, 2-, 8- ja 16-järjestelmän luvut $0_{10}...15_{10}$. Lisäksi on esitetty mikä olisi vastaavan binääriluvun 2-komplementti -tulkinta.

10-järj.	2-järj.	8-järj.	16-järj.	2-komplementti
0	0000	00	0	0
1	0001	01	1	1
2	0010	02	2	2
3	0011	03	3	3
4	0100	04	4	4
5	0101	05	5	5
6	0110	06	6	6
7	0111	07	7	7
8	1000	10	8	-8
9	1001	11	9	-7
10	1010	12	A	-6
11	1011	13	B	-5
12	1100	14	C	-4
13	1101	15	D	-3
14	1110	16	E	-2
15	1111	17	F	-1

24.6 Liukuluku (floating-point)

Liukulukua käytetään siis reaalilukujen esitykseen tietokoneissa. Liukulukuesitykseen kuuluu neljä osaa: etumerkki (s), mantissa (m), kantaluku (k) ja eksponentti (c). Kantaluvulla ja eksponentilla määritellään luvun suuruusluokka ja mantissa kuvaa luvun merkitseviä numeroita. Luku x saadaan laskettua kaavalla:

$$x = (-1)^s m k^c$$

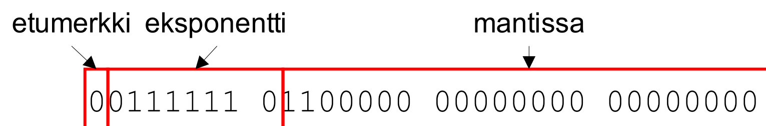
Tietotekniikassa yleisimmin käytetyssä standardissa IEE 754 kantaluku on 2, jolloin kaava saadaan muotoon:

$$x = (-1)^s m 2^c$$

IEE 754-standardissa luvun etumerkki (s) ilmoitetaan bittimuodossa ensimmäisellä bitillä, jolloin s voi saada joko arvon 0, joka tarkoittaa positiivista lukua tai arvon 1, joka tarkoittaa siis negatiivista lukua.

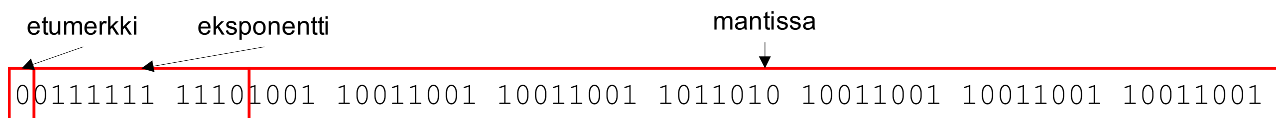
Tutustutaan seuraavaksi kuinka float ja double esitetään bittimuodossa.

`float` on kooltaan 32 bittiä. Siinä ensimmäinen bitti siis tarkoittaa etumerkkiä, seuraavat 8 bittiä eksponenttia ja jäljelle jäävät 23 bittiä mantissaa.



Kuva 34: Float 0.875 liukulukuna bittimuodossa

`double` on kooltaan 64 bittiä. Siinäkin ensimmäinen bitti tarkoittaa etumerkkiä, seuraavat 11 bittiä eksponenttia ja jäljelle jäävät 52 bittiä kuvaavat mantissaa.



Kuva 35: Double 0.800 liukulukuna bittiesityksenä

Eksponentti esitetään niin, että siitä vähennetään ns. BIAS arvo. BIAS arvo `float`issa on 127 ja doublessa se on 1023. Näin samalla binääriluvulla saadaan esitettyä sekä positiiviset että negatiiviset eksponentit. Jos floatin eksponenttia kuvaavat bitit olisivat esimerkiksi 01111110, eli desimaalimuodossa 126, niin eksponentti olisi 126 - 127 = -1.

Mantissa puolestaan esitetään niin, että se on aina vähintään 1. Mantissaa kuvaavat bitit esittävätkin ainoastaan mantissan desimaaliosaa. Jos floatin mantissaa kuvaavat bitit olisivat esimerkiksi 10100000000000000000000, olisi mantissa tällöin binäärimuodossa 1.101 eli desimaalimuodossa 1.625.

24.6.1 Liukuluvun binääriesityksen muuttaminen 10-järjestelmään

Kokeillaan nyt muuttaa muutama binäärimuodossa oleva float kokonaisuudessaan 10-järjestelmän luvuksi. Esimerkkinä liukuluku:

```
00111111 10000000 00000000 00000000
```

Bitit on järjestetty nyt tavuittain. Voisimme järjestellä bitit niin, että liukuluvun eri osat näkyvät selkeämmin:

```
0 01111111 00000000000000000000000
```

Ensimmäinen bitti on nolla, eli luku on positiivinen. Seuraavat 8 bittiä ovat 01111111, joka on 10-järjestelemän lukuna 127 eli eksponentti on 127-127 = 0. Mantissaa esittäviksi biteiksi jää pelkkiä nollia, eli mantissa on 1.0, koska mantissahan oli aina vähintään 1. Nyt liukuluvun kaavalla voidaan laskea mikä luku on kyseessä:

$$x = (-1)^0 * 1.0 * 2^0 = 1.0$$

Kyseessä olisi siis reaaliluku 1.0. Kunhan muistetaan ottaa huomioon ensimmäinen bitti etumerkkinä, voidaan liukuluvun laskemiseen käyttää vielä yksinkertaisempaa kaavaa:

$$x = m2^c$$

Muutetaan vielä toinen liukuluvun binääriesitys 10-järjestelmän luvuksi.

```
00111111 01100000 00000000 00000000
```

Ensimmäinen bitti on jälleen 0, eli luku on positiivinen. Seuraavat 8 bittiä ovat 01111110, joka on

desimaalilukuna 126. Eksponentti on siis 126-127 = -1. Mantissaan jää nyt bitit 11000000000000000000000 eli mantissa on binääriluku 1.11, joka on 10-järjestelmässä luku 1.75. Liukuluvun esittämäksi reaaliluvuksi saadaan siis:

$$1.75*2^{-1} = 0.875$$

24.6.2 10-järjestelmän luku liukuluvun binääriesitykseksi

Kun muutetaan 10-järjestelmän luku liukuluvun binääriesitykseksi, täytyy ensiksi selvittää liukuluvun eksponentti. Tämä saadaan selville skaalaamalla luku välille [1,2[kertomalla tai jakamalla lukua toistuvasti luvulla 2, niin, että luku x on aluksi muodossa:

$$x*2^0$$

Nyt jos jaamme luvun kahdella, niin samalla eksponentti kasvaa yhdellä. Jos taas kerromme luvun kahdella, niin eksponentti vähenee yhdellä. Näin luvun arvo ei muutu ja saamme luvun muotoon

$$m*2^c$$

jossa m on välillä [1,2[. Tämä onkin jo liukuluvun esitysmuoto. Enää meidän ei tarvitsisi kuin muuttaa se tietokoneen ymmärtämäksi binääriesitykseksi.

Muutetaan esimerkkinä 10-järjestelmän luku -0.1 liukuluvun binääriesitykseksi. Etumerkki huomioidaan sitten ensimmäisessä bitissä, joten nyt voidaan käsitellä lukua 0.1. Luku voidaan nyt kirjoittaa muodossa :

```
0.1 = 0.1*2
```

Nyt kerrotaan lukua kahdella kunnes se on välillä [1,2[ja muistetaan vähentää jokaisen kertomisen jälkeen eksponenttia yhdellä, jotta luvun arvo ei muutu.

$$0.1 = 0.1*2^0 = 0.2*2^{-1} = 0.4*2^{-2} = 0.8*2^{-3} = 1.6*2^{-4}$$

Eksponentiksi saatiin -4, eli liukuluvun binääriesitykseen siihen lisätään BIAS, eli saadaan 10-järjestelmän luku -4 + 127 = 123, joka on binäärilukuna 1111011. Muutetaan nyt mantissa binääriluvuksi. Muista, että mantissan kokonaisosaa ei merkitty liukuluvun binääriesitykseen.

```
Ensimmäinen bitti  => 1  (jota ei merkitä)
0.6 * 2  = 1.2     => 1
0.2 * 2  = 0.4     => 0
0.4 * 2  = 0.8     => 0
0.8 * 2  = 1.6     => 1
0.6 * 2  = 1.2     => 1
```

Tästä nähdään jo, että kyseessä on päättymätön luku, koska meidän täytyy jälleen kertoa lukua 0.6 kahdella. Laskeminen voidaan siis lopettaa, sillä jakso on jo nähtävillä. Kun jaksoa jatketaan 23 bitin mittaiseksi, saadaan mantissaksi binääriluku 10011001100110011001100. Seuraavat kaksi bittiä olisivat 11, joten luku pyöristyy vielä muotoon 10011001100110011001101. Nyt kaikki liukuluvun osat ovat selvillä:

- Etumerkkibitti: 1, sillä alkuperäinen luku oli -0.1
- Eksponentti: 1111011
- Mantissa: 10011001100110011001101

Eli yhdistämällä saadaan:

```
1 1111011 10011001100110011001101
```

Binääriluku voidaan vielä järjestellä tavuittain:

```
1111101 11001100 11001100 11001101
```

Lukua 0.1 ei siis voi esittää liukulukuna tarkasti, vaan pientä heittoa tulee aina.

24.6.3 Huomio: doublen lukualue

Liukuluku esitys on siitä näppärä, että eksponentin ansiosta sillä saadaan todella suuri lukualue käyttöön. double:n eksponenttiin oli käytössä 11 bittiä. Tällöin suurin mahdollinen eksponentti on binääriluku 11111111111 vähennettynä double:n BIAS arvolla. Tästä saadaan desimaalilukuna 2047 - 1023 = 1024. Kun mantissa voi olla välillä [1,2[, saadaan double:n maksimiarvoksi $2*2^{1024}$, joka on likimain $3.59*10^{308}$. double:n lukualue on siis suunnilleen [$-3.59*10^{308}$, $3.59*10^{308}$], kun `long`-tyypin lukualue oli [-2^{63}, 2^{63}[. `double`-tyypillä pystytään siis esittämään paljon suurempia lukuja kuin `long`-tyypillä.

24.6.4 Liukulukujen tarkkuus

Liukuluvut ovat tarkkoja, jos niillä esitettävä luku on esitettävissä mantissan bittien määrän mukaisena kahden potenssien kombinaatioina. Esimerkiksi luvut 0.5, 0.25 jne. ovat tarkkoja. Harmittavasti kuitenkin edellä todettiin että 10-järjestelmän luku 0.1 ei ole tarkka. Siksi esimerkiksi rahalaskuissa on käytettävä joko senttejä tai esimerkiksi Javan `BigDecimal`-luokkaa. Laskuissa kuitenkin nämä erikoistyypit ovat hitaampia, tilanteesta riippuen eivät kuitenkaan välttämättä merkitsevästi.

Toisaalta liukuluvulla voi esittää tarkasti kokonaislukuja aina arvoon $2^{\text{mantissan bittien lukumäärä}}$ saakka. Eli `double`lla (52 bittiä mantissalle) voi tarkasti käsitellä suurempia kokonaislukuja kuin `int`-tyypillä (32 bittiä luvun esittämiseen). `long`-tyypin 64-bitillä päästään vielä `double`a suurempiin tarkkoihin kokonaislukuihin. Valmiit kokonaislukutyypit ovat yleensä laskennassa liukulukutyyppejä hitaampia, joten siksi kokonaislukutyyppejä kannattaa suosia. Nykyprosessoreissa sen sijaan `double` ja `float` tyyppien laskut eivät merkittävästi poikkea suoritusnopeudeltaan, joten siksi `double`a on pidettävä ensisijaisena valintana kun tarvitaan reaalilukua. Kaikissa mobiilialustoissa ei välttämättä ole käytössä liukulukutyyppejä ja tämä on otettava erikoistapauksissa huomioon. Joissakin tapauksissa kieli (esimerkiksi Java) voi tukea liukulukuja, mutta kohdealustassa ei ole niille prosessoritason tukea. Tällöin liukulukujen käyttö voi olla hidasta. Tarvittaessa laskuja voi suorittaa niin, että skaalaa lukualueen kuvitteellisesti niin, että vaikka sisäisesti luku 1000 on loogisesti 1 ja 1 on loogisesti 0.001 (**fixed point arithmetic**).

24.6.5 Intelin prosessorikaan ei ole aina osannut laskea liukulukuja oikein

Wired-lehden 10 pahimman ohjelmistobugin listalle on päässyt Intelin prosessorit, joissa ilmeni vuonna 1993 virheitä, kun suoritettiin jakolaskuja tietyllä välillä olevilla liukuluvuilla. Prosessorien korvaaminen aiheutti Intelille arviolta 475 miljoonan dollarin kulut. Tosin virhe esiintyi käytännössä vain muutamissa harvoissa erittäin matemaattisissa ongelmissa, eikä oikeasta häirinnyt tavallista toimistokäyttäjää millään tavalla. Tästä ja muista listan bugeista voi lukea lisää alla olevasta linkistä.

http://www.wired.com/software/coolapps/news/2005/11/69355

25. ASCII-koodi

ASCII (American Standard Code for Information Interchange) on merkistö, joka käyttää seitsemänbittistä koodausta. Sillä voidaan siis esittää ainoastaan 128 merkkiä. Nimestäkin voi päätellä, että skandinaaviset merkit eivät ole mukana, mistä seuraa ongelmia tietotekniikassa vielä tänäkin päivänä, kun siirrytään ”skandeja” tukevasta koodistosta ASCII-koodistoon.

ASCII-koodistossa siis jokaista merkkiä vastaa yksi 7-bittinen binääriluku. Vastaavuudet näkyvät alla olevasta taulukosta, jossa selkeyden vuoksi binääriluku on esitetty 10-järjestelmän lukuna, sekä heksalukuna.

Des	Hex	Merkki									
0	0	NUL (null)	32	20	Space	64	40	@	96	60	`
1	1	SOH (otsikon alku)	33	21	!	65	41	A	97	61	a
2	2	STX (tekstin alku)	34	22	"	66	42	B	98	62	b
3	3	ETX (tekstin loppu)	35	23	#	67	43	C	99	63	c
4	4	EOT (end of transmission)	36	24	$	68	44	D	100	64	d
5	5	ENQ (enquiry)	37	25	%	69	45	E	101	65	e
6	6	ACK (acknowledge)	38	26	&	70	46	F	102	66	f
7	7	BEL (bell)	39	27	'	71	47	G	103	67	g
8	8	BS (backspace)	40	28	(	72	48	H	104	68	h
9	9	TAB (tabulaattori)	41	29	)	73	49	I	105	69	i
10	A	LF (uusi rivi)	42	2A	*	74	4A	J	106	6A	j
11	B	VT (vertical tab)	43	2B	+	75	4B	K	107	6B	k
12	C	FF (uusi sivu)	44	2C	,	76	4C	L	108	6C	l
13	D	CR (carriage return)	45	2D	-	77	4D	M	109	6D	m
14	E	SO (shift out)	46	2E	.	78	4E	N	110	6E	n
15	F	SI (shift in)	47	2F	/	79	4F	O	111	6F	o
16	10	DLE (data link escape)	48	30	0	80	50	P	112	70	p
17	11	DC1 (device control 1)	49	31	1	81	51	Q	113	71	q
18	12	DC2 (device control 2)	50	32	2	82	52	R	114	72	r
19	13	DC3 (device control 3)	51	33	3	83	53	S	115	73	s
20	14	DC4 (device control 4)	52	34	4	84	54	T	116	74	t
21	15	NAK (negative acknowledge)	53	35	5	85	55	U	117	75	u
22	16	SYN (synchronous table)	54	36	6	86	56	V	118	76	v
23	17	ETB (end of trans. block)	55	37	7	87	57	W	119	77	w
24	18	CAN (cancel)	56	38	8	88	58	X	120	78	x
25	19	EM (end of medium)	57	39	9	89	59	Y	121	79	y
26	1A	SUB (substitute)	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC (escape)	59	3B	;	91	5B	[	123	7B	{
28	1C	FS (file separator)	60	3C	<	92	5C	\	124	7C	|
29	1D	GS (group separator)	61	3D	=	93	5D	]	125	7D	}
30	1E	RS (record separator)	62	3E	>	94	5E	^	126	7E	~
31	1F	US (unit separator)	63	3F	?	95	5F	_	127	7F	DEL

Taulukko 1: ASCII-merkit

Monissa ohjelmointikielissä, kuten myös Javassa, ASCII-merkkien desimaaliarvoja voidaan sijoittaa suoraan `char`-tyyppisiin muuttujiin. Esimerkiksi pikku-a:n (a) voisi sijoittaa muuttujaan `c` seuraavasti:

```
char c = 97;
```

Esimerkiksi tiedosto, jonka sisältö olisi loogisesti

```
Kissa istuu
puussa
```

koostuisi oikeasti Windows-käyttöjärjestelmässä biteistä (joiden arvot on lukemisen helpottamiseksi seuraavassa kuvattu heksana):

```
4B 69 73 73 61 20 69 73 74 75 75 0D 0A 70 75 75 73 73 61
```

Erona eri käyttöjärjestelmissä on se, miten rivinvaihto kuvataan. Windowsissa rivinvaihto on CR LF (0D 0A) ja Unix-pohjaisissa järjestelmissä pelkkä LF (0A).

Tiedoston sisältöä voit katsoa esimerkiksi antamalla komentoriviltä komennot (jos tiedosto on kirjoitettu tiedostoon kissa.txt)

```
C:\MyTemp>debug kissa.txt
-d
0D2F:0100  4B 69 73 73 61 20 69 73-74 75 75 0D 0A 70 75 75   Kissa istuu..puu
0D2F:0110  73 73 61 61 61 6D 65 74-65 72 73 20 34 00 1E 0D   ssaaameters 4...
...
-q
```

26. Syntaksin kuvaaminen

26.1 BNF

Syntaksia (kielioppia) kuvataan usein BNF:llä (Backus-Naur Form). Kielen peruselementit on käyty läpi alla olevassa taulukossa:

<>	BNF-kaavio koostuu *non-terminaaleista* (välikesymbolit) ja *terminaaleista* (päätesymbolit). Non-terminaalit kirjoitetaan pienempi kuin (<)- ja suurempi kuin (>)-merkkien väliin. Jokaiselle non-terminaalille on oltava jossain määrittely. Terminaali sen sijaan kirjoitetaan koodin sellaisenaan.
::=	Aloittaa non-terminaalin määrittelyn. Määrittely voi sisältää uusia non-terminaaleja ja terminaaleja.
|	”|”-merkki kuvaa sanaa ”tai”. Tällöin ”|”-merkin vasemmalla puolella olevan osan sijasta voidaan kirjoittaa oikealla puolella oleva osa.

Määrittely on yleisessä muodossa seuraava:

```
<nonterminaali> ::= _lause_
```

Jossa `_lause_` voi sisältää uusia non-terminaaleja ja terminaaleja, sekä "|"-merkkejä.

Kielen syntaksin kuvaaminen aloitetaan käännösyksikön (**complitaton unit**) määrittelystä. Tämä on Javassa .java-päätteinen tiedosto. Tämä on siis ensimmäinen non-terminaali, joka määritellään. Tämä määrittely sisältää sitten toisia non-terminaaleja, joille kaikille on olemassa omat määrittelyt. Näin jatketaan, kunnes lopulta on jäljellä pelkkiä terminaaleja ja kielen syntaksi on yksiselitteisesti määritelty.

Esimerkiksi muuttujan määrittelyn syntaksin voisi kuvata seuraavasti. Esimerkissä on lihavoituna kaikki terminaalit.

```
<local variable declaration statement> ::= <local variable declaration> ;
<local variable declaration> ::= <type> <variable declarators>

<type> ::= <primitive type> | <reference type>
<primitive type> ::= <numeric type> | boolean
<numeric type> ::= <integral type> | <floating-point type>
<integral type> ::= byte | short | int | long | char
<floating-point type> ::= float | double
<reference type> ::= <class or interface type> | <array type>
<class or interface type> ::= <class type> | <interface type>
<class type> ::= <type name>
<interface type> ::= <type name>
<array type> ::= <type> []

<variable declarators> ::= <variable declarator> | <variable declarators> ,
                           <variable declarator>
<variable declarator> ::= <variable declarator id> | <variable declarator id> =
<variable initializer>
<variable declarator id> ::= <identifier> | <variable declarator id> []
<variable initializer> ::= <expression> | <array initializer>
```

Lopetetaan muuttujan määrittelyn kuvaaminen tähän. Kokonaisuudessaan siitä tulisi todella pitkä. Koko Javan syntaksin BNF:nä löytää seuraavasta linkistä.

http://www.daimi.au.dk/dRegAut/JavaBNF.html

26.2 Laajennettu BNF (EBNF)

Alkuperäisellä BNF:llä syntaksin kuvaaminen on melko työlästä. Tämän takia on otettu käyttöön laajennettu BNF (extended BNF). Siinä terminaalit kirjoitetaan lainausmerkkien sisään ja non-terminaalit nyt ilman ”<>”-merkkejä. Lisäksi tulee kaksi uutta ominaisuutta.

{}	Aaltosulkeiden sisään kirjoitetut osat voidaan jättää joko kokonaan pois tai toistaa yhden tai useamman kerran.
[]	Hakasulkeiden sisään kirjoitetut osat voidaan suorittaa joko kerran tai ei ollenkaan.

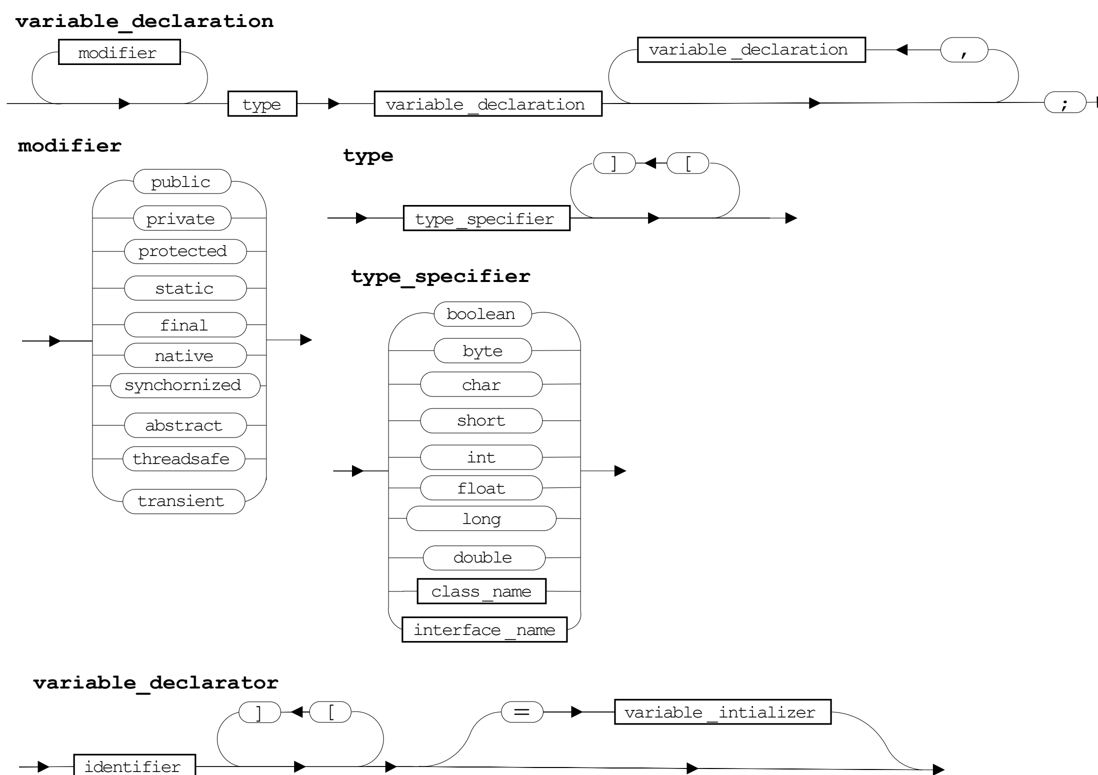
Nyt muuttujan määrittelyn syntaksi saadaan kuvattua hieman helpommin:

```
variable_declaration ::= { modifier } type variable_declarator
                             { "," variable_declarator } ";"
modifier ::= "public" | "private" | "protected" | "static" | "final" | "native" |
             "synchronized" | "abstract" | "threadsafe" | "transient"
type ::= type_specifier { "[" "]" }
type_specifier  ::= "boolean" | "byte" | "char" | "short" | "int" | "float" | "long"
                             | "double" | class_name | interface_name
variable_declarator ::= identifier { "[" "]" } [ "="variable_initializer ]
identifier ::= "a..z,$,_" { "a..z,$,_,0..9,unicode character over 00C0" }
variable_initializer ::= expression | ( "{" [ variable_initializer
                             { "," variable_initializer } [ "," ] ] "}" )
```

Lausekkeen (**expression**) avaamisesta aukeaisi jälleen uusia ja uusia non-terminaaleja, joten muuttujan määrittelyn kuvaaminen kannattaa lopettaa tähän. Voit katsoa loput seuraavasta linkistä:

http://tioswww.unige.ch/db-research/Enseignement/analyseinfo/JAVA/BNFindex.html

Vastaavasti syntaksia voidaan kuvata ”junaradoilla”. Tämä on eräs graafinen tapa kuvata syntaksia. Kuvataan seuraavaksi muuttujaan määrittelyä ”junaratojen” avulla.



Kuva 36: Muuttujan määrittelyn syntaksia "junaradoilla" esitettynä

”Junaradoissa” non-terminaalit on kuvattu suorakulmiolla ja terminaalit vähän pyöreämmillä suorakulmiolla. Vaihtoehdot kuvataan taas niin, että risteyskohdassa voidaan valita vain yksi

vaihtoehtoisista raiteista. Lisäksi raiteissa on ”silmukoita”, joissa voidaan tehdä useampi kierros. Silmukoilla kuvataan siis ”{}”-merkkien välissä olevia lauseita. Lisäksi on ”ohitusraiteita”, joilla voidaan ohittaa joku osa kokonaan. Tällä kuvataan ”[]”-merkkien välissä olevia lauseita.

Kuvasta puuttuu vielä tekstiesimerkissä olevien `identifier` *ja* `variable_intializer` *non-terminaalien junarataesitys. Piirrä niiden ”junaradat” samaan tapaan.*

Lisätietoa:

- TIEA241 Automaatit ja kieliopit .
- Paavo Niemisen 2007 pitämän Ohjelmointi 1-kurssin luentokalvot: http://users.jyu.fi/~nieminen/ohj1/materiaalia/luento02.pdf. Junaratoja löytyy myös muista Paavon kalvoista.
- Javan syntaksi EBNF:nä kuvattuna. Sisältää myös graafiset junaradat. http://tioswww.unige.ch/db-research/Enseignement/analyseinfo/JAVA/BNFindex.html
- Wikipedia: http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form
- Googlella löytyy lisää

27. Jälkisanat

Joskus ohjelmoidessa tulee vaan tämmöinen olo:

http://www.youtube.com/watch?v=K21fuhDo5Bo

Totu siihen ja keitä lisää kahvia.

Liite: Hirsipuu olioilla tehtynä

Divide and conquer -roomalainen sananlasku

Hirsipuupelistä saa paremman olioita käyttämällä. Tee aluksi johonkin projektiin hirsipuu-paketti, ellei sinulla ole jo. Siirrä aikaisemmin tehdyn hirsipuupelin tiedosto tähän pakettiin, koska käytämme siinä tehtyjä aliohjelmia. Luodaan nyt kaksi uutta tiedostoa. Toinen on olioluokka `Hirrenpiirto`. Tästä luokasta luodaan sitten ilmentymä eli olio, joka tietää mikä osa hirsipuusta pitää milloinkin piirtää. `Hirrenpiirto`-luokan koodi on seuraava.

```
package hirsipuu;
import fi.jyu.mit.graphics.Axis;
import fi.jyu.mit.graphics.BasicShape;
import fi.jyu.mit.graphics.Circle;
import fi.jyu.mit.graphics.Line;
import fi.jyu.mit.graphics.Rotator;
import fi.jyu.mit.graphics.Window;
import fi.jyu.mit.ohj2.Syotto;

/**
 * Kokeillaan hirsipuun piirtämistä
 * @author vesal
 * @version 21.10.2008
 */
public class Hirrenpiirto {

    private static final BasicShape[] hirrenOsat = {
       new Circle(0,0,1),
       new Line(0,1,0,7),
       new Line(0,7,3,7),
       new Line(3,7,3,6),
       new Circle(3,5.5,0.5),
       new Line(3,5,3,3),
       new Line(3,3,2,1),
       new Line(3,3,4,1),
       new Line(2,4.5,4,4.5)
    };

    final private Window ikkuna;
    private int vaihe = 0;

    /**
     * Alustetaan hirren piirtäminen.
     * @param n montako vaihetta hirttä piirretään aluksi.
     *
     */
    public Hirrenpiirto(int n) {
        ikkuna = new Window(400,400);
        ikkuna.move(3, 0, 0);
        ikkuna.scale(0,0,10,10);
        ikkuna.showWindow();
        for (int i=0; i<n; i++)
          ikkuna.add(hirrenOsat[i]);
        vaihe = n;
    }


    /**
     * Kerrotaan onko kuva jo valmis.
     * @return true jos kuva on jo valmis, false muuten.
     */
    public boolean onValmis() {
        return  vaihe >= hirrenOsat.length;
    }


    /**
     * Piirretään seuraava hirren osa.
     * Jos kuva tulaa valmiiksi, sitä ruvetaan pyörittämään
     * Palautetaan tosi jos koko kuva on valmis
     * @return true jos kuva valmis, muuten false
     */
    public boolean piirraSeuraavaOsa() {
        if ( onValmis() ) return true;

        ikkuna.add(hirrenOsat[vaihe]);
```

```
        vaihe++;
        if ( !onValmis() ) return false;
        new Rotator(ikkuna,Axis.Y,5,100);
        return true;
    }


    /**
     * Palautetaan montako vaihetta on jäljellä hirren piirtämisestä
     * @return monta vaihetta jäljellä.
     */
    public int vaiheitaJaljella() {
        return hirrenOsat.length-vaihe;
    }


    /**
     * @param args
     */
    public static void main(String[] args) {
        Hirrenpiirto hirsi = new Hirrenpiirto(2);

        do {
            Syotto.kysy("Paina Enter");
        } while ( !hirsi.piirraSeuraavaOsa() );
    }

}
```

`Hirrenpiirto`-luokassa määritellään `Hirrenpiirto`-olioille kolme attribuuttia:

1. `private static final BasicShape[] hirrenOsat` - Tämä taulukko sisältää kaikki hirsipuun osat.
2. `final private Window ikkuna` - `ikkuna`-vakioon piirretään hirsipuuta
3. `private int vaihe = 0` - Muuttajassa `vaihe` pidetään muistissa, missä vaiheessa hirsipuuta ollaan menossa. Se alustetaan luonnollisesti arvoon 0.

Kuten huomaat, kaikki olioiden attribuutit määritellään yksityisiksi (**private**). Tällöin niitä ei voi muuttaa suoraan muista luokista käsin (vrt. **public**). Muista luokista olion attribuutteja voidaankin muuttaa ainoastaan olion metodien avulla, jos olio meille tällaisen palvelun tarjoaa.

```
    public Hirrenpiirto(int n) {
```

Yllä oleva rivi aloittaa Hirrenpiirto -luokan konstruktorin. Se saa parametrina kokonaisluvun, joka ilmoittaa montako vaihetta hirsipuuta aluksi piirretään. Konstruktorissa tehdään kaikki toimenpiteet, joita olion luonnin yhteydessä halutaan tehdä. Nyt skaalaamme ikkunan ja laitamme sen näkyville. Lisäksi piirretään hirsipuusta valmiiksi niin monta osaa kuin parametrissa `n` on määritelty. Konstruktoreja voi olla useita eri parametreilla. Konstruktori on aina julkinen ja samanniminen kuin luokka.

`Hirrenpiirto`-luokasta luoduilla oliolla on seuraavat metodit:

- `public boolean onValmis()` Metodi palauttaa `true`, jos kuva on valmis ja muuten `false`.
- `public boolean piirraSeuraavaOsa()` Metodi piirtää hirsipuuhun seuraavan osan. Palauttaa `true`, jos kuva on valmis ja muuten `false`.
- `public int vaiheitaJaljella()` Metodi palauttaa montako vaihetta hirren piirtämisestä on vielä jäljellä.

Luokan pääohjelmassa kokeillaan olion toimintaa. Varsinainen hirsipuupelin toiminnallisuus löytyy kuitenkin luokasta `Hirsipuu2`:

```
package hirsipuu;
import java.util.Random;

import fi.jyu.mit.ohj2.Syotto;
```

```java
import fi.jyu.mit.ohj2.Tiedosto;
import static hirsipuu.Hirsipuu.*;

/**
 * Ohjelmalla pelataan Hirsipuu-peliä.
 * Edelliseen verrattuna nyt myös piirretään hirsipuuta.
 * Arvattavat sanamahdollisuudet luetaan tiedostosta
 * ja yksi niistä arvotaan.

 * @author vesal
 * @version 21.10.2008
 * @version 22.10.2008 piirretään hirsipuuta
 */
public class Hirsipuu2 {

    /**
     * Aliohjelmalla pelataan yksi sana hirsipuupeliö
     * @param sana
     */
    public static void pelaaPeli(String sana) {
        String vaaria = ""; // Sisältää ne väärät arvaukset
        int oikeita = 0;
        StringBuilder tulos = luoTulosjono(sana);

        Hirrenpiirto hirsi = new Hirrenpiirto(2);
        int maxvaaria = hirsi.vaiheitaJaljella();

        tulostaLogo();

        while ( true ) {
            System.out.println();
            System.out.println("Sana: "+harvakseen(tulos));
            String syote = Syotto.kysy("Anna kirjain");
            if ( syote.isEmpty() ) continue;
            char c = syote.charAt(0);
            System.out.println("Annoit kirjaimen " + c);
            int lkm = tutkiOikeat(sana,c,tulos);
            if ( lkm == 0 ) {
                vaaria += c;
                System.out.printf("Virheitä: %d/%d%n", vaaria.length(),maxvaaria);
                System.out.println("Vääriä kirjaimia: " + harvakseen(vaaria));
                if ( hirsi.piirraSeuraavaOsa() ) {
                    System.out.println("Hävisit!");
                    break;
                }
            }
            oikeita += lkm;
            if ( oikeita >= sana.length() ) {
                System.out.println("Voitit!");
                break;
            }

        }

        System.out.println("Sana: " + sana);

    }


    /**
     * Funktiolla arvotaan yksi merkkijono taulukosta
     * @param jonot taulukko josta jono arvotaan
     * @return satunnainen jonot-taulukon rivi
     */
    public static String arvo(String[] jonot) {
        Random rand = new Random();
        int n = rand.nextInt(jonot.length);
        return jonot[n];
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        String sana = "kissa";
        String[] sanat = Tiedosto.lueTiedosto("sanat.txt");
        if ( sanat != null ) sana = arvo(sanat);
        pelaaPeli(sana);
    }
}
```

Pelin varsinainen toiminnallisuus tapahtuu `pelaaPeli`-aliohjelmassa, joka saa parametrina arvailtavan sanan. Pääohjelmassa arvotaan sana taulukosta, kuten aikaisemmin tehdyssä versiossakin. Tämän jälkeen kutsutaan `pelaaPeli`-aliohjelmaa parametrinaan arvottu sana.

Tutkitaan `pelaaPeli`-aliohjelmaa tarkemmin.

```
Hirrenpiirto hirsi = new Hirrenpiirto(2);
```

Aliohjelman neljäs lause luo `Hirrenpiirto`-luokasta `hirsi`-olion. Muut lauseet olivatkin tuttuja aikaisemmasta versiosta. Parametriksi on annettu arvo `2`, eli aluksi piirretään nyt kaksi osaa hirsipuuta. Seuraavaksi alustetaan `maxVaaria`-niminen muuttuja. Muuttujaan talletetaan kuinka monta kertaa käyttäjä saa arvata väärin. Tämä saadaan selville kutsumalla `hirsi`-olion `vaiheitaJaljella`-metodia.

Seuraavaksi alkaa `while`-silmukka, jossa toistetaan arvausten kyselyä ja tästä seuraavia toimenpiteitä. Tämäkin osa on lähes samanlainen kuin aikaisemmassa versiossa. Nyt vain `if`-lohkossa, johon mennään jos arvaus oli väärä, on seuraava koodinpätkä

```
if ( hirsi.piirraSeuraavaOsa() ) {
   System.out.println("Hävisit!");
   break;
}
```

Väärän arvauksen tapauksessa kutsutaan `hirsi`-olion `piirraSeuraavaOsa`-metodia, joka piirtää seuraavan hirsipuun osan. Metodi palauttaa `true`, jos hirsipuu tuli valmiiksi. Tällöin peli päättyy ja poistutaan silmukasta `break`-lauseella.

Hienoa tässä ratkaisussa on, että varsinaisessa `pelaaPeli`-aliohjelmassa meidän ei tarvitse huolehtia miten ja mikä osa hirsipuusta piirretään, vaan `hirsi`-olio osaa osaa tämän. Meidän tarvitsee vain sanoa metodille, että piirrä seuraava osa. Olio-ohjelmoinnin idea onkin, että ohjelma koostuu pienistä palasista, olioista, jotka kommunikoivat keskenään. Näin saadaan tehtyä laajoja monimutkaisia ohjelmia luomalla monia pieniä yksinkertaisia osia. Tästä tekniikasta käytetäänkin myös nimitystä **divide and conquer** - hajota ja hallitse.

Lisätietoa:

- TIEP111 Ohjelmointi 2
- TIEA212 Graafisten käyttöliittymien ohjelmointi
- ITKA111 Oliosuuntautunut analyysi ja suunnittelu

Liite: Sanasto

Internetistä löytyy ohjelmoinnista paremmin tietoa englanniksi. Tässä tiedonhakua auttava sanasto ohjelmoinnin perustermeistä.

aliohjelma	**subprogram, subroutine, procedure**	komentorivi	**Command Prompt**	rajapinta	**interface**
alirajapinta	**subinterface**	konstruktori	**constructor**	roskienkeruu	**garbage collection**
alivuoto	**underflow**	koodauskäytänteet	**code conventions**	roskienkerääjä	**garbage collector**
alkeistietotyyppi	**primitive types**	kääntäjä	**compiler**	sijoituslause	**assignment statement**
alkio	**element**	kääriä	**wrap**	sijoitusoperaattori	**assignment operator**
alustaa	**intialize**	lause	**statement**	silmukka	**loop**
aritmeettinen operaatio	**arithmetic operation**	lippu	**flag**	sovelluskehitin	**Integrated Development Environment**
aritmeettinen lauseke	**arithmetic expression**	lohko	**block**	staattinen	**static**
bugi	**bug**	luokka	**class**	standardi syöttövirta	**standard input stream**
destruktori	**destructor**	metodi	**method**	standardi tulostusvirta	**standard output stream**
dokumentaatio	**documentation**	muuttaja	**variable**	standardi virhetulostusvirta	**standard error output stream**
funktio	**function**	määritellä	**declare**	syntaksi	**syntax**
globaali vakio	**global constant**	olio	**object**	taulukko	**array**
globaali muuttuja	**global variable**	ottaa kiinni	**catch**	testaus	**testing**
indeksi	**index**	paketti	**package**	toteuttaa	**implement**
Java alusta	**Java Platform**	parametri	**parameter**	tuoda	**import**
Java-virtuaalikonetta	**Java Virtual Machine**	periytyminen	**inheritance**	vakio	**constant**
Javan kääntäjä	**Java compiler**	poikkeus	**exception**	yksikkötestaus-rajapinta	**unit testing framework**
julkinen	**public**	poikkeustenhallinta	**exception handling**	ylivuoto	**overflow**
keskeytyskohta	**breakpoint**				

Liite: Yleisimmät virheilmoitukset ja niiden syyt

Aloittavan Java-ohjelmoijan voi joskus olla vaikeaa saada selvää kääntäjän virheilmoituksista. Kootaan tänne muutamia.

27.1 ArrayIndexOutOfBoundsException

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
      at VirheTesteja.main(VirheTesteja.java:9)
```

Kyseessä on taulukon yli-indeksointi VirheTesteja.java-tiedoston 9-rivillä. Yllä olevassa tilanteessa koitetaan viitata taulukon indeksiin 4, jota ei ole olemassa koko taulukossa. Tällainen virheilmoitus tulee aluksi usein kun käsitellään taulukoita silmukalla ja silmukan ehto on väärin.

27.2 Unresolved compilation problem

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
      Type mismatch: cannot convert from double to int
  at VirheTesteja.main(VirheTesteja.java:14)
```

Tässä yritetään rivillä asettaa `int`-tyyppiseen muuttujaan väkisin `double`-tyyppiä. Eclipse osaa monesti valittaa tämän tyylisistä virheistä etukäteen.

27.3 NullPointerException

```
Exception in thread "main" java.lang.NullPointerException
      at ei_monisteessa.VirheTesti.main(VirheTesti.java:18)
```

Yllä olevassa virheilmoituksessa rivillä 18 yritetään tehdä jotain oliolle, mutta ongelmana on ettei viitemuuttuja osoita mihinkään olioon. Viitemuuttujan arvo on siis `null`. Yritetään esimerkiksi kutsua jonkun olion metodia, mutta jos viitemuuttuja ei osoita mihinkään olioon, ei se tietenkään onnistu. Tämä virheilmoitus on saatu aikaan hieman väkinäisesti seuraavalla koodinpätkällä:

```
Integer luku = new Integer(12);
luku = null;
double lukuDoublena = luku.doubleValue();
```

27.4 NoSuchElementException

```
Exception in thread "main" java.util.NoSuchElementException
      at java.util.StringTokenizer.nextToken(Unknown Source)
      at esimerkit.Pilkkominen.main(Pilkkominen.java:69)
```

Tällainen virheilmoitus tulee, jos pyytää `StringTokenizer`-oliolta seuraavaa palasta, vaikka sellaista ei olisi olemassa.

Lähdeluettelo

VES: Vesterholm, Mika; Kyppö, Jorma, Java-ohjelmointi, 2003
DEI: Deitel, H.M; Deitel, P.J, Java How to Program, 2003
KOS: Kosonen, Pekka; Peltomäki, Juha; Silander, Simo, Java 2 Ohjelmoinnin peruskirja, 2005
DOC: Sun, , , http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
KOSK: Jussi Koskinen, Ohjelmistotuotanto-kurssin luentokalvot(Osa: Ohjelmistojen ylläpito),
LAP: Vesa Lappalainen, Ohjelmointi 2, , http://users.jyu.fi/~vesal/kurssit/ohj2/moniste/html/m-Title.htm
MÄN: Männikkö, Timo, Johdatus ohjelmointiin- moniste, 2002
LIA: Y. Daniel Liang, Introduction to Java programming, 2003

Jyväskylän yliopisto
Tietotekniikan laitos

Luentomoniste 15

University of Jyväskylä
Department of Mathematical Information Technology

Lecture Notes 15

1. MÄKINEN, RAINO A. E., Numeeriset menetelmät. 1999 (107 s.)
2. LAPPALAINEN, VESA ja RISTO LAHDELMA, Olio-ohjelmoiti 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, Ohjelmointi 2. 2004 (214 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. 2010. (134 s.)

ISBN 978-951-39-4022-5
ISSN 1456-9787